08: AXI4 Lite

Test

→ Jupiter Notebook
↑
→ P3 issues

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University

Announcements

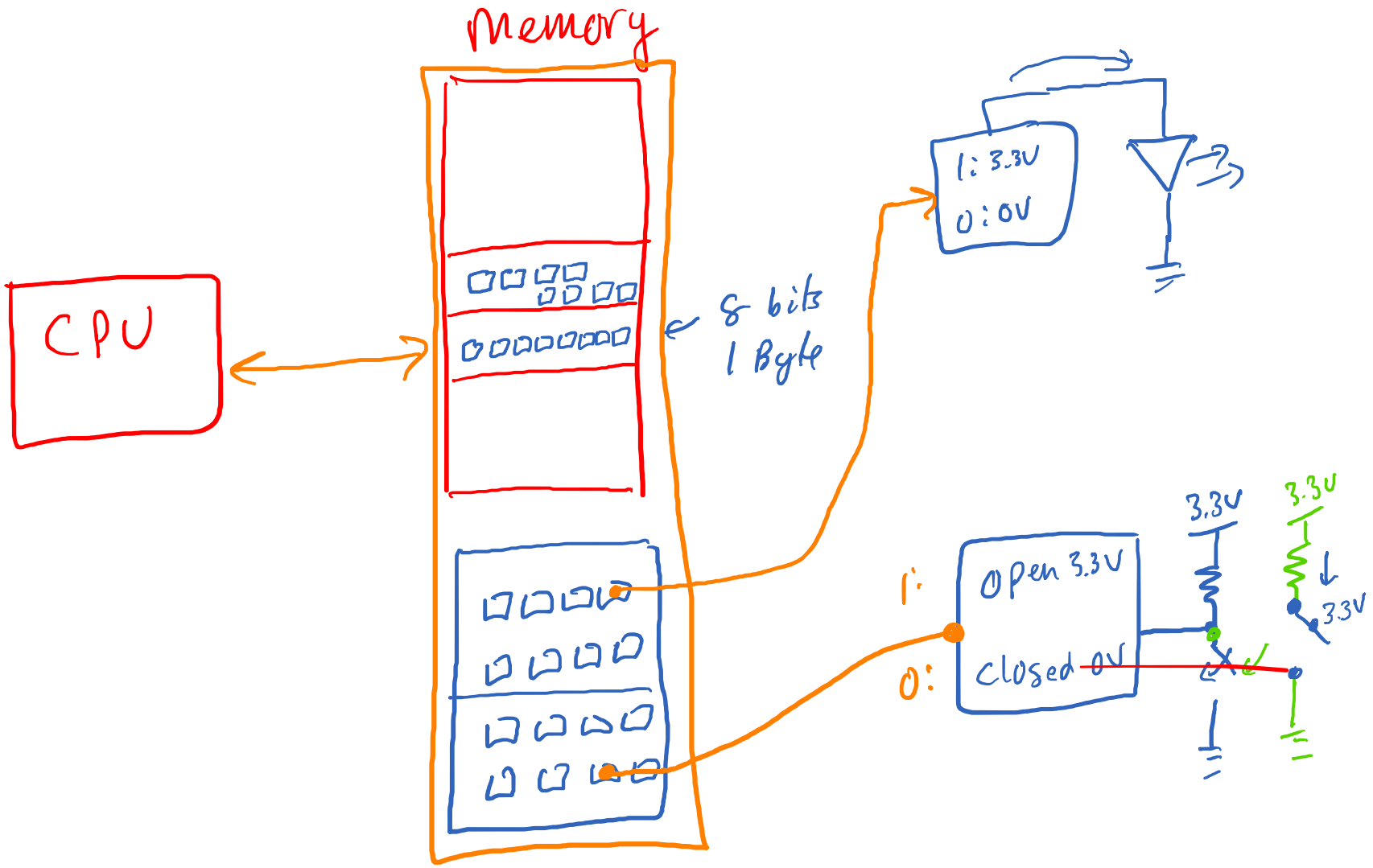
- Office Hours – Times Unchanged
- P3: Due Friday
- P4: Out now. AG hopefully today.

Project 3 Testbench

Optimizations thus far

- Algorithmic complexity
- Removing redundant computation
- ~~Multithreading~~
- ~~Multiprocessing*~~
- Python/C/Asm Interfacing
- Map to Hardware

Review: Memory-Mapped I/O

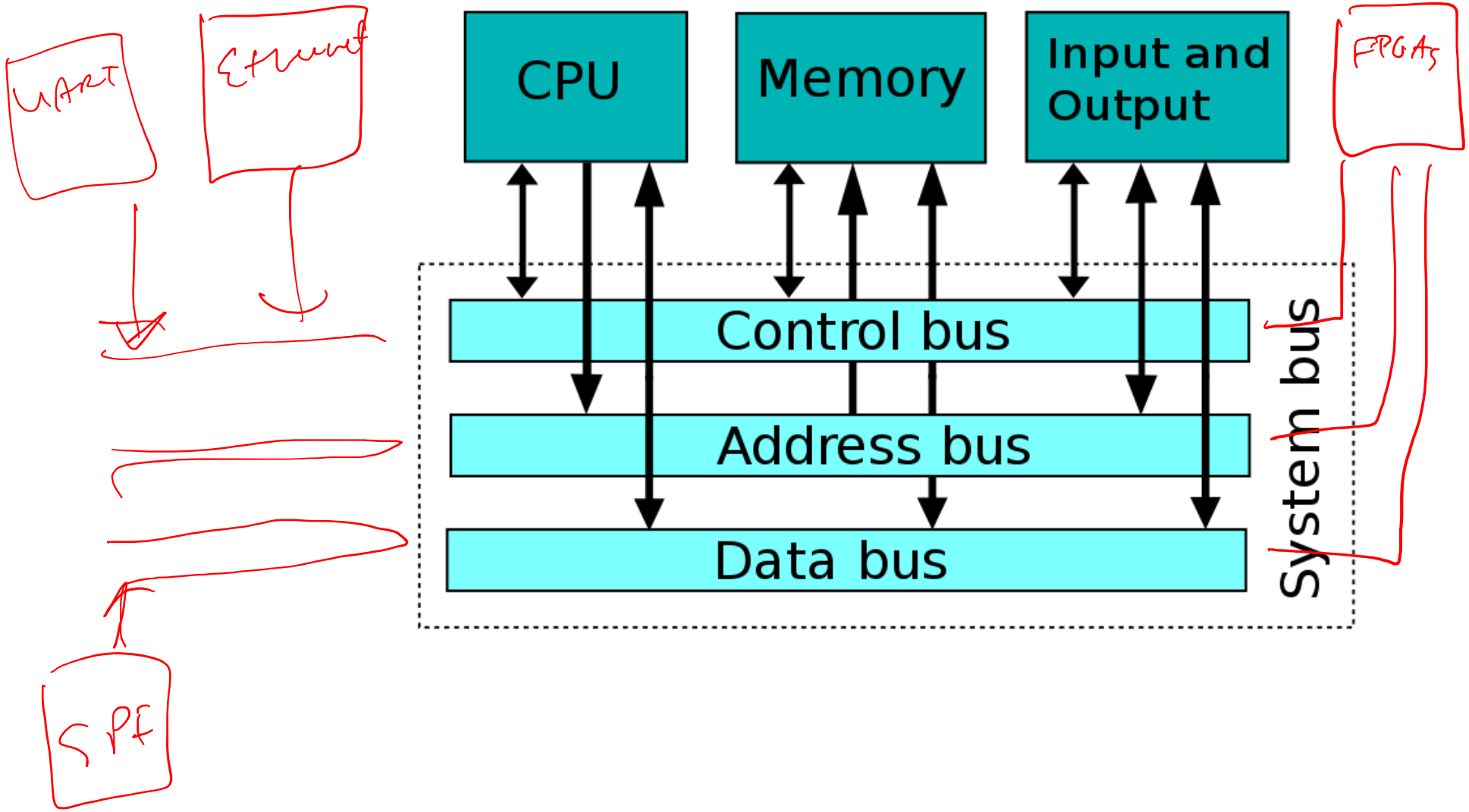


Use `volatile` for MMIO addresses!

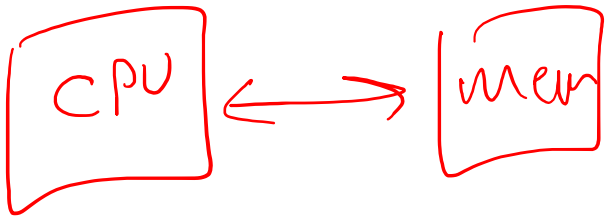
```
#define SW_ADDR 0xfffe
volatile uint32_t * SW_REG = (uint32_t * SW_ADDR);

int quit = (*SW_REG);
while(!quit)
{
     //more code
     quit = (*SW_REG);
}
```

The System Bus



Hypothetical Bus Example



- Characteristics
  - Asynchronous (no clock) – hay, why no?
  - One Initiator and One Target

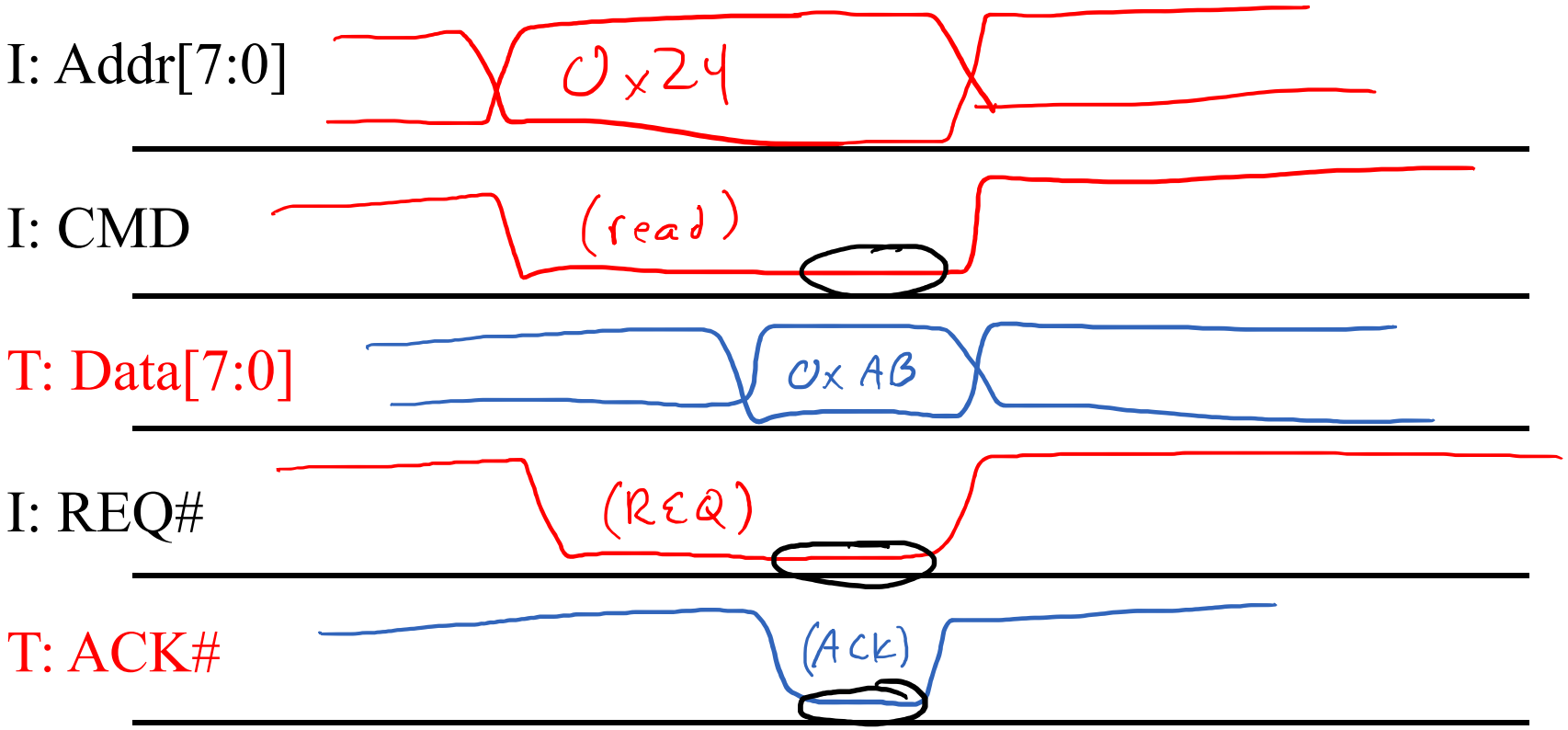
- Signals
  - Addr[7:0], Data[7:0], CMD, REQ#, ACK#
    - CMD=0 is read, CMD=1 is write.
    - REQ# low means initiator is requesting something.
    - ACK# low means target is acknowledging the job is done.

