Test

08: AXI4 Lite

Engr 315: Hardware / Software Codesign
Andrew Lukefahr
Indiana University

Announcements

@ Andrew: post slides 5-8

- ~~P2: Demo due by Friday~~
- P3: Out now! → due Friday
  - Need a Pynq
  - Groups of 2 allowed
- P4: Out soon…

Project 3 Testbench

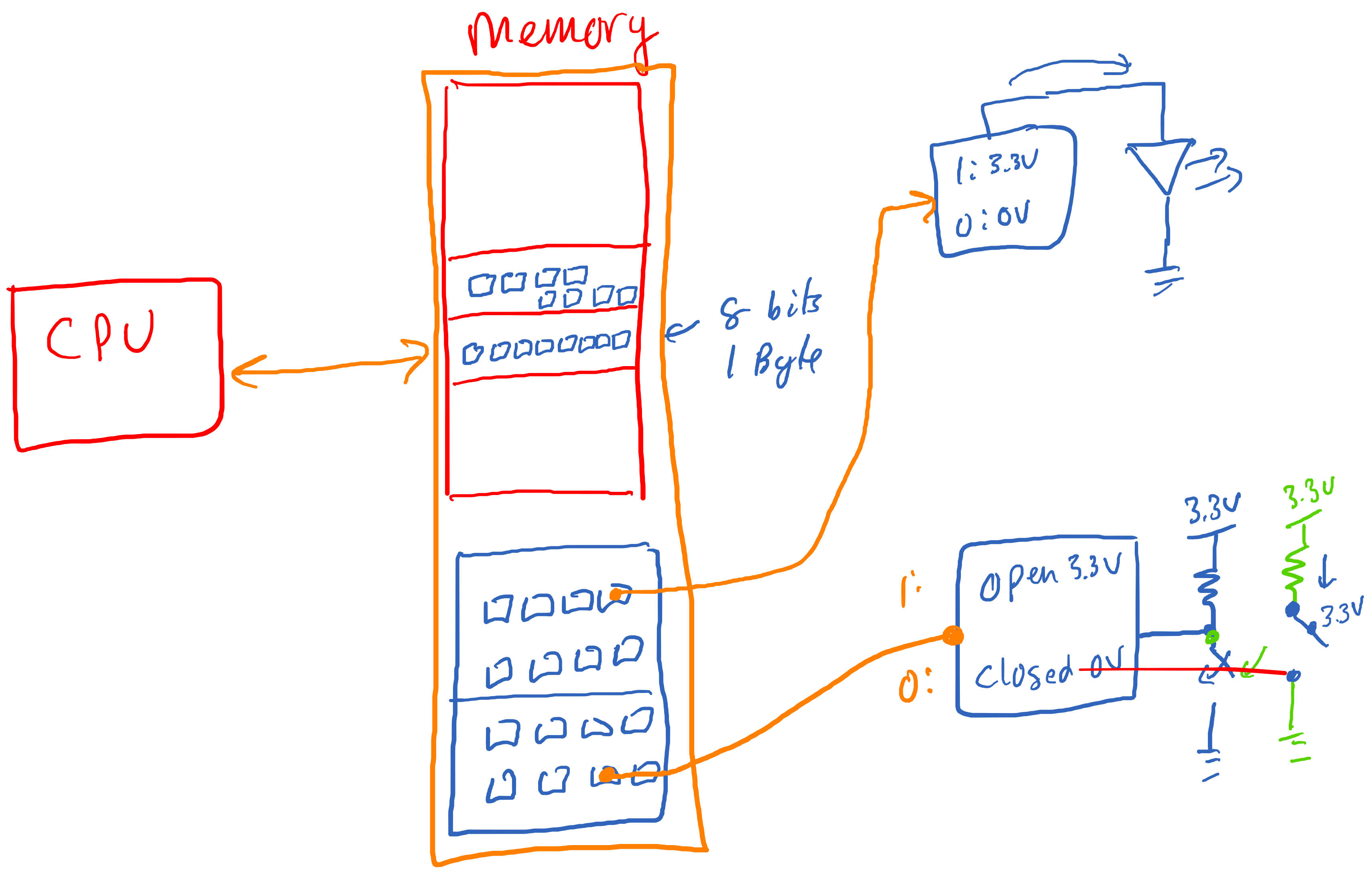
Optimizations thus far

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

↳ Bus

↳ MMIO

Review: Memory-Mapped I/O

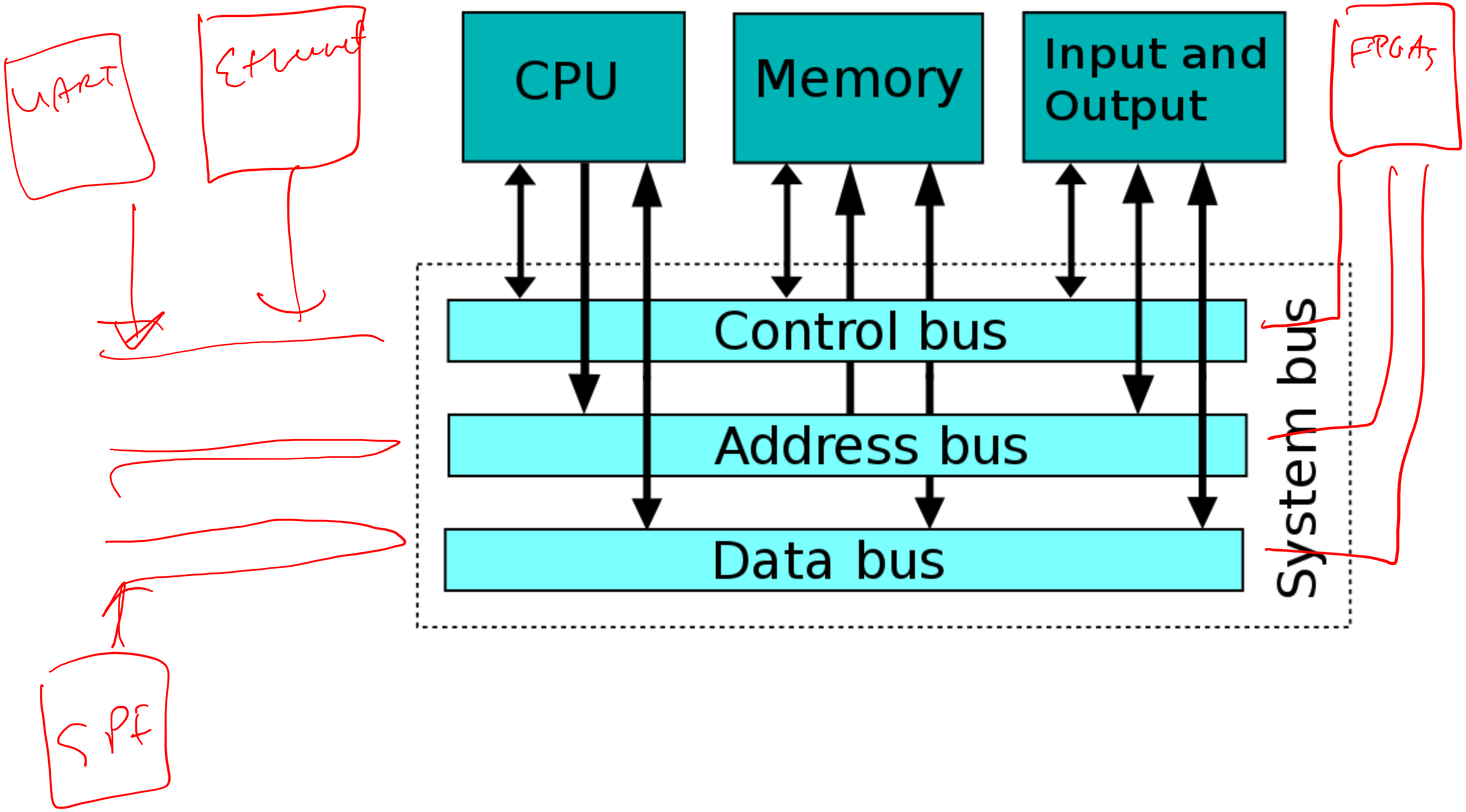


Use volatile for MMIO addresses!

