# 10: High Performance Buses

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
*Indiana University*

Some material taken from:
EECS 373, University of Michigan
https://developer.arm.com/documentation/102202/0300/Transfer-behavior-and-transaction-ordering

## Announcements

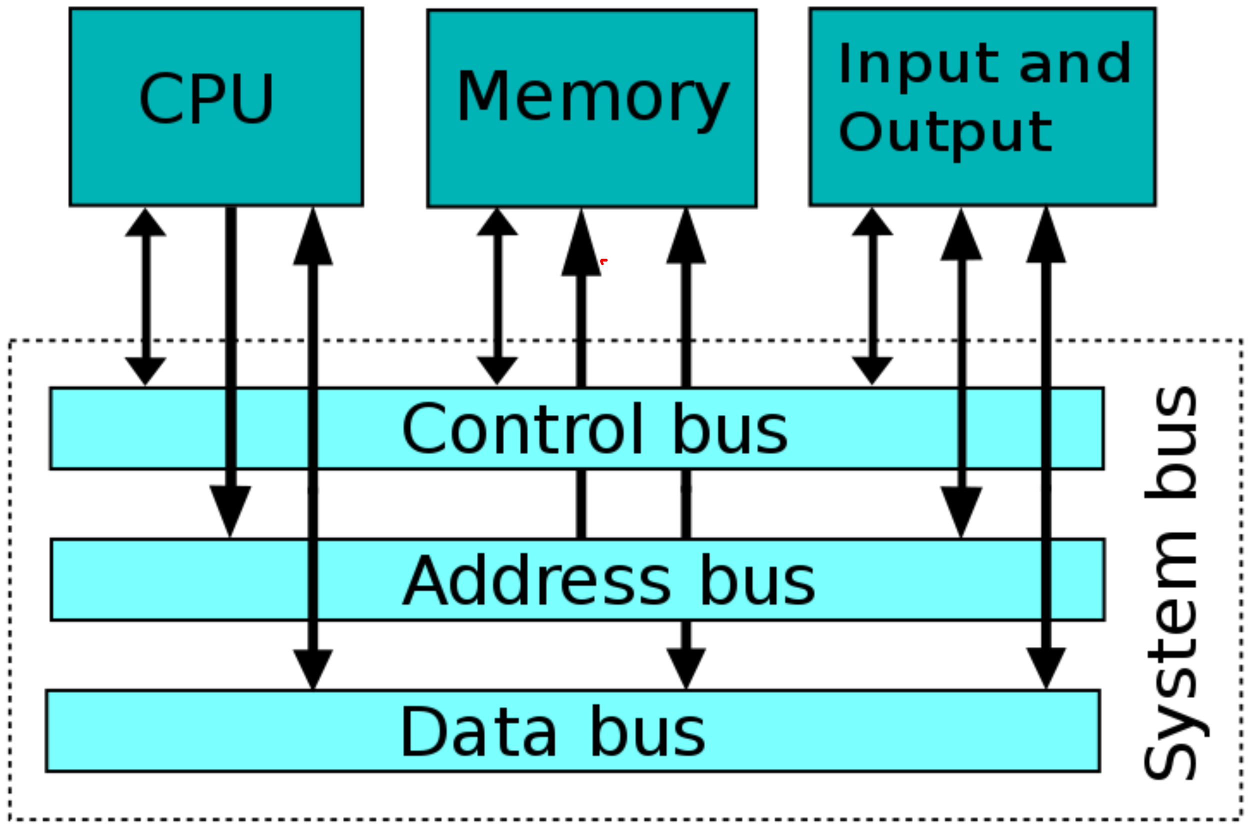
- P4: Due Next Wednesday. → AG tb different than project
- P5: Out soon.