CPU Load = read
Store = write

Read transaction

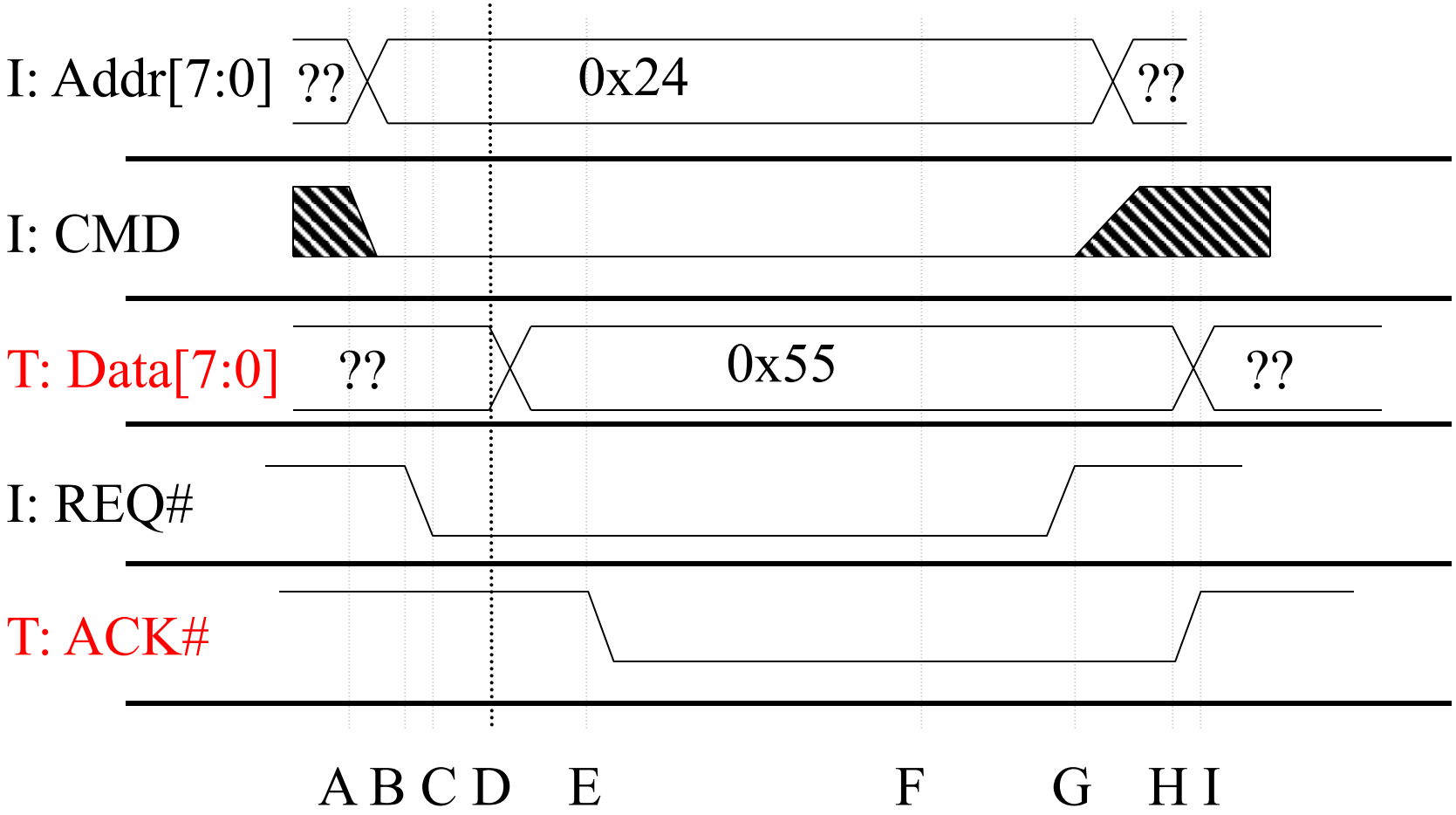
Initiator wants to read location 0x24

CMD=0 is read, CMD=1 is write.
REQ# low means initiator is requesting.
ACK# low means target is acknowledging.



Read transaction

Initiator wants to read location 0x24



A read transaction

Say initiator wants to read location 0x24

A. Initiator sets Addr=0x24, CMD=0
B. Initiator *then* sets REQ# to low
C. Target sees read request
D. Target drives data onto data bus
E. Target *then* sets ACK# to low
F. Initiator grabs the data from the data bus
G. Initiator sets REQ# to high, stops driving Addr and CMD
H. Target stops driving data, sets ACK# to high terminating the transaction
I. Bus is seen to be idle

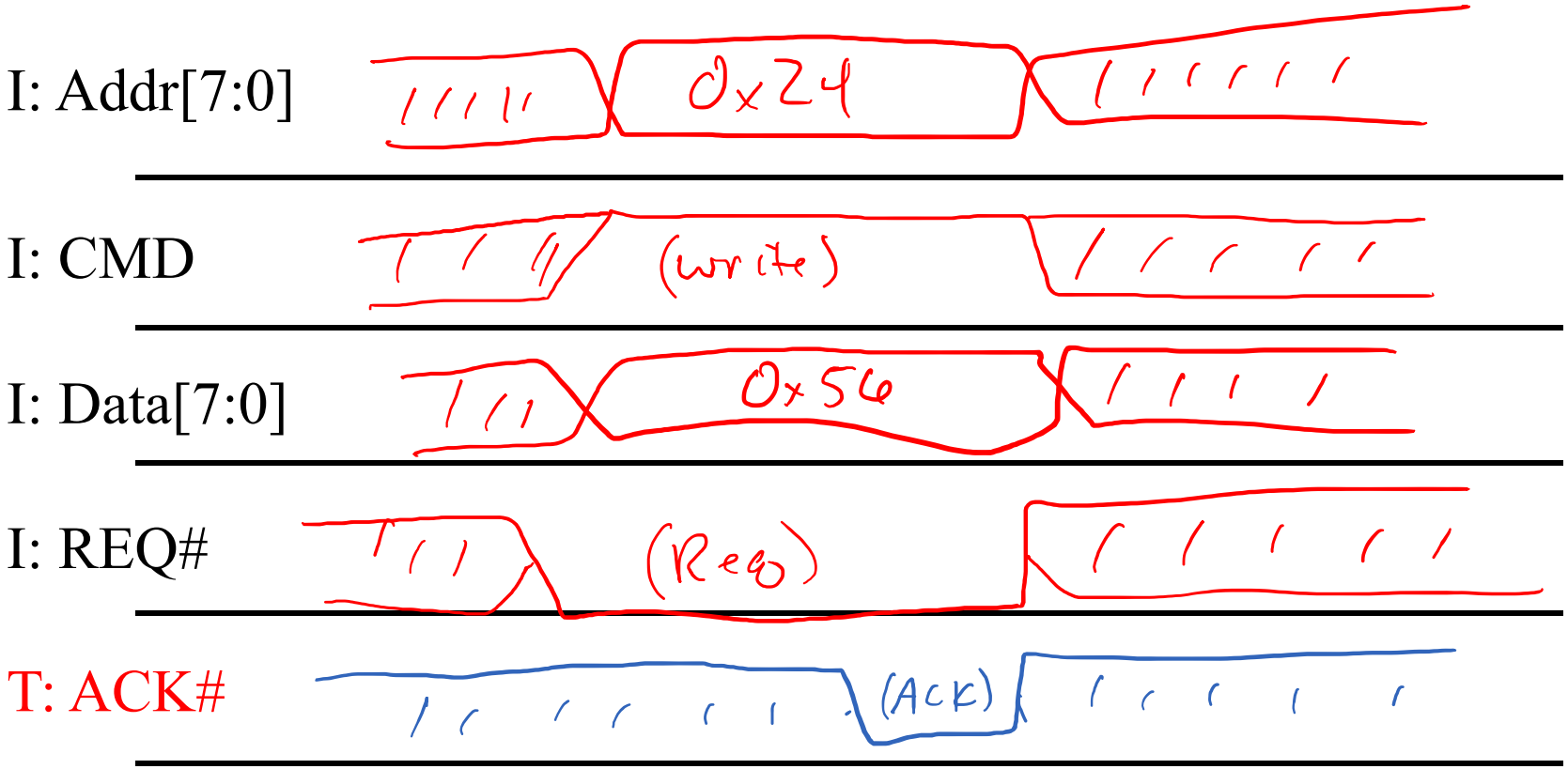
CPU store

(Mem)

Write transaction

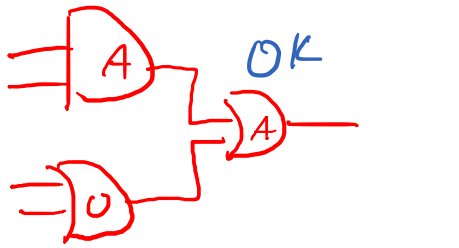
Initiator wants to write 0x56 to location 0x24

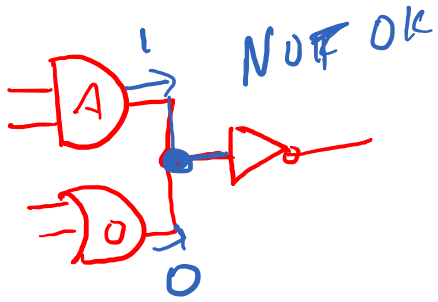
CMD=0 is read, CMD=1 is write.
REQ# low means initiator is requesting.
ACK# low means target is acknowledging.

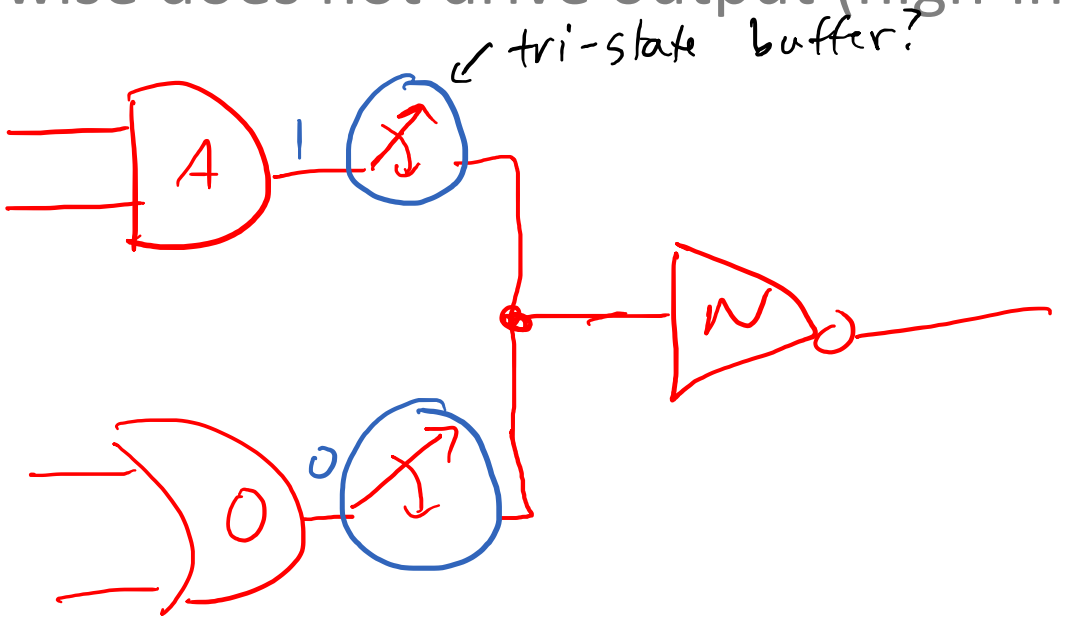


Tri-State Buffer

- Drives output when enabled
- Otherwise does not drive output (high-impedance)



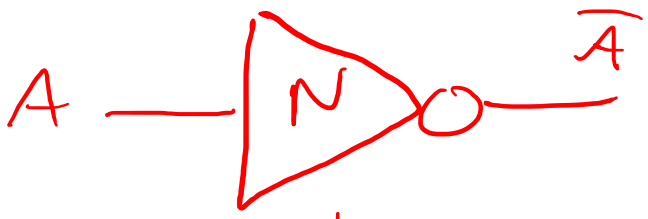




close both = bad!

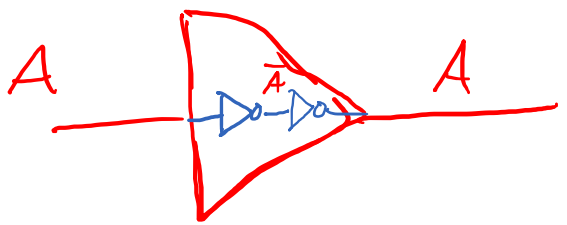
only close 1, ok?

NOT

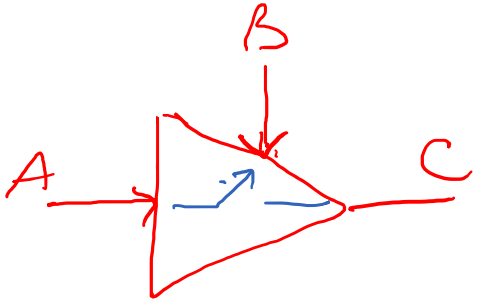


A	$\bar{A}$
0	1
1	0

Buffer

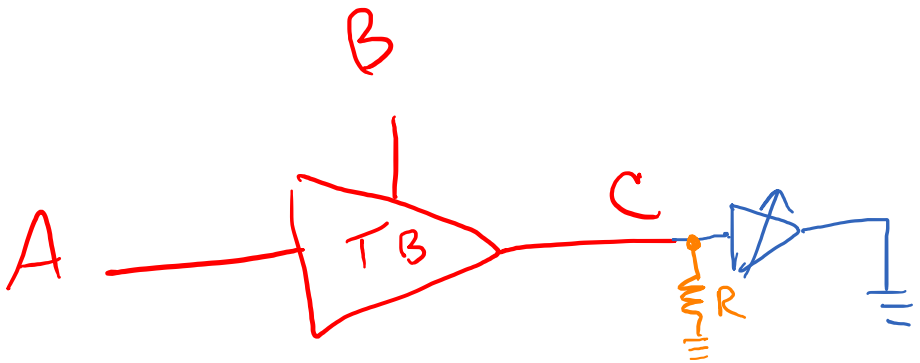


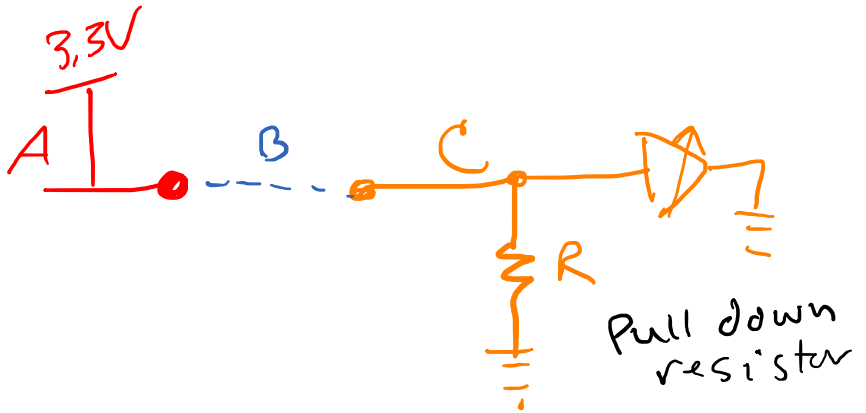
A	A
1	1
0	0



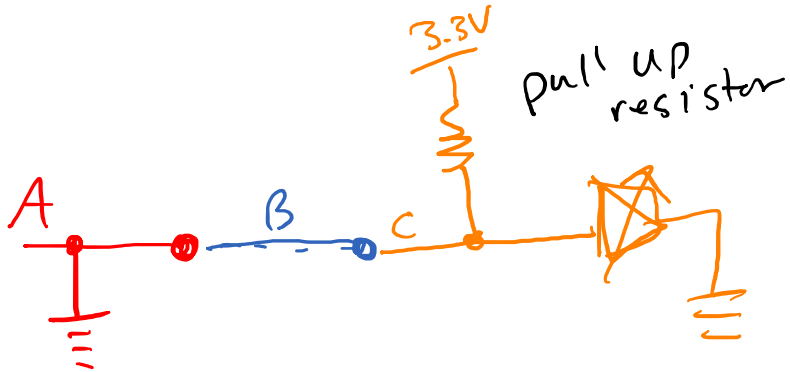
A	B	C
0	1	0
1	1	1
0	0	z
1	0	z

B is 1 ⇒ switch closed





A	B	C	LED
1	1	1 (3.3V)	Glow
0	1	0 (0V)	OFF
X	0	Z (??V)	



Can MMIO behave as memory?