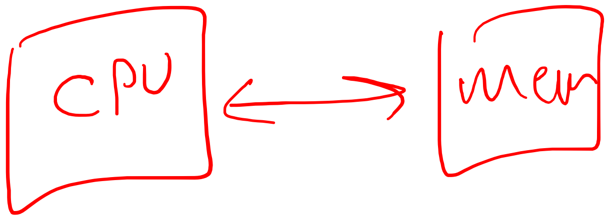
```
#define SW_ADDR 0xfffe
volatile uint32_t * SW_REG = (uint32_t * SW_ADDR);

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

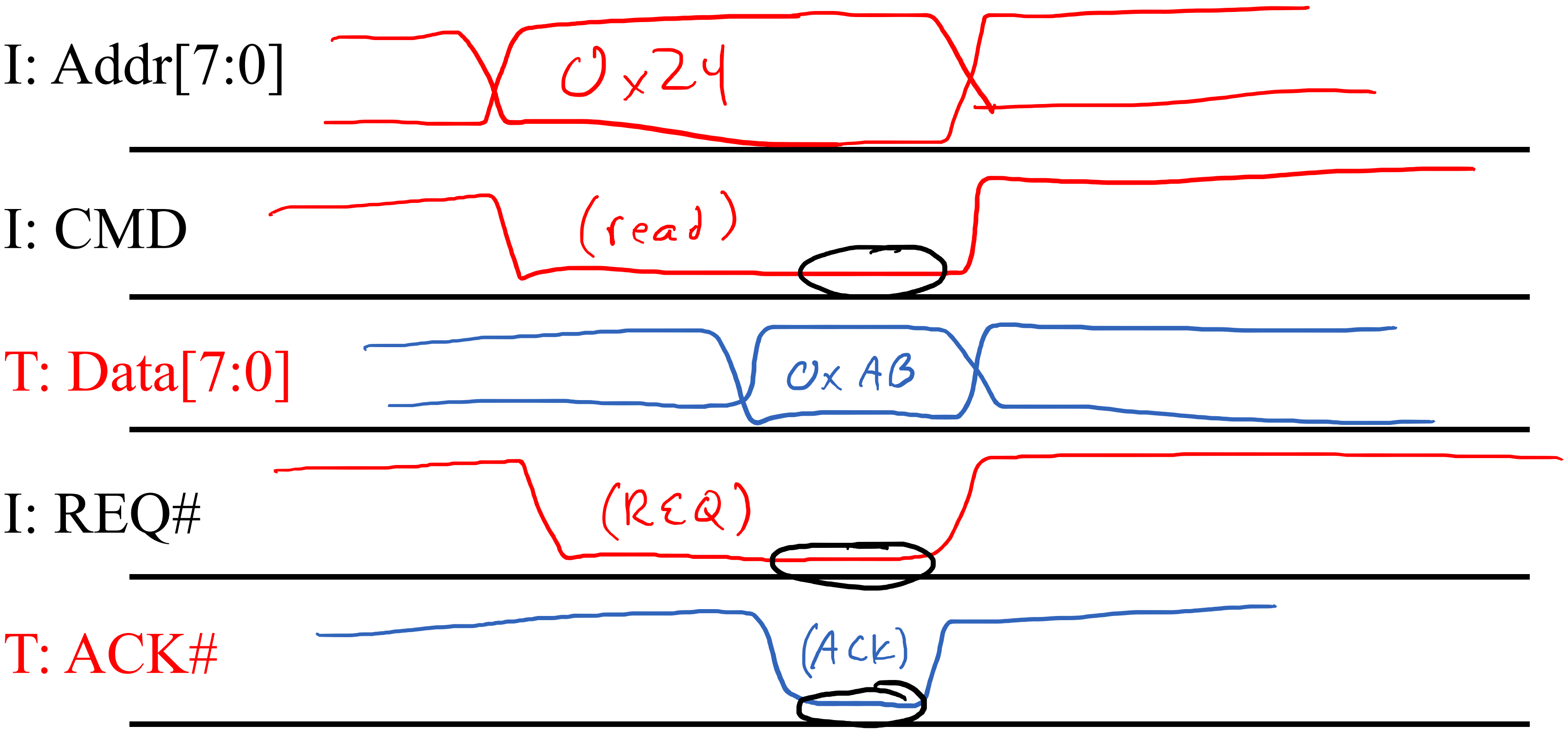
- Signals
  - Addr[7:0], Data[7:0], CMD, REQ#, ACK#
    - CMD=0 is read, CMD=1 is write.
    - REQ# low means initiator is **requesting** something.
    - ACK# low means target is **acknowledging** the job is done.

CPU		Mem
Load	=	read
Store	=	write

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

Read transaction

(CPU) load
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

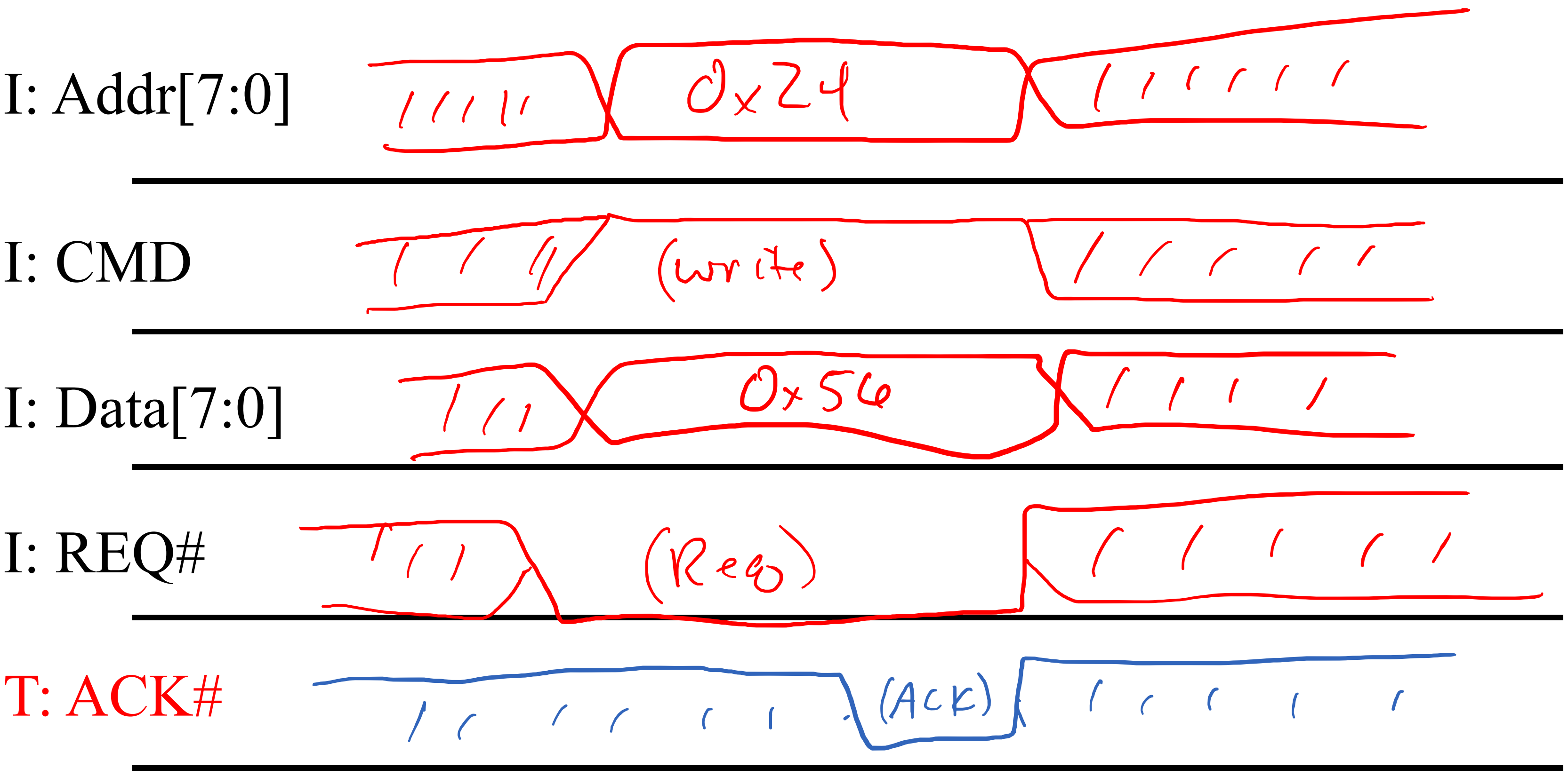
CPU store

(Mem)

Write transaction

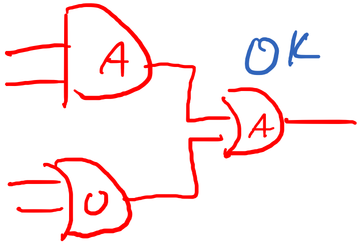
Initiator wants to write 0x56 to location 0x24

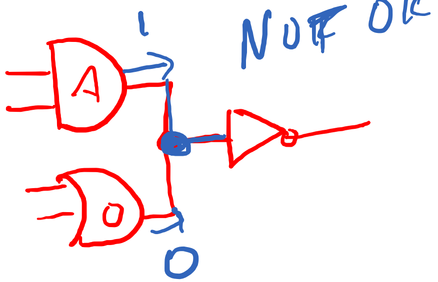
CMD=0 is read, CMD=1 is write.
REQ# low means initiator is requesting.
ACK# low means target is acknowledging.

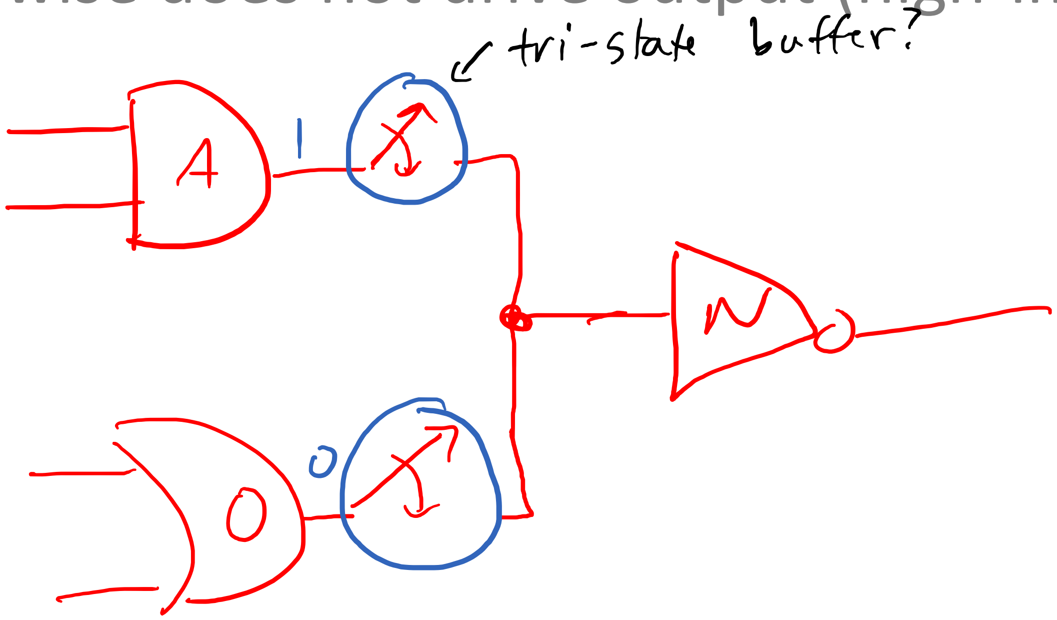


Tri-State Buffer

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



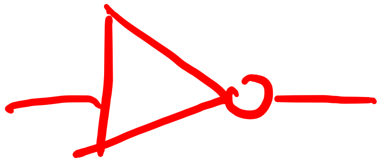




close both = bad!

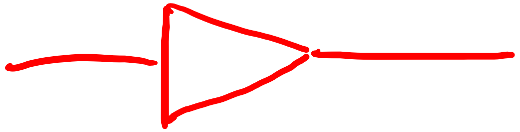
only close 1, ok?

NOT



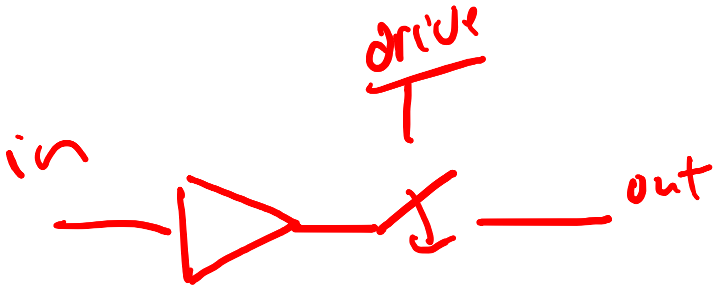
in	out
0	1
1	0

BUFF



0	0
1	1

TBUFF



in	drive	out
0	1	0
1	1	1
x	0	high-Z

Can MMIO behave as memory?

Example peripherals

0x04: Push Button - Read-Only

Pushed -> 1

Not Pushed -> 0

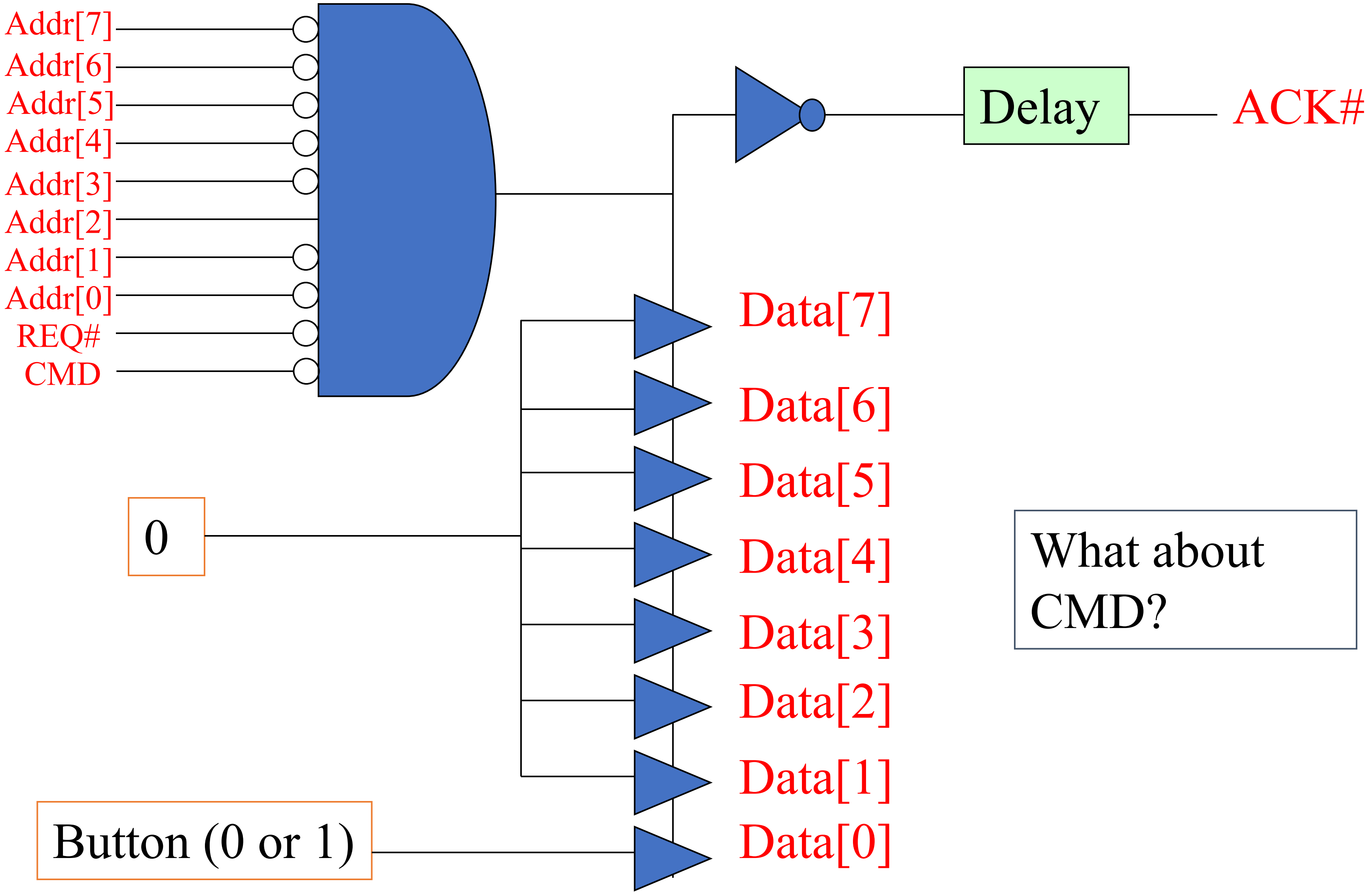
0x05: LED Driver - Write-Only

On -> 1

Off -> 0

The push-button

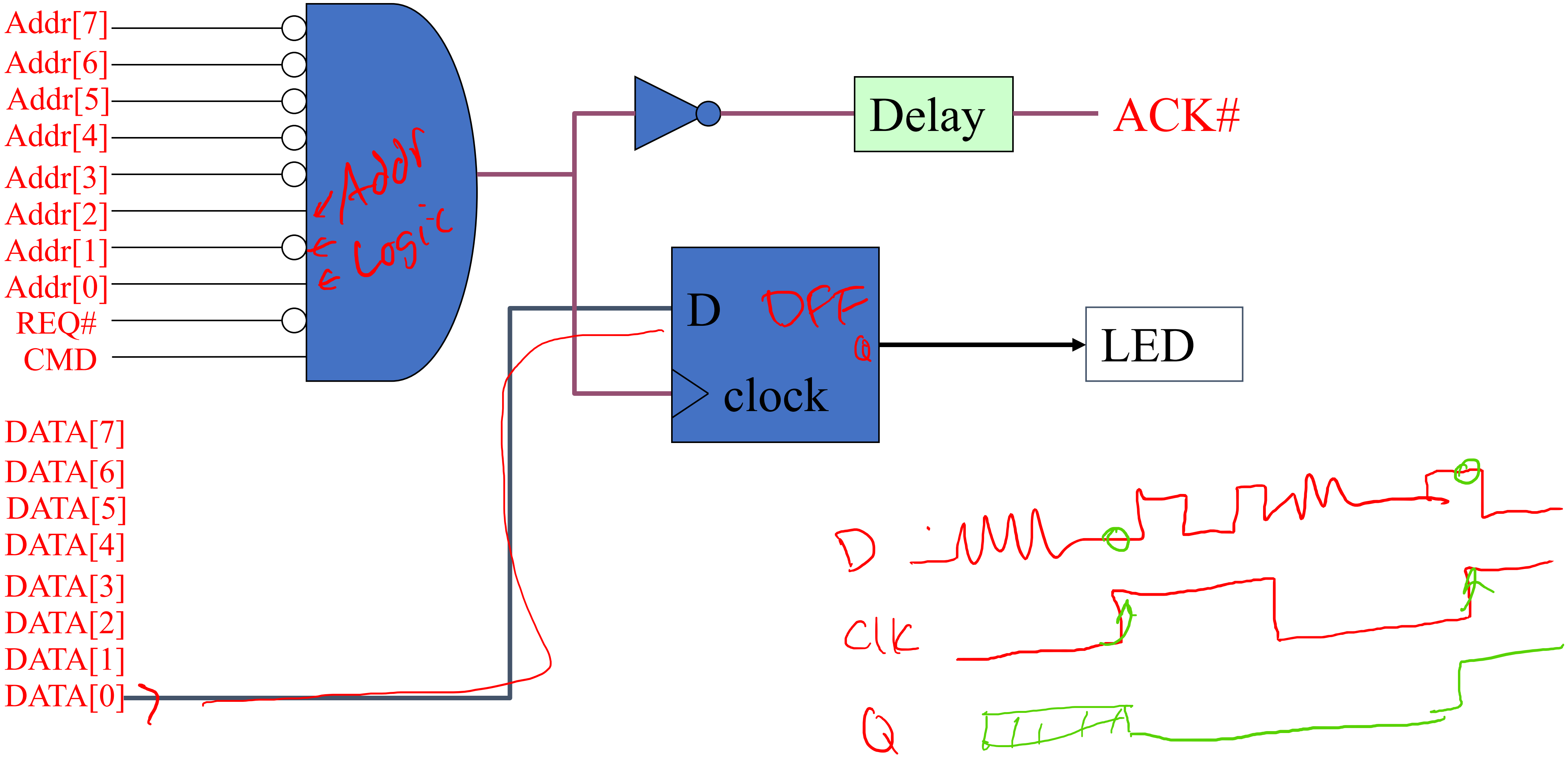
(if Addr=0x04 read 0 or 1 depending on button)



What about CMD?

The LED

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



Let's write a simple ~~C~~ Asm 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

