# Reading Data from Memory

May 1 and 8, 2023

Sungjoo Yoo

Computing Memory Architecture Lab.

CSE, SNU

http://cmalab.snu.ac.kr

# Weekly Lecture / Lab Schedule

- W1 (March 6) Class introduction / (March 8) Orientation & team formation
- W2 13 Verilog 1 Combinational circuits / 15 Verilog 1 (tool installation, adder & multiplier combinational logic)
- W3 20 Verilog 2 Sequential circuits / 22 Verilog 2 (memory i/o, FSM sequential logic)
- W4 27 AI application introduction 1, Amaranth introduction 1 / 29 Amaranth (tool installation, MAC, adder tree)
- W5 4/3 AI application introduction 2, Amaranth introduction 2 (memory i/o, FSM sequential logic) / 5 Amaranth (PE)
- W6 10 AI application introduction 3, Neural network accelerator 1 / 12 Amaranth (stacked PEs)
- W7 17 Neural network accelerator 2 / 19 Amaranth (stacked PEs)
- W8 24 **Mid-term exam** / 26 Amaranth (stacked PEs)
- W9 5/1 Reading data from memory 1 (VA2PA, interconnect) / 3 Convolution lowering
- W10 8 Reading data from memory 2 (DRAM main memory), Compressing networks 1 (pruning) / 10 Tiling
- W11 15 Compressing networks 2 (pruning, low precision) / 17 PyTorch model – Amaranth simulator communication
- W12 22 Zero-skipping & low-precision hardware accelerator / 24 Quantization, project introduction
- W13 29 Invited talks (commercial solutions: Google Edge TPU, Furiosa AI) / 31 Homework Q&A
- W14 6/5 **Final exam** / 7 Project Q&A
- W15 12 Claim & Project Q&A / 14 Project submission

[Stevens, 2011]

# Reading Data from Memory (in Cache Miss)

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

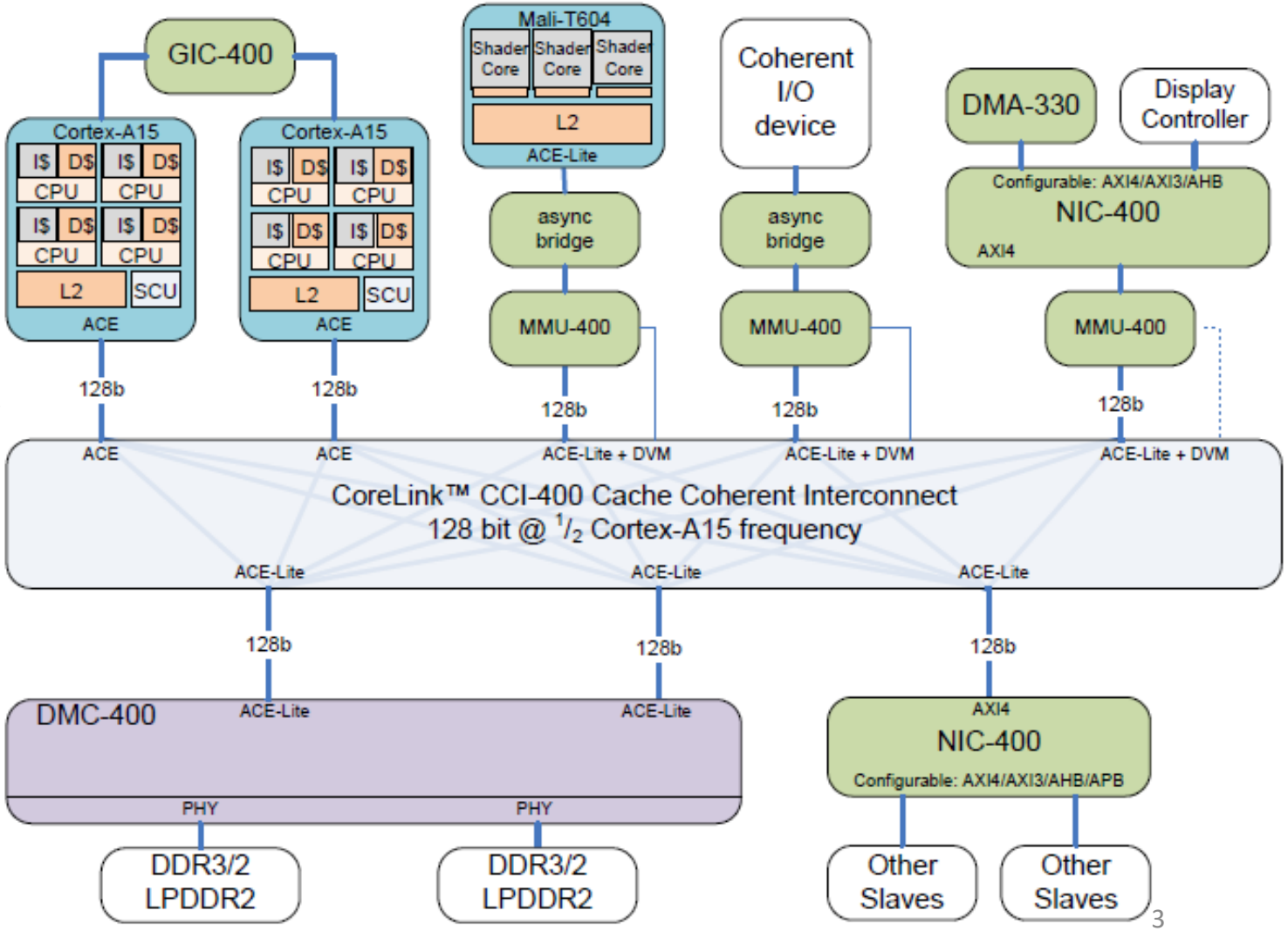
VA(x) → PA(x) translation on TLB

CPU sends to memory, via the interconnect, a read request of PA(x)

The read request arrives at memory controller

Memory controller reads data@PA(x) from DRAM

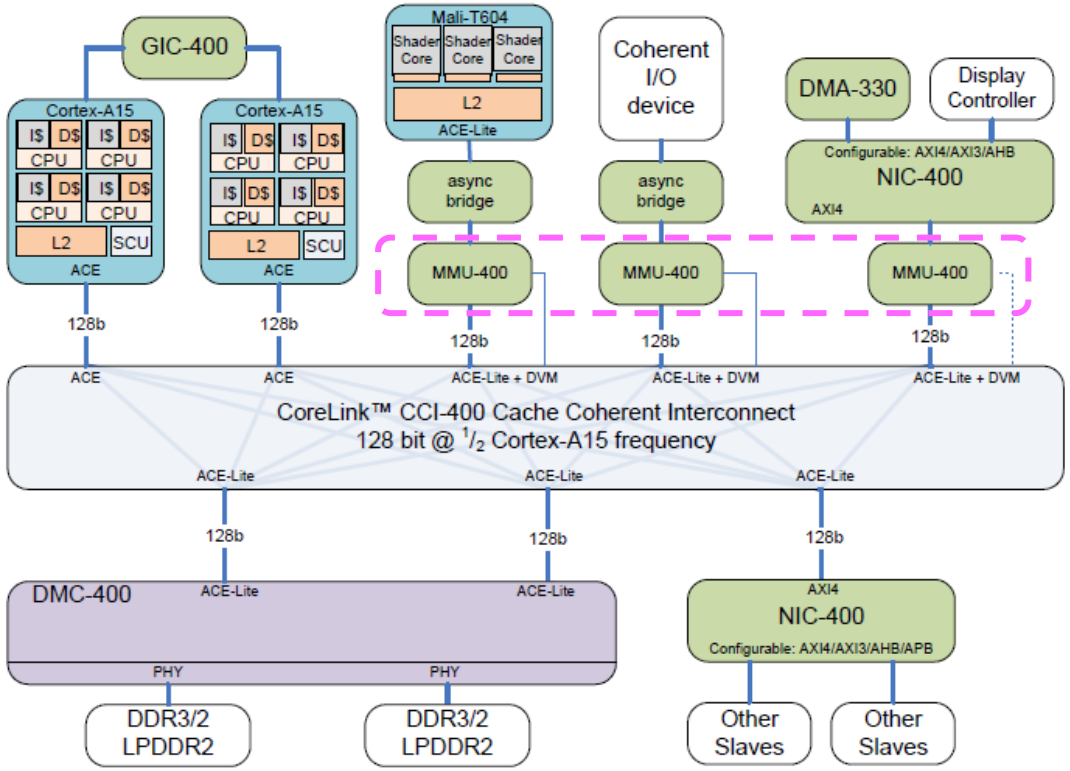
Memory controller sends the data to CPU via the interconnect



[Stevens, 2011]

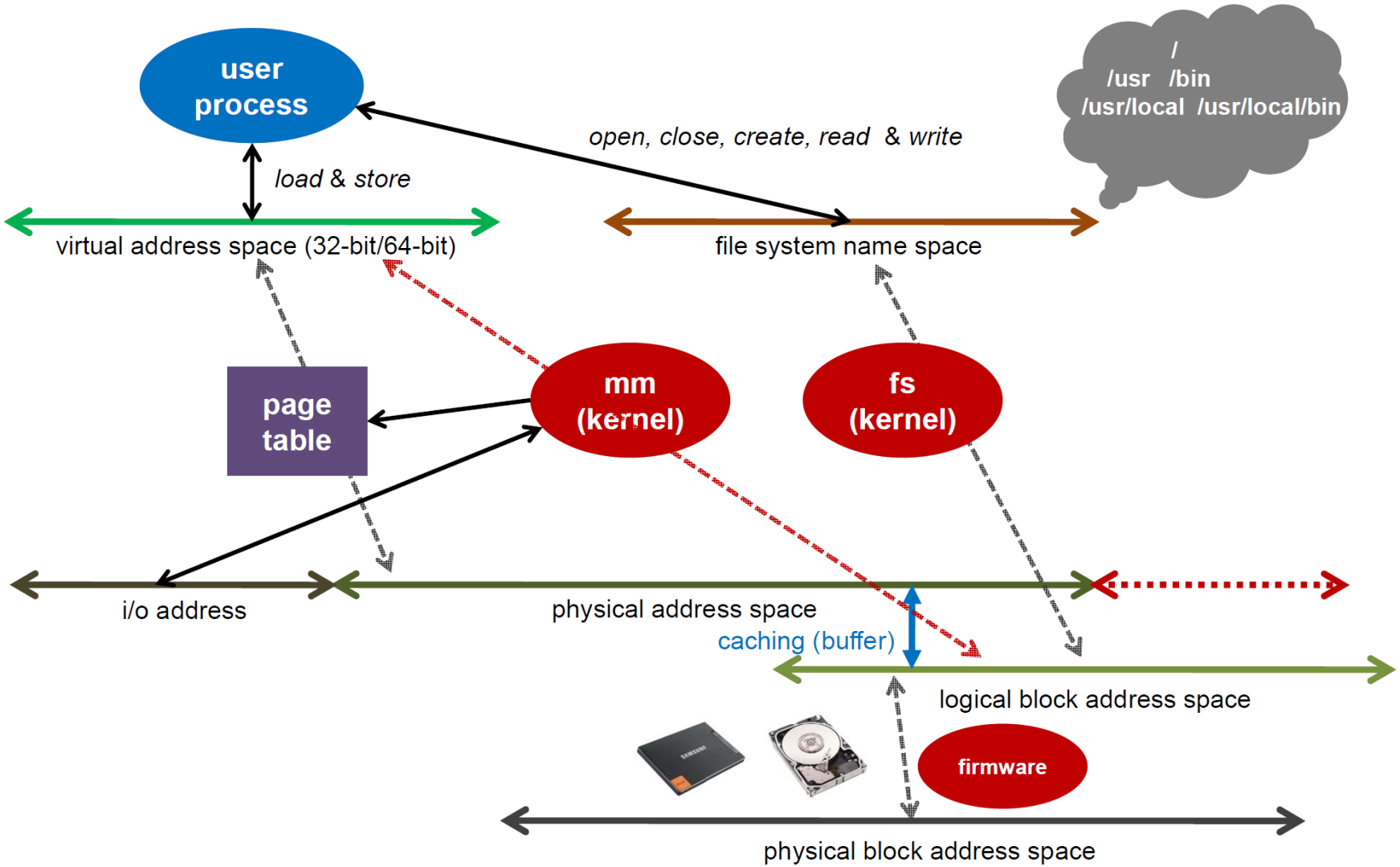
# IOMMU for Hardware Devices on Application Processor

- Each hardware function, i.e., device has its own MMU (memory management unit) for virtual to physical address translation



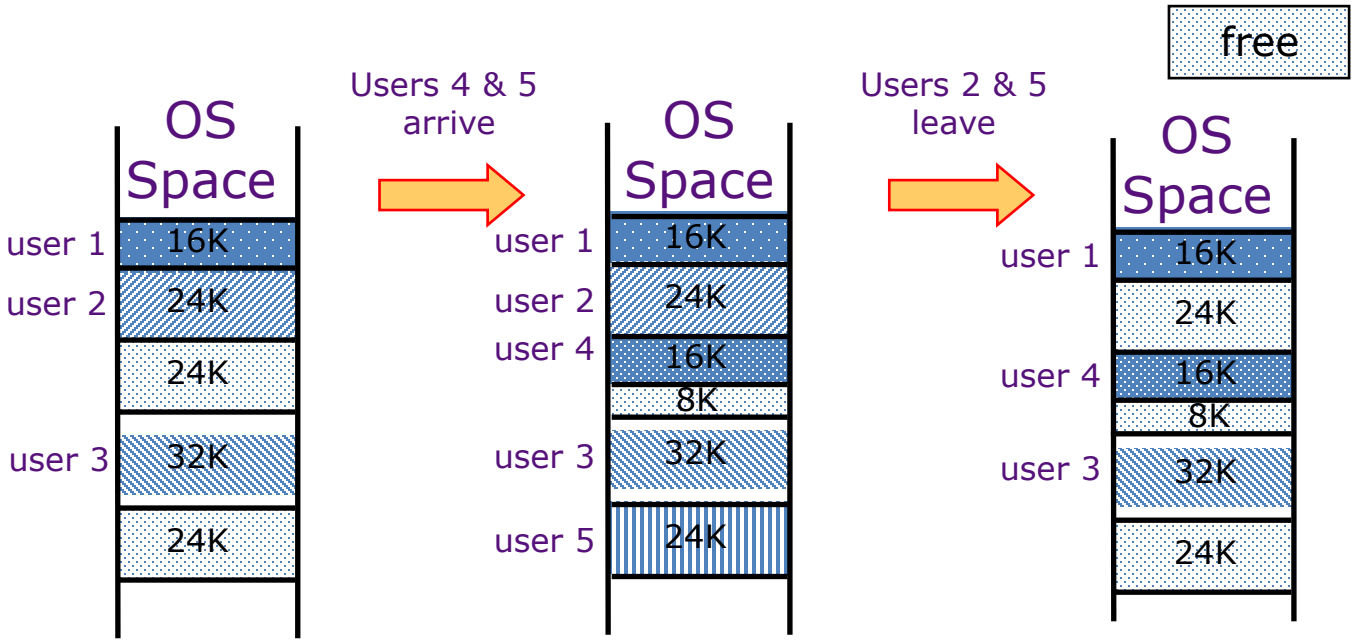
[Cho, 2015]

# Memory/Storage Address Space



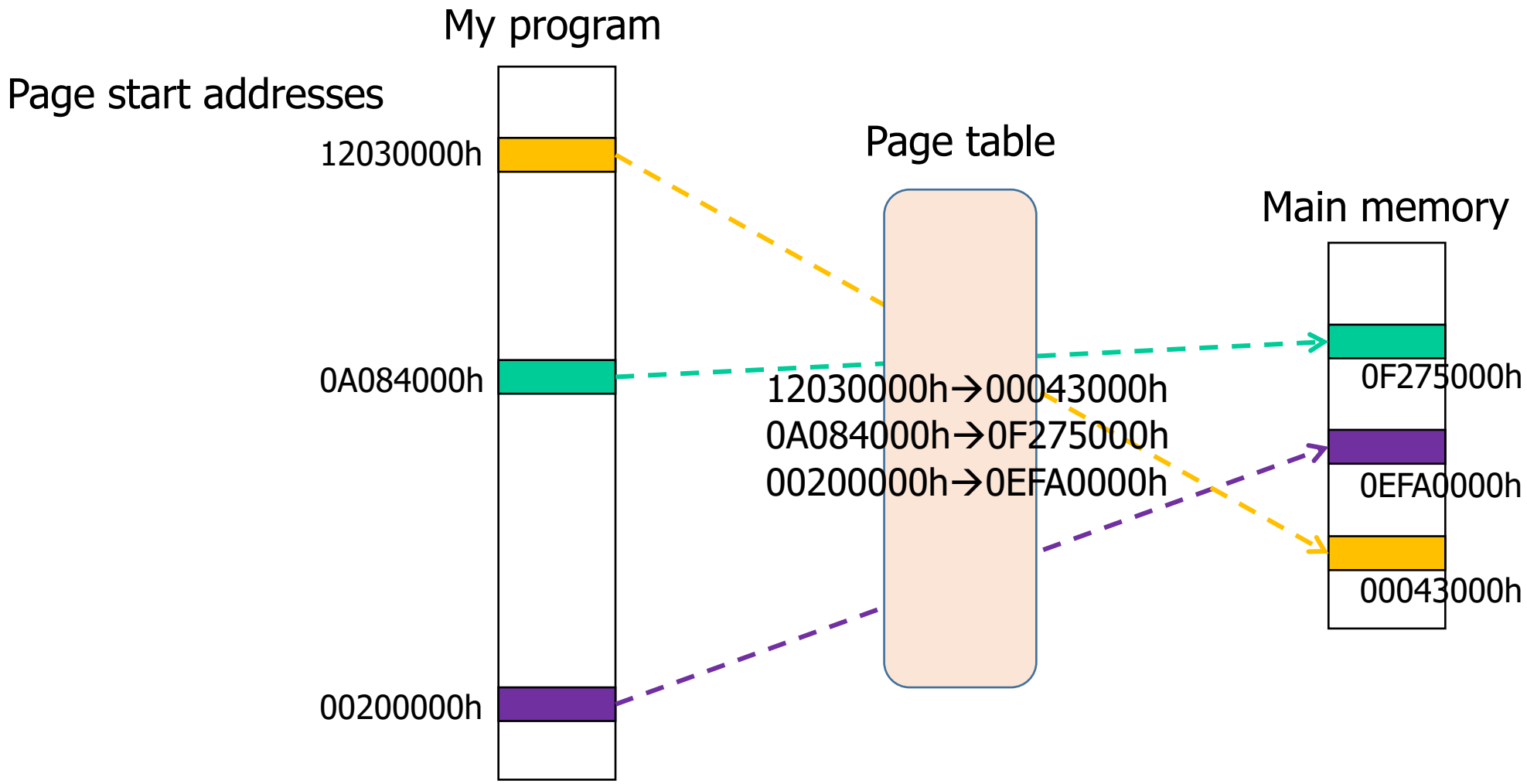
[K. Asanovic, 2008]

# Memory Fragmentation

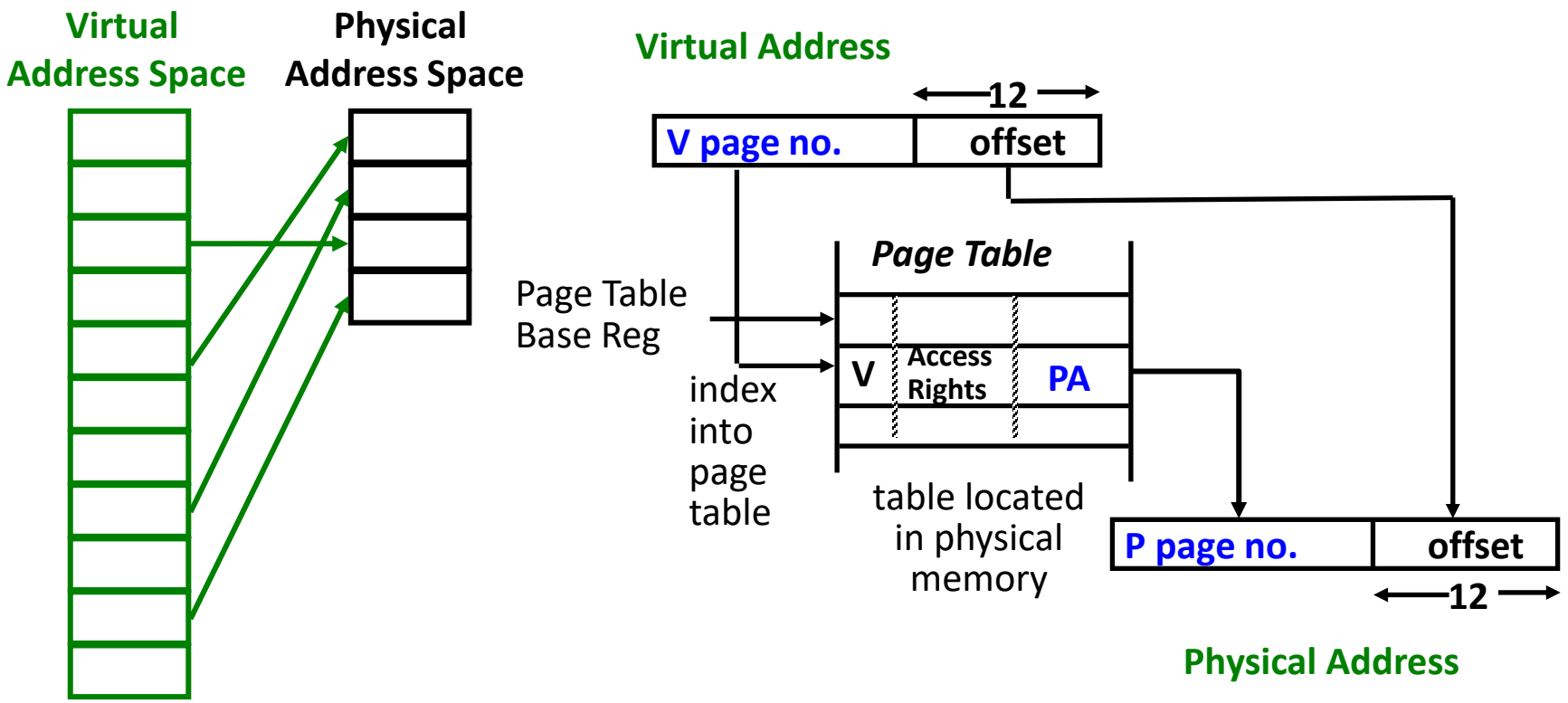



As users come and go, the memory/storage is "fragmented"
→ Large contiguous memory resource is hard to find

# Page Table to Keep Virtual to Physical Address Mapping Information



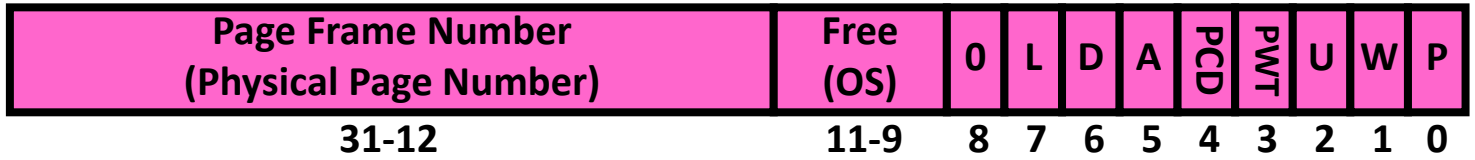
# Virtual Memory: An Introduction




- **Page table maps virtual page numbers to physical frames**
  - "PTE" = Page Table Entry (arrow from virtual to physical address space)

# Virtual Memory: An Introduction

- **What is in a Page Table Entry (or PTE)?**
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
  - Example: Intel x86 architecture PTE:
    - Address same format previous slide (10, 10, 12-bit offset)
    - Intermediate page tables called “Directories”



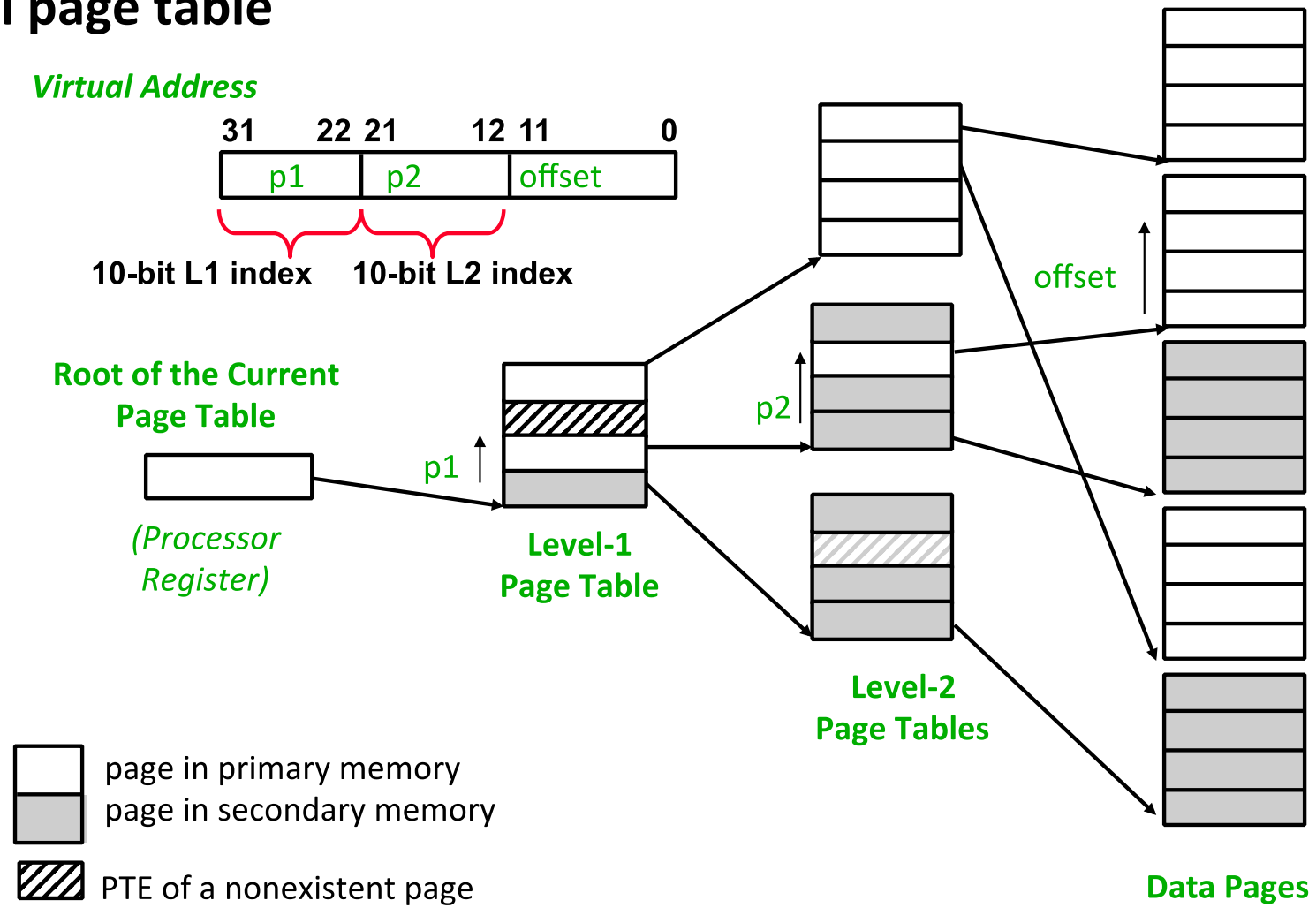

| | |
|---|---|
| P: | Present (same as “valid” bit in other architectures) |
| W: | Writeable |
| U: | User accessible |
| PWT: | Page write transparent: external cache write-through |
| PCD: | Page cache disabled (page cannot be cached) |
| A: | Accessed: page has been accessed recently |
| D: | Dirty (PTE only): page has been modified recently |
| L: | L=1 ⇒ 4MB page (directory only). Bottom 22 bits of virtual address serve as offset |

# Virtual Memory: An Introduction

- **Size of linear page table**
  - With 32-bit addresses, 4-KB pages, and 4-byte PTEs:
    - $2^{20}$ PTEs, i.e, 4 MB page table per user
    - 4 GB of swap needed to back up full virtual address space
- **Larger pages?**
  - more internal fragmentation (don't use all memory in page)
  - larger page fault penalty (more time to read from disk)
- **What about 64-bit virtual address space???**
  - Even 1MB pages would require $2^{44}$ 8-byte PTEs (35 TB!)
- **What is the "*saving grace*"?**
  - Only a small fraction of the pages are populated

# Virtual Memory: An Introduction

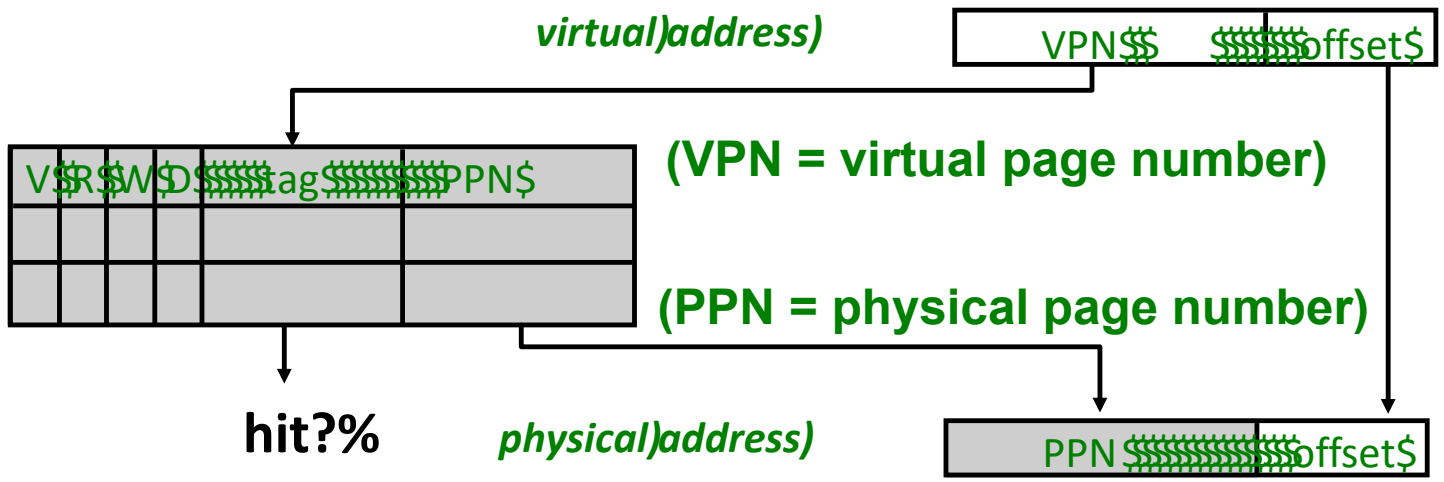
- **Hierarchical page table**



# Translation Look-aside Buffers (TLB)

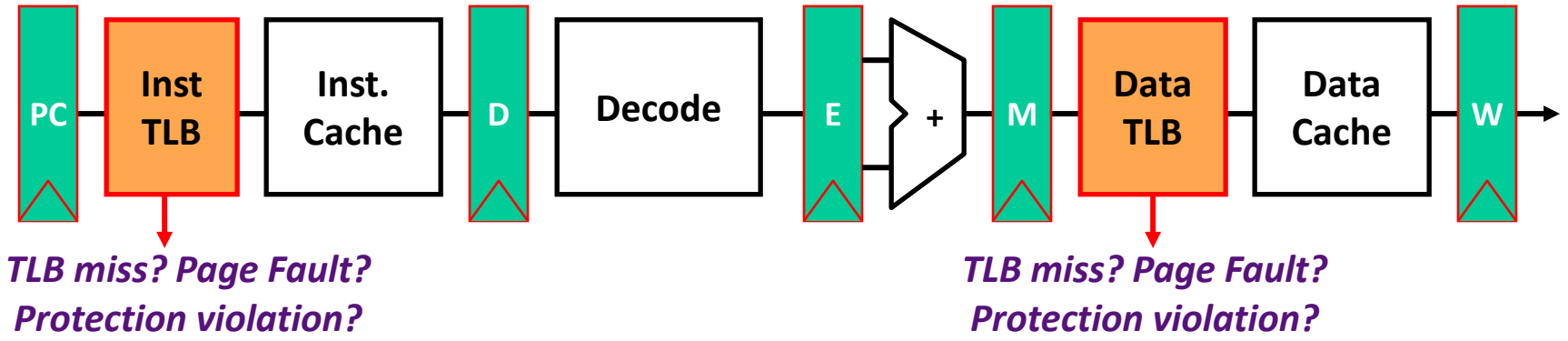
- **Address translation is very expensive in terms of latency!**
  - In a 2-level page table, each memory access requires 3 memory accesses

    **Level 1 page table access + Level 2 page table access + Access to the physical address**
- **Solution: Cache translations in TLB by exploiting locality in memory access behavior**
  - TLB hit ⇒ Single Cycle Translation
  - TLB miss ⇒ Page Table Walk to refill



# Cache and TLB Interactions

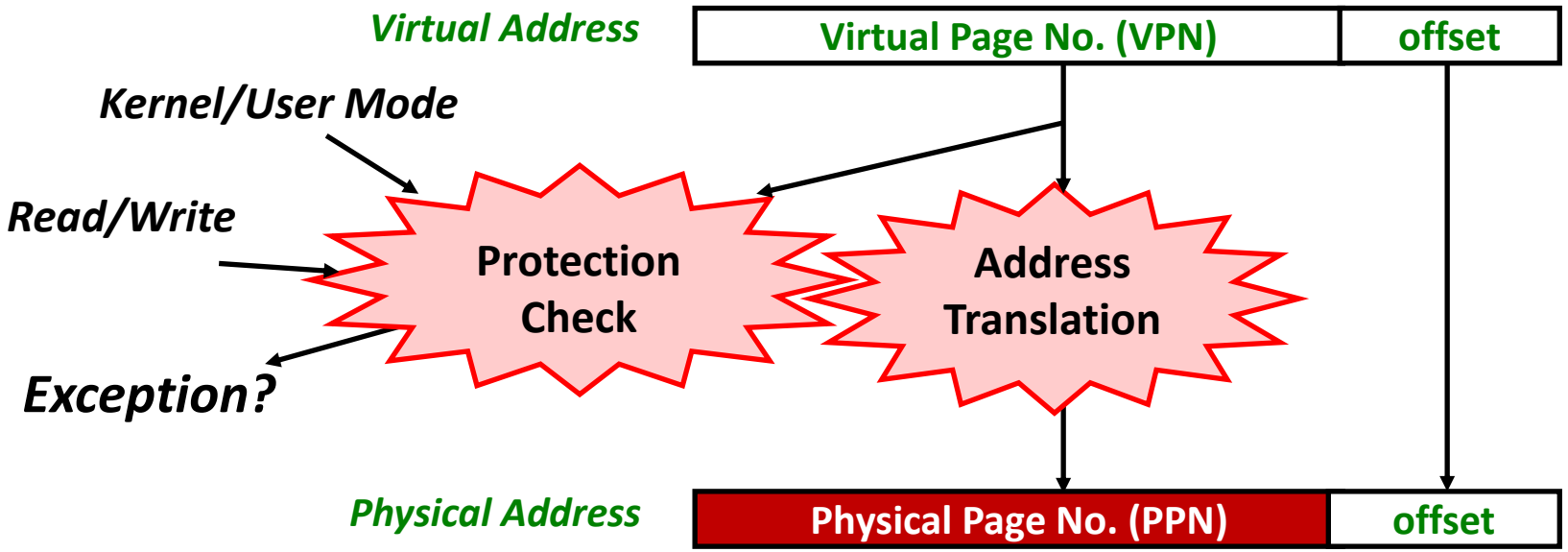
- **Address translation in CPU pipeline**




- SW handlers need a restartable exception on page fault or protection violation
- Handling a TLB miss needs a hardware or software mechanism to refill TLB
- Need mechanisms to cope with the additional latency of a TLB:
  - Slow down the clock
  - Pipeline the TLB and cache access
  - Virtual address caches
  - Parallel TLB/cache access

# Virtual Memory: An Introduction

- **Address translation and protection**




  - Every instruction and data access needs address translation and protection checks
  - *A good VM design needs to be fast (~ one cycle) and space efficient*

# Translation Look-aside Buffers (TLB)

- **TLB Designs**
  - Typically 32-128 entries
  - Usually fully associative
    - Each entry maps a large page, hence less spatial locality across pages -> more likely that two entries conflict
    - Sometimes larger TLBs are 4-8 way set-associative
  - Random or FIFO replacement policy
  - Typically only one page mapping per entry
  - **No process information in TLB ⇒ Flush TLB on context switch**

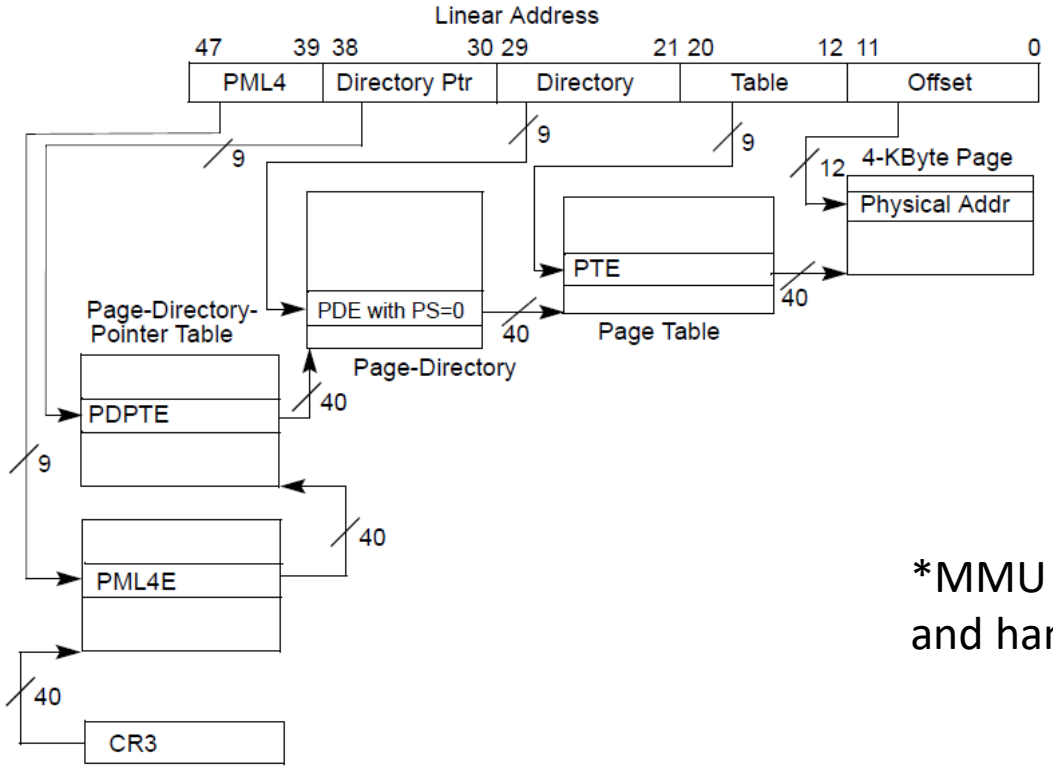
  - TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB
    - Example: 64 TLB entries, 4KB pages, one page per entry
    - TLB Reach = **64 entries * 4 KB = 256 KB** ?

  - In case of TLB miss, we need to access multi-level page tables

[Intel® 64 and IA-32 Architectures Software Developer's Manual]

# 48-bit Virtual Address Case: TLB and MMU Cache

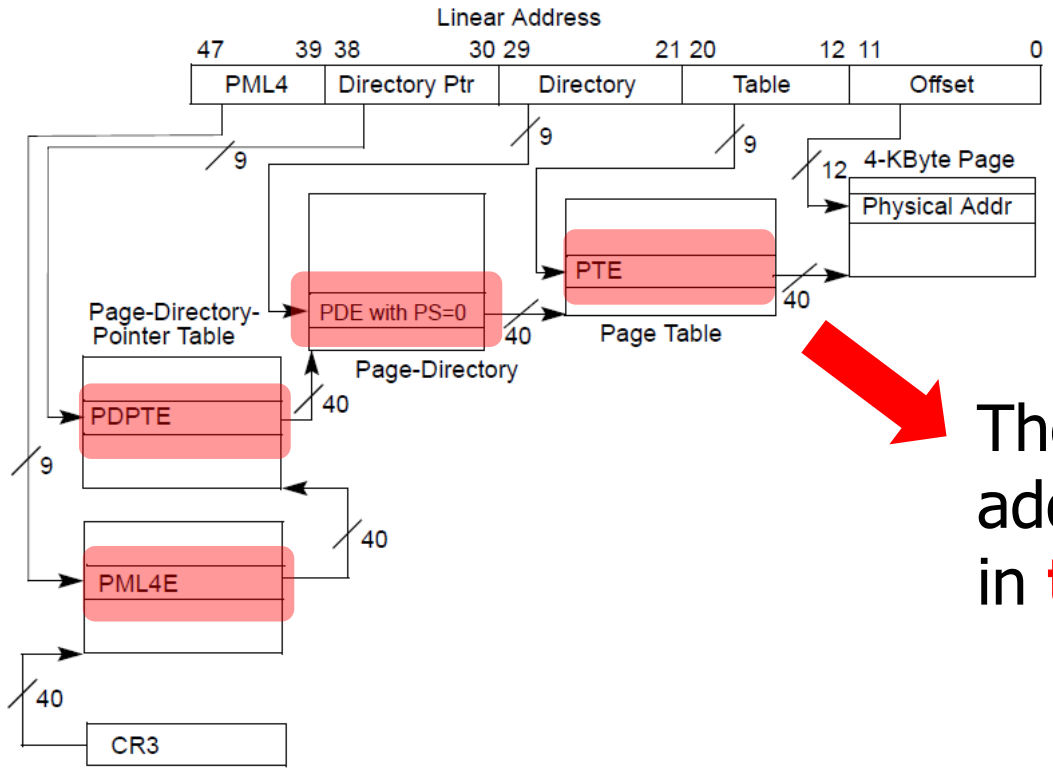
- All the page tables are stored in main memory
- In case of TLB miss, how many memory accesses for address translation?




*MMU (memory management unit) resides inside of CPU and handles TLB miss and protection violations

# TLB Miss Penalty

- Since all the 4 tables are stored in main memory, in the worst case, 4 memory accesses for address translation occur, e.g., ~50ns/memory access x 4=200ns! (400 instructions @ 2GHz)

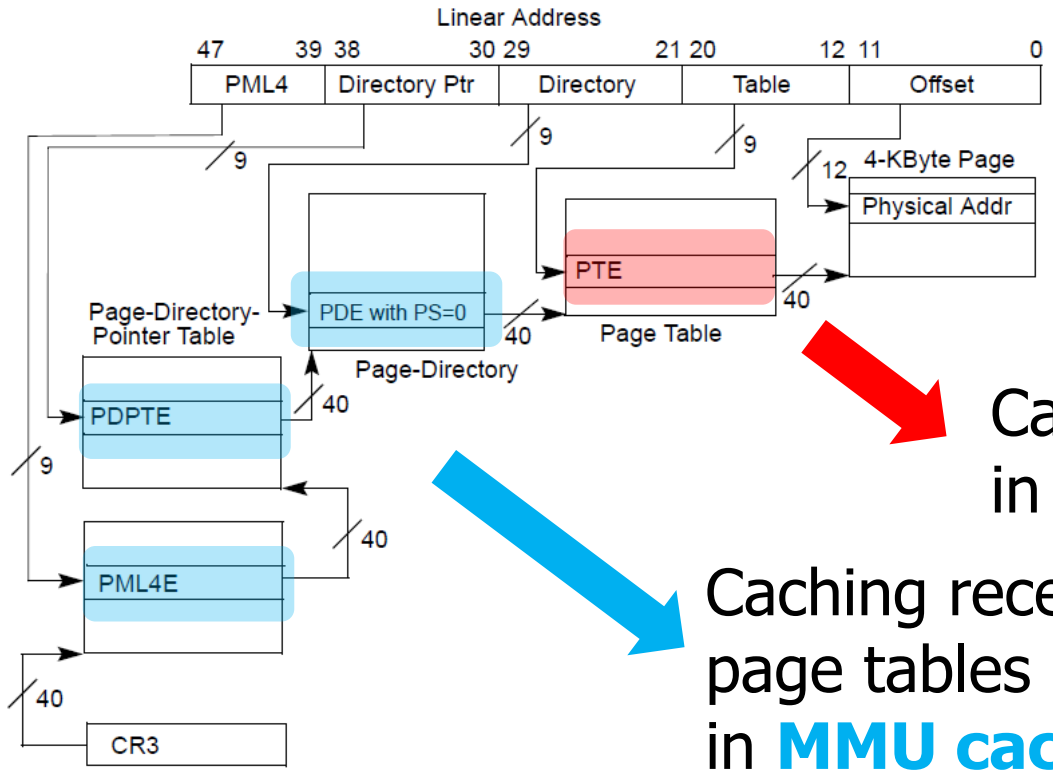



Then, the PTE is used for address translation and stored in **translation lookaside buffer (TLB)**

[Intel® 64 and IA-32 Architectures Software Developer's Manual]

# 48-bit Virtual Address Case: TLB and MMU Cache

- TLB and MMU cache store recently accessed entries to reduce the average latency of address translation

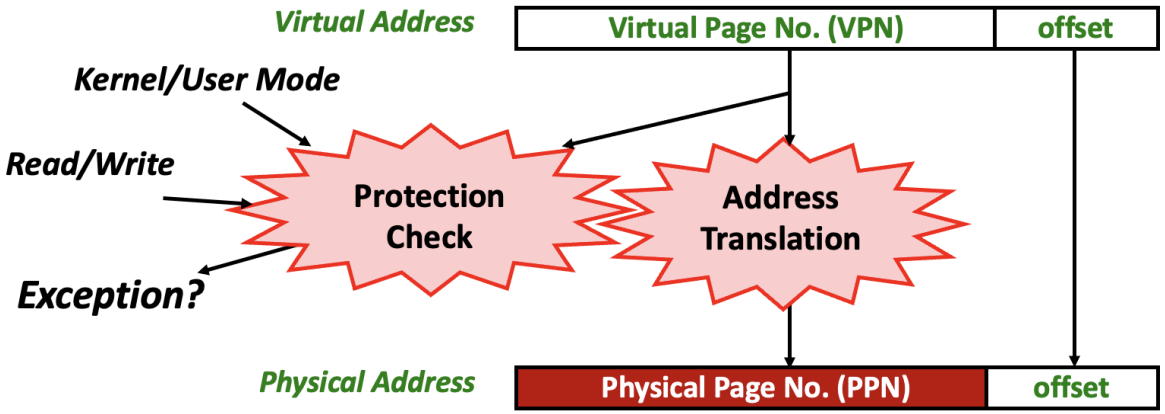


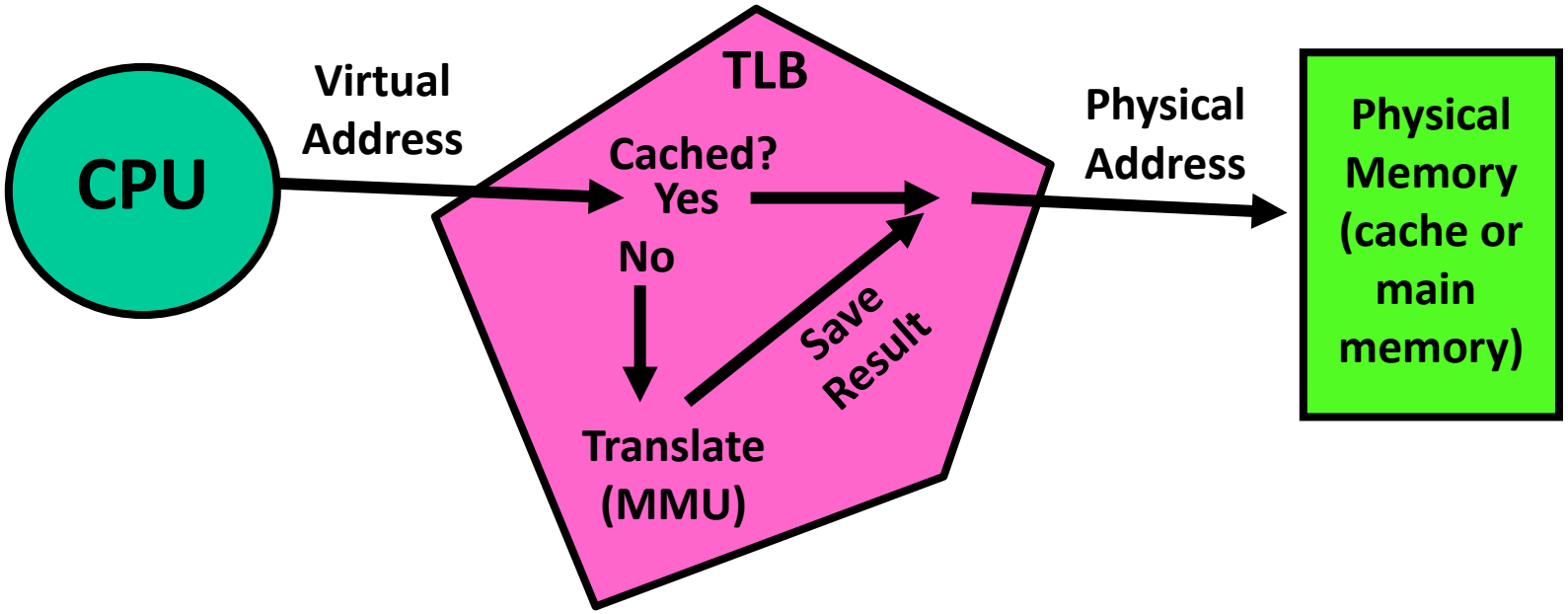

Caching recently accessed PTEs in **translation lookaside buffer (TLB)**

Caching recently accessed entries of intermediate page tables (a.k.a. page directories) in **MMU cache**

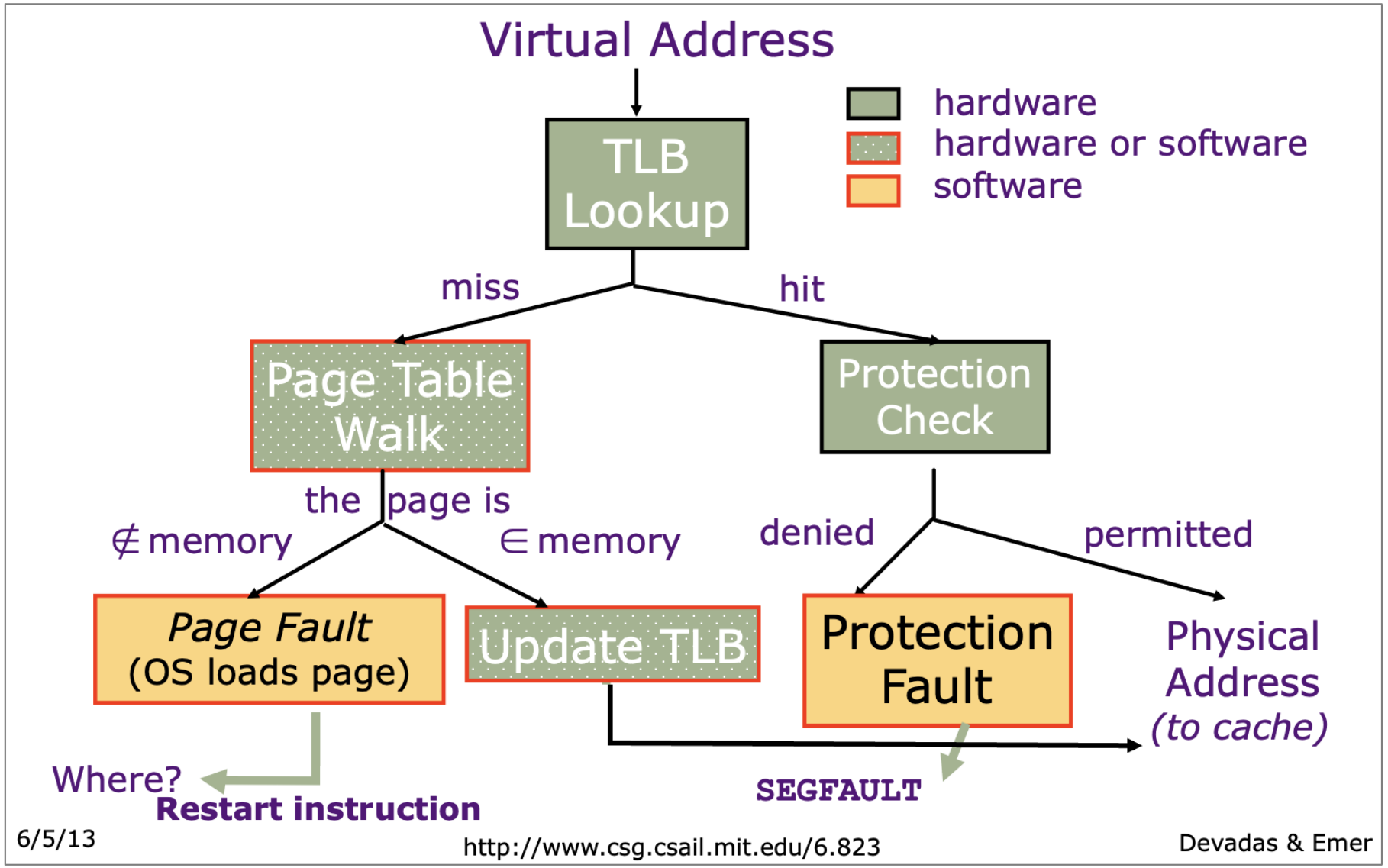
# Translation Look-aside Buffers (TLB)

- **Translation Look-aside Buffers (TLB)**
  - Cache on translations
  - Fully associative, set associative, or direct mapped



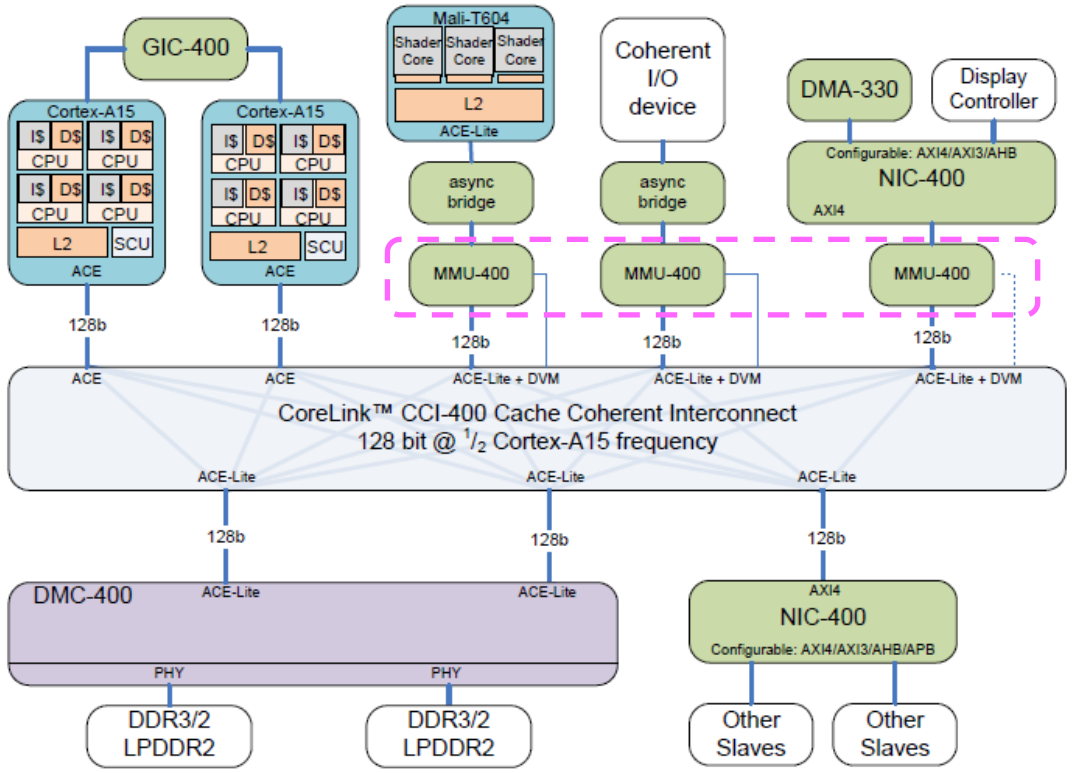
# Address Translation: Putting It All Together



[Stevens, 2011]

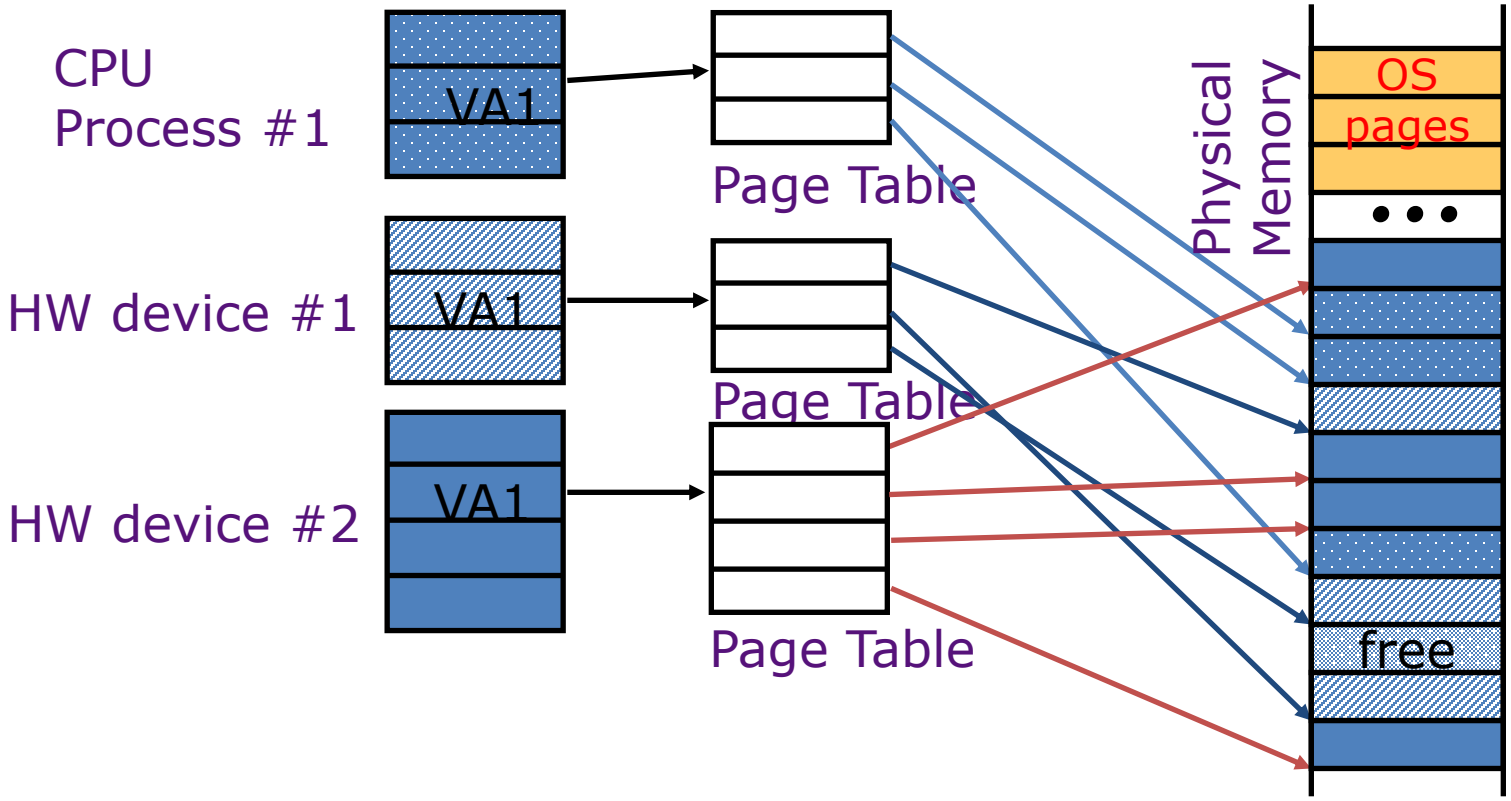
# IOMMU for Hardware Devices on Application Processor

- Each hardware function, i.e., device has its own MMU



[K. Asanovic, 2008]

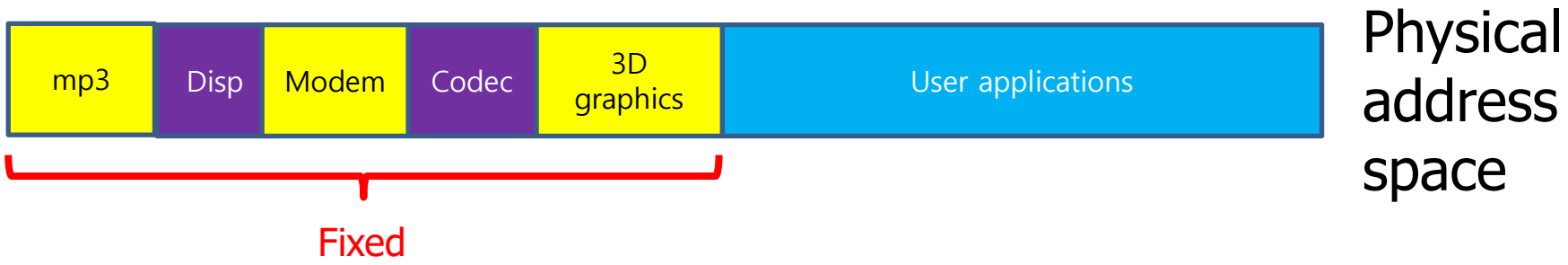
# Private Address Space per CPU and HW Devices




- Each of HW devices has its own virtual address space → Each needs its own page table

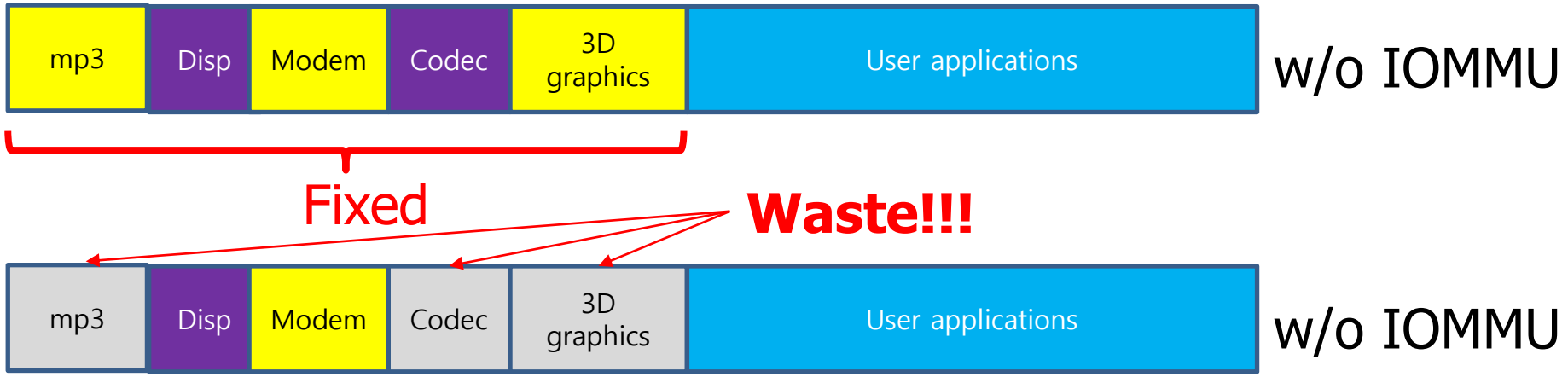
# What Happens without IOMMU?