Example peripherals

0x04: Push Button - Read-Only

Pushed -> 1

Not Pushed -> 0

0x05: LED Driver - Write-Only

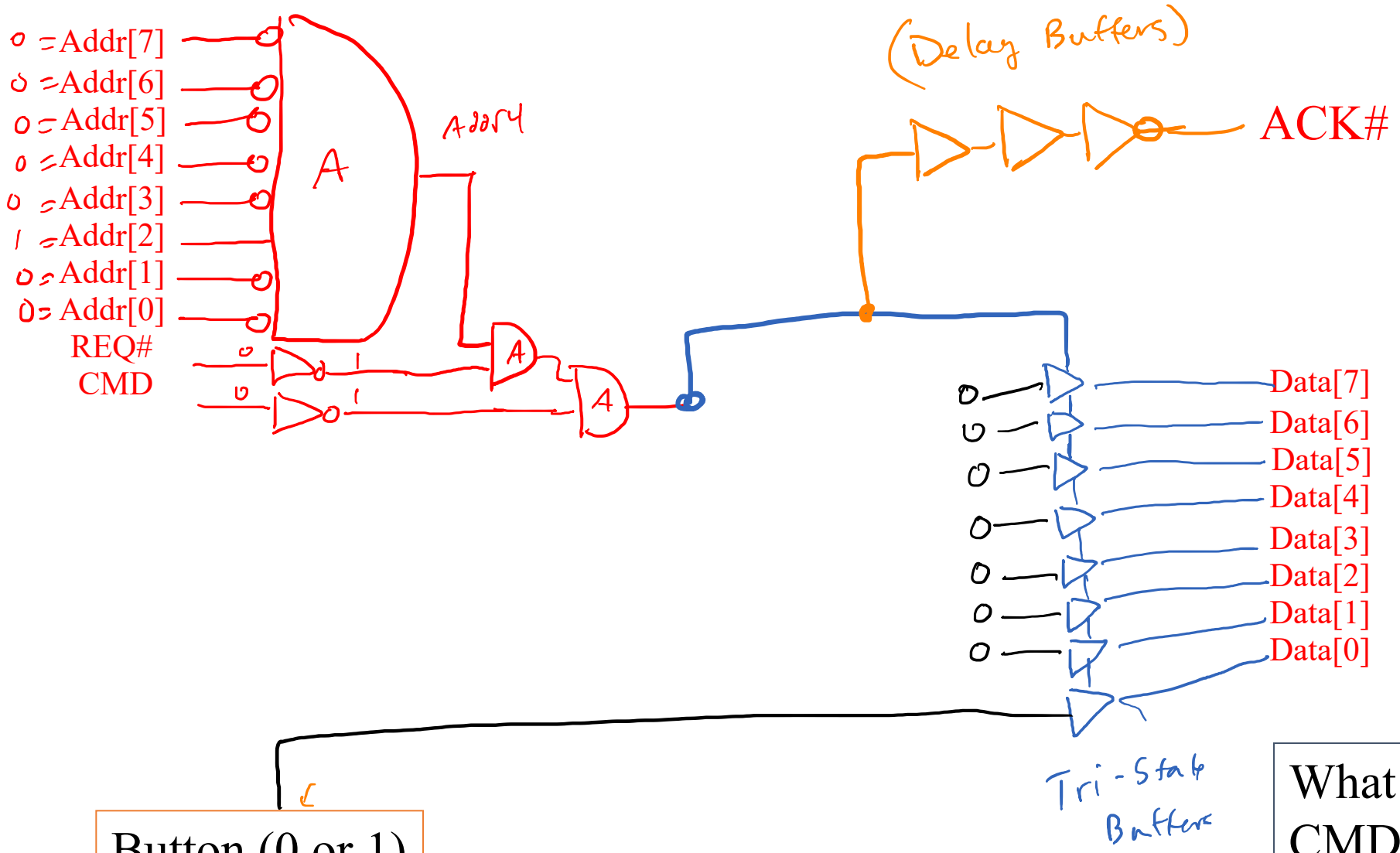
On -> 1

Off -> 0

The push-button

(if Addr=0x04 read 0 or 1 depending on button)

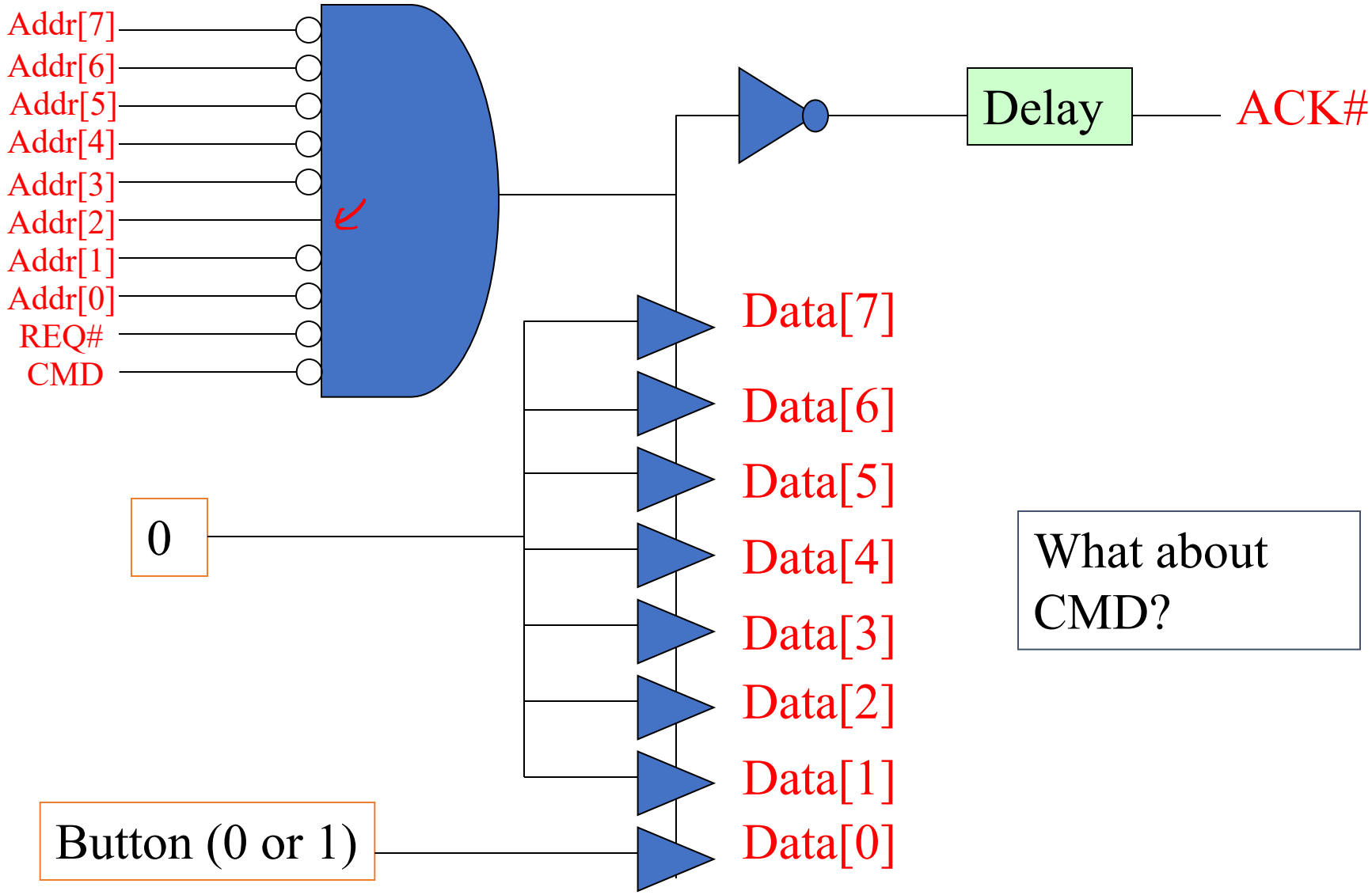
CMD=0 is read, CMD=1 is write.
REQ# low means initiator is requesting.
ACK# low means target is acknowledging.



What about CMD?

The push-button

(if Addr=0x04 ~~write~~ read 0 or 1 depending on button)



The LED

(1 bit reg written by LSB of address 0x05)

CMD=0 is read, CMD=1 is write.
REQ# low means initiator is requesting.
ACK# low means target is acknowledging.

Addr[7]
Addr[6]
Addr[5]
Addr[4]
Addr[3]
Addr[2]
Addr[1]
Addr[0]
REQ#
CMD

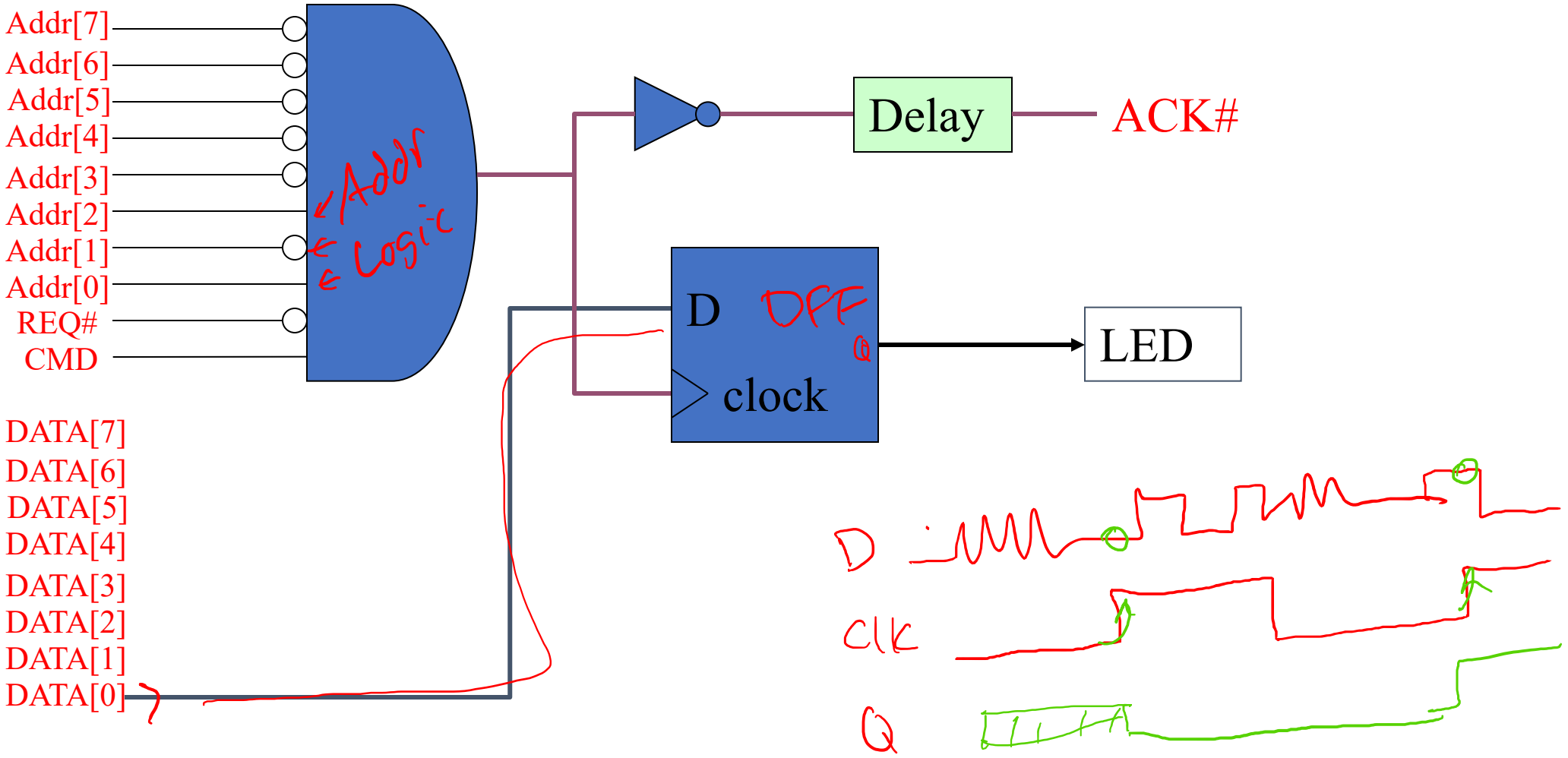
DATA[7]
DATA[6]
DATA[5]
DATA[4]
DATA[3]
DATA[2]
DATA[1]
DATA[0]

ACK#

LED

The LED

(1 bit reg written by LSB of address 0x05)



Let's write a simple C program to turn the LED on if button is pressed.

Peripheral Details

0x04: Push Button - Read-Only

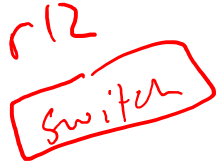
Pushed -> 1

Not Pushed -> 0

0x05: LED Driver - Write-Only

On -> 1

Off -> 0




```
loop:   mov   r8, 0x4
        mov   r0, 0x5
        ldr   r12, [r8]
        str   r12, [r0]
        b     loop
```

In ASM:

```
        mov r0, #0x4   % PB
        mov r1, #0x5   % LED
loop:   ldr r2, [r0, #0]
        str r2 [r1, #0]
        b loop
```

Let's write a simple C program to turn the LED on if button is pressed.