```
      mov r0, 0x4
      mov r1, 0x5
loop: ldr r2, [r0]
      str r2, [r1]
      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

```
for(;;)
while (1)
```

```
#define PB 0x4
#define LED 0x5
int main () {
volatile uint8_t * PB_reg = (uint8_t *)(PB);
volatile uint8_t * LED_reg = (uint8_t *)(LED)
while (1) {
    *LED_reg = *PB_reg;
}
}
```

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

ARM AXI Bus

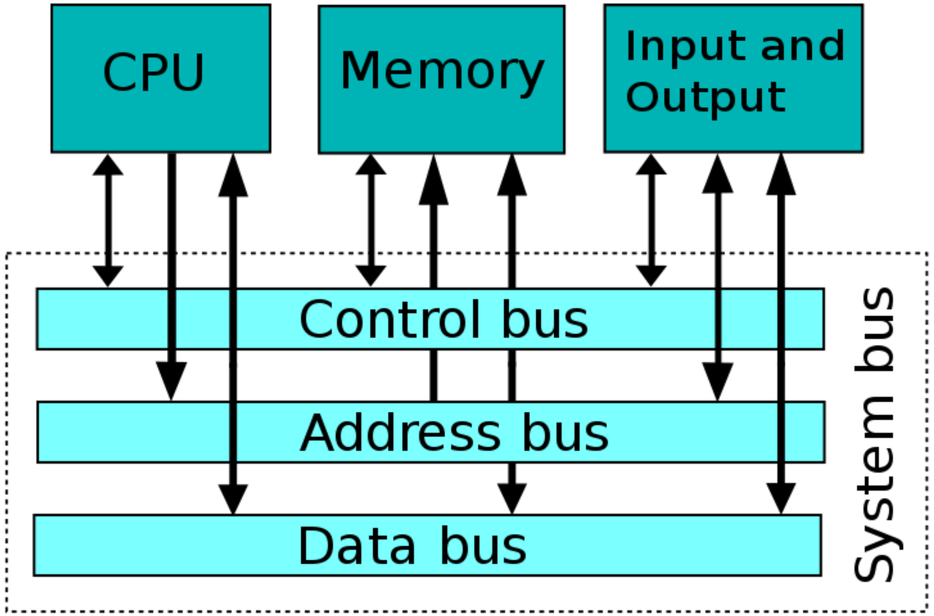
- “Advanced eXtensible Interface” Bus Version 4, “AXI4”

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

    P3 ↑
  - AXI4 Stream: Fast and simple; Not memory-mapped
    - ~~P3~~ P2 uses this

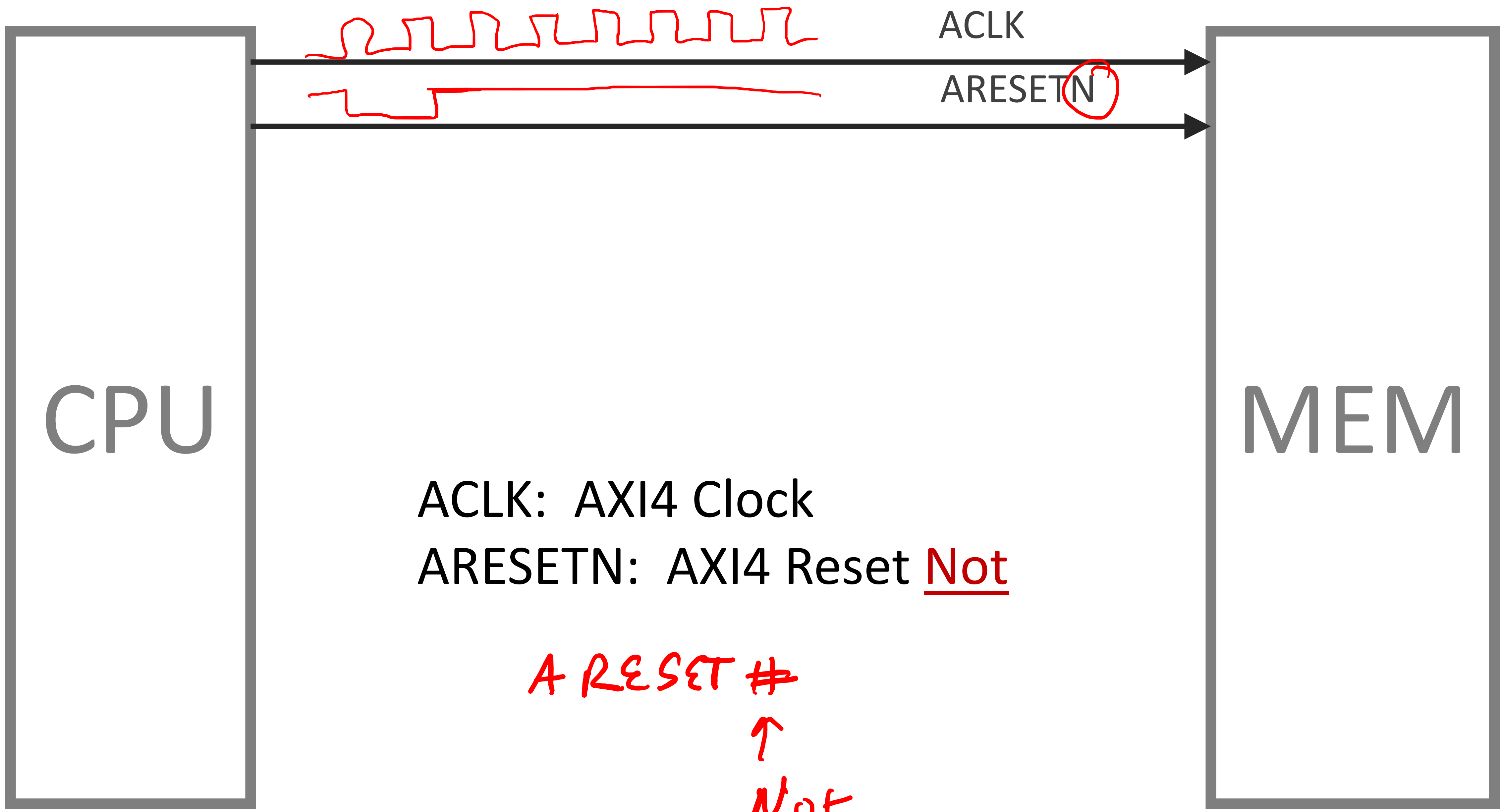
Why AXI4 Lite?