- Each device needs a contiguous region of physical address (real design practice ~15 years ago)



# What Happens without IOMMU?

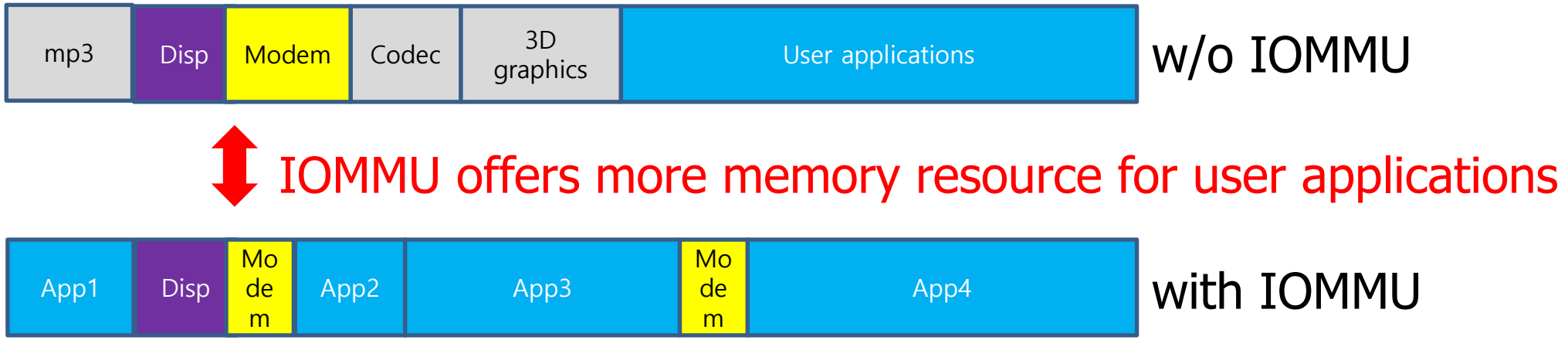
- Each device needs a contiguous region of physical address




- What if you use only text messaging for now?
  - **Waste of main memory resource** due to the constraint of contiguous address regions

# What Happens without IOMMU?

- Each device needs a contiguous region of physical address

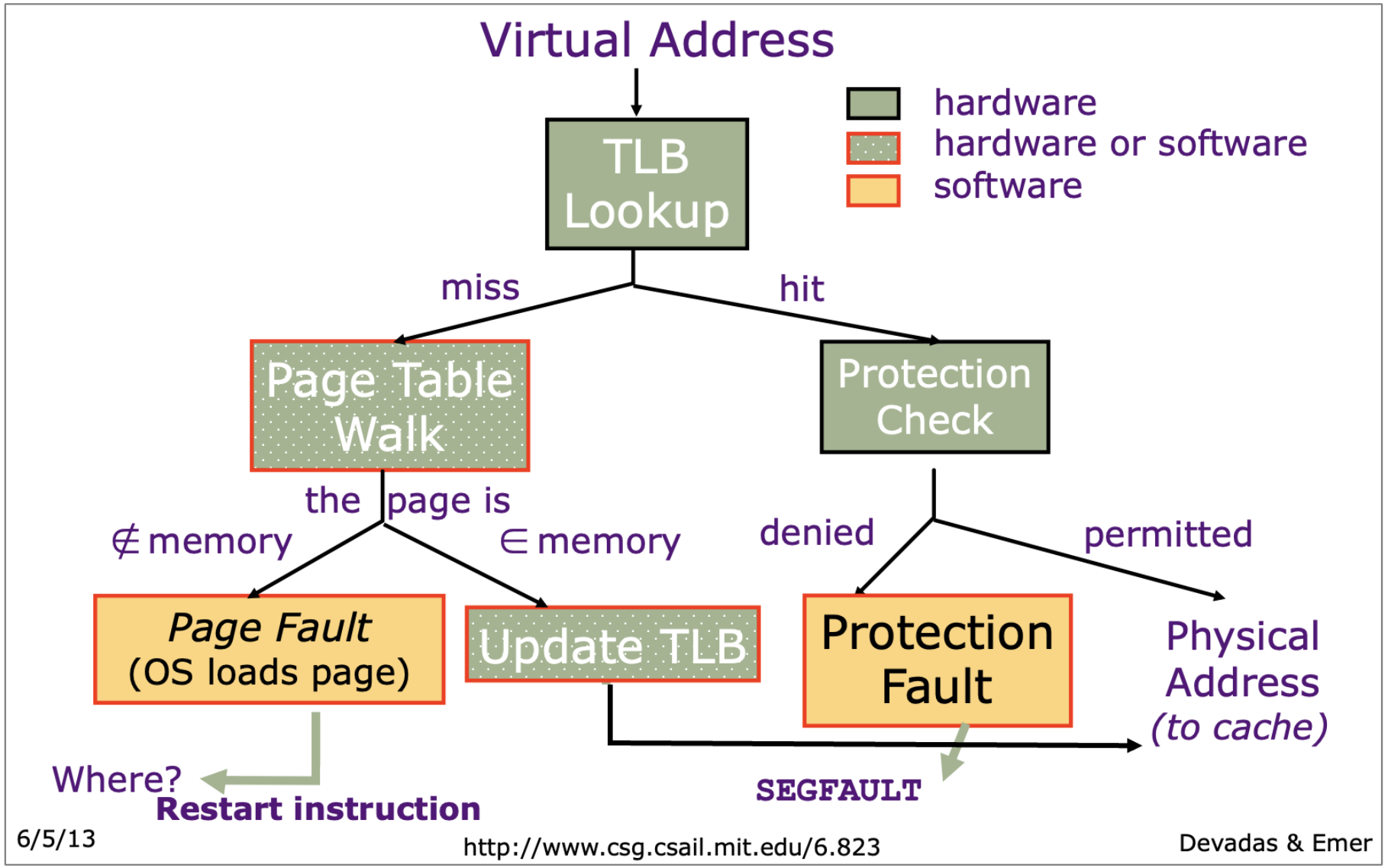



- IOMMU provides demand paging thereby enabling better utilization of memory resource
- Currently used on smartphones

# IOMMU shares the same benefits and problems (solutions) as CPU MMU

- Benefits
  - Better utilization of memory resource
  - No need of contiguous physical pages
  - Protection from DMA attacks
- Problems (and solutions)
  - Each HW device needs its own page table (PT) and TLB → sharing PT with CPU
  - TLB miss penalty → L2 TLB (shared by devices and CPU) as well as L1 TLB (local to device)

# Address Translation: Putting It All Together



[Stevens, 2011]

# Reading Data from Memory (in Cache Miss)

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

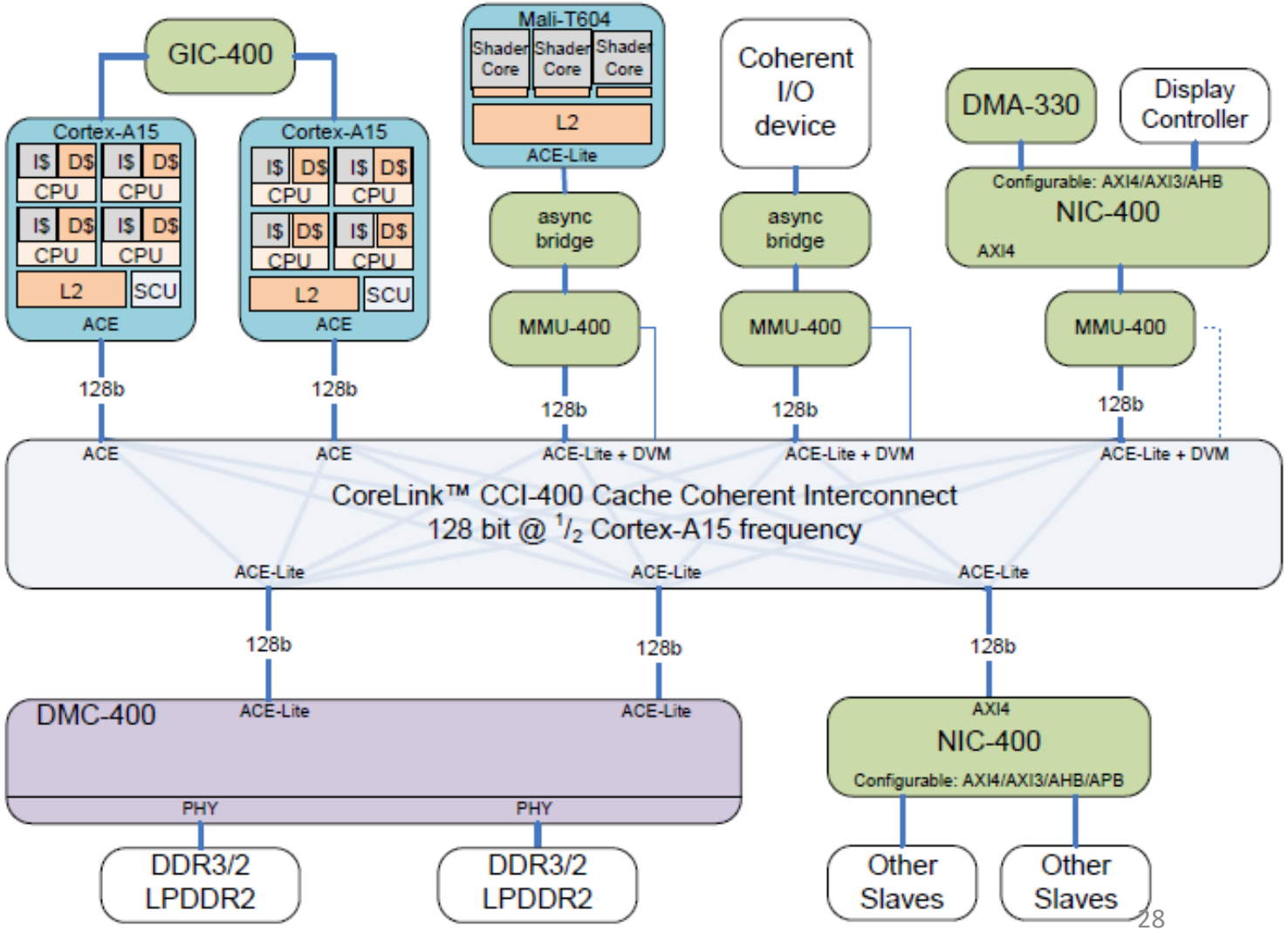
VA(x) → PA(x) translation on TLB

CPU sends to memory, via the interconnect, a read request of PA(x)

The read request arrives at memory controller

Memory controller reads data@PA(x) from DRAM

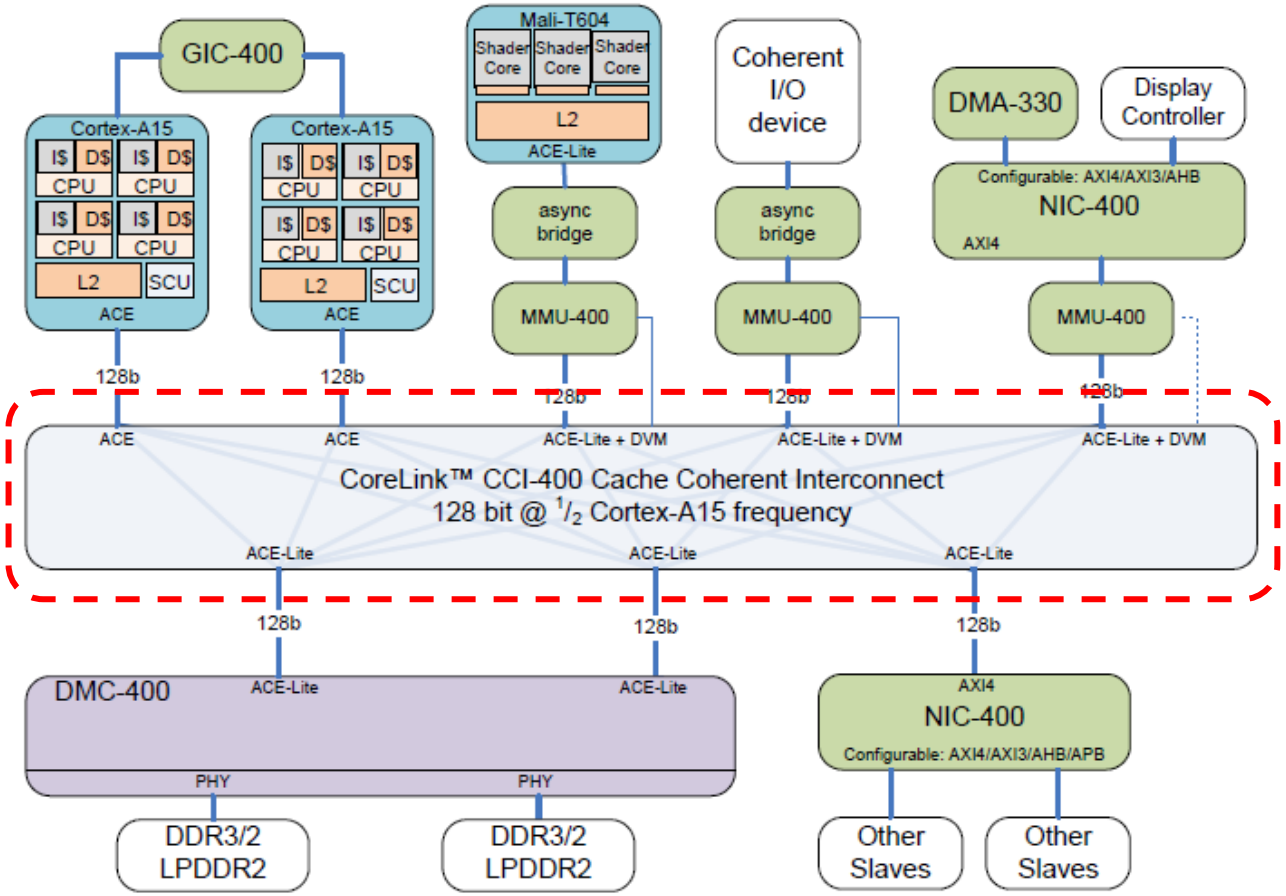
Memory controller sends the data to CPU via the interconnect



[Stevens, 2011]

# On-Chip Bus or Interconnect

- Hardware components are connected to this for communication



[Source: AXI Spec]

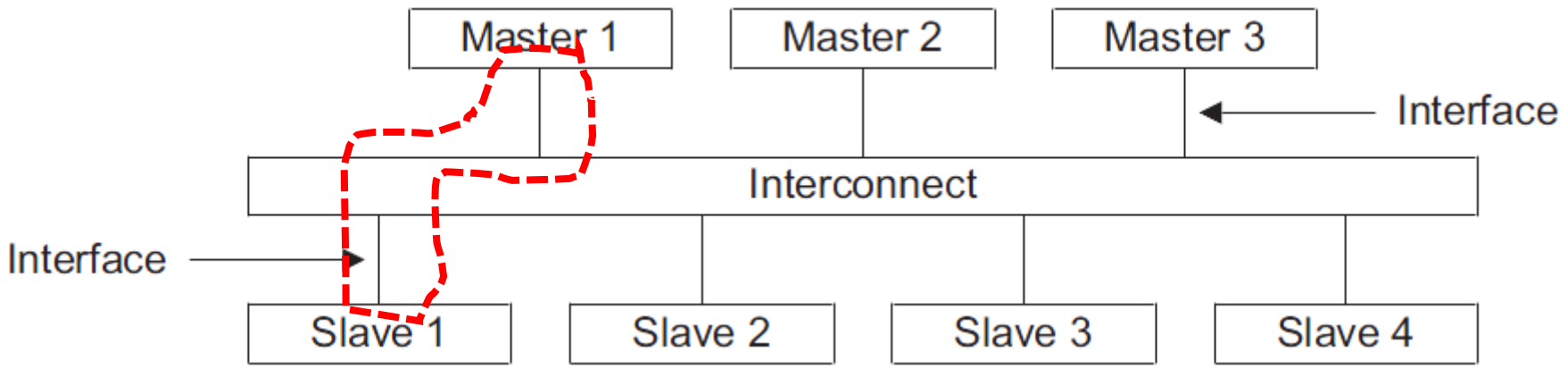
# AMBA3 Advanced eXtensable Interface (AXI) Protocol

- Multiple channels
- Narrow and wide transfer
- Single credit-based flow control: valid and ready signals
- Burst
- Split transaction to overlap request and data transfer
- Multiple outstanding requests
- Implementation issue: connectivity and arbitration

[Source: AXI Spec]

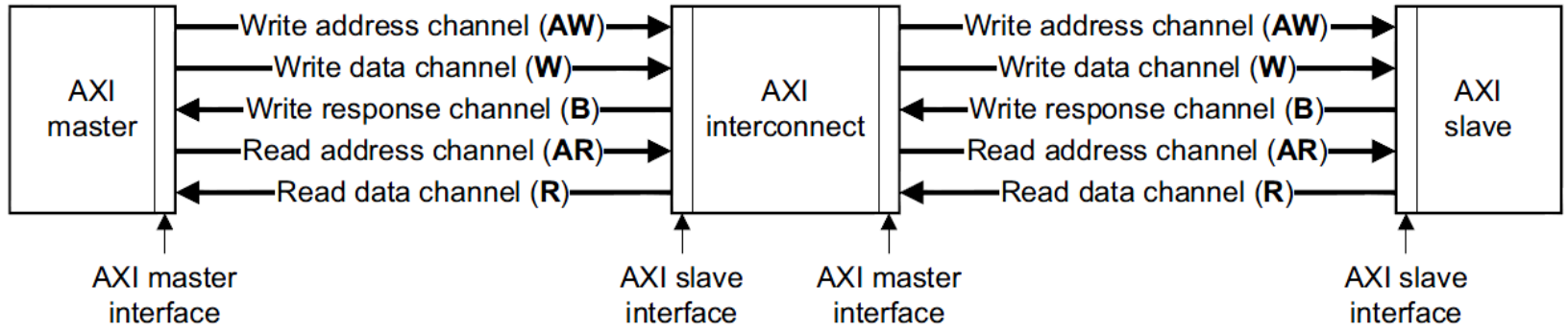
# Interconnect, Interface & Channel

CPU with L2 cache




Memory controller for DRAM

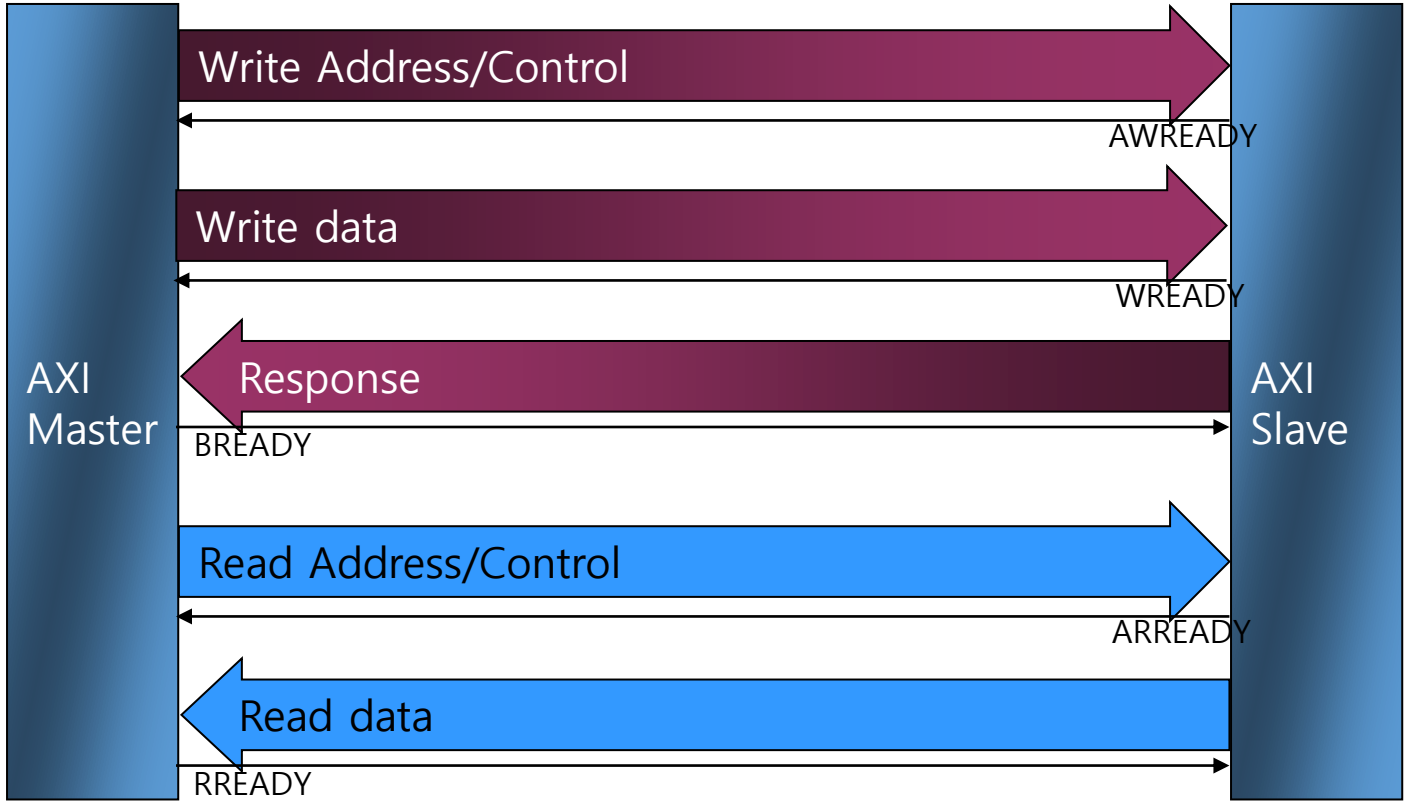
CPU with L2 cache



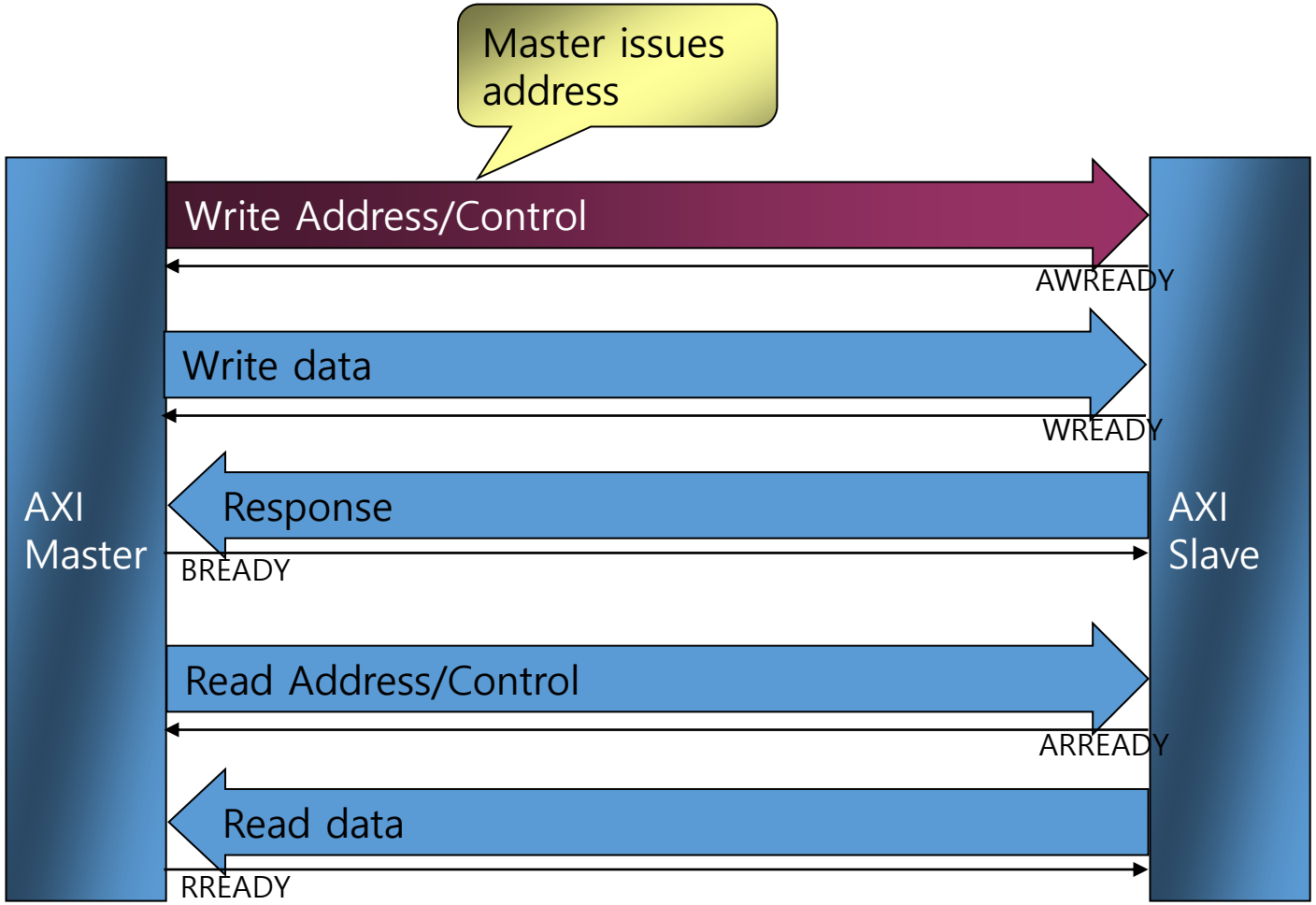

Memory controller for DRAM

# Split Transaction

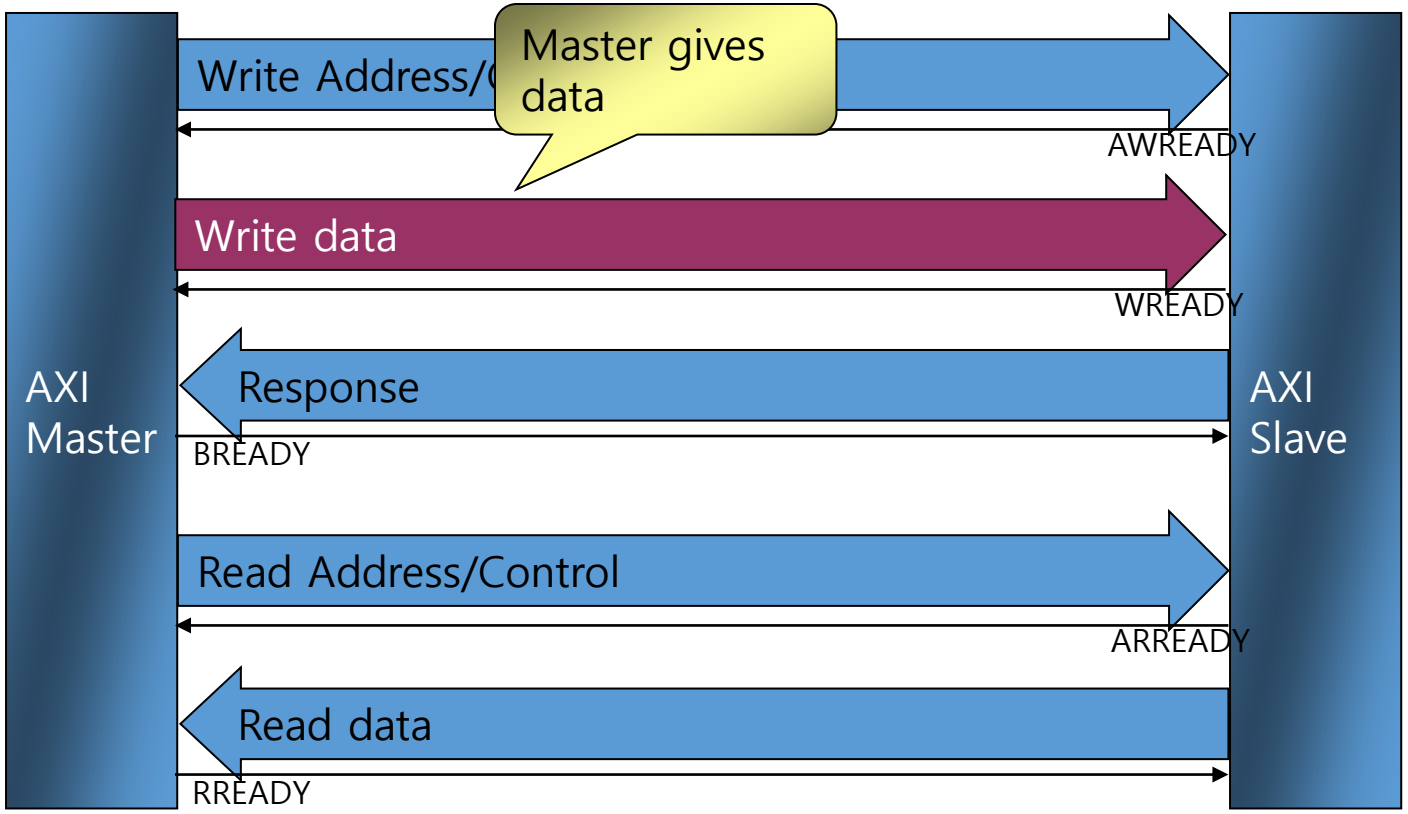
- Address, data, and response are handled separately.



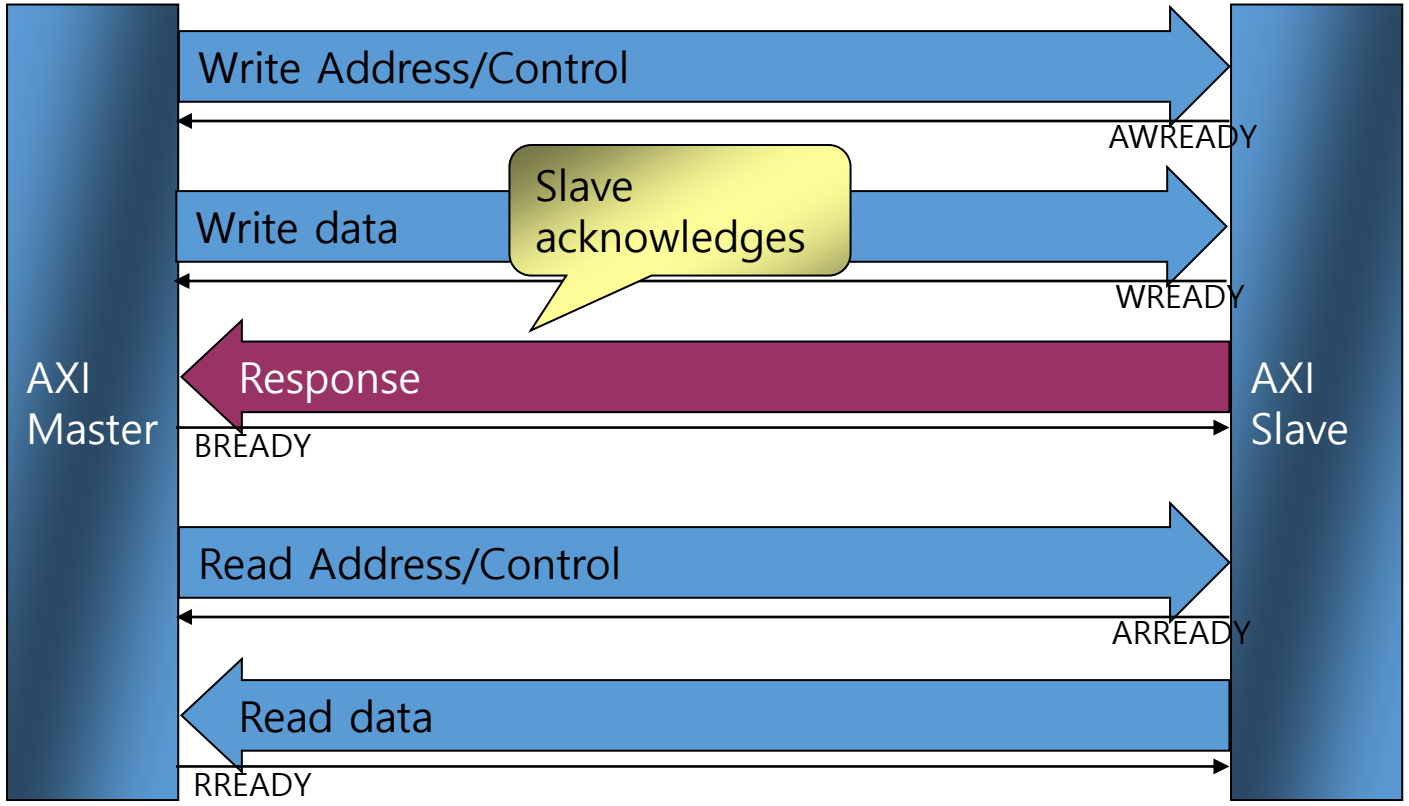
# Split Transaction: Write (1/3)



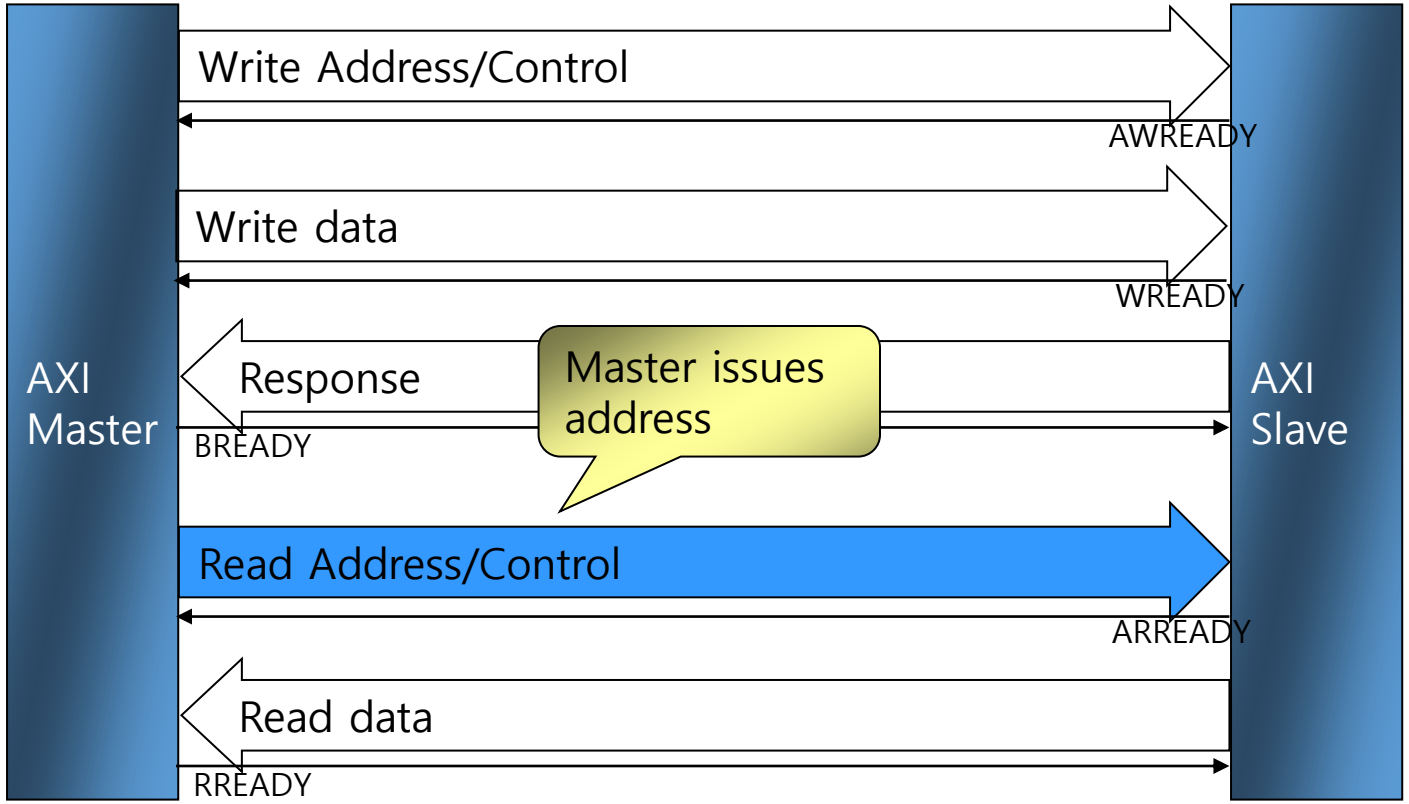
# Split Transaction: Write (2/3)



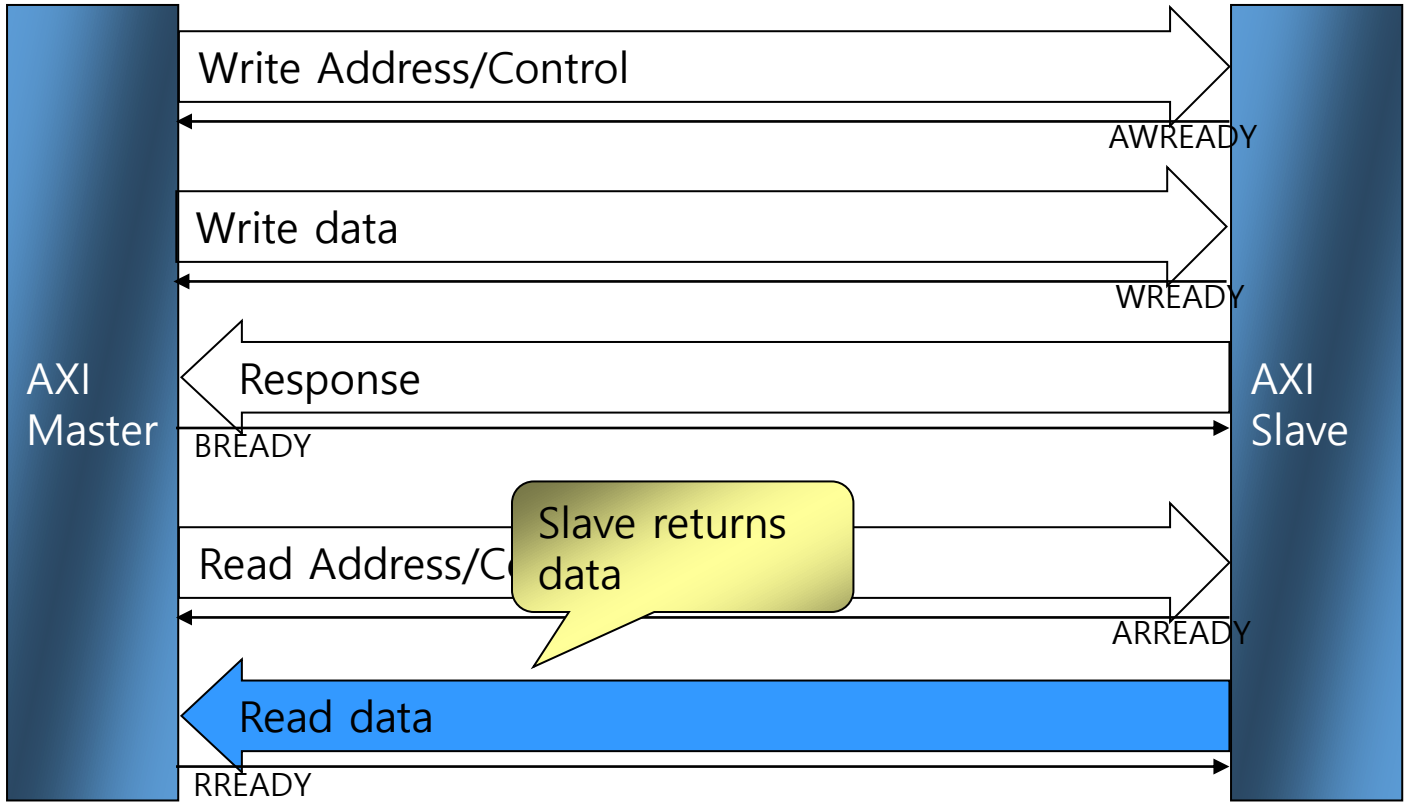
# Split Transaction: Write (3/3)



# Split Transaction: Read (1/2)

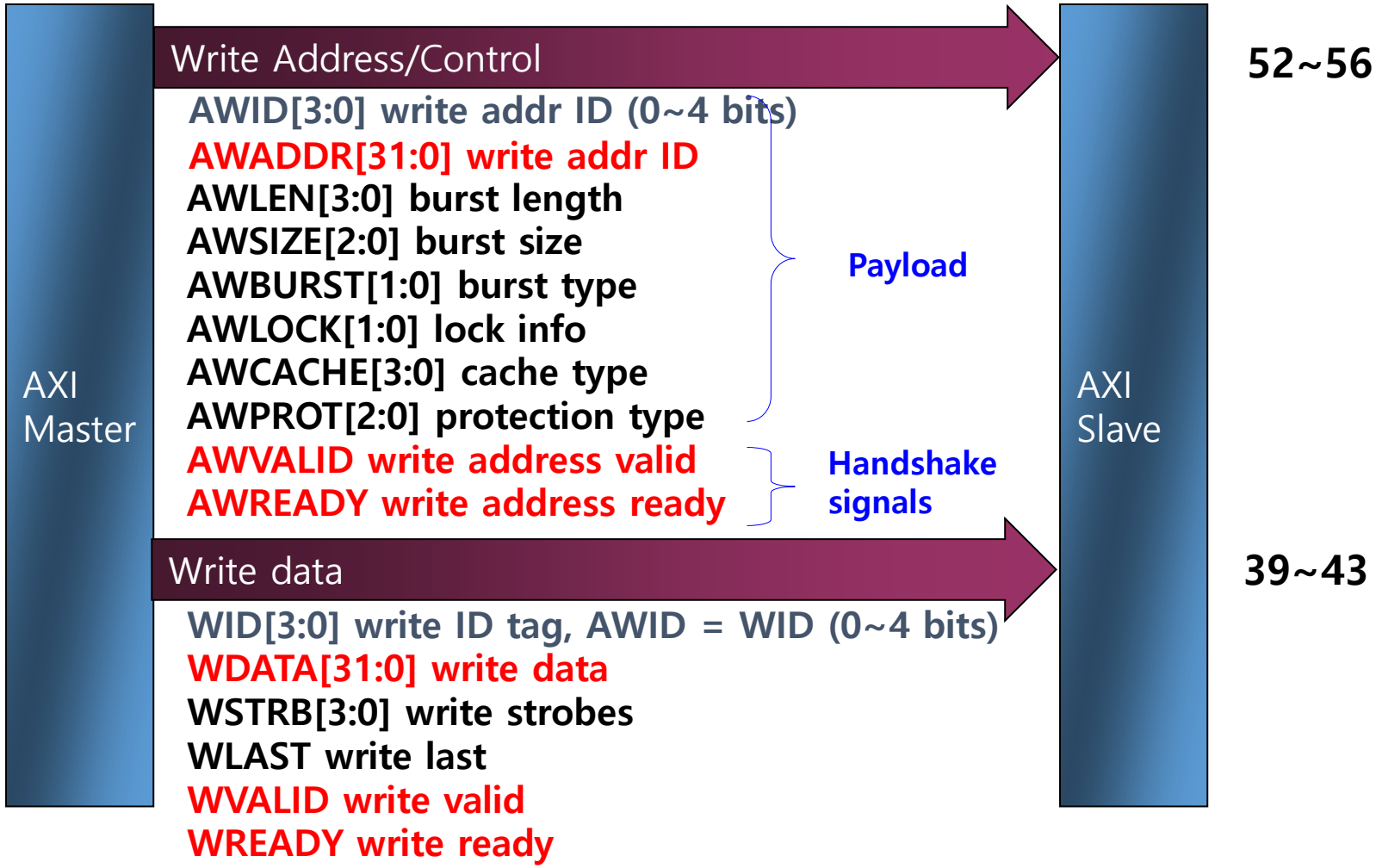


# Split Transaction: Read (2/2)



# 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



[Source: AXI Spec]

# AMBA3 Advanced eXtensable Interface (AXI) Protocol

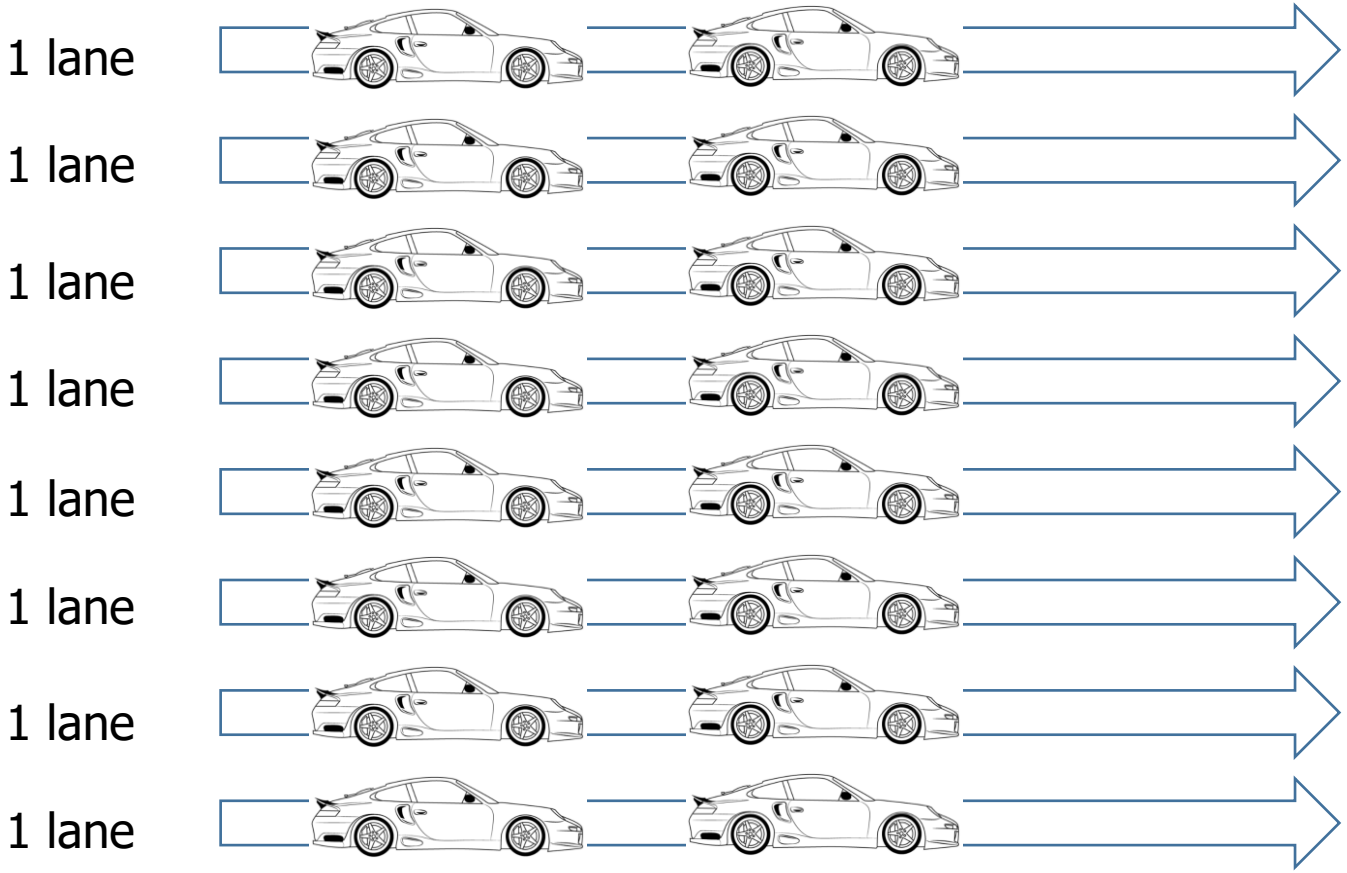
- Multiple channels
- Narrow and wide transfer
- Single credit-based flow control: valid and ready signals
- Burst
- Split transaction to overlap request and data transfer
- Multiple outstanding requests
- Implementation issue: connectivity and arbitration

# On-Chip Bus for Multiple-Byte Data Transfer

- On-chip bus ~ Road with multiple lanes (1 lane = 1 byte)

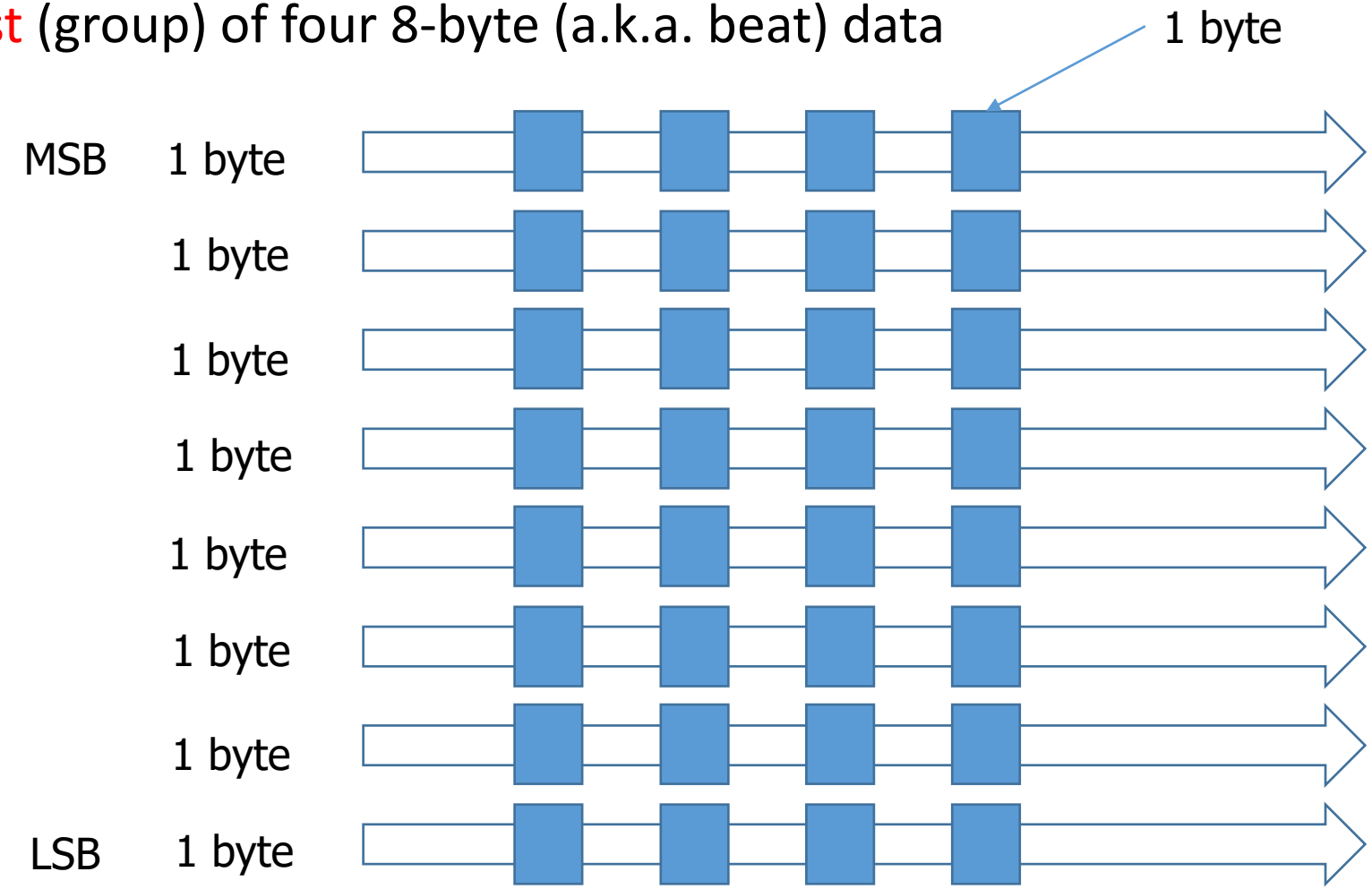
# On-Chip Bus for Multiple-Byte Data Transfer

- A byte on the bus ~ a car on the road



# On-Chip Bus for Multiple-Byte Data Transfer

- A burst (group) of four 8-byte (a.k.a. beat) data

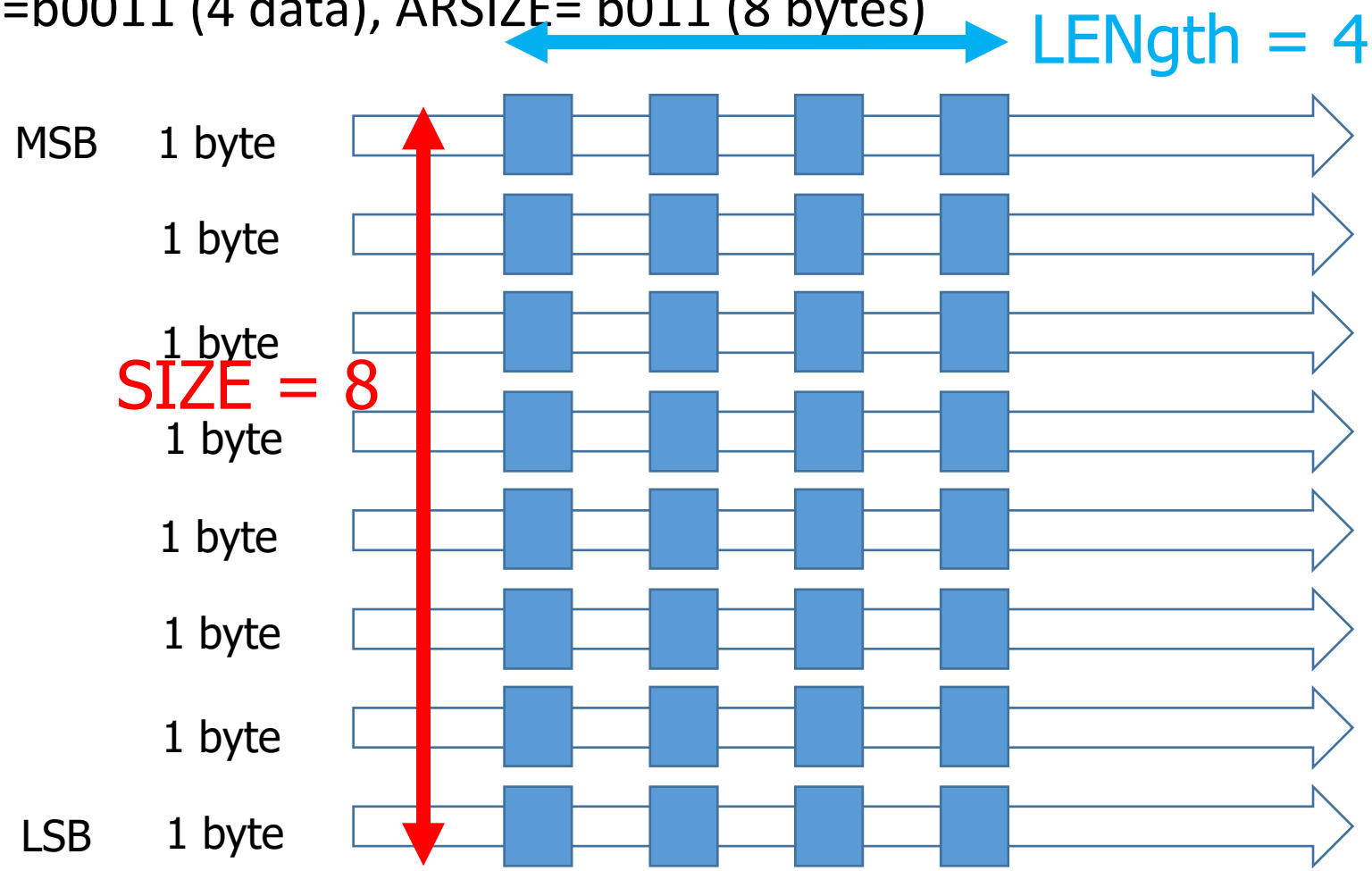


# (Burst) Length and (Beat) Size

- ARLEN=b0011 (4 data), ARSIZE= b011 (8 bytes)

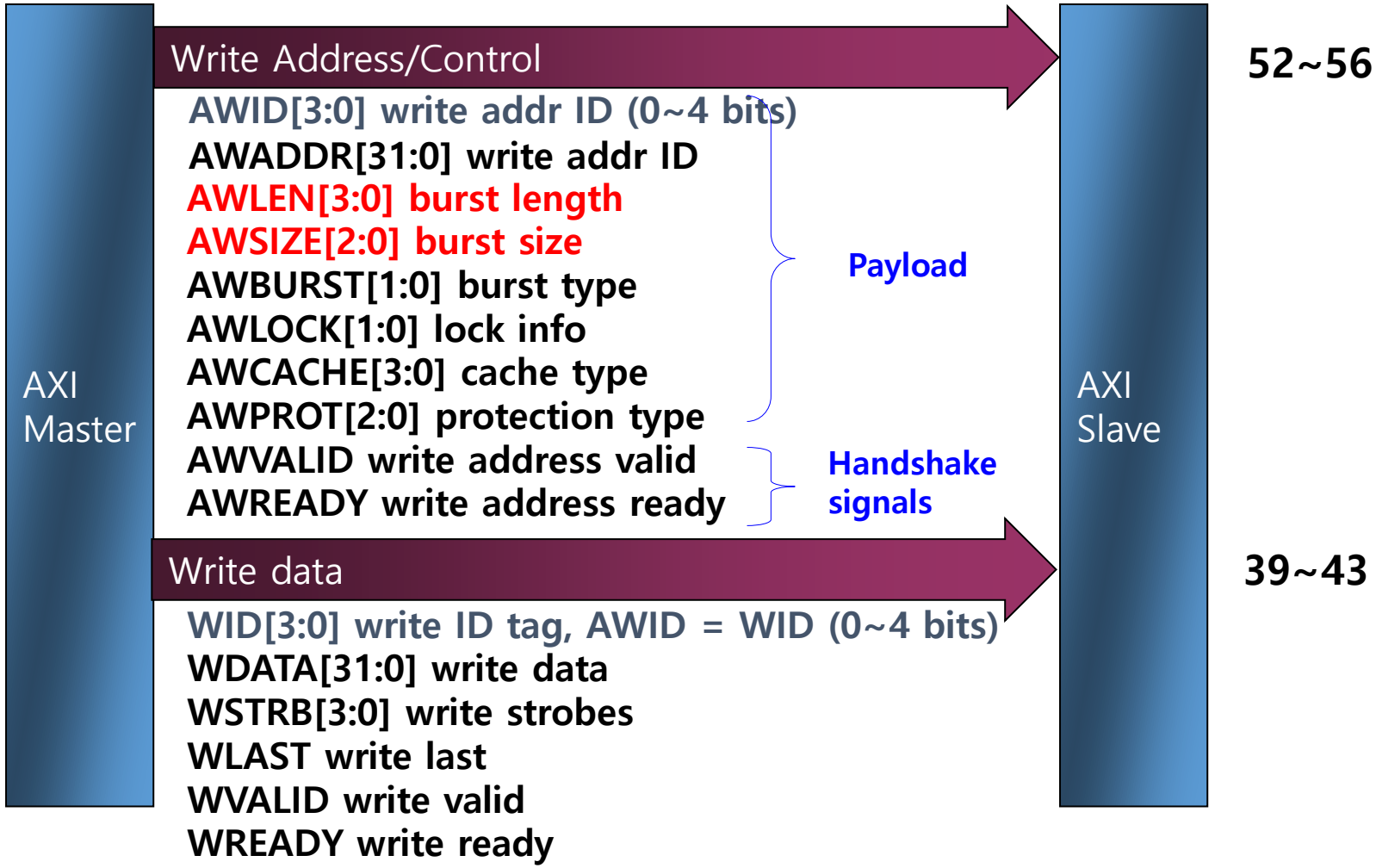
Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)



# 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



# Burst Length, Size and Type

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| . . . | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

**Table 4-2 Burst size encoding**

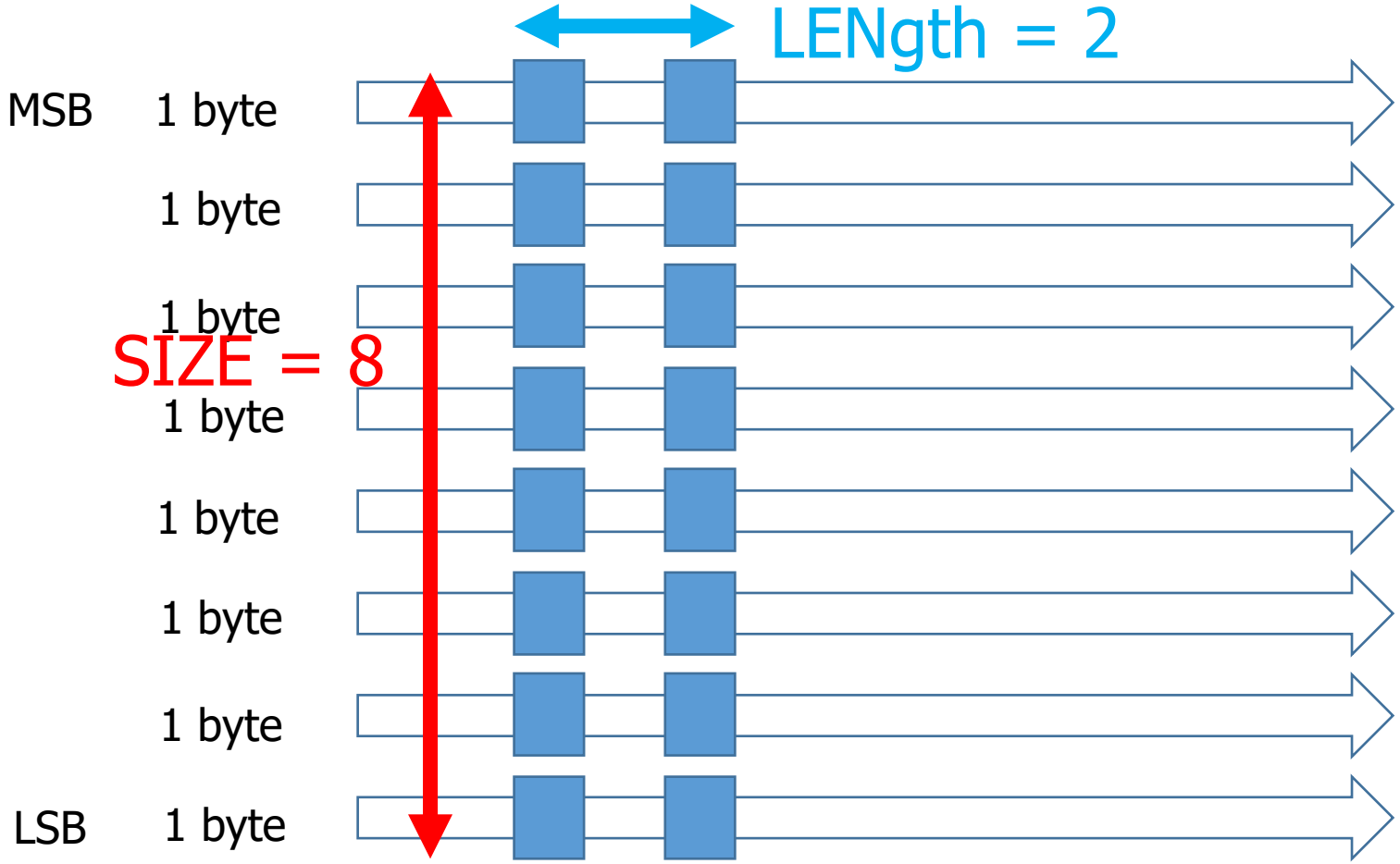
| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-3 Burst type encoding**

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

# (Burst) Length and (Beat) Size

- ARLEN=b0001 (2 data), ARSIZE= b011 (8 bytes)




Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)

**Table 4-2 Burst size encoding**

| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# (Burst) Length and (Beat) Size

- ARLEN=b0000 (1 data), ARSIZE= b010 (4 bytes)

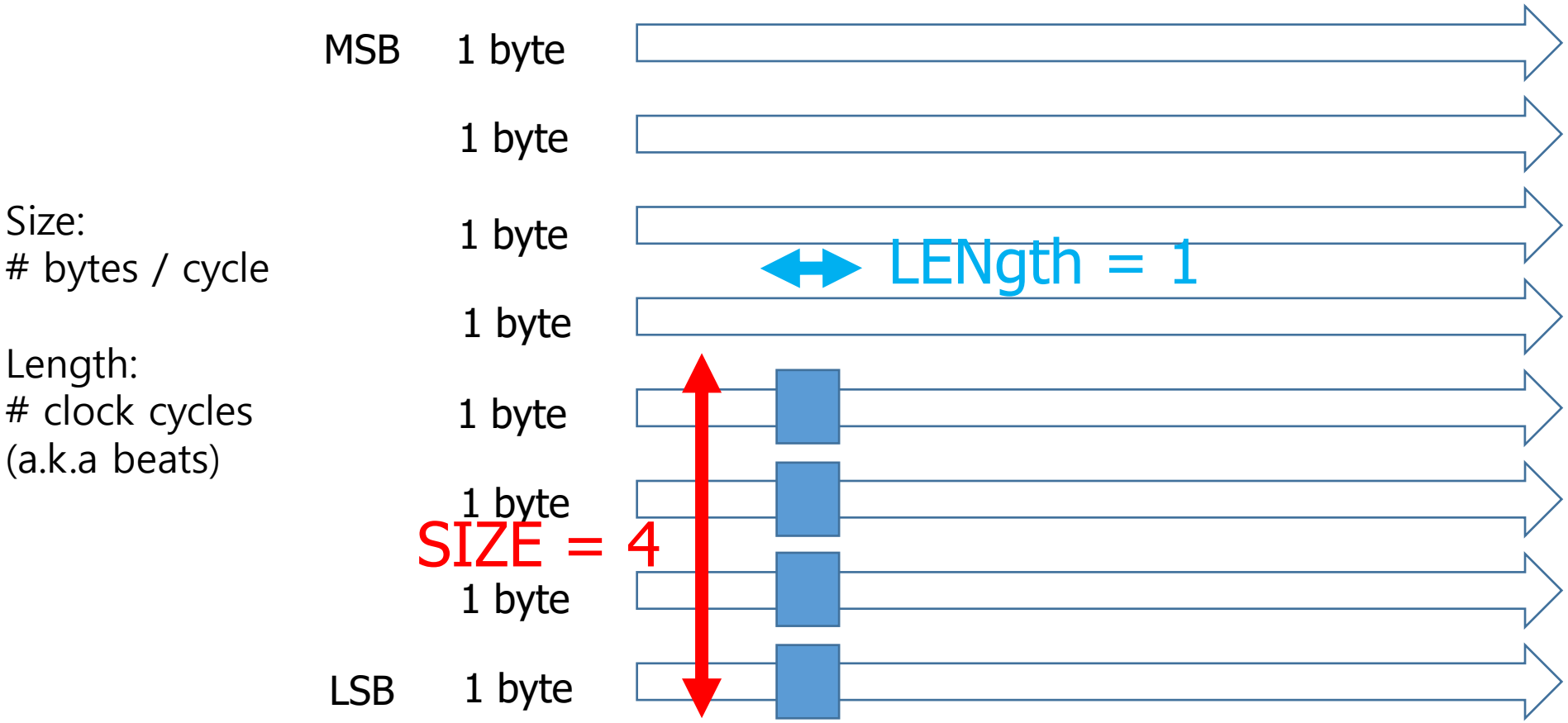



**Table 4-2 Burst size encoding**

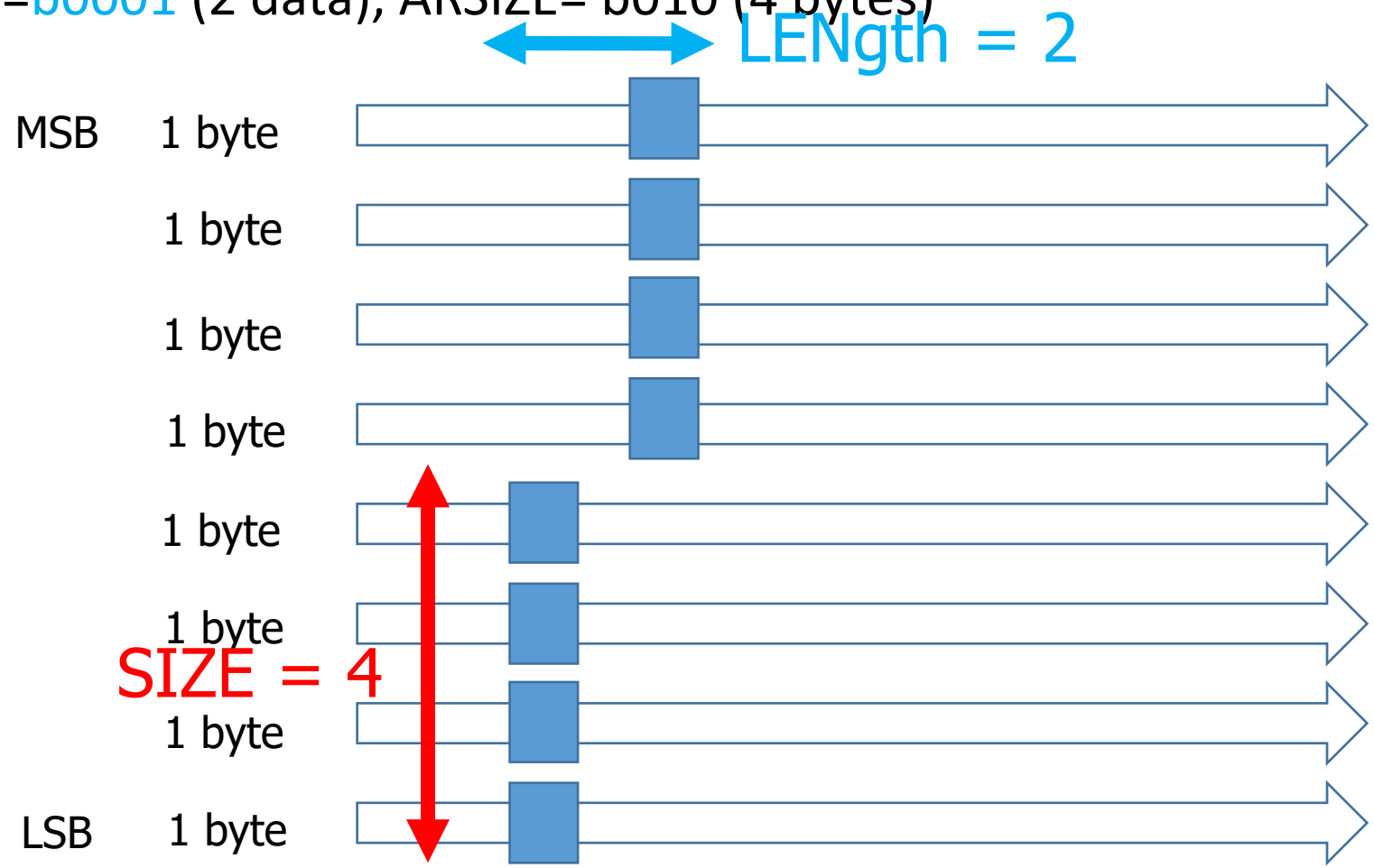
| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# (Burst) Length and (Beat) Size

- ARLEN=b0001 (2 data), ARSIZE= b010 (4 bytes)




Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)

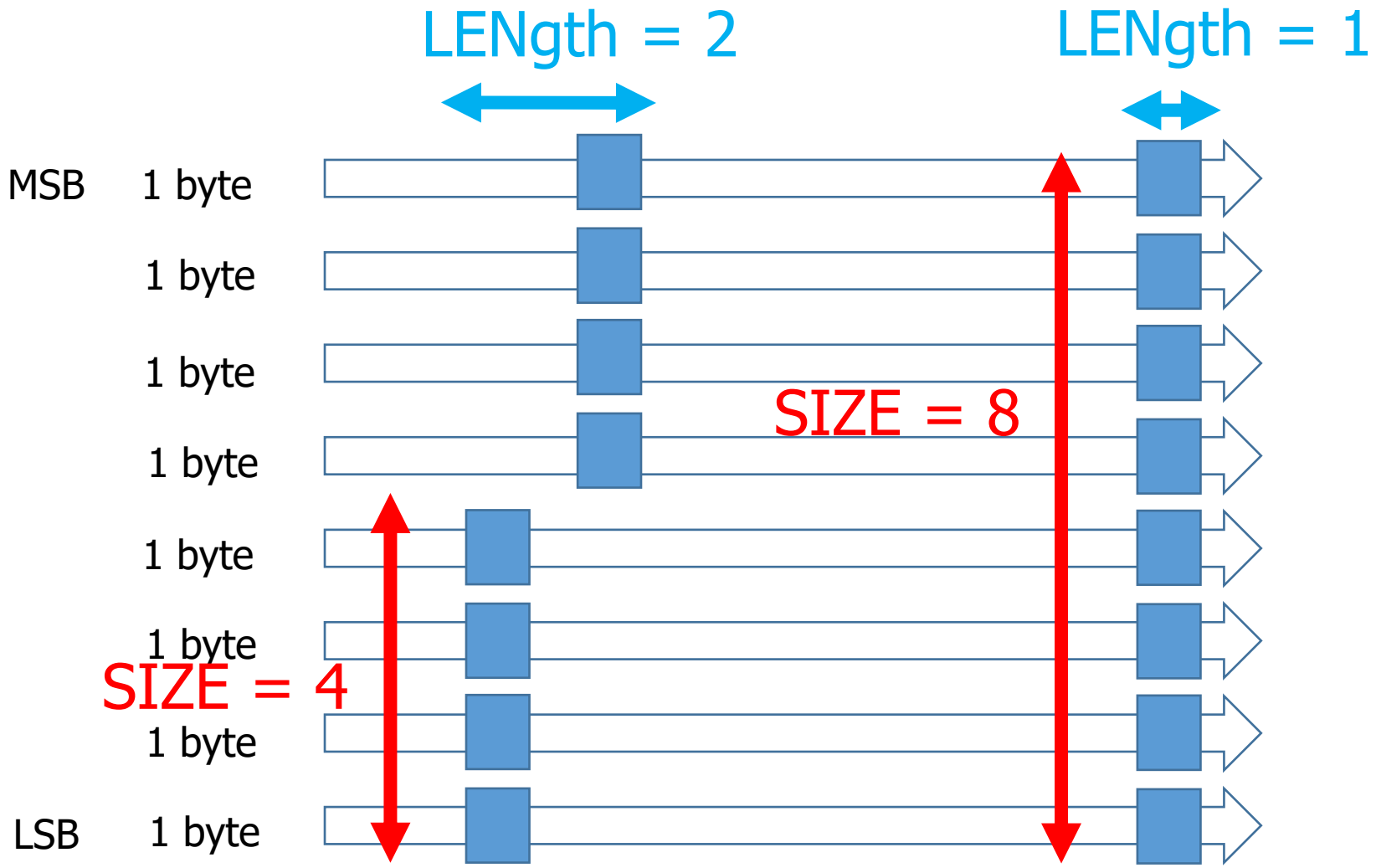
**Table 4-2 Burst size encoding**

| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

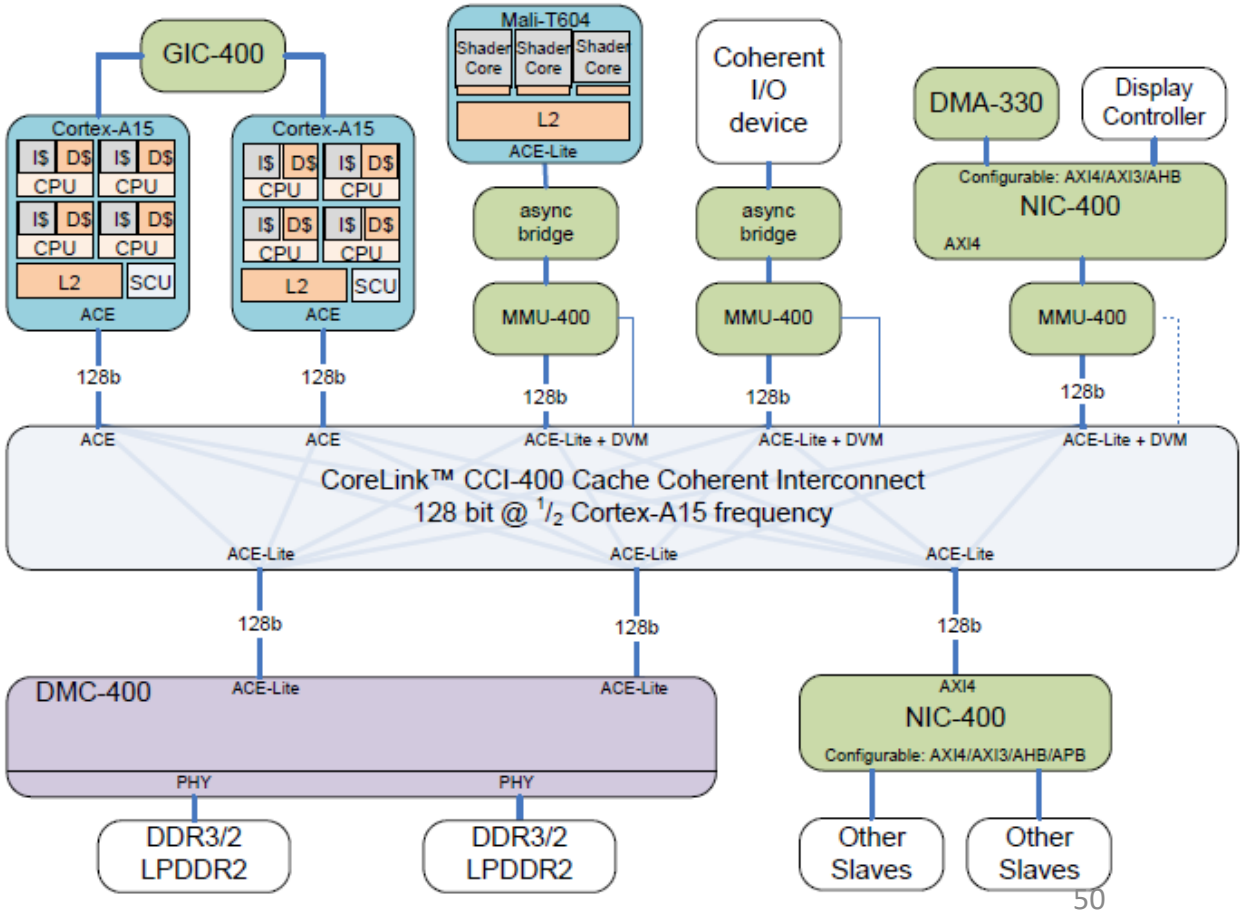
# Why Length=2 and Size=4, Instead of Length=1 and Size=8?