↳ New AG tb

## Use volatile for MMIO addresses!

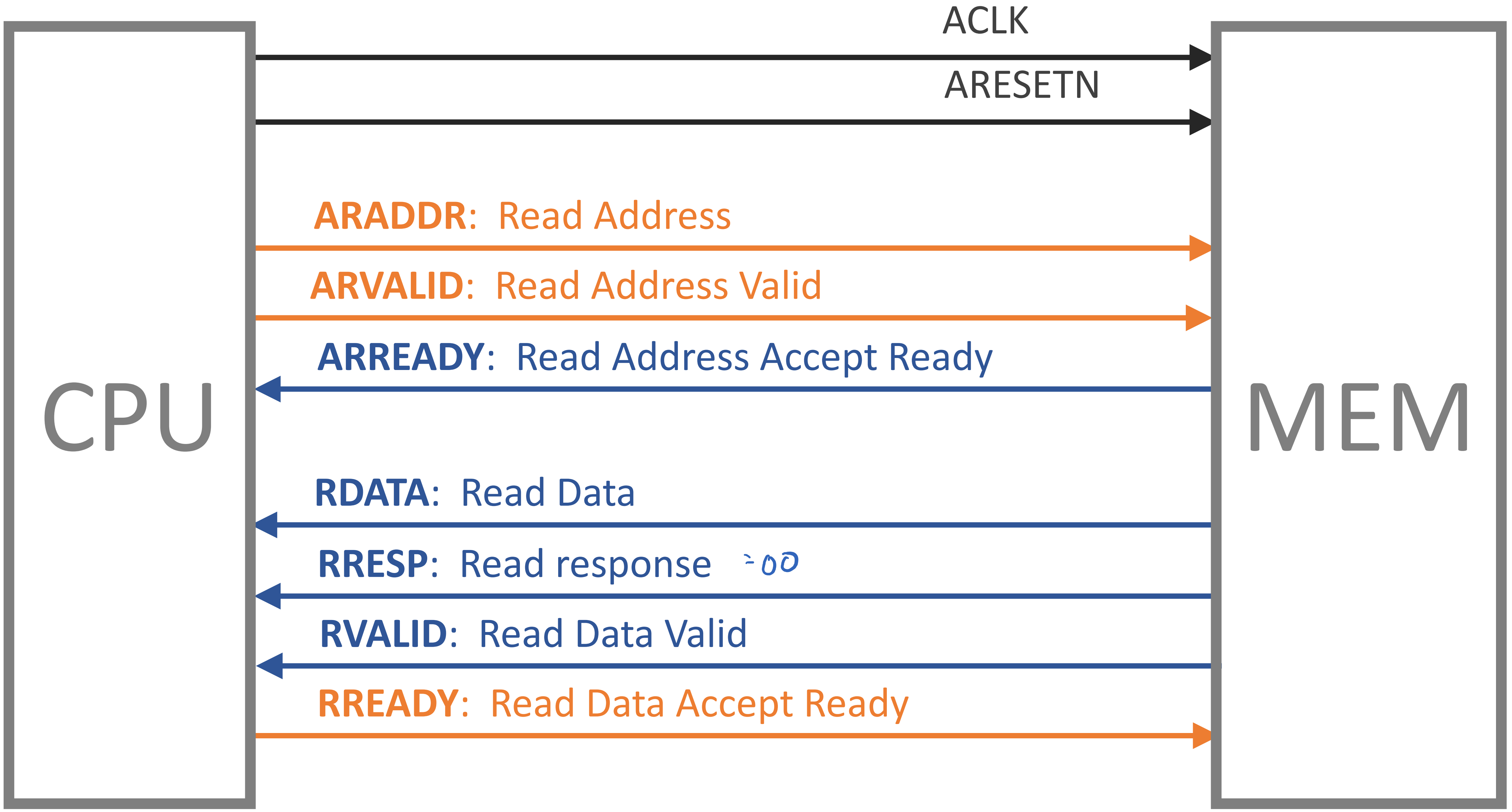
```
#define SW_ADDR 0xfffe
volatile uint32_t * SW_REG = (uint32_t * SW_ADDR);

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

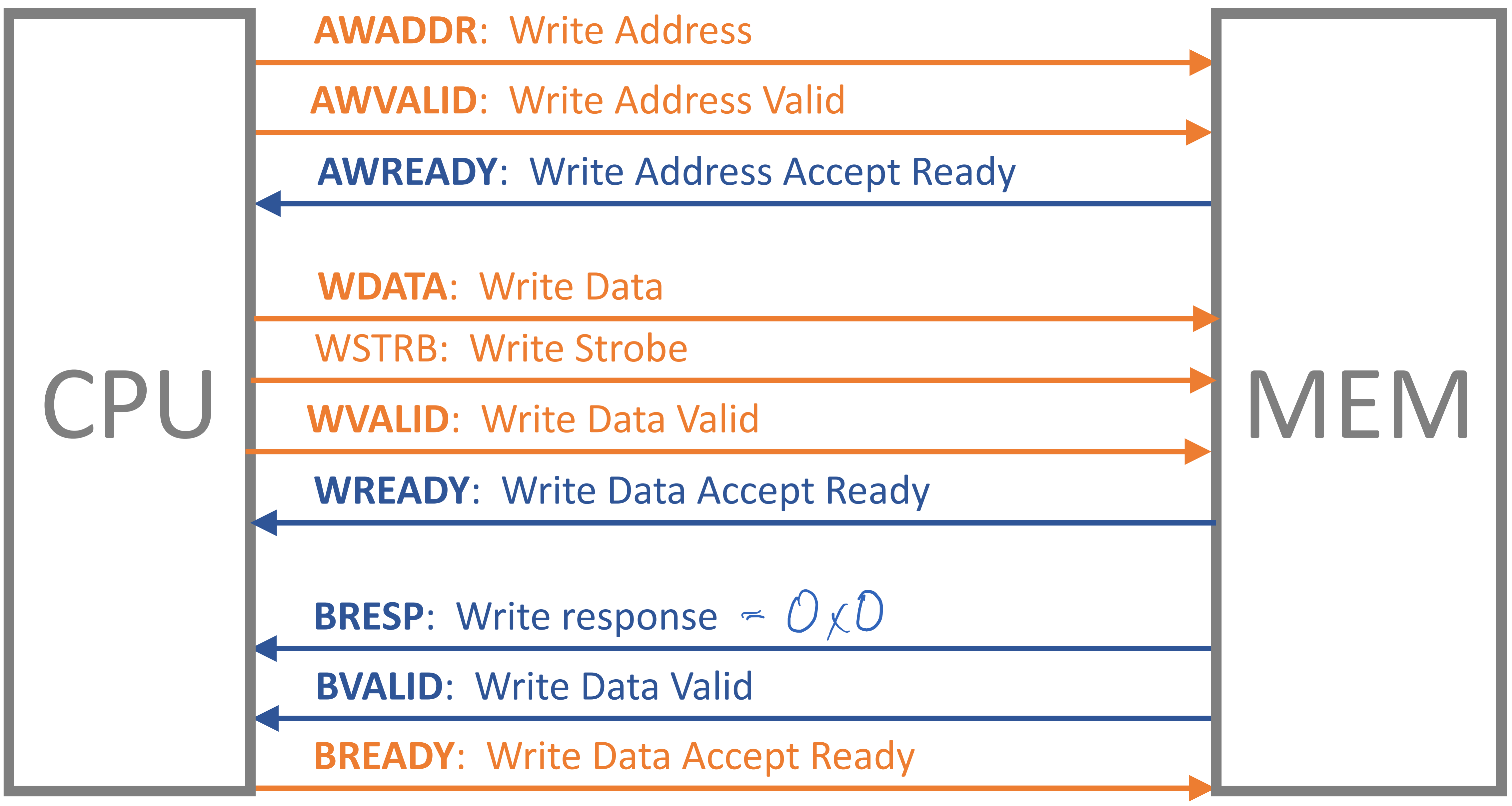
## The System Bus



## ARM AXI Bus

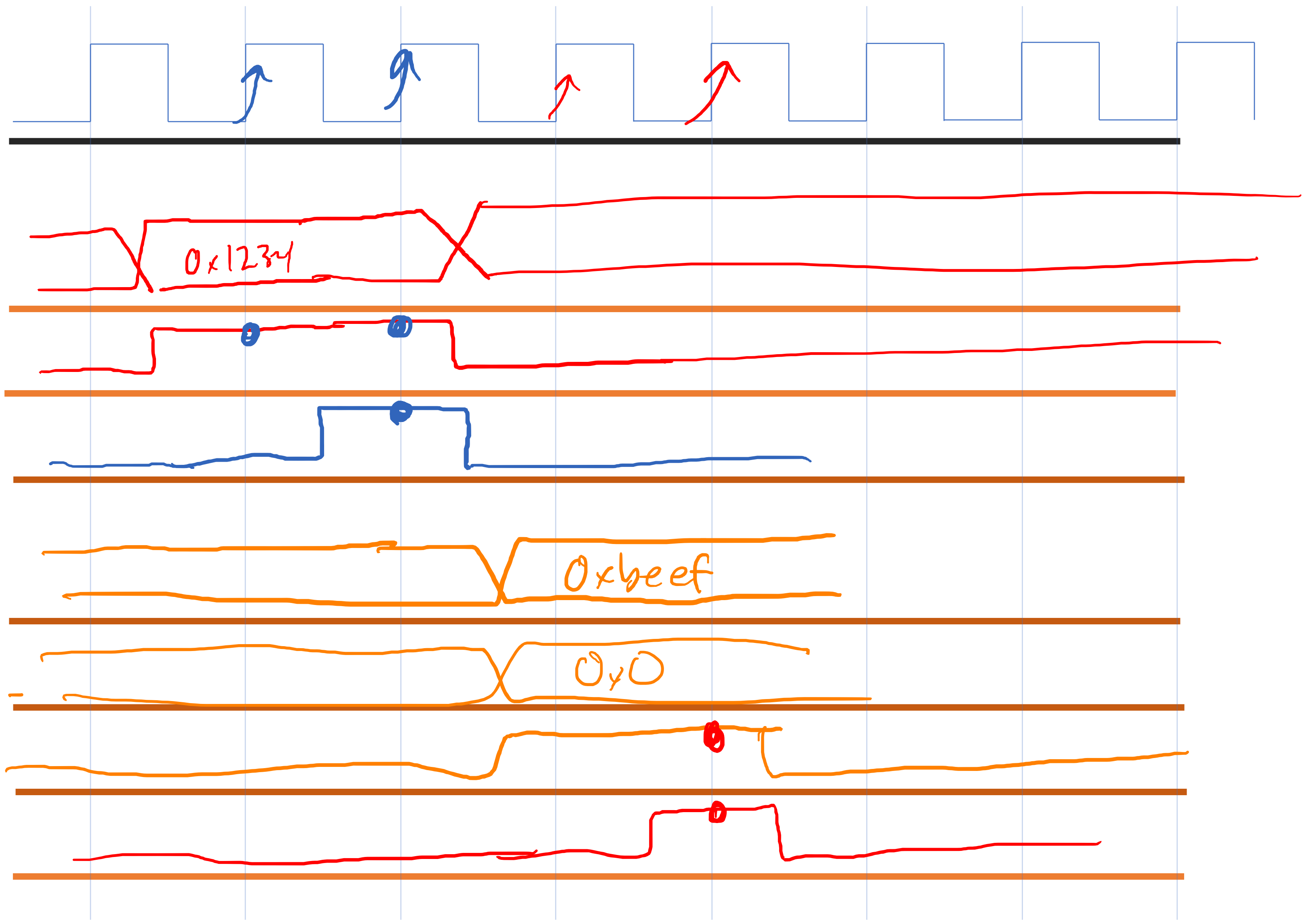
- “Advanced eXtensible Interface” Bus Version 4,  “AXI4”
- Three Variants
  - AXI4:  Fast but complicated; Memory-mapped
  - AXI4 Lite: Slow but simple; Memory-mapped
  - AXI4 Stream:  Fast and simple; Not memory-mapped


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


AWADDR: Write Address
AWVALID: Write Address Valid
AWREADY: Write Address Accept Ready
WDATA: Write Data
WSTRB: Write Strobe
WVALID: Write Data Valid
WREADY: Write Data Accept Ready