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



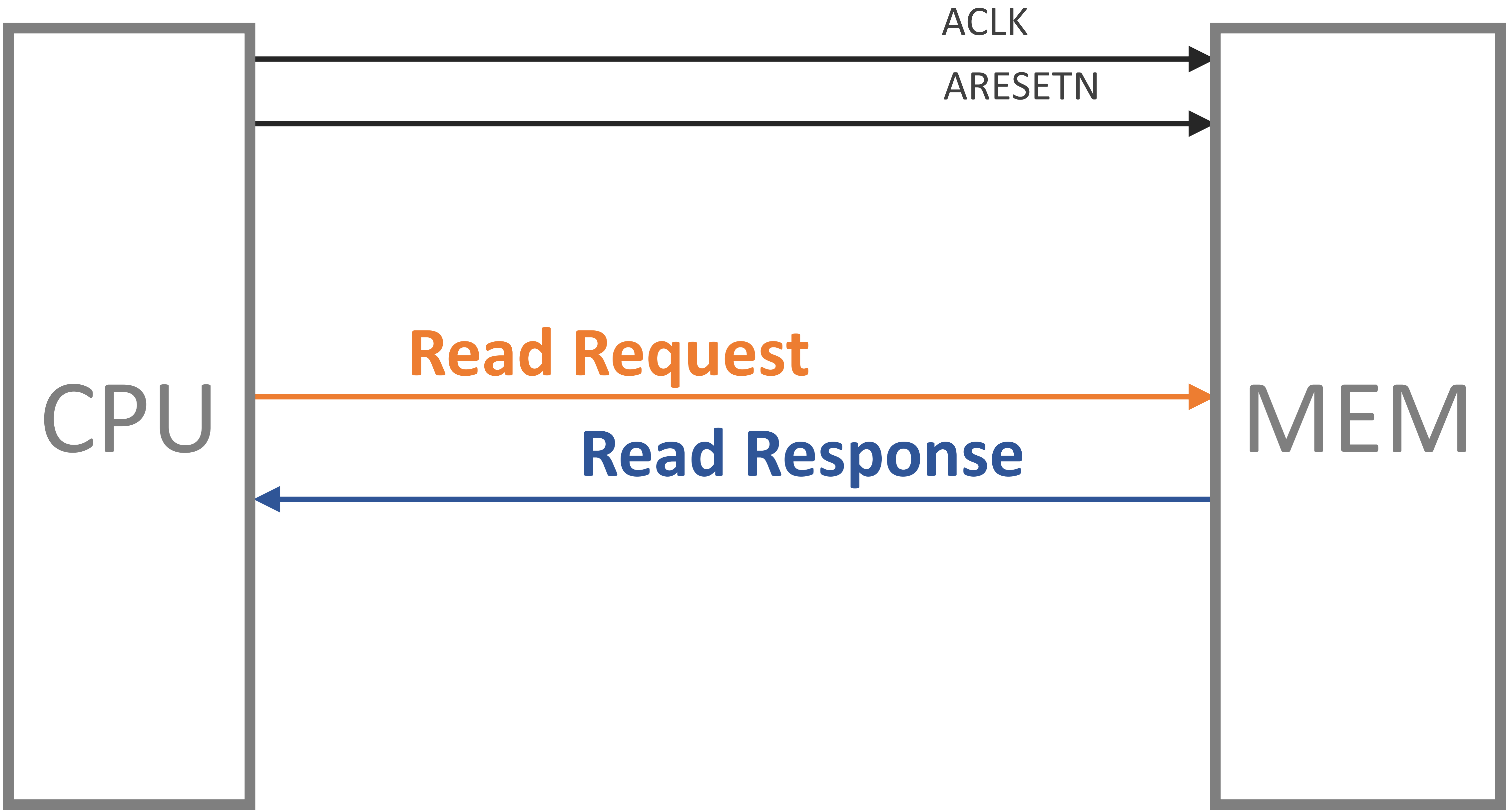
ACLK: AXI4 Clock

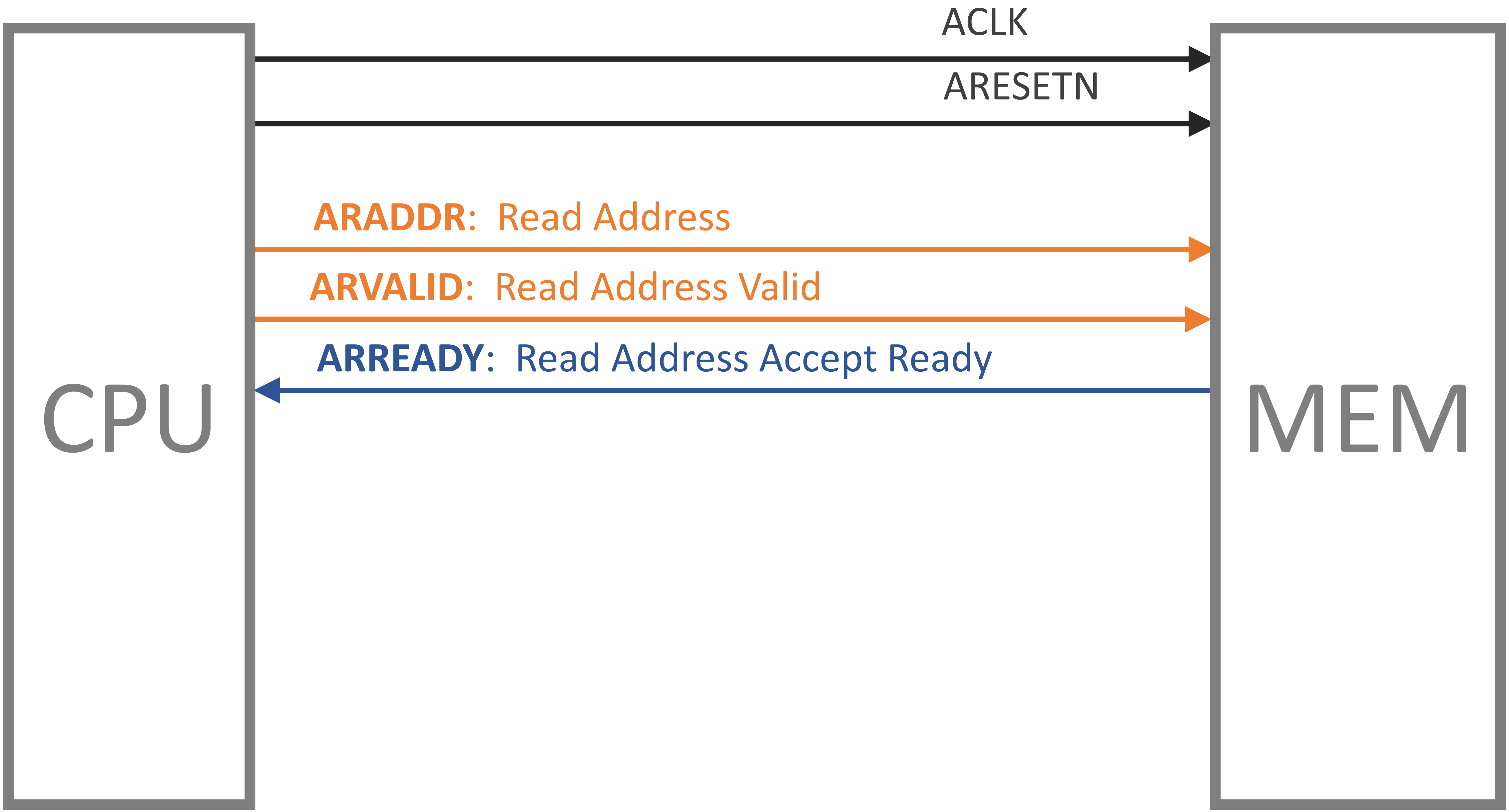
ARESETN: AXI4 Reset Not

ARESET#

↑

Not


ACLK
ARESETN
Read Request
Read Response
CPU
MEM


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

AXI4 Handshaking

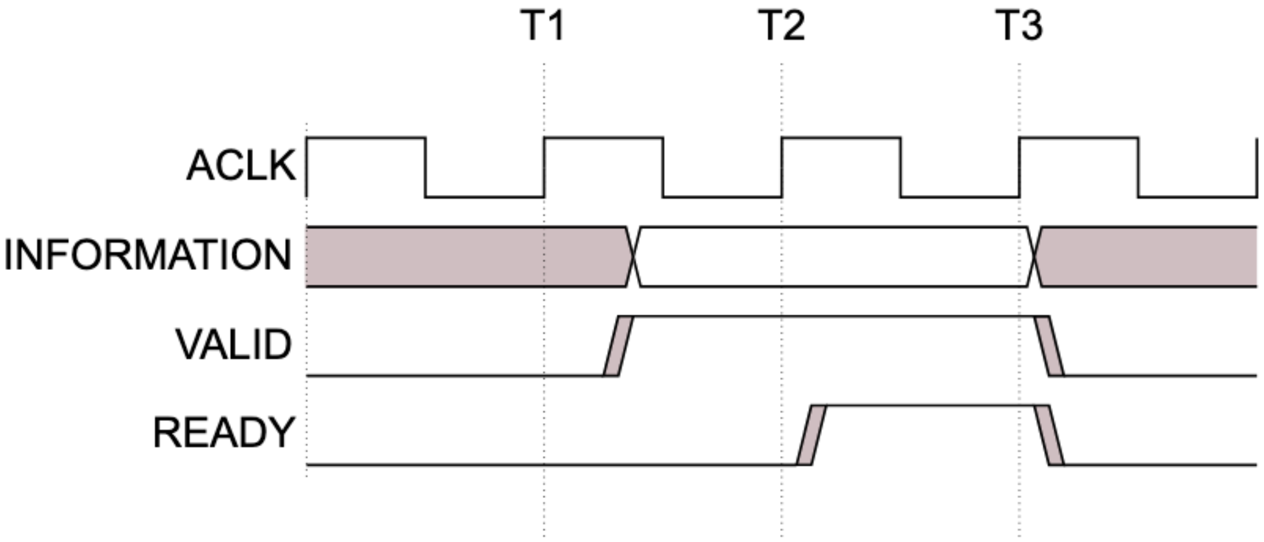
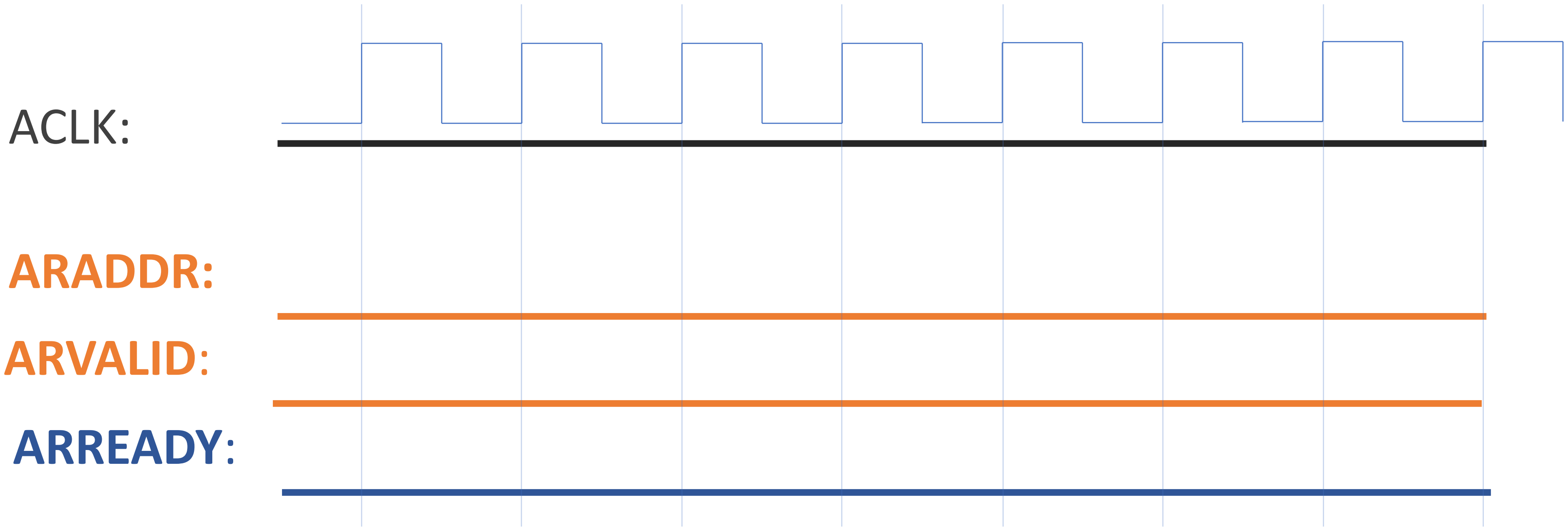
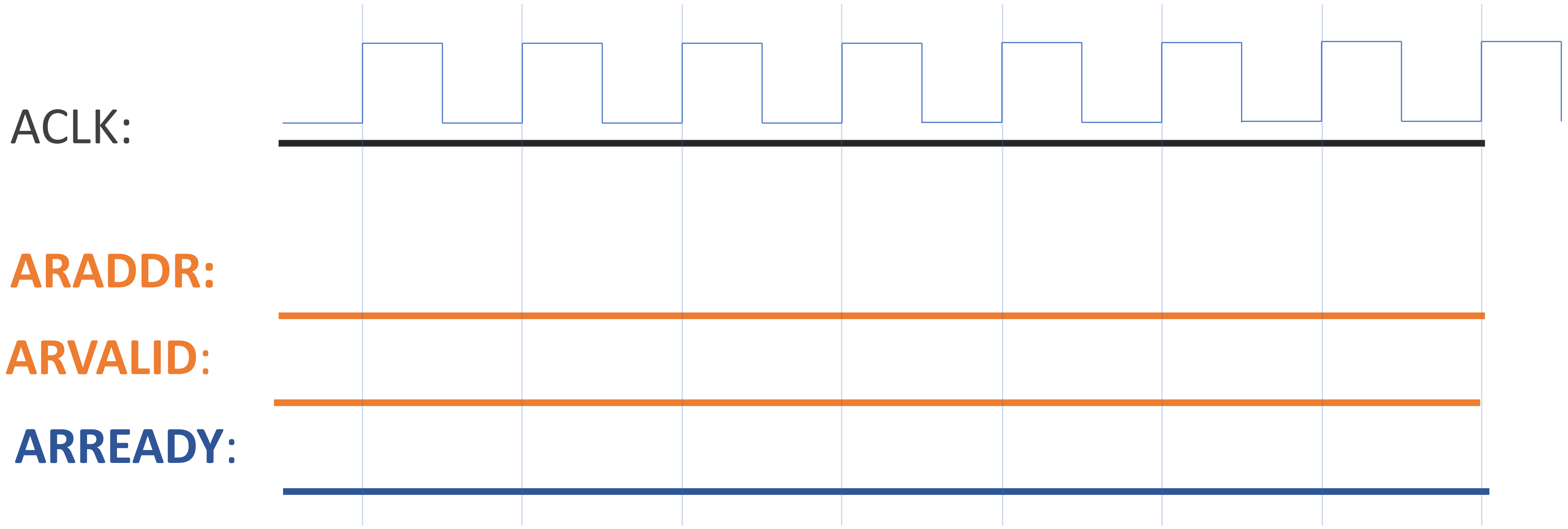