[Stevens, 2011]

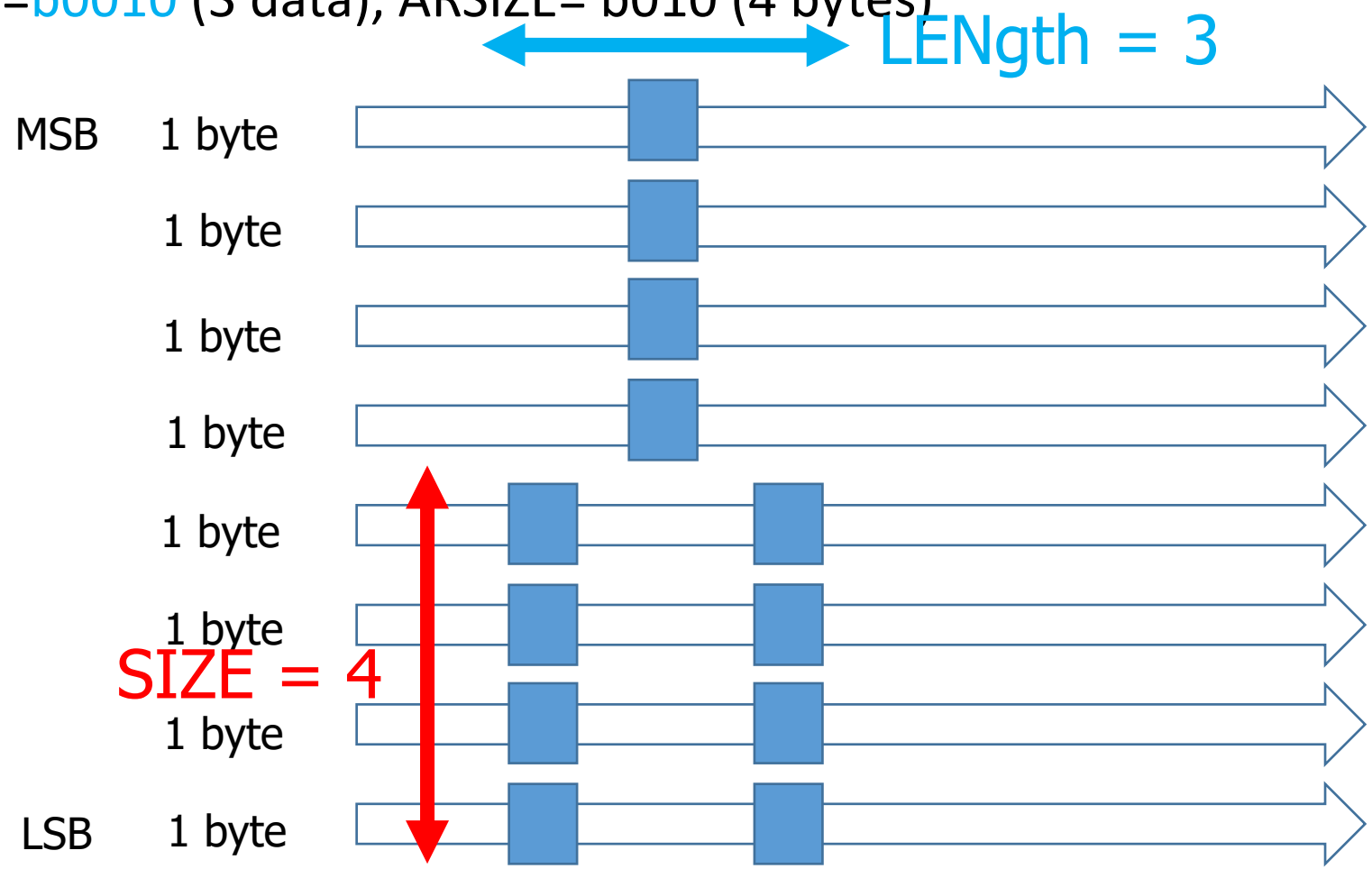
# Hardware Components are Mostly Reused from Previous Designs

- A legacy (i.e., old) HW component with narrow data (e.g., 4 bytes) can be used on a wider (e.g., 8 byte wide) interconnect
- In such a case, there are two options
  - Adding buffer logic which can make 8 byte data for 2 cycles and send it in 1 cycle
    - Size = 8 and Length = 1
  - Use the legacy HW as it is
    - Size = 4 and Length = 2



# (Burst) Length and (Beat) Size

- ARLEN=b0010 (3 data), ARSIZE= b010 (4 bytes)




Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)

**Table 4-2 Burst size encoding**

| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# Narrow Transfer

- ARLEN=b0001 (2 data), ARSIZE= b000 (1 byte)

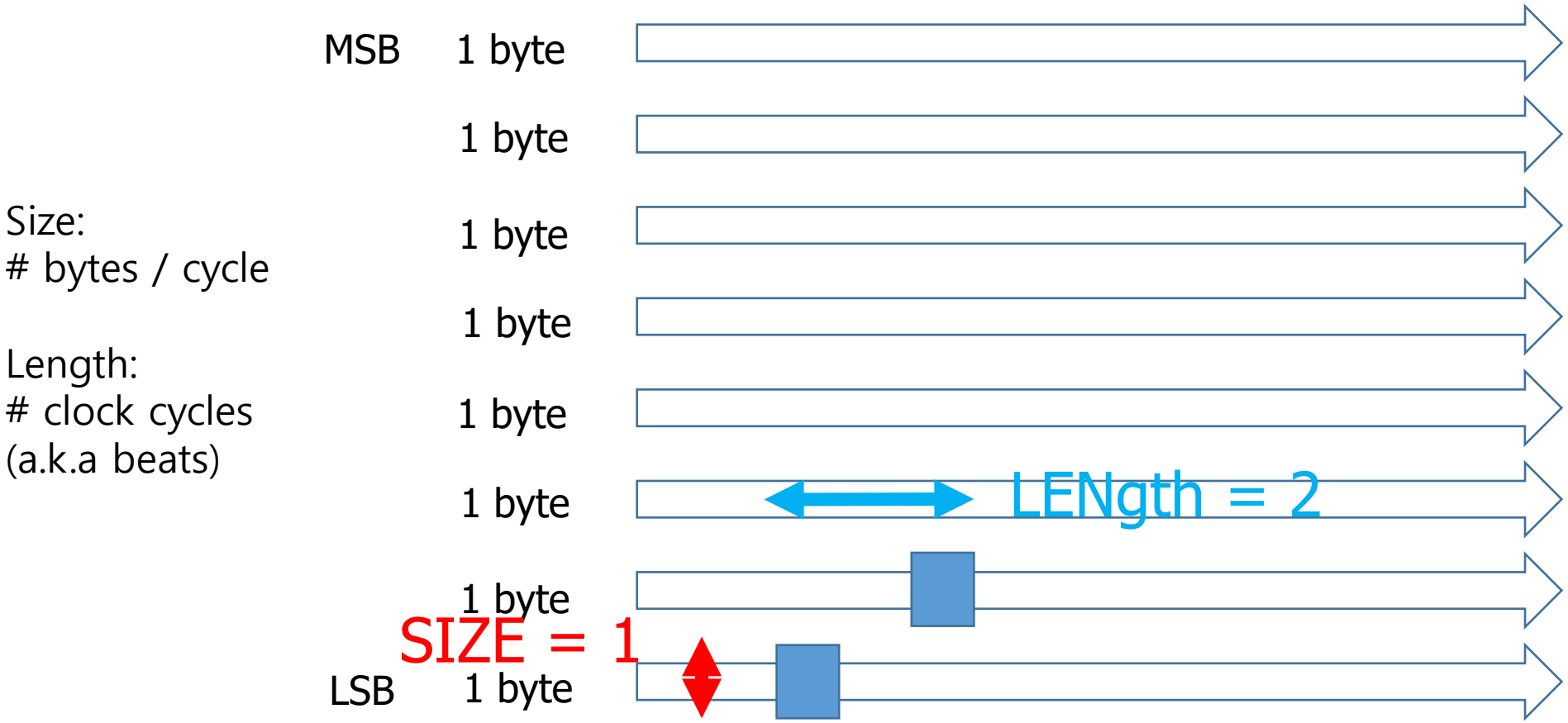



**Table 4-2 Burst size encoding**

| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# Narrow Transfer

- ARLEN=b0011 (4 data), ARSIZE= b000 (1 byte)

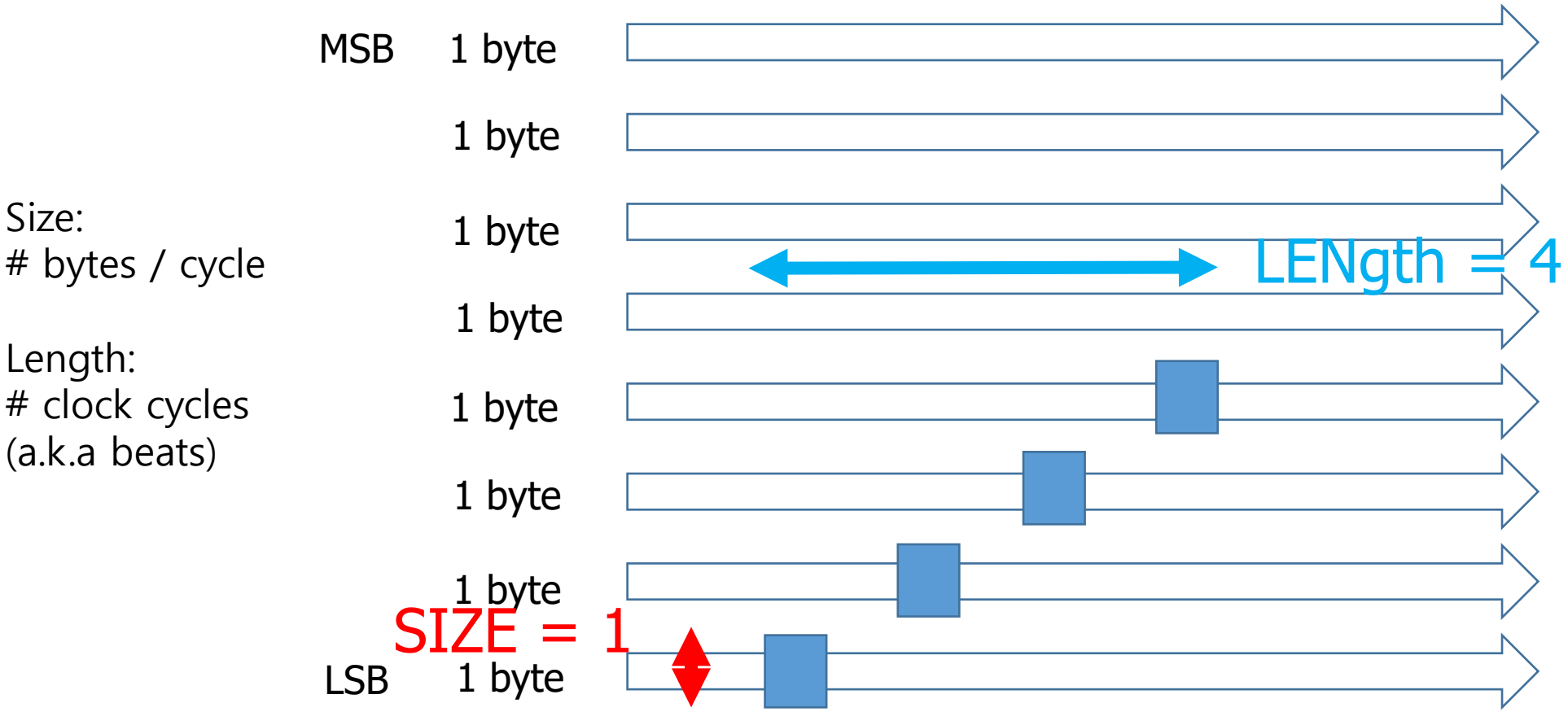



**Table 4-2 Burst size encoding**

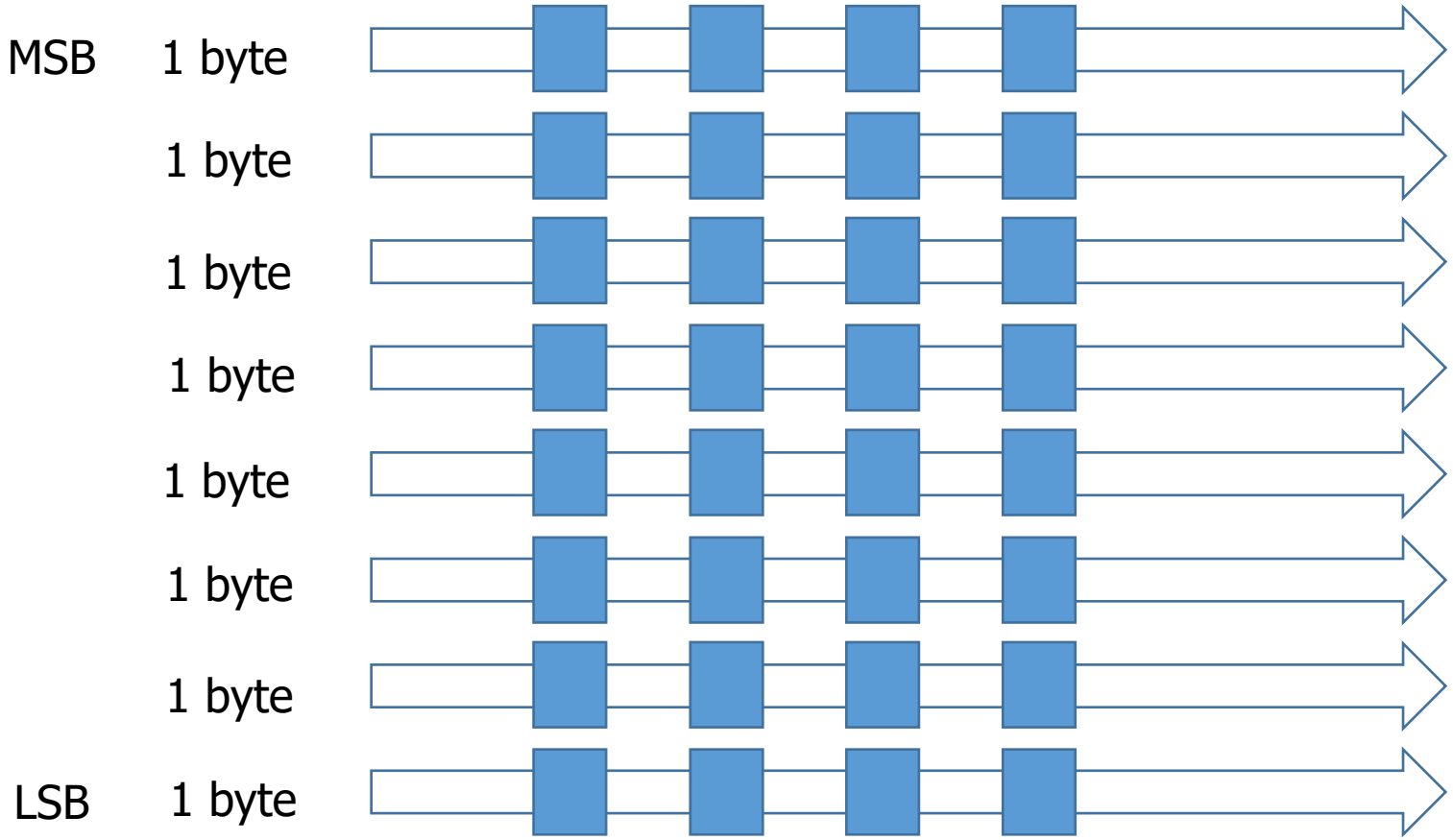
| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# Wide Transfer

- ARLEN=b0011 (4 data), ARSIZE= b011 (8 bytes)




Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)

**Table 4-2 Burst size encoding**

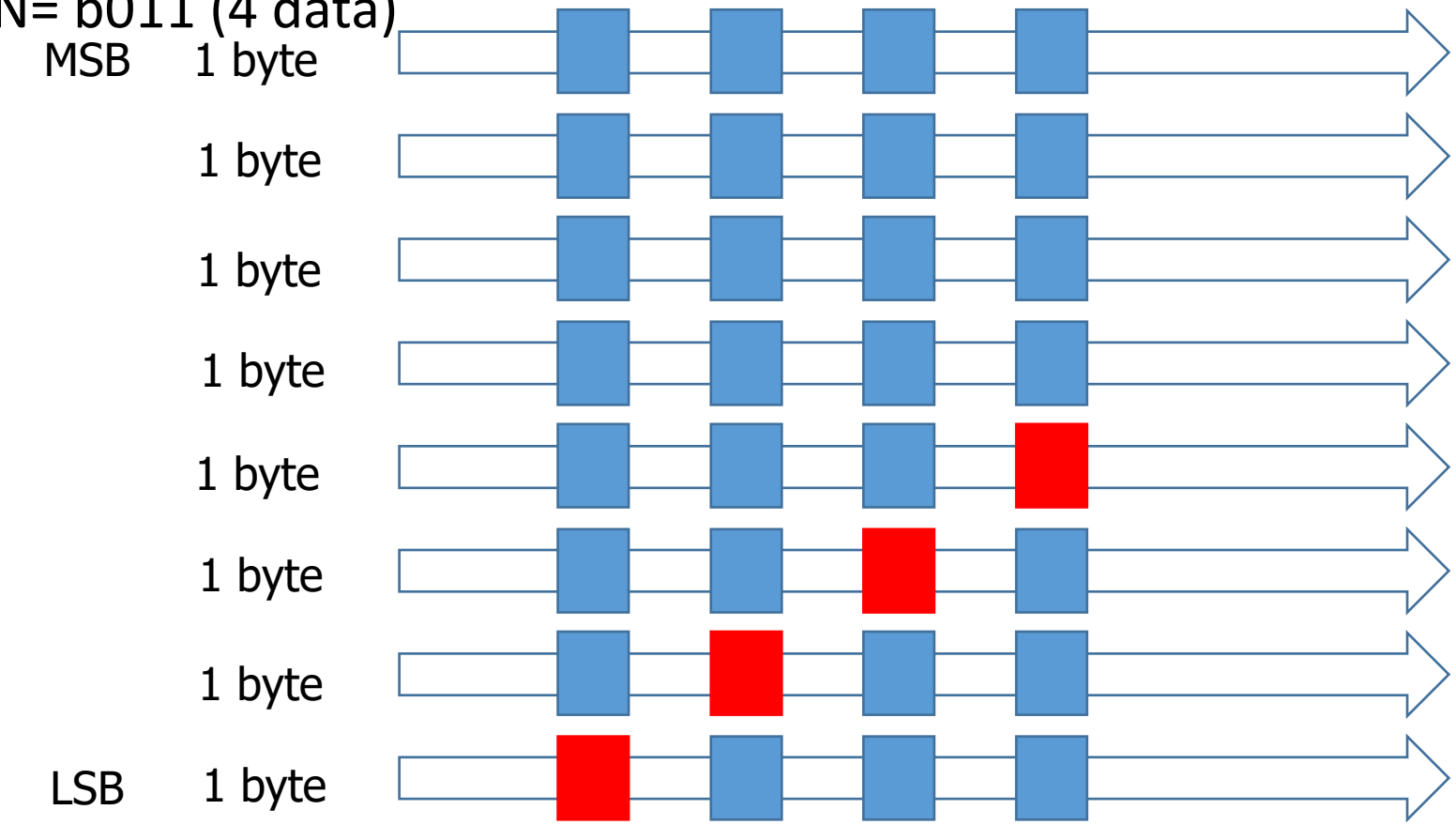
| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# Wide vs. Narrow Transfer

- ARSIZE= b011 (8 bytes) vs. b000 (1 byte)
- ARLEN= b011 (4 data)




Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)

**Table 4-2 Burst size encoding**

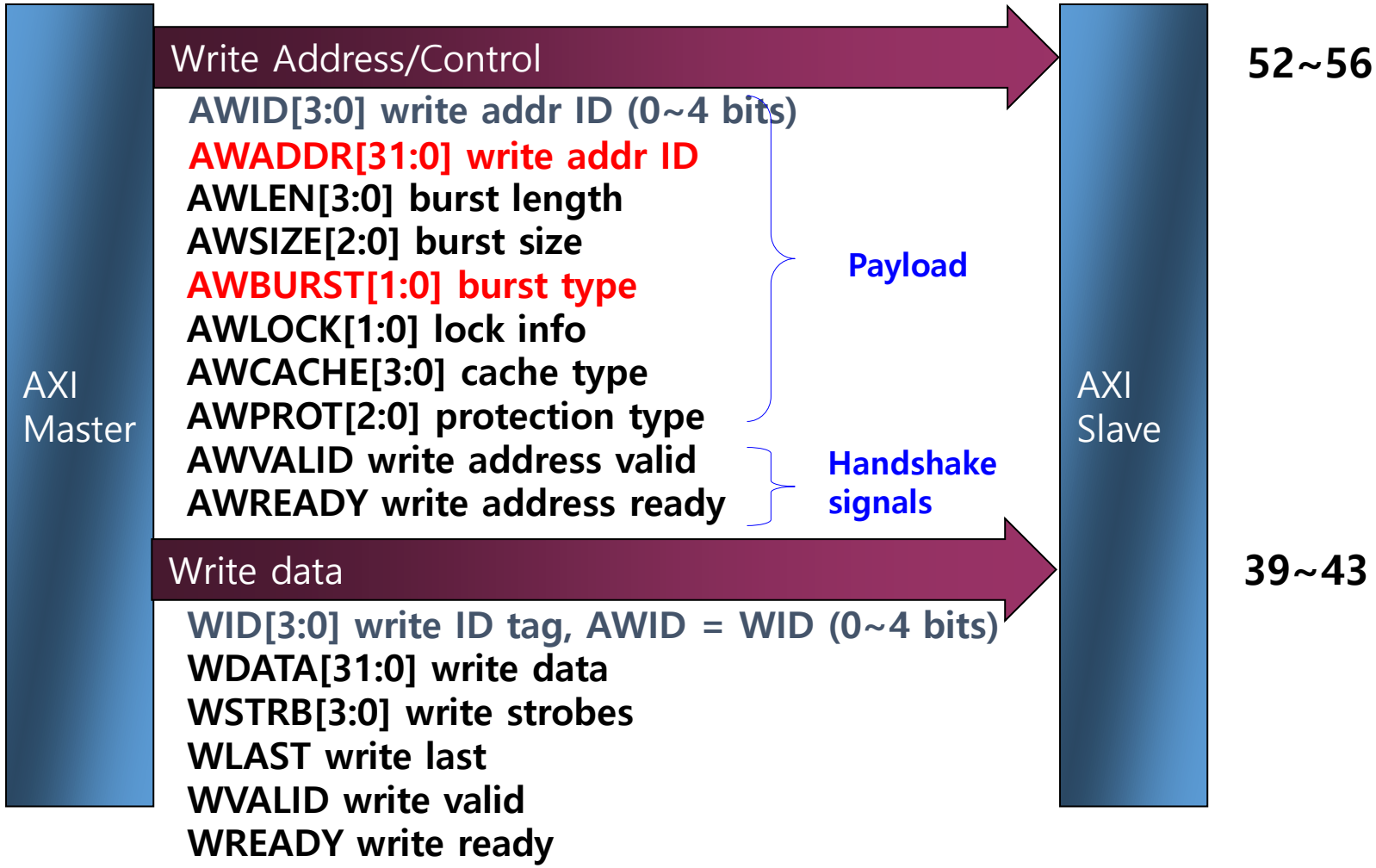
| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

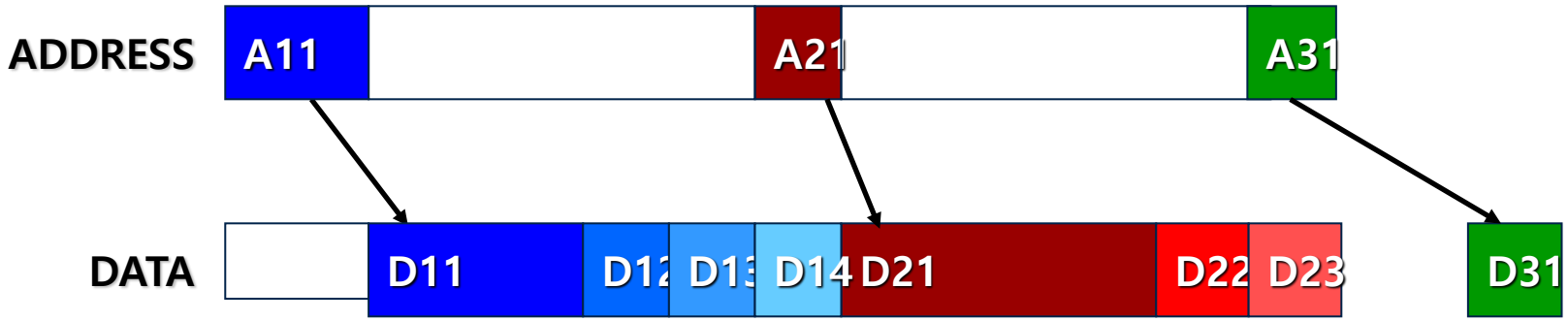
| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# 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



# One Address for 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)

# Burst Length, Size and Type

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| . . . | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

**Table 4-2 Burst size encoding**

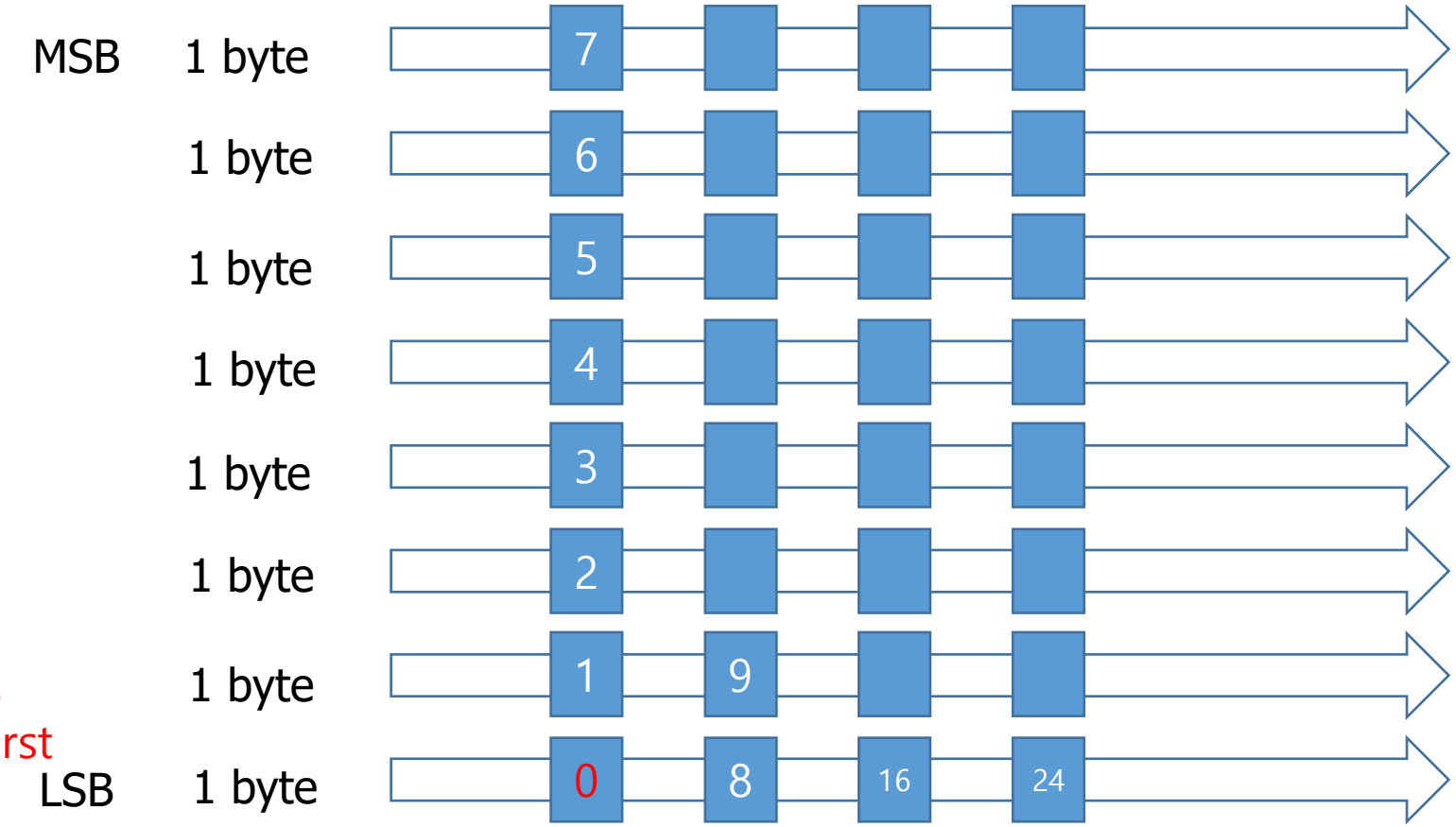
| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-3 Burst type encoding**

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

# Aligned Transfer

- AWADDR=b000, ARLEN=b0011 (4 data), ARSIZE= b010 (8 bytes)




Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)

AWADDR:
Starting address
of write data burst

**Table 4-2 Burst size encoding**

| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

# Unaligned Transfer

- AWADDR=b001, ARLEN=b0011 (4 data), ARSIZE= b010 (8 bytes)

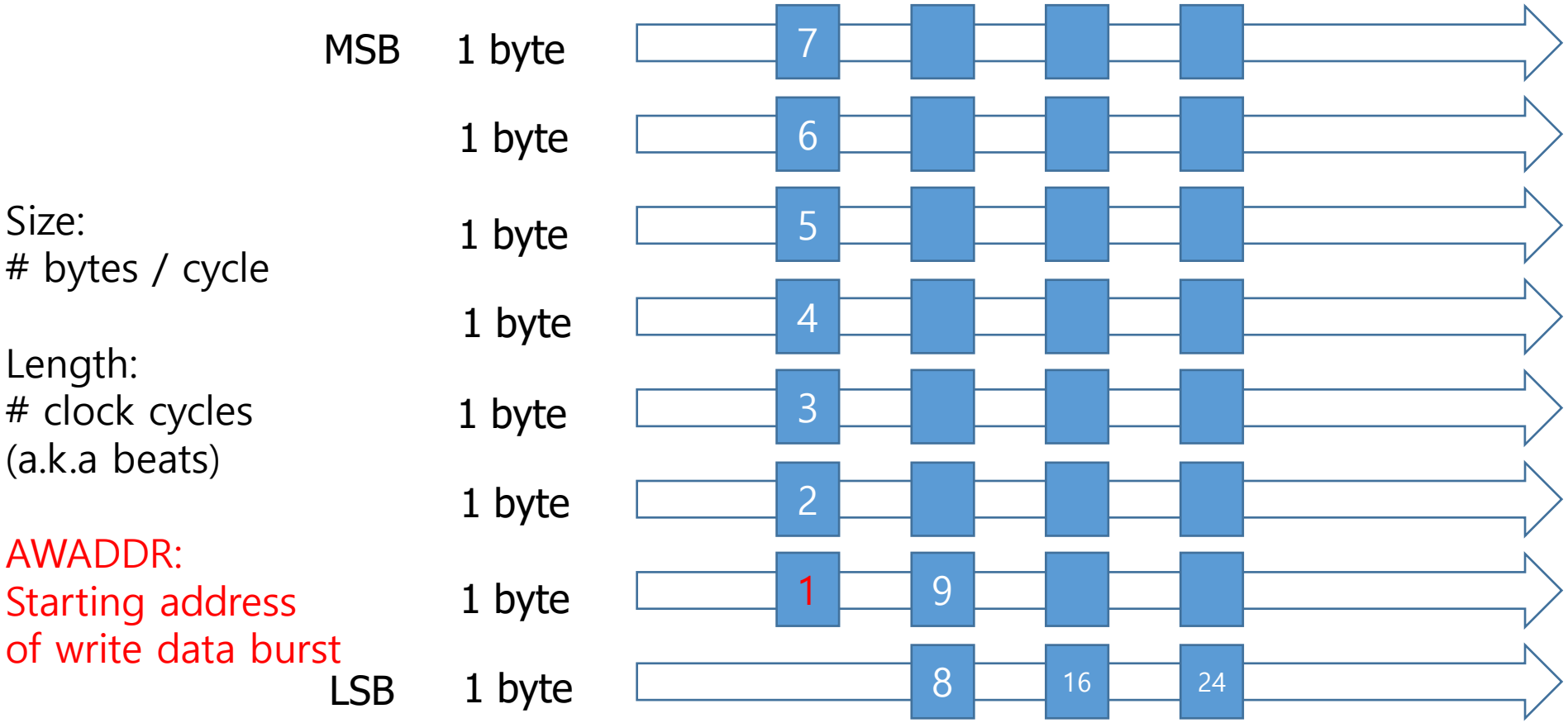



**Table 4-2 Burst size encoding**

| ARSIZE[2:0] AWSIZE[2:0] | Bytes in transfer |
|---|---|
| b000 | 1 |
| b001 | 2 |
| b010 | 4 |
| b011 | 8 |
| b100 | 16 |
| b101 | 32 |
| b110 | 64 |
| b111 | 128 |

**Table 4-1 Burst length encoding**

| ARLEN[3:0] AWLEN[3:0] | Number of data transfers |
|---|---|
| b0000 | 1 |
| b0001 | 2 |
| b0010 | 3 |
| ... | |
| b1101 | 14 |
| b1110 | 15 |
| b1111 | 16 |

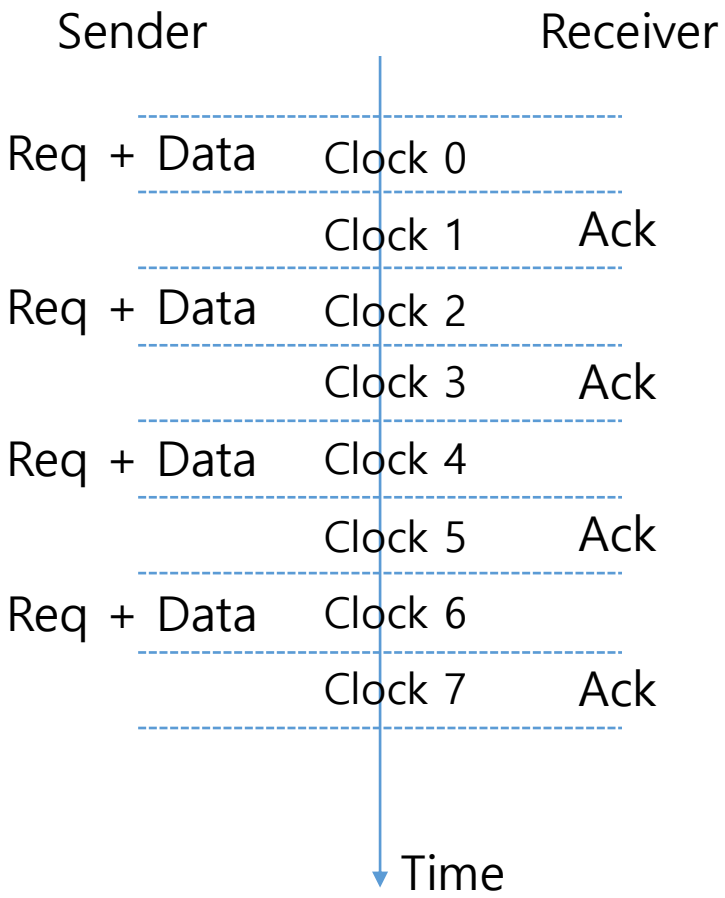
[Source: AXI Spec]

# AMBA3 Advanced eXtensable Interface (AXI) Protocol

- Multiple channels
- Narrow and wide transfer
- Single credit-based flow control: valid and ready signals
- Burst
- Split transaction to overlap request and data transfer
- Multiple outstanding requests
- Implementation issue: connectivity and arbitration

# Stop-and-Wait

- Request and Acknowledge

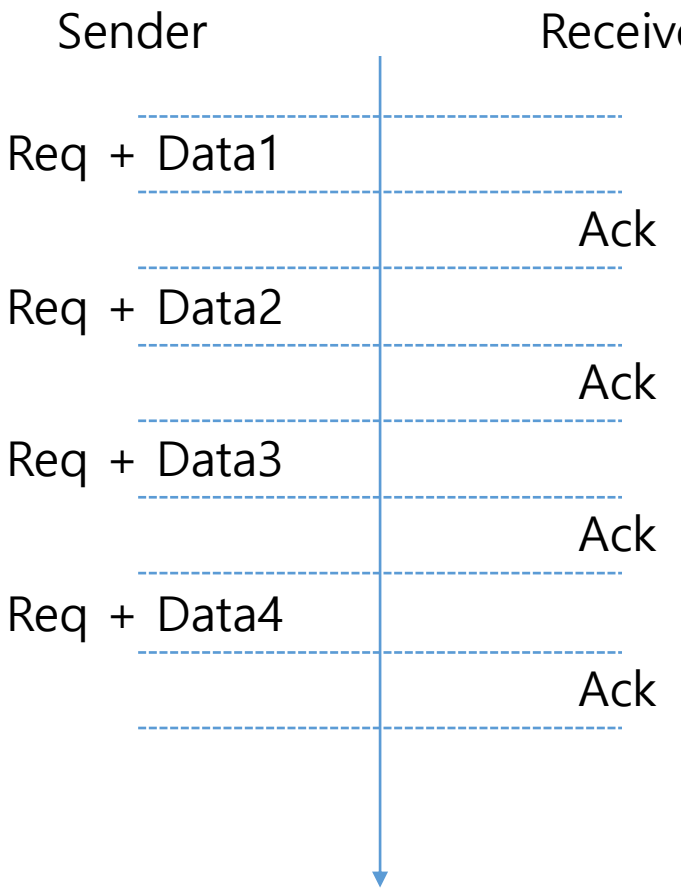


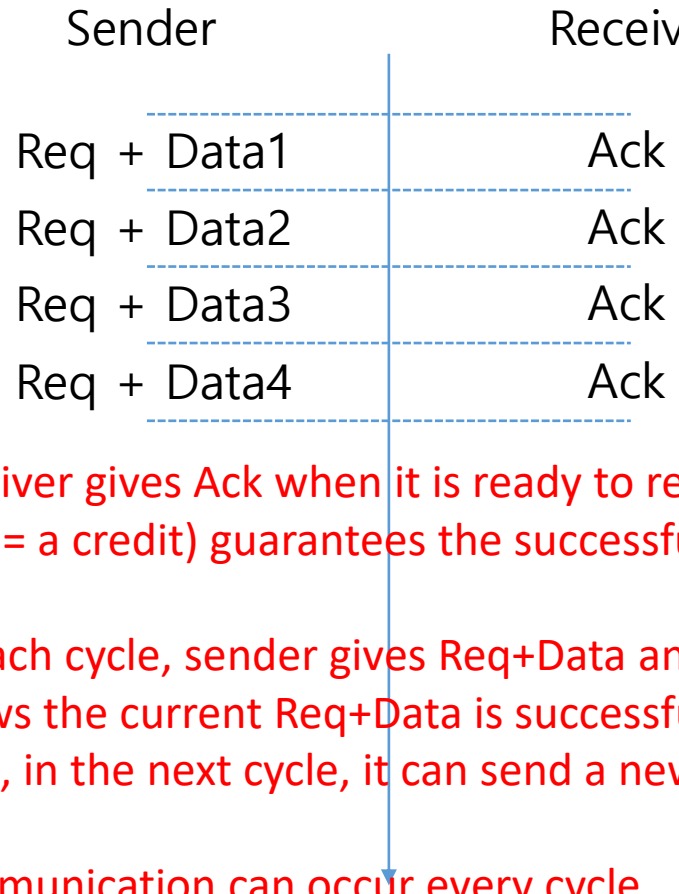

Receiver gives Ack as soon as it receives Req + Data from the sender (at clock 0)
However, the sender sees the Ack in the next cycle (clock 1)
Thus, the sender sends a new Req+Data in the next cycle (clock 2)

Communication occurs every two cycles

# Single Credit-based Flow Control

- e.g., AMBA3 protocol. Ack (ready) = Credit




Receiver gives Ack when it is ready to receive Req+Data
Ack (= a credit) guarantees the successful data transfer

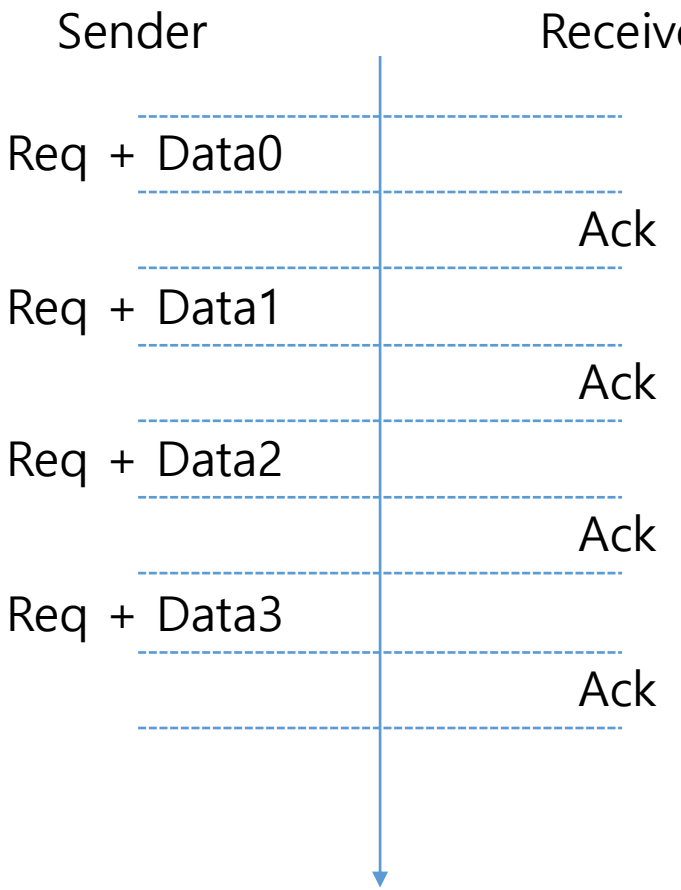
At each cycle, sender gives Req+Data and, if Ack=1,
knows the current Req+Data is successfully received by Receiver
Thus, in the next cycle, it can send a new Req+Data

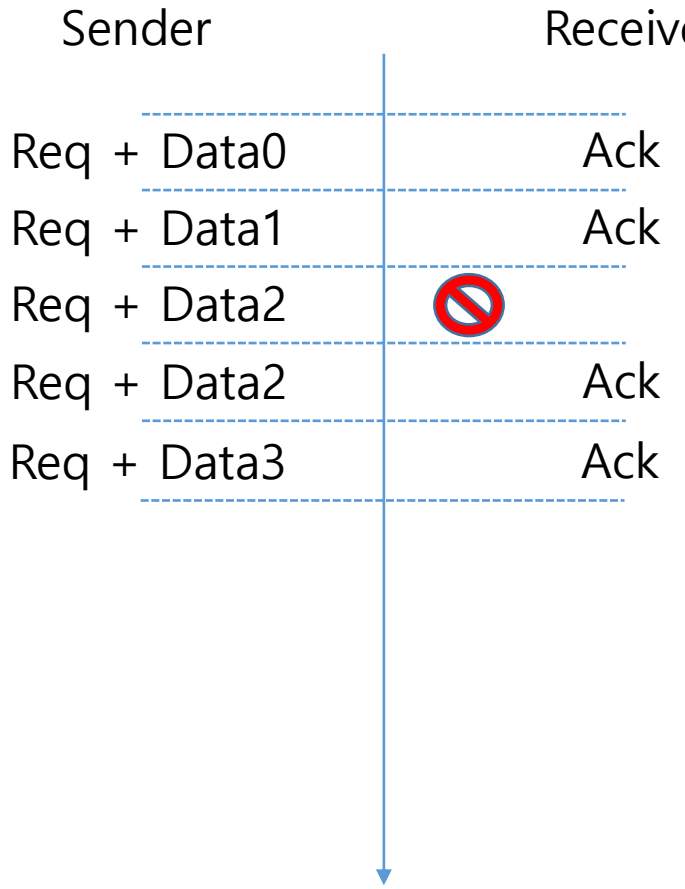
Communication can occur every cycle

# Single Credit-based Flow Control

- e.g., AMBA3 protocol. Ack (ready) = Credit

If the receiver can take the input, it sets Ack (ready) to high.




In this cycle, the receiver cannot take the input due to some reason

# Handshaking and Flow Control

- Valid and ready signals are used for handshaking and flow control
- At every clock cycle, sender and receiver perform the followings
- Sender
  - If there is any information to send, then it places information to send on the associated signals by raising valid signal
  - If the receiver raised ready signal, the sender considers that the information of the current cycle (that the sender placed on the bus signal) will be received by the receiver. Thus, the sender considers the communication of the current information is completed.
- Receiver
  - The receiver receives all the information and stores it temporarily.
  - If the receiver is busy, then it sets its ready signal to ‘0’ and ignore the information received from the master.
  - If the receiver is ready, then it sets its ready signal to ‘1’.
- Communication occurs only when both valid and ready are ‘1’ at clock rising edge.

# A Possible Deadlock in Valid/Ready Signaling

- Deadlock can occur in the following scenario
- Initial state: valid = ready = 0
- Sender
  - S.Valid = R.Ready // at clock rising edge, sender's valid is set to (=) receiver's ready
- Receiver
  - R.Ready = S.Valid // at clock rising edge, receiver's ready is set to (=) sender's valid
- Both sender and receiver cannot change the current state, i.e., valid=ready=0
- Thus, both cannot communicate with each other forever

# Avoiding Deadlock in Valid/Ready Signaling

- Sender
  - S.Valid = R.Ready // at clock rising edge, sender's valid = receiver's ready

**Dependency of valid on ready is NOT allowed! Valid should be independent of ready**

  - Instead, "the sender can assert valid anytime it wants to do a transfer regardless of the state of ready"
    - https://www.edaboard.com/threads/axi-arvalid-signal-issue.388604/

In AXI specification document

- the **VALID** signal of the AXI interface sending information must not be dependent on the **READY** signal of the AXI interface receiving that information
- an AXI interface that is receiving information can wait until it detects a **VALID** signal before it asserts its corresponding **READY** signal.

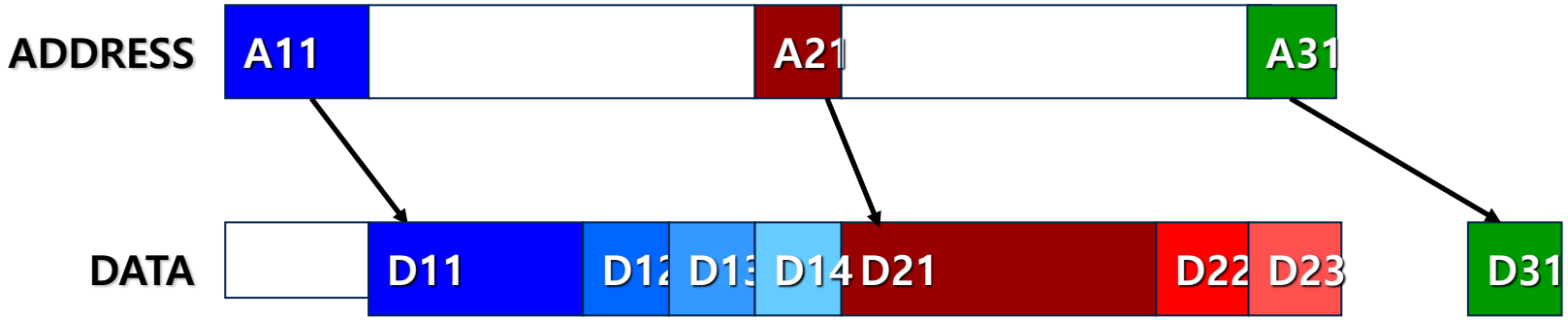
https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Relationships-between-the-channels/Dependencies-between-channel-handshake-signals?lang=en

[Source: AXI Spec]

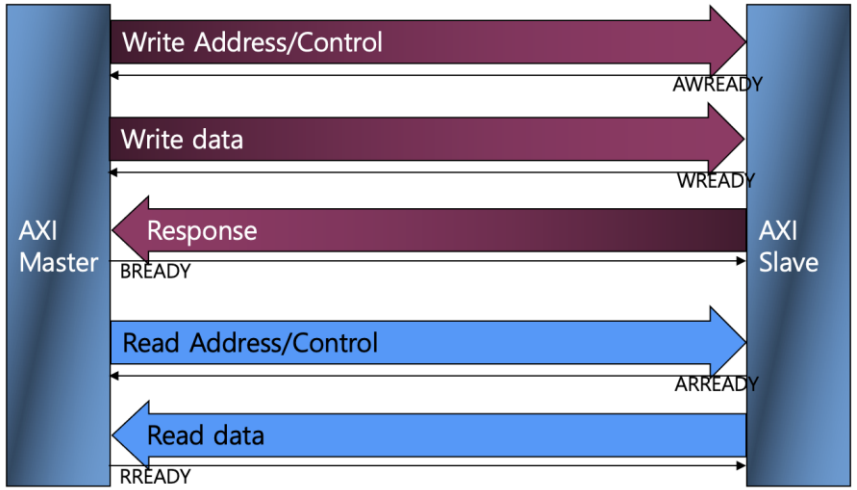
# AMBA3 Advanced eXtensable Interface (AXI) Protocol

- Multiple channels
- Narrow and wide transfer
- Single credit-based flow control: valid and ready signals
- Burst
- Split transaction to overlap request and data transfer
- Multiple outstanding requests
- Implementation issue: connectivity and arbitration

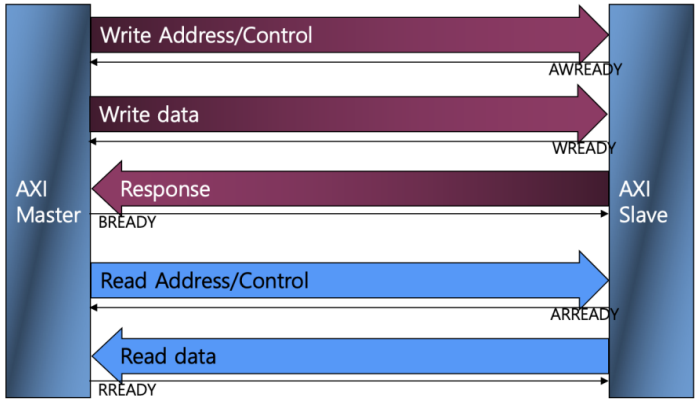
# One Address for Burst

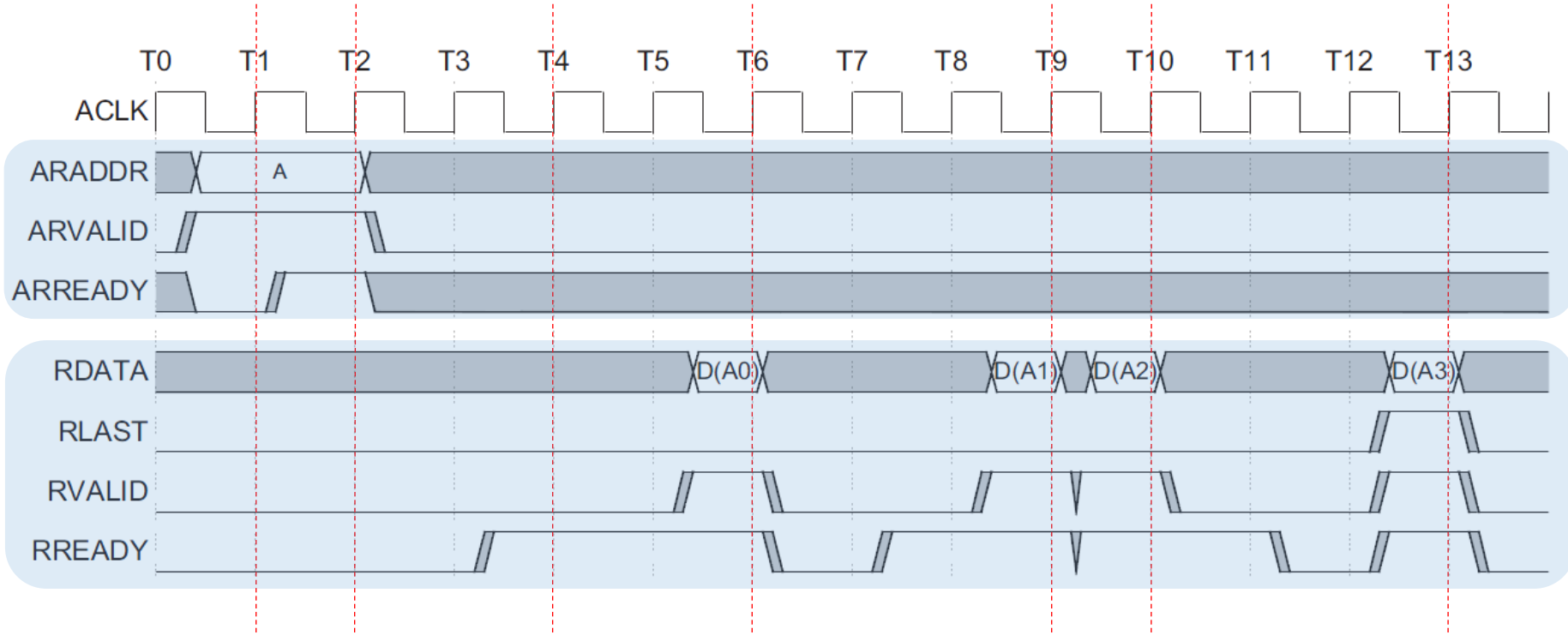



- Separation of address and data channel
  - Master provides the start address of burst
  - Slave needs to generate the remaining addresses based on burst type (FIXED, INCR, WRAP)



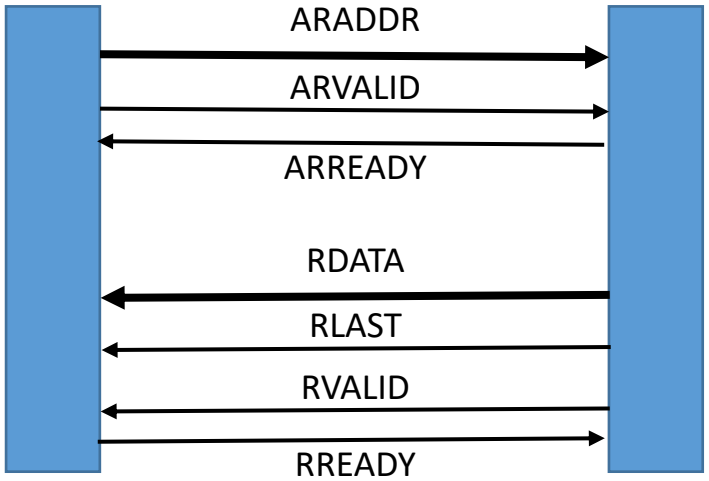
# Read Burst Operation




Master
e.g., CPU

Slave
e.g., Bus




Read request
is initiated

Data read
is ready

1st data
is transferred

The last data
is transferred

Read request
is accepted

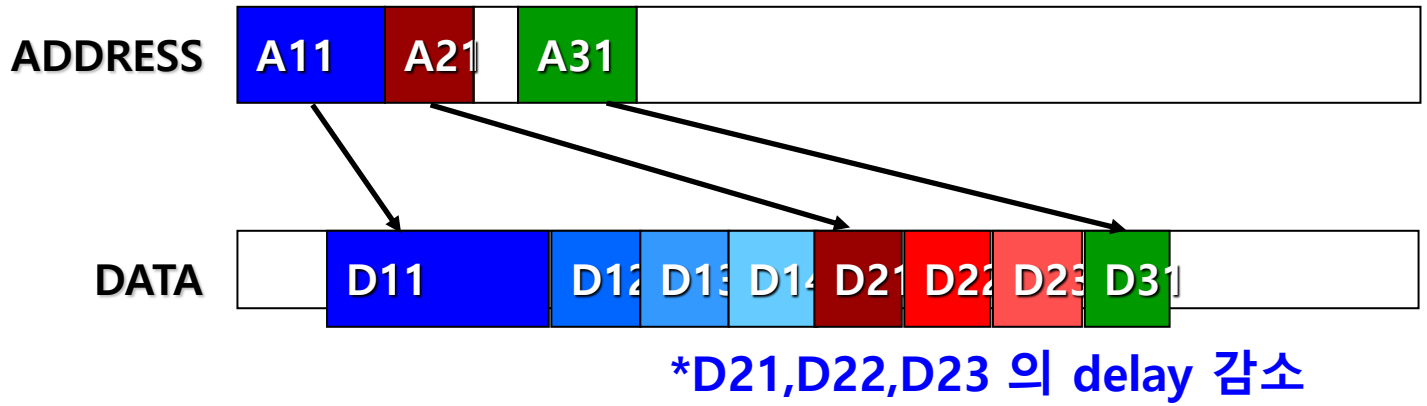
**Note: data transfer only when valid = ready = 1**

[Source: AXI Spec]

# AMBA3 Advanced eXtensable Interface (AXI) Protocol

- Multiple channels
- Narrow and wide transfer
- Single credit-based flow control: valid and ready signals
- Burst
- Split transaction to overlap request and data transfer
- Multiple outstanding requests
- Implementation issue: connectivity and arbitration

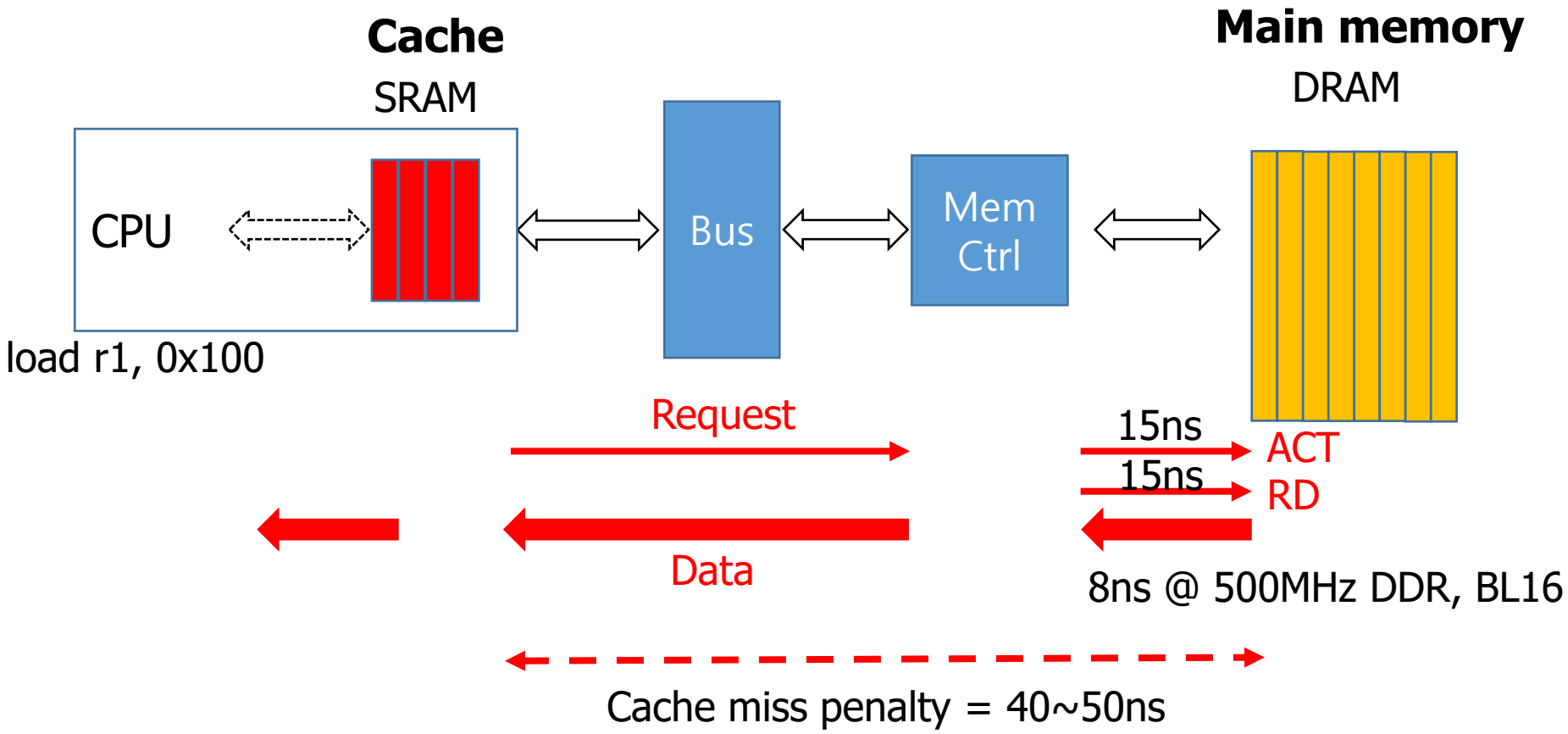
# Benefit of Split Transaction: Multiple Outstanding Requests




- Parameters for multiple outstanding requests
  - Master I/F: Issuing capability → master가 generation 할 수 있는 outstanding request의 개수
  - Slave I/F: Acceptance capability → slave가 받아 들일 수 있는 outstanding request의 개수

# CPU, Cache, and Main Memory

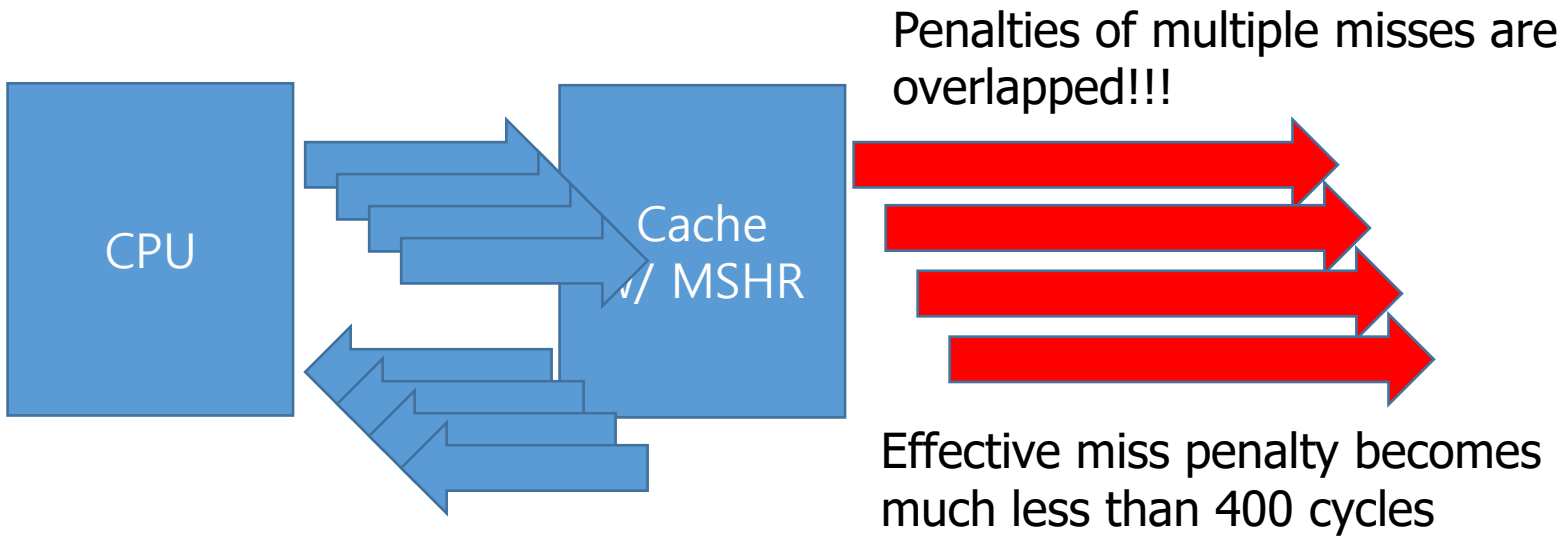
- In case of cache miss



[Source: J. Kubiatowicz, 2000]

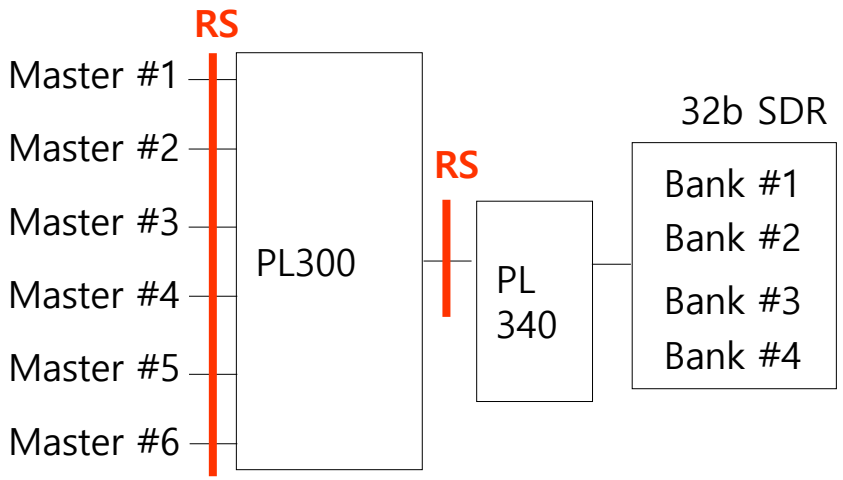
# Non-blocking Caches to Reduce Stalls on Misses

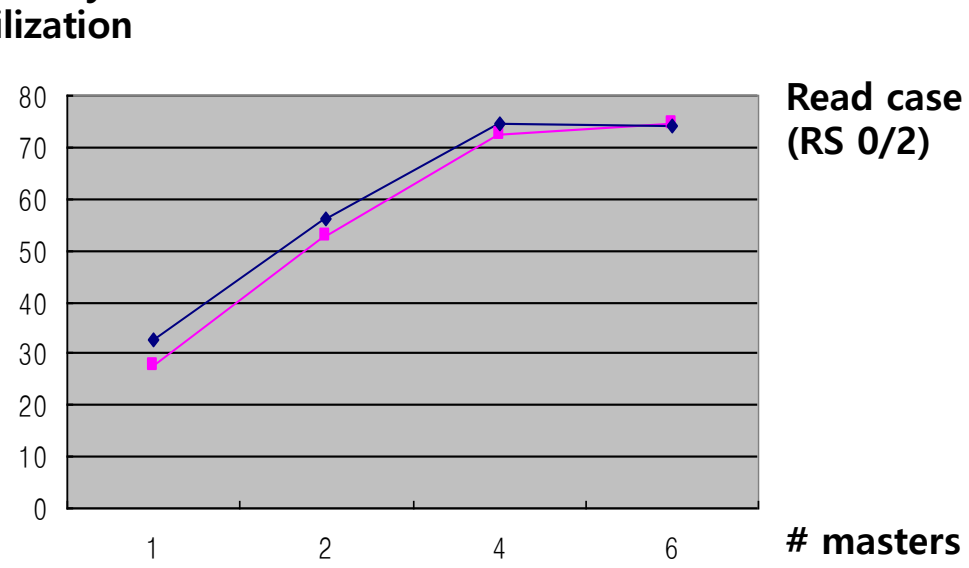
- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses



# Effect of Multiple Outstanding Requests

- Setup
  - Single outstanding by each master
  - 5000 clock cycles simulation

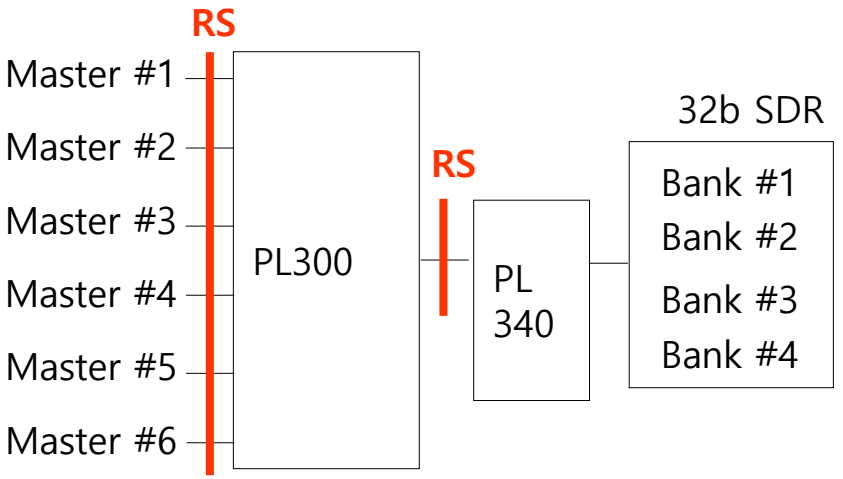


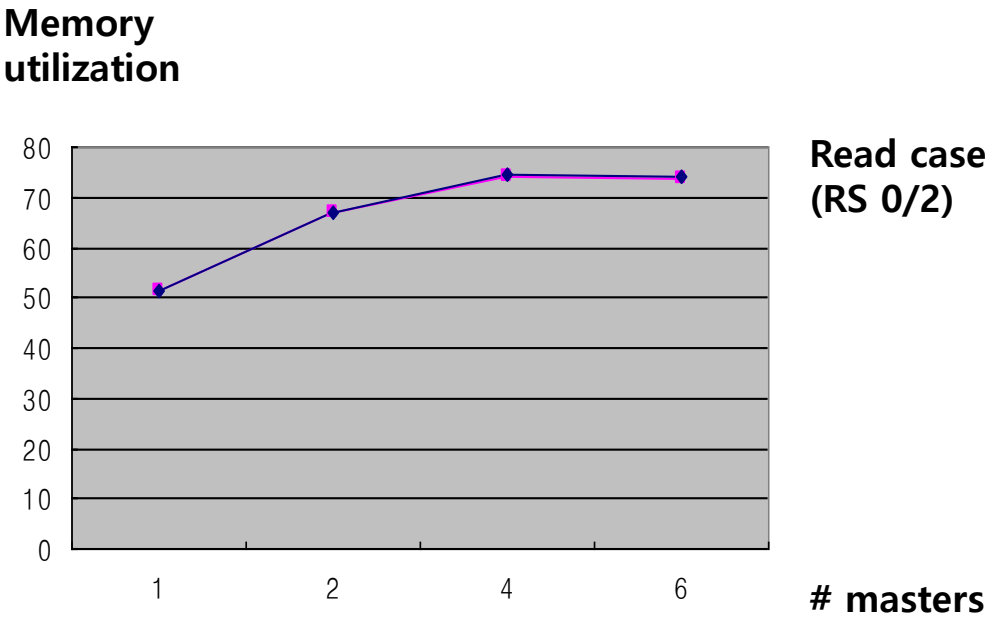




- Analysis
  - A single master w/o outstanding requests can achieve only about 30% utilization.
  - The effects of bank interleaving by different masters are significant, up to 74%.
  - 6 master case may suffer from bank conflicts.