BRESP: Write response = 0x0
BVALID: Write Data Valid
BREADY: Write Data Accept Ready
CPU
MEM

ACLK and ARESETN not shown

# How long does a read(load) take?



when is store (write)

## High-Performance Bus Ideas

- Make single transaction faster

# AXI Handshake Speedup

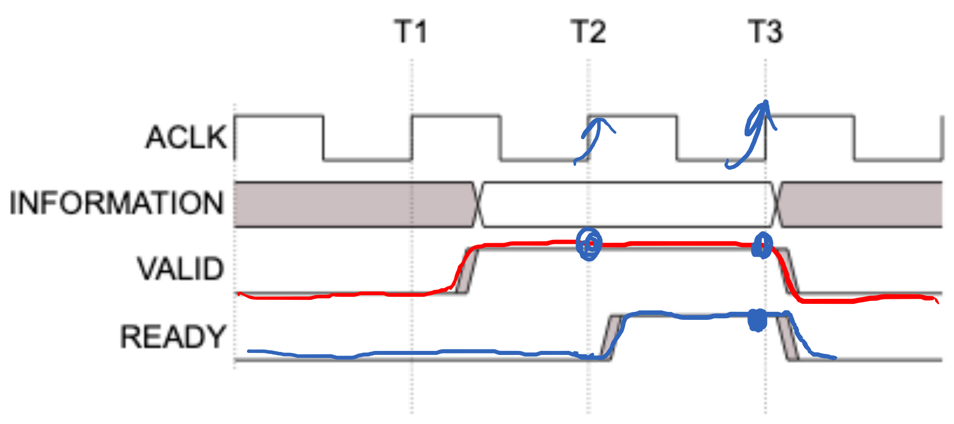


Figure A3-2 VALID before READY handshake

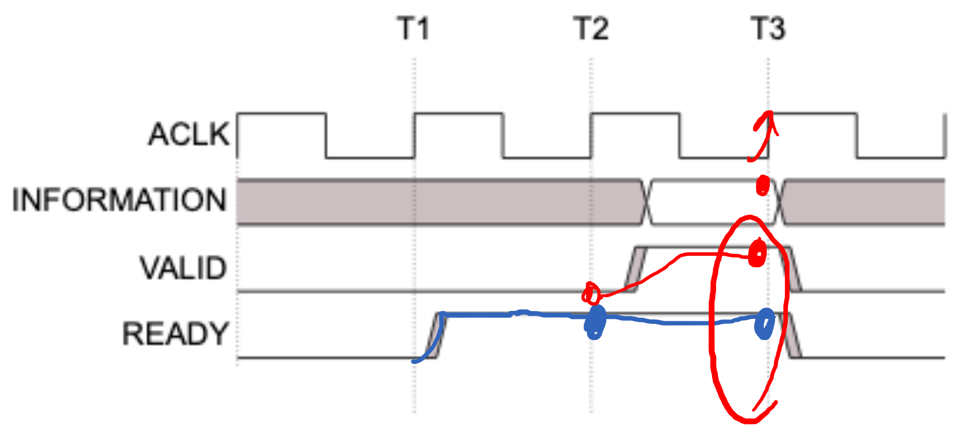


Figure A3-3 READY before VALID handshake

- Both are valid
- Figure A3-3 is faster

# What happens here?

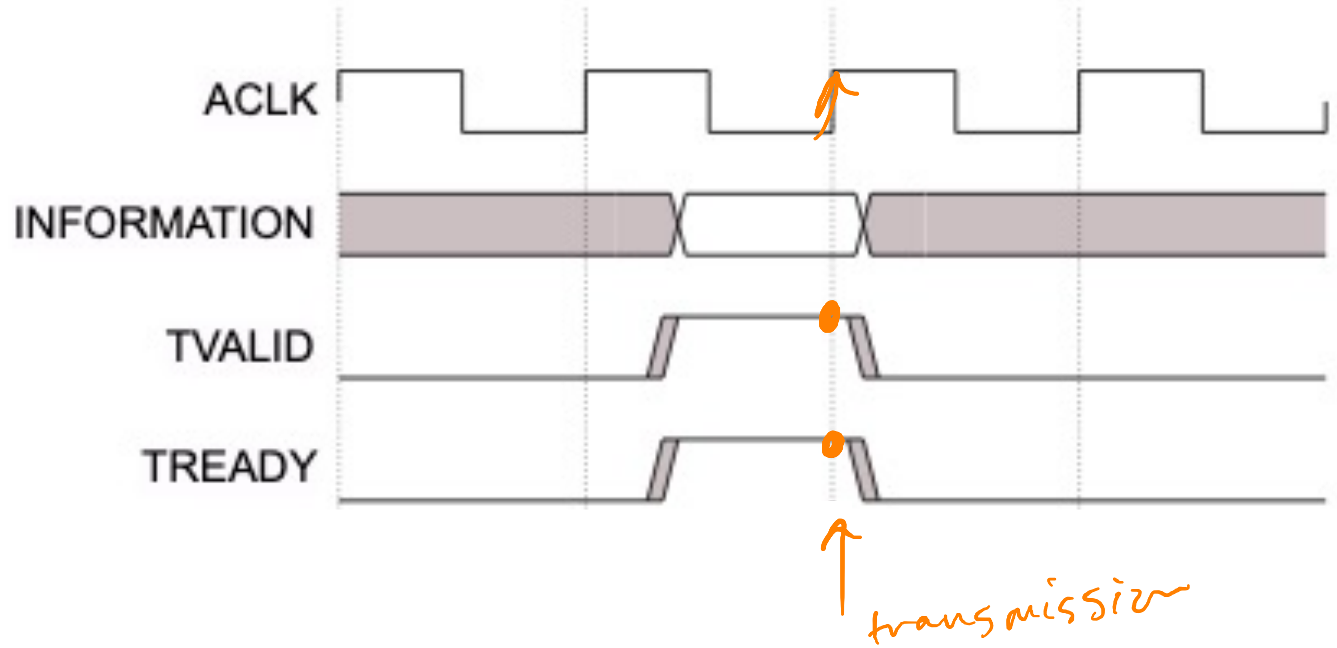
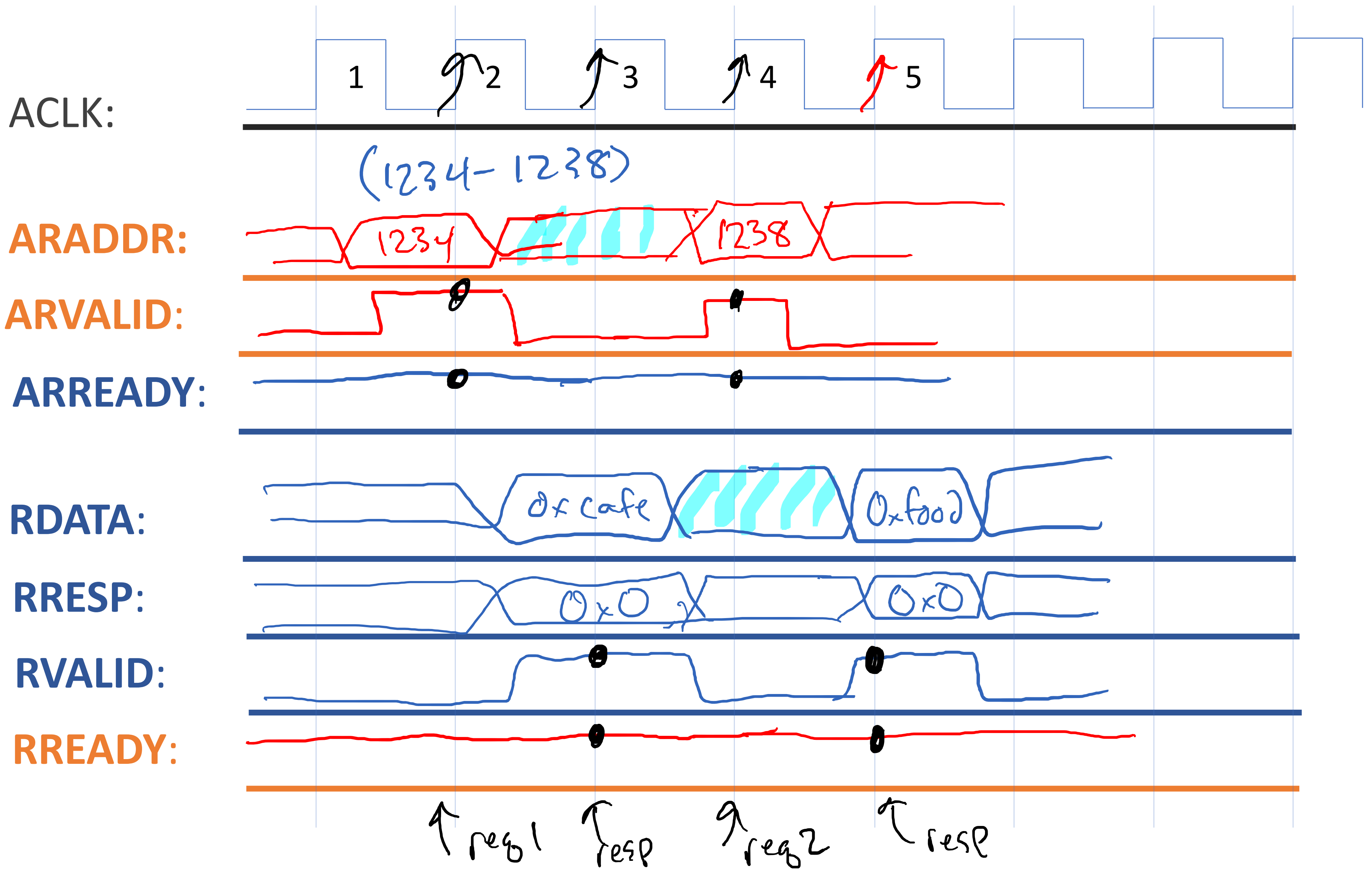


Figure 2-3 TVALID with TREADY handshake

## What can we do to make this faster?

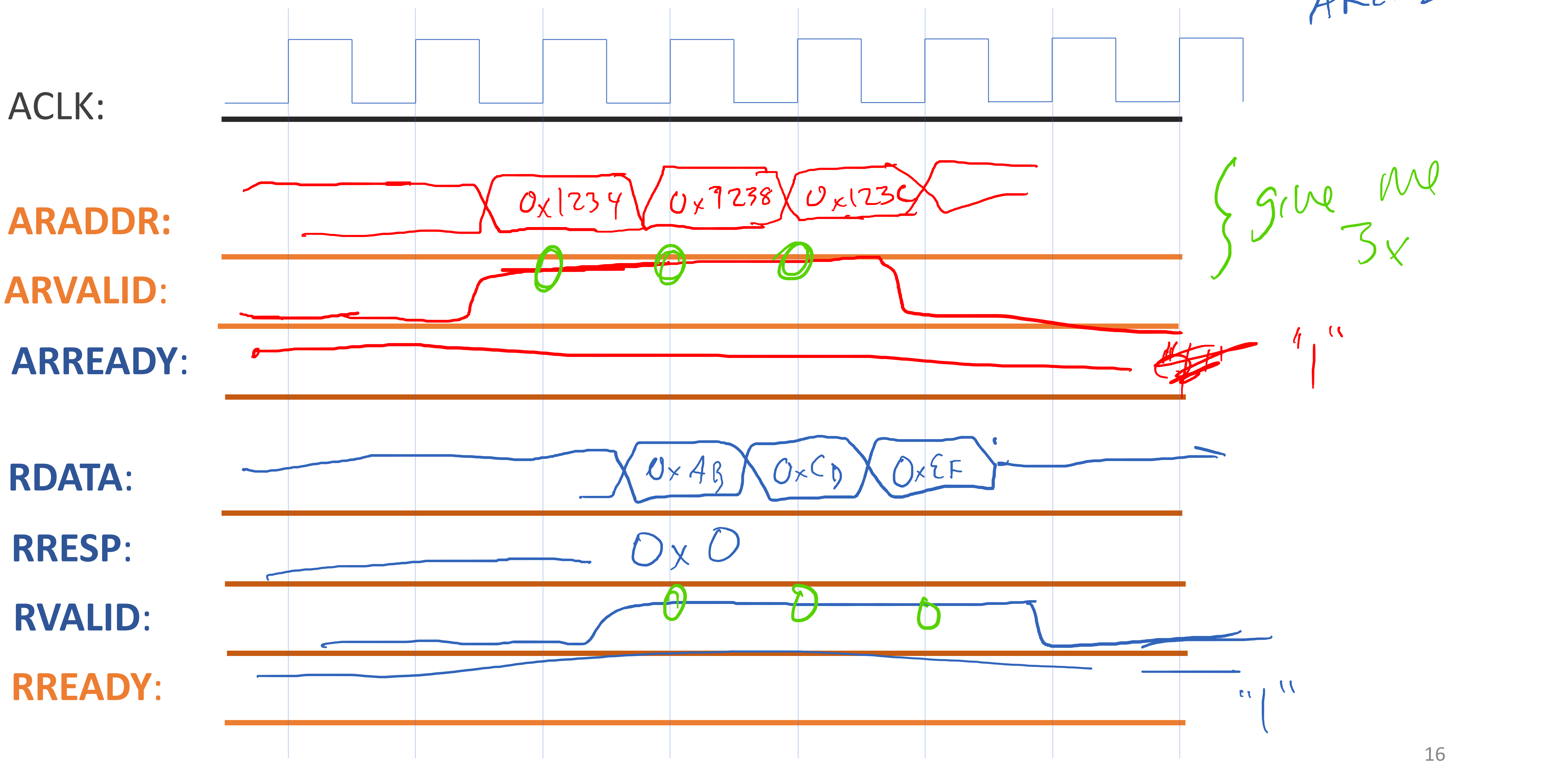


## High-Performance Bus Ideas

- Make single transaction faster
- Overlap multiple transactions