Peripheral Details

0x04: Push Button - Read-Only
- Pushed -> 1
- Not Pushed -> 0

0x05: LED Driver - Write-Only
- On -> 1
- Off -> 0

```
#define PB  0x4
#define LED 0x5

int main(){
    int value; (r12)
    for(;;){
        value = *((volatile uint32_t *)(PB));
        *(uint32_t *)(LED) = value;
    }
}
```

volatile

ARM AXI Bus

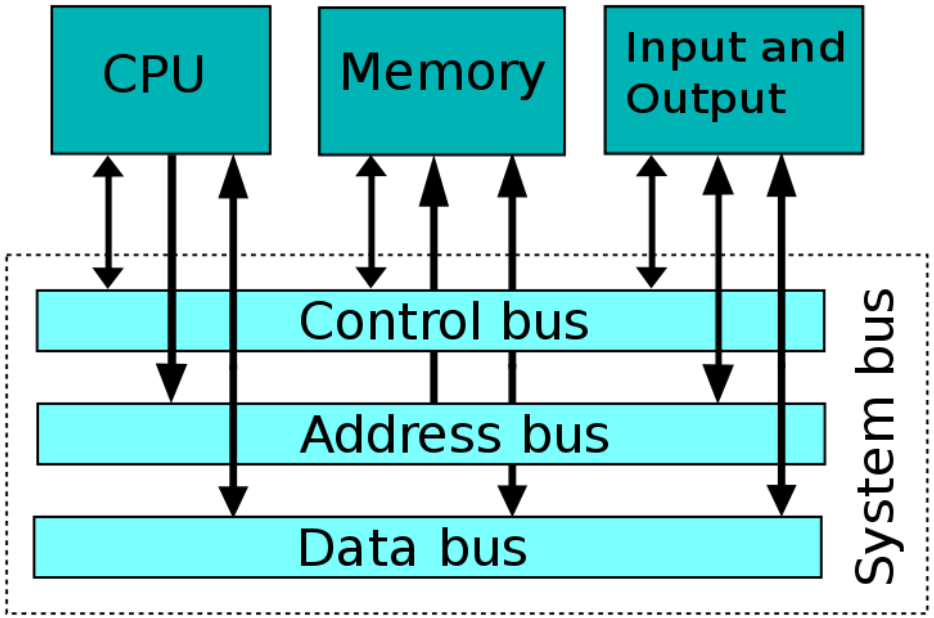
Stop here!

- "Advanced eXtensible Interface" Bus Version 4, "AXI4"

ARM AXI Bus

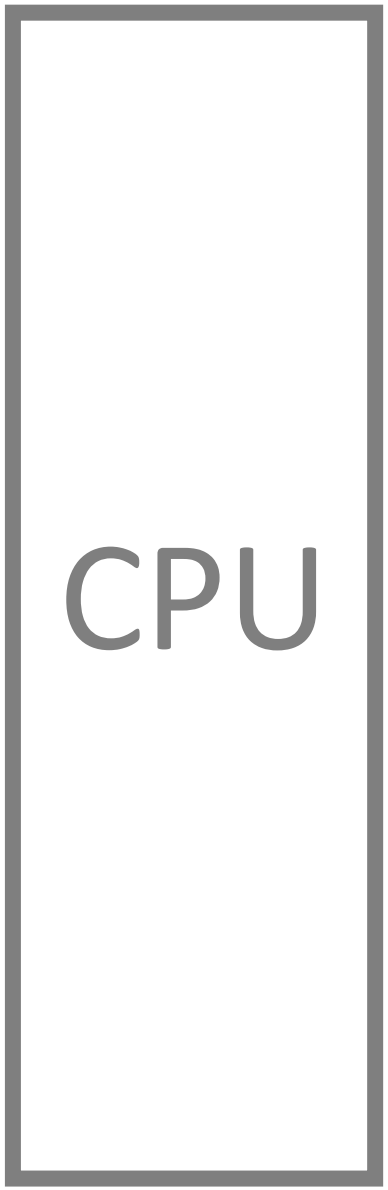
- "Advanced eXtensible Interface" Bus Version 4, "AXI4"
- Three Variants
  - AXI4: Fast but complicated; Memory-mapped
  - AXI4 Lite: Slow but simple; Memory-mapped
  - AXI4 Stream: Fast and simple; Not memory-mapped
    - PS uses this

Why AXI4 Lite?

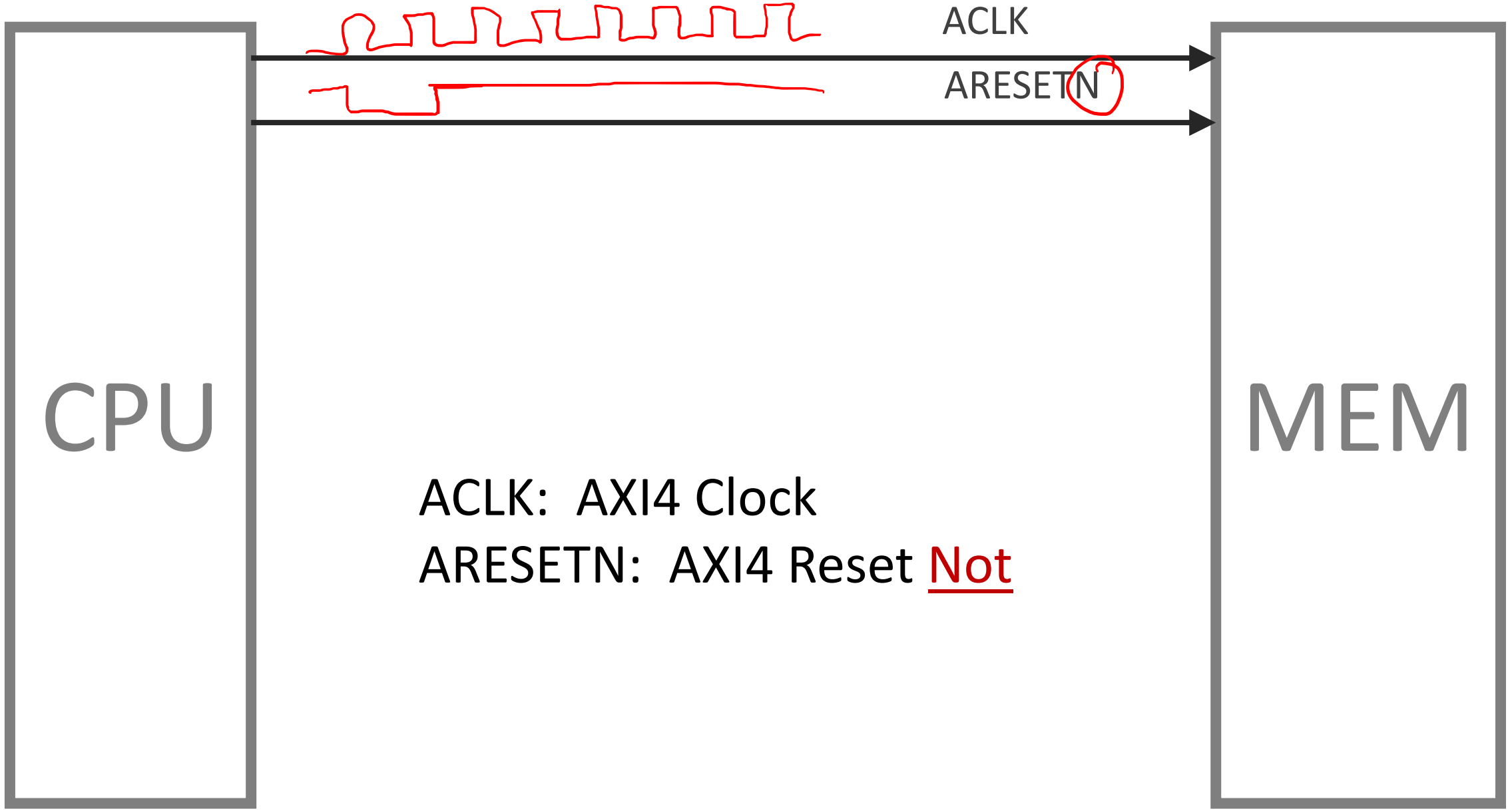


Xilinx AXI Reference Guide:

"AXI4-Lite is a light-weight, single transaction memory mapped interface. It has a small logic footprint and is a simple interface to work with both in design and usage. "