# Effect of Multiple Outstanding Requests

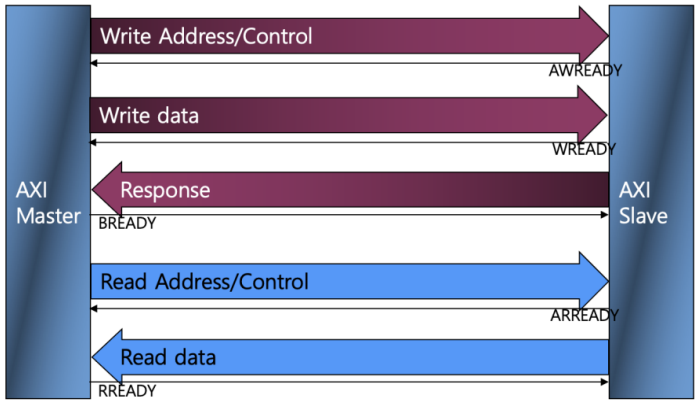
- Setup
  - Multiple outstanding request by each master

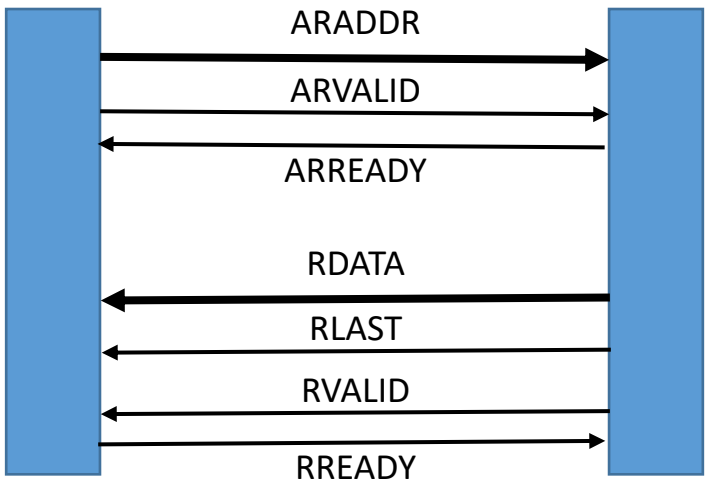


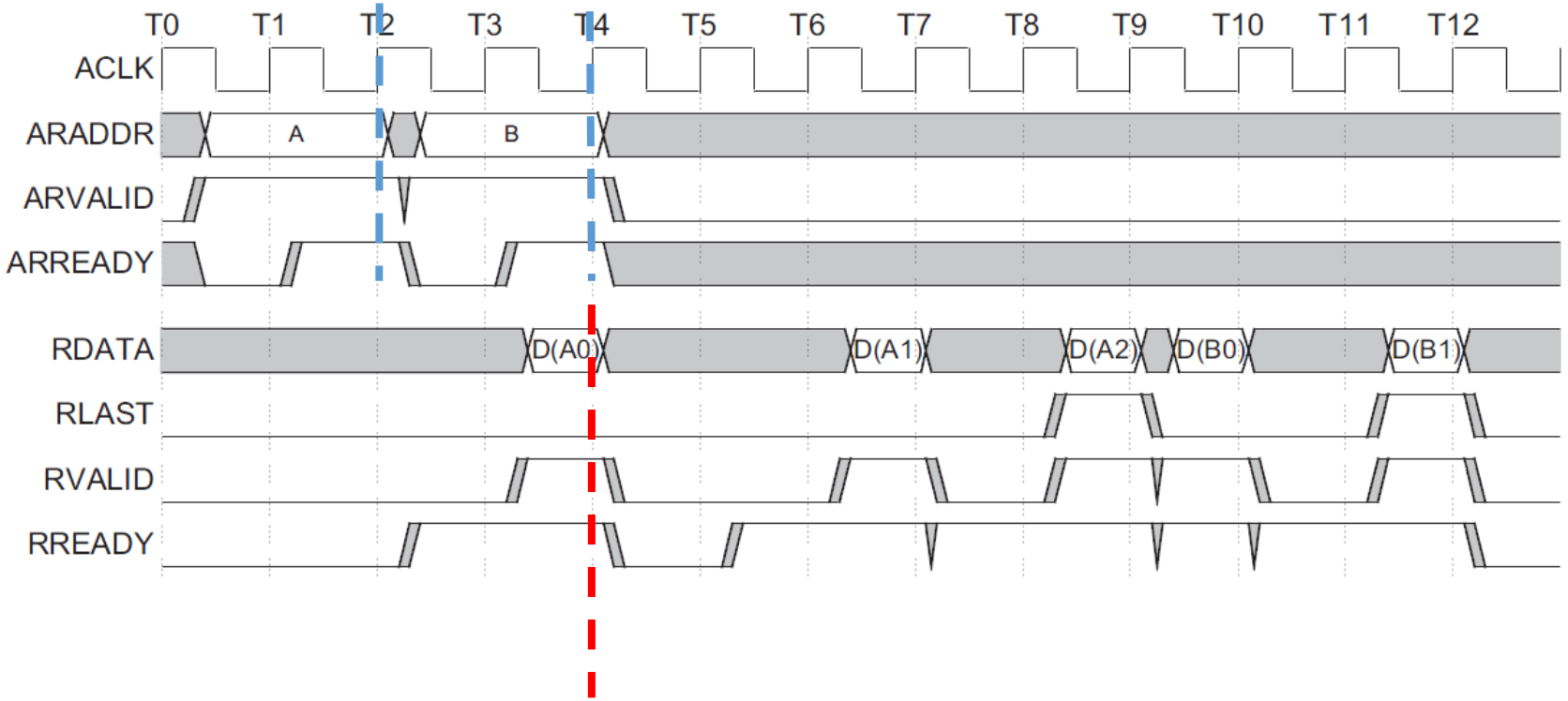




- Analysis
  - A single master w/ multiple outstanding requests can achieve >50% utilization.
  - The effects of bank interleaving by different masters are significant, up to 74%.
  - Register slice does not degrade the overall performance, i.e. utilization since multiple outstanding hides its latency.

# Overlapping Read Bursts



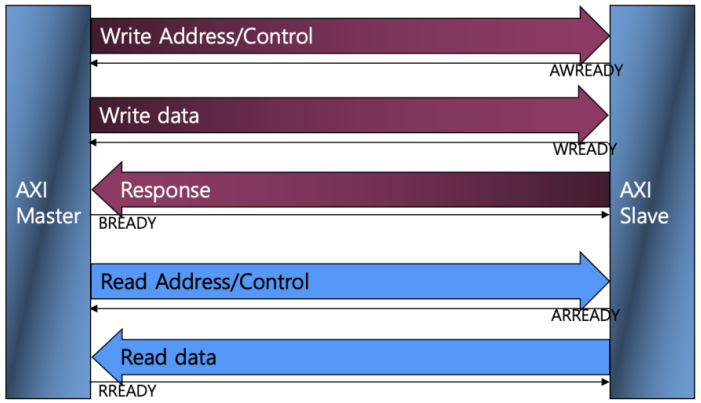



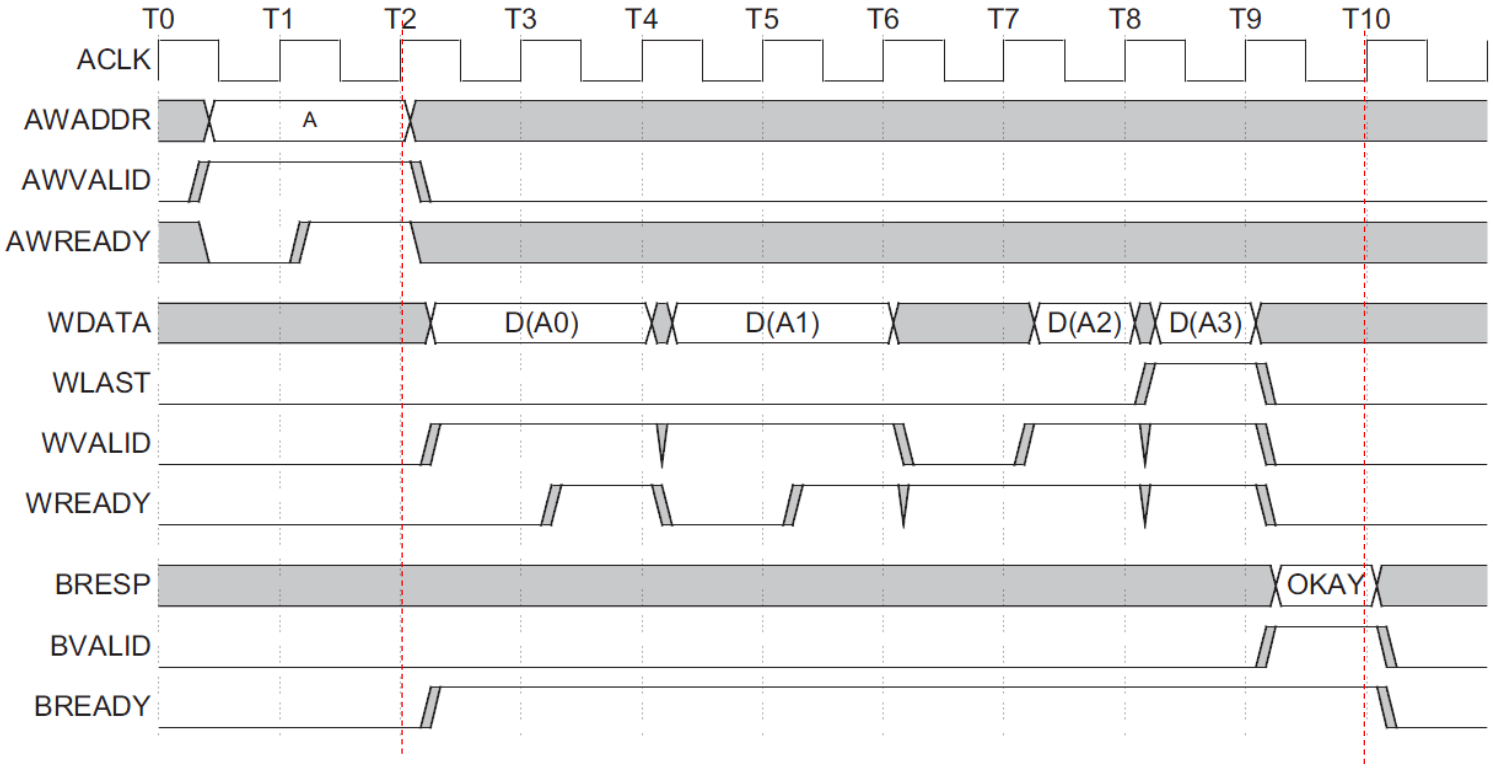




Read request A
is accepted

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

[Source: AXI Spec]

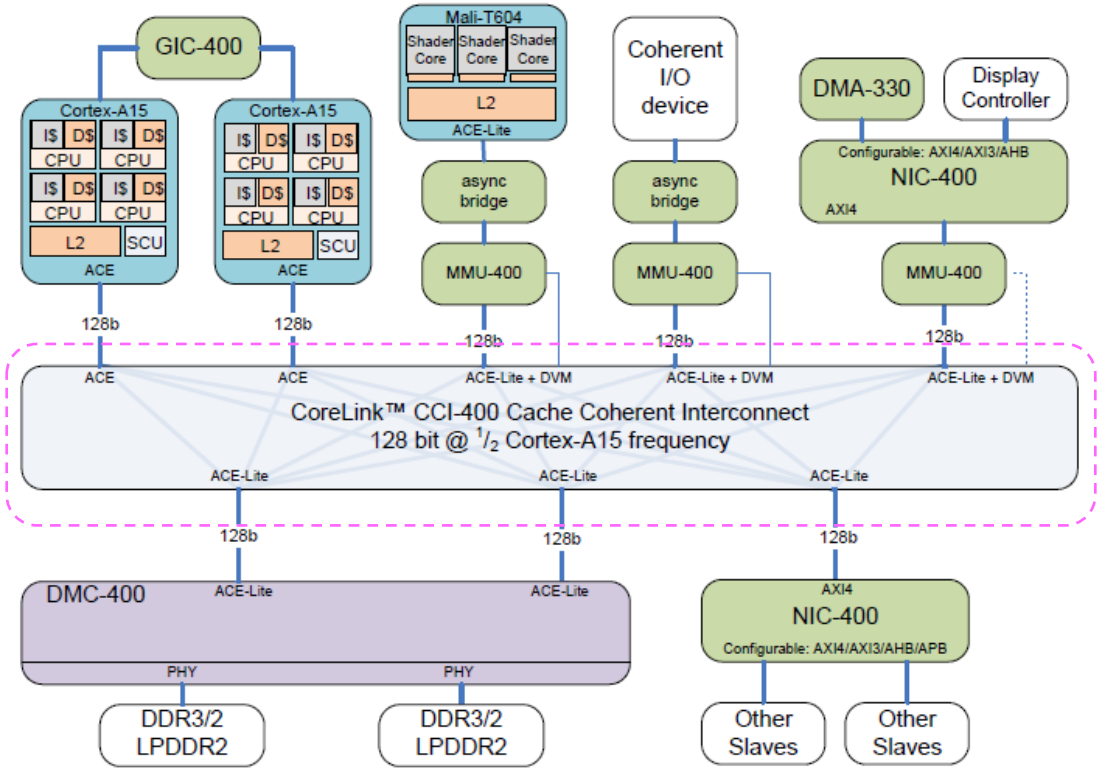
# AMBA3 Advanced eXtensable Interface (AXI) Protocol

- Multiple channels
- Narrow and wide transfer
- Single credit-based flow control: valid and ready signals
- Burst
- Split transaction to overlap request and data transfer
- Multiple outstanding requests
- Implementation issue: connectivity and arbitration

[Stevens, 2011]

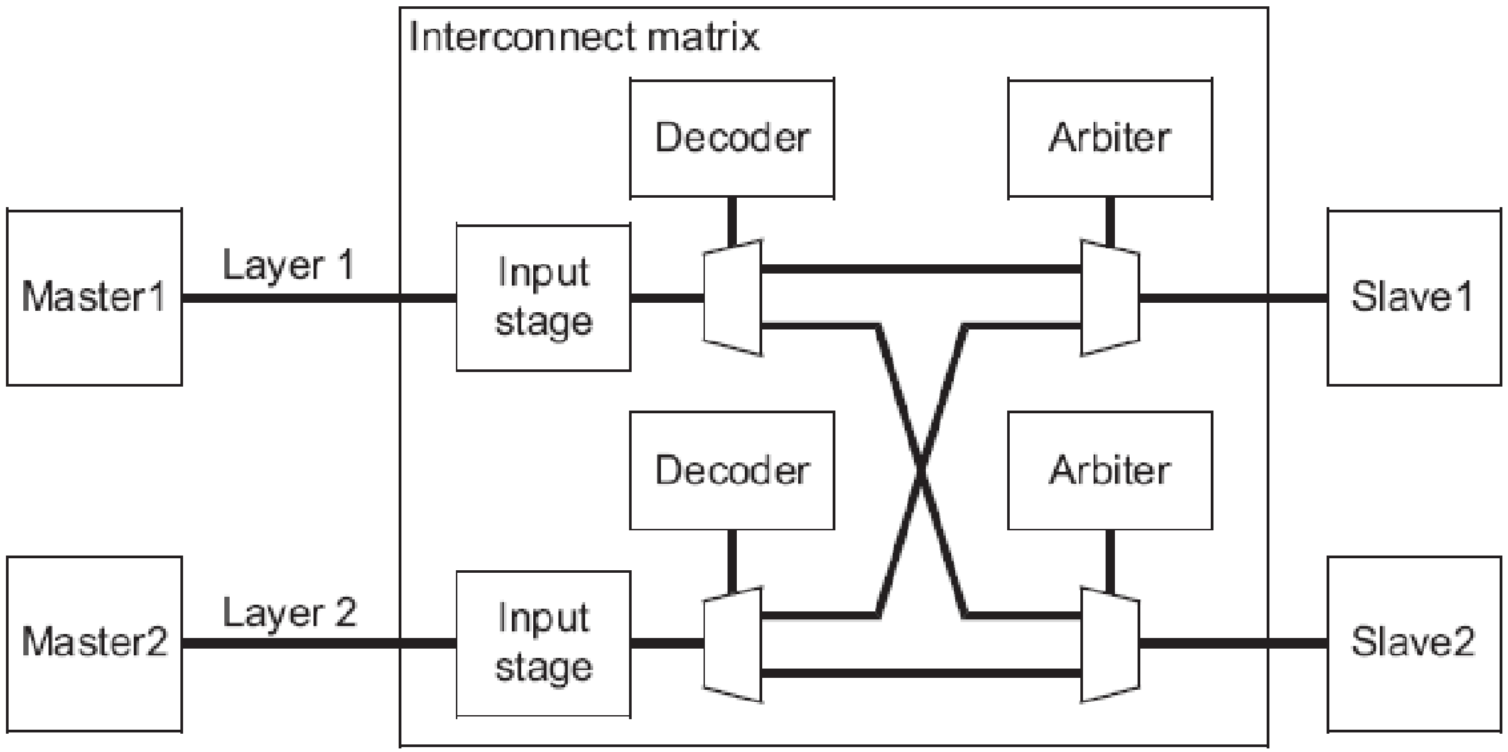
# On-Chip Bus Implementation

- AMBA3 or 4 is a protocol
- There can be different implementations



# AXI Crossbar Bus
# ARM PrimeCell PL301

- PL301: PrimeCell High Performance Matrix

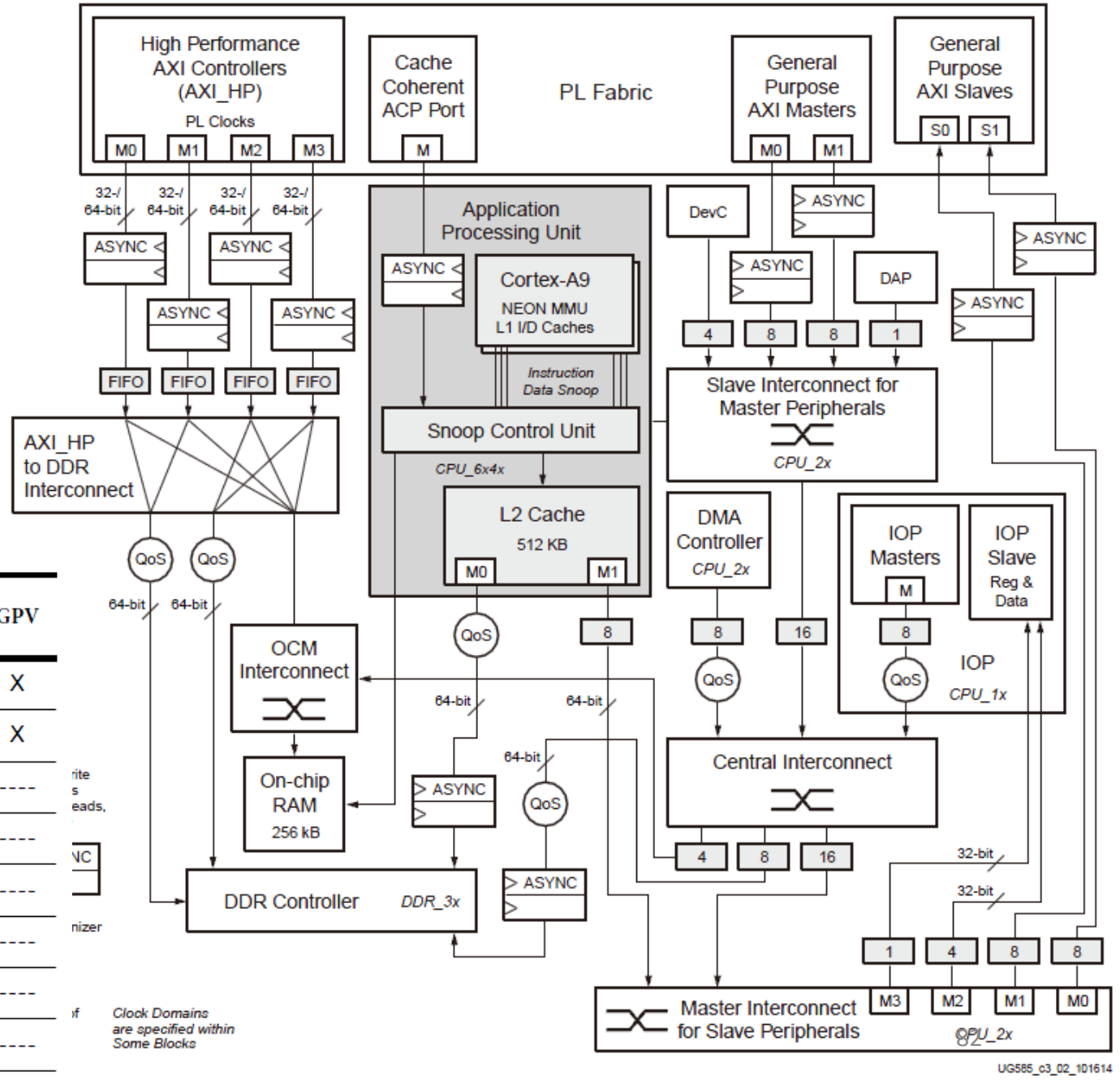


[UG585 Zynq TRM, 2016]

# Connectivity in Zynq

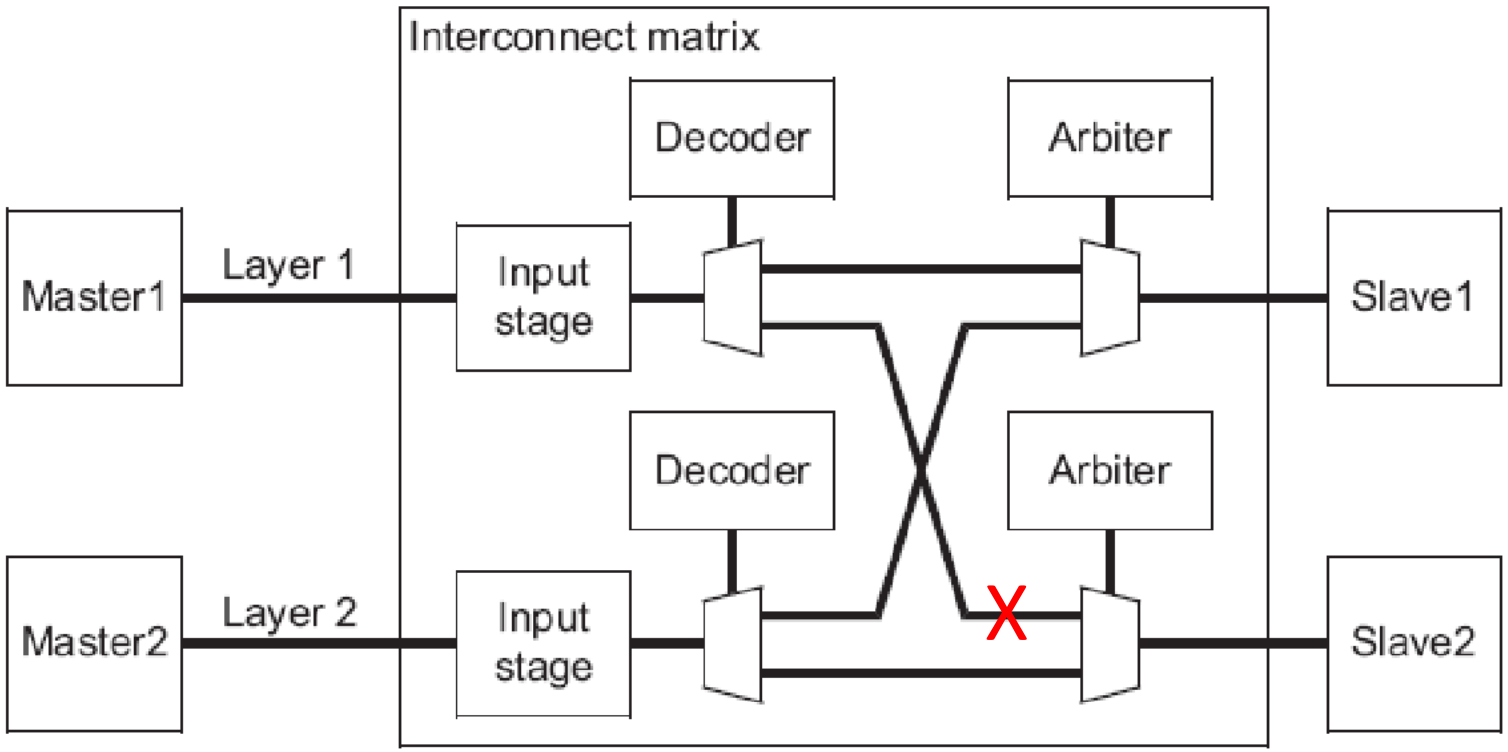
- Not a full cross-bar!

| Master \ Slave | OCM | DDR Port 0 | DDR Port 1 | DDR Port 2 | DDR Port 3 | M_AXI_GP | AHB Slaves | APB Slaves | GPV |
|---|---|---|---|---|---|---|---|---|---|
| CPUs | X | X | ---- | ---- | ---- | X | X | X | X |
| AXI_ACP | X | X | ---- | ---- | ---- | X | X | X | X |
| AXI_HP{0,1} | X | ---- | ---- | ---- | X | ---- | ---- | ---- | ---- |
| AXI_HP{2,3} | X | ---- | ---- | X | ---- | ---- | ---- | ---- | ---- |
| S_AXI_GP{0,1} | X | ---- | X | ---- | ---- | X | X | X | ---- |
| DMA Controller | X | ---- | X | ---- | ---- | X | X | X | ---- |
| AHB Masters | X | ---- | X | ---- | ---- | X | X | X | ---- |
| DevC, DAP | X | ---- | X | ---- | ---- | X | X | X | ---- |

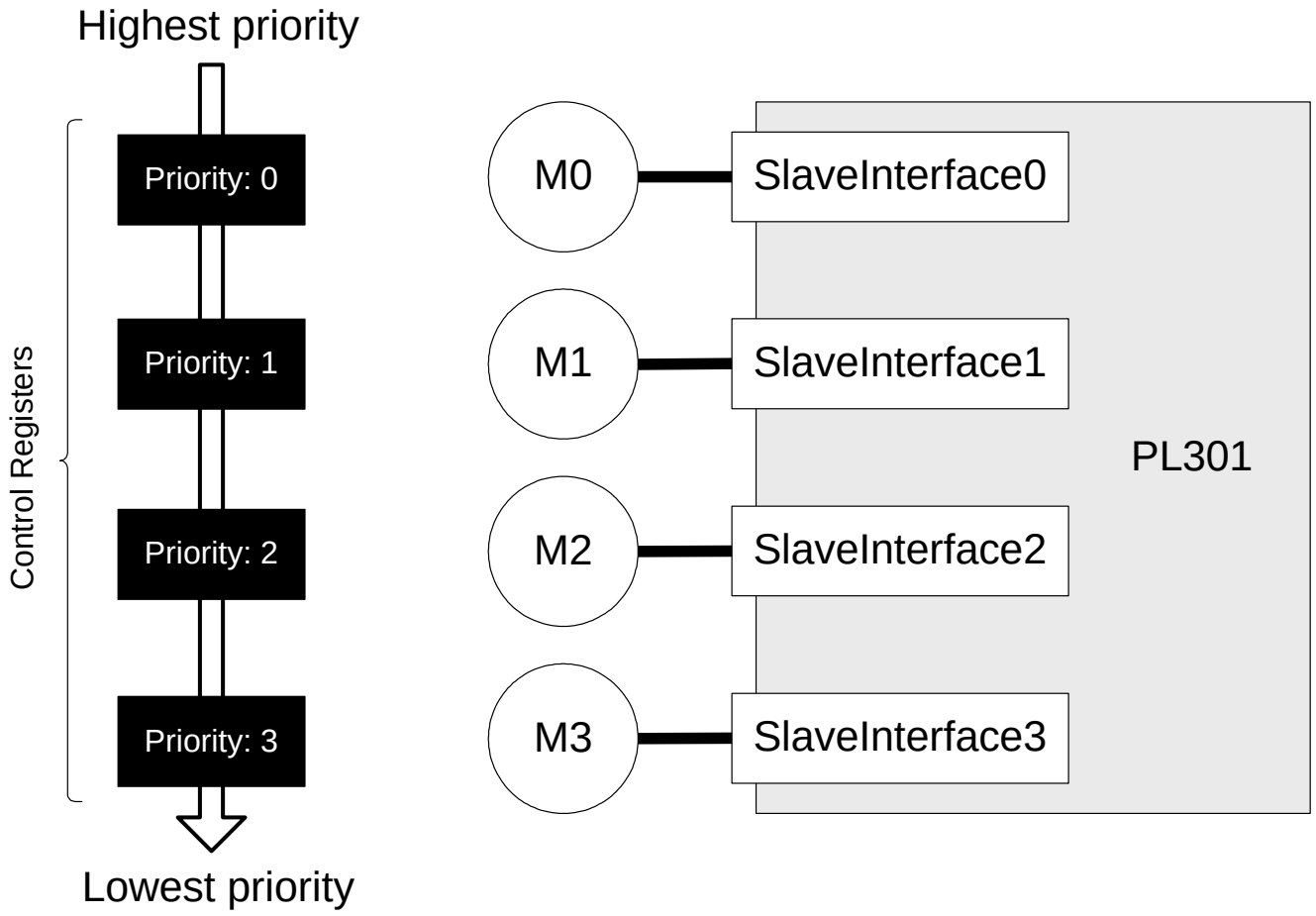


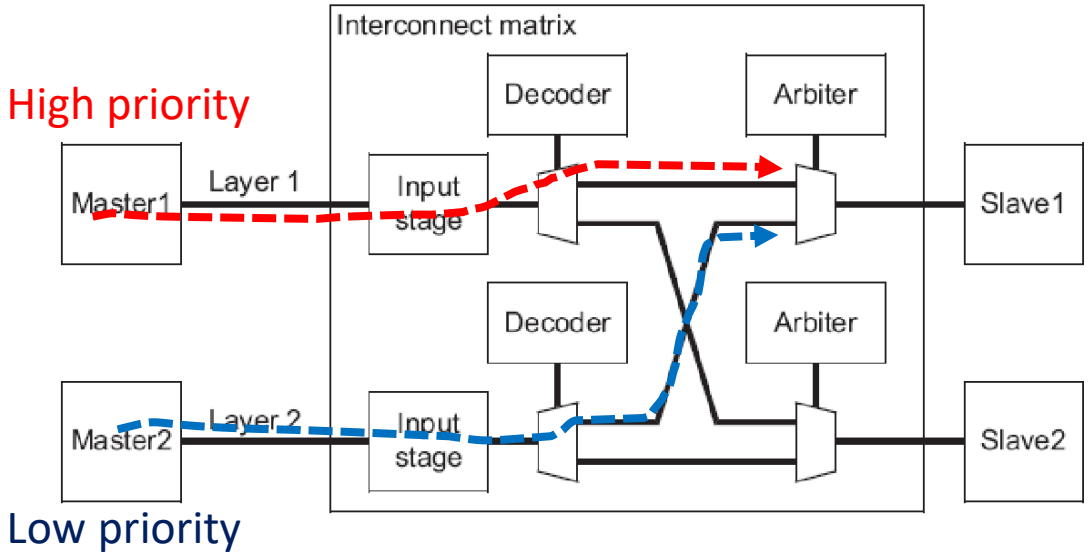
# AXI Crossbar Bus
# ARM PrimeCell PL301

- PL301: PrimeCell High Performance Matrix



# Arbitration Scheme: Fixed Priority



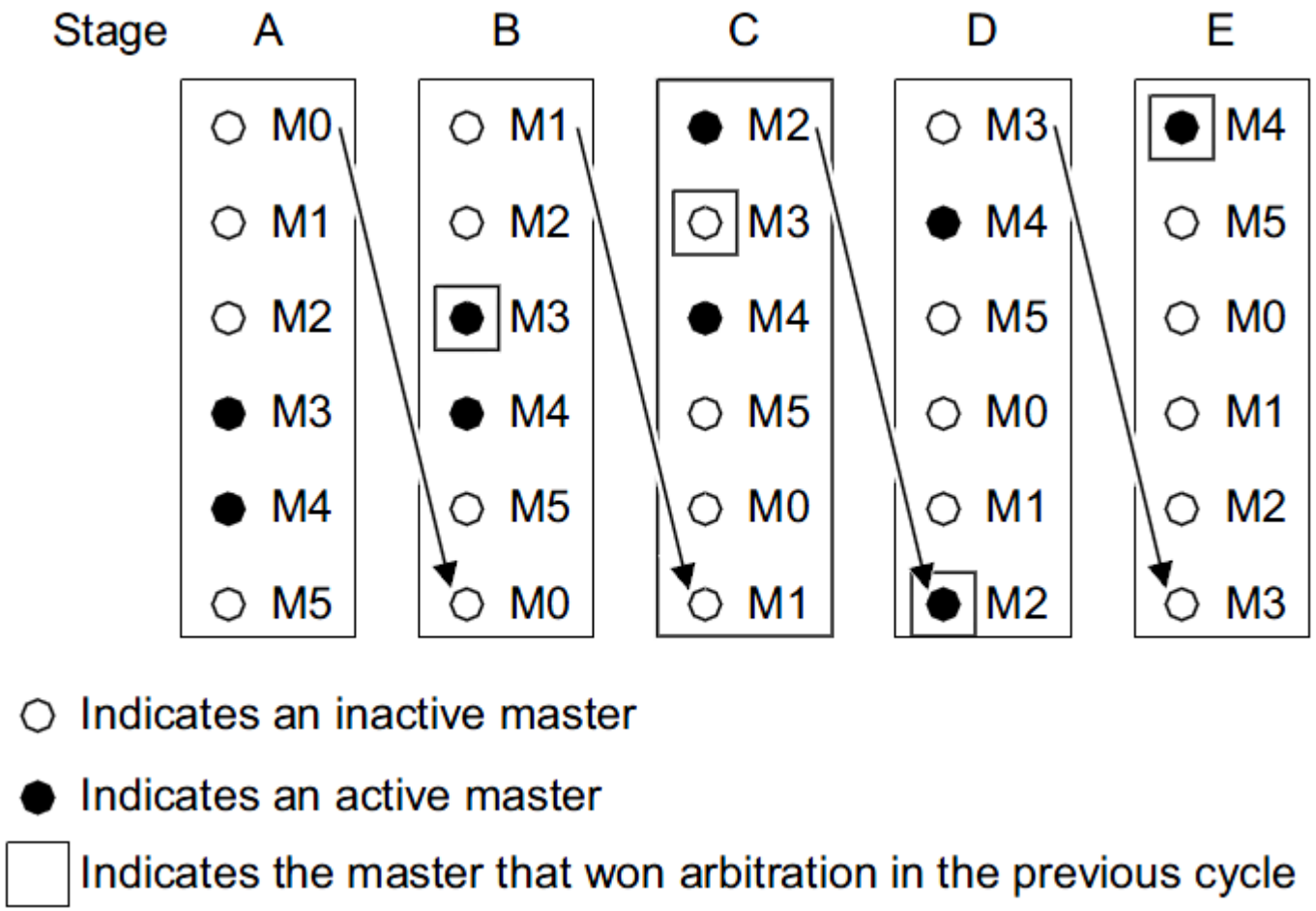



# Starvation Problem of Fixed Priority Scheme

- Starvation
  - Requests from low priority masters can be delayed due to high priority requests
  - In the worst case, low priority masters cannot send their requests to the slave, which will finally make the system fail
- How to resolve the starvation problem?
  - Fair arbitration?

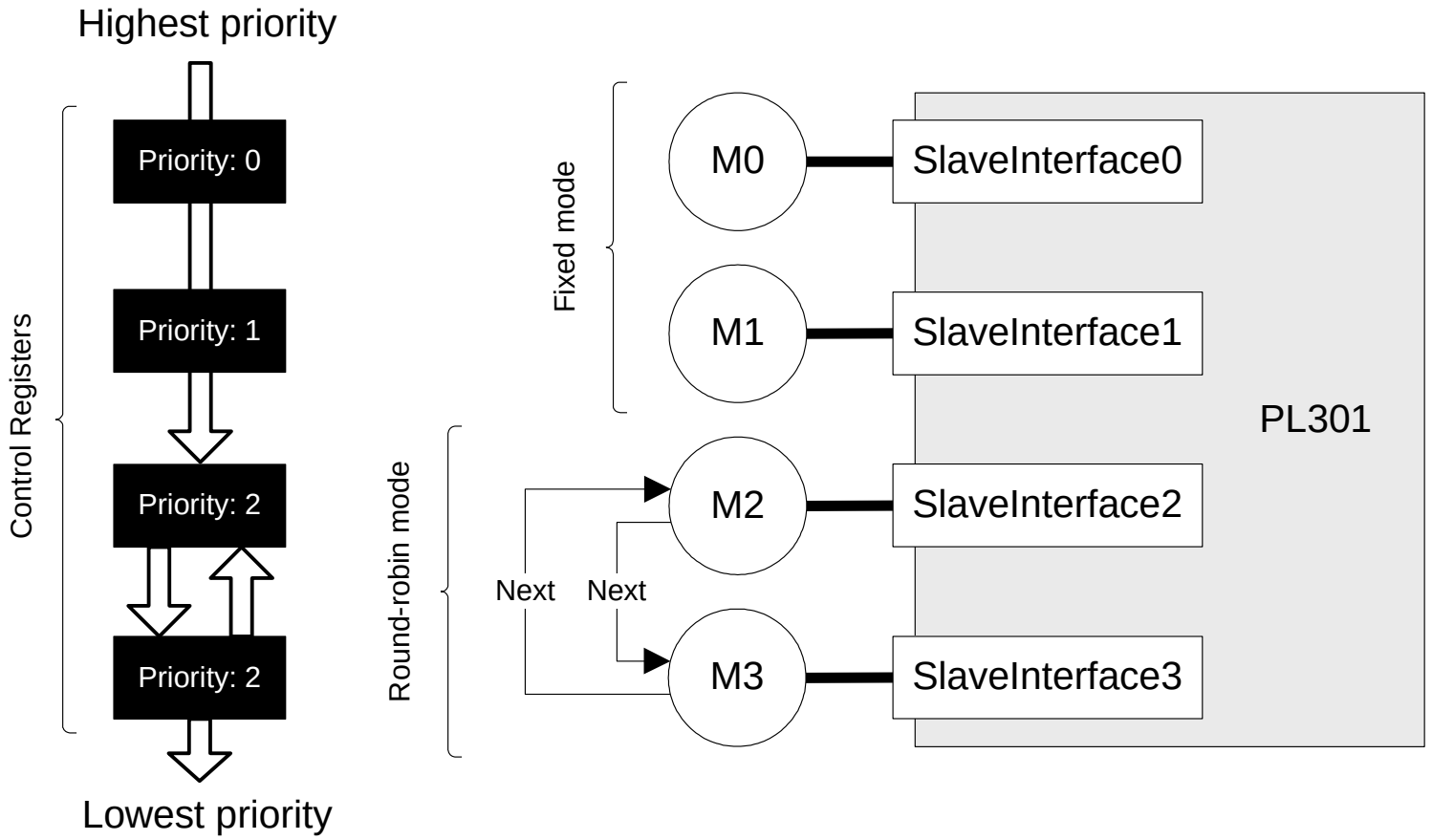
[Source: PL301 TS]

# Arbitration Scheme: Round Robin



# Arbitration Scheme: Hybrid

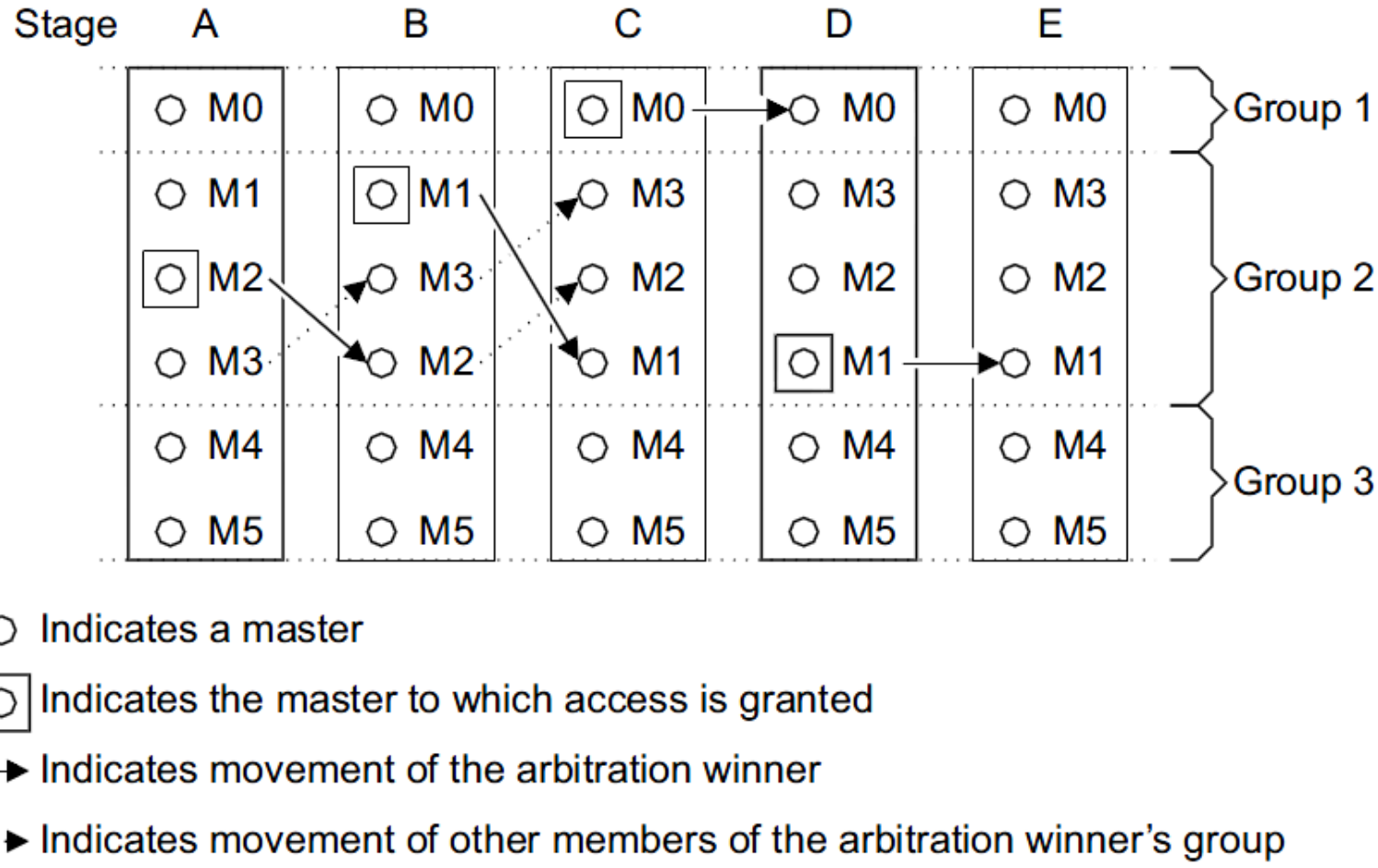
- Combination of round robin and fixed priority



[Source: PL301 TS]

# Fair Arbitration Scheme

- LRG (least recently granted) scheme



[Source: AXI Spec]

# AMBA3 Advanced eXtensable Interface (AXI) Protocol

- Multiple channels
- Narrow and wide transfer
- Single credit-based flow control: valid and ready signals
- Burst
- Split transaction to overlap request and data transfer
- Multiple outstanding requests
- Implementation issue: connectivity and arbitration

[Stevens, 2011]

# Reading Data from Memory (in Cache Miss)

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

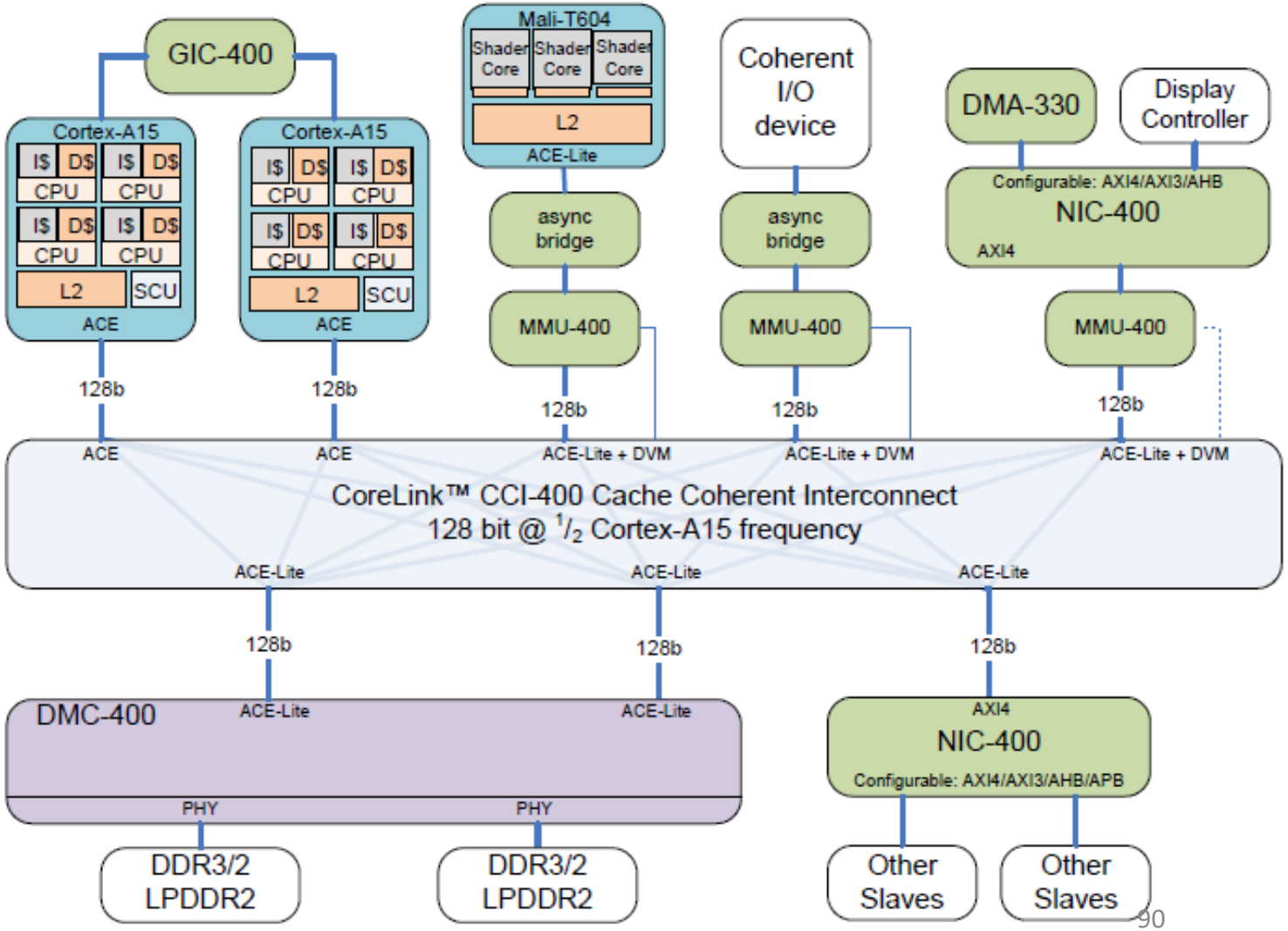
VA(x) → PA(x) translation on TLB

CPU sends to memory, via the interconnect, a read request of PA(x)

The read request arrives at memory controller

Memory controller reads data@PA(x) from DRAM

Memory controller sends the data to CPU via the interconnect



# Weekly Lecture / Lab Schedule

- W1 (March 6) Class introduction / (March 8) Orientation & team formation
- W2 13 Verilog 1 Combinational circuits / 15 Verilog 1 (tool installation, adder & multiplier combinational logic)
- W3 20 Verilog 2 Sequential circuits / 22 Verilog 2 (memory i/o, FSM sequential logic)
- W4 27 AI application introduction 1, Amaranth introduction 1 / 29 Amaranth (tool installation, MAC, adder tree)
- W5 4/3 AI application introduction 2, Amaranth introduction 2 (memory i/o, FSM sequential logic) / 5 Amaranth (PE)
- W6 10 AI application introduction 3, Neural network accelerator 1 / 12 Amaranth (stacked PEs)
- W7 17 Neural network accelerator 2 / 19 Amaranth (stacked PEs)
- W8 24 **Mid-term exam** / 26 Amaranth (stacked PEs)
- W9 5/1 Reading data from memory 1 (VA2PA, interconnect) / 3 Convolution lowering
- W10 8 Reading data from memory 2 (DRAM main memory), Compressing networks 1 (pruning) / 10 Tiling
- W11 15 Compressing networks 2 (pruning, low precision) / 17 PyTorch model – Amaranth simulator communication
- W12 22 Zero-skipping & low-precision hardware accelerator / 24 Quantization, project introduction
- W13 29 Invited talks (commercial solutions: Rebellions, Furiosa AI) / 31 Homework Q&A
- W14 6/5 **Final exam** / 7 Project Q&A
- W15 12 Claim & Project Q&A / 14 Project submission

# Summary of Steps 1 & 2 in Reading from Memory

[Stevens, 2011]

# Step 1: Virtual to Physical Address Translation before Accessing Memory Subsystem

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

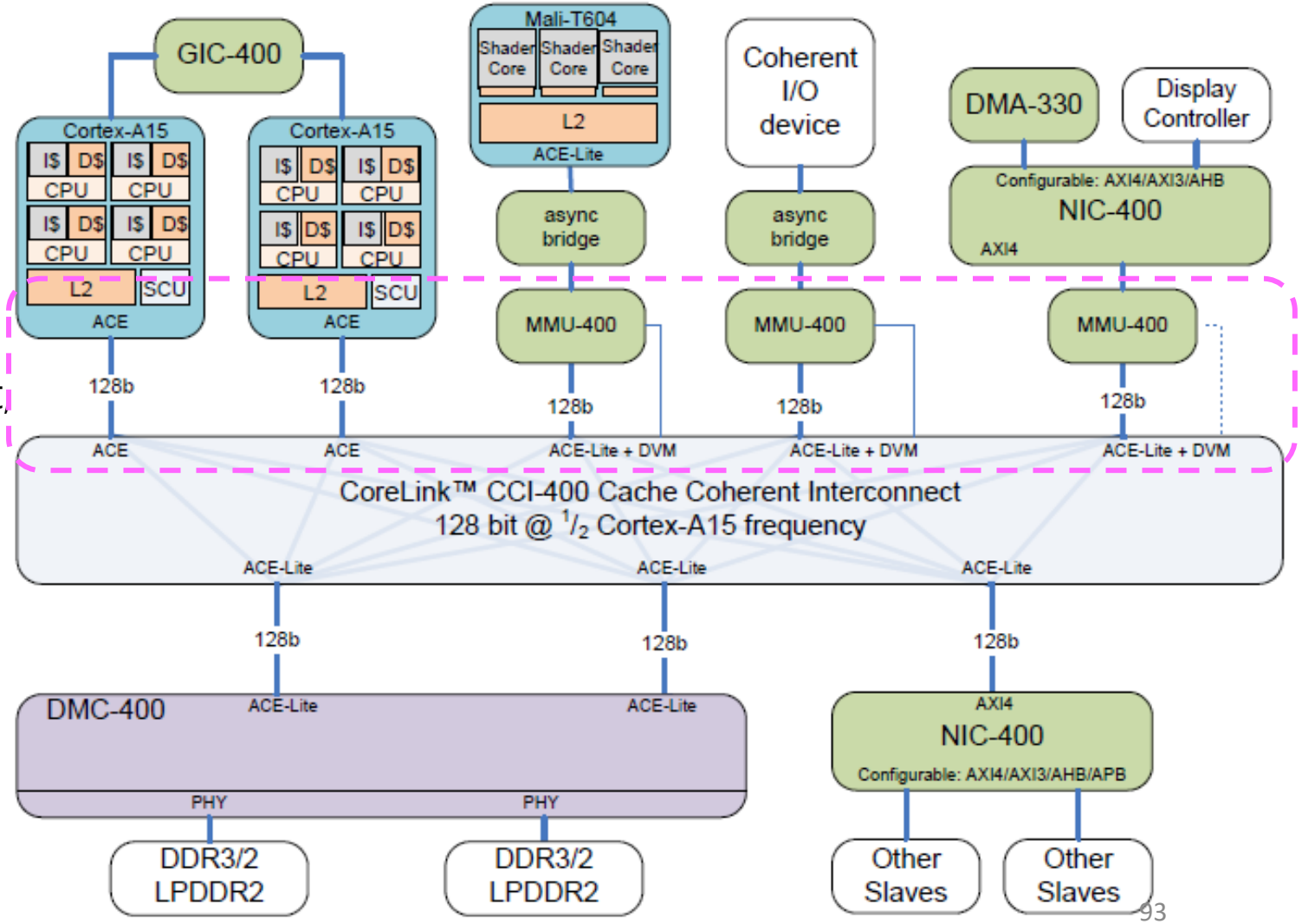
VA(x) → PA(x) translation on TLB

CPU sends to memory, via the interconnect, a read request of PA(x)

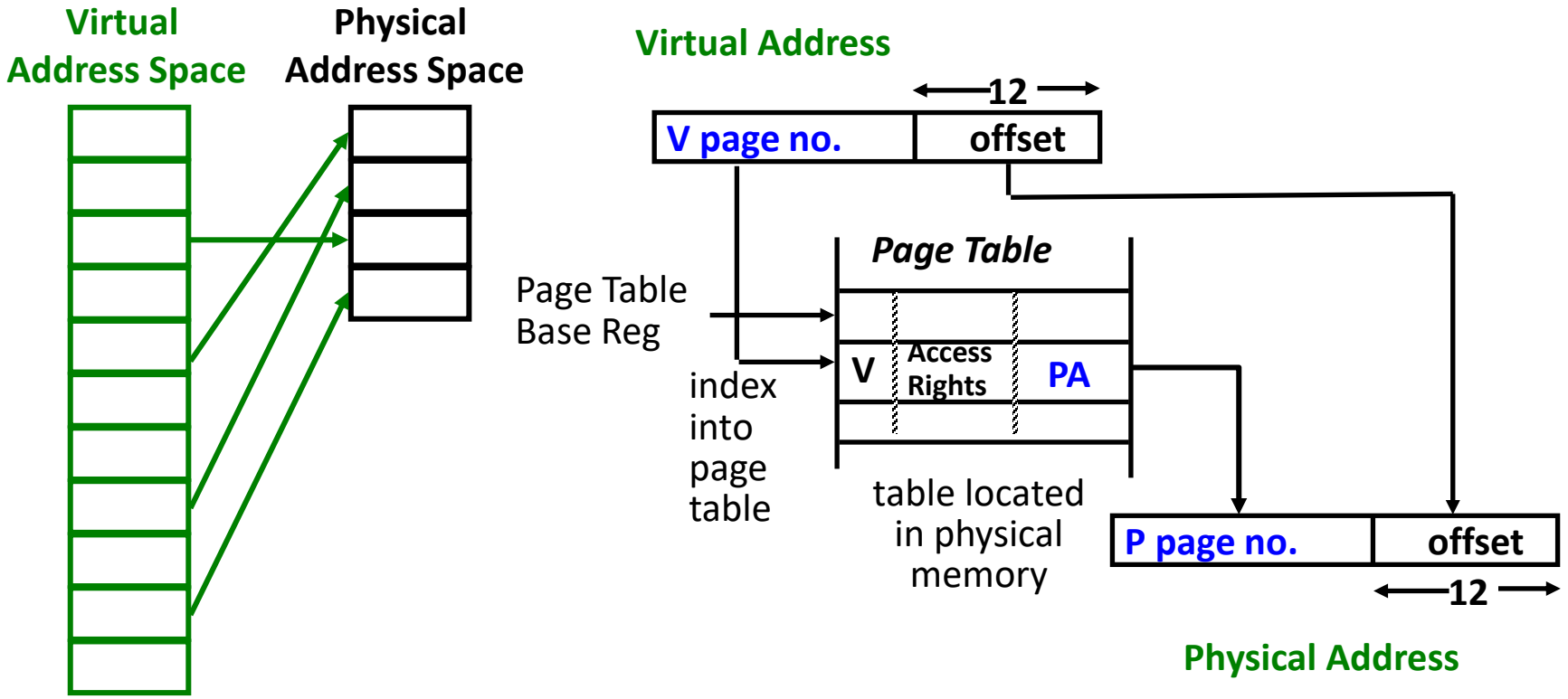
The read request arrives at memory controller

Memory controller reads data@PA(x) from DRAM

Memory controller sends the data to CPU via the interconnect



# Virtual Memory: An Introduction




- Page table maps virtual page numbers to physical frames
  - "PTE" = Page Table Entry (arrow from virtual to physical address spa ce)

[Stevens, 2011]

# Step 2: Sending Read Request via Interconnect

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

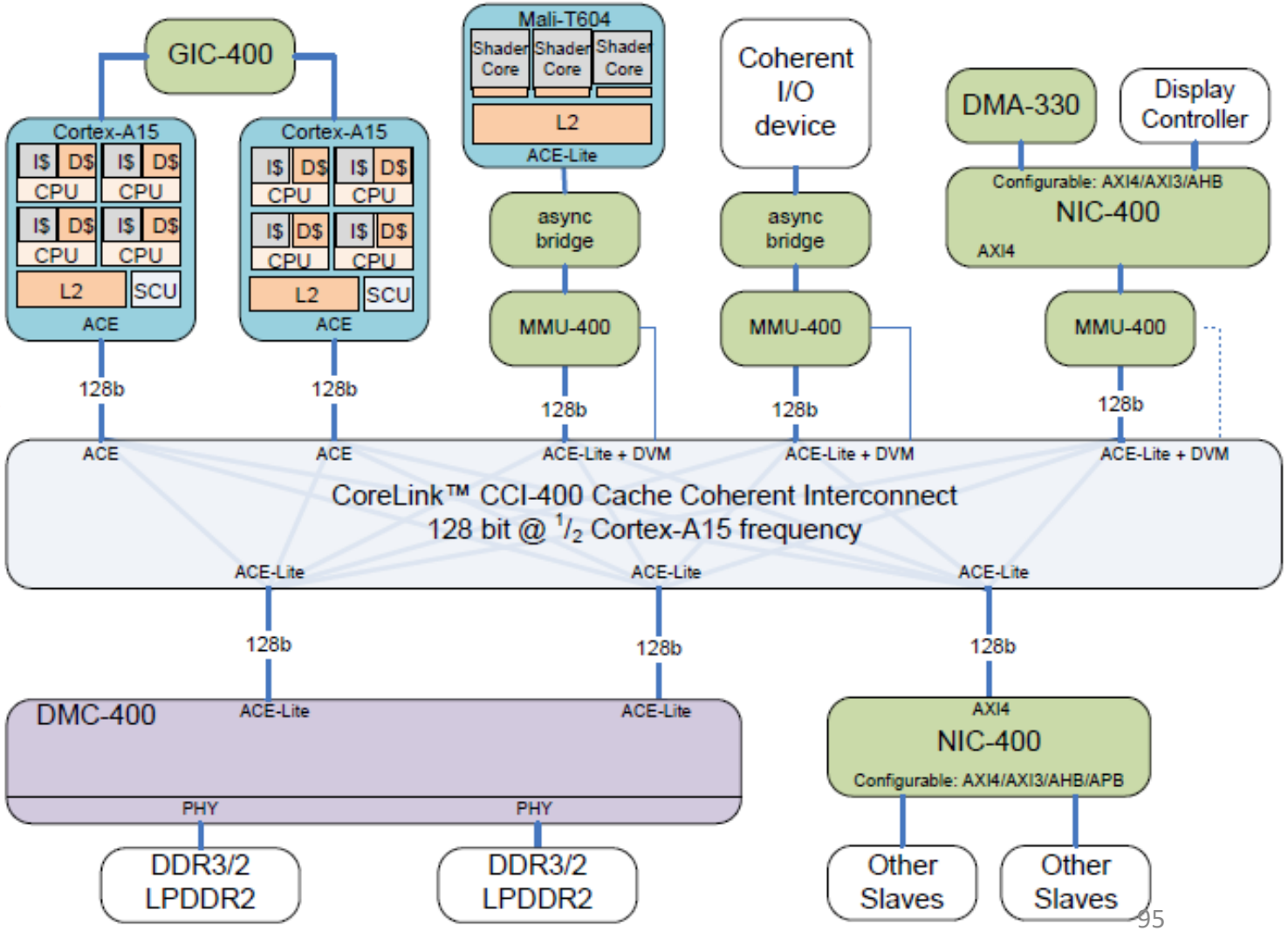
VA(x) → PA(x) translation on TLB

CPU sends to memory, via the interconnect, a read request of PA(x)

The read request arrives at memory controller

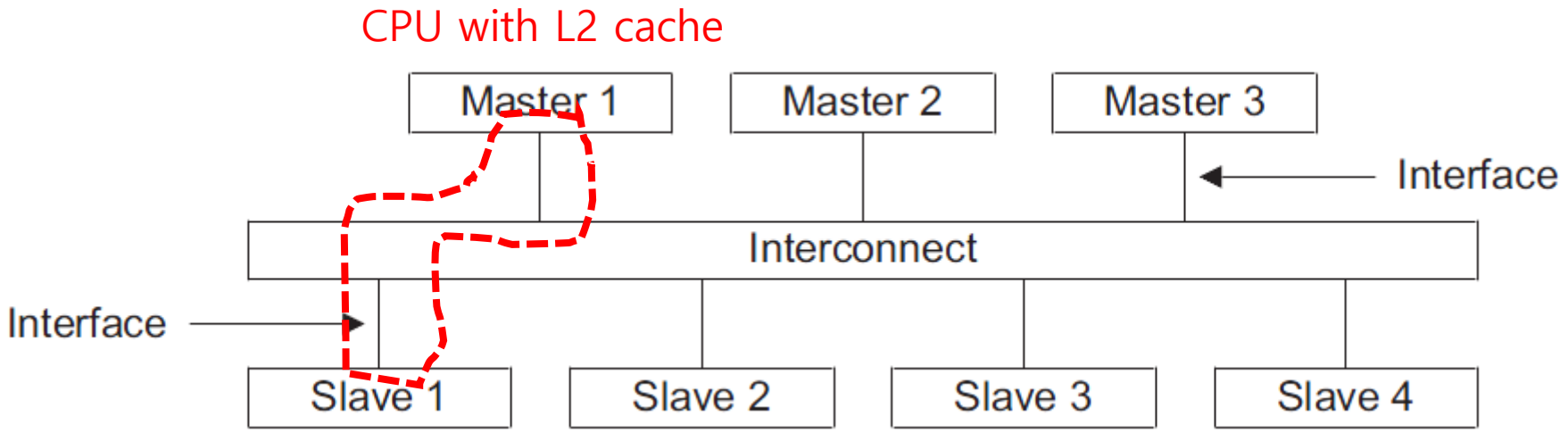
Memory controller reads data@PA(x) from DRAM

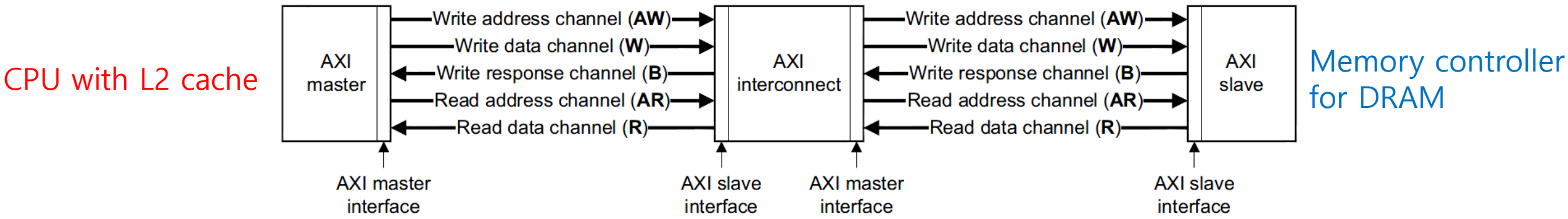
Memory controller sends the data to CPU via the interconnect



[Source: AXI Spec]

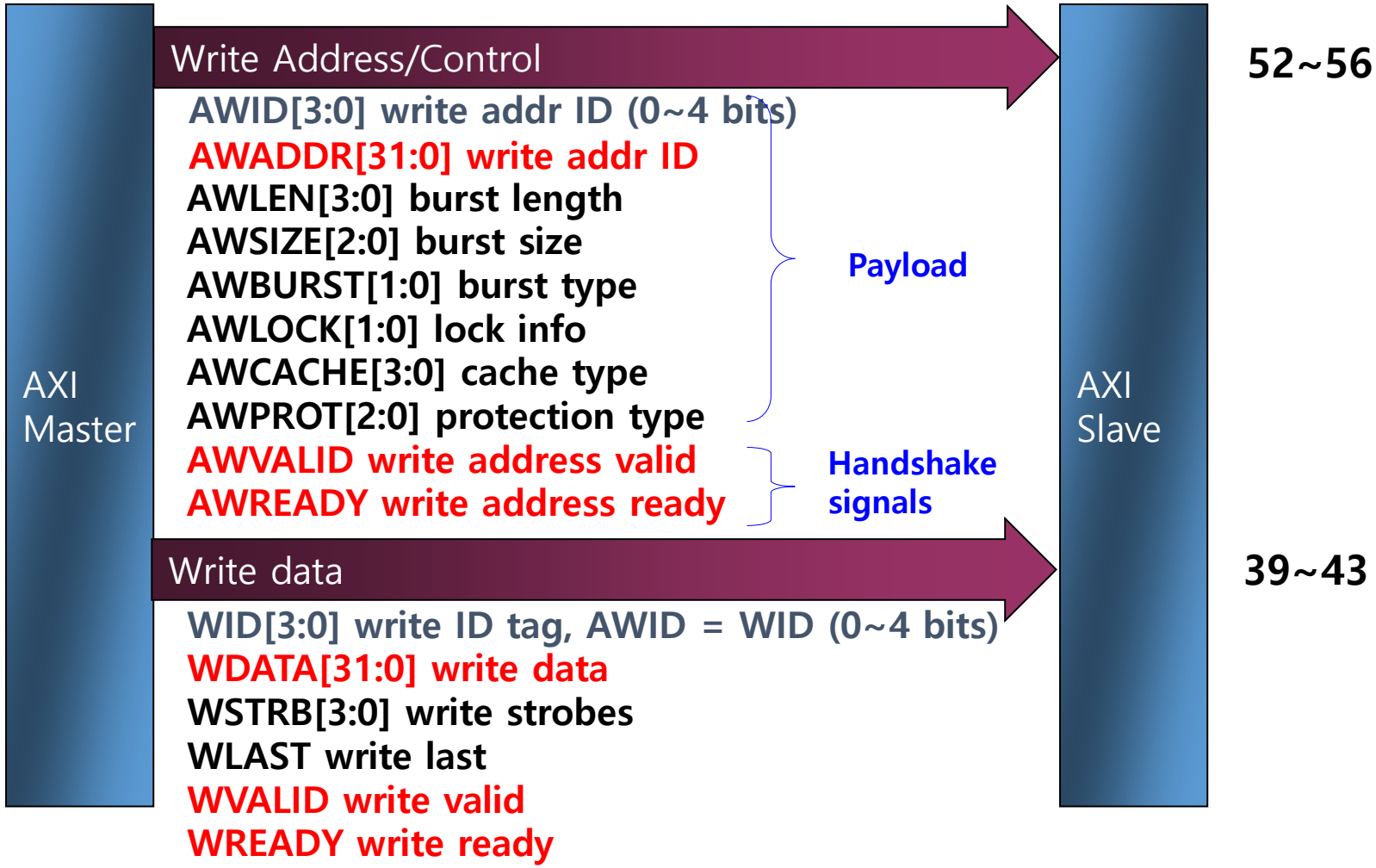
# Interconnect, Interface & Channel



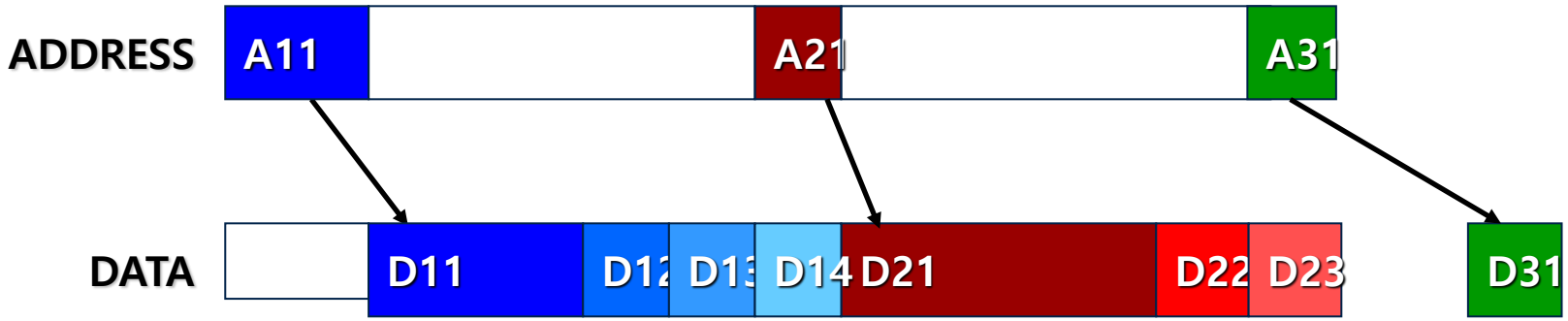



# Wire Counts

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



# One Address for Burst



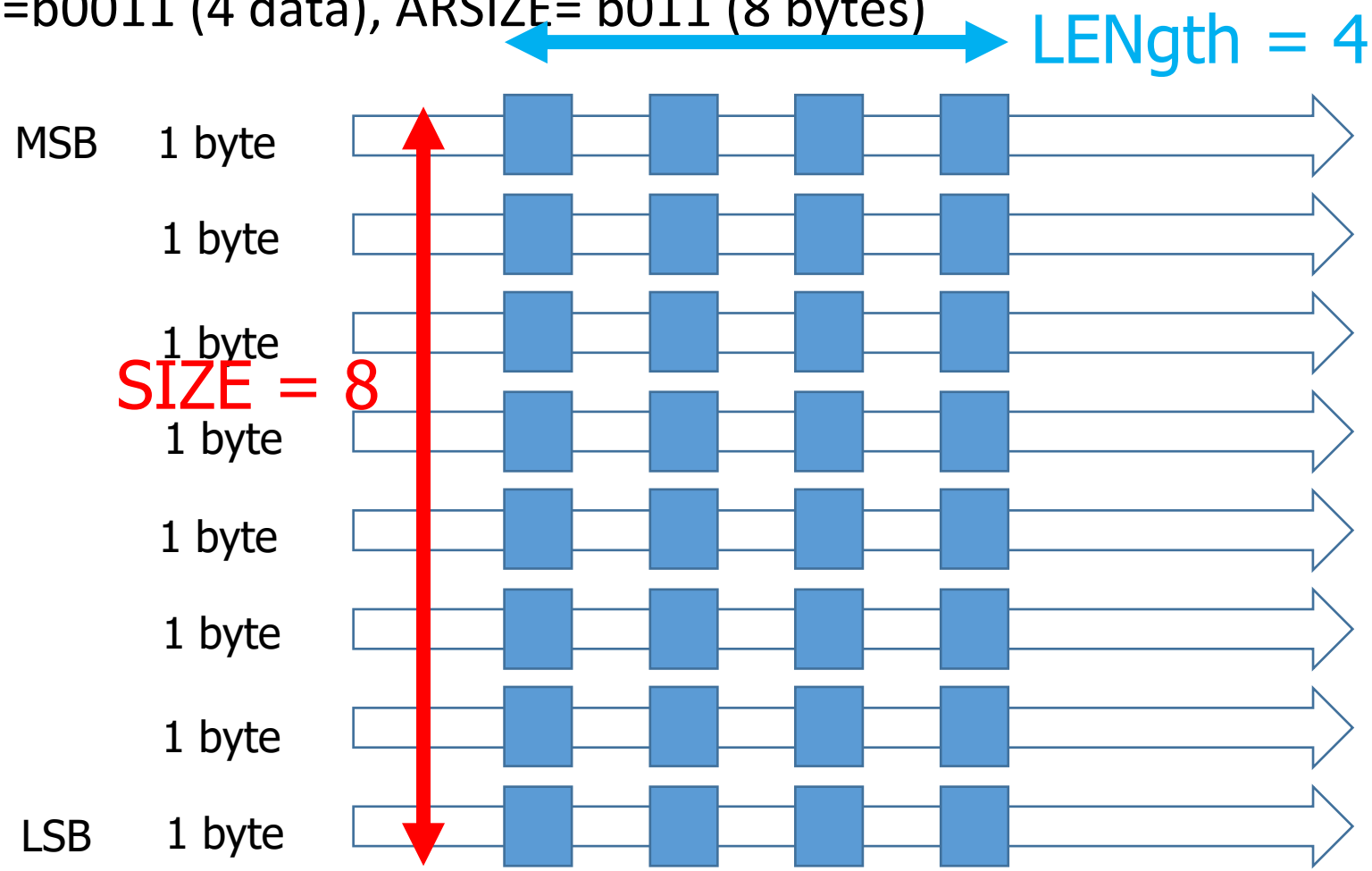

- Separation of address and data channel
  - Master provides the start address of burst
  - Slave needs to generate the remaining addresses based on burst type (FIXED, INCR, WRAP)