# Can we load 0x1234 and 0x1238?

assume ARESETN=1



## Burst Transactions

- When a device is **transmitting data repeatedly without** going through **all the steps** required to transmit each piece of data in a separate transaction

## Burst Transaction

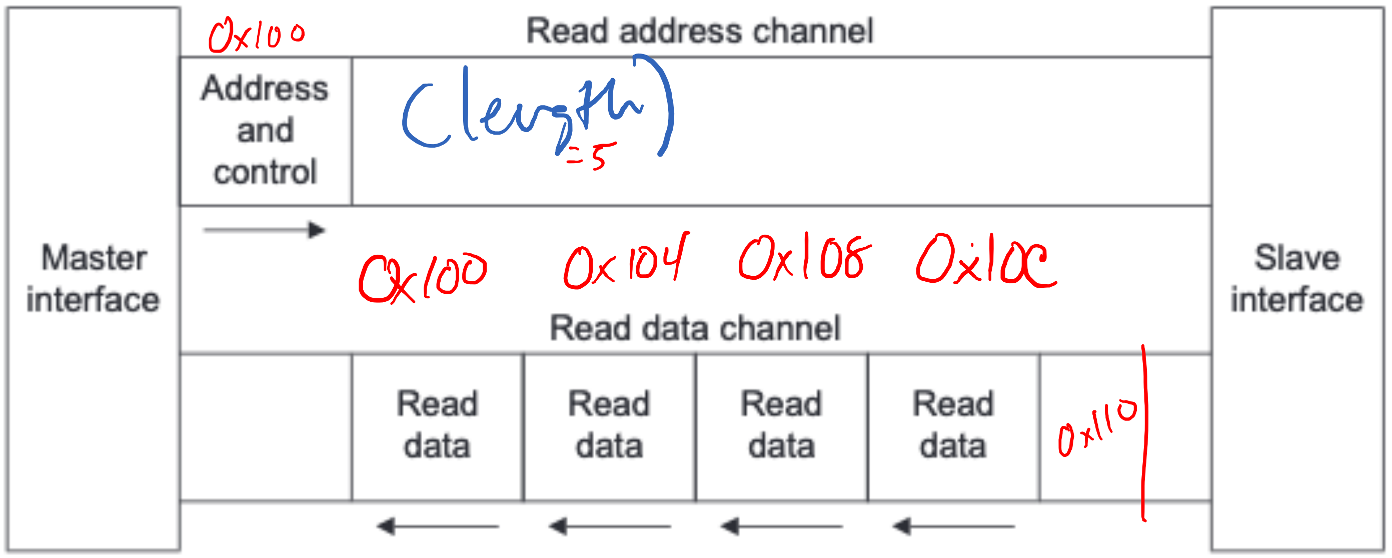
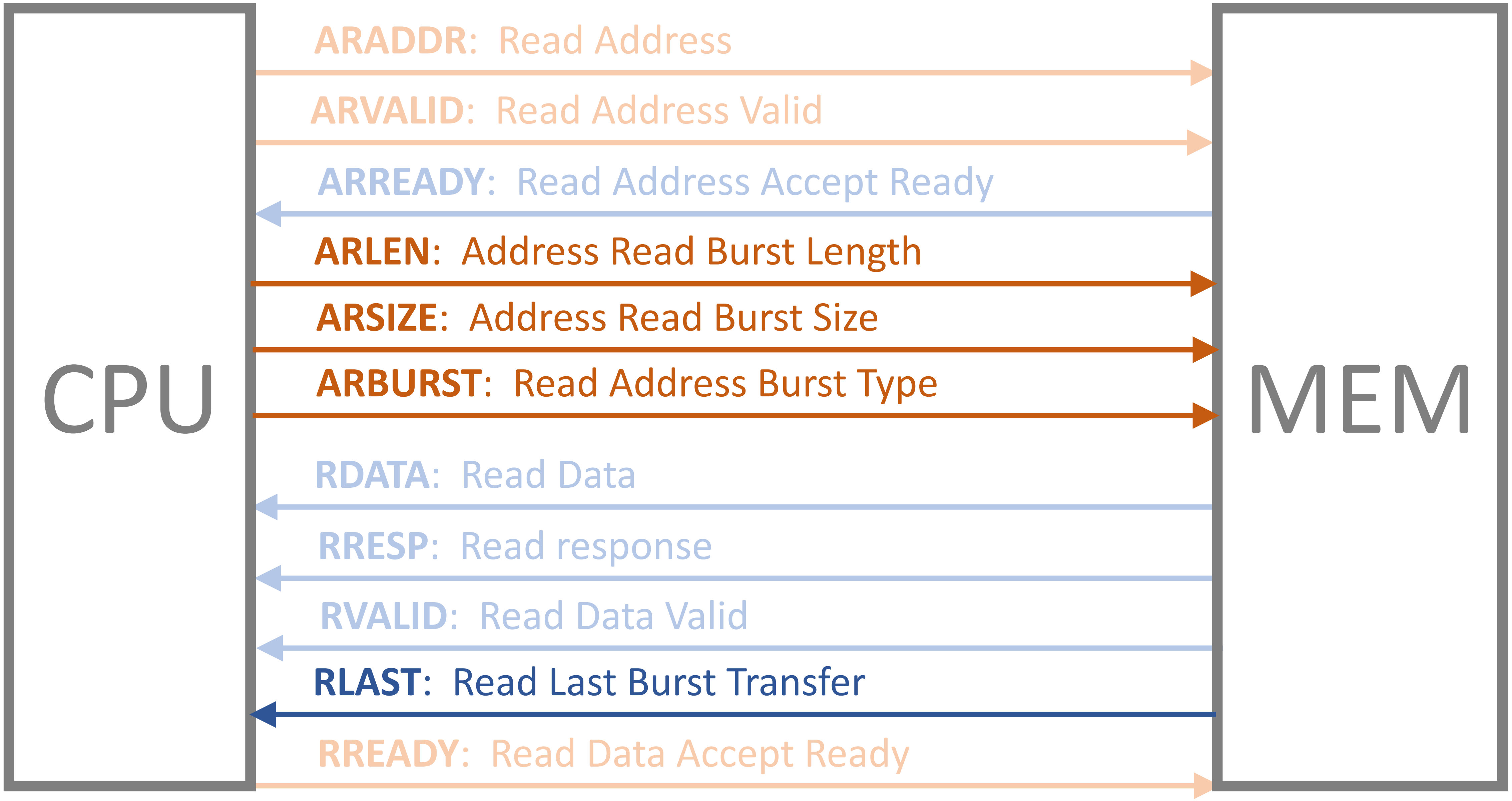


**Figure 1-1 Channel architecture of reads**


CPU
MEM
ARADDR: Read Address
ARVALID: Read Address Valid
ARREADY: Read Address Accept Ready
ARLEN: Address Read Burst Length
ARSIZE: Address Read Burst Size
ARBURST: Read Address Burst Type
RDATA: Read Data
RRESP: Read response
RVALID: Read Data Valid
RLAST: Read Last Burst Transfer
RREADY: Read Data Accept Ready

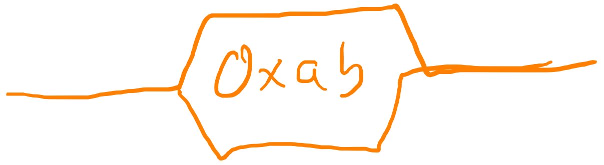
## What do the new signals do?

**ARLEN**: Address Read Burst Length

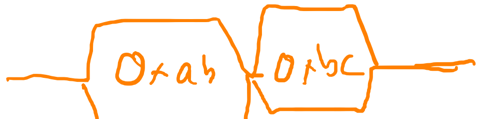