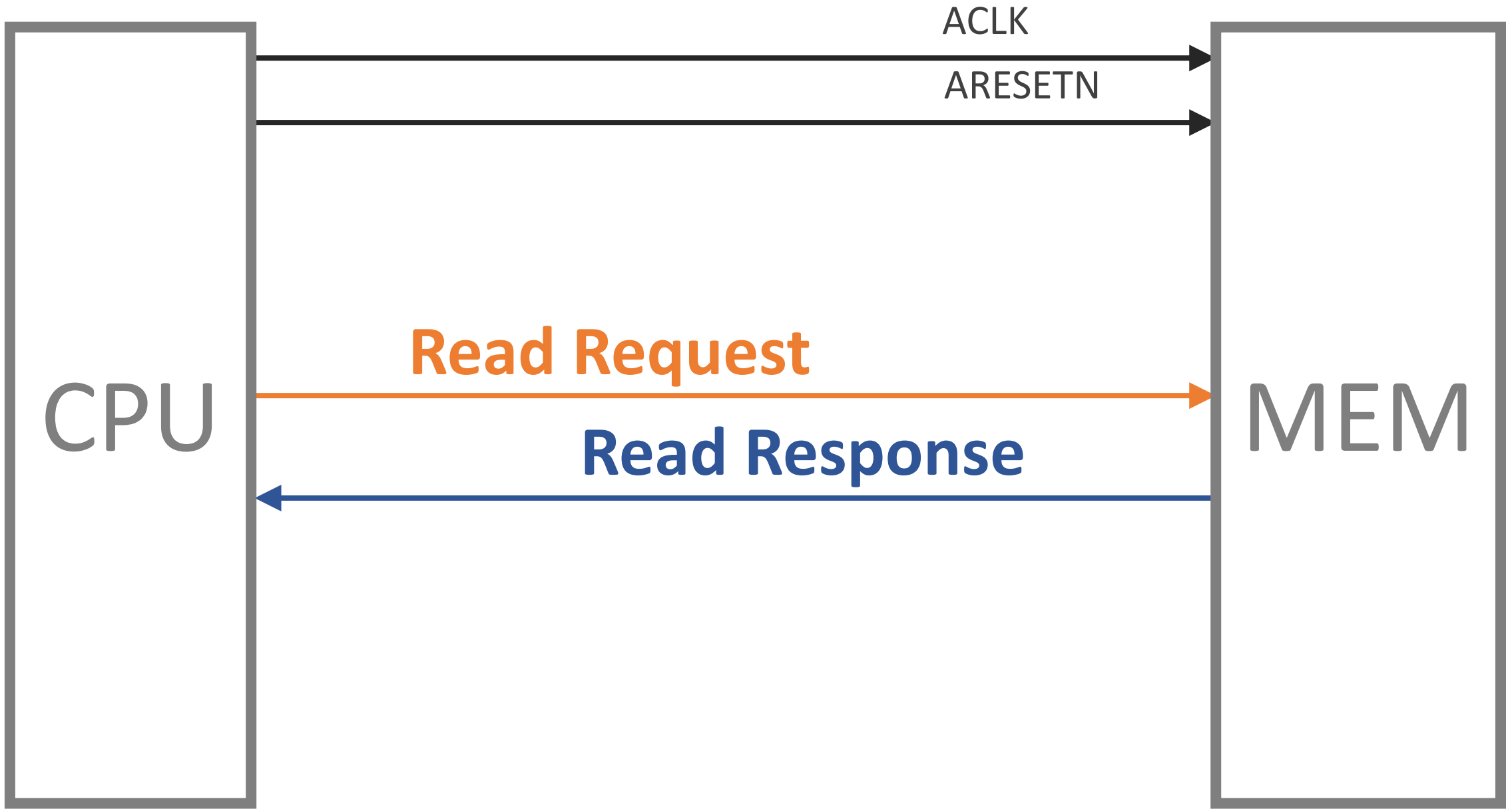

CPU

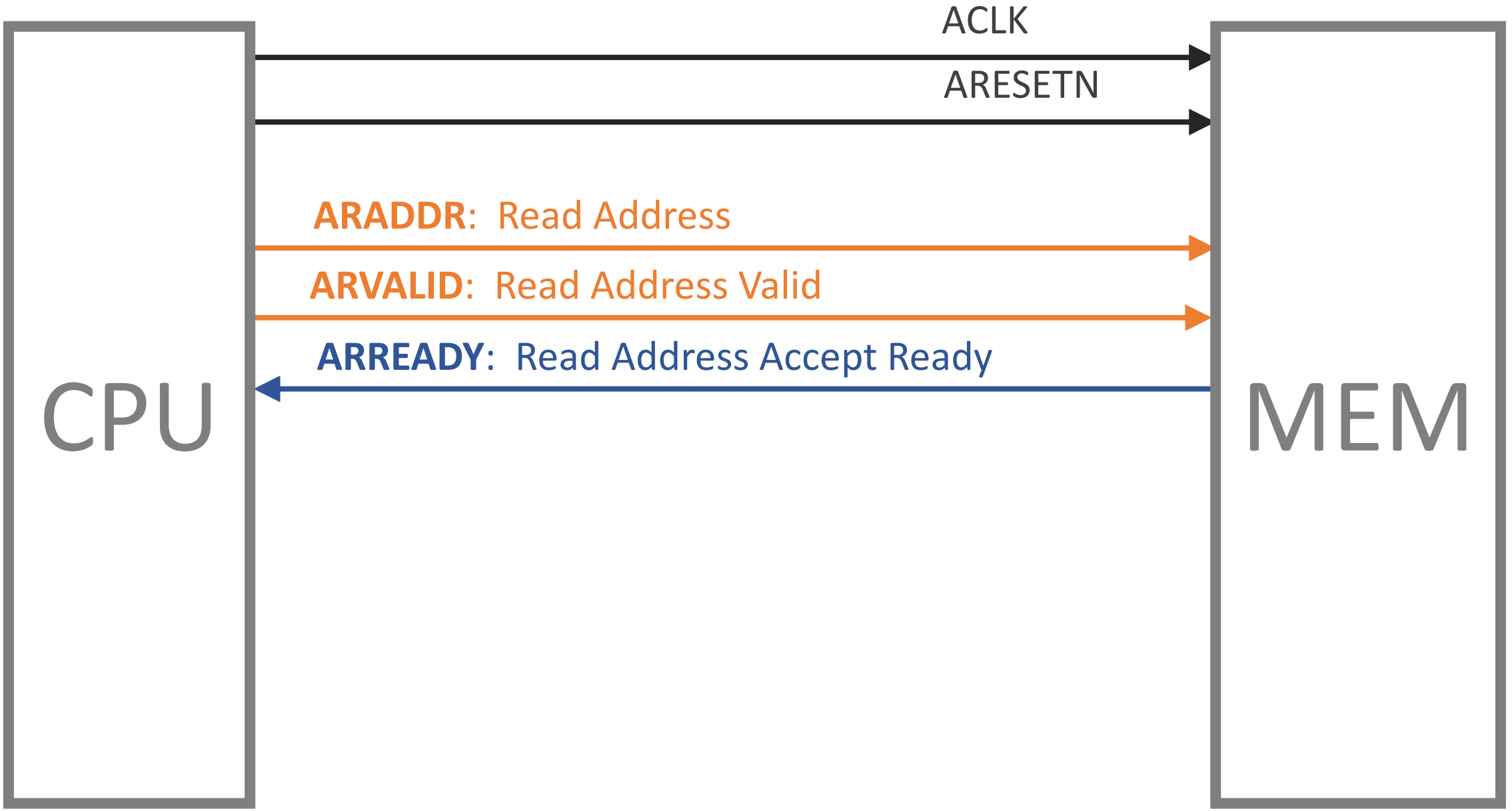

MEM



ACLK: AXI4 Clock

ARESETN: AXI4 Reset Not


ACLK
ARESETN
CPU
Read Request
Read Response
MEM


ACLK
ARESETN
ARADDR: Read Address
ARVALID: Read Address Valid
ARREADY: Read Address Accept Ready
CPU
MEM

AXI4 Handshaking

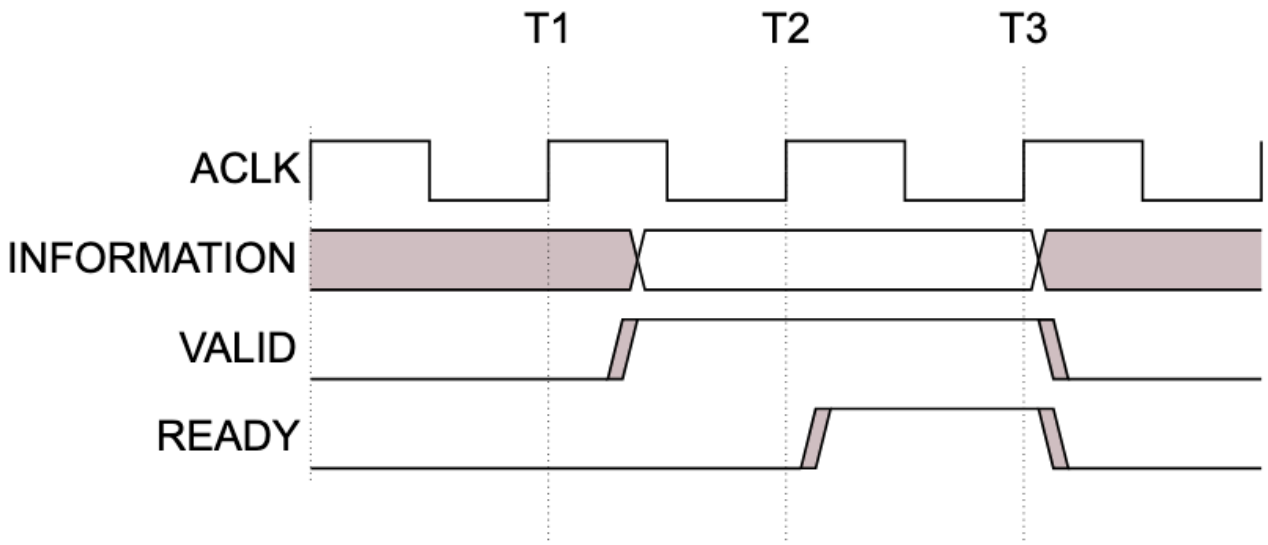
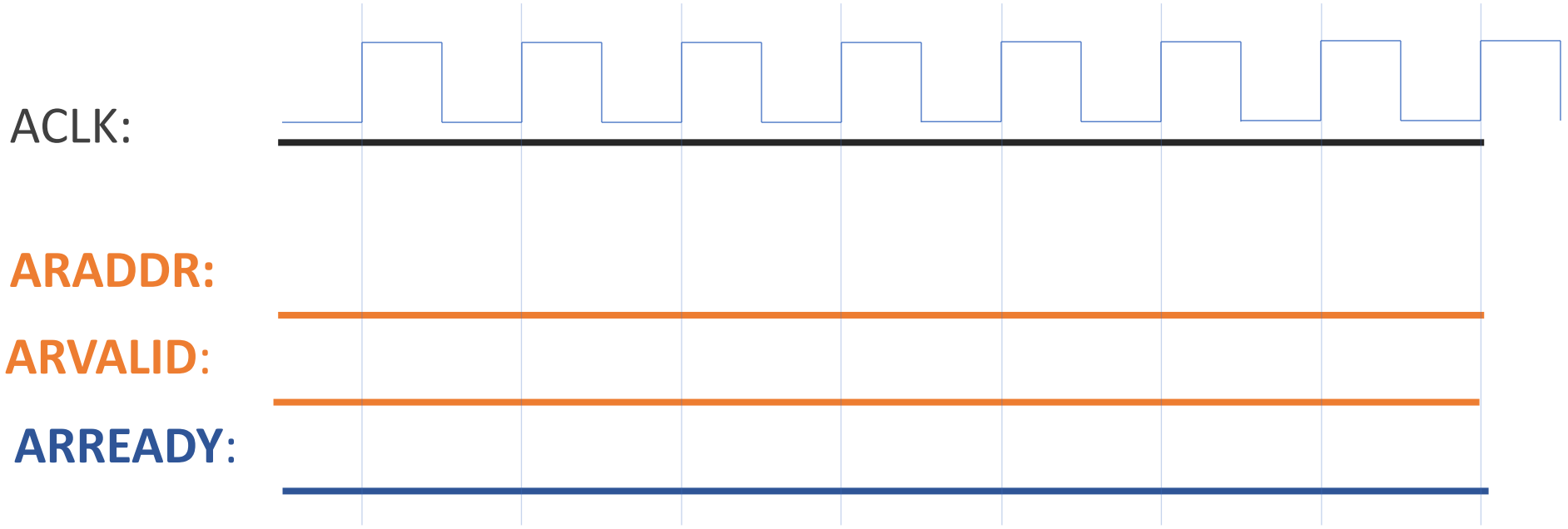
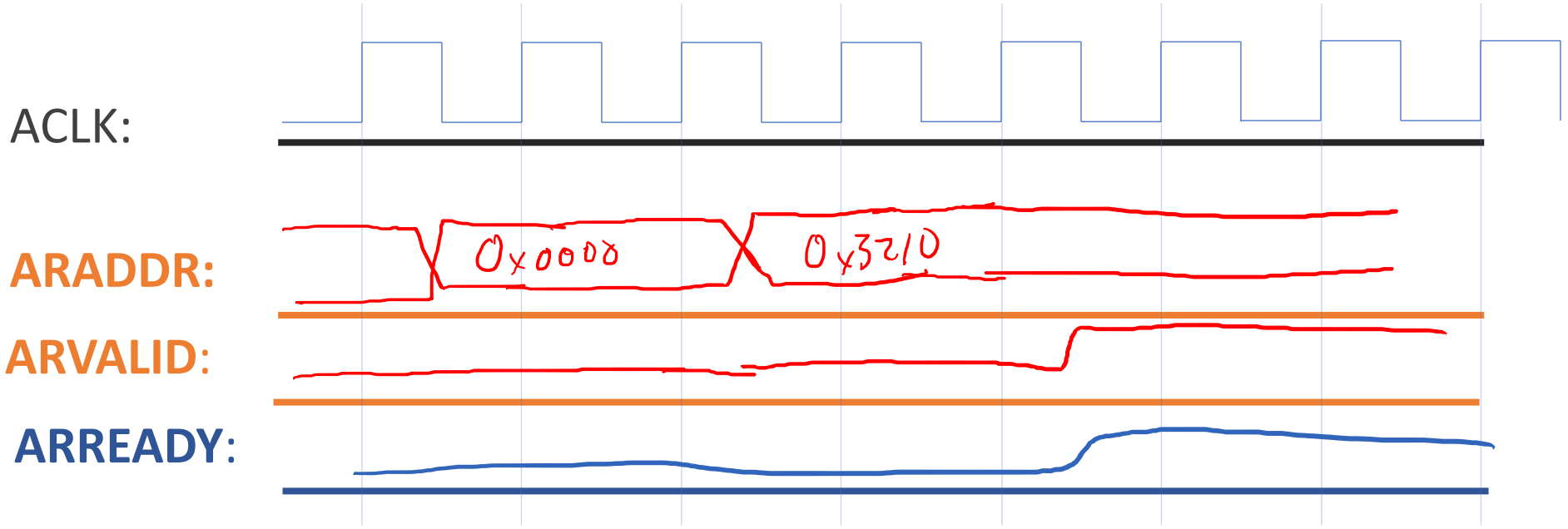


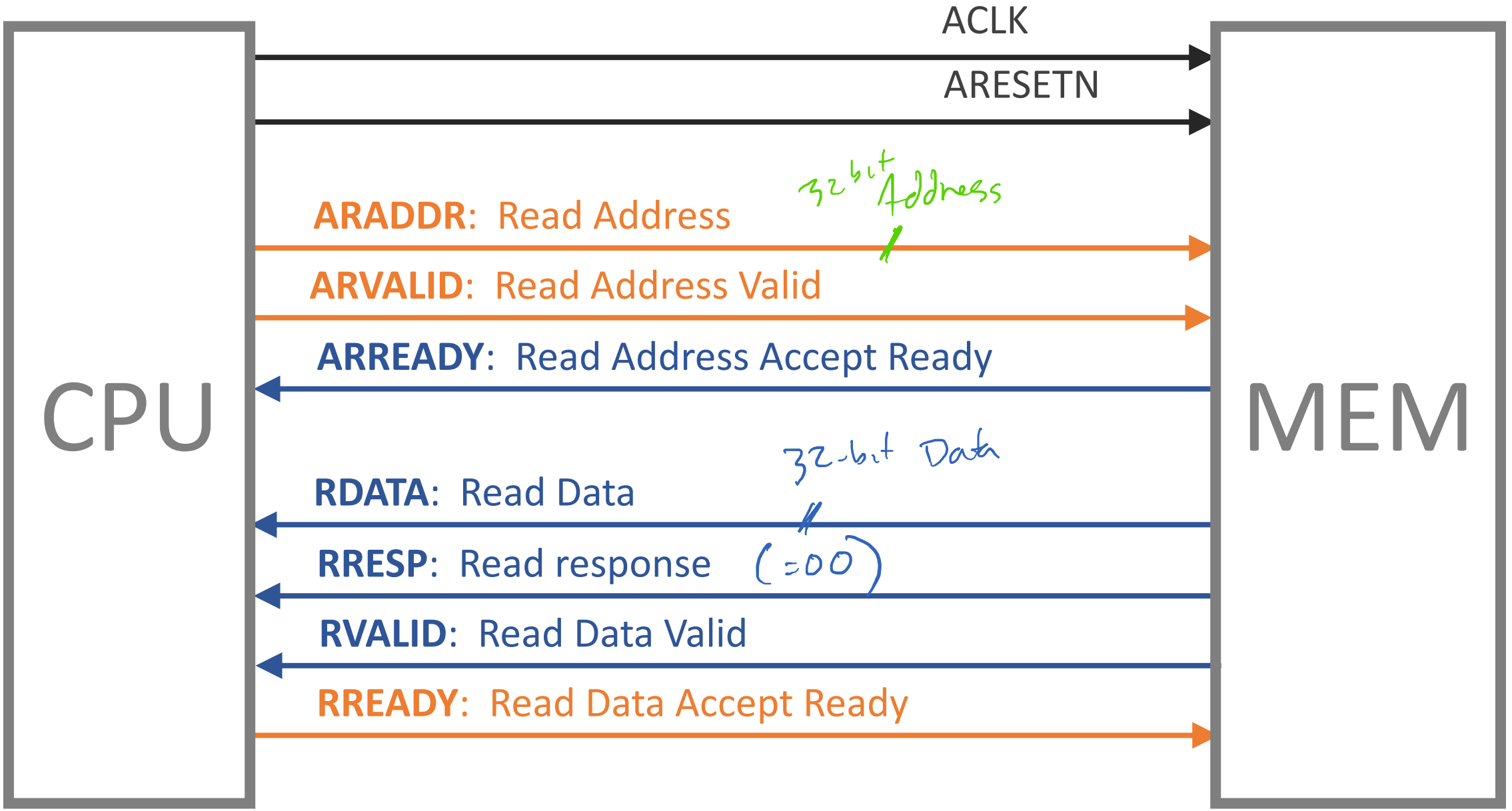
Figure A3-2 VALID before READY handshake

AXI4 Lite Read Transaction



What if?




ACLK
ARESETN
32-bit Address
ARADDR: Read Address
ARVALID: Read Address Valid
ARREADY: Read Address Accept Ready
CPU
MEM
32-bit Data
RDATA: Read Data
RRESP: Read response
(=00)
RVALID: Read Data Valid
RREADY: Read Data Accept Ready

What is RRESP?