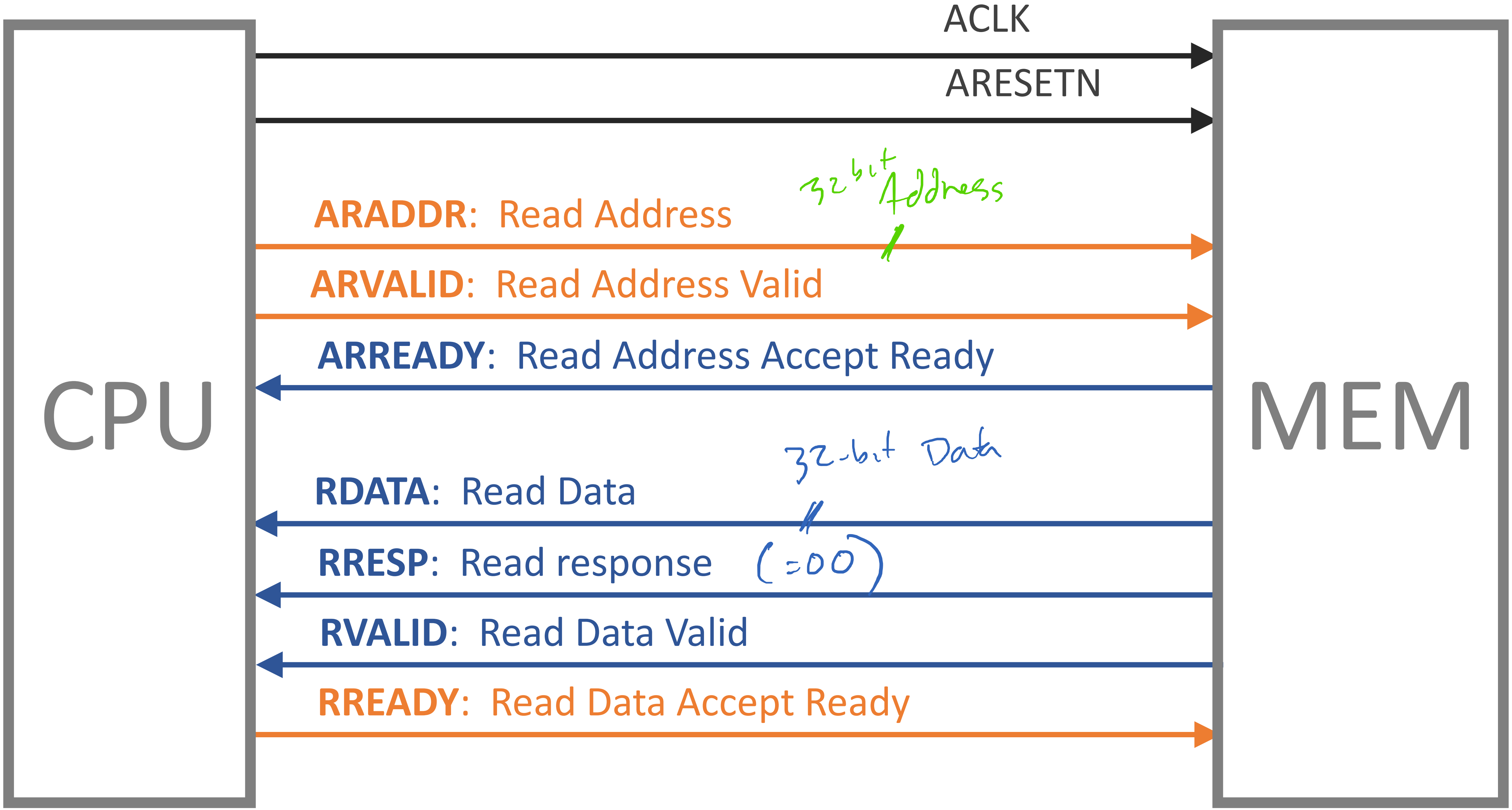
Figure A3-2 VALID before READY handshake

AXI4 Lite Read Transaction



What if?




ACLK
ARESETN
32bit 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

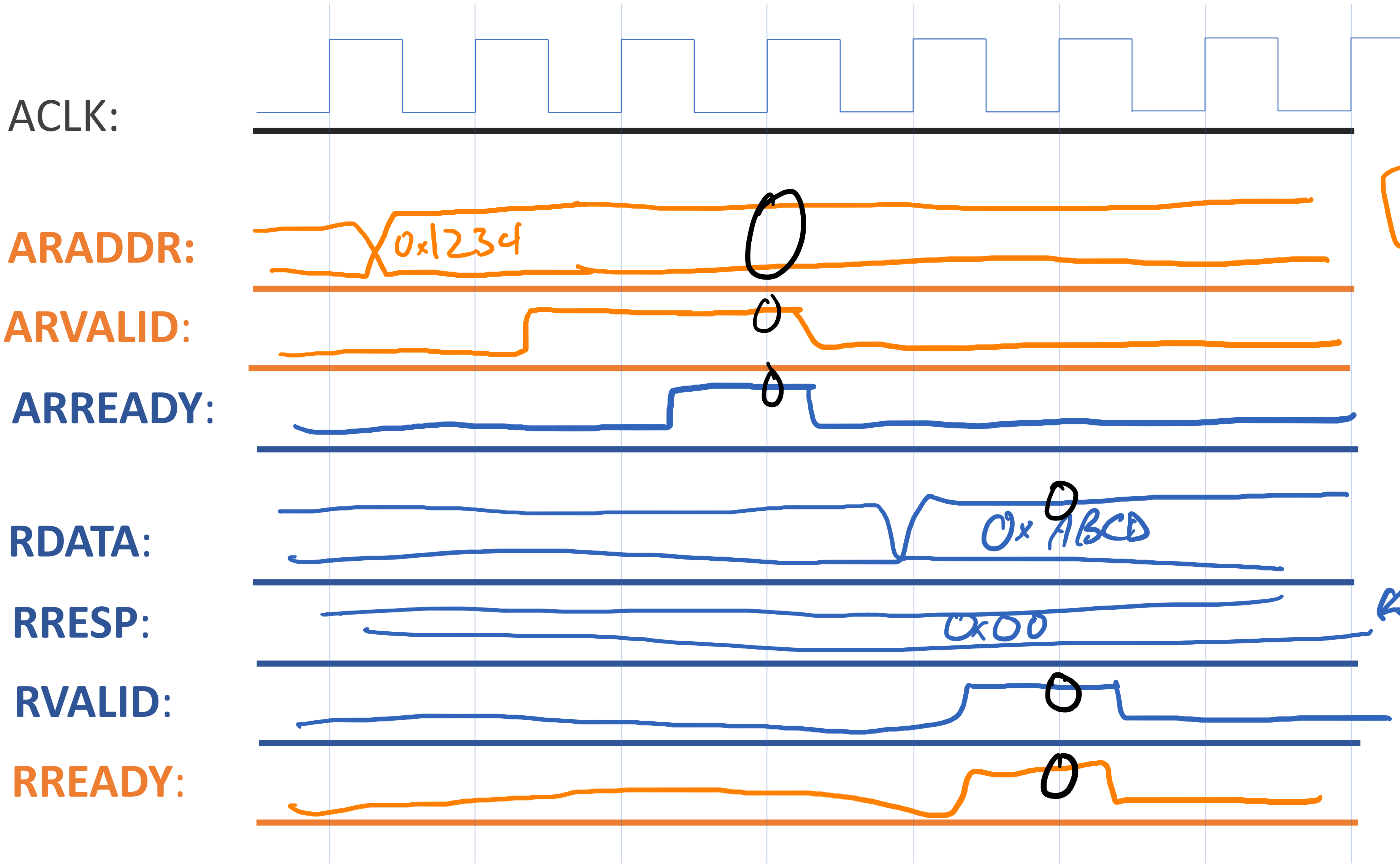
RRESP[1:0] BRESP[1:0]	Response
0b00	OKAY
0b01	EXOKAY
0b10	SLVERR
0b11	DECERR

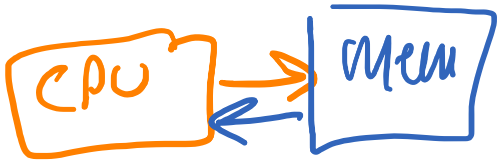
- Mostly used to send error codes back to CPU