How many bursts should occur? (+1)

Burst_Length = ARLEN[7:0] + 1

ARLEN = 0 => Burst Length = 1



ARLEN = 1 => Burst = 2



## What do the new signals do?

**ARLEN**: Address Read Burst Length

How many bursts should occur? (+1)

**ARSIZE**: Address Read Burst Size

How many bytes should be in each burst?

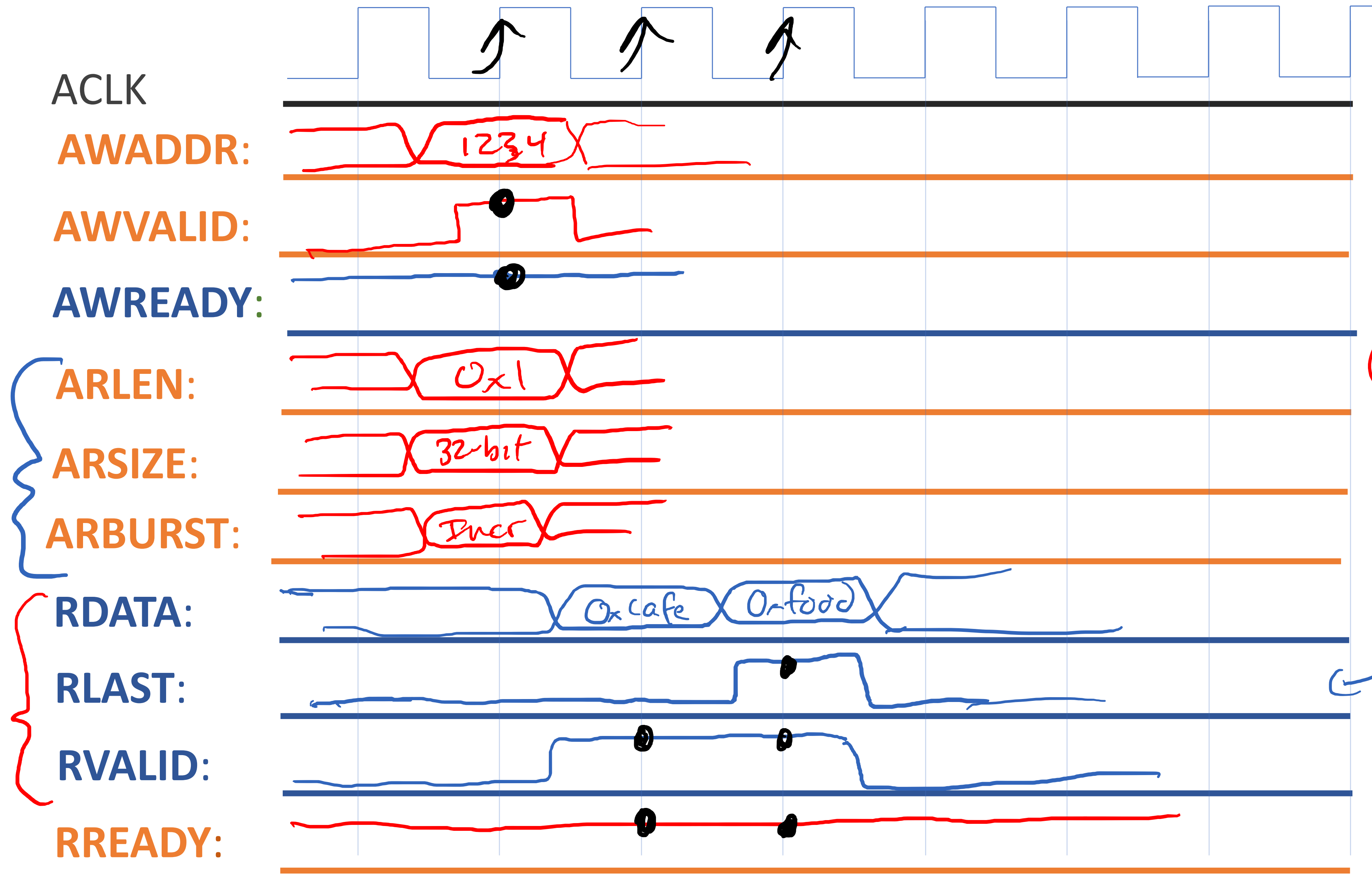
**ARBURST**: Read Address Burst Type

Are the addresses incrementing, or repeating?

**RLAST**: Read Last Burst Transfer

Are we done yet?

# Reading 0x1234 and 0x1238



| Addr | Value |
|---|---|
| 0x1234 → | 0xcafe |
| 0x1238 → | 0xfood |

(+1)

~~old~~ 2·128 = 256 cycles

New: 1 + 128 = 129 cycles

indicates done

ARESETN and RRESP omitted

## Read Burst Example

Burst Writes also exist

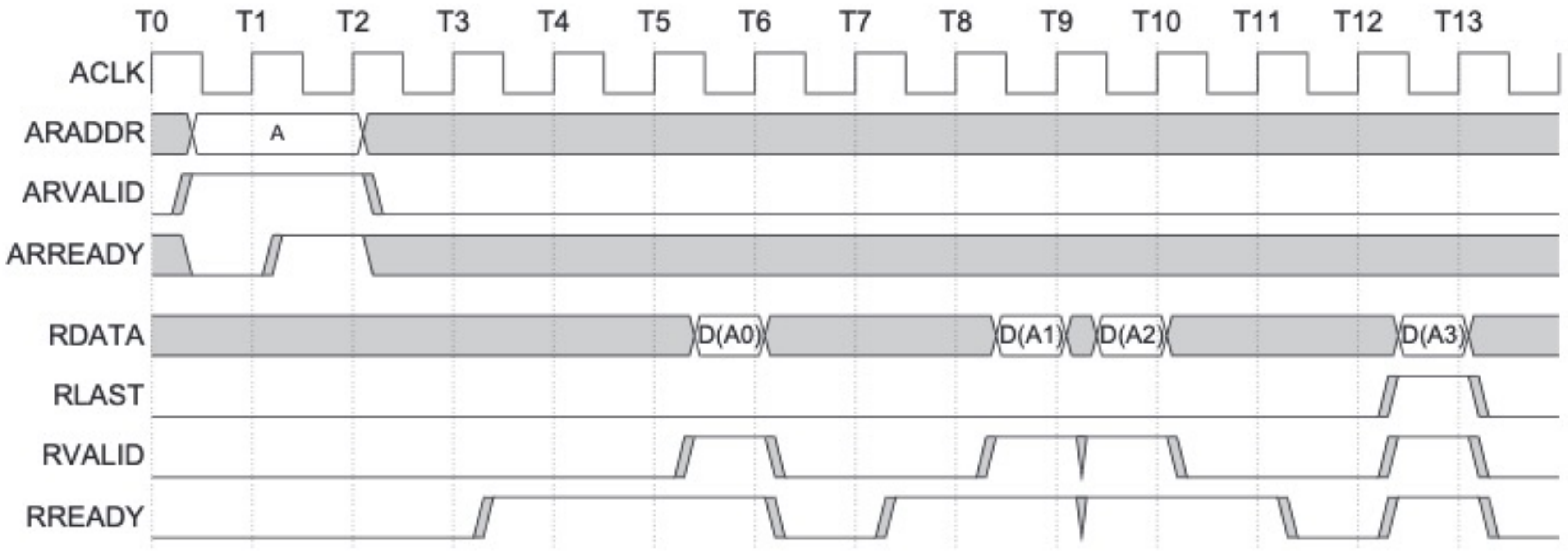