Table A3-4 RRESP and BRESP encoding

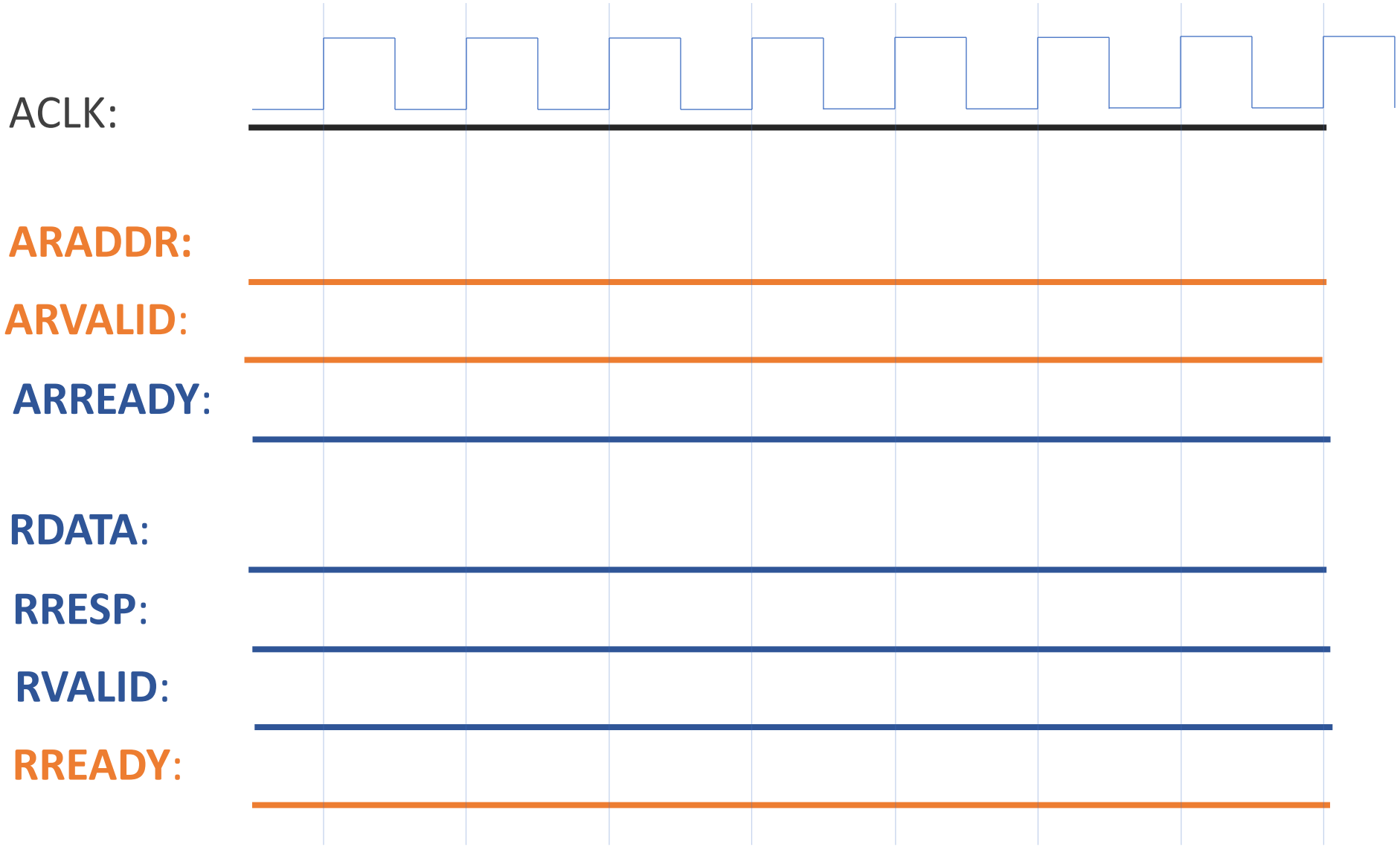
RRESP[1:0] BRESP[1:0]	Response
0b00	OKAY
0b01	EXOKAY
0b10	SLVERR
0b11	DECERR

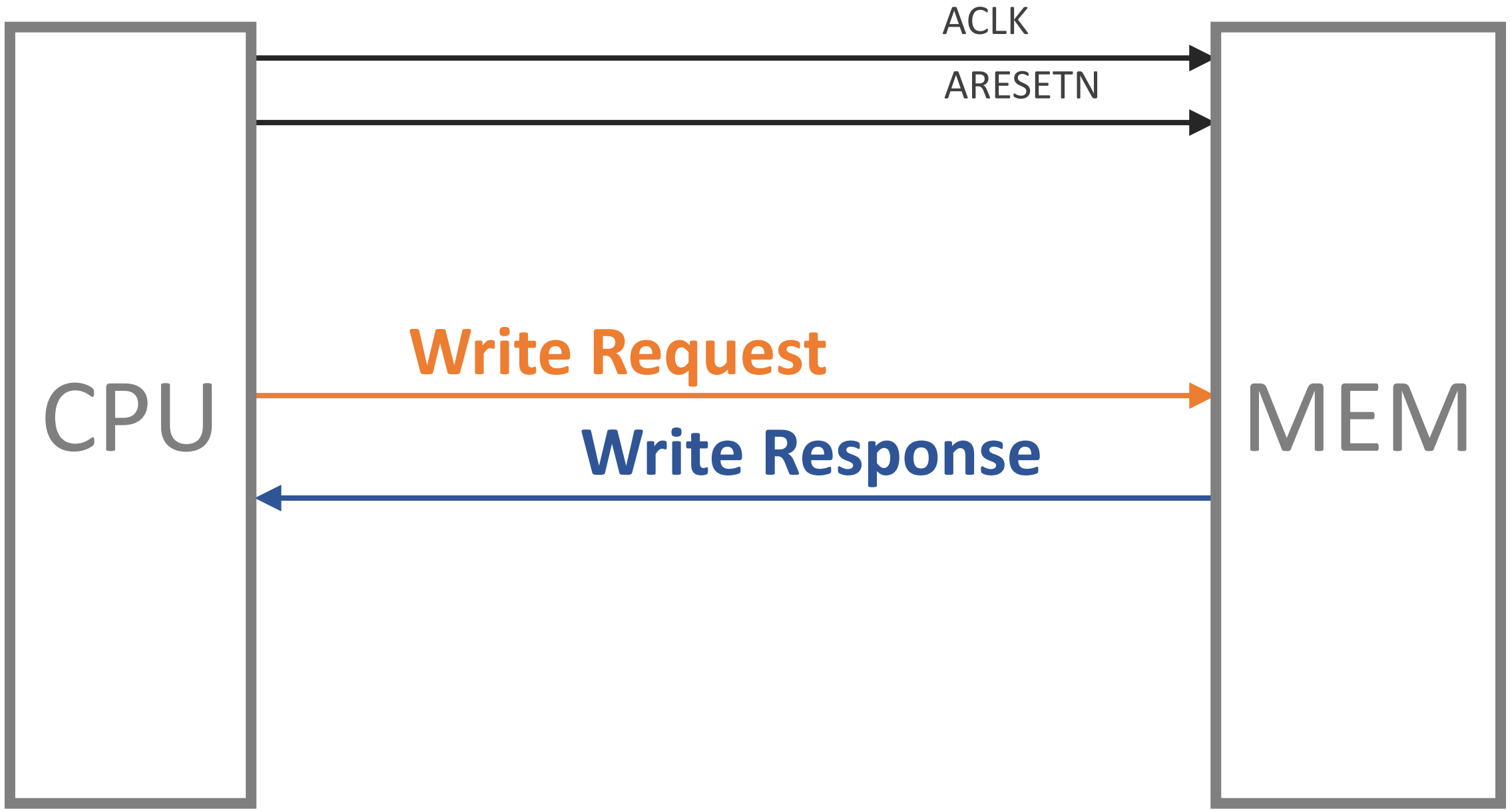
- Mostly used to send error codes back to CPU

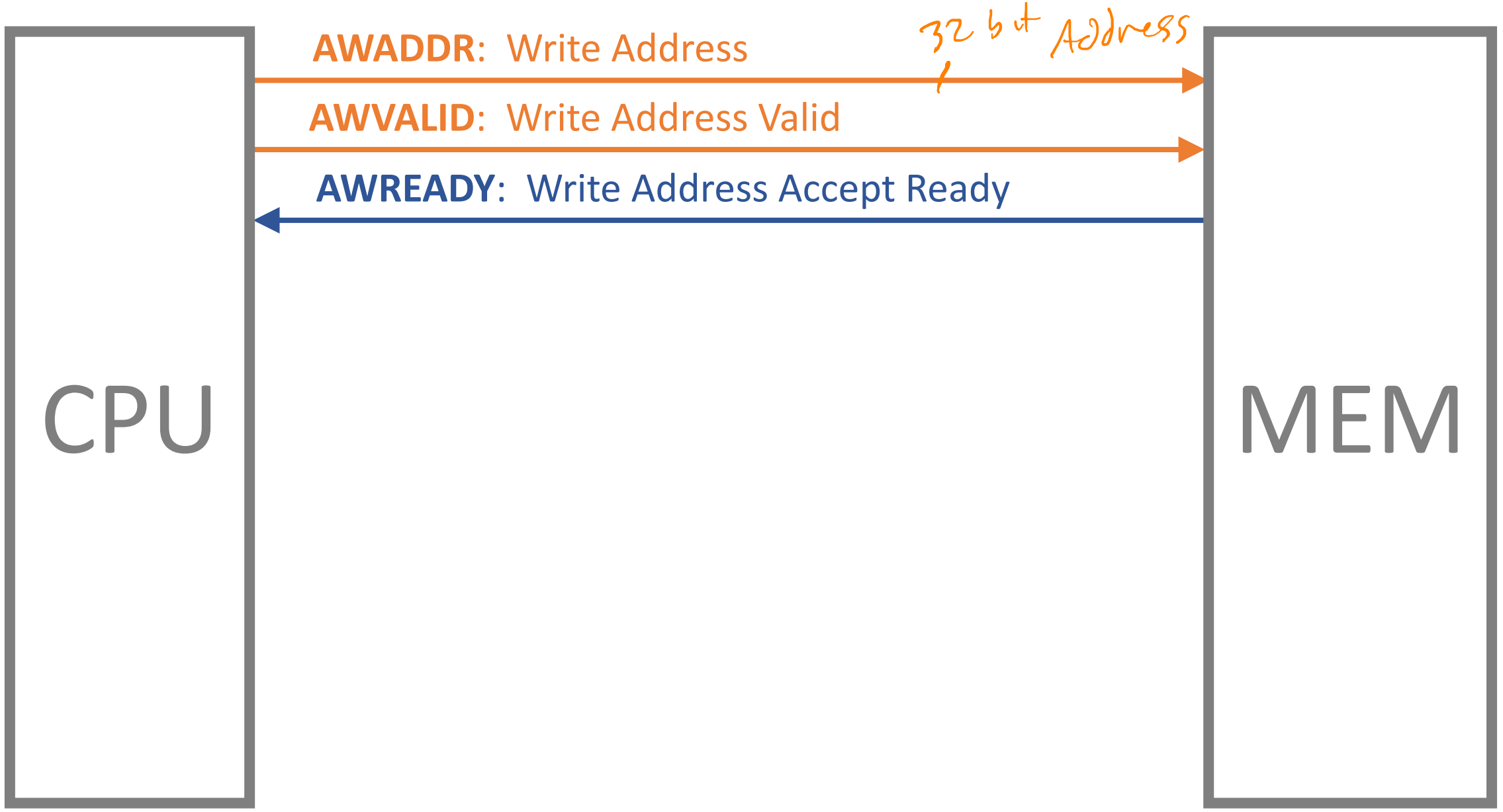
- We'll always just use 0b00

Load 0x1234, response: 0xabcd

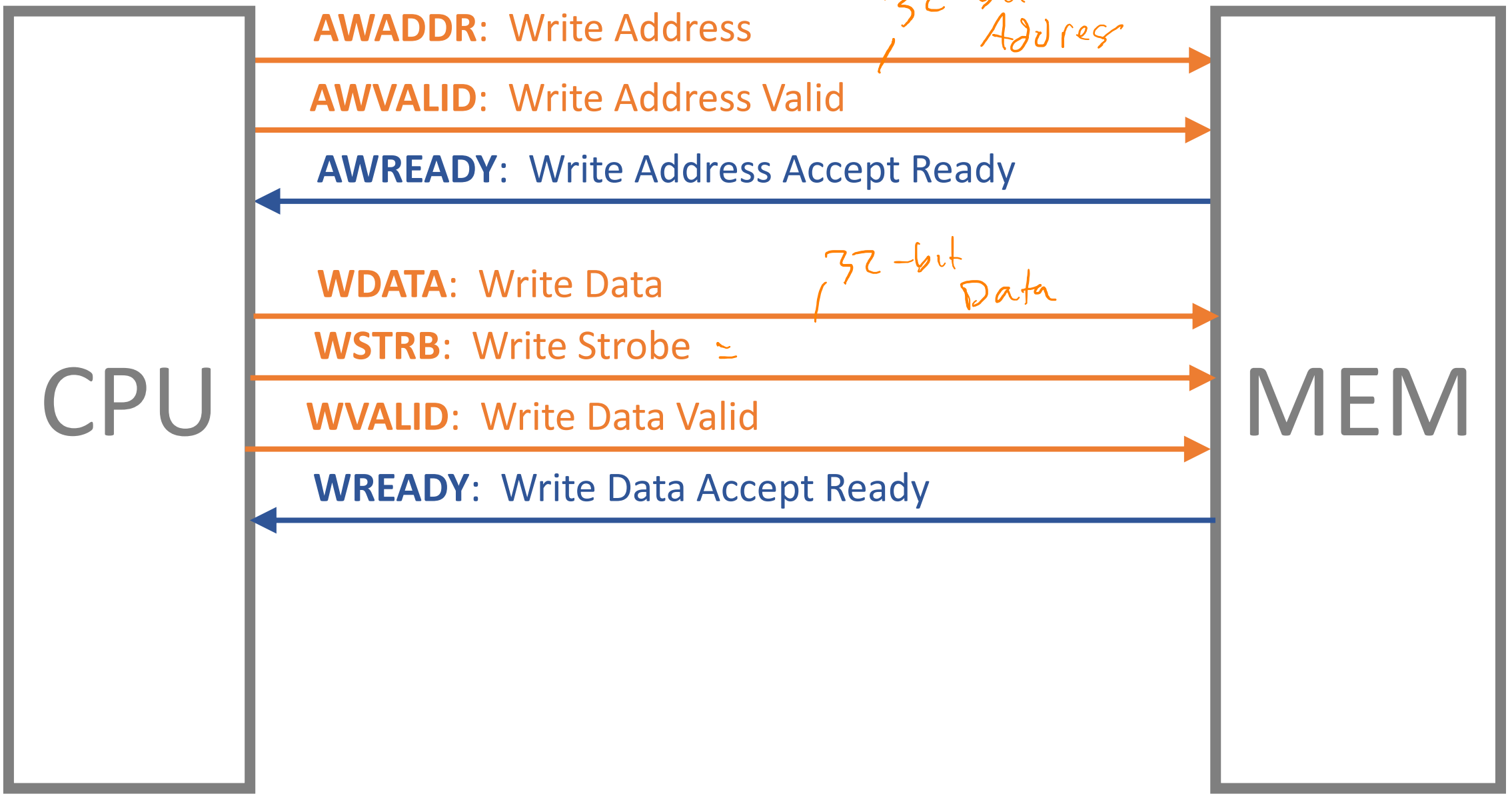
assume ARESETN = 1




ACLK
ARESETN
CPU
Write Request
Write Response
MEM



ACLK and ARESETN not shown



ACLK and ARESETN not shown

Q: How do you send a 1-byte (8-bit) value on a 32-bit bus?