# (Burst) Length and (Beat) Size

- ARLEN=b0011 (4 data), ARSIZE= b011 (8 bytes)

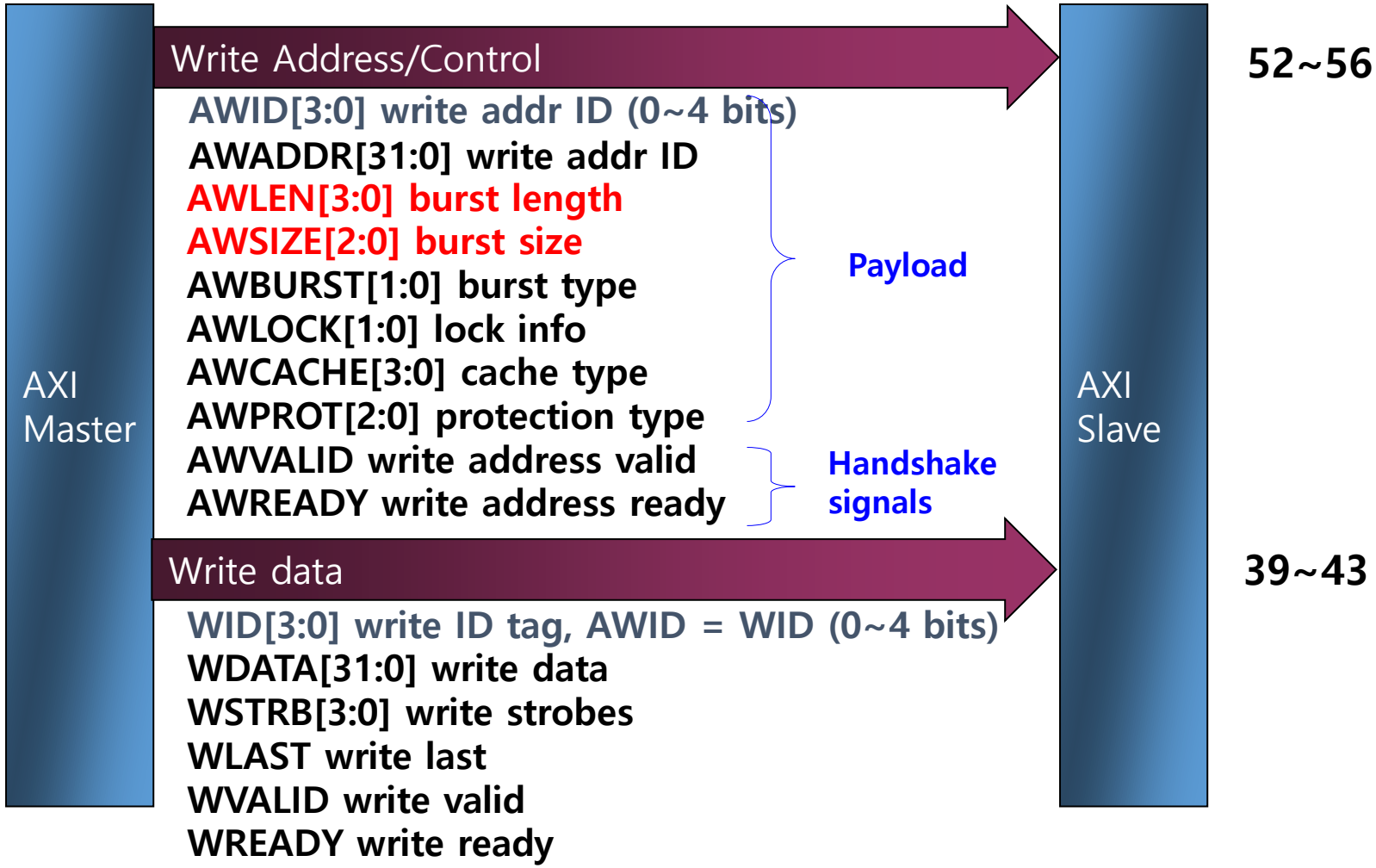
Size:
# bytes / cycle

Length:
# clock cycles
(a.k.a beats)

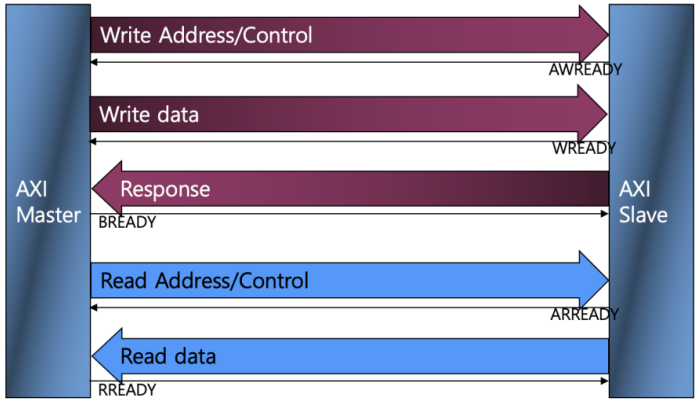


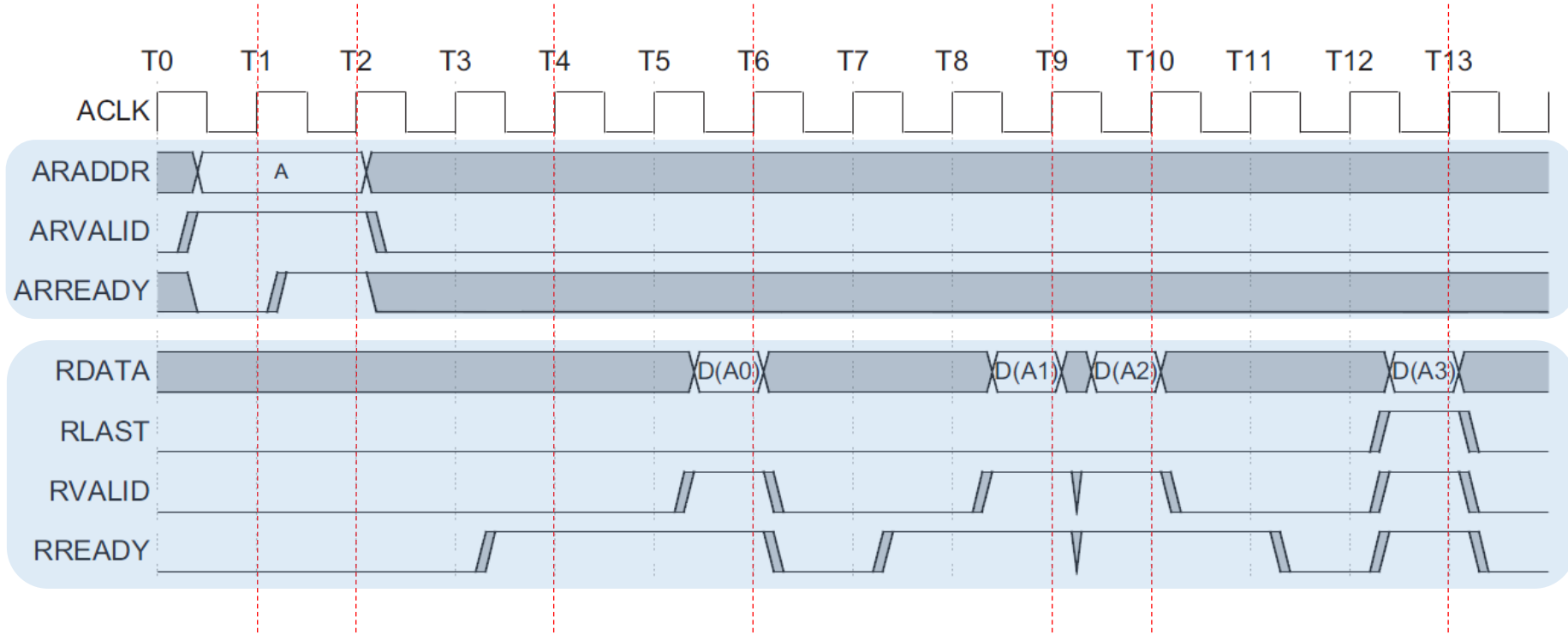
# Wire Counts

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



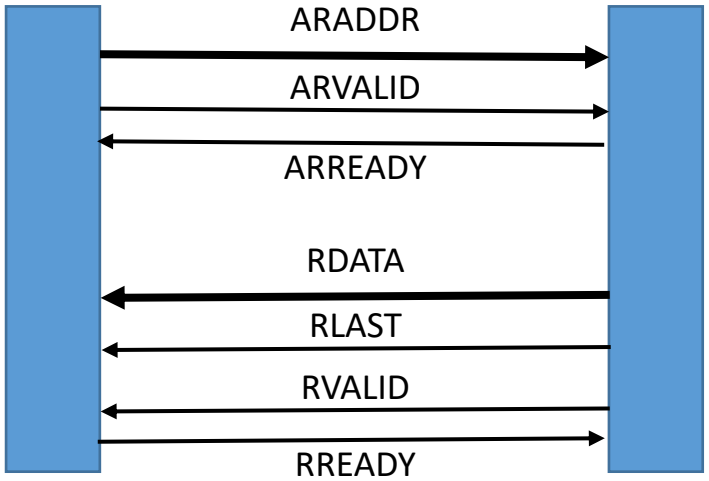
# Read Burst Operation




Master
e.g., CPU

Slave
e.g., Bus




Read request is initiated

Data read is ready

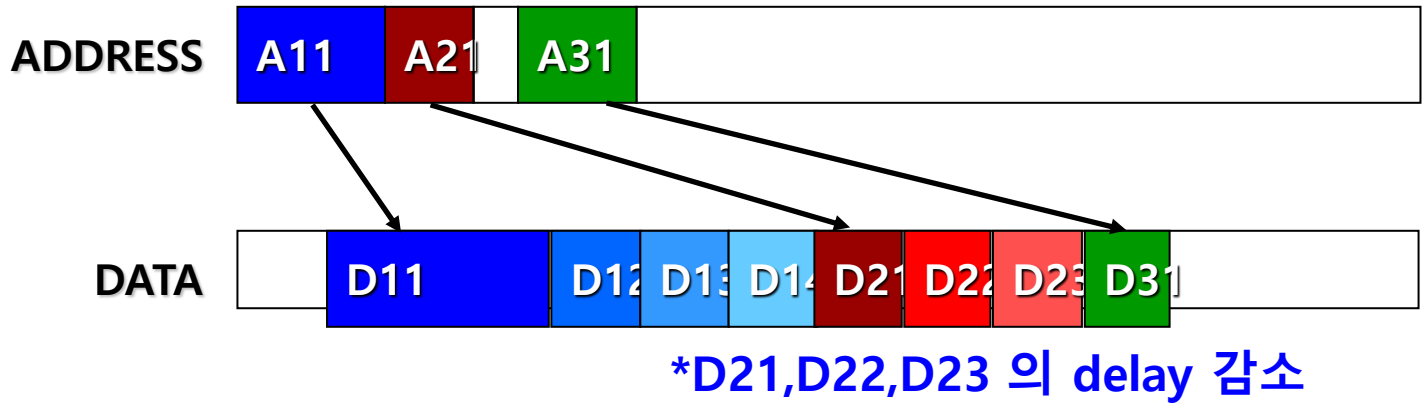
1st data is transferred

The last data is transferred

Read request is accepted

**Note: data transfer only when valid = ready = 1**

# Benefit of Split Transaction: Multiple Outstanding Requests

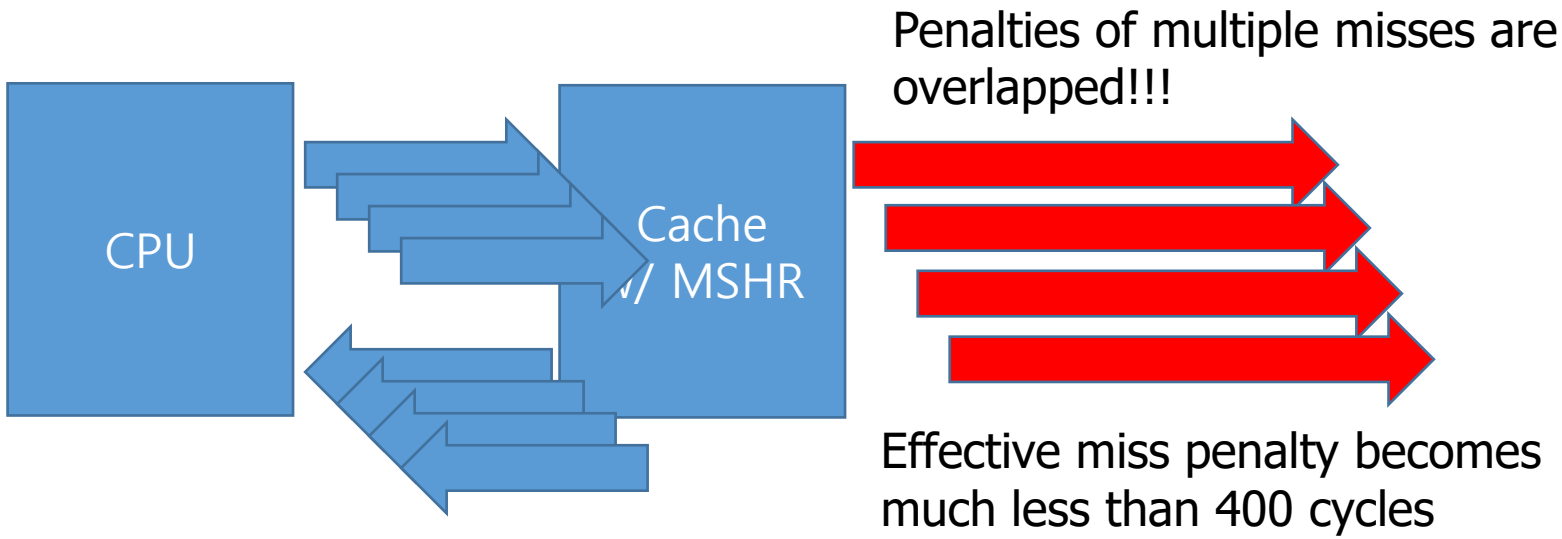



- Parameters for multiple outstanding requests
  - Master I/F: Issuing capability → master가 generation 할 수 있는 outstanding request의 개수
  - Slave I/F: Acceptance capability → slave가 받아 들일 수 있는 outstanding request의 개수

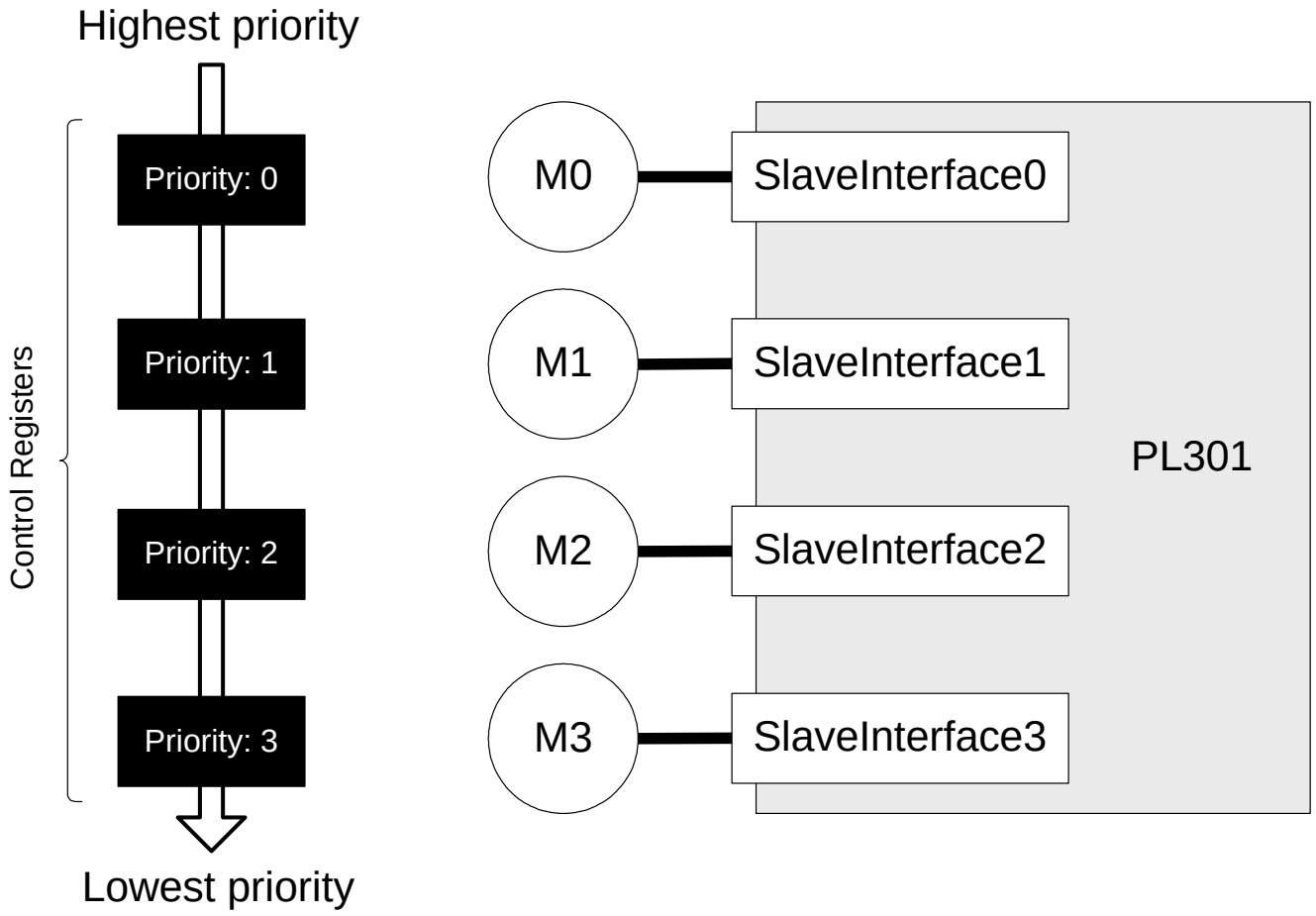
[Source: J. Kubiatowicz, 2000]

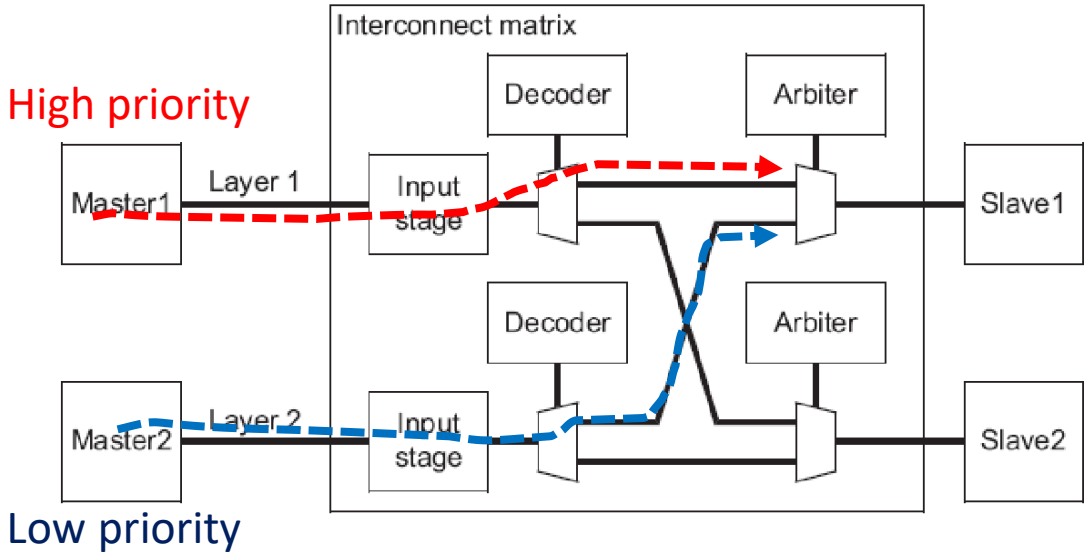
# Non-blocking Caches to Reduce Stalls on Misses

- "*hit under multiple miss*" or "*miss under miss*" may further lower the eff ective miss penalty by overlapping multiple misses



# Arbitration Scheme: Fixed Priority



[Stevens, 2011]

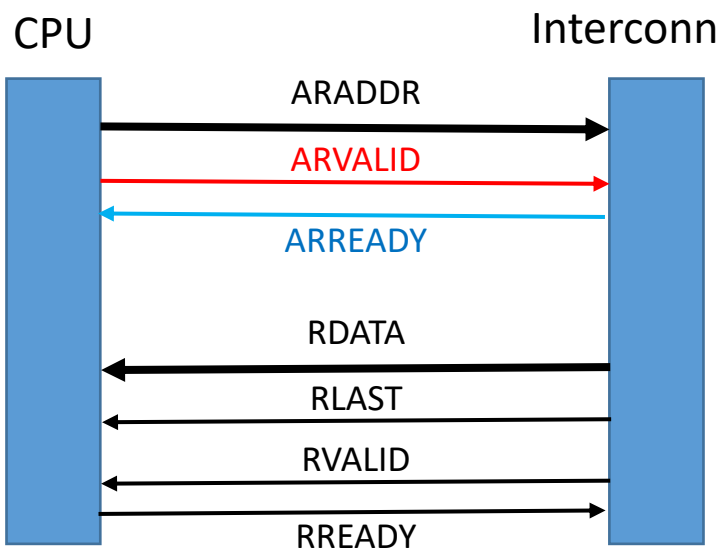
# Step 2: Sending Read Request via Interconnect

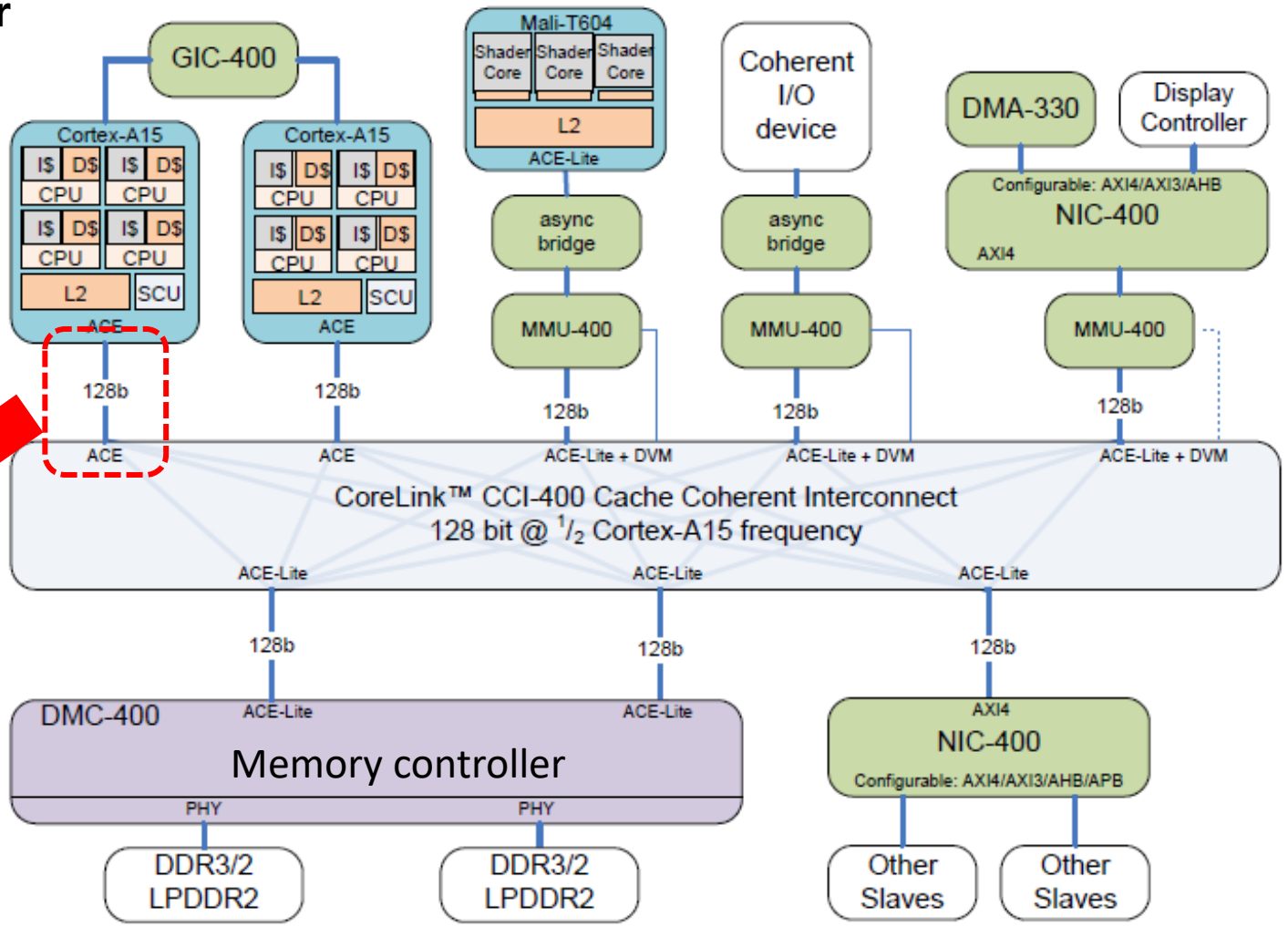
**Sending a read request to memory controller**

VA to PA translation on TLB

Send a read request on read address (AR) channel, i.e., ARVALID = '1'

Interconnect receives it, i.e., ARREADY = '1'



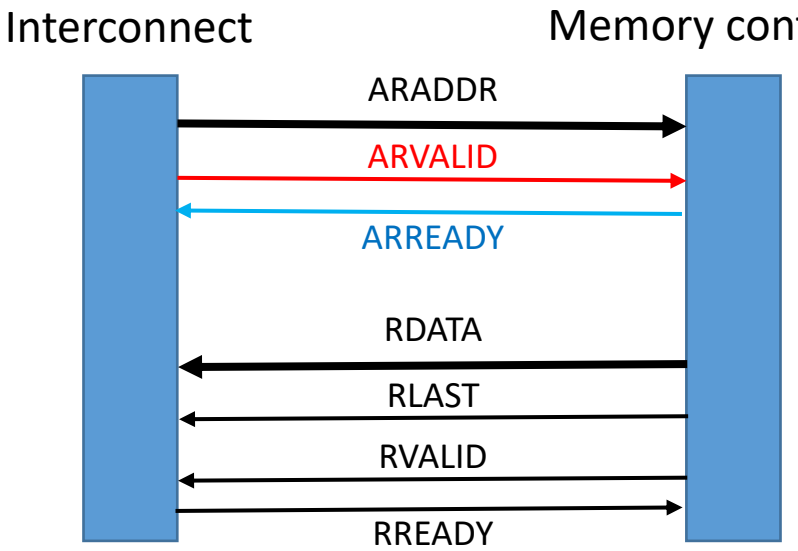
[Stevens, 2011]

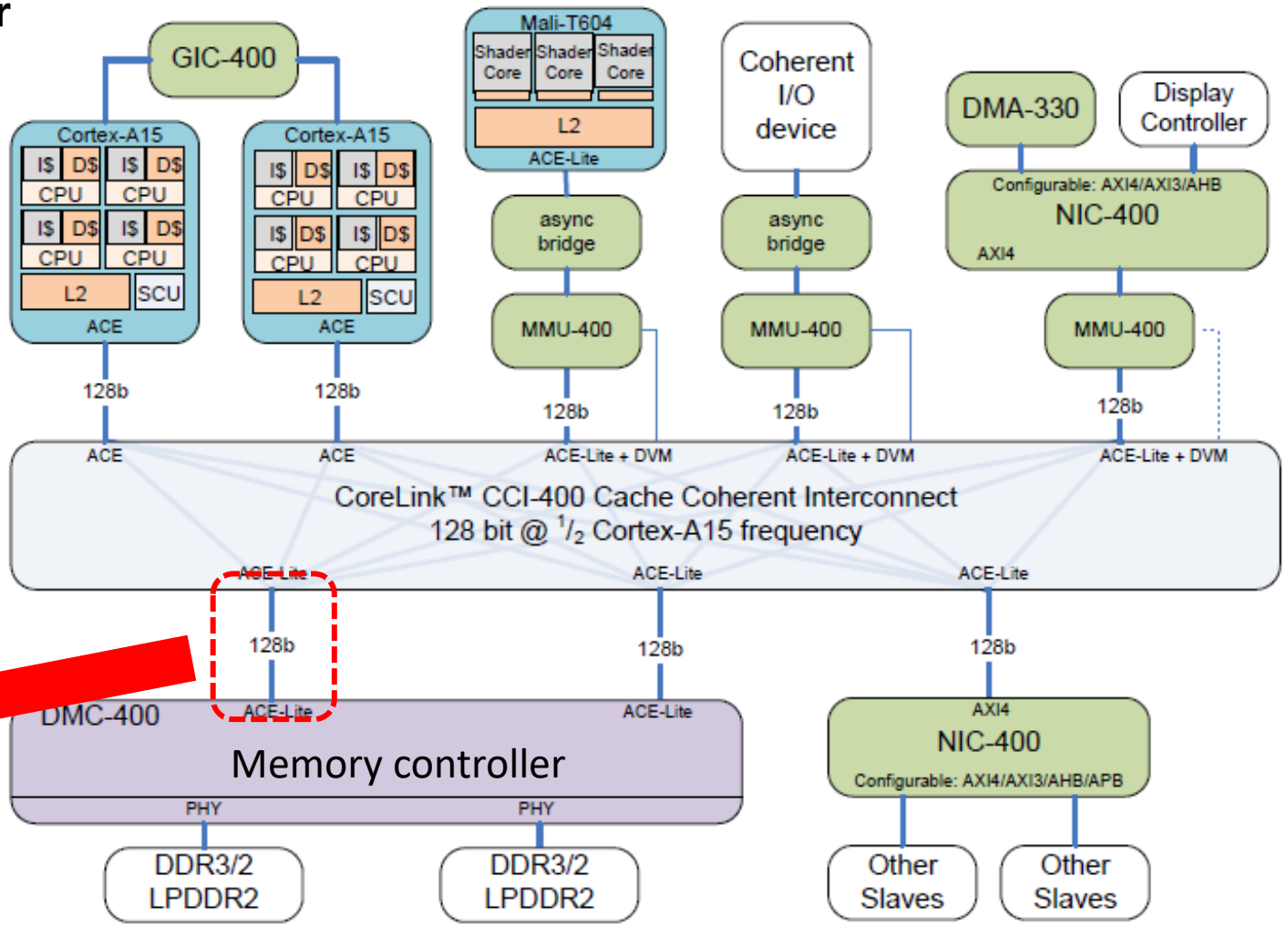
# Step 2: Sending Read Request to Memory Controller

**Sending a read request to memory controller**

Interconnect forwards the read request to the memory controller, i.e., ARVALID = '1'

Note #1: This happens on a different connection from CPU-Interconnect.
Note #2: Interconnect is the master in this communication



[Stevens, 2011]

# Step 3: Accessing DRAM (in Parallel)

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

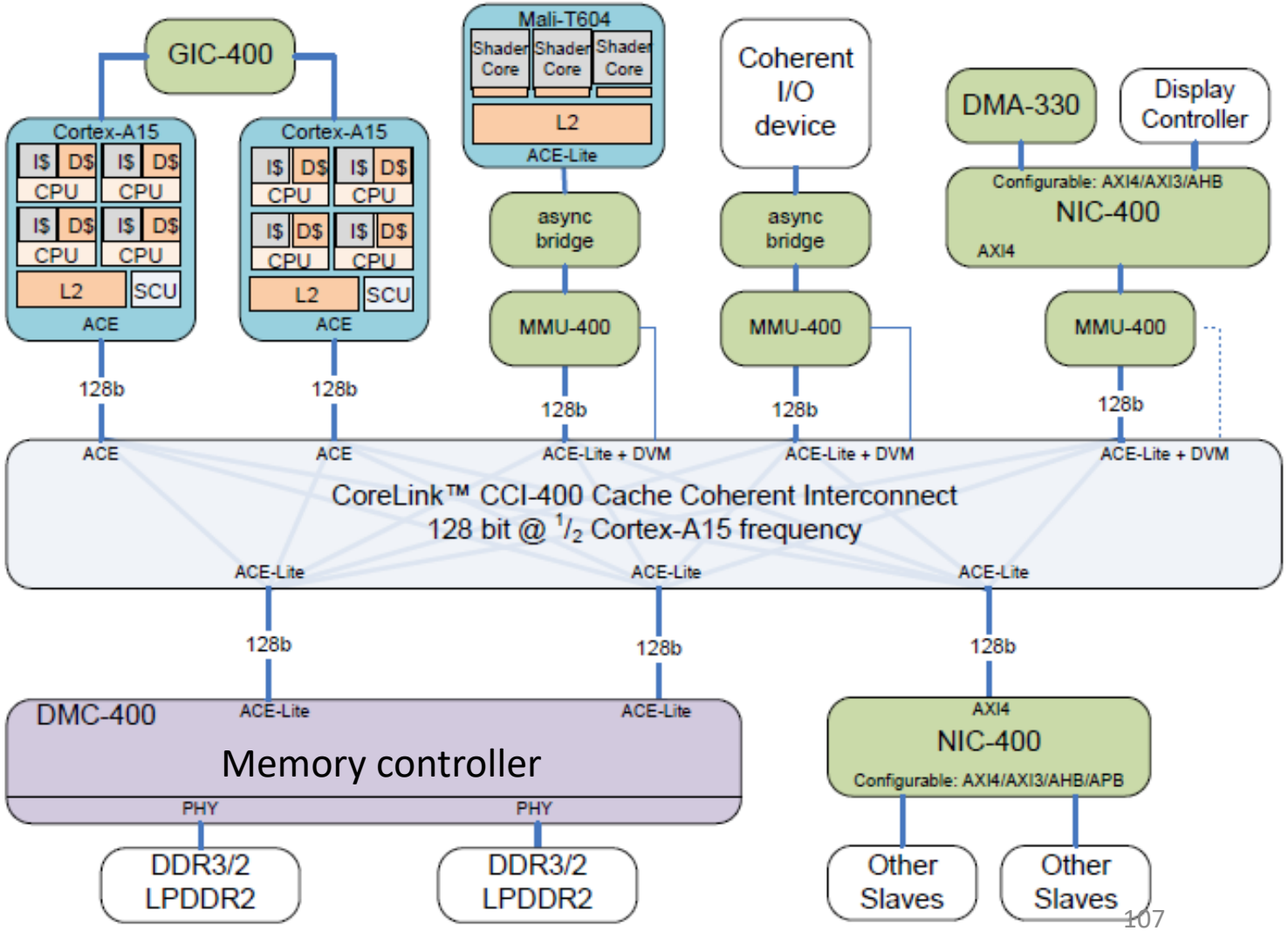
VA(x) → PA(x) translation on TLB

CPU sends to the interconnect
a read request for PA(x)

The read request arrives
at memory controller

Memory controller reads data@PA(x)
from DRAM

Memory controller sends the data
to CPU via the interconnect

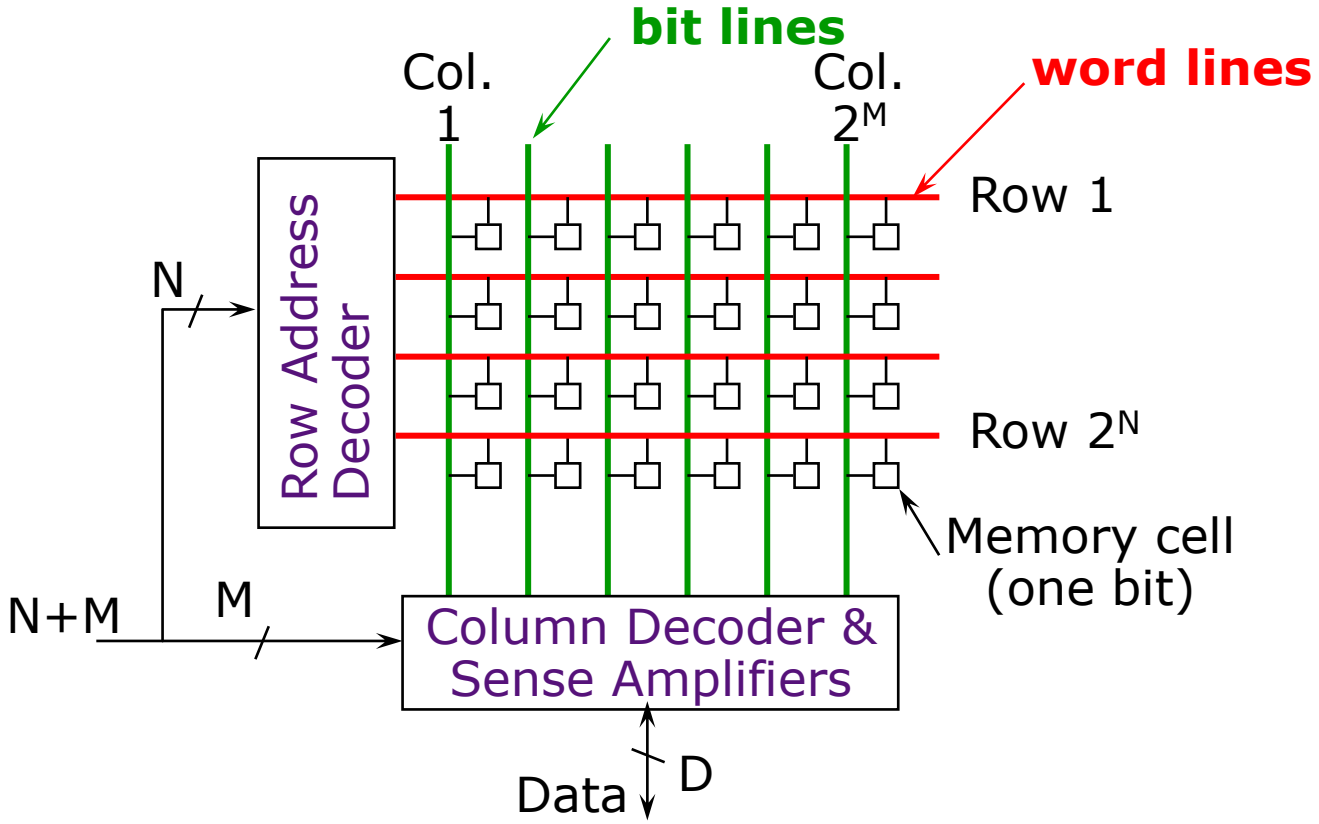


# Agenda

- DRAM architecture
- Memory access scheduling
- Refresh
- Row hammer
- Error correction code
- Summary

[K. Asanovic, 2008]

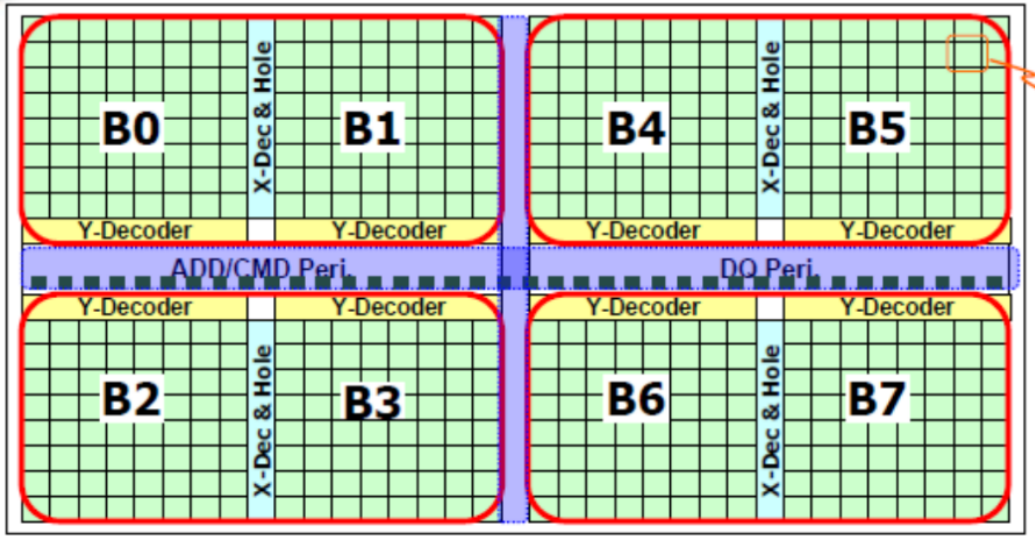
# DRAM Architecture

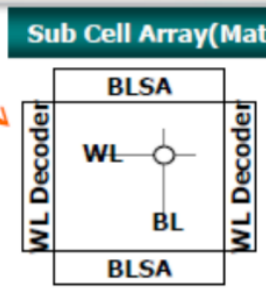



- Bits stored in 2-dimensional arrays on chip
- Modern chips have around 4~16 logical banks on a DRAM chip
  - each logical bank physically implemented as many smaller arrays

[SK Hynix]

# CORE , Array(Mat,Dec,SA,hole) & Peripheral




Cell Array영역(혹은 Bank)에 인접한 회로부분을 Core 라고 통칭한다.

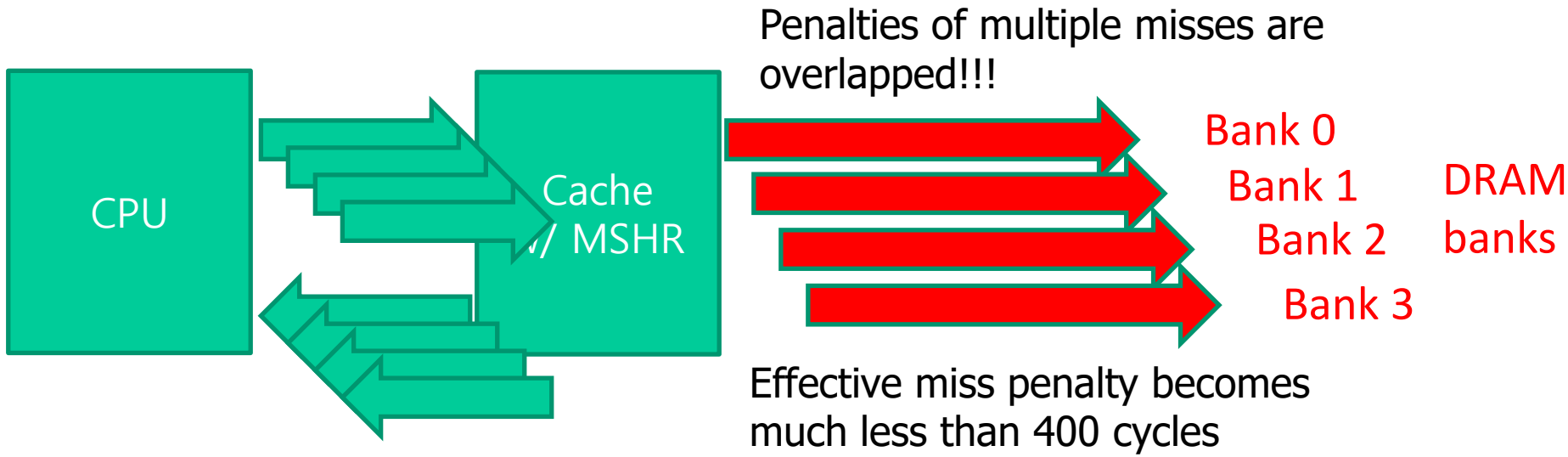
Core 이외의 영역으로서, Address/Command, Data In/out 의 입출력 및 제어를 위한 회로부분을 Peripheral 이라고 통칭한다.

| 분류 | | 기능 | 위치 |
|---|---|---|---|
| Core | BLSA array | B/L Sense Amplifier Array | Mem Cell array unit 외곽 |
| | WLD array | W/L Driver Array | Mem Cell array unit 외곽 |
| | Sub-hole | BLSA 및 WLD 구동 신호의 driver, IO switch 포함 | BLSA와 WLD array의 교차지점 |
| Dec/Hole | X-decoder | W/L Decoder | W/L방향의Mem cell array 인접 |
| | X-Hole | W/L Decoder 및 BLSA 동작 관련 controller<br>Row 동작(Active, Precharge, Refresh) controller | Bank와 Bank 사이 혹은<br>1 Bank의 center, X-dec 인접 |
| | Y-decoder | Y-select (BLSA array중 data 입출력이 되는 BLSA를 선택) 신호의 decoder | B/L방향의Mem cell array 인접 |
| | WTD,DBSA | BLSA와 연결된 Local IO선의 Write Driver, Sense Amplifier(DBSA) | Y-dec 하단 |
| Peripheral | | Address, Command, DQ, Power, IO등 Function Control 담당 | 칩 중앙 또는 외곽, core주변 |

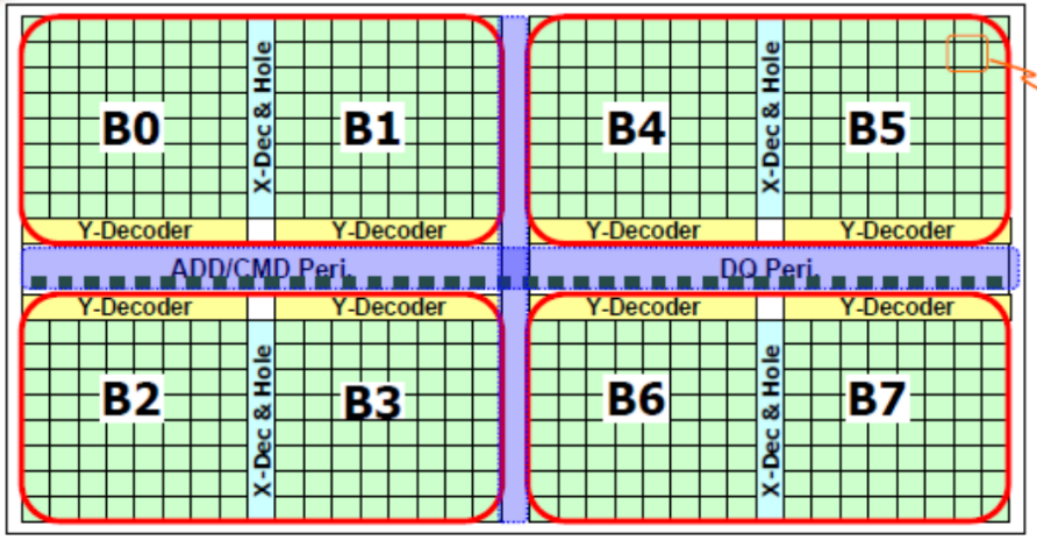
[Source: J. Kubiatowicz, 2000]

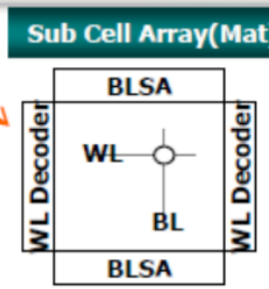
# Reduce Miss Penalty: Non-blocking Caches to reduce stalls on misses

- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
- Under the condition that such multiple misses, e.g., memory read requests, are served by multiple banks (via bank parallelism) or by the same row (via row buffer hits) in DRAM



# CORE , Array(Mat,Dec,SA,hole) & Peripheral



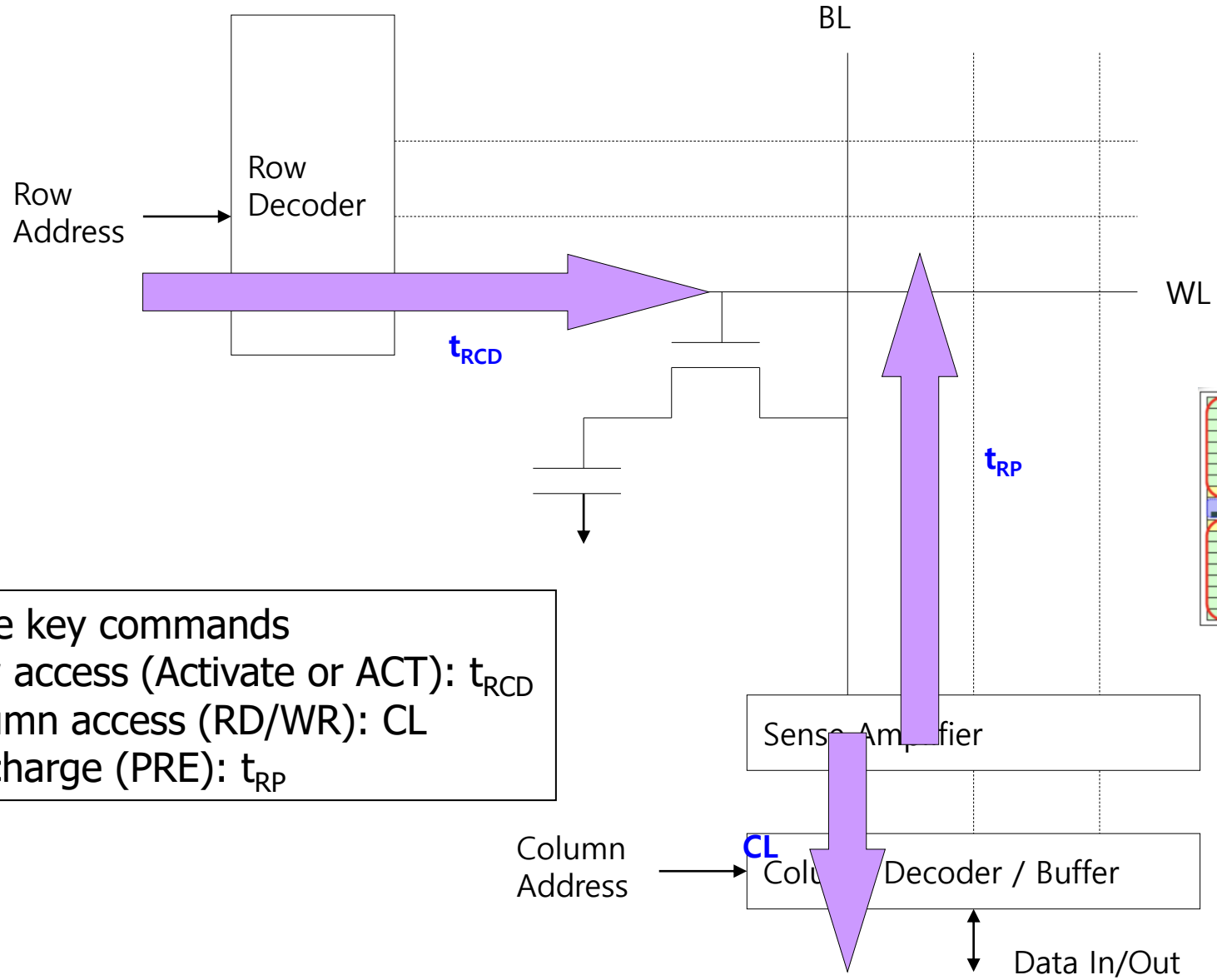

Cell Array영역(혹은 Bank)에 인접한 회로부분을 Core 라고 통칭한다.

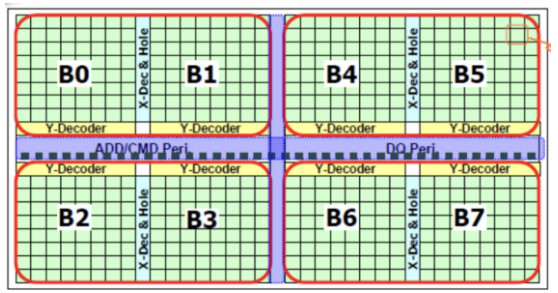
Core 이외의 영역으로서, Address/Command, Data In/out 의 입출력 및 제어를 위한 회로부분을 Peripheral 이라고 통칭한다.

| 분류 | | 기능 | 위치 |
|---|---|---|---|
| Core | BLSA array | B/L Sense Amplifier Array | Mem Cell array unit 외곽 |
| | WLD array | W/L Driver Array | Mem Cell array unit 외곽 |
| | Sub-hole | BLSA 및 WLD 구동 신호의 driver, IO switch 포함 | BLSA와 WLD array의 교차지점 |
| Dec/Hole | X-decoder | W/L Decoder | W/L방향의Mem cell array 인접 |
| | X-Hole | W/L Decoder 및 BLSA 동작 관련 controller<br>Row 동작(Active, Precharge, Refresh) controller | Bank와 Bank 사이 혹은<br>1 Bank의 center, X-dec 인접 |
| | Y-decoder | Y-select (BLSA array중 data 입출력이 되는 BLSA를 선택) 신호의 decoder | B/L방향의Mem cell array 인접 |
| | WTD,DBSA | BLSA와 연결된 Local IO선의 Write Driver, Sense Amplifier(DBSA) | Y-dec 하단 |
| Peripheral | | Address, Command, DQ, Power, IO등 Function Control 담당 | 칩 중앙 또는 외곽, core주변 |

[B. Jacobs, 2002]

# Basic DRAM Operations (i.e., Commands)



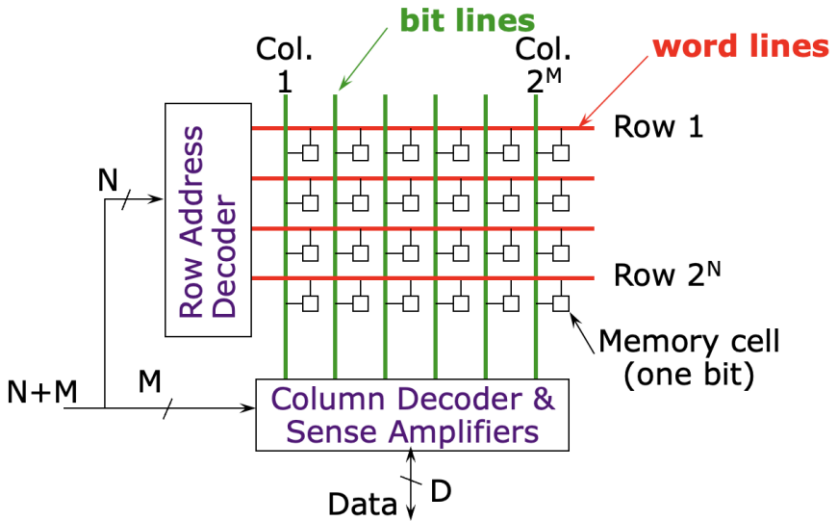



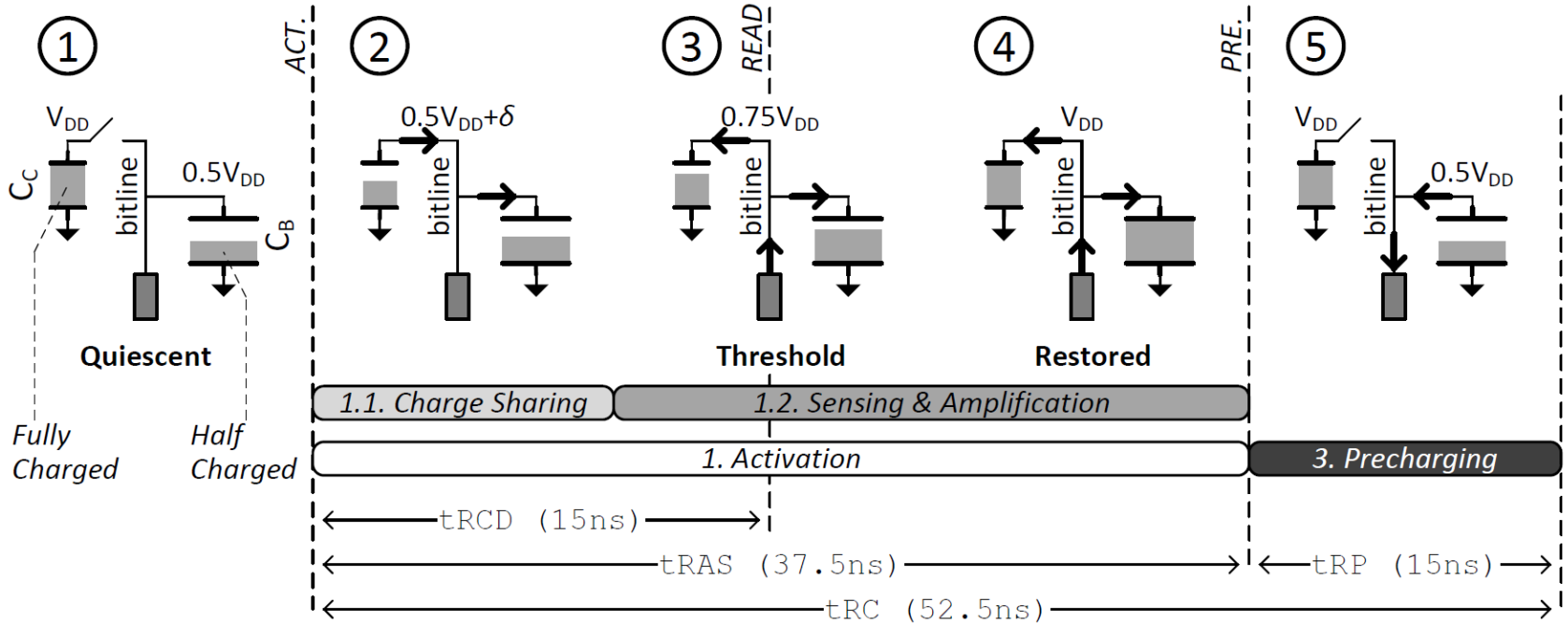

Three key commands
-Row access (Activate or ACT): $t_{RCD}$
-Column access (RD/WR): CL
-Precharge (PRE): $t_{RP}$

[Lee, 2013]

# Accessing DRAM Cells with Timing Constraints: tRCD, tRAS, tRP and tRC

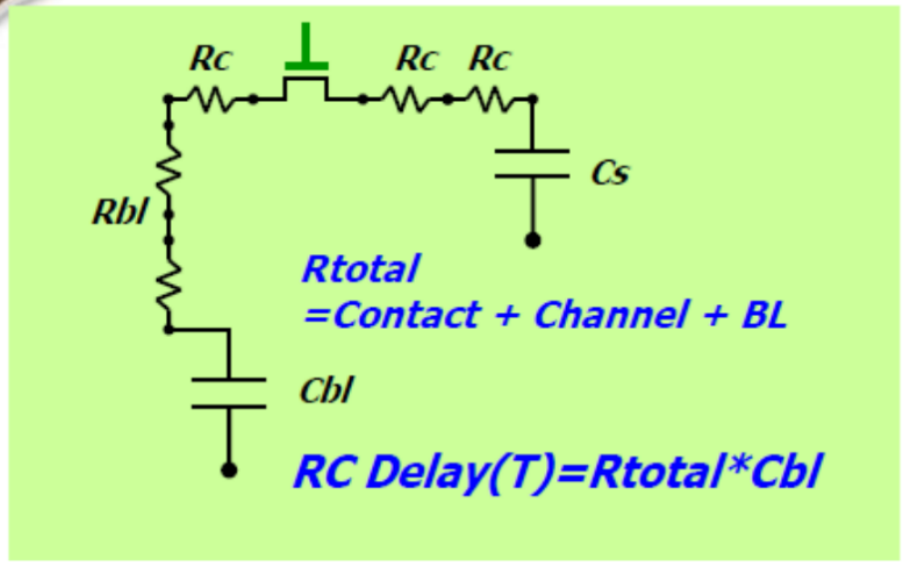
- Assuming $C_{BL} = 3.5C_{cell}$
- Bitline capacitance is large due to parasitic capacitance

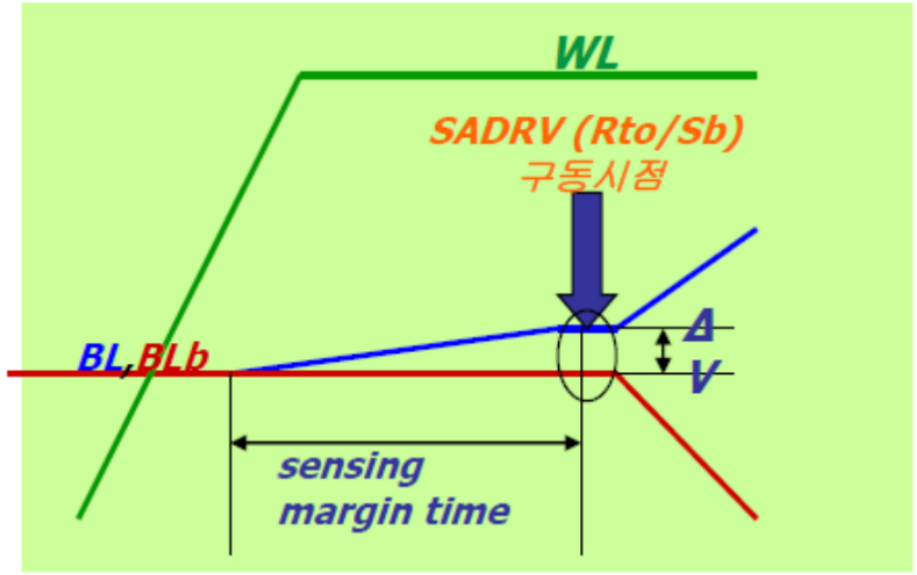


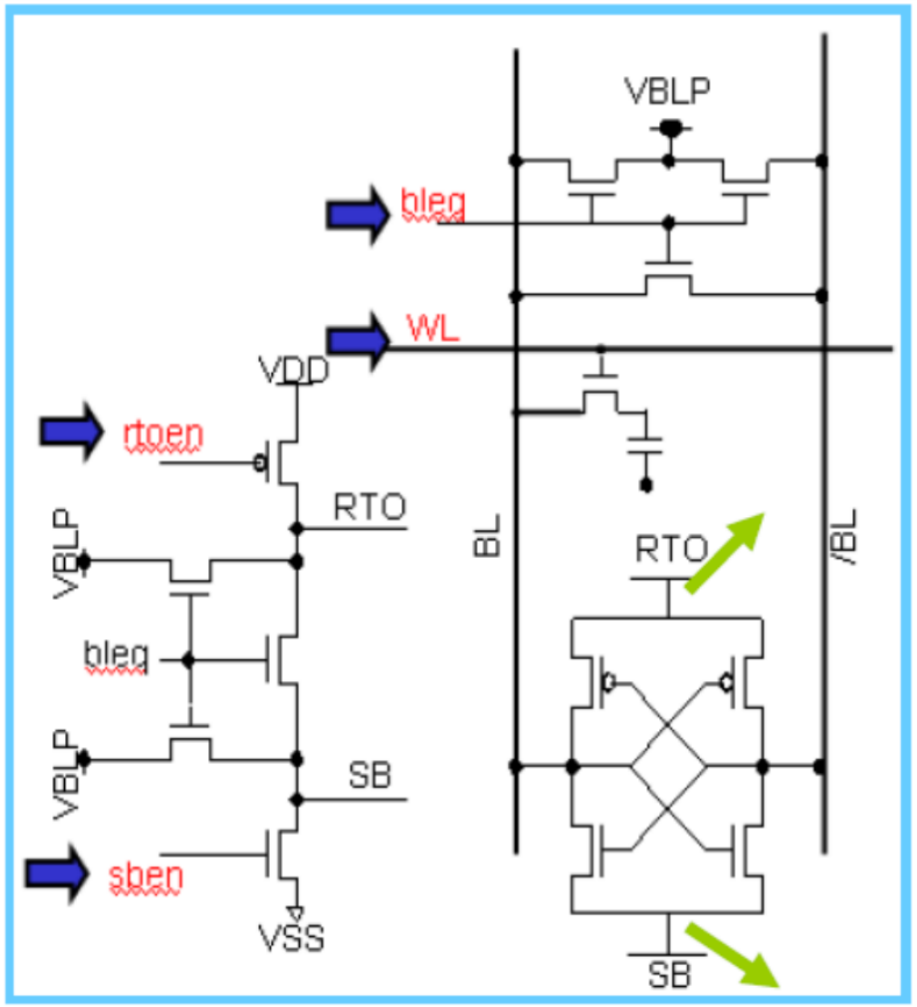



# Charge Sharing & Sense Amplifier

[SK Hynix]



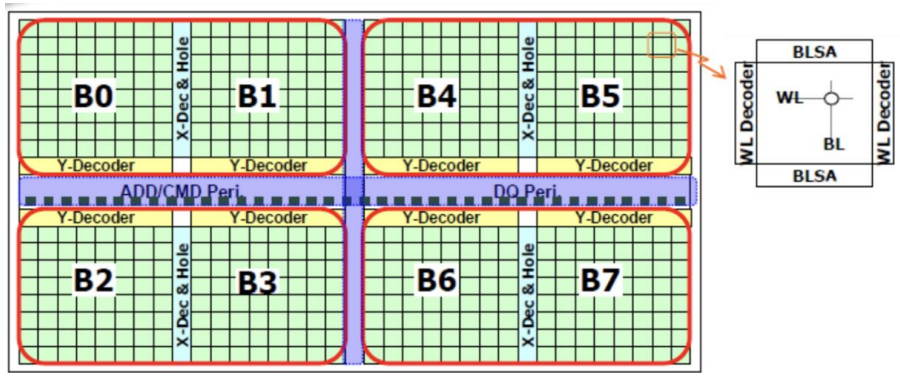



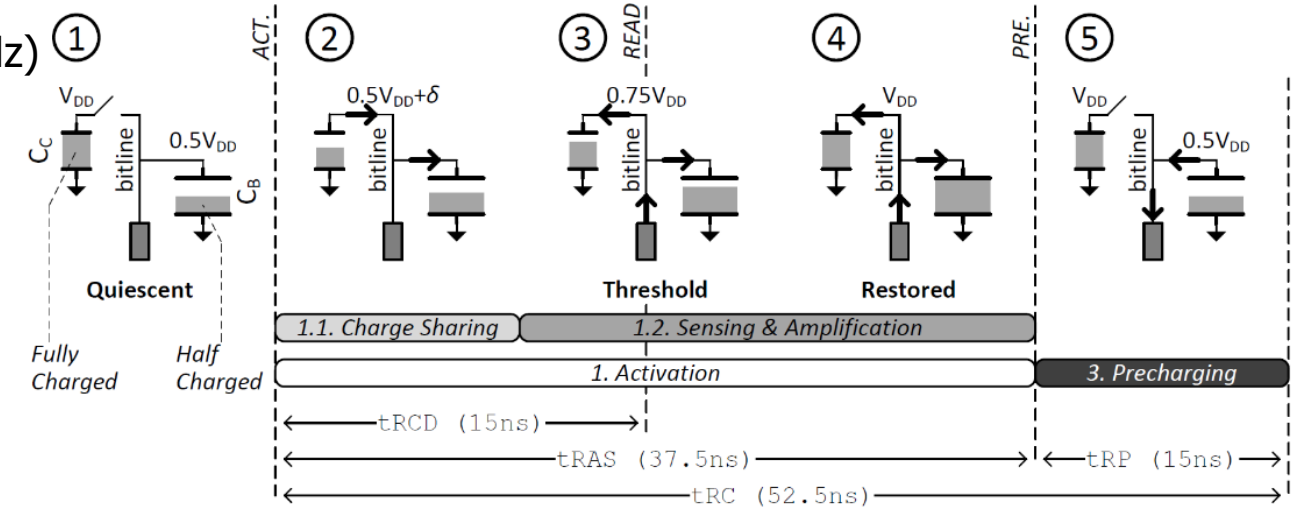

Active순서 : Bleq Off → WL On → Rto/Sb enable

Precharge순서 : WL Off → Rto/Sb disable → Bleq On

# Key DRAM Timing Parameters

- $t_{RC}$: minimum time from the start of one row access to the start of the next (ACT-to-ACT) = tRAS + tRP
  - $t_{RC}$ = 57.5 ns for DDR2-800
- $t_{RCD}$: minimum time from ACT to CAS.
  - $t_{RCD}$ = 12.5 ns for DDR2-800 (5 cycles @ 400MHz)
- CL: CAS latency, minimum time from the start of column access (CAS) command to the start of read data
  - CL = 12.5 ns for DDR2-800 (5 cycles @ 400MHz)
- $t_{RP}$: Precharge latency, minimum time from the precharge command to a row access of the same bank
  - $t_{RP}$ = 12.5 ns for DDR2-800 (5 cycles @ 400MHz)
- E.g., CL-$t_{RP}$-$t_{RCD}$ = 5-5-5