**Figure 1-4 Read burst**

**Note**

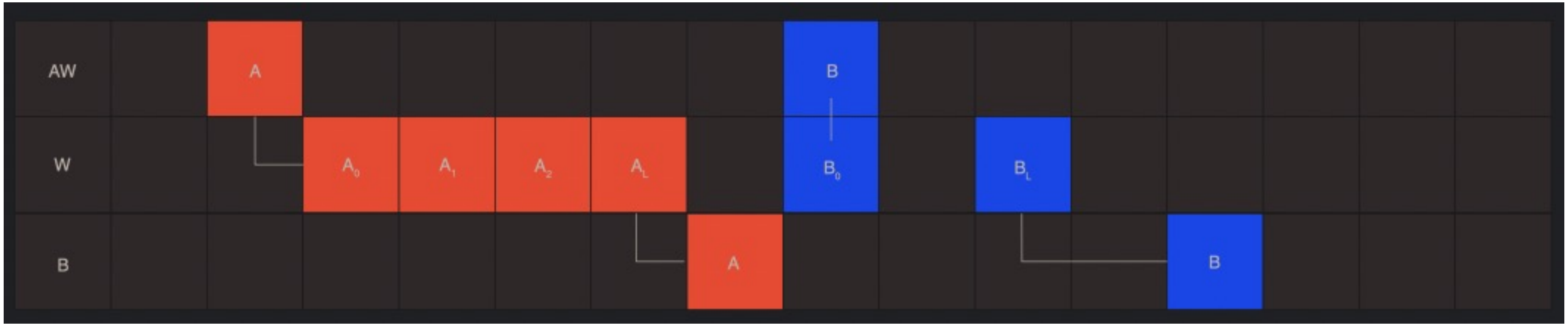
The master also drives a set of control signals showing the length and type of the burst, but these signals are omitted from the figure for clarity.

# Fully Overlapped + Interleaved Transactions

- AXI4 allows overlapped transactions
- 2nd request allowed before 1st request complete
- AXI4 also allows for “Transfer IDs”
- Transfer IDs allow transactions to progress out of request order.

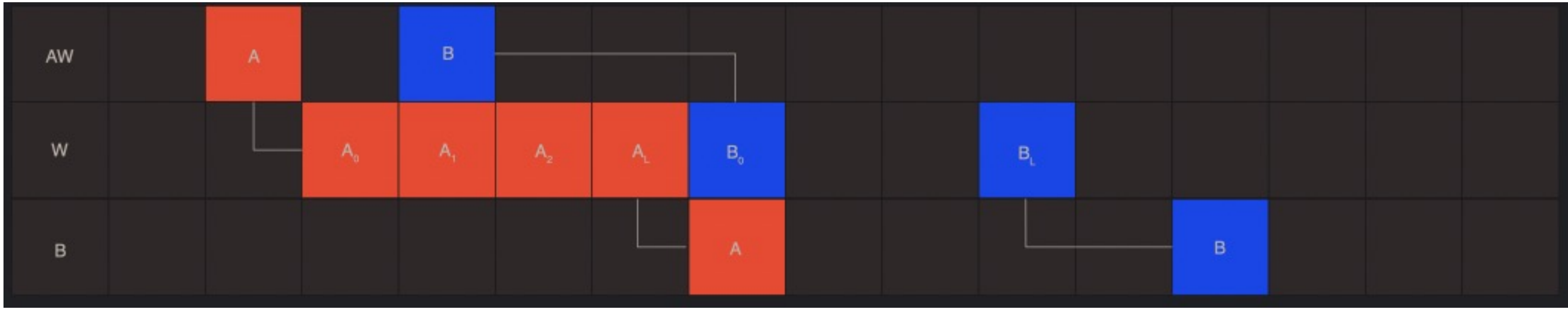
## Two Burst Writes – No Overlap

- Request A finishes before B is started



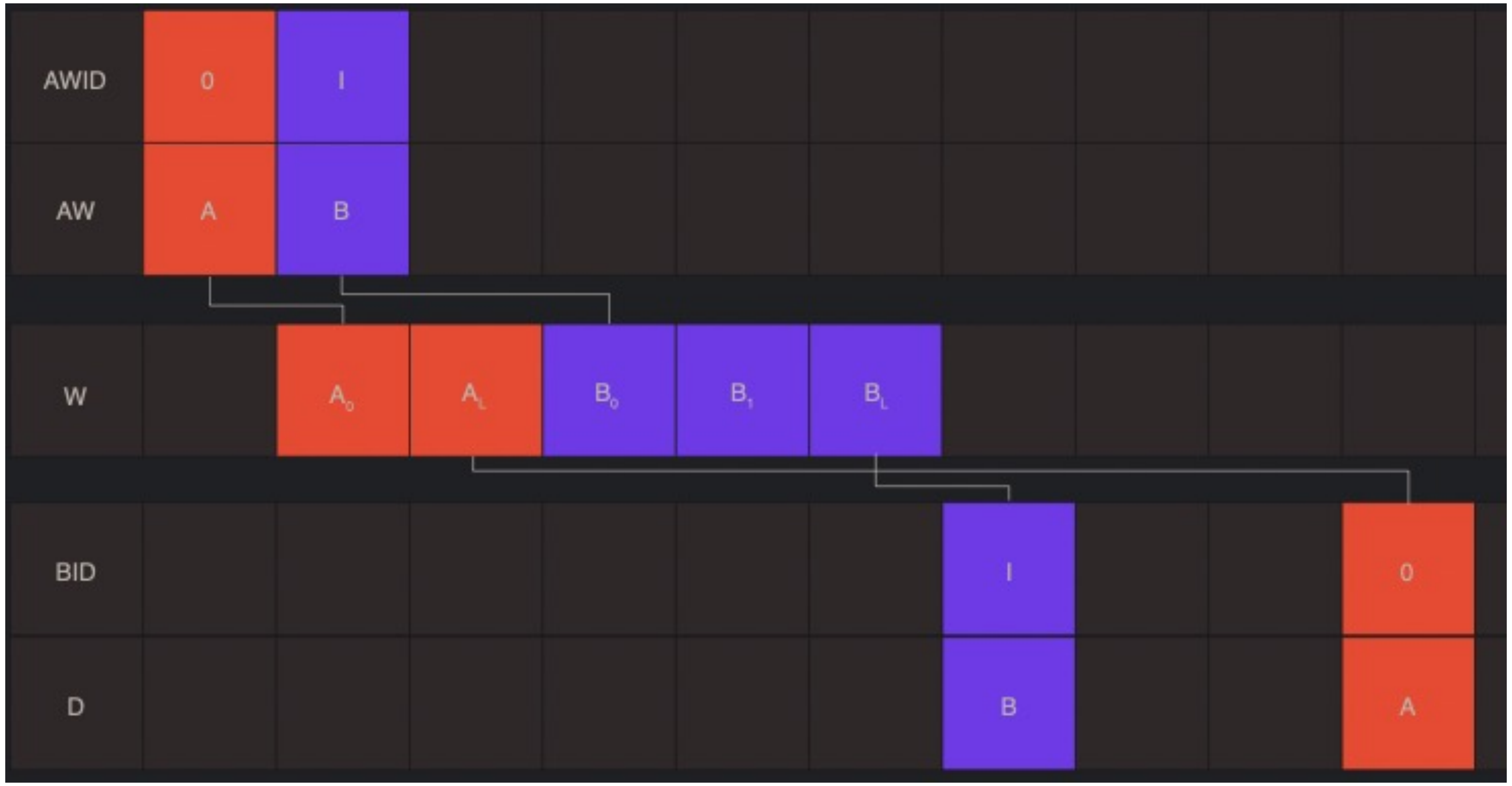
## Two Burst Writes – Overlap Allowed

- B starts before A is complete.  Still maintains ordering between A and B.



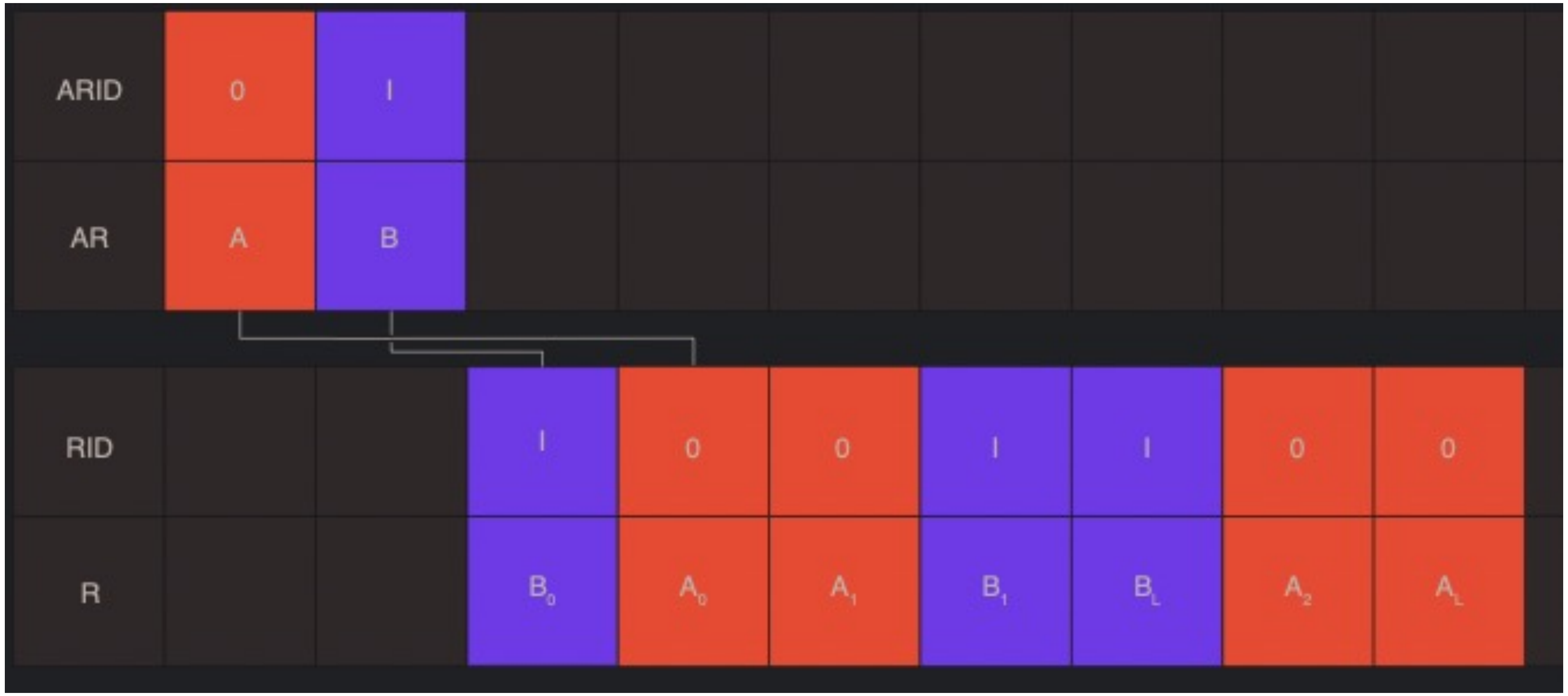