- We'll always just use 0b00

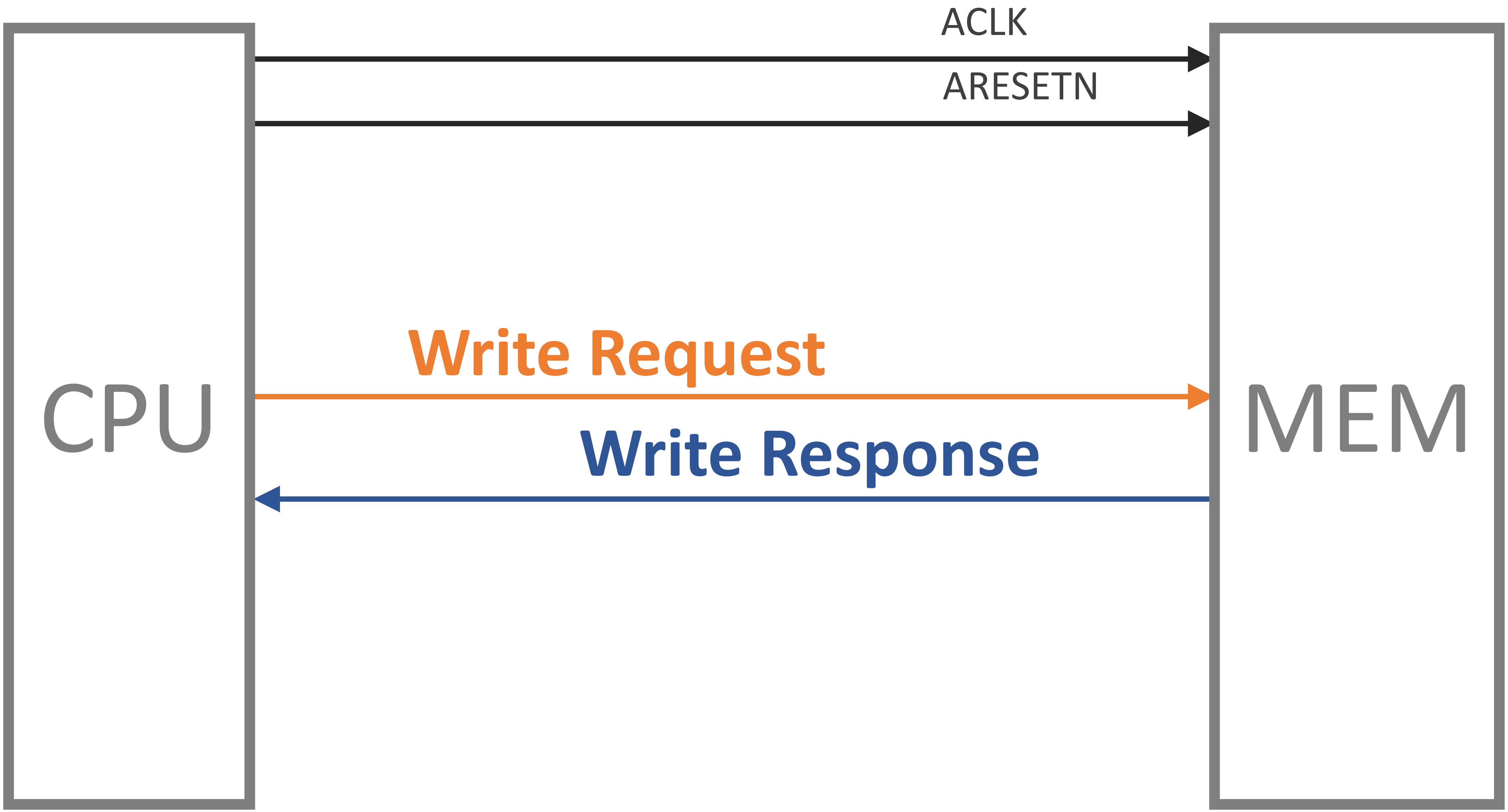
Load 0x1234, response: 0xabcd

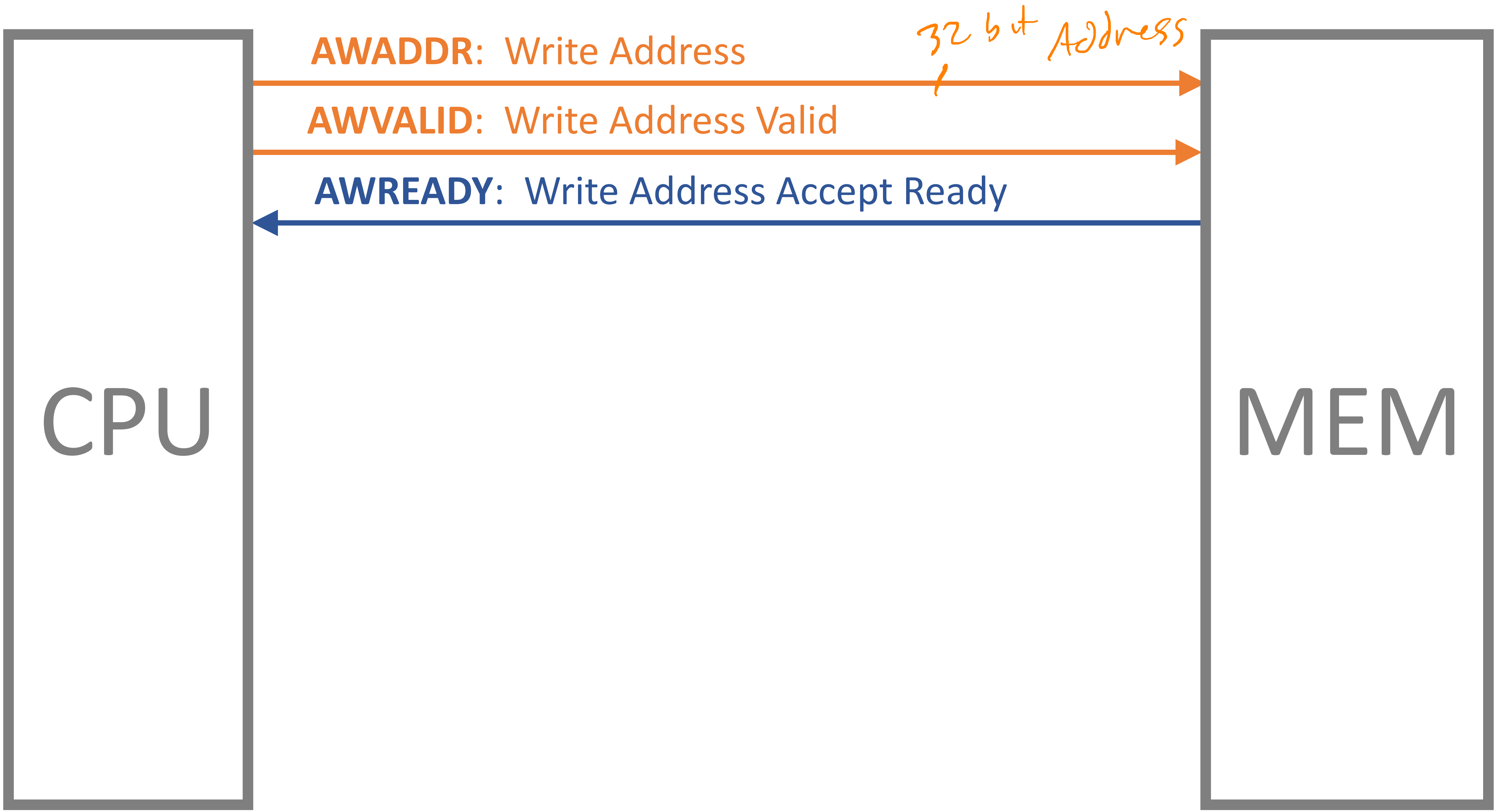
assume ARESETN=1



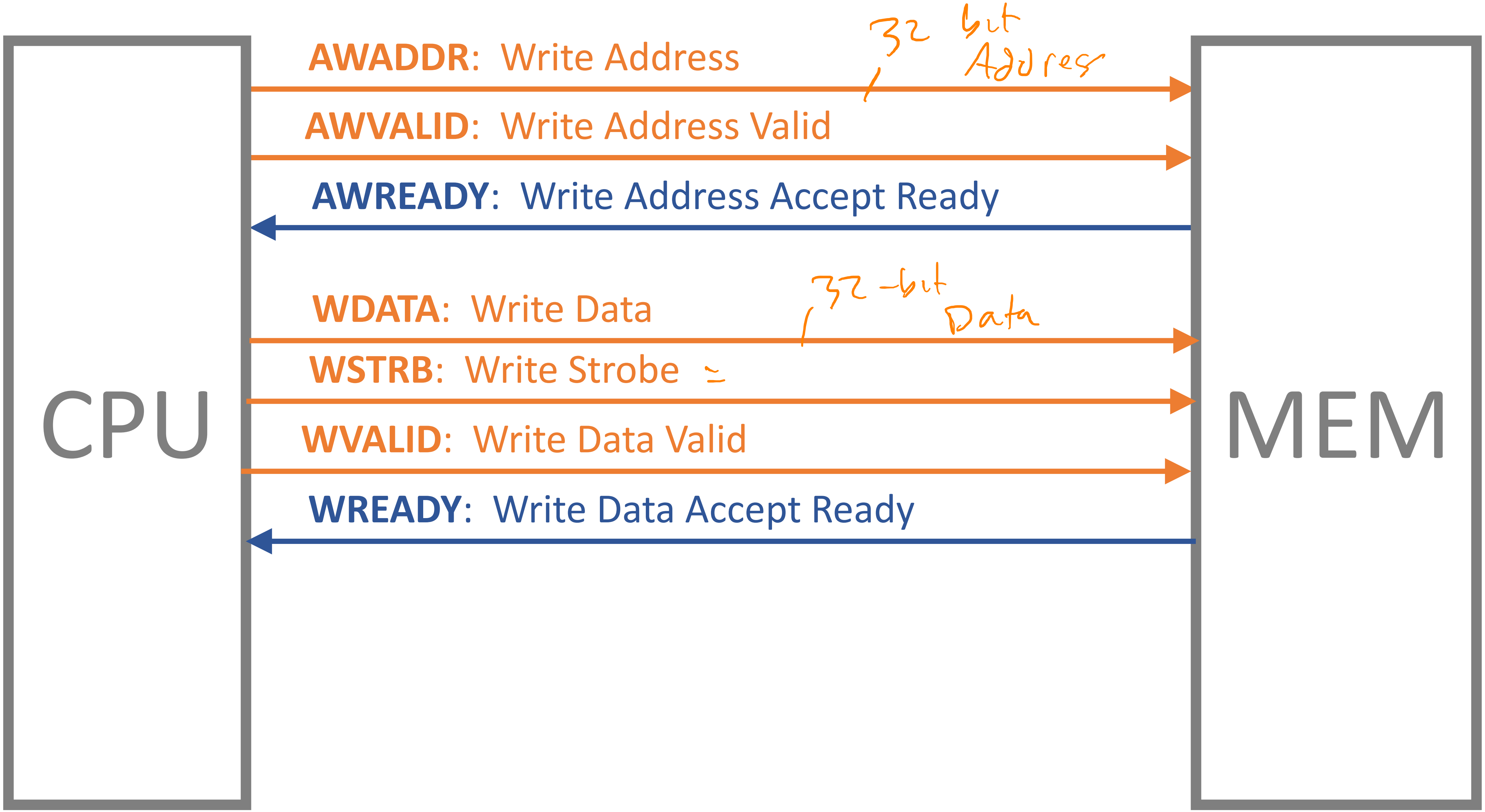


definition


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?