- A: WSTB: Write Strobe

What is WSTRB?

The **WSTRB[n:0]** signals when HIGH, specify the byte lanes of the data bus that contain valid information. There is one write strobe for each eight bits of the write data bus, therefore **WSTRB[n]** corresponds to **WDATA[(8n)+7: (8n)]**

Just like TKEEP of AXI-Stream

What is WSTRB here?

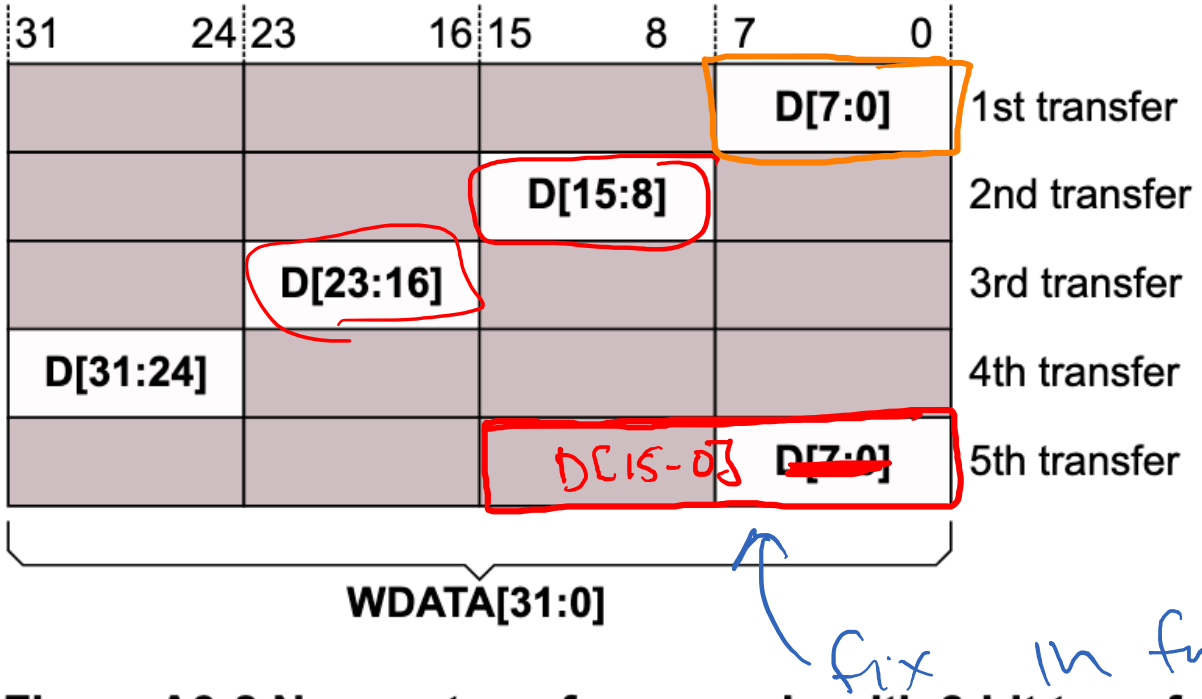


Figure A3-8 Narrow transfer example with 8-bit transfers

MSB LSB

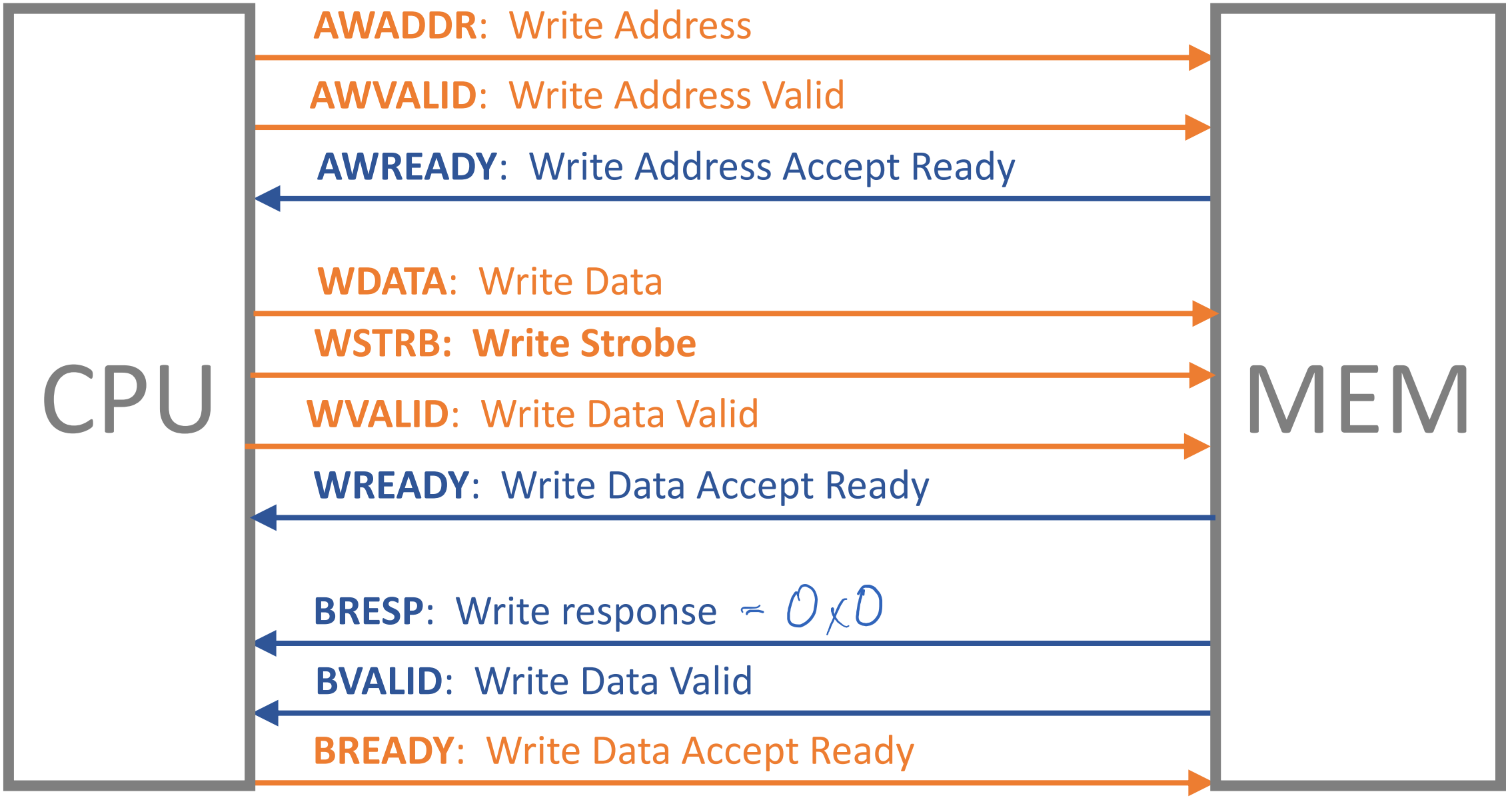
WSTRB = 0001 = 0x1

= 0010 = 0x2

= 0100 = 0x4

= 1000 = 0x8

= 0011 = 0x3



ACLK and ARESETN not shown

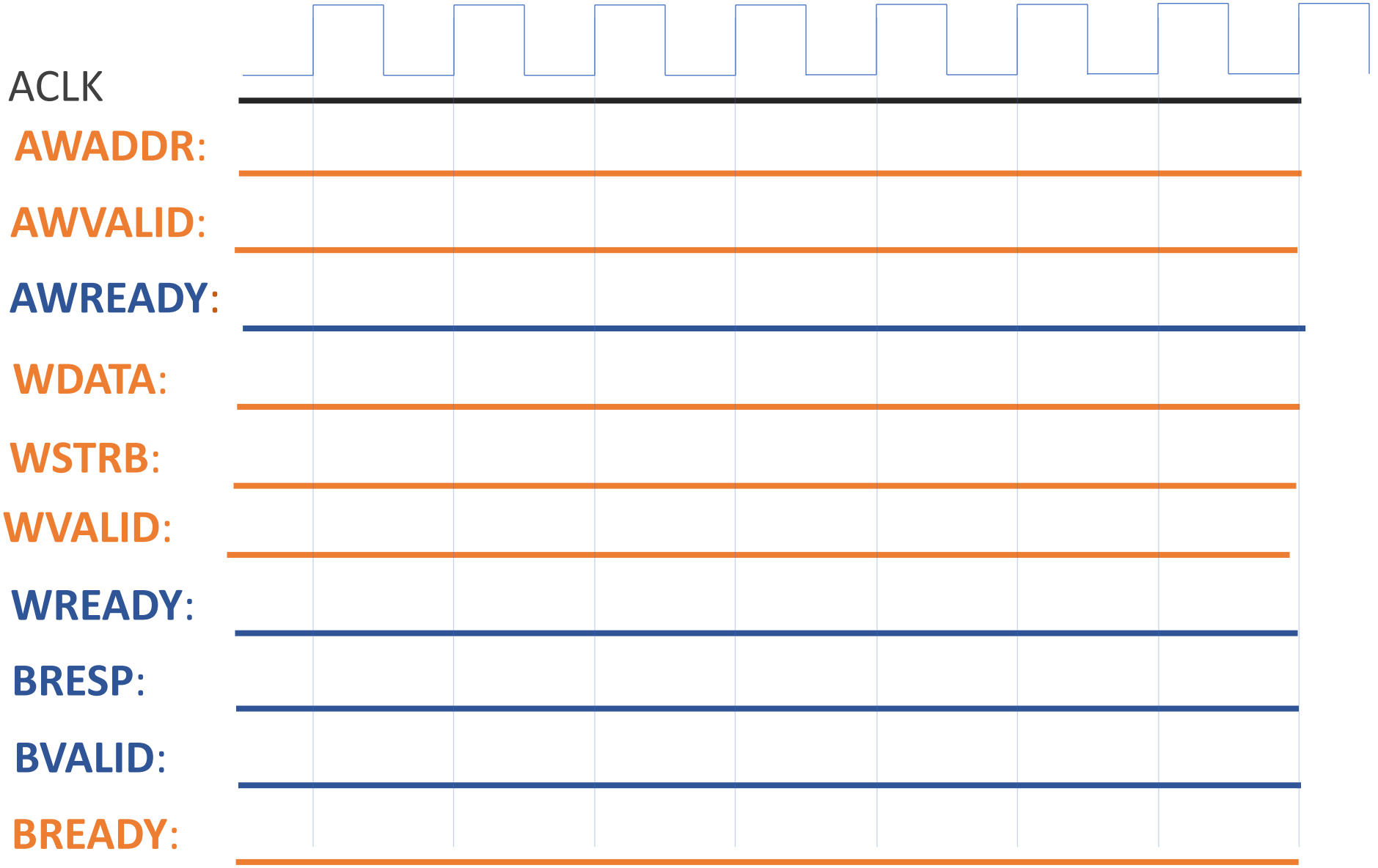
BRESP is just like RRESP

Table A3-4 RRESP and BRESP encoding

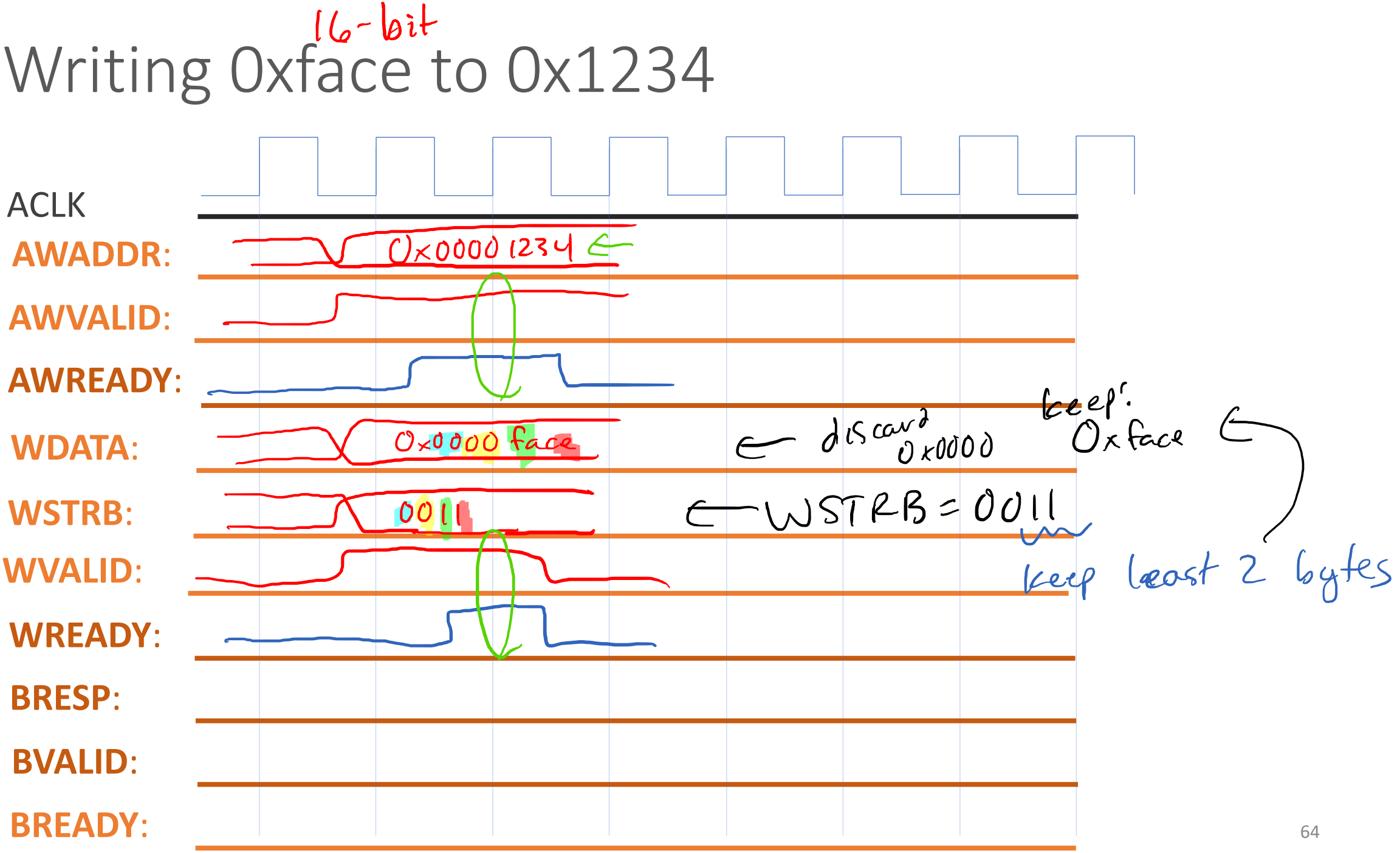
RRESP[1:0] BRESP[1:0]	Response
0b00	OKAY
0b01	EXOKAY
0b10	SLVERR
0b11	DECERR