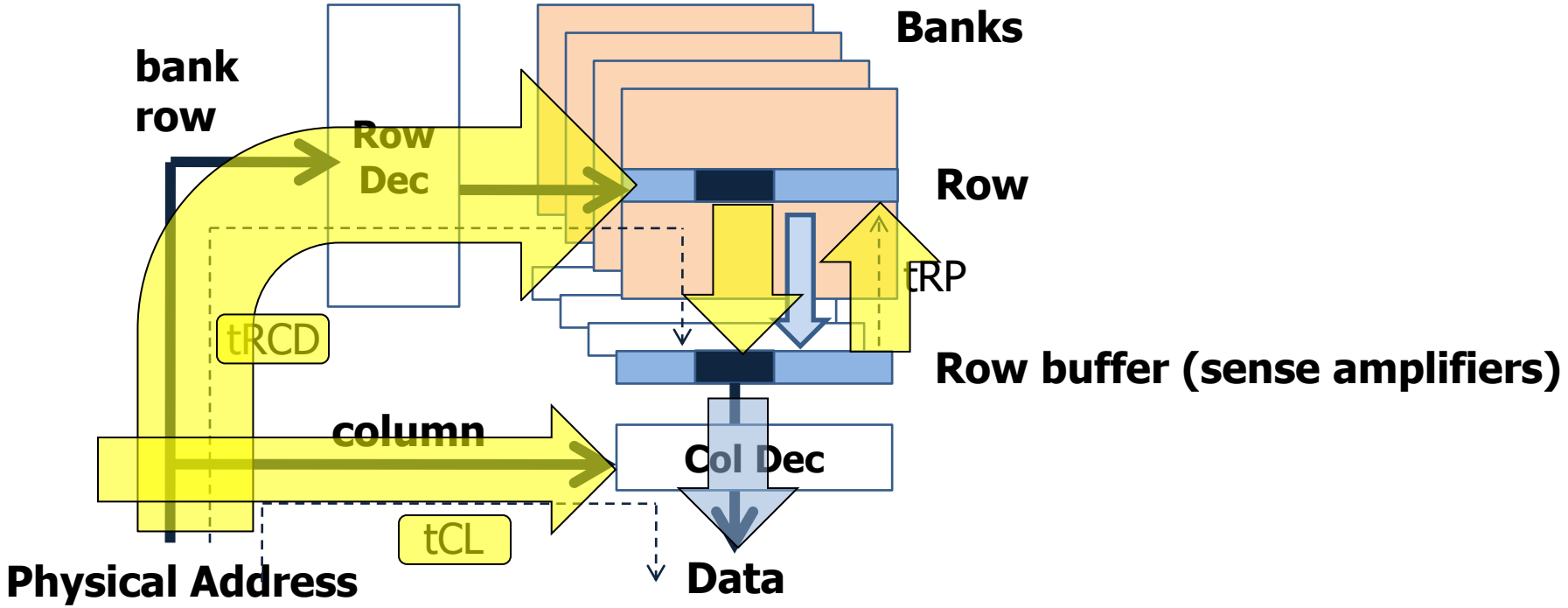


[D. Lee, 2008]

# DDR SDRAM

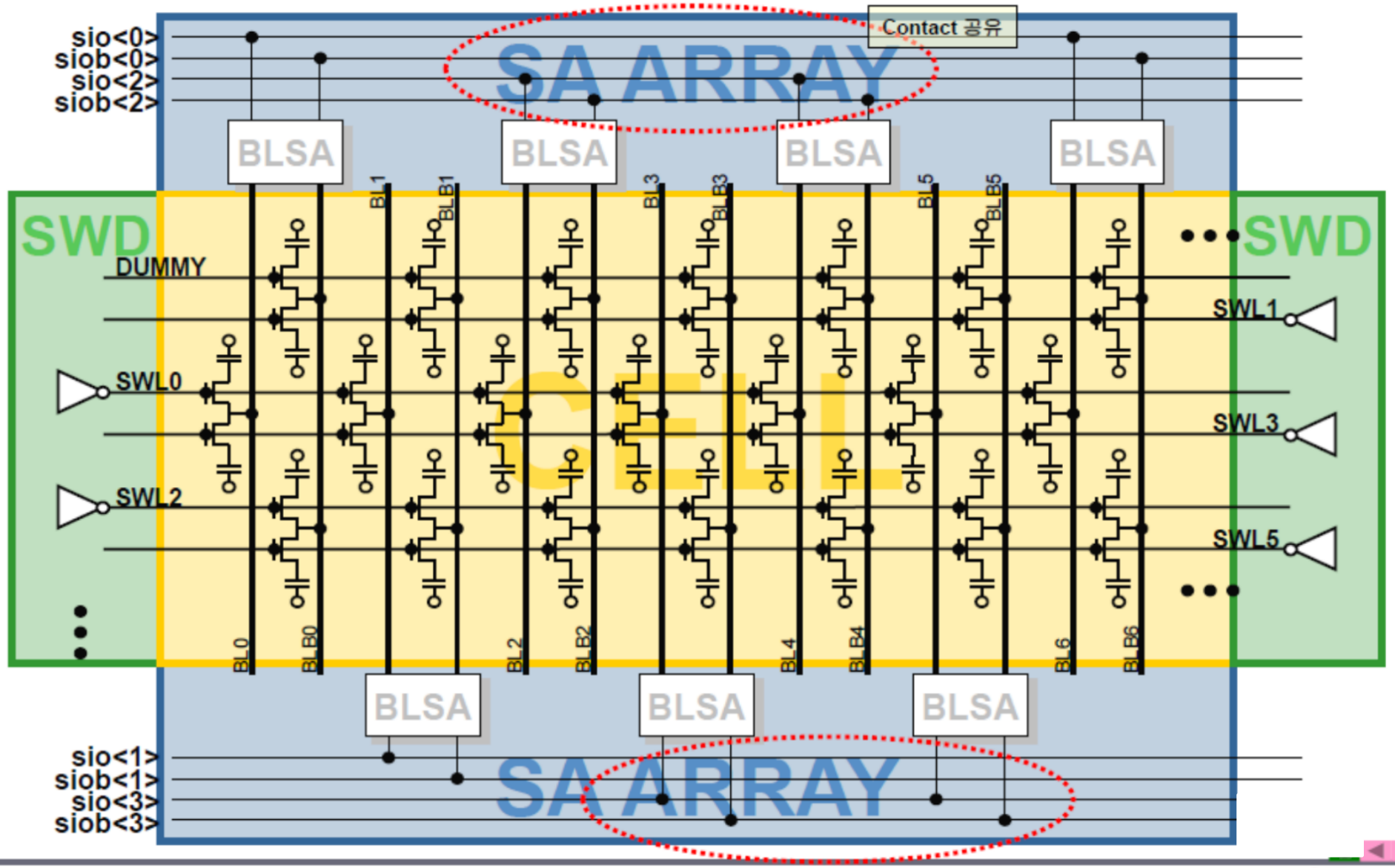
- Three dimensions: bank, row, and column




- **Memory access latency**
  - **E.g., 3-3-3: 3 cycles for each of ACT, RD/WR, & PRE**

# MAT Array Assignment

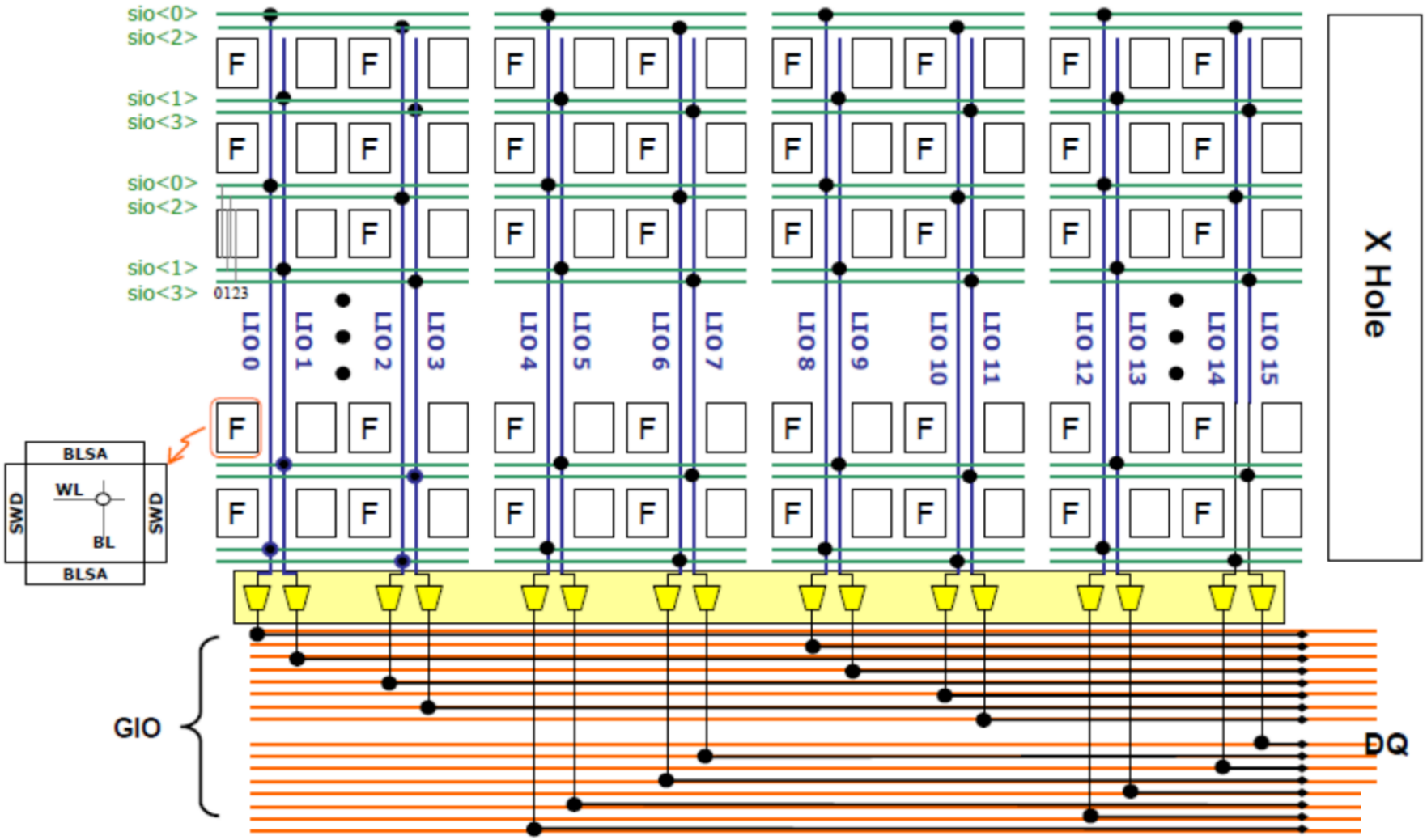
[SK Hynix]



# CORE Data I/O Line 구성(예)

[SK Hynix]

## DRAM Internal Data Bus Line Connection



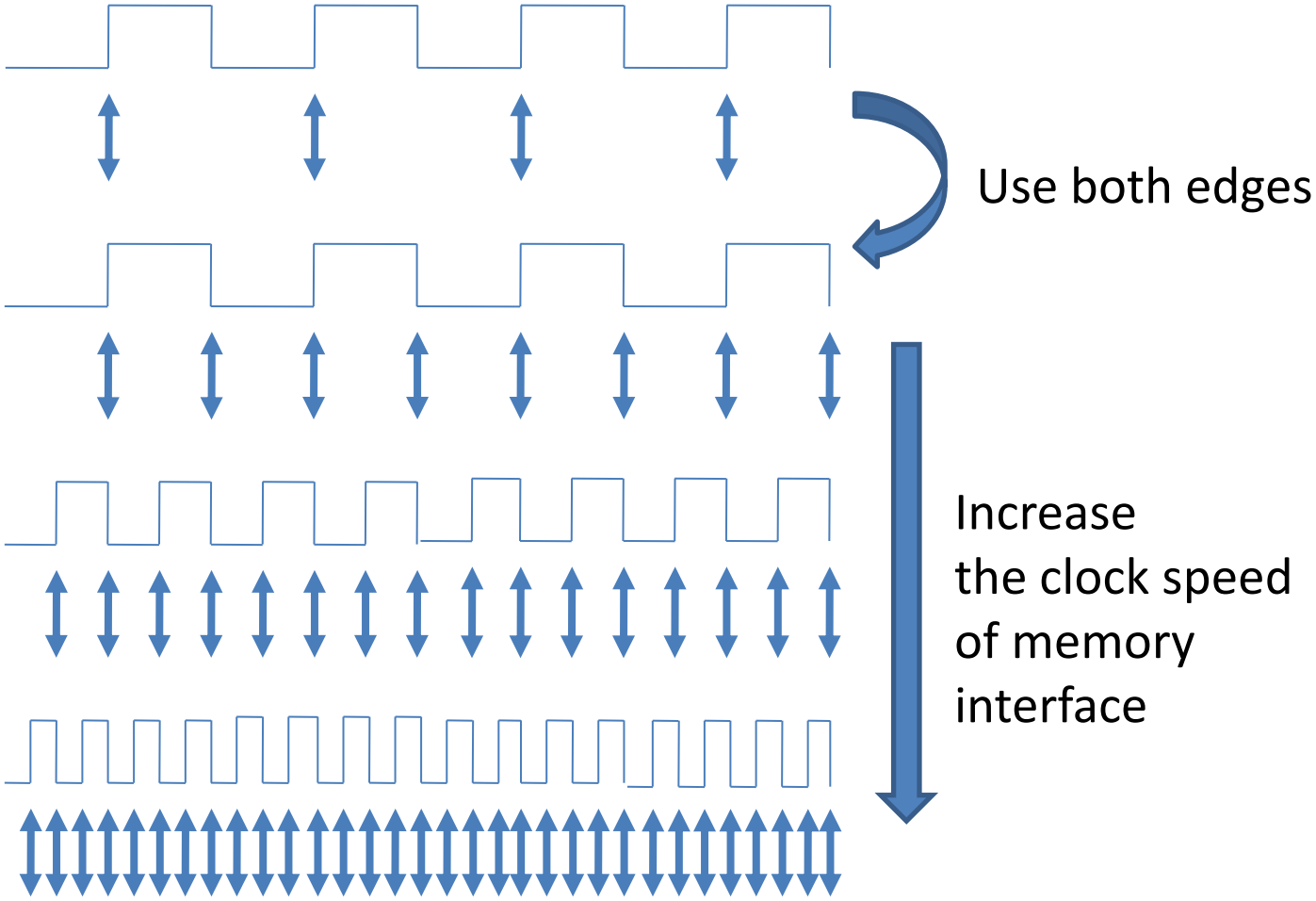
# DDR Memory Operation: SDRAM, DDR, and DDR2/3

- SDR SDRAM
- DDR
- DDR2

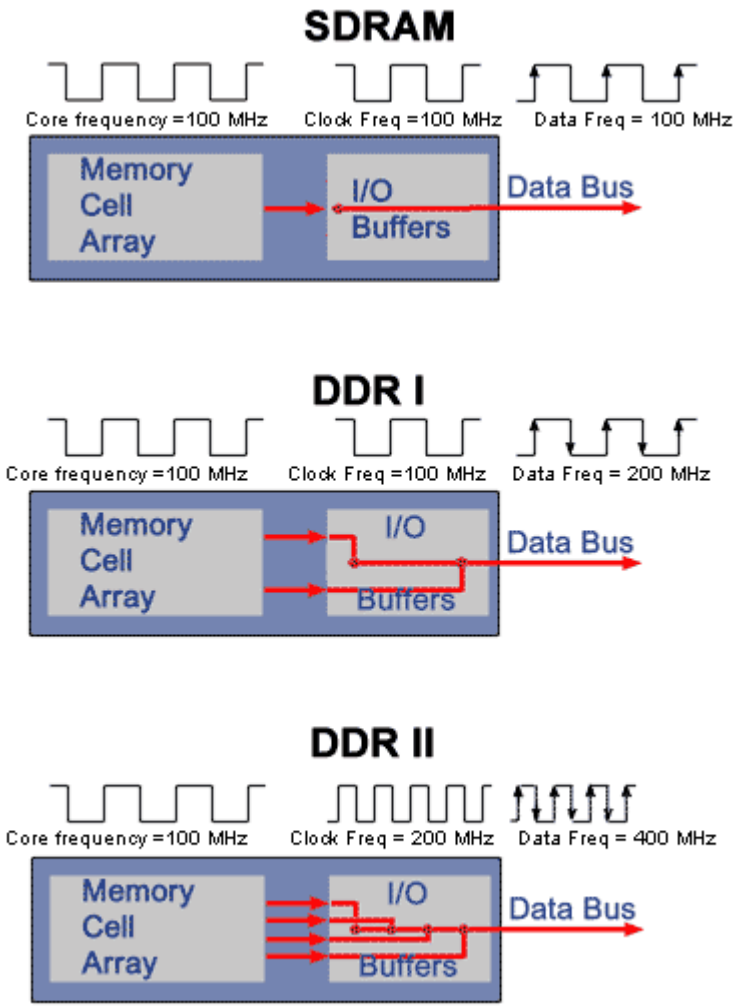
Max bus clock
400MHz

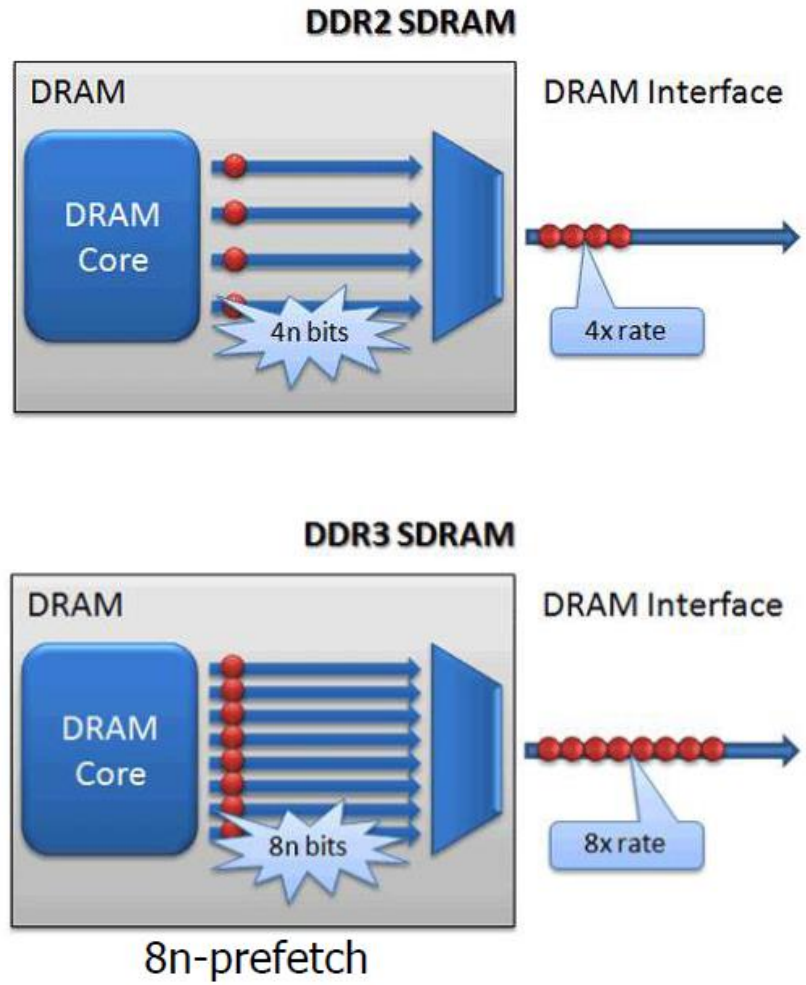
- DDR3

Max bus clock
800MHz

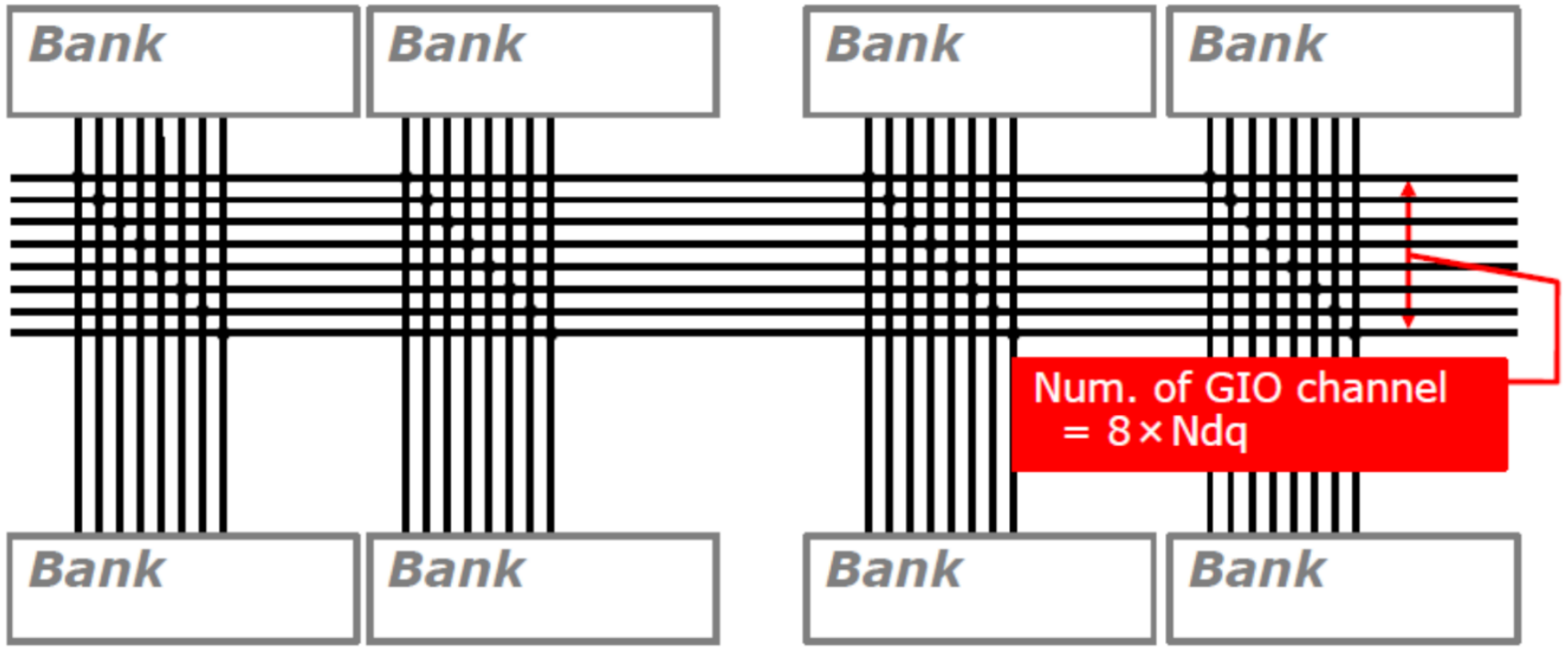


# SDRAM, DDR, DDR2, and DDR3



# Pre-fetch Diagram(DDR3)




- Pre-fetch operation
  - 8-bit pre-fetch
  - **[8 × Ndq]** data access

  (If the output data rate is **1.6Gbps**, the internal data rate is **200Mbps, same as DDR1**)

# Agenda

- DRAM architecture
- Memory access scheduling
- Refresh
- Row hammer
- Error correction code
- Summary

[Stevens, 2011]

# Step 3: Accessing DRAM (in Parallel)

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

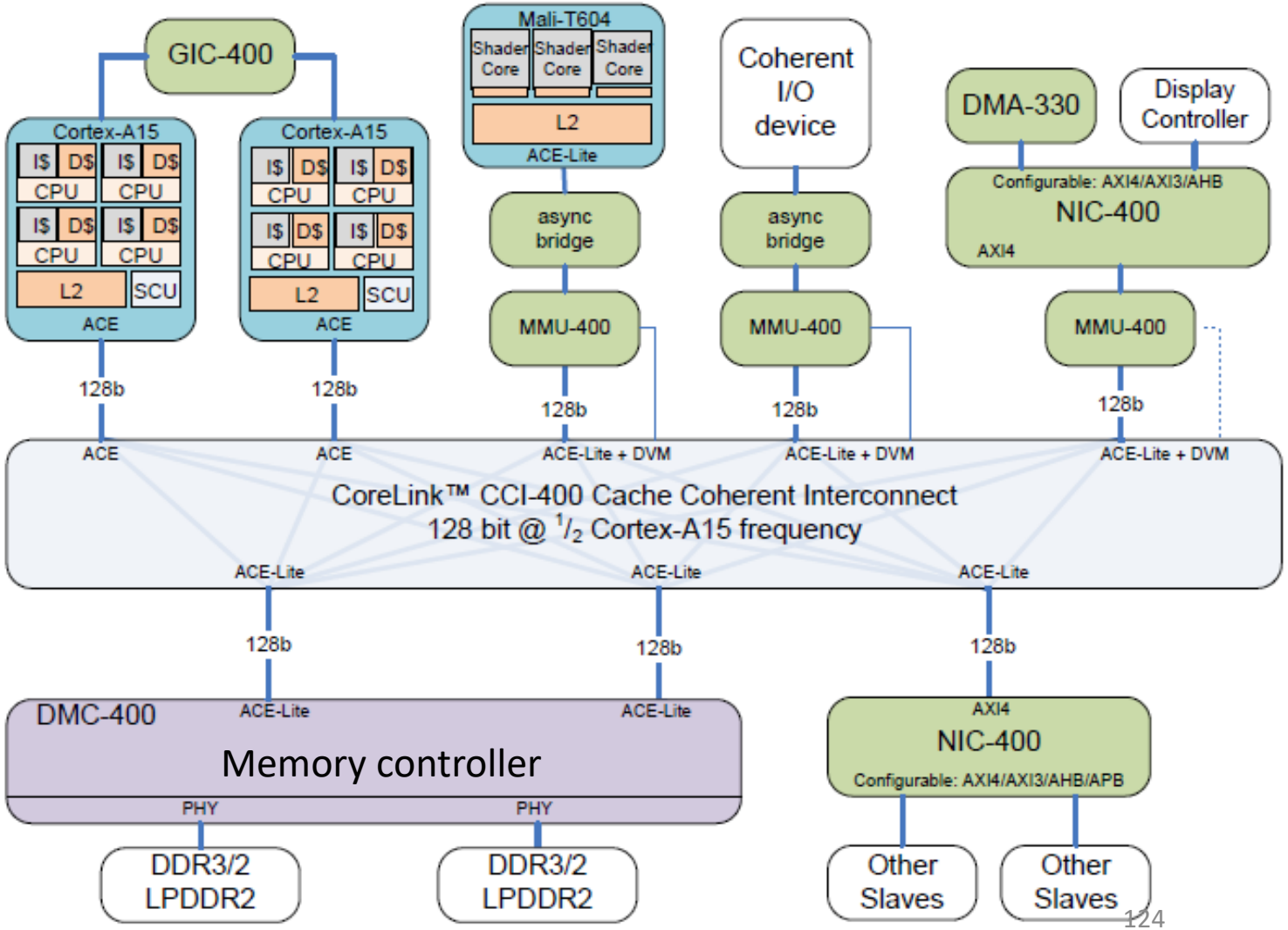
VA(x) → PA(x) translation on TLB

CPU sends to the interconnect
a read request for PA(x)

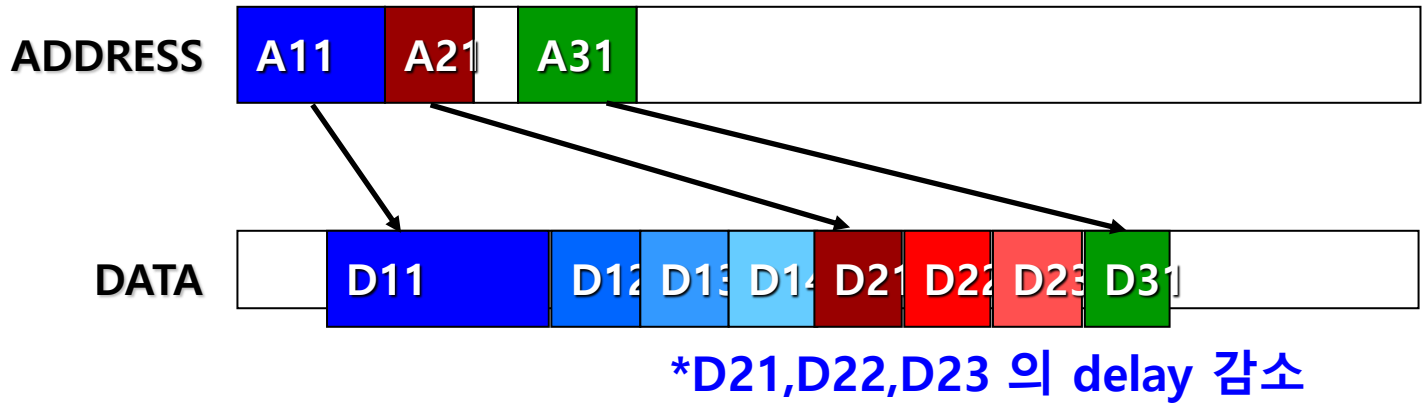
The read request arrives
at memory controller

Memory controller reads data@PA(x)
from DRAM

Memory controller sends the data
to CPU via the interconnect



# Benefit of Split Transaction: Multiple Outstanding Requests

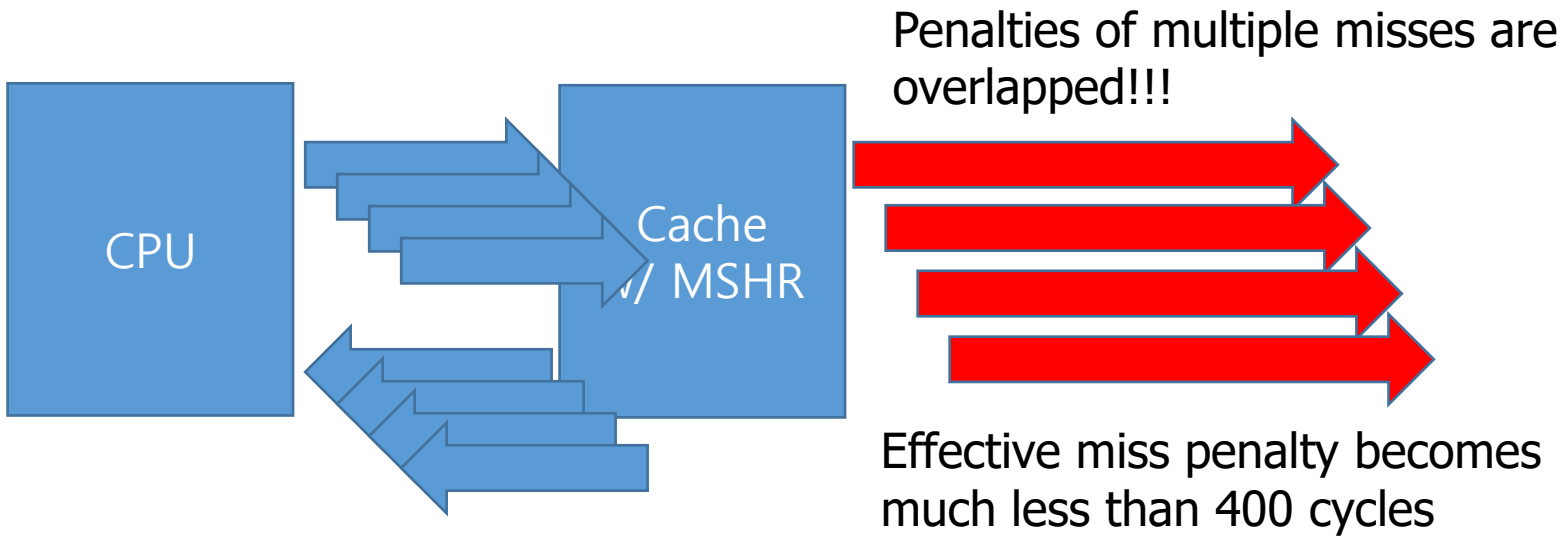



- Parameters for multiple outstanding requests
  - Master I/F: Issuing capability → master가 generation 할 수 있는 outstanding request의 개수
  - Slave I/F: Acceptance capability → slave가 받아 들일 수 있는 outstanding request의 개수

[Source: J. Kubiatowicz, 2000]

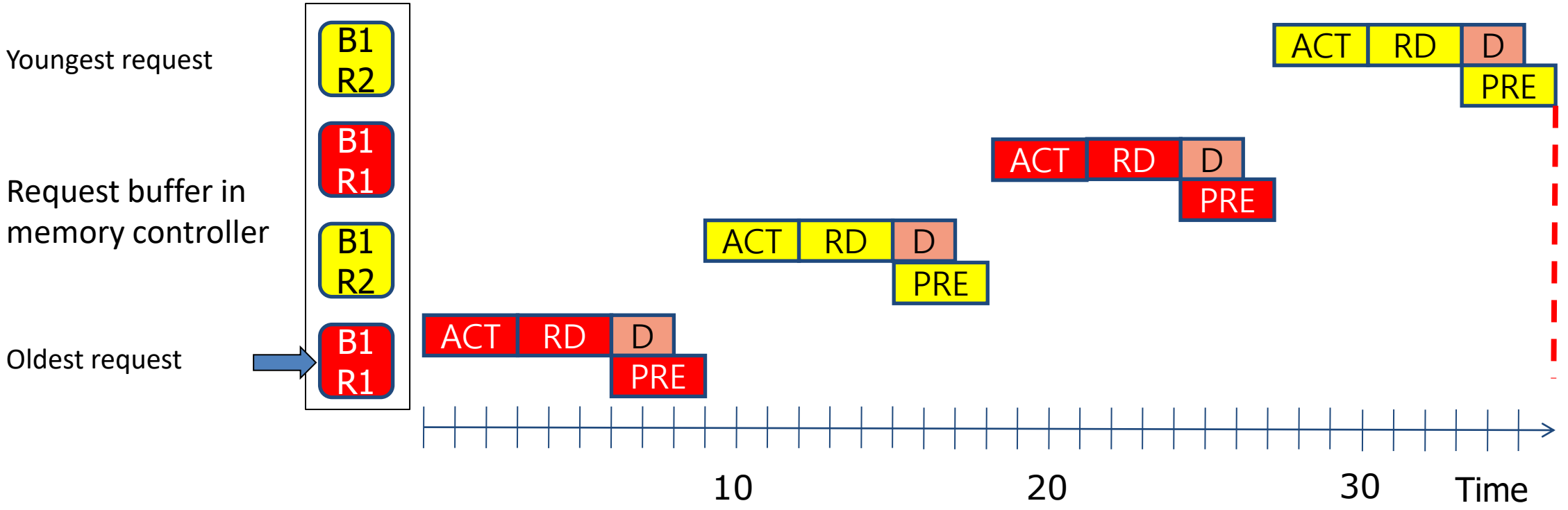
# Non-blocking Caches to Reduce Stalls on Misses

- "*hit under multiple miss*" or "*miss under miss*" may further lower the eff ective miss penalty by overlapping multiple misses



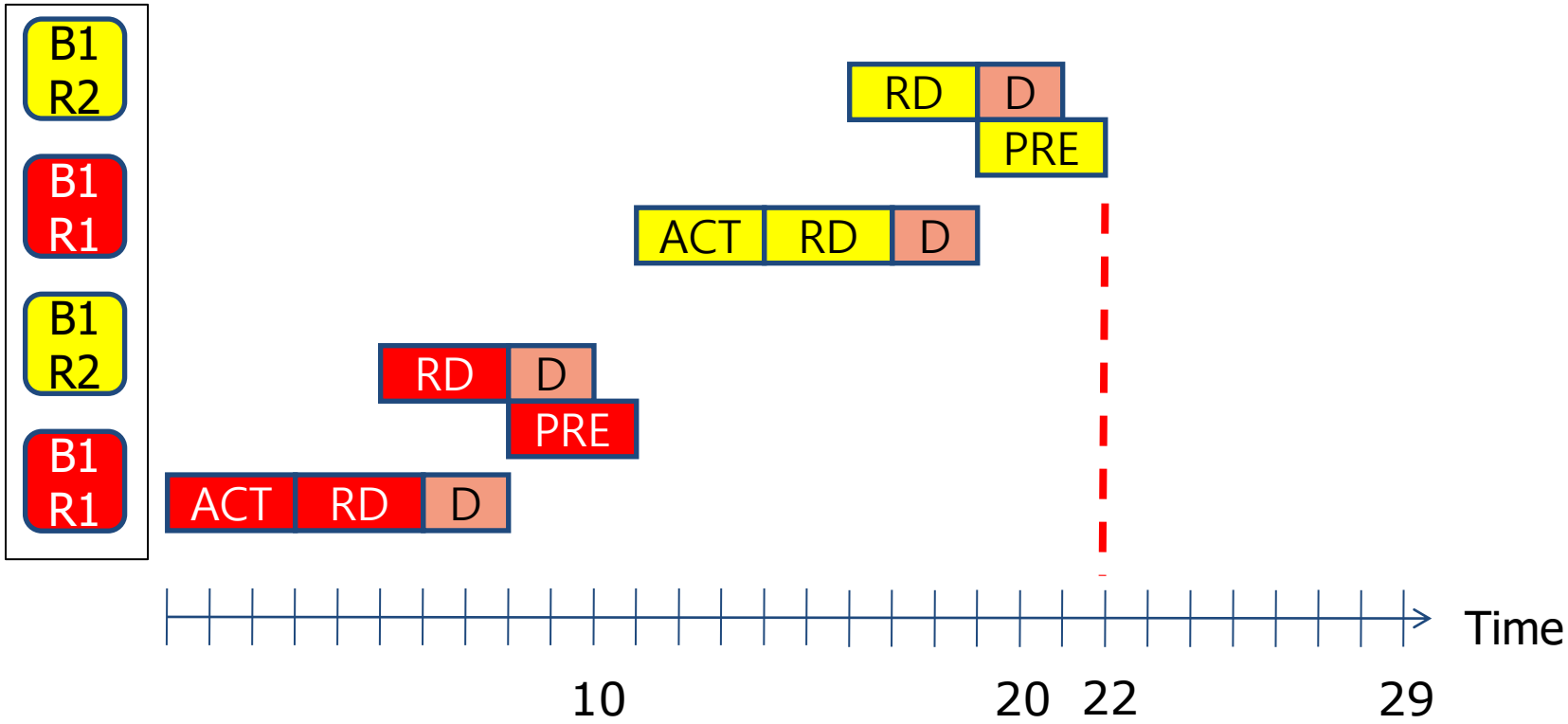
[D. Lee, 2008]

# Memory Access Scheduling Problem



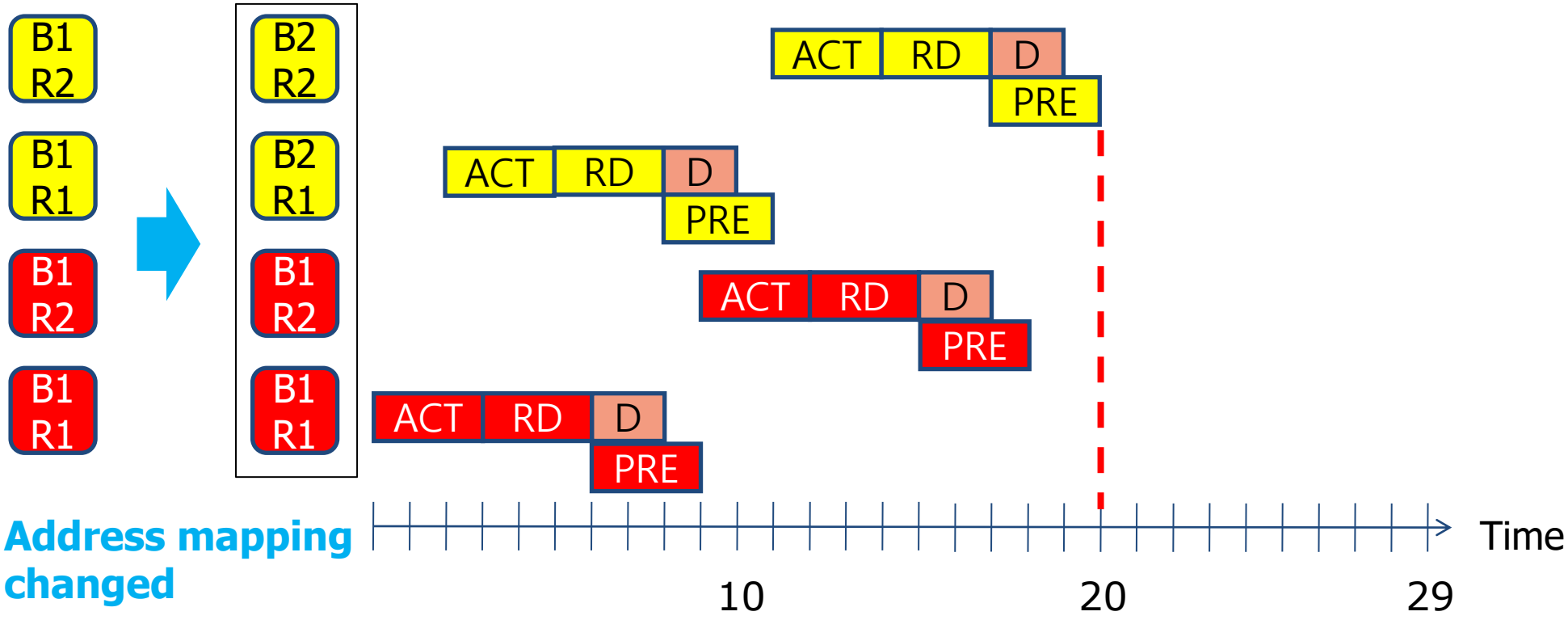

Memory Utilization = 8 cycles / 36 cycles = 22.2%

# Reordering Accesses to Exploit Open Row (or Row Buffer Hits)



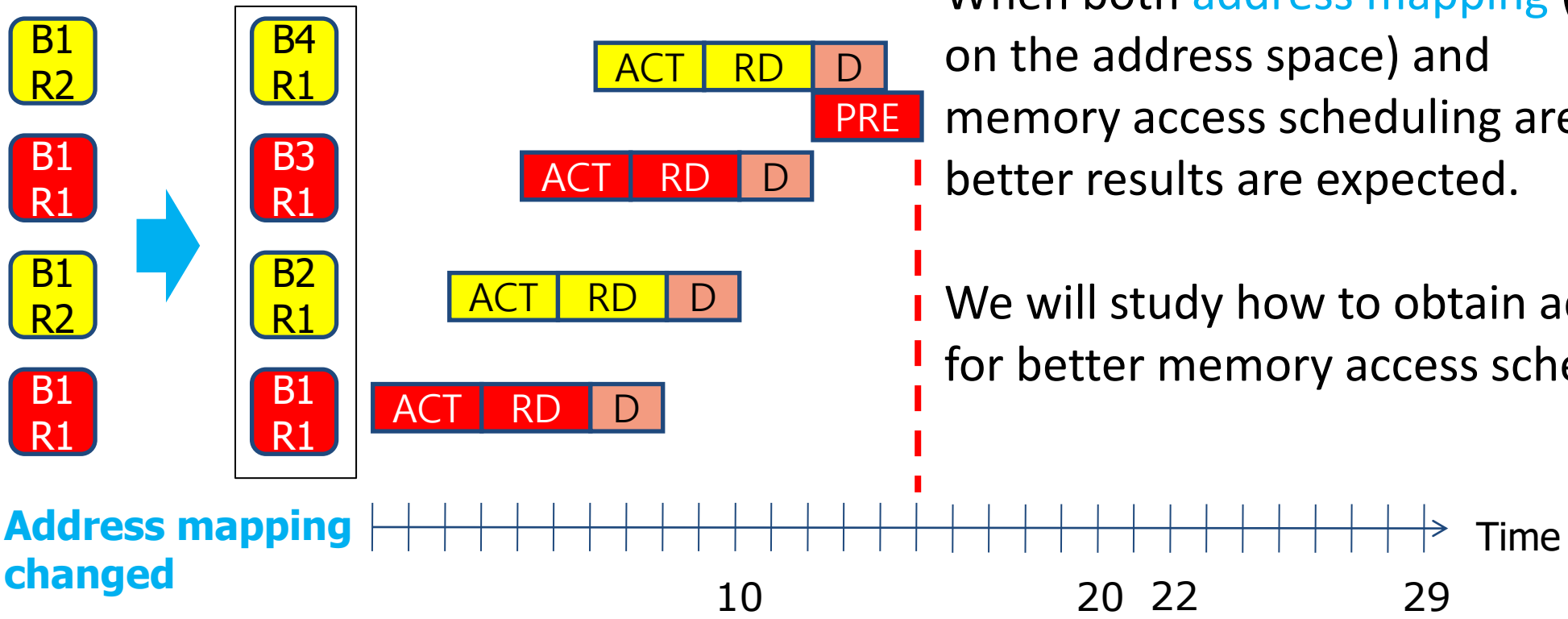

Memory Utilization = 8/22 = 36.4%

# Reordering Accesses when Accessing Data Distributed on Two Banks




Memory Utilization = 8/20 = 40%

# What if Accessing Data Distributed on Four Different Banks?




When both address mapping (placing data on the address space) and memory access scheduling are considered, better results are expected.

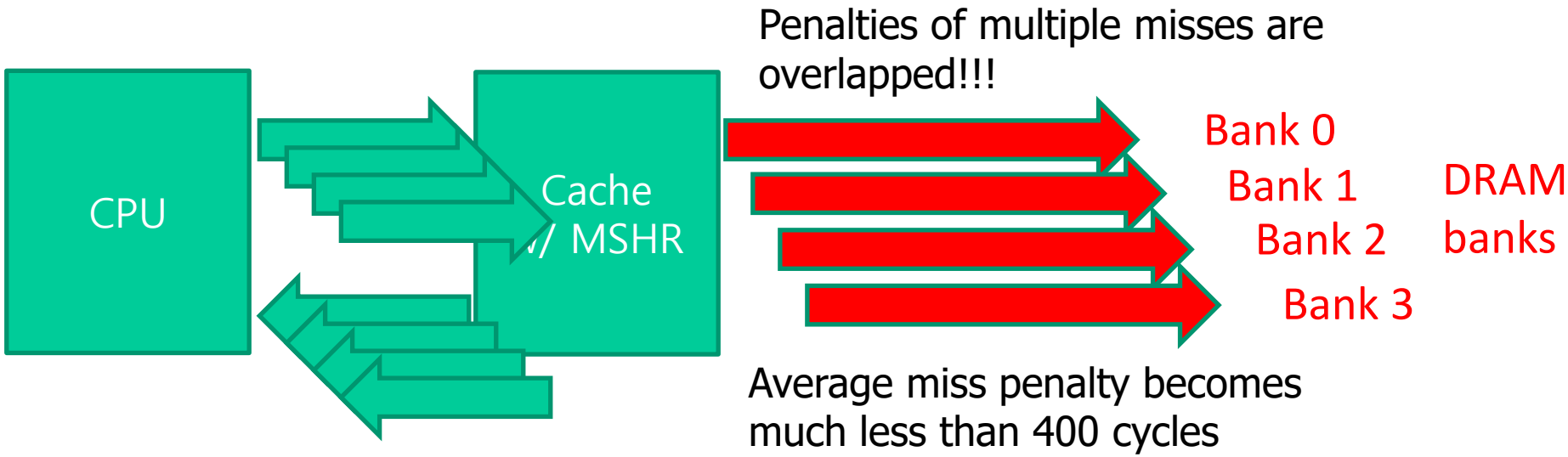
We will study how to obtain address mapping for better memory access scheduling later

Memory Utilization = 8/15 = 53.3%

[Source: J. Kubiatowicz, 2000]

# Reduce Miss Penalty: Non-blocking Caches to reduce stalls on misses

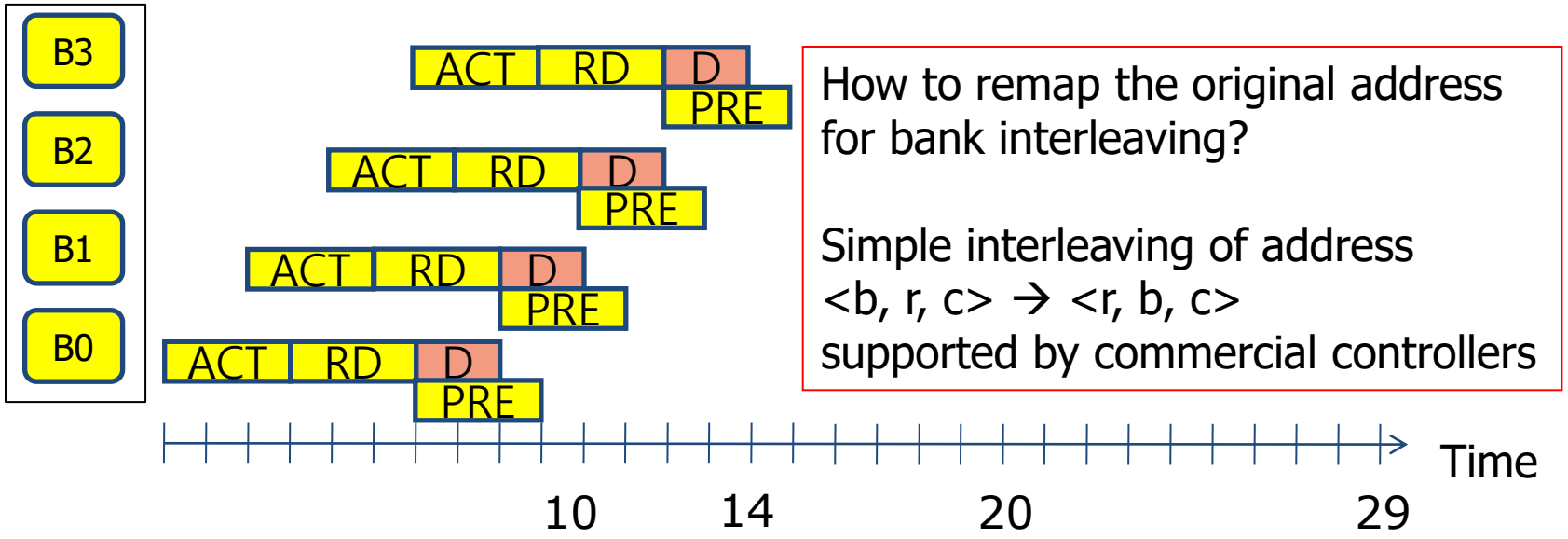
- "*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses
- Under the condition that such multiple misses, e.g., memory read requests, are served by multiple banks (via bank parallelism) or by the same row (via row buffer hits) in DRAM

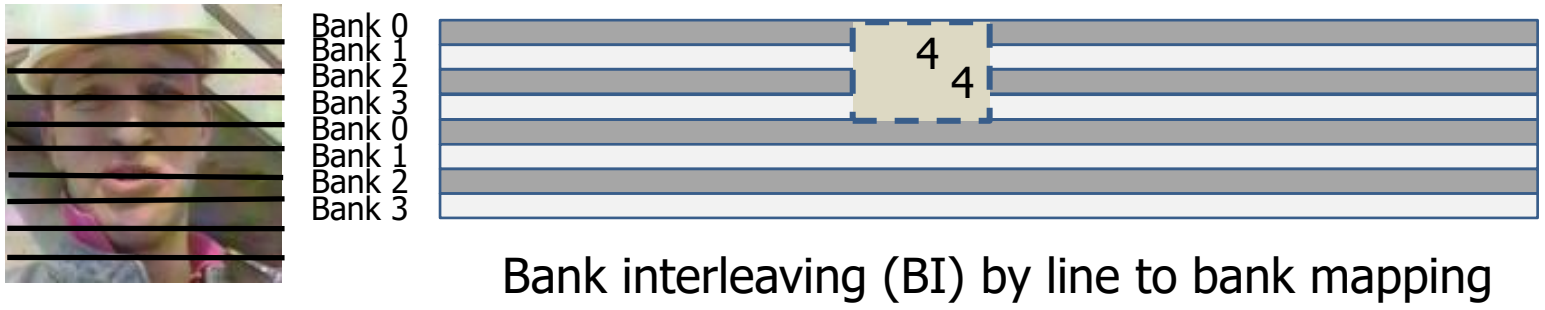


[D. Lee, 2008]

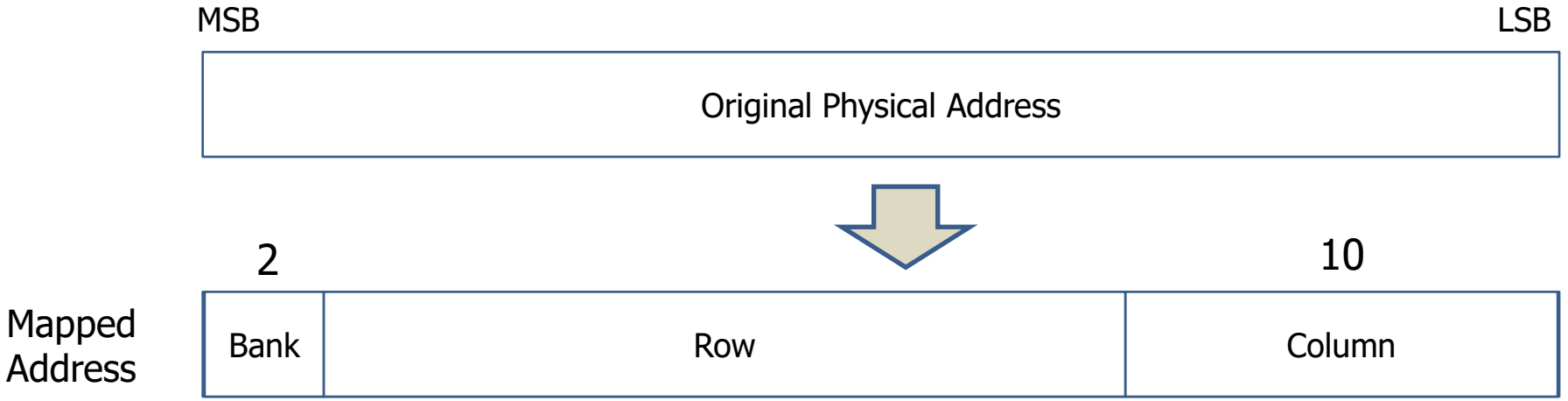
# Address Mapping for Bank Parallelism

- Accesses to multiple different banks



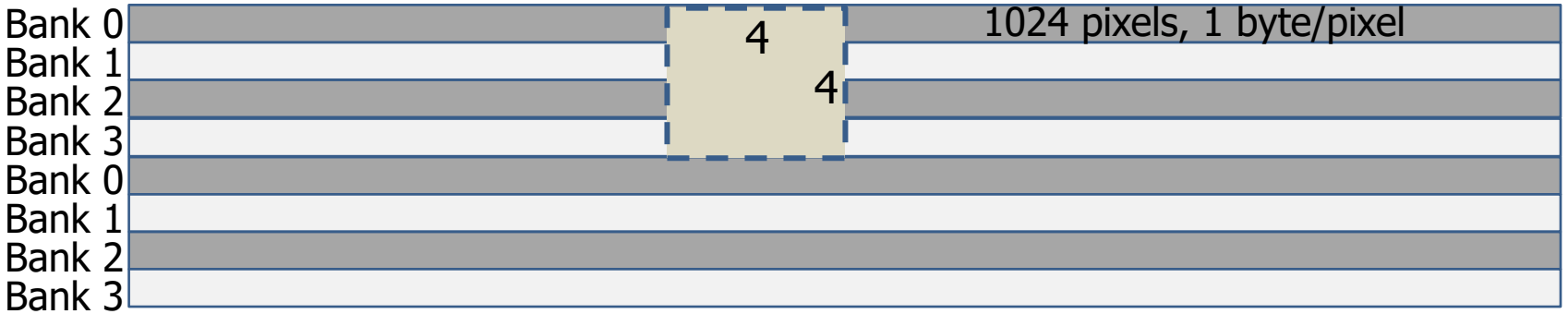




Bank interleaving (BI) by line to bank mapping




(a) Conventional address mapping to DDR memory


(b) Address remapping for bank interleaving

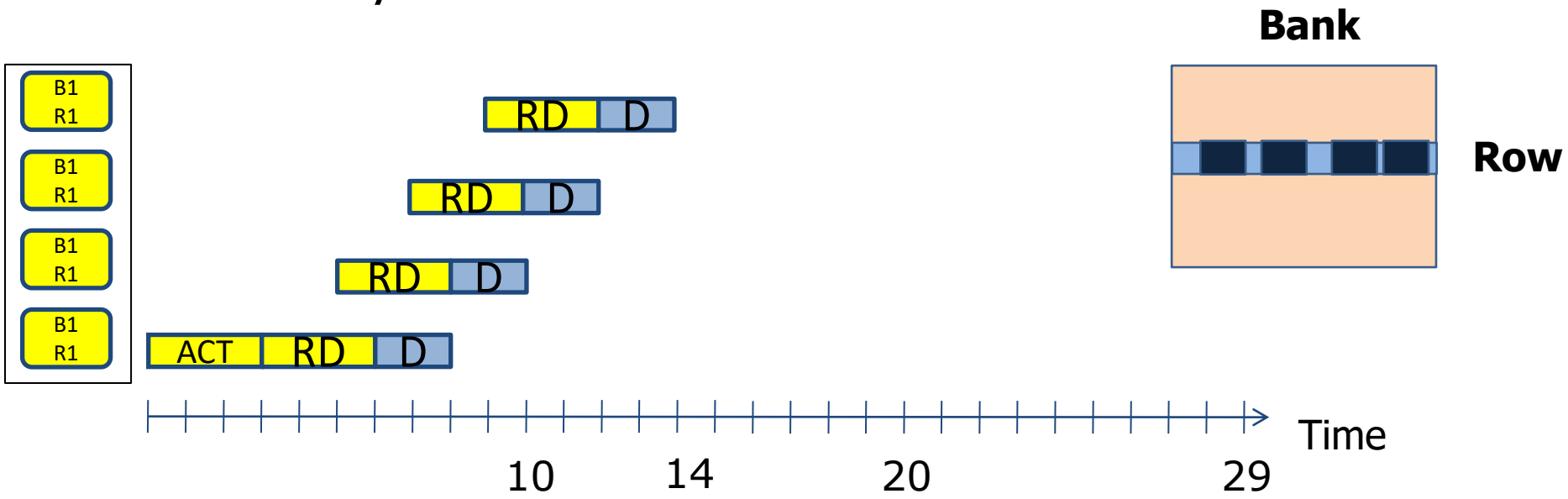



(c) Line to bank mapping

[D. Lee, 2008]

# Further Improving Memory Accesses by Exploiting Spatial Locality

- Accesses to the already activated row

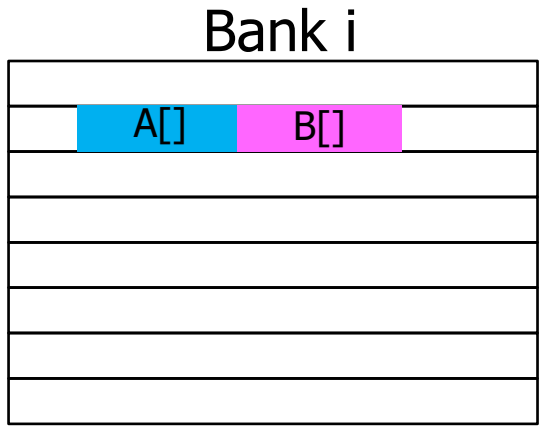



- The most desired memory access pattern
  - since it does not need additional ACT commands
- How to obtain such memory traffics in my program?

# Improving Row Buffer Locality in Our Program

- Idea: place highly correlated hot data on the same DRAM row
- Step 1: Profile # accesses/variable during program execution
- Step 2: For hot variables, profile access correlations
  - E.g., # accesses to two hot variables A and B during a time window T
- Step 3: Place highly correlated hot variables on the same DRAM row

```
int A[100], B[100];
…
A[i] = B[i] + …;
…
```




Both hot and highly correlated arrays are allocated to the same row

[Stevens, 2011]

# Step 3: Accessing DRAM (in Parallel)

SW code: a = x + 1;

CPU executes load instruction with VA(x)
*VA (PA) = virtual (physical) address

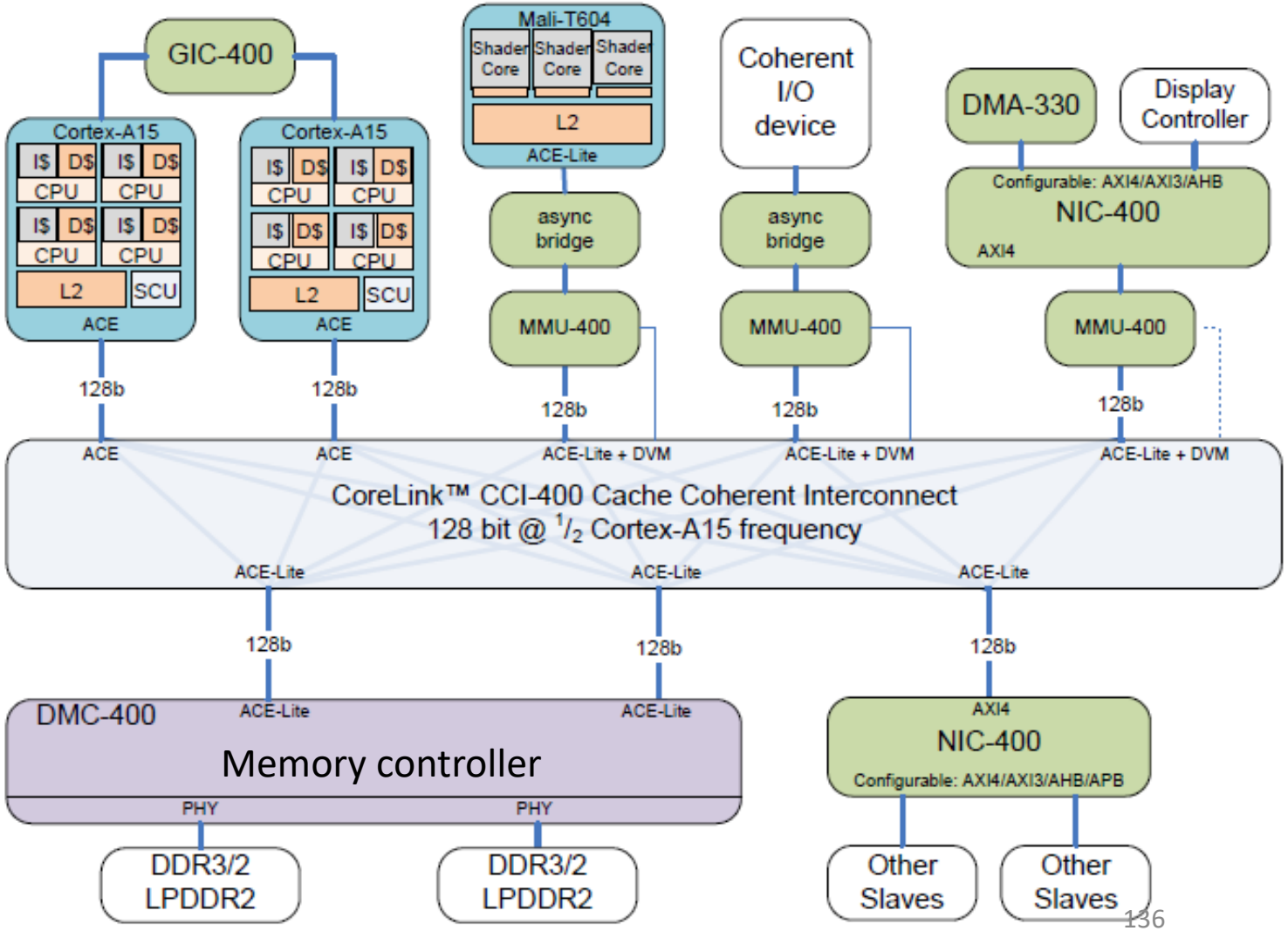
VA(x) → PA(x) translation on TLB

CPU sends to the interconnect
a read request for PA(x)

The read request arrives
at memory controller

Memory controller reads data@PA(x)
from DRAM

Memory controller sends the data
to CPU via the interconnect



[Stevens, 2011]

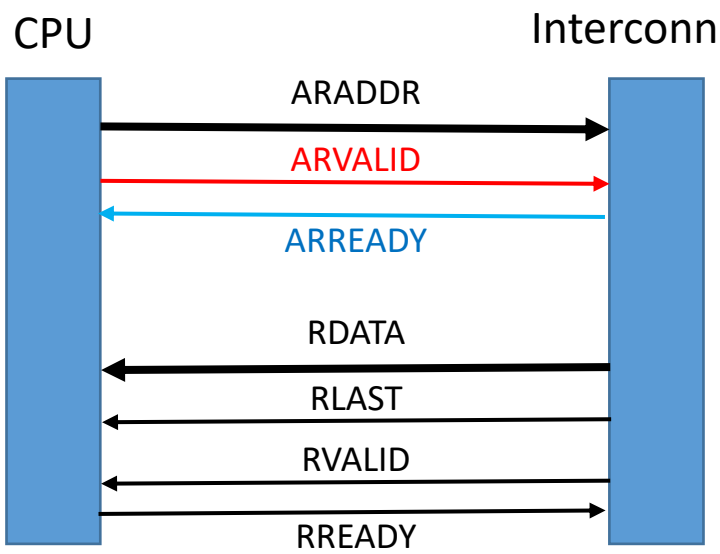
# Step 2: Sending Read Request via Interconnect

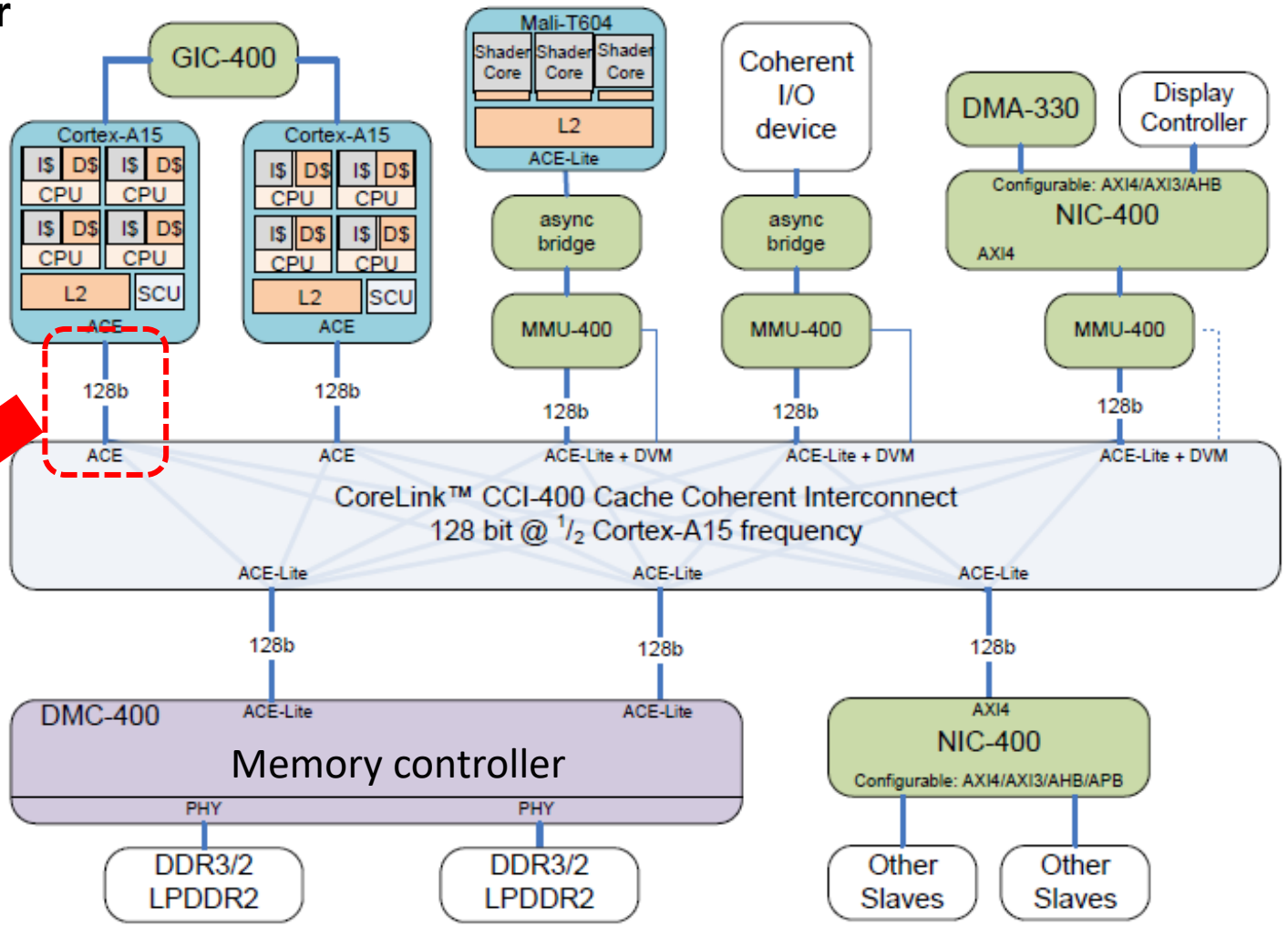
**Sending a read request to memory controller**

VA to PA translation on TLB

Send a read request on read address (AR) channel, i.e., ARVALID = '1'

Interconnect receives it, i.e., ARREADY = '1'



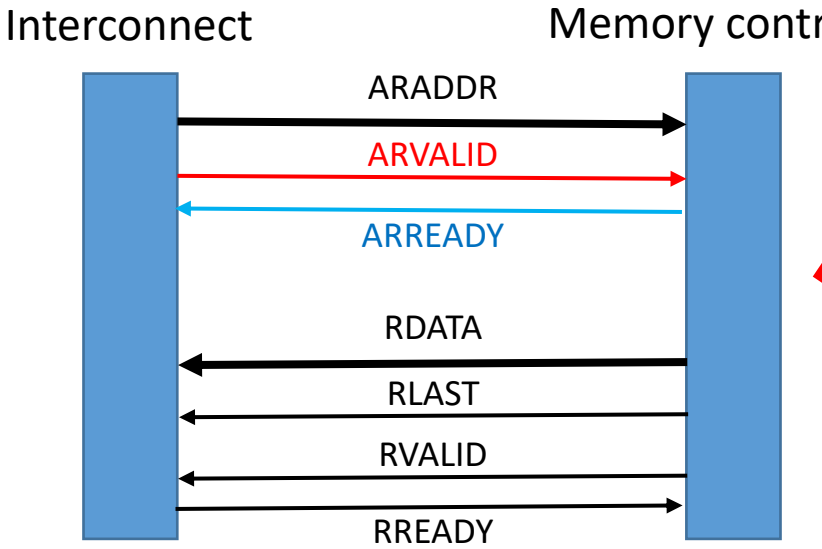
[Stevens, 2011]