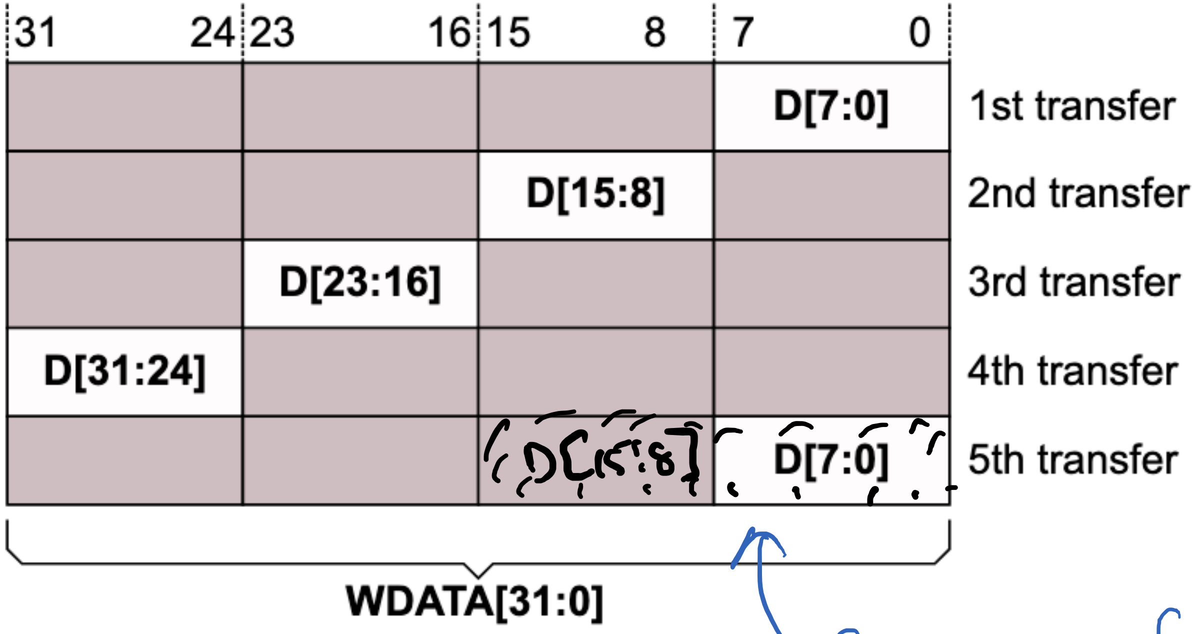
- A: WSTB: Write Strobe

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?



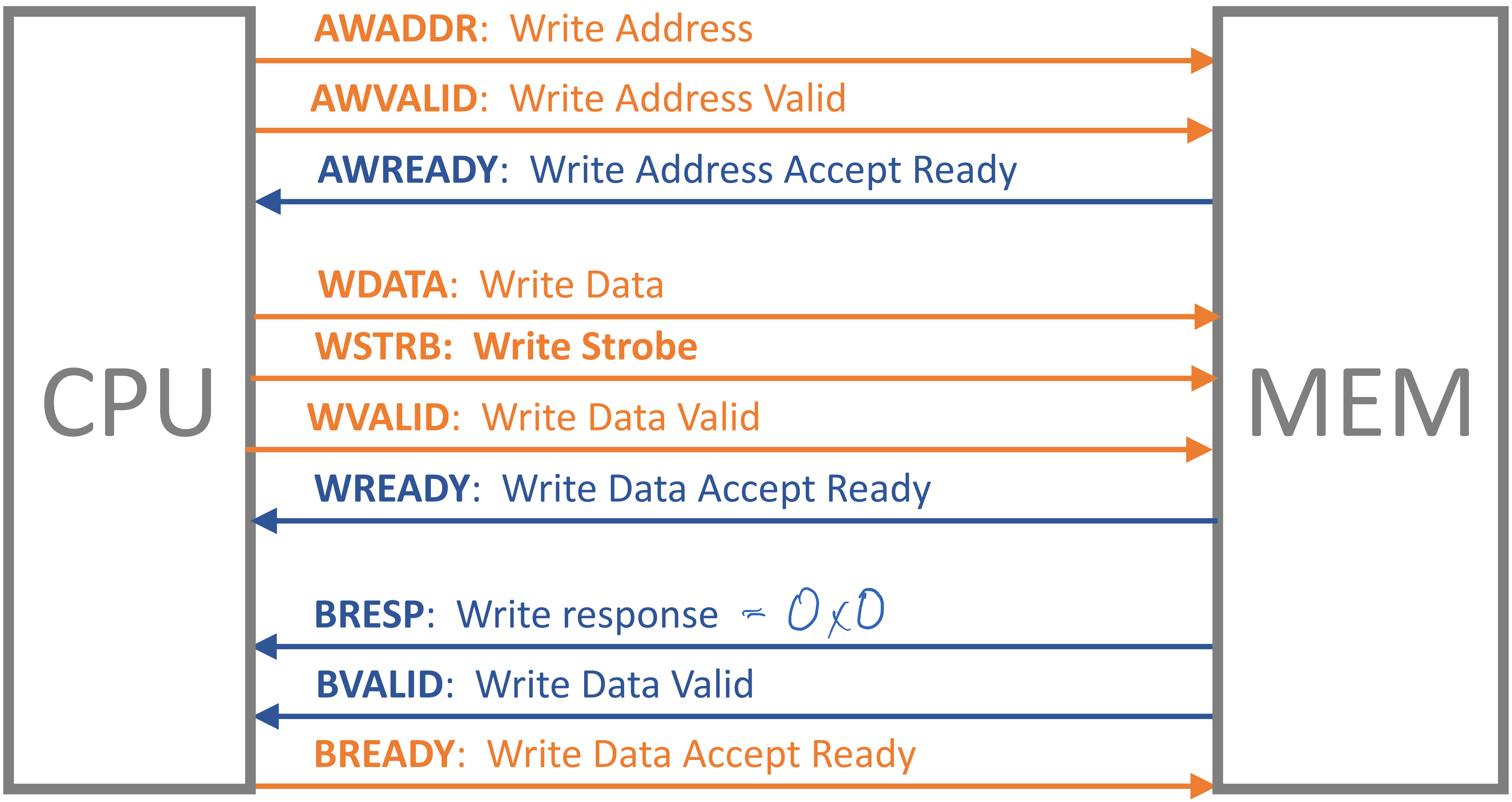
fix in future

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

WSTRB

0001 → 0x1
0010 → 0x2
0100 → 0x4

0011 → 0x3



ACLK and ARESETN not shown

BRESP is just like RRESP

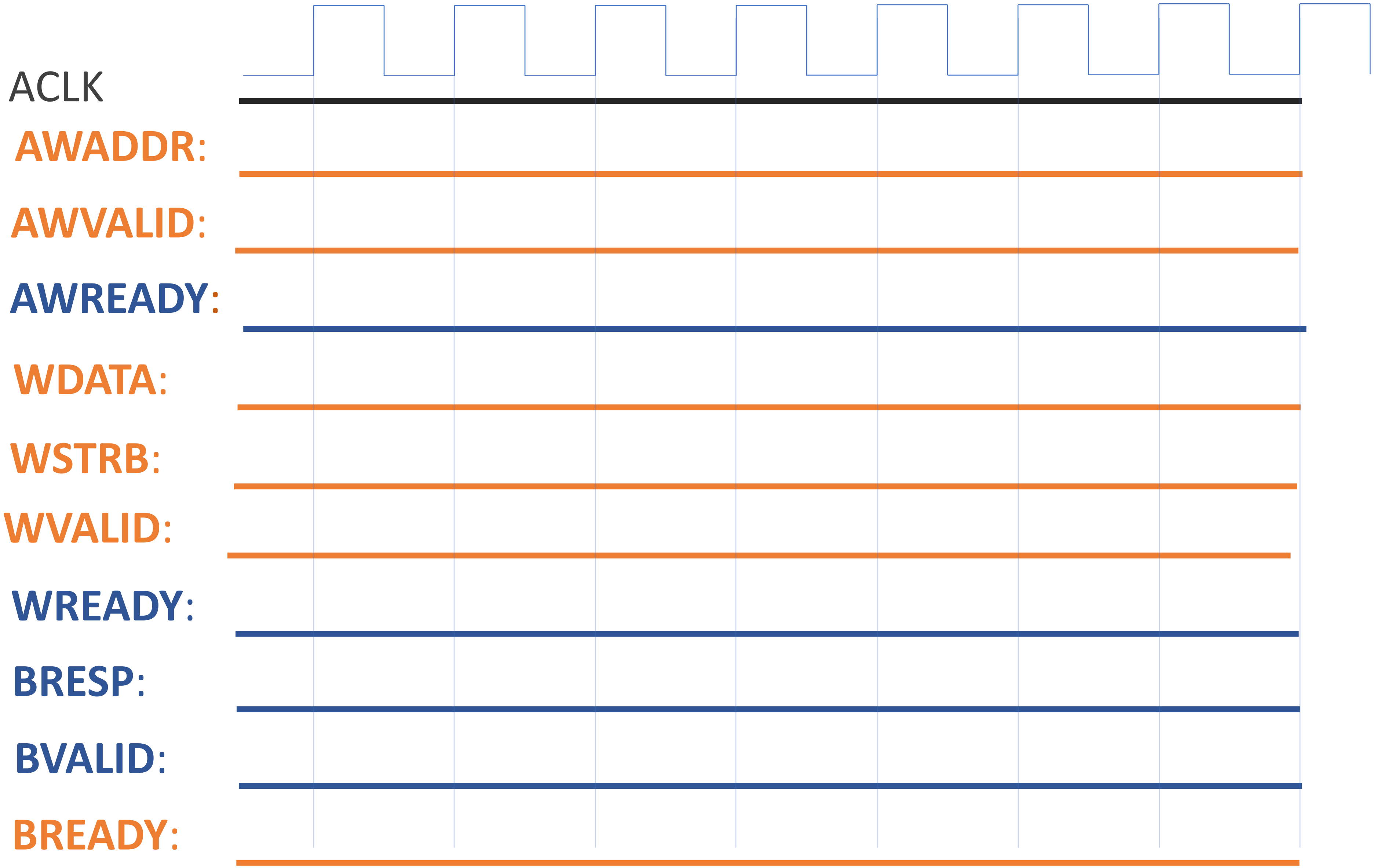
Table A3-4 RRESP and BRESP encoding

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

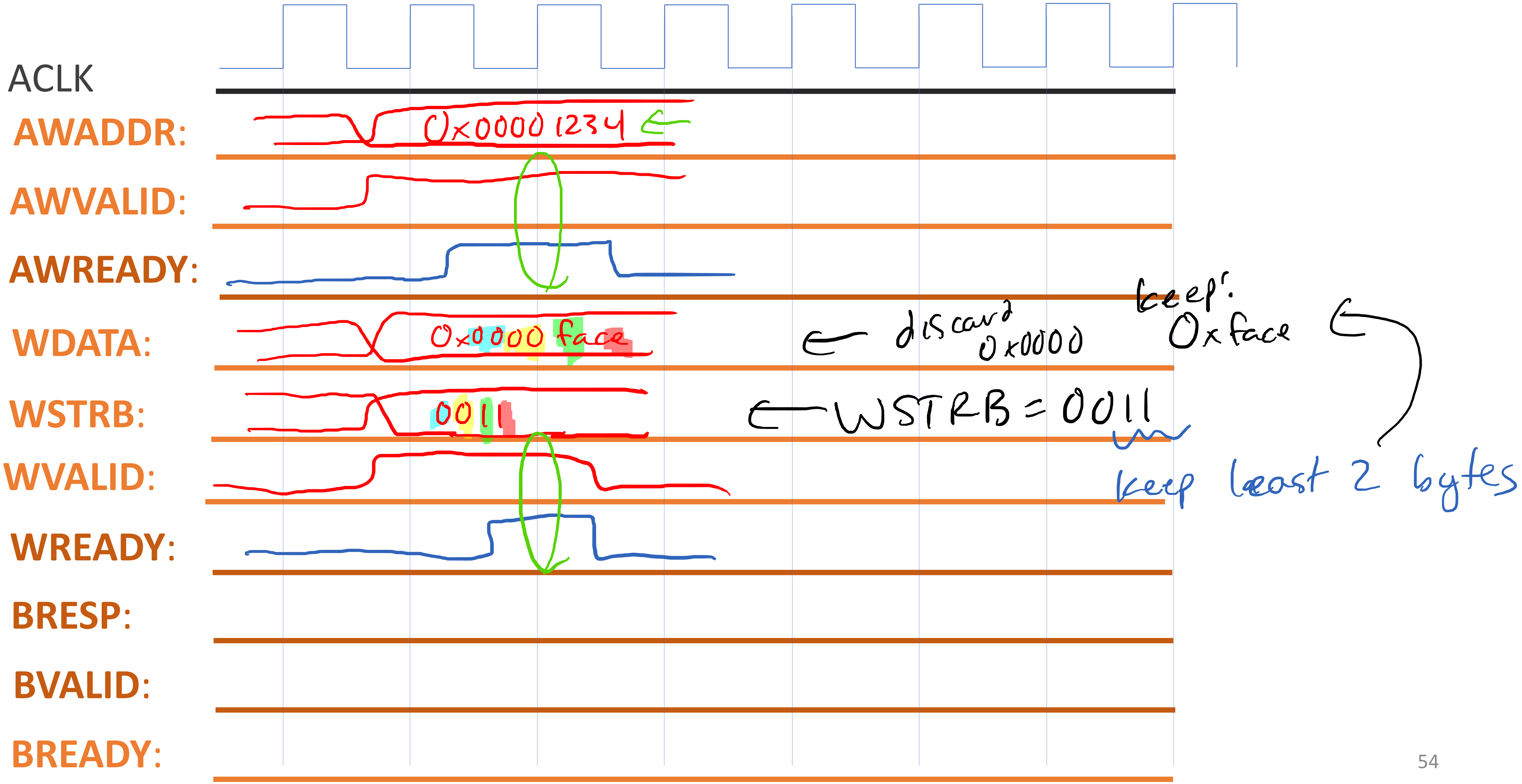
Writing 0xdeadbeef to 0x1234

Stop here!