- Mostly used to send error codes back to CPU
- We'll always just use 0b00

Writing 0xdeadbeef to 0x1234



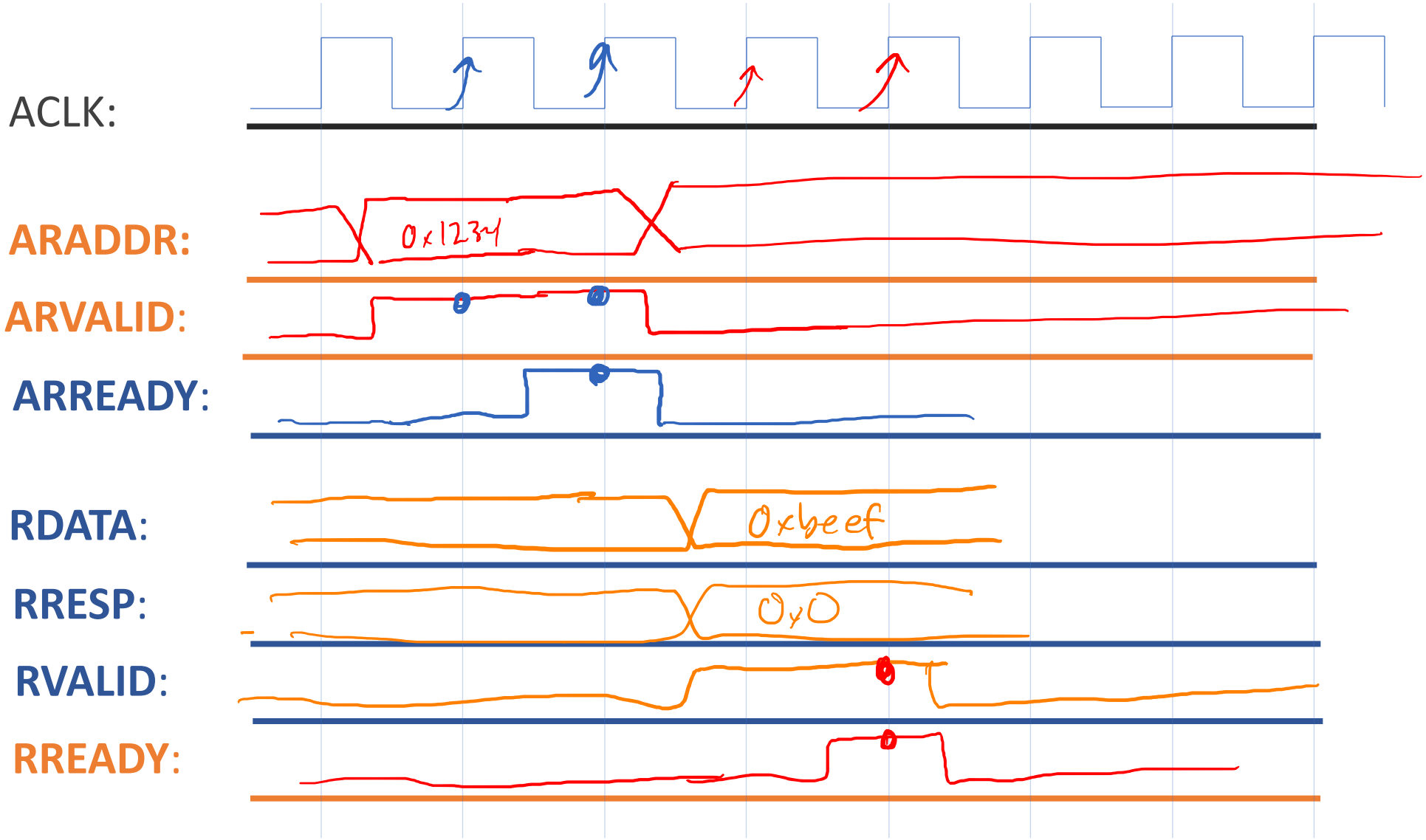
Writing 0xface to 0x1234



ARM AXI Bus

- "Advanced eXtensible Interface" Bus Version 4, "AXI4"
- Three Variants
  - AXI4: Fast but complicated; Memory-mapped
  - AXI4 Lite: Slow but simple; Memory-mapped
  - AXI4 Stream: Fast and simple; Not memory-mapped

How long does a read(load) take?



High-Performance Bus Ideas

- Make single transaction faster

AXI Handshake Speedup

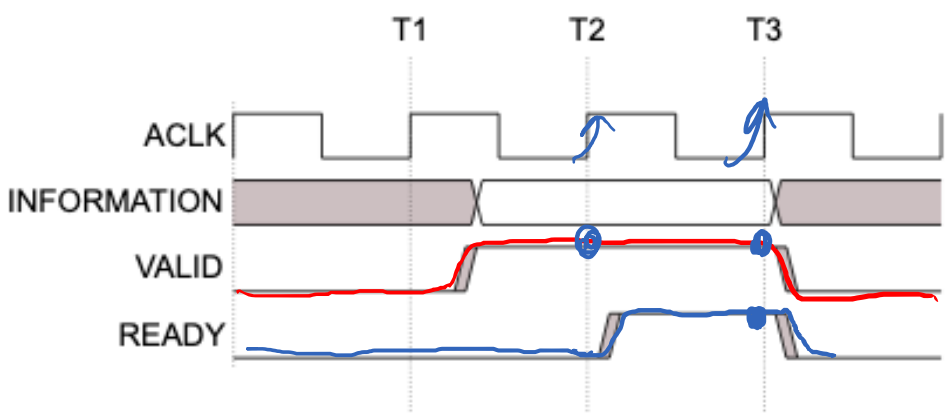


Figure A3-2 VALID before READY handshake

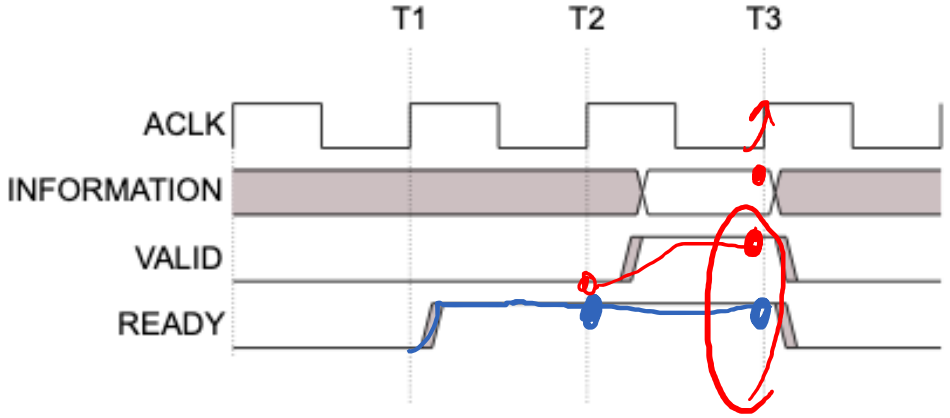
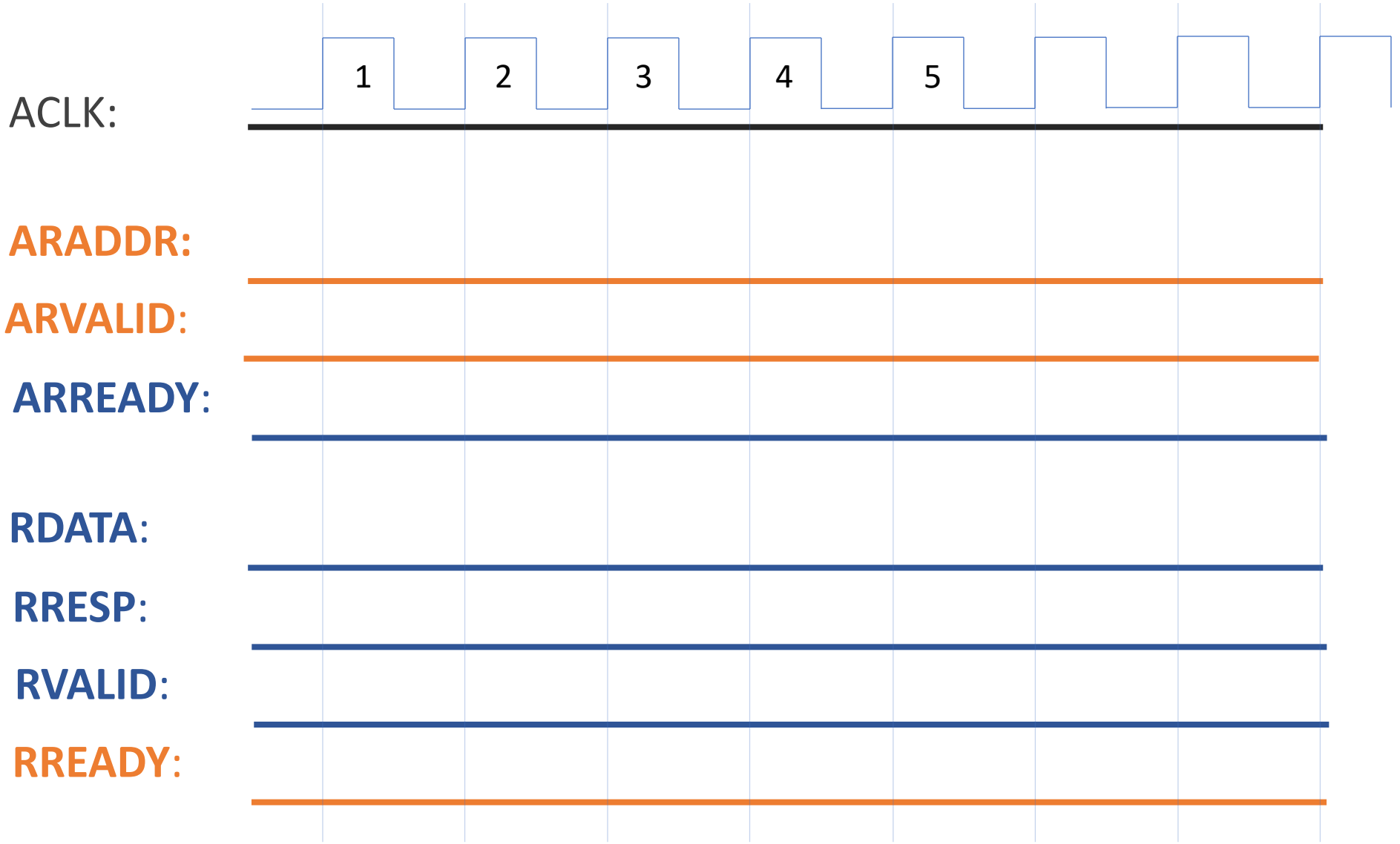


Figure A3-3 READY before VALID handshake

- Both are valid
- Right is faster

What can we do to make this faster?



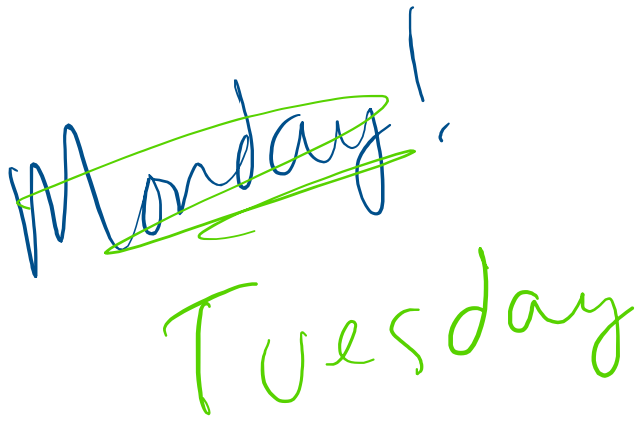
High-Performance Bus Ideas

- Make single transaction faster

- Overlap multiple transactions

Next Time

- High-Performance Busses



References

- https://www.youtube.com/watch?v=okiTzvihHRA
- https://web.eecs.umich.edu/~prabal/teaching/eecs373/
- https://en.wikipedia.org/wiki/File:Computer_system_bus.svg
- https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081

- AMBA® AXI™ and ACE™ ProtocolSpecification

08: AXI4 Lite

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University