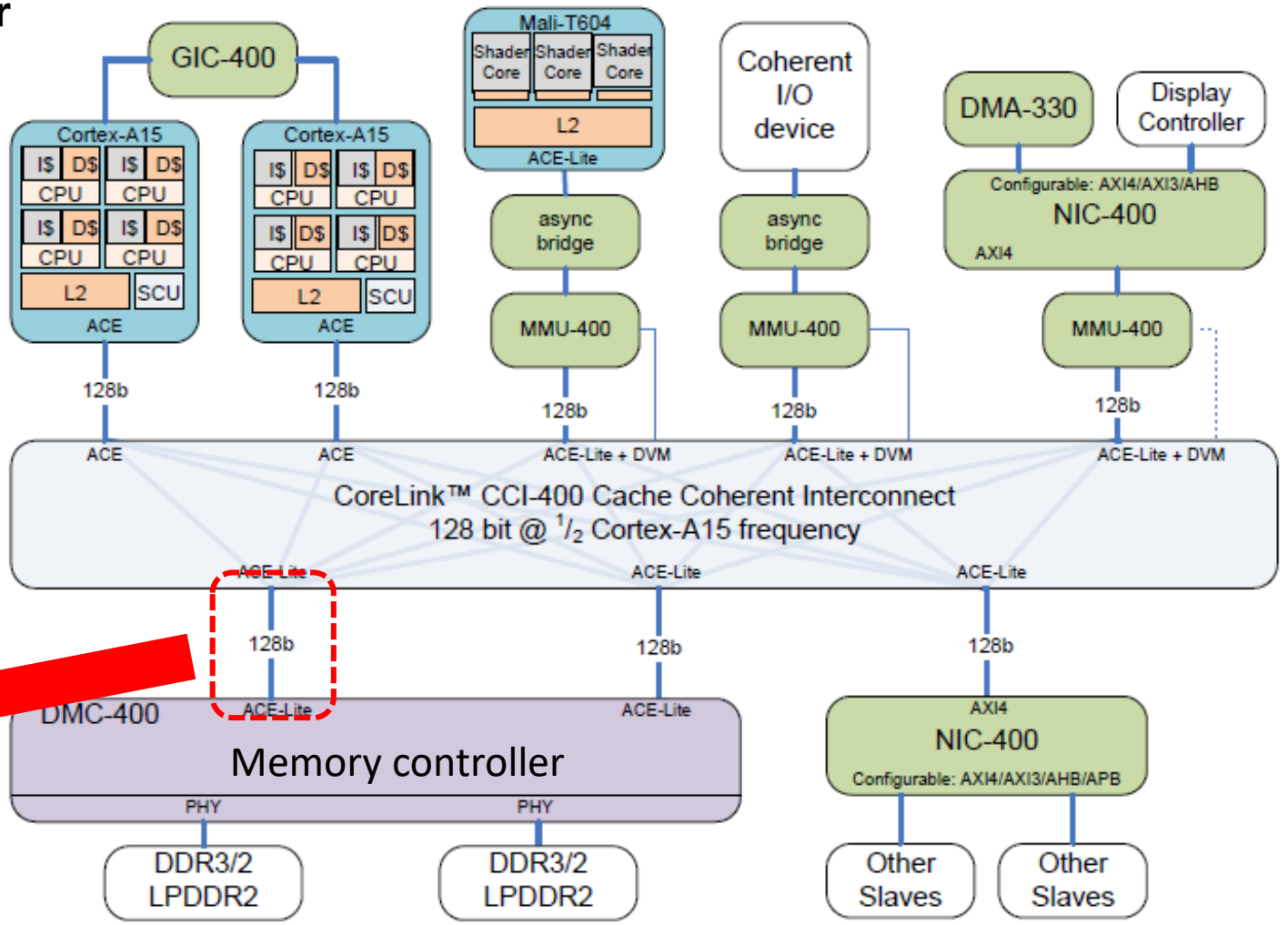
# Step 2: Sending Read Request to Memory Controller

**Sending a read request to memory controller**

Interconnect forwards the read request to the memory controller, i.e., ARVALID = '1'

Note #1: This happens on a different connection from CPU-Interconnect.
Note #2: Interconnect is the master in this communication



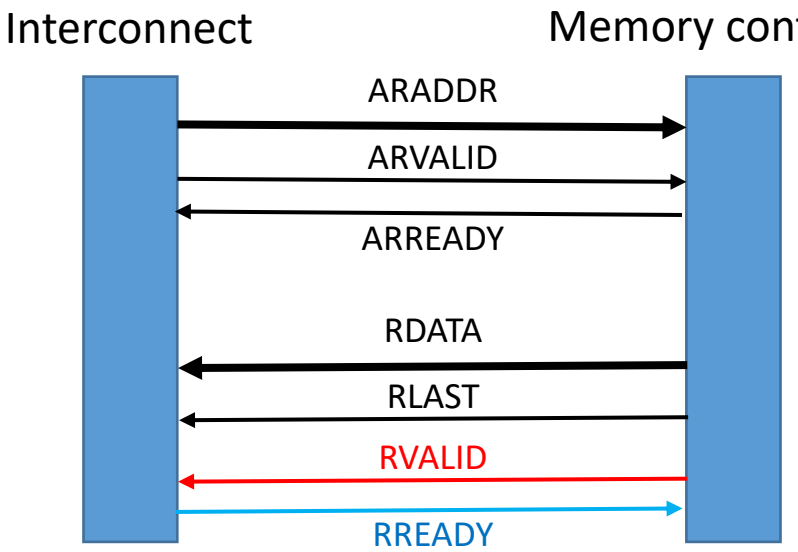
[Stevens, 2011]

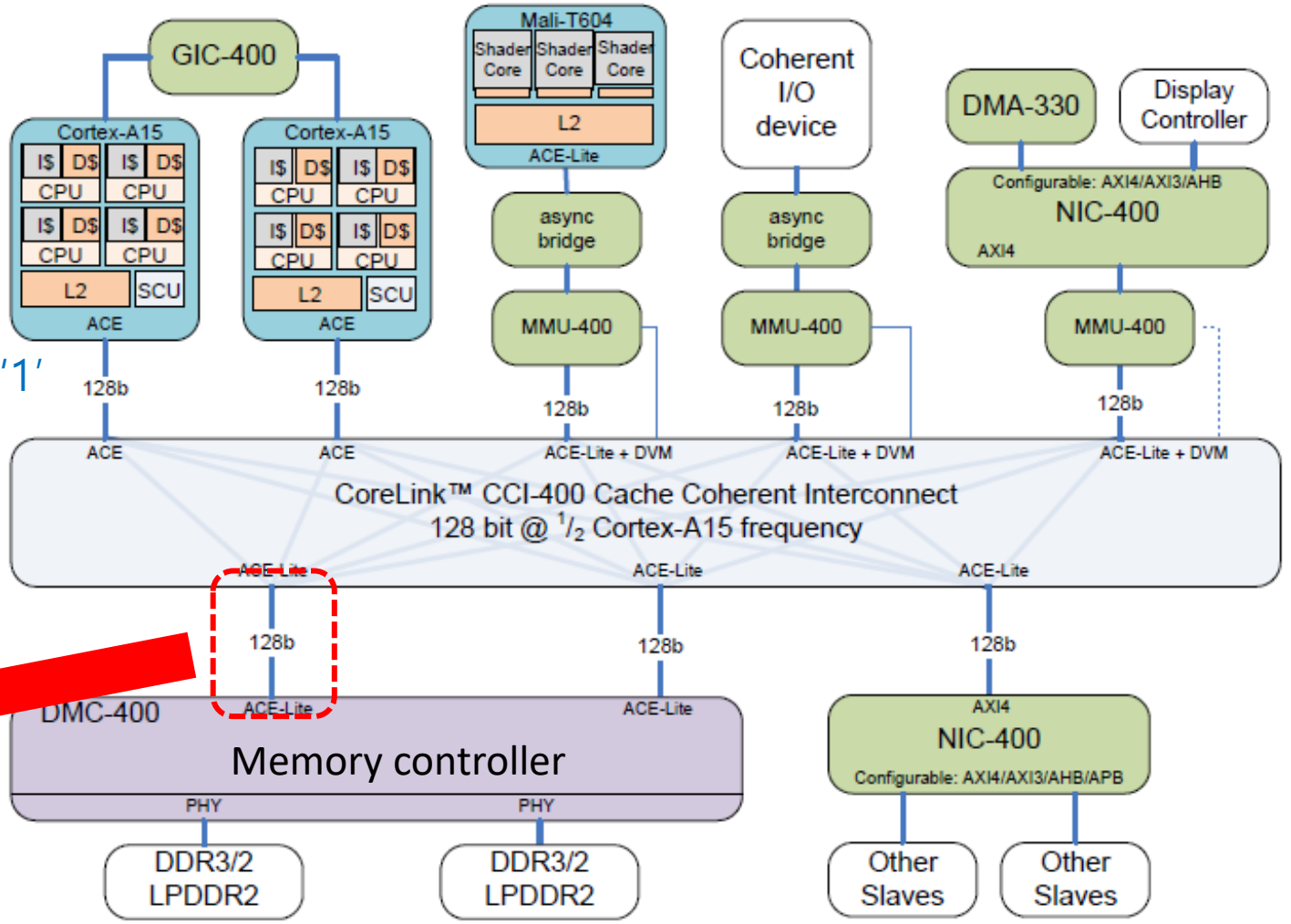
# Step 3: Sending Read Data to Interconnect

**Transferring data from memory controller to CPU via interconnect**

**Memory controller** sends the data to **Interconnect** on Read Data channel
i.e., RVALID = '1'

**Interconnect** receives the data, i.e., RREADY = '1'



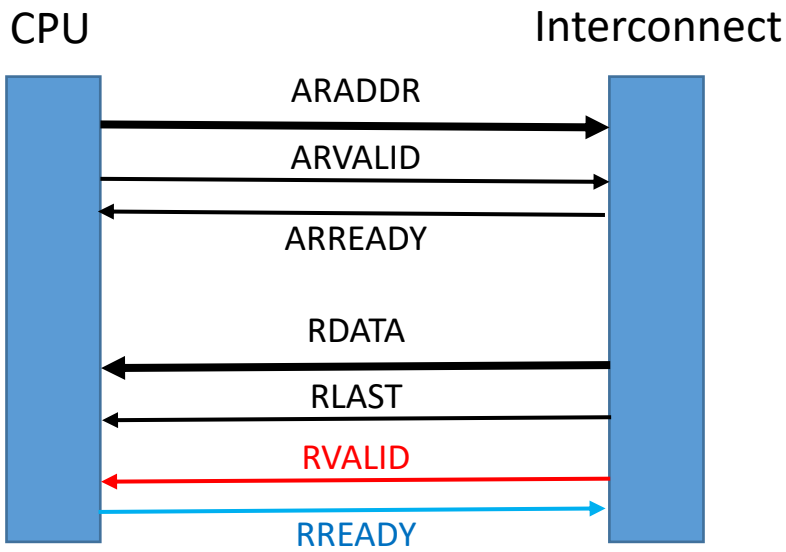
[Stevens, 2011]

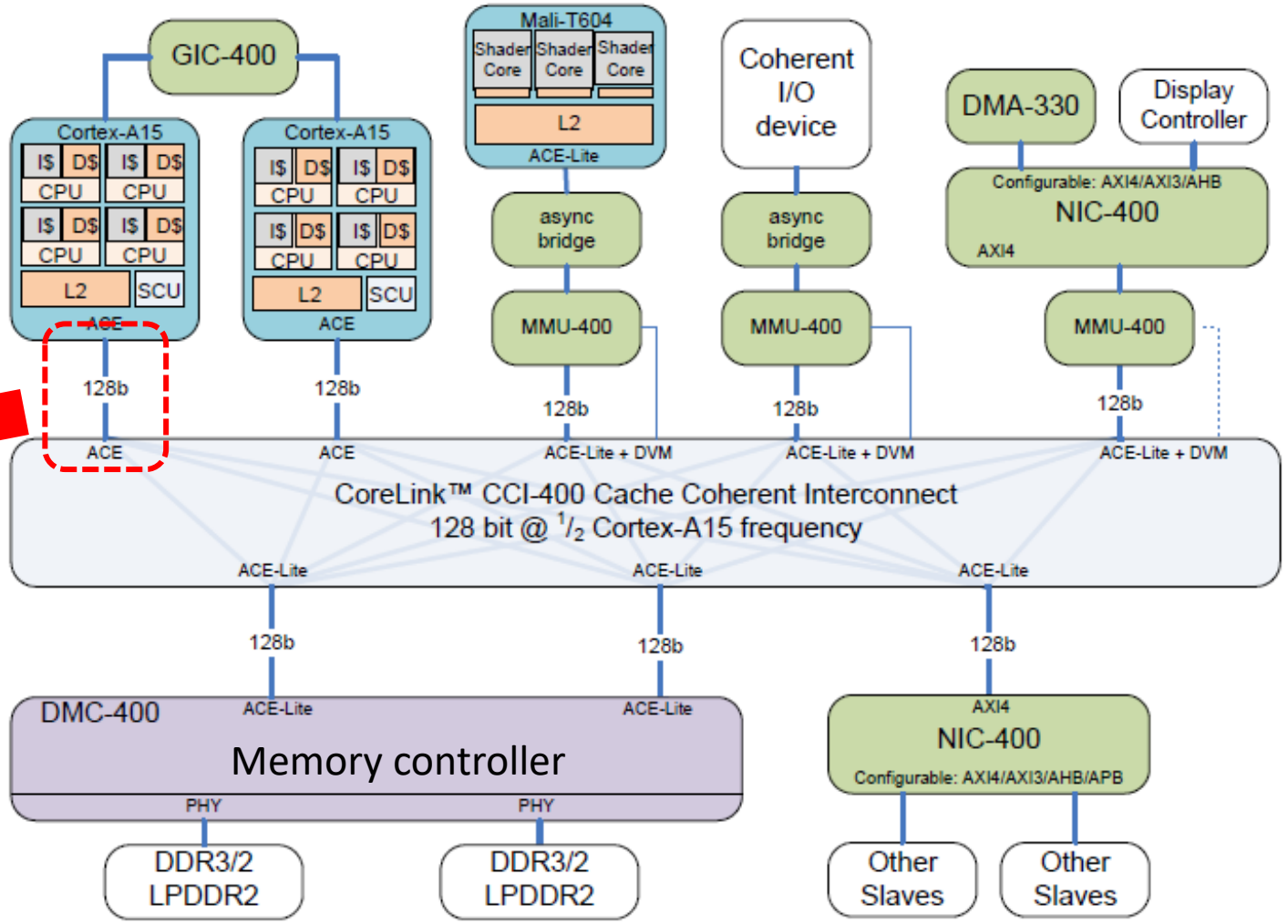
# Step 3: Sending Read Data to CPU

**Transferring data from memory controller to CPU via interconnect**

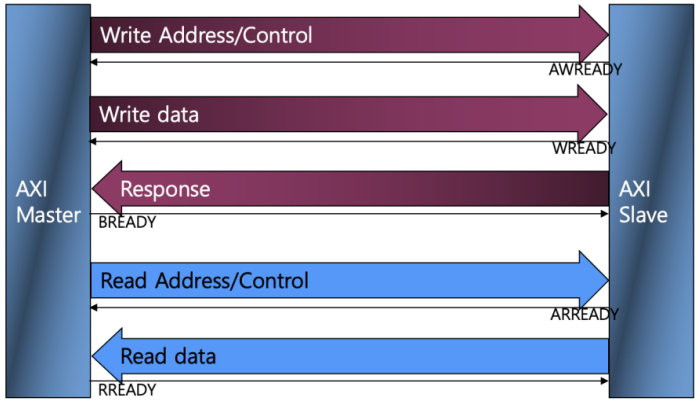
**Interconnect** sends the data to **CPU** on Read Data channel
i.e., RVALID = '1'

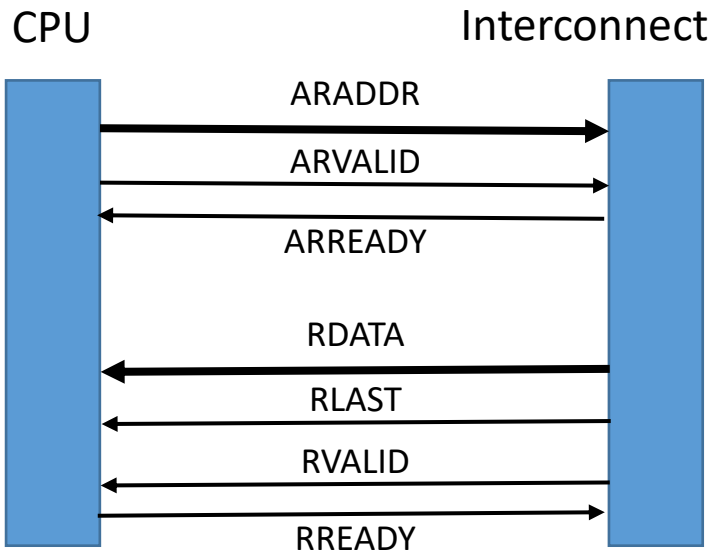
**CPU** receives the data, i.e., RREADY = '1'

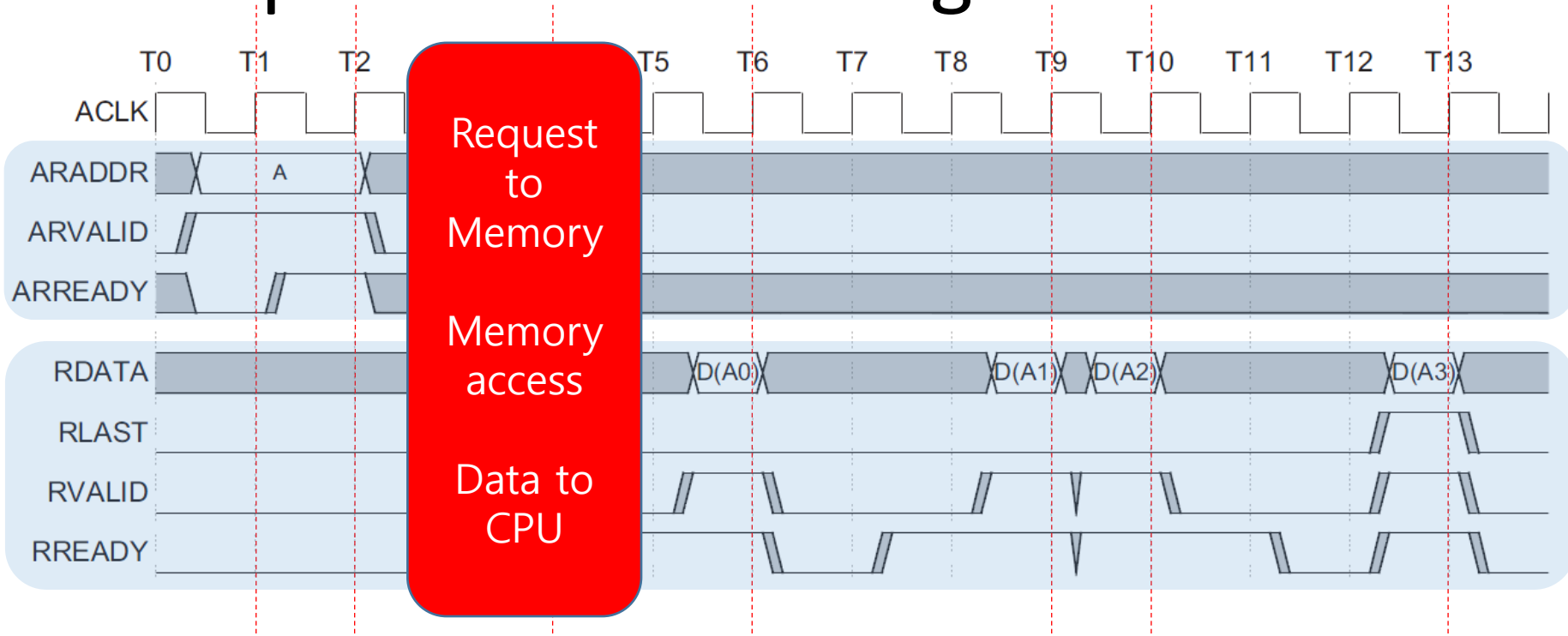


# From the Viewpoint of CPU: Sending a Read Request and Receiving the Data




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**

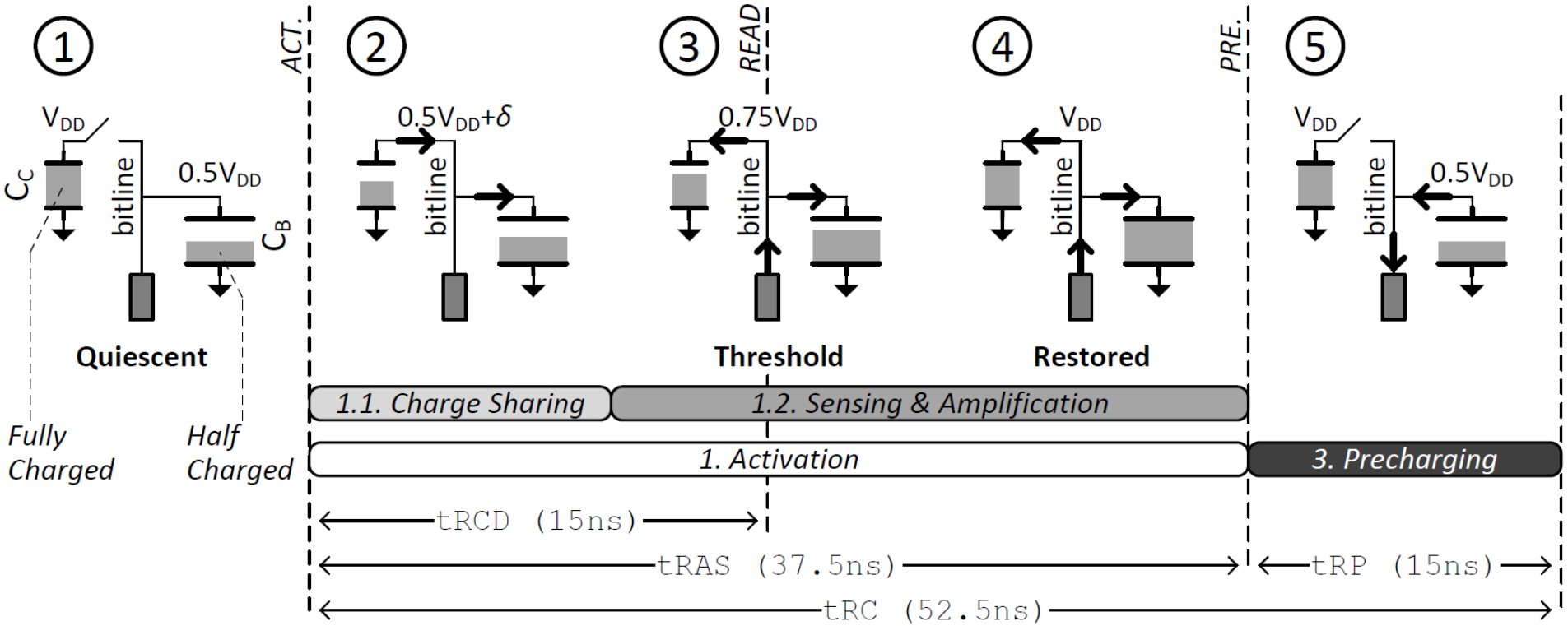
# Agenda

- DRAM architecture
- Memory access scheduling
- Refresh
- Row hammer
- Error correction code
- Summary

[Lee, 2013]

# Refresh = Activation + Precharge

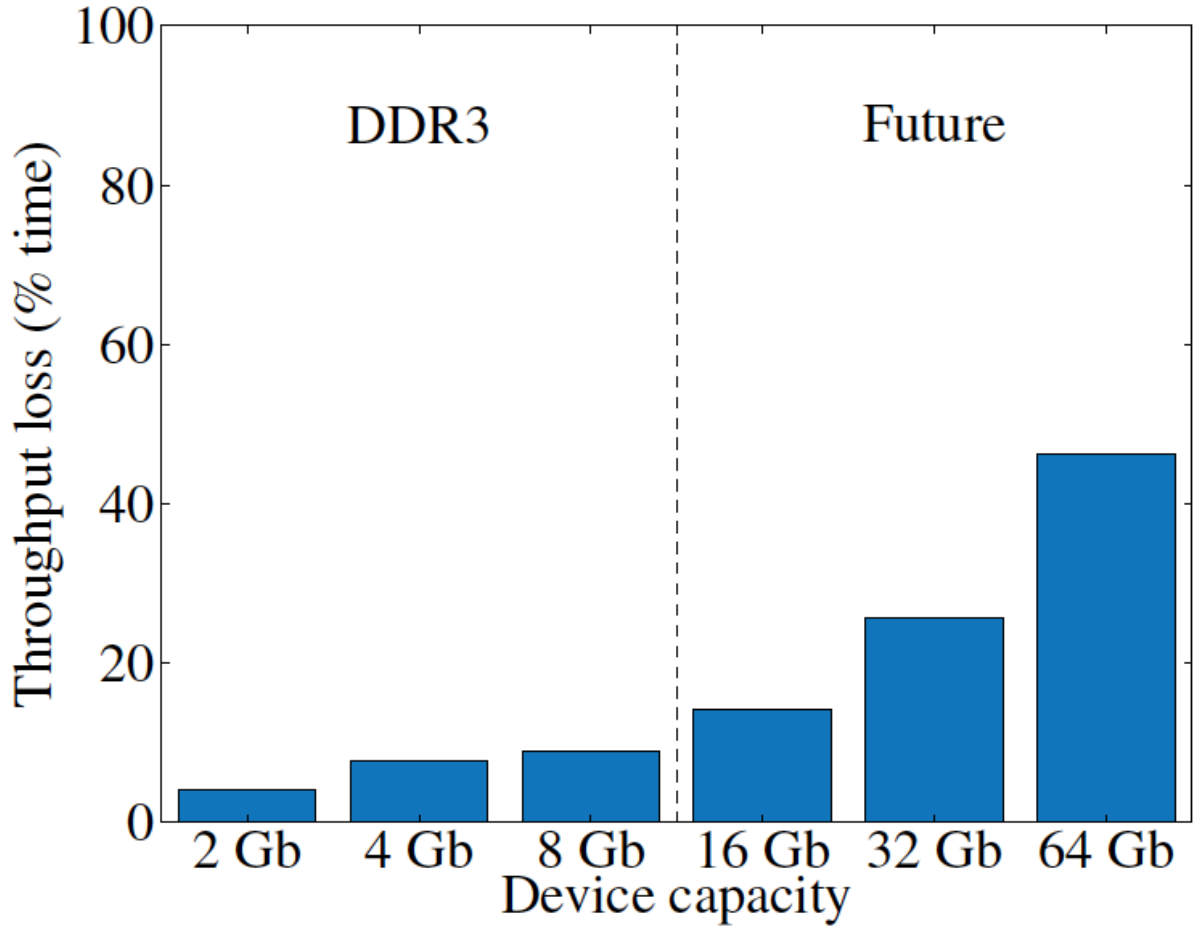
- Typically, each DRAM row is refreshed every 64ms
- In case of hot temperature, every 32ms



[Mutlu, 2012]

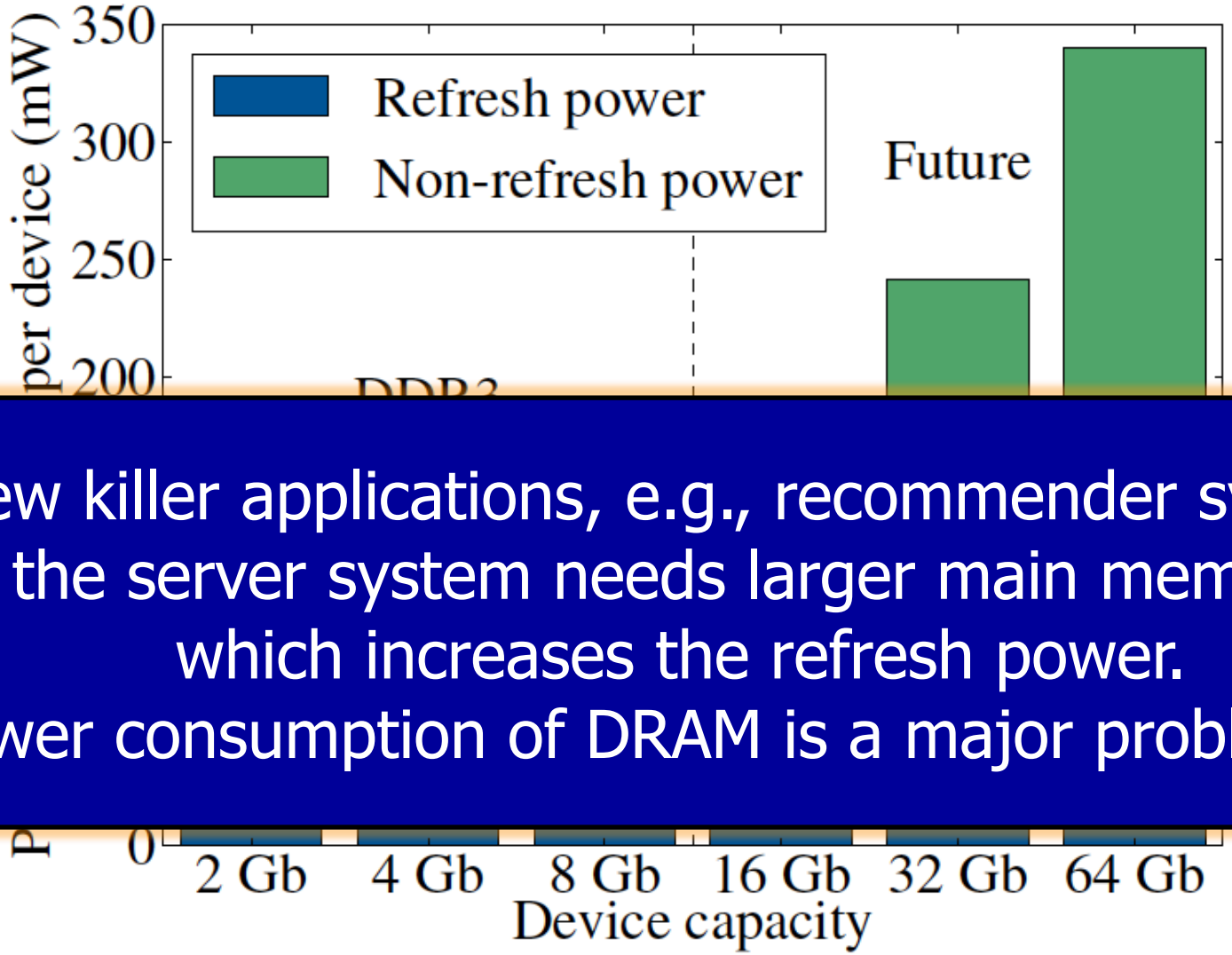
# Refresh Overhead: Performance

- DRAM bank cannot be accessed during refresh operation



[Mutlu, 2012]

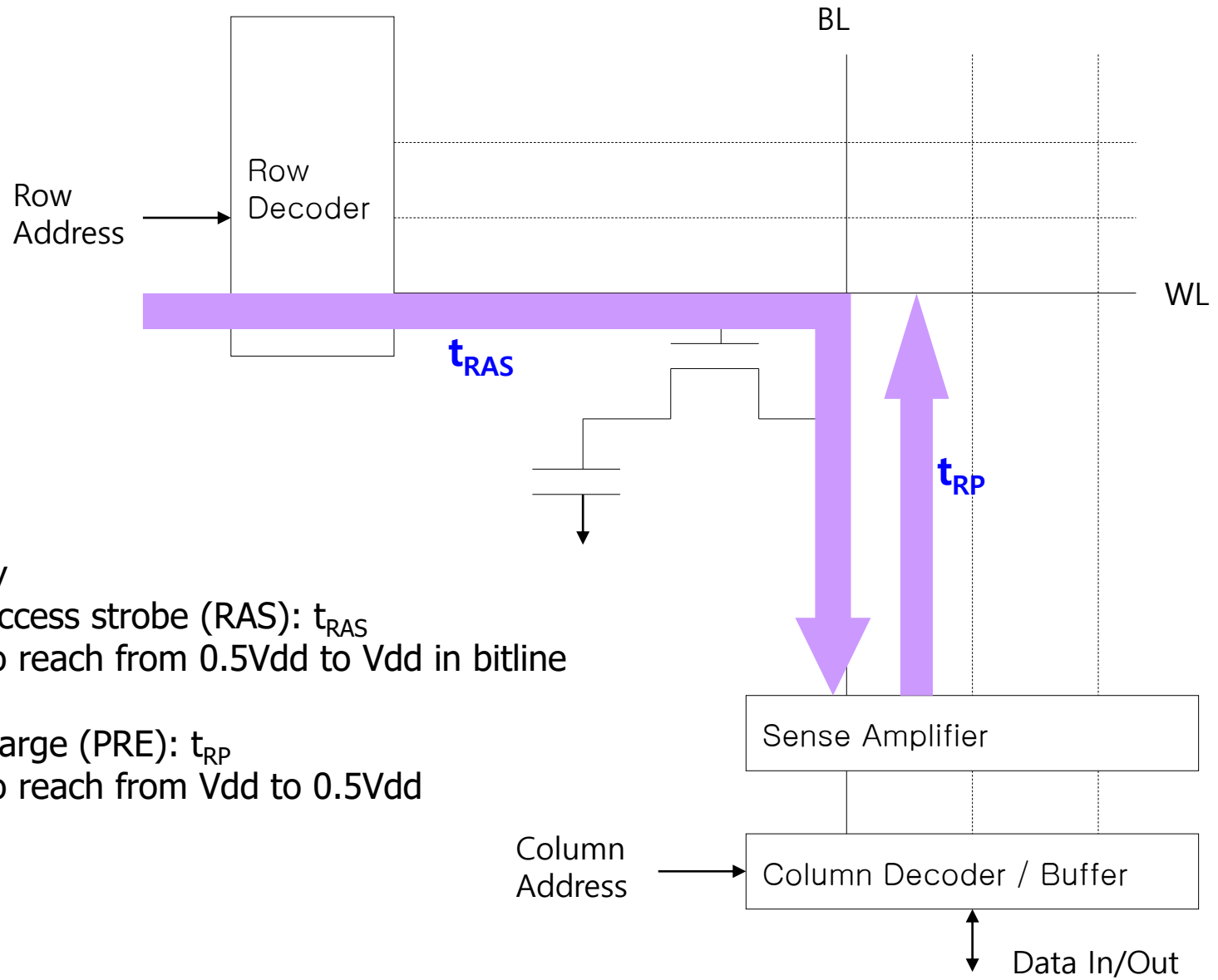
# Refresh Overhead: Power




Due to new killer applications, e.g., recommender system and ML training, the server system needs larger main memory capacity, which increases the refresh power.
Now, power consumption of DRAM is a major problem in server.

[B. Jacobs, 2008]

# Refresh in DRAM



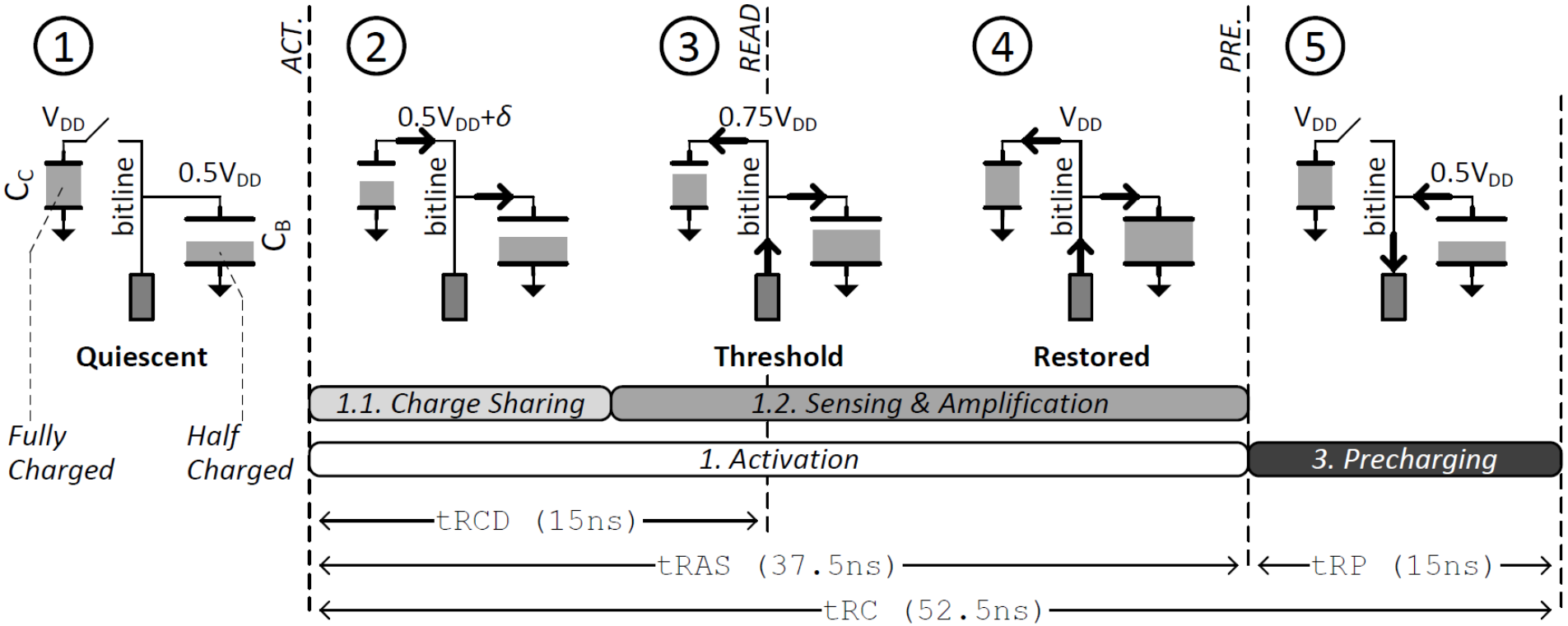

Latency
-Row access strobe (RAS): $t_{RAS}$
Time to reach from 0.5Vdd to Vdd in bitline

- Precharge (PRE): $t_{RP}$
Time to reach from Vdd to 0.5Vdd

[Lee, 2013]

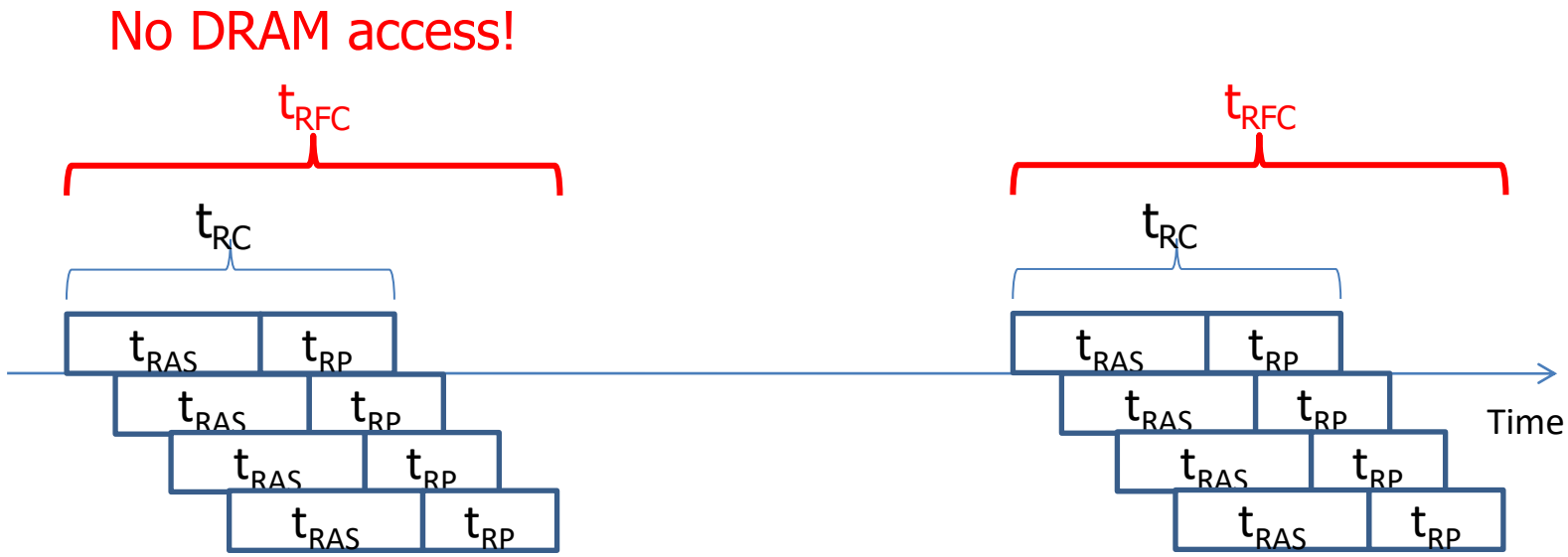
# Refresh = Activation + Precharge

- Typically, each DRAM row is refreshed every 64ms
- In case of hot temperature, every 32ms



# Refresh

- During active period, the memory controller issues auto-refresh command every 7.8μs (=8192 auto-refreshes per 64ms) to DRAM
- On receiving an auto-refresh, DRAM performs refreshes for 1~8 rows (depending on capacity)
  - $t_{RFC}$ = refresh cycle time per auto-refresh command



# Refresh Overhead: Runtime Problems and Solutions

- Problems
  - Performance and energy overhead
    - As the memory capacity increases, larger overhead
  - Worst case memory access latency

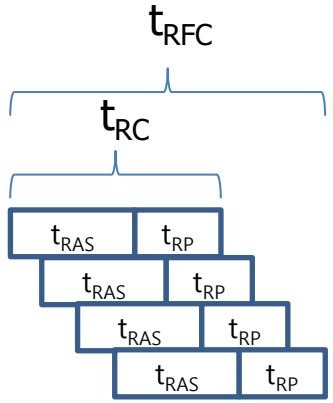
- Solutions
  - Adjusting the issue timing of auto-refresh commands
  - Per-bank refresh: perform memory accesses and refresh in parallel
  - Fine-grained refresh: adjust auto-refresh rate to reduce the worst-case design due to refresh

[B. Jacobs, 2008]

# Refresh Overhead in Performance

- A case study: 2Gb DDR2 with 8 banks
- Refresh overhead calculation
  - 8192 auto-refresh commands / 64 ms
  - 8192*197.5ns / 64ms = 2.5% of system runtime
  - 4 rows are refreshed on each auto-refresh command

| Chip capacity | # banks | # rows | tRFC |
|---|---|---|---|
| 256Mb | 4 | 8,192 | 75ns |
| 512Mb | 4 | 16,384 | 105ns |
| 1Gb | 8 | 16,384 | 127.5ns |
| 2Gb | 8 | 32,768 | 197.5ns |
| 4Gb | 8 | 65,536 | 327.5ns |

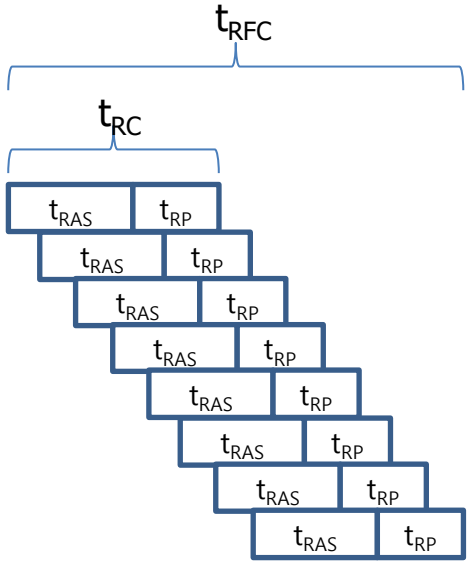


[B. Jacobs, 2008]

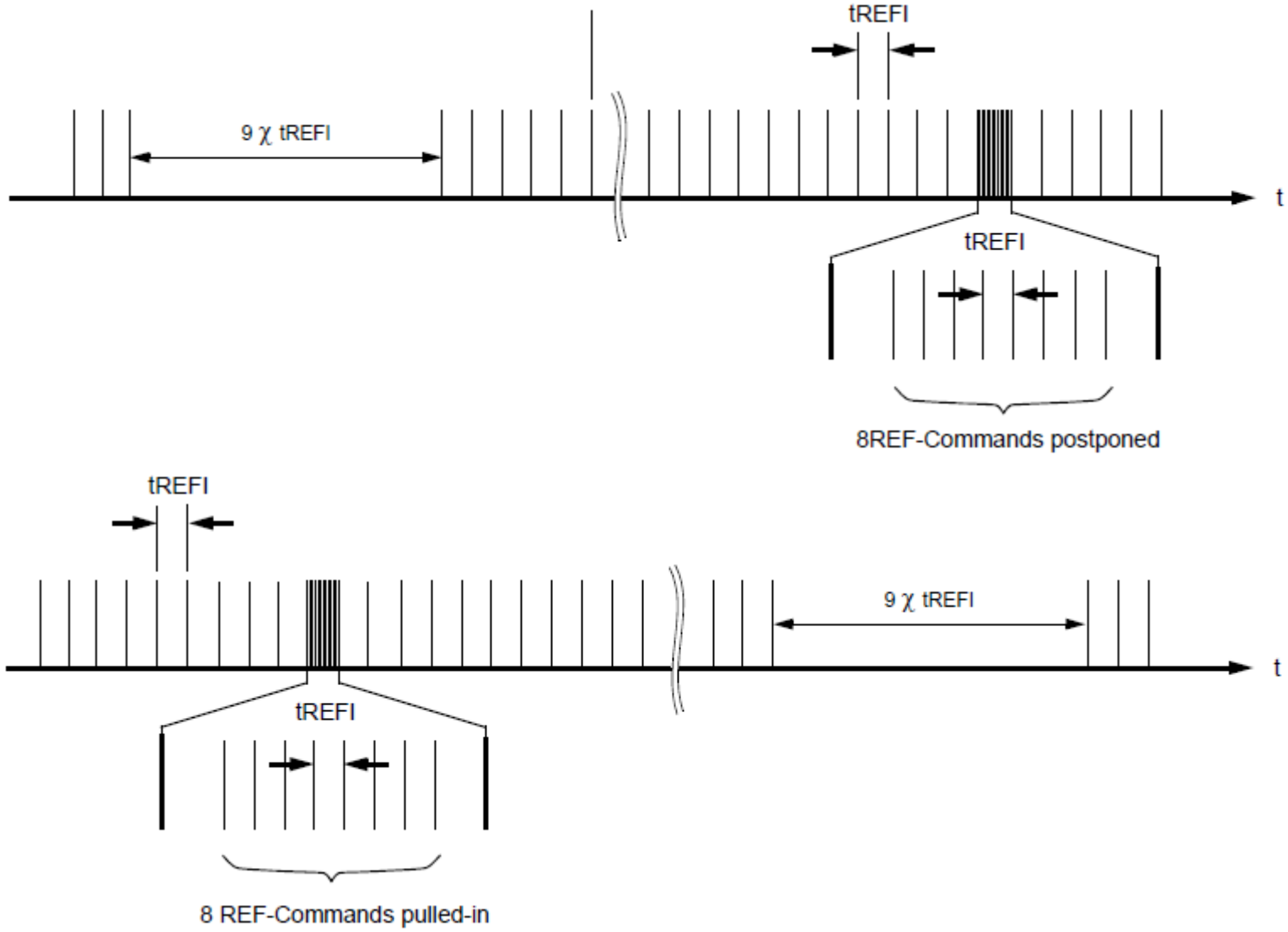
# Refresh Overhead in Performance

- A case study: 4Gb DDR2 with 8 banks
- Refresh overhead calculation
  - 8192 auto-refresh commands / 64 ms
  - 8192*327.5ns / 64ms = 4.2% of system runtime!
  - 8 rows are refreshed on each auto-refresh command due to 2x larger capacity

| Chip capacity | # banks | # rows | tRFC |
|---|---|---|---|
| 256Mb | 4 | 8,192 | 75ns |
| 512Mb | 4 | 16,384 | 105ns |
| 1Gb | 8 | 16,384 | 127.5ns |
| 2Gb | 8 | 32,768 | 197.5ns |
| 4Gb | 8 | 65,536 | 327.5ns |



# Postponing or Pulling in Refresh commands



[JEDEC 2014]

# Per-Bank Refresh to Overlap Refresh and Memory Access

## Per Bank Refresh Control (Direct Addressing)

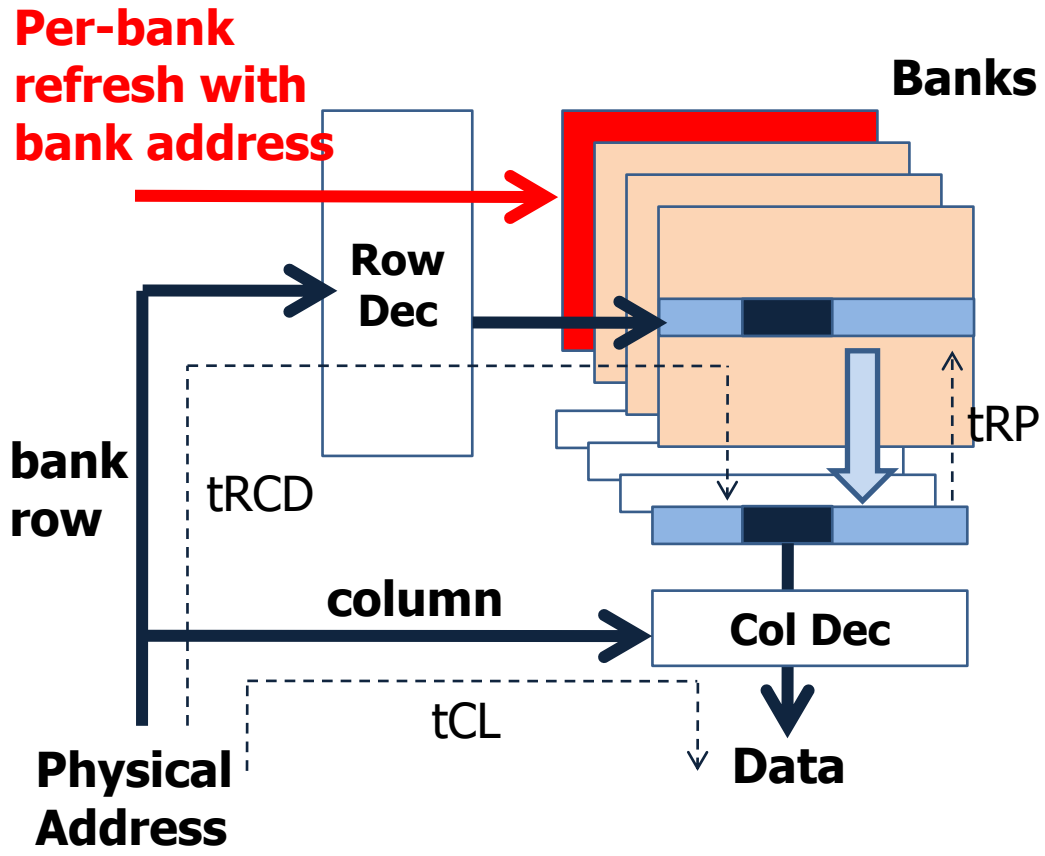
- *New to LPDDR4:*
  - *Enhanced per bank refresh control for controller scheduling flexibility*

| SDRAM Command | CS | CA0 | CA1 | CA2 | CA3 | CA4 | CA5 |
|---|---|---|---|---|---|---|---|
| Refresh (REF) (Per Bank, All Bank) | H | L | L | L | H | L | AB |
| | L | BA0 | BA1 | BA2 | V | V | V |

LPDDR3
: PB refresh w/o bank address
(Fixed bank sequence)
- round robin order (0-1-2-3-4-5-6-7-0-1..)

LPDDR4
: PB refresh w/ bank address
- User to select bank address
- Can be issued in any bank order

It is illegal to send a per-bank REFRESH command to the same bank unless all eight banks have been refreshed using the per-bank REFRESH command.



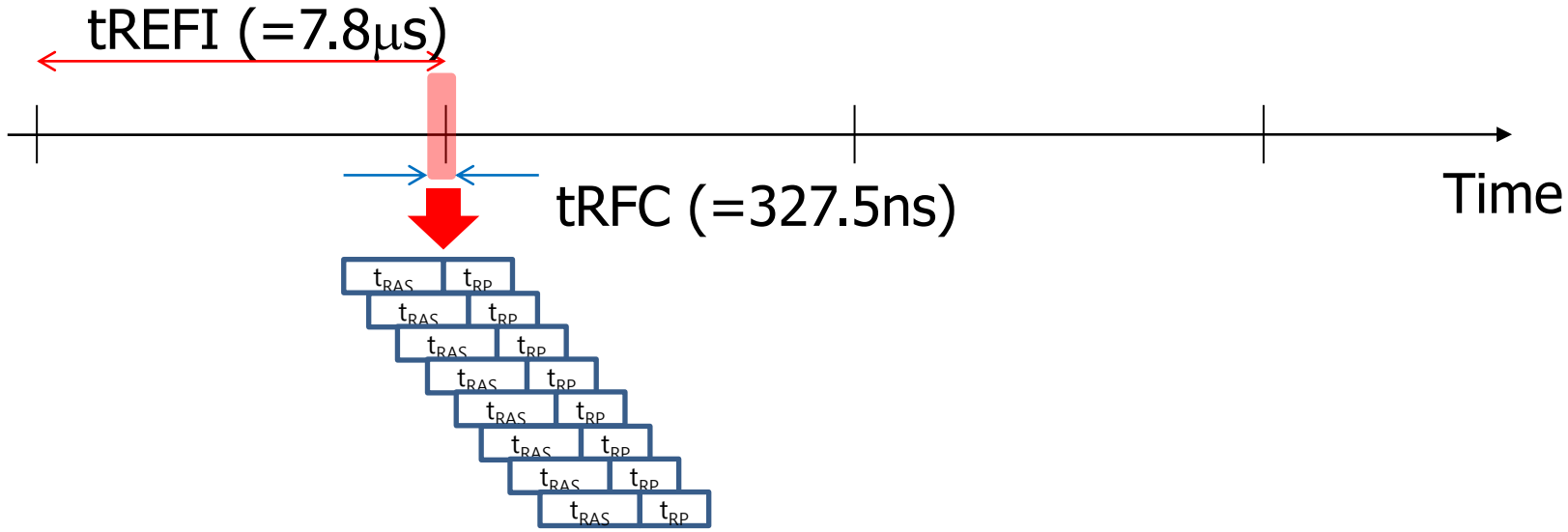
# Refresh Overhead: Runtime Problems and Solutions

- Problems
  - Performance and energy overhead
    - As the memory capacity increases, larger overhead
  - Worst case memory access latency

- Solutions
  - Adjusting the issue timing of auto-refresh commands
  - Per-bank refresh: perform memory accesses and refresh in parallel
  - Fine-grained refresh: adjust auto-refresh rate to reduce the worst-case design due to refresh

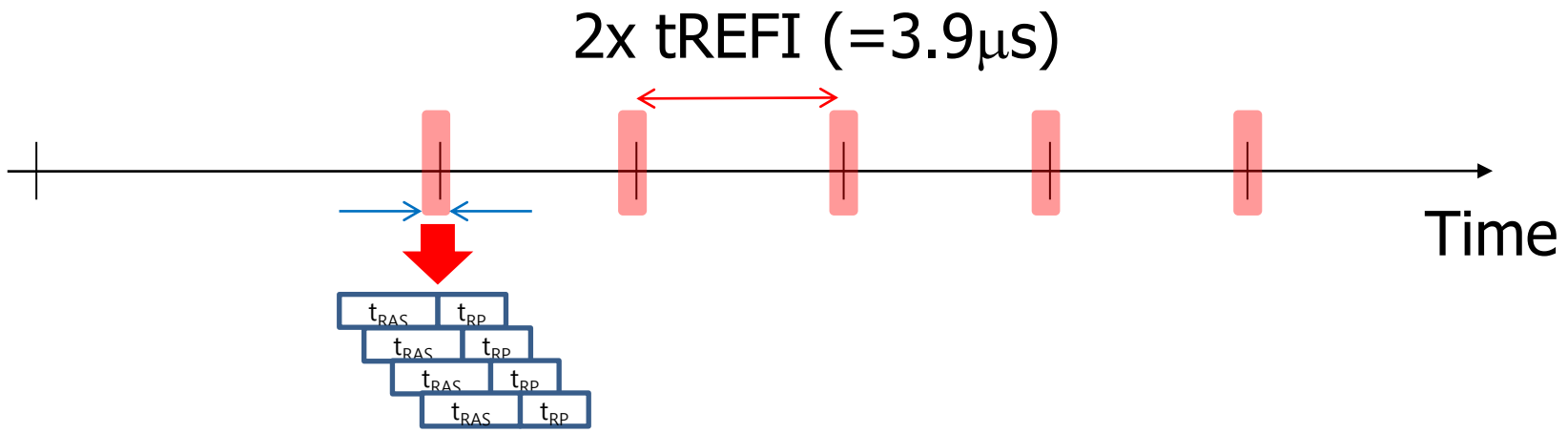
# Worst-Case Memory Access Latency

- Every 64ms, 8192 auto-refresh commands are issued by the memory controller to DRAM
  - An auto-refresh command every 7.8μs
  - During tRFC, DRAM cannot be accessed
- The worst case latency occurs when a read request arrives just after an auto-refresh command is issued



# Fine-Grained Auto-Refresh to Reduce Worst-Case Memory Access Latency

- LPDDR4 allows us to issue 2x or 4x more auto-refreshes per 64ms
- The worst case latency is reduced since each fine-grained auto-refresh has smaller tRFC than the original one (327.5ns)




Reduced tRFC (=197.5ns)

[JEDEC 2014]

# Flexibility in Refresh Control

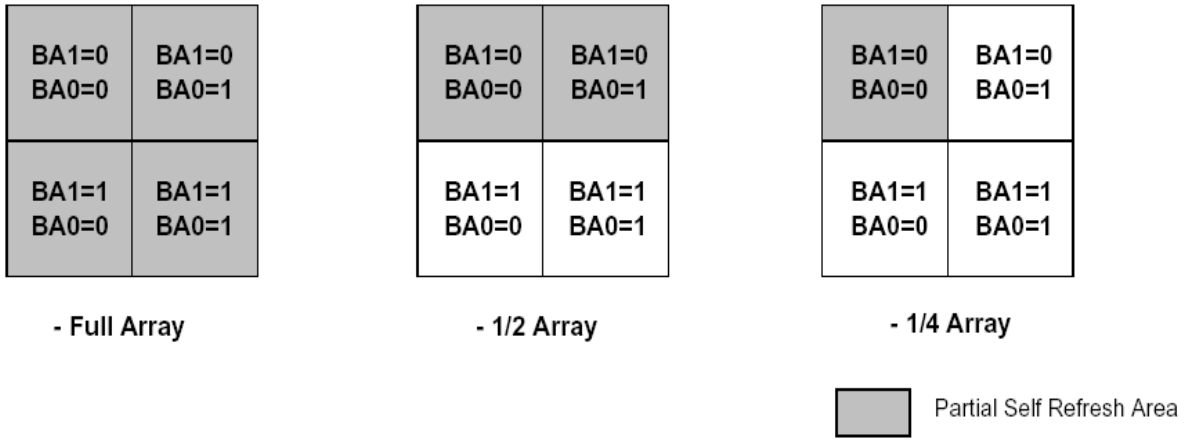
- Basic refresh control rule in LPDDR3 & LPDDR4
  - All bank refresh command in every tREFI interval.
  - Max 8 refresh can be postponed
  - Max 8 refresh can be issued in advance
  - At any given time, a maximum of 16 REF commands can be issued within 2 x tREFI x refresh rate multiplier

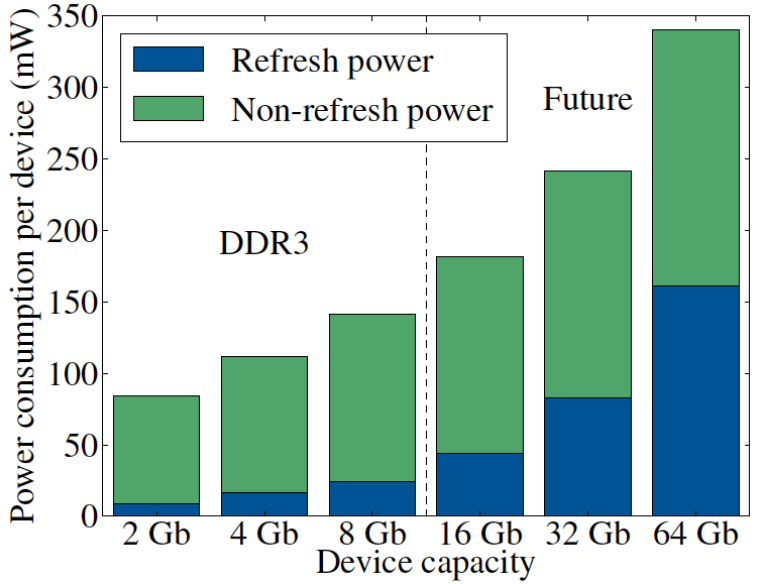
**Table 24 - Legacy Refresh Command Timing Constraints**

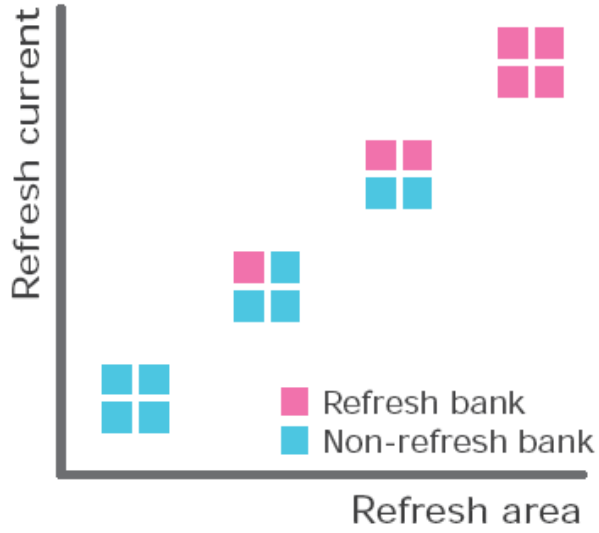
| MR4 OP[2:0] | Refresh rate | Max. No. of pulled-in or postponed REFab | Max. Interval between two REFab | Max. No. of REFab within max(2xtREFI x refresh rate multiplier, 16xtRFC) | Per-bank Refresh |
|---|---|---|---|---|---|
| 000B | Low Temp. Limit | N/A | N/A | N/A | N/A |
| 001B | 4x tREFI | 8 | 9 x 4 x tREFI | 16 | 1/8 of REFab |
| 010B | 2x tREFI | 8 | 9 x 2 x tREFI | 16 | 1/8 of REFab |
| 011B | 1x tREFI | 8 | 9 x tREFI | 16 | 1/8 of REFab |
| 100B | 0.5x tREFI | 8 | 9 x 0.5 x tREFI | 16 | 1/8 of REFab |
| 101B | 0.25x tREFI | 8 | 9 x 0.25 x tREFI | 16 | 1/8 of REFab |
| 110B | 0.25x tREFI | 8 | 9 x 0.25 x tREFI | 16 | 1/8 of REFab |
| 111B | High Temp. Limit | N/A | N/A | N/A | N/A |

# Reducing Power Overhead of Self Refresh

- During system idle time, DRAM refreshes itself, which is called self refresh
- Partial array self refresh (PASR) was introduced to reduce self-refresh power
  - By refreshing only the power-on bank while not refreshing the other banks
- In reality, no real design adopting this due to data migration cost



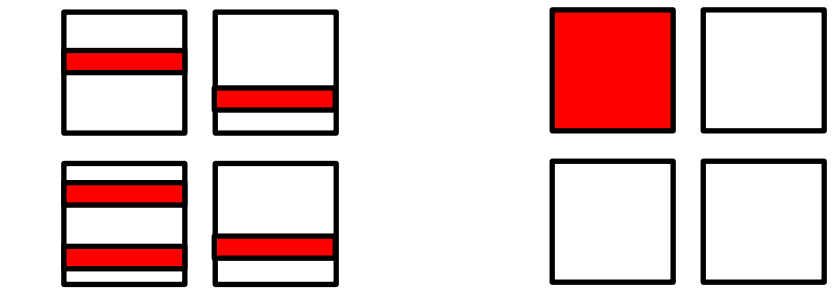




[Elpida]

# Better Solution than PASR?

- Currently, fine-grained self refresh is used without requiring data migration
  - Operating system (designer) needs to give the address information of power-on data to DRAM
  - DRAM refreshes only those data during self refresh
  - Adopted in current smartphones



Fine-grained SR



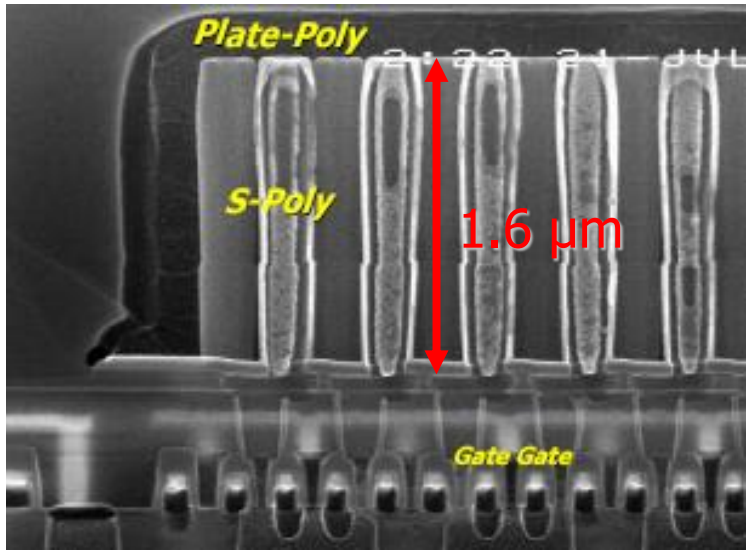
PASR

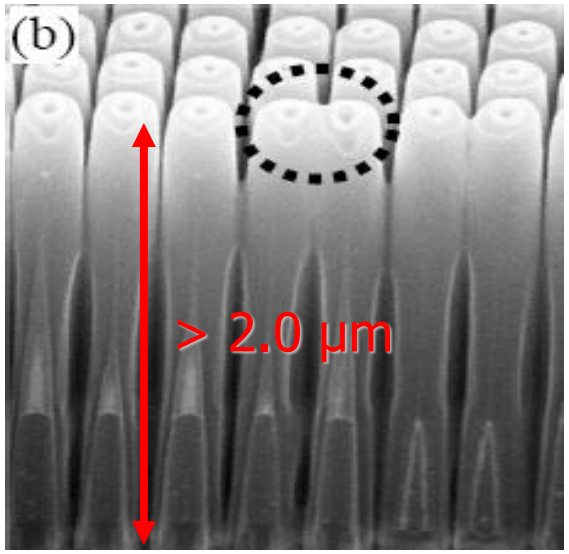
# Agenda

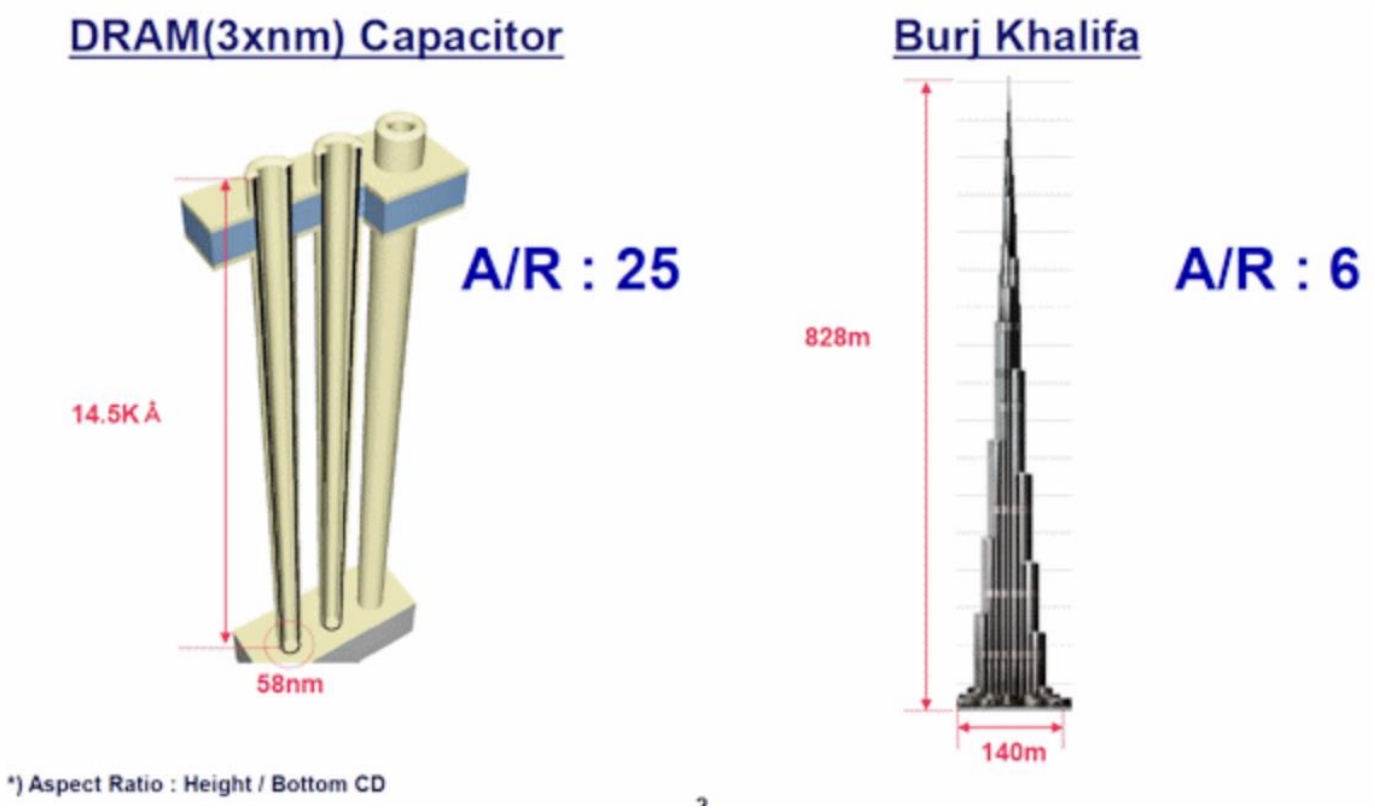
- DRAM architecture
- Memory access scheduling
- Refresh
- Row hammer
- Error correction code
- Summary

[https://denalimemoryreport.wordpress.com/2012/08/27/want-to-know-why-sk-hynix-is-placing-its-bets-on-three-different-alternatives-to-dram-and-flash/]