# Two Burst Writes – Out of Order Completion

- B starts before A is complete.  B completes before A.



# Two Burst Reads – Out of Order Completion

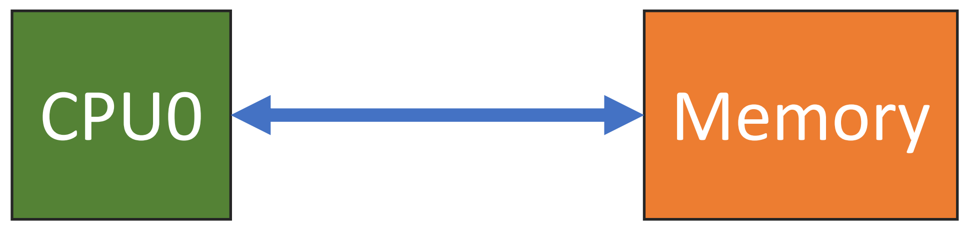
- B starts before A is complete.  A and B are interleaved



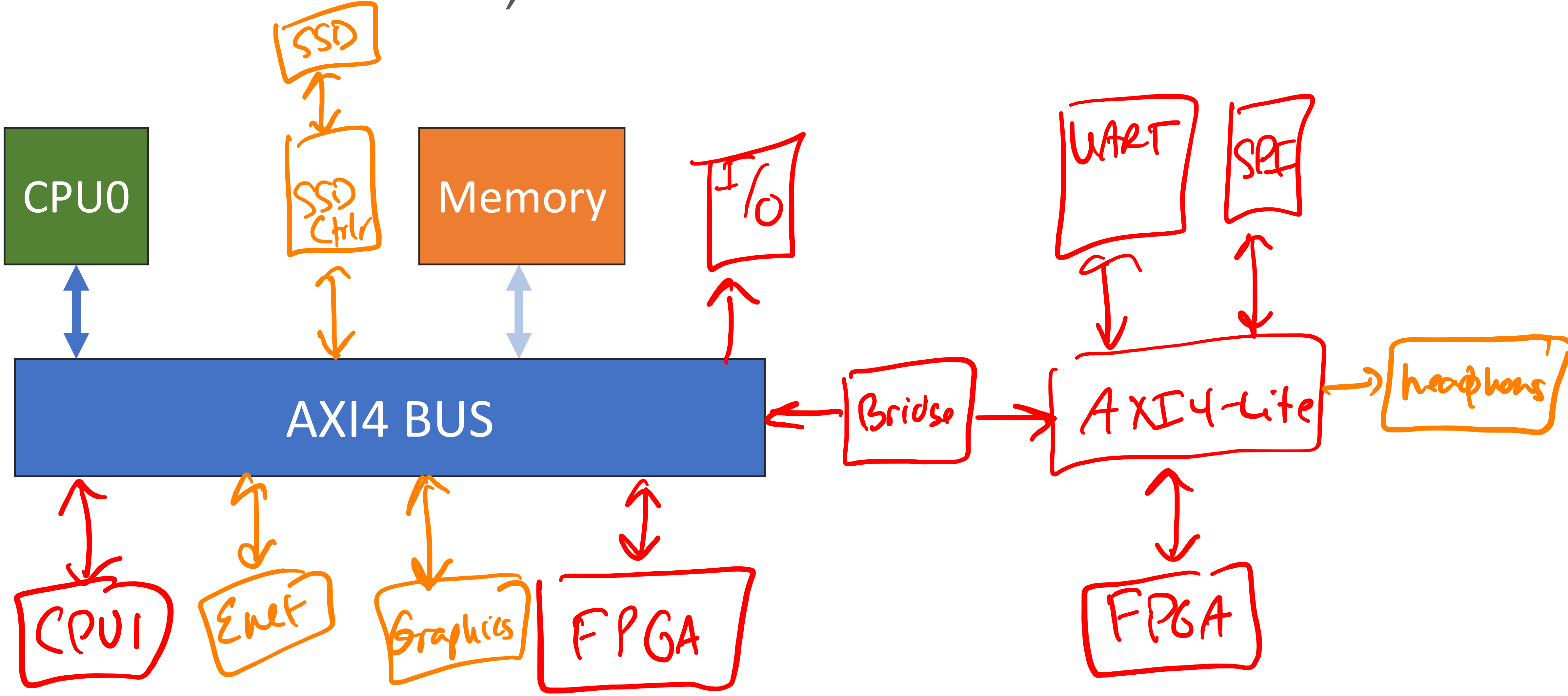
## Topic Shift:

- Virtual Memory + Linux

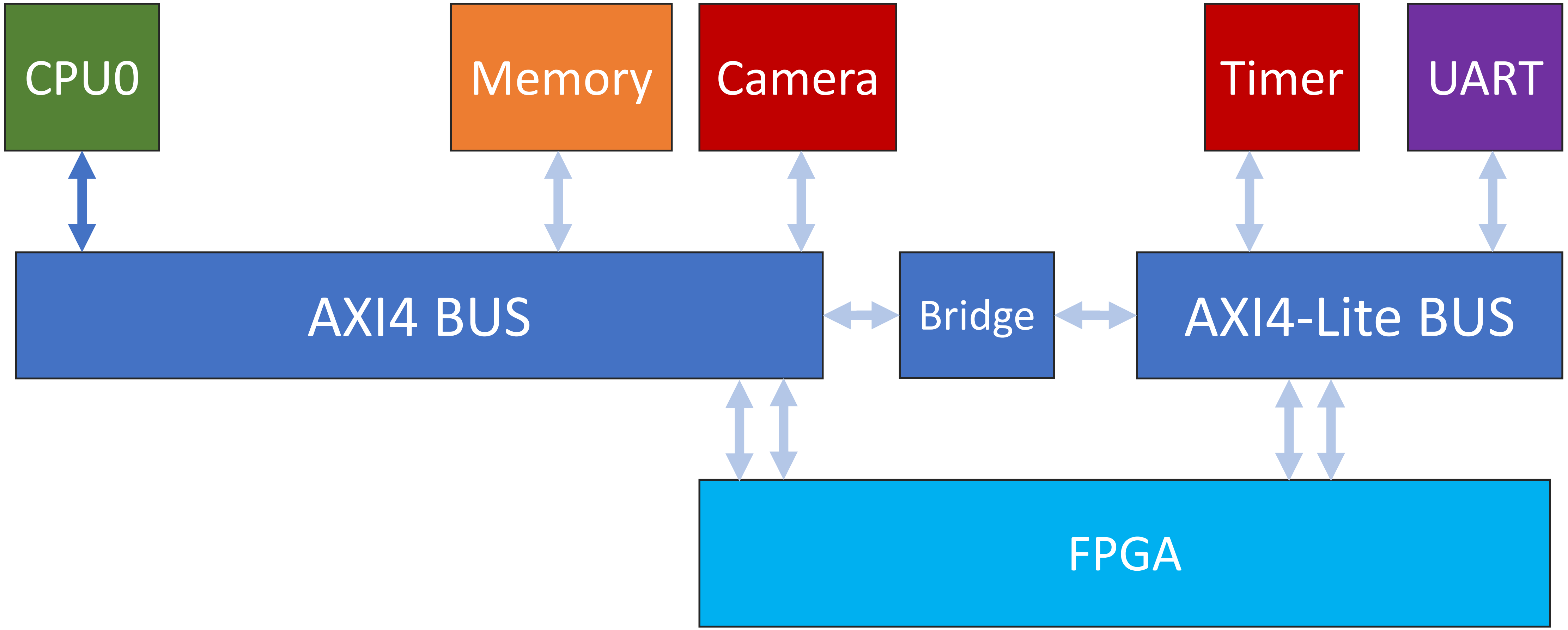
## Machine Model, Version 0



# Machine Model, V1



## Machine Model, V2



## MMIO from C.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define EMA_MMIO 0x40000000
int main () {
    volatile uint32_t * ema_ptr = (volatile uint32_t*)(EMA_MMIO);
    int32_t val = 0x1000;
    while (1) {
        //push new value into EMA
        *ema_ptr = val;
        //load value from EMA
        val = *ema_ptr;
        printf("Val: %d\n", val);
    }
    return 0;
}
```