16-bit

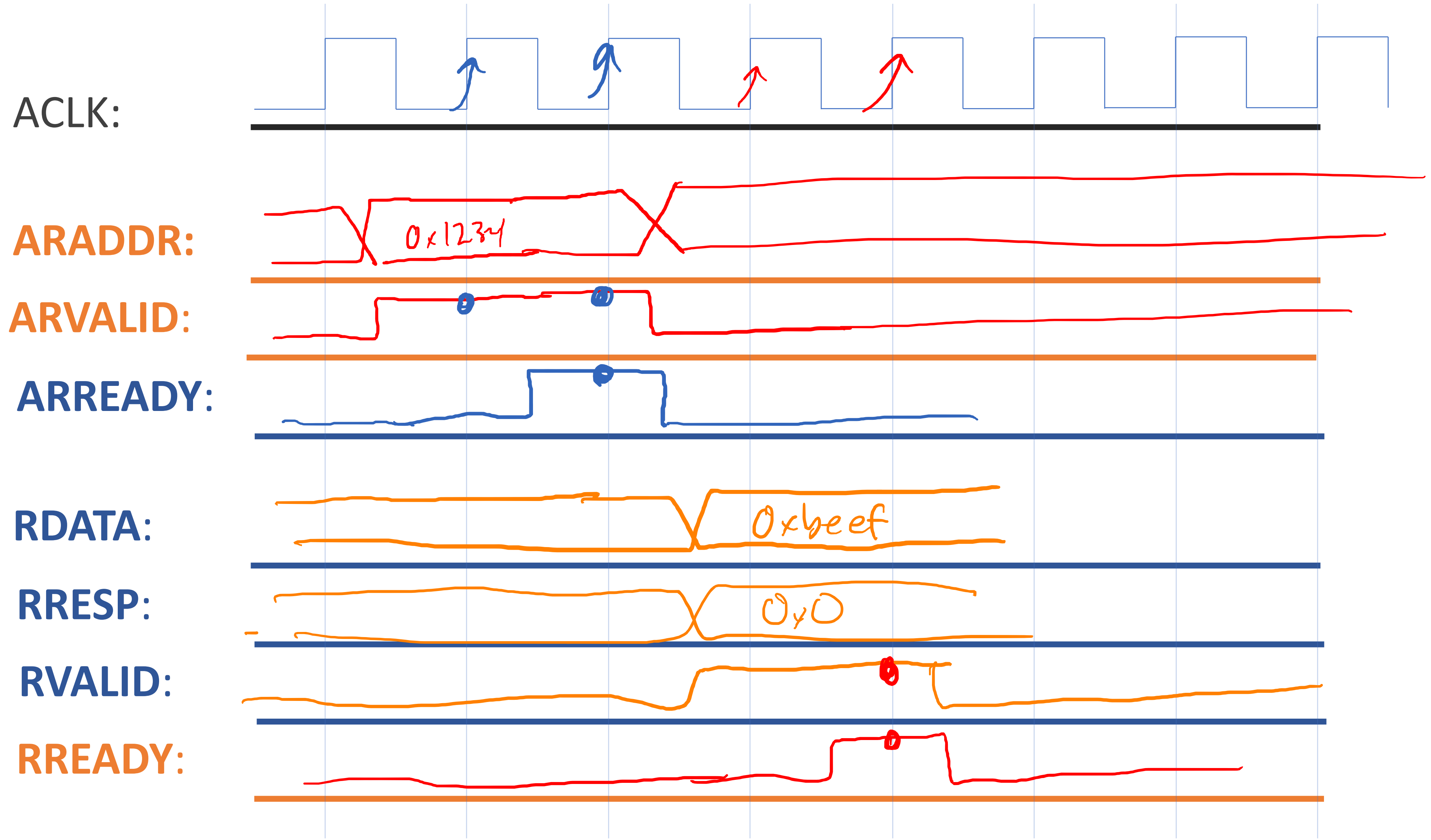
Writing 0xface to 0x1234



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

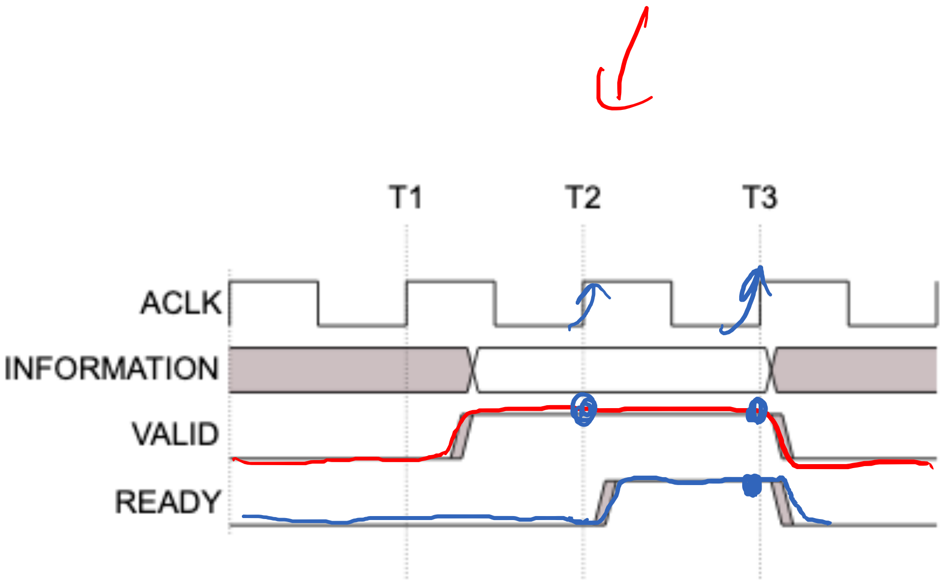


Figure A3-2 VALID before READY handshake

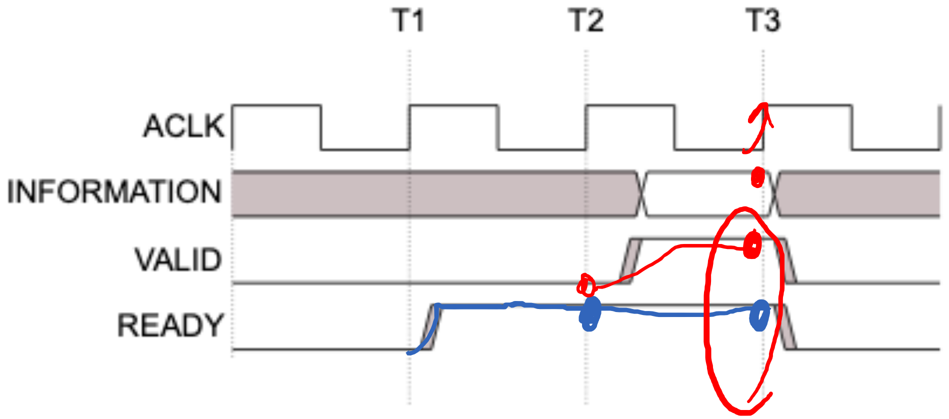
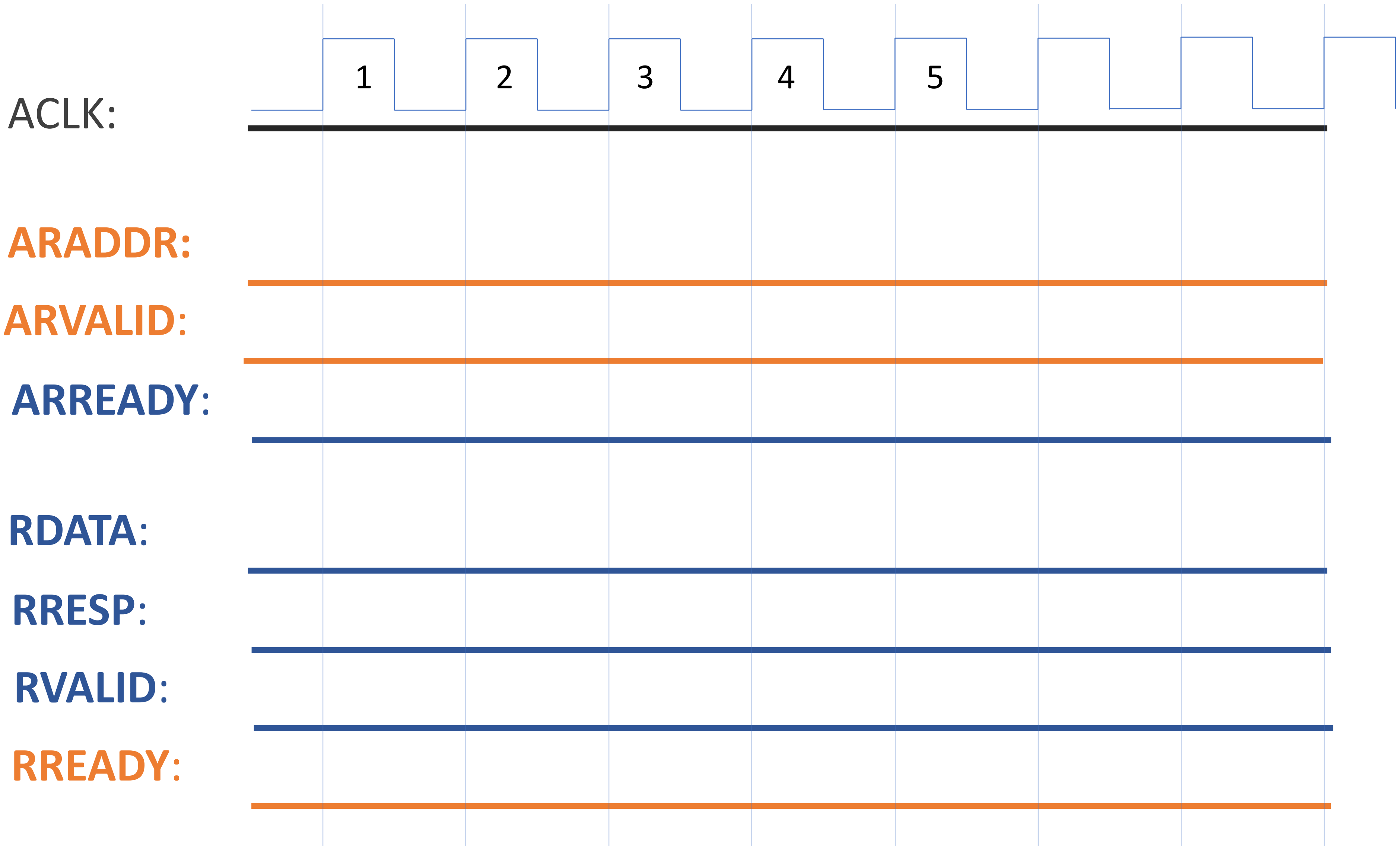


Figure A3-3 READY before VALID handshake

- 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

~~Monday~~!

Tuesday

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