# Very Tall Capacitor in DRAM Cell is Vulnerable to Faults and Errors



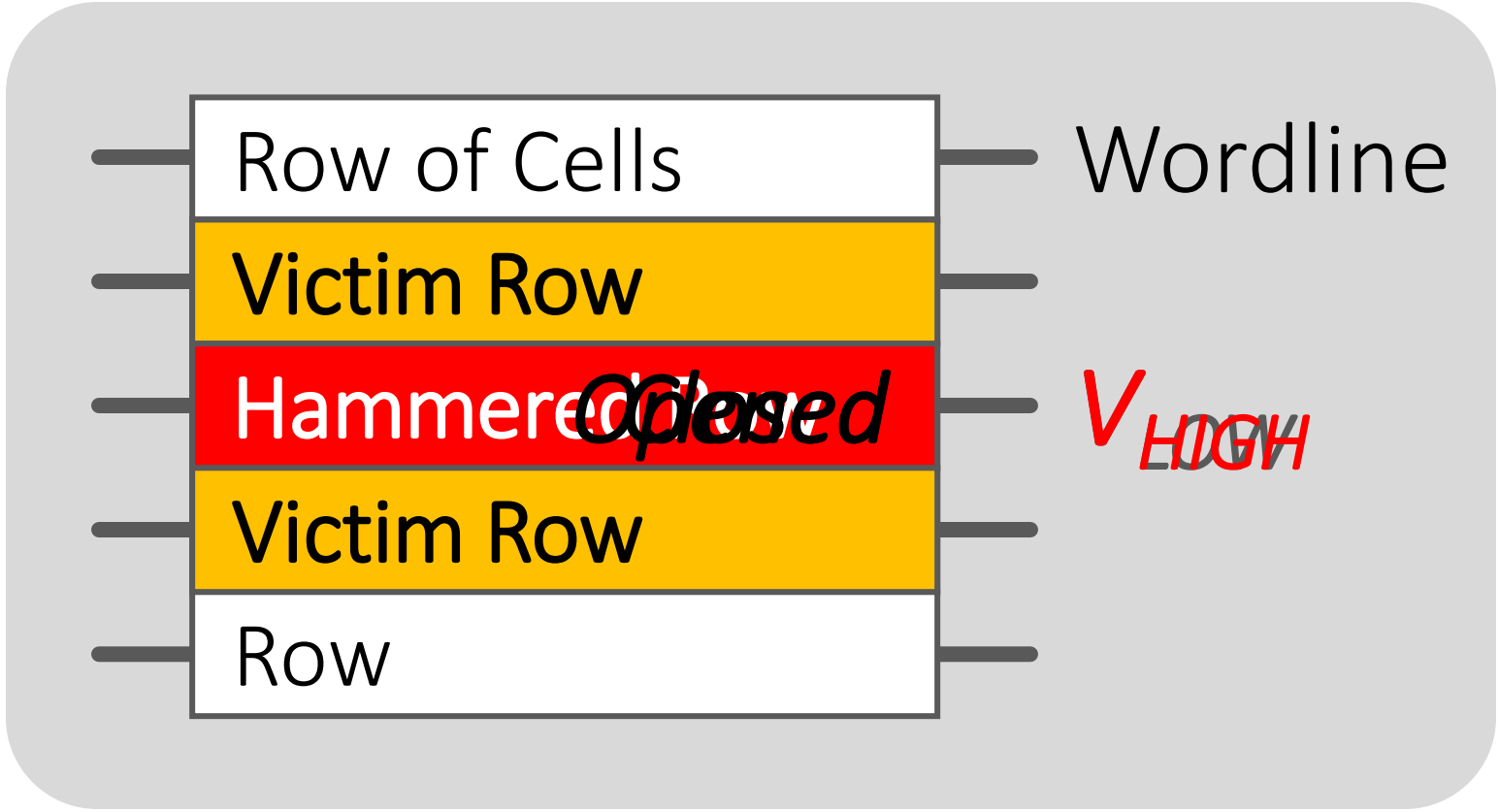




In order to realize the required capacitance while reducing the cell area, tall capacitor (and very dielectric material) is mandatory

[Onur Mutlu]

# Modern DRAM is Prone to Disturbance Errors

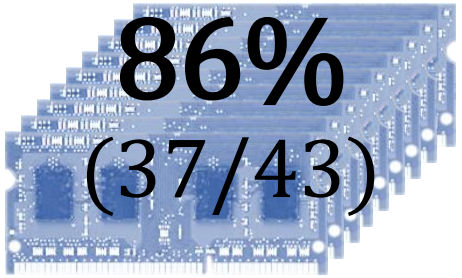
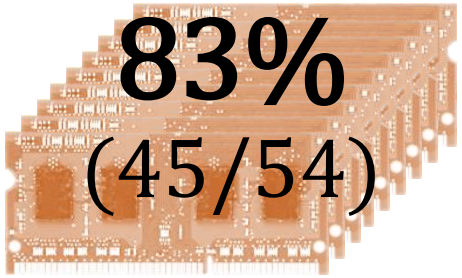
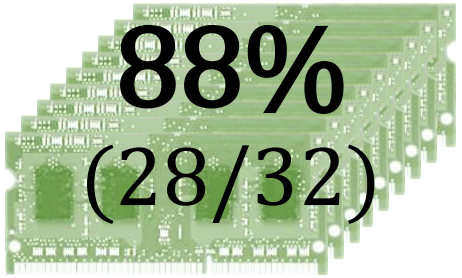



Repeatedly opening and closing a row enough times within a refresh interval induces disturbance errors in adjacent rows in most real DRAM chips you can buy today

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors, (Kim et al., ISCA 2014)

[Onur Mutlu]

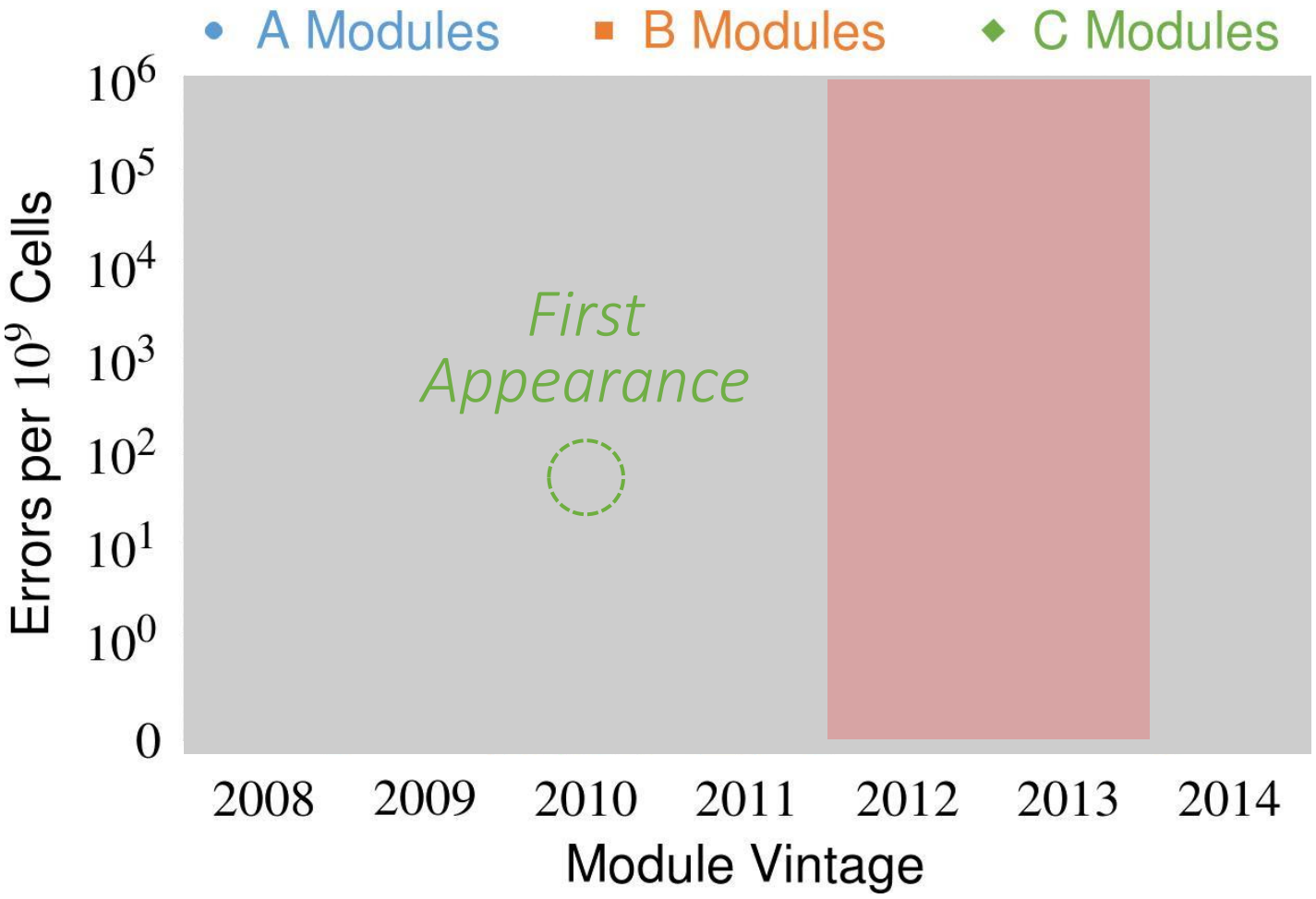
# Most DRAM Modules Are Vulnerable

| A company | B company | C company |
|---|---|---|
|  |  |  |
| **86%** (37/43) | **83%** (45/54) | **88%** (28/32) |
| Up to **$1.0\times10^7$** errors | Up to **$2.7\times10^6$** errors | Up to **$3.3\times10^5$** errors |

[Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors](), (Kim et al., ISCA 2014)

[Onur Mutlu]

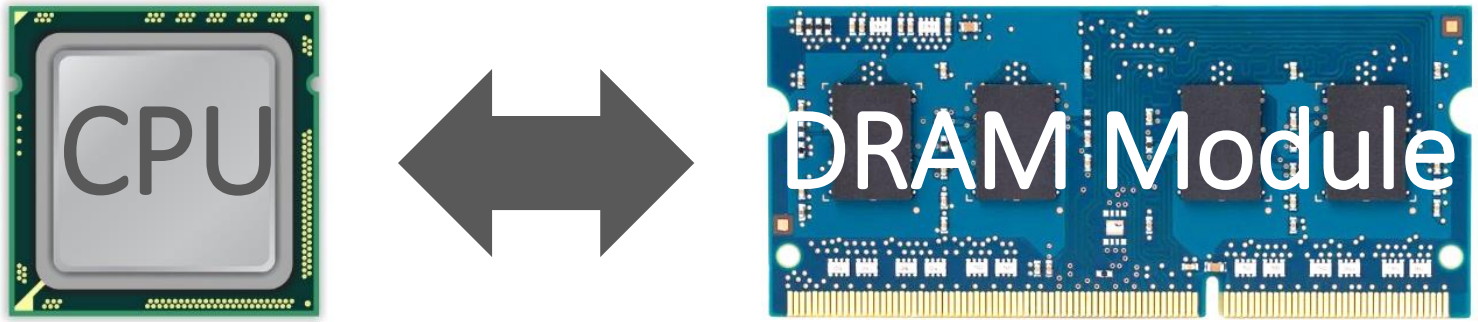
# Recent DRAM Is More Vulnerable



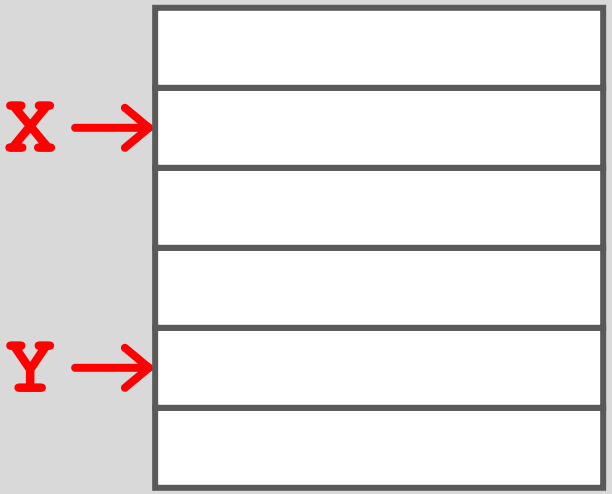

*All modules from 2012–2013 are vulnerable*

[Onur Mutlu]

# A Simple Program Can Induce Many Errors



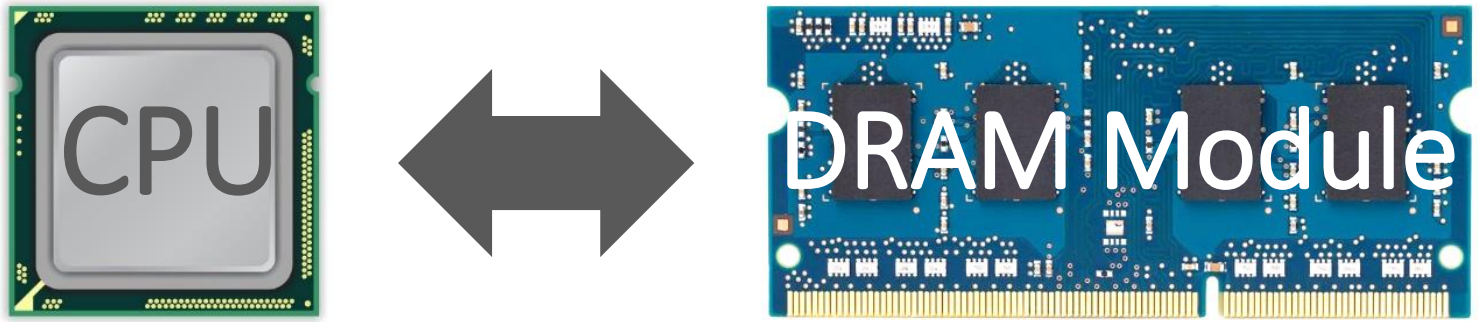

```
loop:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  mfence
  jmp loop
```




Download from: https://github.com/CMU-SAFARI/rowhammer

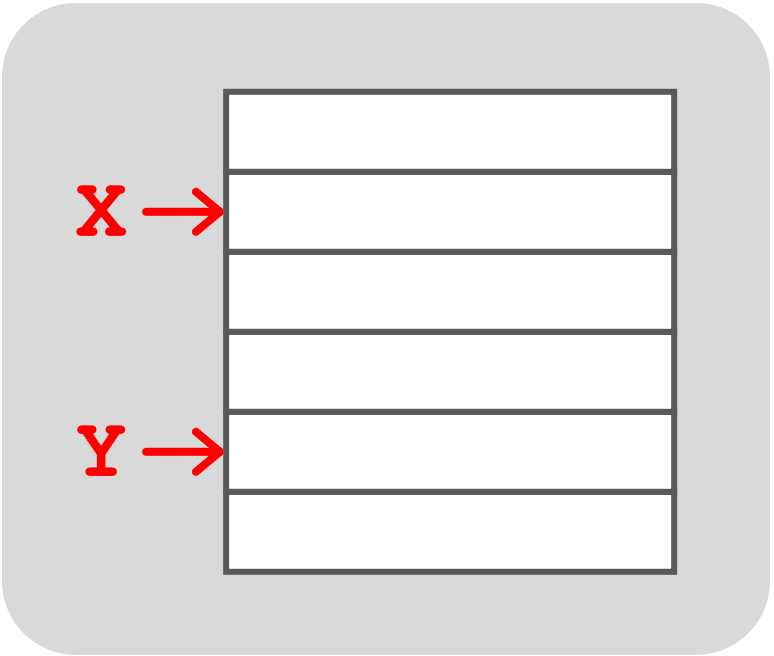
[Onur Mutlu]

# A Simple Program Can Induce Many Errors




1. Avoid ***cache hits***
   - Flush **X** from cache

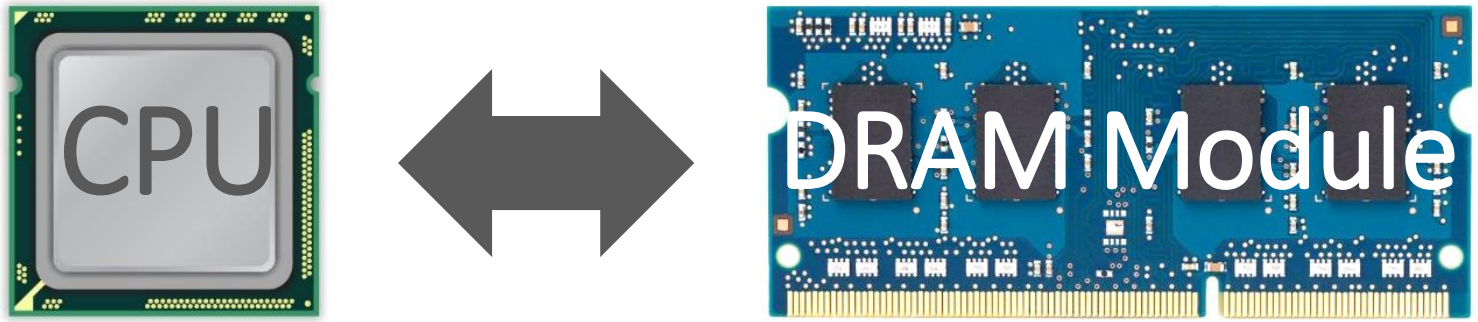
2. Avoid ***row hits*** to **X**
   - Read **Y** in another row



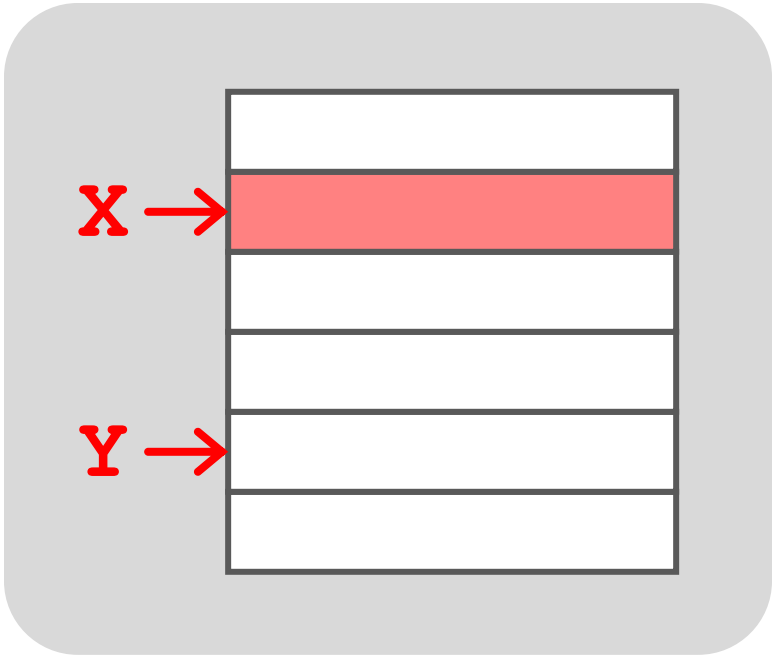

Download from: https://github.com/CMU-SAFARI/rowhammer

[Onur Mutlu]

# A Simple Program Can Induce Many Errors



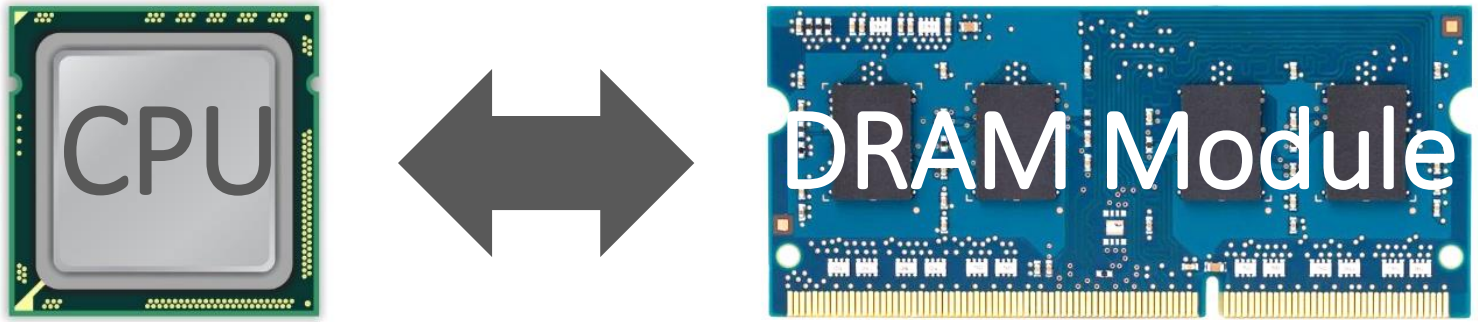

```
loop:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  mfence
  jmp loop
```



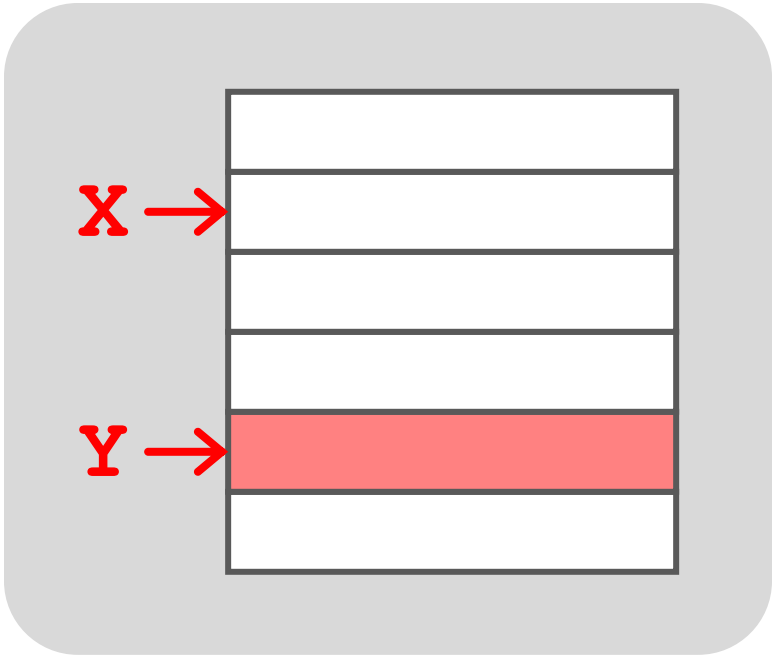

Download from: https://github.com/CMU-SAFARI/rowhammer

[Onur Mutlu]

# A Simple Program Can Induce Many Errors



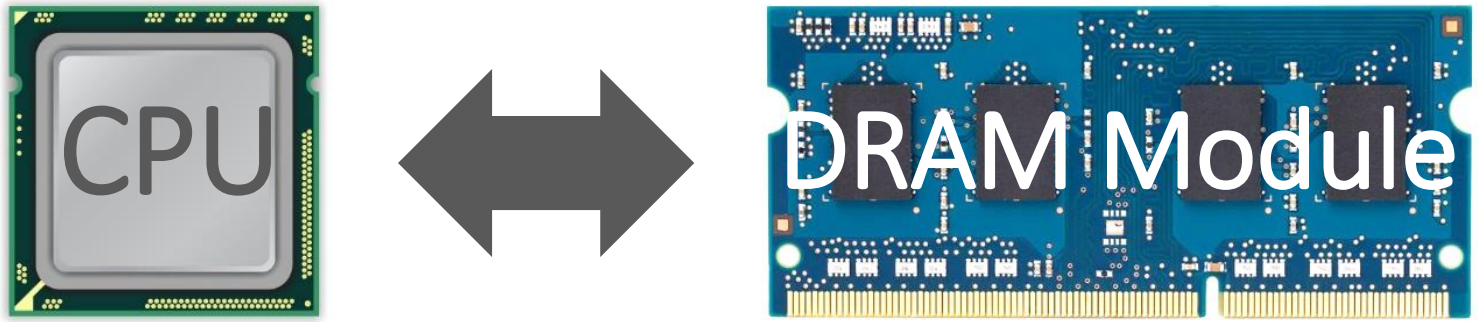

```
loop:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  mfence
  jmp loop
```



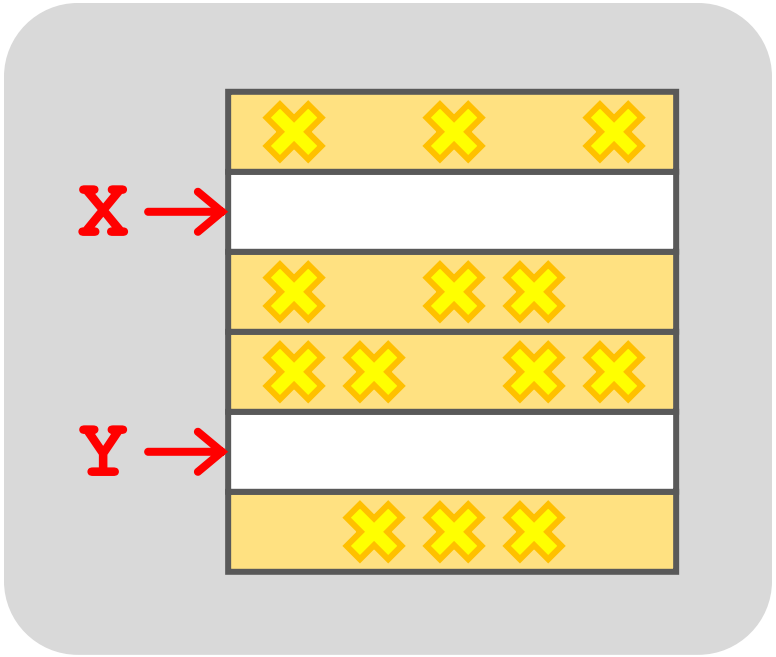

Download from: https://github.com/CMU-SAFARI/rowhammer

[Onur Mutlu]

# A Simple Program Can Induce Many Errors




```
loop:
   mov (X), %eax
   mov (Y), %ebx
   clflush (X)
   clflush (Y)
   mfence
   jmp loop
```




Download from: https://github.com/CMU-SAFARI/rowhammer

[Onur Mutlu]

# Observed Errors in Real Systems

| CPU Architecture | Errors | Access-Rate |
|---|---|---|
| Intel Haswell (2013) | 22.9K | 12.3M/sec |
| Intel Ivy Bridge (2012) | 20.7K | 11.7M/sec |
| Intel Sandy Bridge (2011) | 16.1K | 11.6M/sec |
| AMD Piledriver (2012) | 59 | 6.1M/sec |

- *A real reliability & security issue*
- *In a more controlled environment, we can induce as many as ten million disturbance errors*

Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

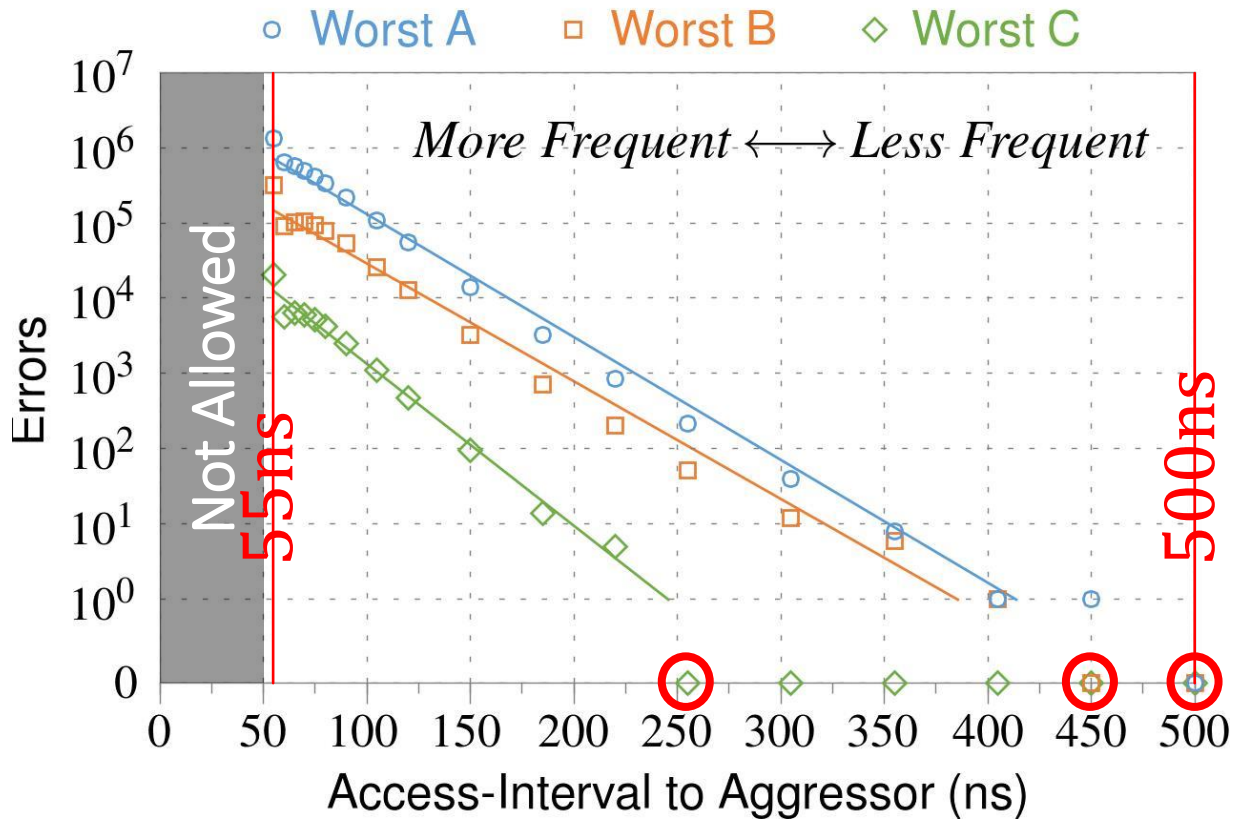
[Onur Mutlu]

# Root Causes of Disturbance Errors

- *Cause 1: Electromagnetic coupling*
  - Toggling the wordline voltage briefly increases the voltage of adjac ent wordlines
  - Slightly opens adjacent rows → Charge leakage
- *Cause 2: Conductive bridges*
- *Cause 3: Hot-carrier injection*

*Confirmed by at least one manufacturer*

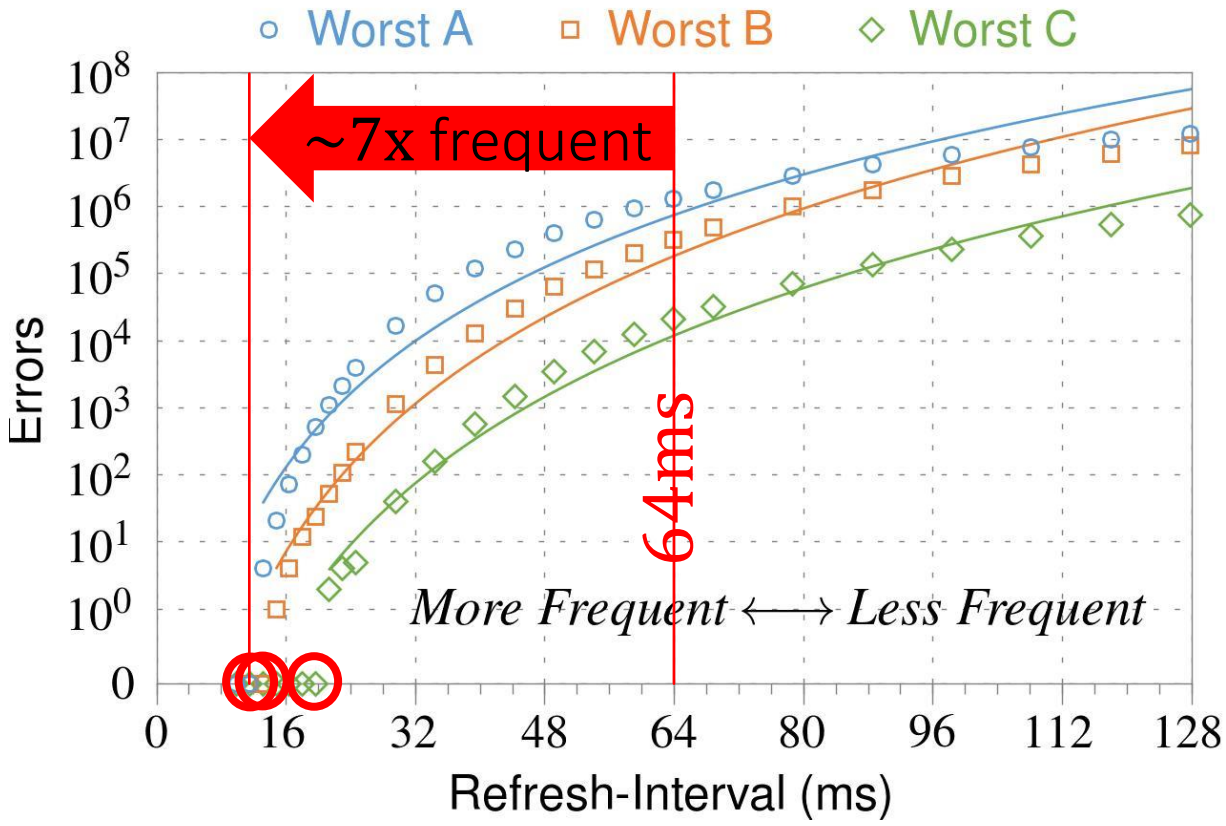
# ❶ Access Interval (Aggressor)




*Note: For three modules with the most errors (only first bank)*

*Less frequent accesses → Fewer errors*

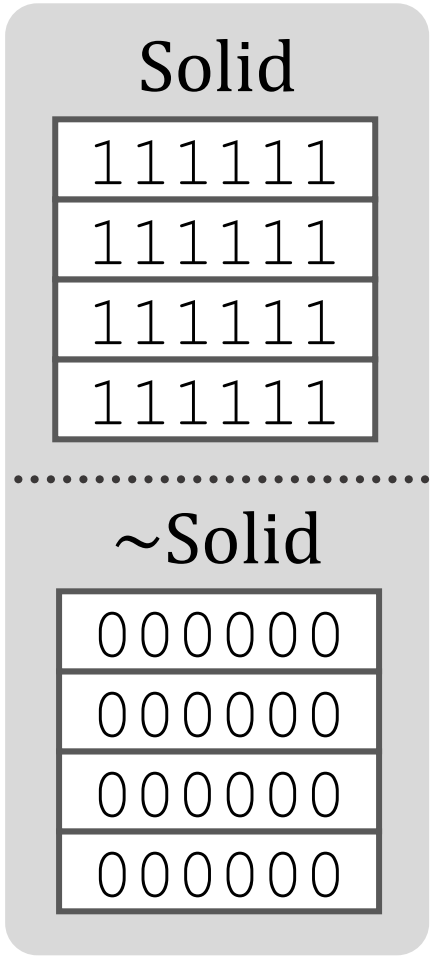
# ❷ Refresh Interval

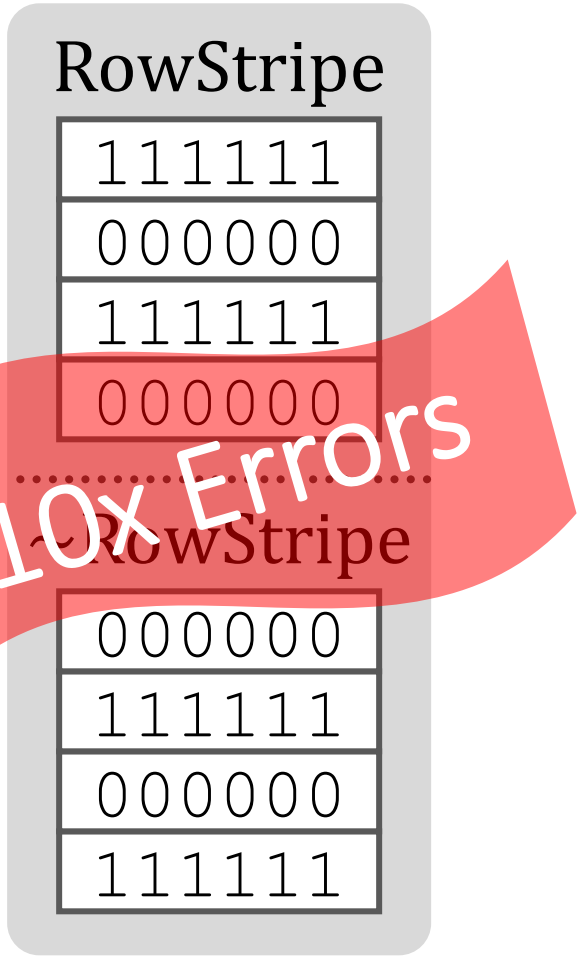



*Note: Using three modules with the most errors (only first bank)*

*More frequent refreshes → Fewer errors*

# ❸ Data Pattern




*Errors affected by data stored in other cells*

# Some Potential Solutions

- Make better DRAM chips — Cost
- Refresh frequently — Power, Performance
- Sophisticated ECC — Cost, Power
- Access counters — Cost, Power, Complexity

# Naive Solutions

1. *Throttle accesses to same row*
   - Limit access-interval: $\geq 500\text{ns}$
   - Limit number of accesses: $\leq 128\text{K}$ $(=64\text{ms}/500\text{ns})$

2. *Refresh more frequently*
   - Shorten refresh-interval by $\sim 7\text{x}$

*Both naive solutions introduce significant overhead in performance and power*

# Apple's Patch for RowHammer

- https://support.apple.com/en-gb/HT204934

Available for: OS X Mountain Lion v10.8.5, OS X Mavericks v10.9.5

Impact: A malicious application may induce memory corruption to escalate privileges

Description: A disturbance error, also known as Rowhammer, exists with some DDR3 RAM that could have led to memory corruption. This issue was mitigated by increasing memory refresh rates.

CVE-ID

CVE-2015-3693 : Mark Seaborn and Thomas Dullien of Google, working from original research by Yoongu Kim et al (2014)

HP and Lenovo released similar patches

[Onur Mutlu, https://arxiv.org/pdf/1703.00626.pdf]

# Proposed Solution

- **PARA:** *Probabilistic Adjacent Row Activation*
- **Key Idea**
  - After closing a row, we activate (i.e., refresh) one of its neighbors with a low probability: $p = 0.005$
- **Reliability Guarantee**
  - When $p=0.005$, errors in one year: $9.4\times10^{-14}$
  - By adjusting the value of $p$, we can provide an arbitrarily strong protection against errors

[Onur Mutlu, https://arxiv.org/pdf/1703.00626.pdf]

# Advantages of PARA

- *PARA refreshes rows infrequently*
  - Low power
  - Low performance-overhead
    - Average slowdown: $0.20\%$ (for $29$ benchmarks)
    - Maximum slowdown: $0.75\%$
- *PARA is stateless*
  - Low cost
  - Low complexity

- *PARA is an effective and low-overhead solution to prevent disturb ance errors*

[Onur Mutlu, https://arxiv.org/pdf/1703.00626.pdf]

# Requirements for PARA

- Better coordination between memory controller and DRAM
  - Memory controller should know which rows are physically adjacent
- In current DRAM chips, there will be some similar DRAM-only solutions which do not rely on memory controllers

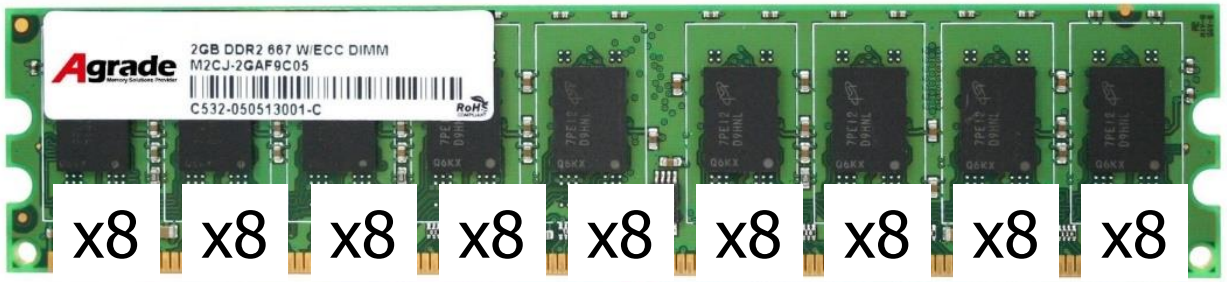
# Agenda

- DRAM architecture
- Memory access scheduling
- Refresh
- Row hammer
- Error correction code
- Summary

# Two DIMMs (Dual In-line Memory Module)


No error correction capability

64-bit data I/O

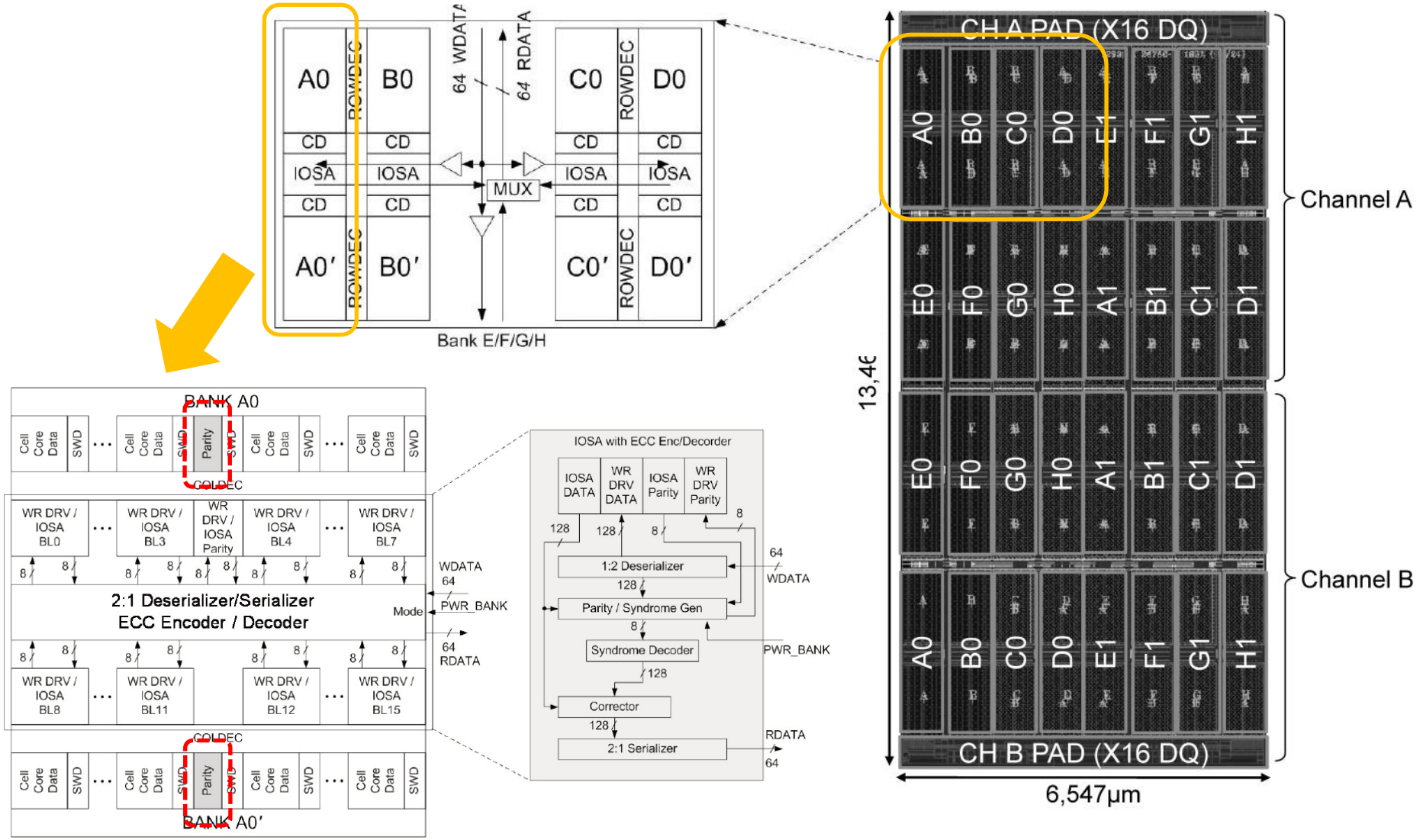



1-bit error correction per 72-bit (64+8)

72-bit data I/O (=64-bit data + 8-bit parity)

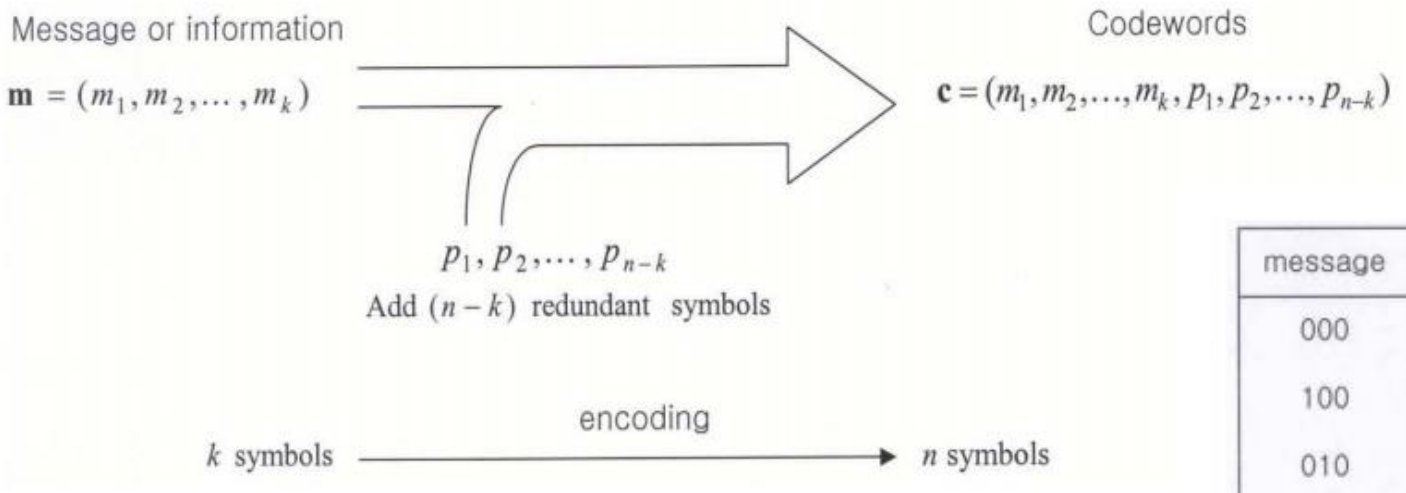
[ISSCC2014, Samsung]

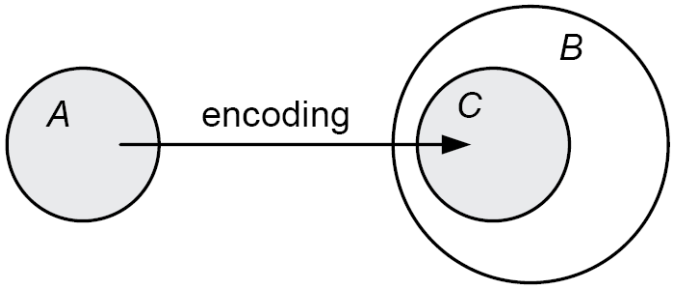
# LPDDR4 DRAM w/ in-DRAM ECC



# Error Control Coding

- Encoding and example




| message | codeword | distance |
|---|---|---|
| 000 | 000000 | 4 |
| 100 | 100101 | 1 |
| 010 | 010011 | 5 |
| 110 | 110110 | 4 |
| 001 | 001111 | 2 |
| 101 | 101010 | 3 |
| 011 | 011100 | 3 |
| 111 | 111001 | 2 |

transmit the information: 100

choose the codeword: 100101

101101 is received

the closest codeword: 100101

extract the information: 100101

comparing with 101101

# Coding Efficiency, k/n

- BCH case: # parity bits ~ $t*(\log_2 n+1)$
  - Single error correction (SEC) for 64-bit data requires 7 bits (11%)
  - Double error correction (DEC) for 64-bit data, 13 bits (20%)
  - LPDDR4 utilizes (136,8) which gives SEC (single error correction) for 128-bit data

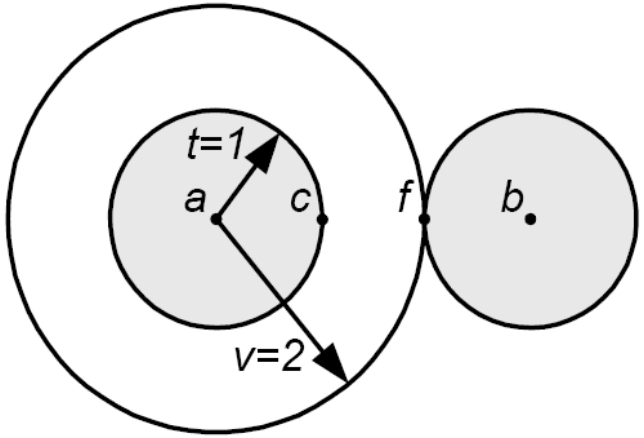
| *n* | *k* | *n–k* | *t* | *k/n* |
|---|---|---|---|---|
| 3 | 1 | 2 | 1 | 0.33 |
| 7 | 4 | 3 | 1 | 0.57 |
| 7 | 2 | 5 | 2 | 0.29 |
| 15 | 11 | 4 | 1 | 0.73 |
| 15 | 8 | 7 | 2 | 0.53 |
| 15 | 5 | 10 | 3 | 0.33 |
| 31 | 26 | 5 | 1 | 0.84 |
| 31 | 22 | 9 | 2 | 0.71 |
| 31 | 18 | 13 | 3 | 0.58 |
| 63 | 57 | 6 | 1 | 0.90 |
| 63 | 52 | 11 | 2 | 0.82 |
| 63 | 47 | 16 | 3 | 0.75 |
| 127 | 120 | 7 | 1 | 0.95 |
| 127 | 114 | 13 | 2 | 0.90 |
| 127 | 108 | 19 | 3 | 0.85 |
| 255 | 247 | 8 | 1 | 0.97 |
| 255 | 240 | 15 | 2 | 0.94 |
| 255 | 233 | 22 | 3 | 0.91 |
| 511 | 502 | 9 | 1 | 0.98 |
| 511 | 494 | 17 | 2 | 0.97 |
| 511 | 486 | 25 | 3 | 0.95 |


SEC for 64 bit data

SEC for 128 bit data

# Simple Example of ECC

- Error correction capability, t
- Table: t=1




| message | codeword | distance |
|---|---|---|
| 000 | 000000 | 4 |
| 100 | 100101 | 1 |
| 010 | 010011 | 5 |
| 110 | 110110 | 4 |
| 001 | 001111 | 2 |
| 101 | 101010 | 3 |
| 011 | 011100 | 3 |
| 111 | 111001 | 2 |

transmit the information: 100

choose the codeword: 100101

101101 is received

the closest codeword: 100101

extract the information: 100101

comparing with 101101

# Linear Block Codes

- Encoding of [7,4] Hamming code case: Given a data to write, obtain codeword computed with generator matrix and store it on the memory

$$\underbrace{\begin{aligned} c_1 &= m_1 \\ c_2 &= m_2 \\ c_3 &= m_3 \\ c_4 &= m_4 \end{aligned}}_{\text{information symbols}} \qquad \underbrace{\begin{aligned} c_5 &= m_2 + m_3 + m_4 \pmod 2 \\ c_6 &= m_1 + m_3 + m_4 \pmod 2 \\ c_7 &= m_1 + m_2 + m_4 \pmod 2 \end{aligned}}_{\text{redundant symbols ("parity-check" symbols)}}$$

Then

$$\mathbf{c} = (c_1, c_2, c_3, \ldots, c_7)$$

$$= \begin{bmatrix} m_1 & m_2 & m_3 & m_4 \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}}_{\mathbf{G}}$$

Note:
Multiplication = AND
Addition (mod 2) = XOR

i.e.,

$$\mathbf{c} = \mathbf{m}\,\mathbf{G}$$

# Linear Block Codes

- Reading codeword (data + parity) from the memory and parity check

**Example:** $[7, 4]$ Hamming code (continued)

$$\begin{aligned} c_i &= m_i\ , && i = 1,2,3,4 \\ c_5 &= m_2 + m_3 + m_4 && \pmod 2 \\ c_6 &= m_1 + m_3 + m_4 && \pmod 2 \\ c_7 &= m_1 + m_2 + m_4 && \pmod 2 \end{aligned} \quad \Rightarrow \quad \begin{aligned} c_2 + c_3 + c_4 + c_5 &= 0 \\ c_1 + c_3 + c_4 + c_6 &= 0 \\ c_1 + c_2 + c_4 + c_7 &= 0 \end{aligned}$$

$$\Rightarrow \underbrace{\left[\begin{array}{cccc:ccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right]}_{\mathbf{H}} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_7 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

That is,

$$\mathbf{H}\,\mathbf{c}^t = \mathbf{0}$$

syndrome

where $\mathbf{H}$: $(n-k) \times n$, $\mathbf{c}^t$: $n \times 1$.

$$\mathbf{G} = \left[\mathbf{I}_k \ \vdots \ \mathbf{P}\right] \longleftrightarrow \mathbf{H} = \left[-\mathbf{P}^t \ \vdots \ \mathbf{I}_{n-k}\right]$$

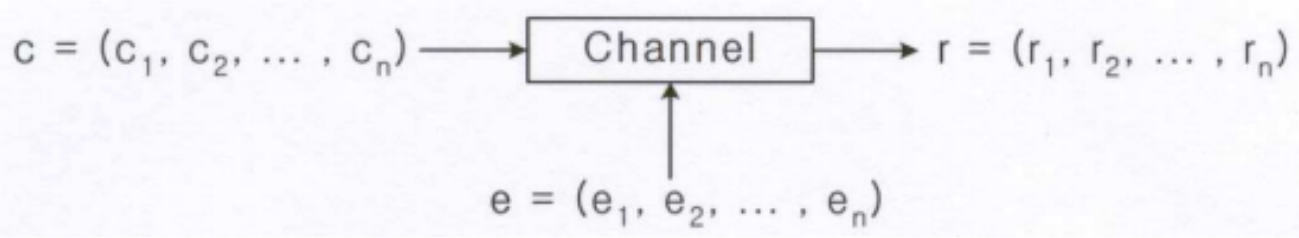
$\mathbf{GH}^t = \mathbf{0}$ for any linear code over $\mathbf{F}_q$.

# Linear Block Codes

- Syndrome (=parity check result) and error detection

Assume that a codeword $\mathbf{c}$ is transmitted and a error vector $\mathbf{e}$ is added to $\mathbf{c}$ (The channel is assumed to be *additive*). Then the received vector $\mathbf{r}$ is given by

$$\mathbf{r} = \mathbf{c} + \mathbf{e}.$$




The decoder gets the information on the unknown $\mathbf{e}$ from the observation $\mathbf{r}$: The decoder computes the so-called *syndrome*.

# Linear Block Codes

- Syndrome (=parity check result) and error detection

The syndrome $\mathbf{s} = (s_1, s_2, \cdots, s_{n-k})$ is defined by $\mathbf{s} \triangleq \mathbf{rH^t}$. Then

$$\mathbf{s} = \mathbf{rH^t} = (\mathbf{c} + \mathbf{e})\mathbf{H^t} = \underbrace{\mathbf{cH^t}}_{\mathbf{0}} + \mathbf{eH^t} = \mathbf{eH^t}.$$

Therefore,

$$\mathbf{s} = \mathbf{eH}^t.$$

# Linear Block Codes

- A syndrome (obtained by multiplying codeword and parity check matrix) can locate an error location: single error location <-> syndrome

**Example:** Syndrome computation in the $[7, 4]$ Hamming code

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{s} = \mathbf{e}\mathbf{H}^t.$$

**Received data r'=r+e, s = r'x $H^t$ = r x $H^t$ + e x $H^t$ = e x $H^t$**

| Syndrome | | Error vector (e) |
|---|---|---|
| [0 0 0] | → | [0 0 0 0 0 0 0] |
| [1 1 0] | → | [1 0 0 0 0 0 0] |
| [0 1 1] | → | [0 1 0 0 0 0 0] |
| [1 1 1] | → | [0 0 1 0 0 0 0] |
| [1 0 1] | → | [0 0 0 1 0 0 0] |
| [1 0 0] | → | [0 0 0 0 1 0 0] |
| [0 1 0] | → | [0 0 0 0 0 1 0] |
| [0 0 1] | → | [0 0 0 0 0 0 1] |

Syndrome gives error location info!

Error correction = Inverting the bit at error location

# Linear Block Codes

- A syndrome (obtained by multiplying codeword and parity check matrix) can locate an error location

**Example:** Syndrome computation in the $[7, 4]$ Hamming code

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{s} = \mathbf{e}\mathbf{H}^t.$$

r=0 1 0 0 1 0 1 → s=0 0 0

**Received data r'=r+e, $s = r'x\ H^t = r\ x\ H^t + e\ x\ H^t = e\ x\ H^t$**

| Syndrome | | Error vector (e) |
|---|---|---|
| [0 0 0] | → | [0 0 0 0 0 0 0] |
| [1 1 0] | → | [1 0 0 0 0 0 0] |
| [0 1 1] | → | [0 1 0 0 0 0 0] |
| [1 1 1] | → | [0 0 1 0 0 0 0] |
| [1 0 1] | → | [0 0 0 1 0 0 0] |
| [1 0 0] | → | [0 0 0 0 1 0 0] |
| [0 1 0] | → | [0 0 0 0 0 1 0] |
| [0 0 1] | → | [0 0 0 0 0 0 1] |

Syndrome gives error location info!

Error correction = Inverting the bit at error location

# Linear Block Codes

- A syndrome can locate an error location

**Example:** Syndrome computation in the $[7, 4]$ Hamming code

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{s} = \mathbf{e}\mathbf{H}^{\mathrm{t}}.$$

r=0 1 0 1 1 0 1 → s= 1 0 1

**Received data r'=r+e, $s = r'x\ H^t = r\ x\ H^t + e\ x\ H^t = e\ x\ H^t$**

| Syndrome | | Error vector (e) |
|---|---|---|
| [0 0 0] | → | [0 0 0 0 0 0 0] |
| [1 1 0] | → | [1 0 0 0 0 0 0] |
| [0 1 1] | → | [0 1 0 0 0 0 0] |
| [1 1 1] | → | [0 0 1 0 0 0 0] |
| [1 0 1] | → | [0 0 0 1 0 0 0] |
| [1 0 0] | → | [0 0 0 0 1 0 0] |
| [0 1 0] | → | [0 0 0 0 0 1 0] |
| [0 0 1] | → | [0 0 0 0 0 0 1] |

Syndrome gives error location info!

Error correction = Inverting the bit at error location

# What about Two Bit Errors?

- Double errors are not detected in its original form of Hamming code

**Example:** Syndrome computation in the $[7, 4]$ Hamming code

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

r=0 1 0 1 1 1 1 → s= 101+010=111

**Received data r'=r+e, s = r'x $H^t$ = r x $H^t$ + e x $H^t$ = e x $H^t$**

| Syndrome | | Error vector (e) | |
|---|---|---|---|
| [0 0 0] | → | [0 0 0 0 0 0 0] | |
| [1 1 0] | → | [1 0 0 0 0 0 0] | |
| [0 1 1] | → | [0 1 0 0 0 0 0] | |
| **[1 1 1]** | → | **[0 0 1 0 0 0 0]** | **Miss-correction!!!** |
| [1 0 1] | → | [0 0 0 1 0 0 0] | |
| [1 0 0] | → | [0 0 0 0 1 0 0] | |
| [0 1 0] | → | [0 0 0 0 0 1 0] | |
| [0 0 1] | → | [0 0 0 0 0 0 1] | |

# Erasure Method

- If there are two uncertain data bits (e.g., analog-to-digital converter can give uncertainty info), first mark them unknown

**Example:** Syndrome computation in the $[7, 4]$ Hamming code

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

r=0 1 0 1 1 1 1

r=0 1 0 X 1 X 1

4 cases!!!

r=0 1 0 0 1 0 1 → s= 000

r=0 1 0 0 1 1 1 → s= 010

r=0 1 0 1 1 0 1 → s= 101

r=0 1 0 1 1 1 1 → s= 111

This one gives no error!
Thus, the corrected one is 0100101

# Coding Efficiency, k/n

- BCH case: # parity bits ~ $t*(\log_2 n+1)$
  - Single error correction (SEC) for 64-bit data requires 7 bits (11%)
  - Double error correction (DEC) for 64-bit data, 13 bits (20%)
  - LPDDR4 utilizes 8-bit parity for SEC (single error correction) in 128-bit data

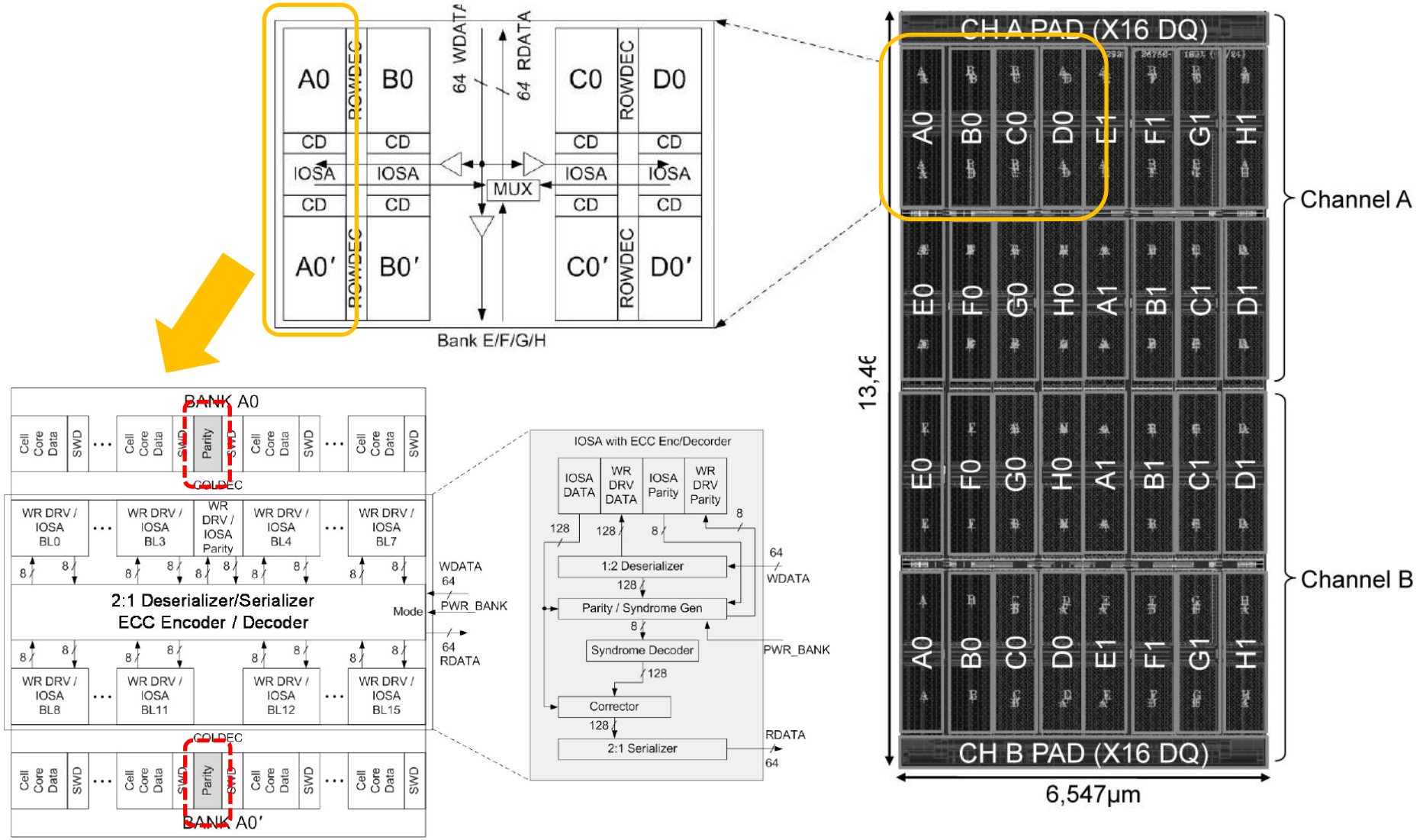
| *n* | *k* | *n–k* | *t* | *k/n* |
|---|---|---|---|---|
| 3 | 1 | 2 | 1 | 0.33 |
| 7 | 4 | 3 | 1 | 0.57 |
| 7 | 2 | 5 | 2 | 0.29 |
| 15 | 11 | 4 | 1 | 0.73 |
| 15 | 8 | 7 | 2 | 0.53 |
| 15 | 5 | 10 | 3 | 0.33 |
| 31 | 26 | 5 | 1 | 0.84 |
| 31 | 22 | 9 | 2 | 0.71 |
| 31 | 18 | 13 | 3 | 0.58 |
| 63 | 57 | 6 | 1 | 0.90 |
| 63 | 52 | 11 | 2 | 0.82 |
| 63 | 47 | 16 | 3 | 0.75 |
| 127 | 120 | 7 | 1 | 0.95 |
| 127 | 114 | 13 | 2 | 0.90 |
| 127 | 108 | 19 | 3 | 0.85 |
| 255 | 247 | 8 | 1 | 0.97 |
| 255 | 240 | 15 | 2 | 0.94 |
| 255 | 233 | 22 | 3 | 0.91 |
| 511 | 502 | 9 | 1 | 0.98 |
| 511 | 494 | 17 | 2 | 0.97 |
| 511 | 486 | 25 | 3 | 0.95 |


SEC for 64 bit data

SEC for 128 bit data

[ISSCC2014, Samsung]

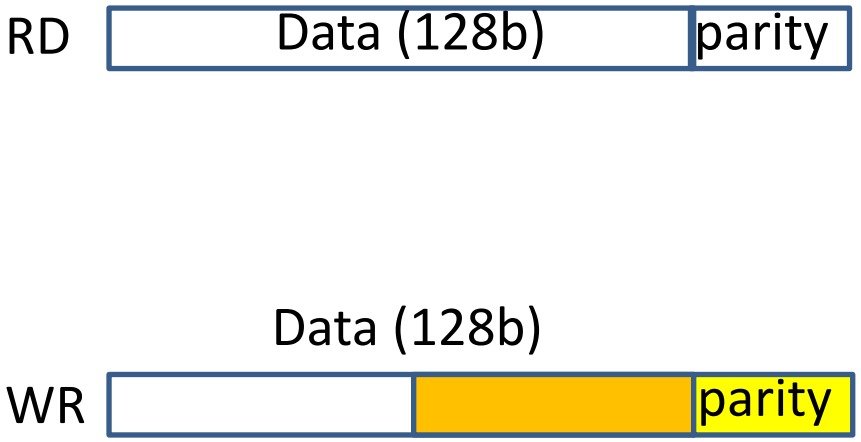
# LPDDR4 DRAM w/ in-DRAM ECC



[JEDEC 2014]

# Partial (i.e., Masked) Write Requires Read and Write

- Read the existing data
- Write new data and compute a new parity
- Thus, it requires longer latency, i.e., tCCDMW than a normal write




| Next CMD / Current CMD | Active | Read (BL=16 or 32) | Write (BL=16 or 32) | Masked Write | Precharge |
|---|---|---|---|---|---|
| Active | illegal | RU(tRCD/tCK) | RU(tRCD/tCK) | RU(tRCD/tCK) | RU(tRAS/tCK) |
| Read with BL = 16 | illegal | 8[1)] | RL+RU(tDQSCK(max) /tCK) +BL/2- WL+tWPRE+tRPST | RL+RU(tDQSCK(max) /tCK) +BL/2- WL+tWPRE+tRPST | BL/2+max{(8,RU(tRTP /tCK)}-8 |
| Read with BL = 32 | illegal | 16[2)] | RL+RU(tDQSCK(max) /tCK) +BL/2- WL+tWPRE+tRPST | RL+RU(tDQSCK(max) /tCK) +BL/2- WL+tWPRE+tRPST | BL/2+max{(8,RU(tRTP /tCK)}-8 |
| Write with BL = 16 | illegal | WL+1+BL/2 +RU(tWTR/tCK) | 8[1)] | tCCDMW[3)] | WL+ 1 + BL/2+RU(tWR/tCK) |
| Write with BL = 32 | illegal | WL+1+BL/2 +RU(tWTR/tCK) | 16[2)] | tCCDMW +8[4)] | WL+ 1 + BL/2+RU(tWR/tCK) |
| Masked Write | illegal | WL+1+BL/2 +RU(tWTR/tCK) | tCCD | tCCDMW[3)] | WL+ 1 + BL/2 +RU(tWR/tCK) |
| Precharge | RU(tRP/tCK), RU(tRPab/tCK) | illegal | illegal | illegal | 4 |

Notes:

1. In the case of BL = 16, tCCD is 8*tCK.
2. In the case of BL = 32, tCCD is 16*tCK.
3. tCCDMW = 32*tCK (4*tCCD at BL=16)
4. Write with BL=32 operation has 8*tCK longer than BL =16.

# Agenda

- DRAM architecture
- Memory access scheduling
- Refresh
- Row hammer
- Error correction code
- Summary

# Summary (from the Viewpoint of Application or Software Designer)

- DRAM architecture and memory access scheduling
  - DRAM-aware data layout to exploit bank parallelism and row buffer hits
- Refresh
  - OS can give info to DRAM to reduce self refresh power
  - Fine-grained auto-refresh option to reduce worst-case memory latency
- Row hammer
  - Most of system errors are from main memory, possibly, row hammer
- Error correction code
  - Partial writes can incur more latency due to additional read required to calculate parity

# Weekly Lecture / Lab Schedule

- W1 (March 6) Class introduction / (March 8) Orientation & team formation
- W2 13 Verilog 1 Combinational circuits / 15 Verilog 1 (tool installation, adder & multiplier combinational logic)
- W3 20 Verilog 2 Sequential circuits / 22 Verilog 2 (memory i/o, FSM sequential logic)
- W4 27 AI application introduction 1, Amaranth introduction 1 / 29 Amaranth (tool installation, MAC, adder tree)
- W5 4/3 AI application introduction 2, Amaranth introduction 2 (memory i/o, FSM sequential logic) / 5 Amaranth (PE)
- W6 10 AI application introduction 3, Neural network accelerator 1 / 12 Amaranth (stacked PEs)
- W7 17 Neural network accelerator 2 / 19 Amaranth (stacked PEs)
- W8 24 **Mid-term exam** / 26 Amaranth (stacked PEs)
- W9 5/1 Reading data from memory 1 (VA2PA, interconnect) / 3 Convolution lowering
- W10 8 Reading data from memory 2 (DRAM main memory), Compressing networks 1 (pruning) / 10 Tiling
- W11 15 Compressing networks 2 (pruning, low precision) / 17 PyTorch model – Amaranth simulator communication
- W12 22 Zero-skipping & low-precision hardware accelerator / 24 Quantization, project introduction
- W13 29 Invited talks (commercial solutions: Rebellions, Furiosa AI) / 31 Homework Q&A
- W14 6/5 **Final exam** / 7 Project Q&A
- W15 12 Claim & Project Q&A / 14 Project submission