four thousand
zero

const uint32_t const * x

## MMIO from C.

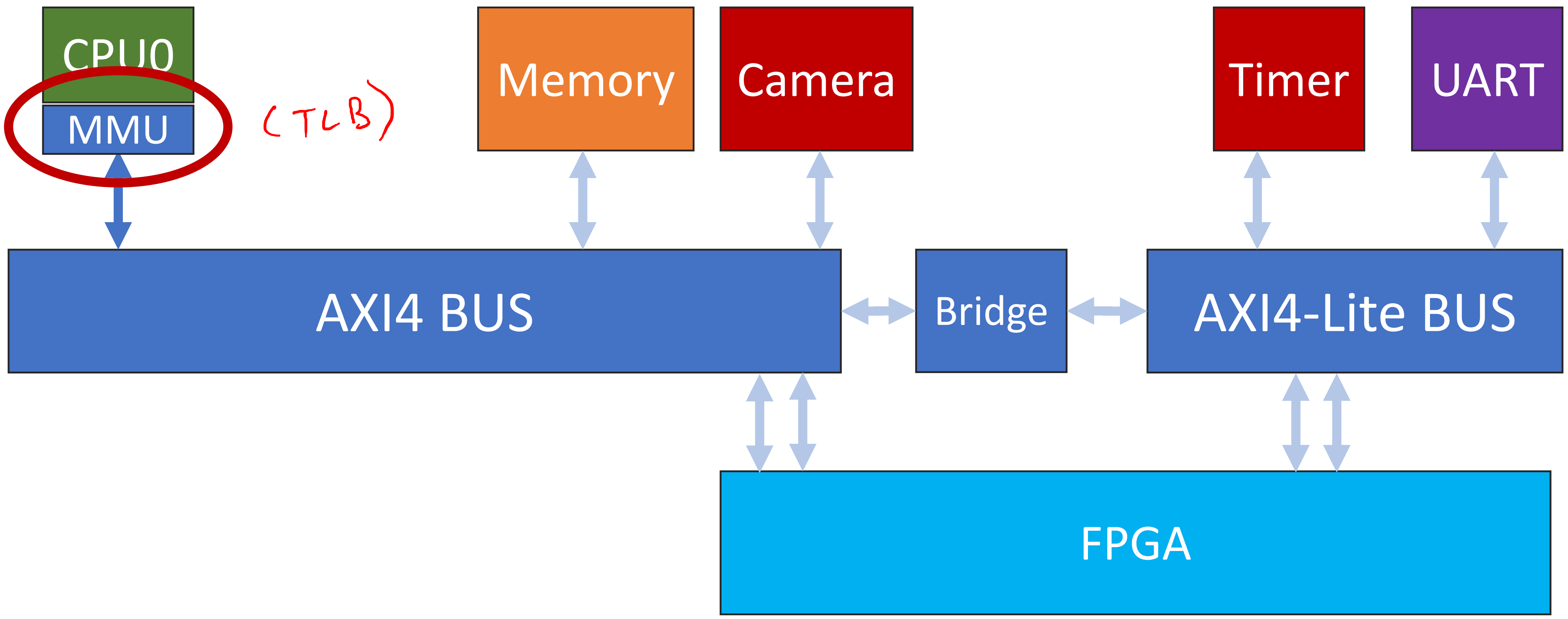
```
#define EMA_MMIO 0x40000000
volatile uint32_t * ema_ptr = (volatile uint32_t*)(EMA_MMIO);
//push new value into EMA
        *ema_ptr = val;
        //load value from EMA
        val = *ema_ptr;
```

```
   volatile uint32_t * ema_ptr = (uint32_t*)(EMA_MMIO);
8224:   e3a05101    mov r5, #1073741824 ; 0x40000000
   *ema_ptr = val;
822c:   e5854000    str r4, [r5]
    val = *ema_ptr;
8230:   e5954000    ldr r4, [r5]
```

## MMIO from C

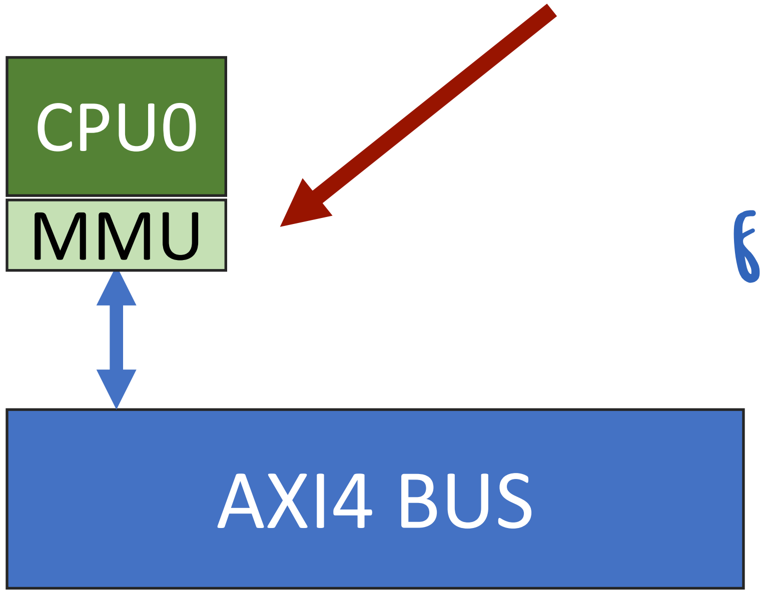
- WONT WORK!
- Why?
- Linux… and MMIOs

# Machine Model, V3:  MMUs



## MMU: Memory Management Unit (TLB)

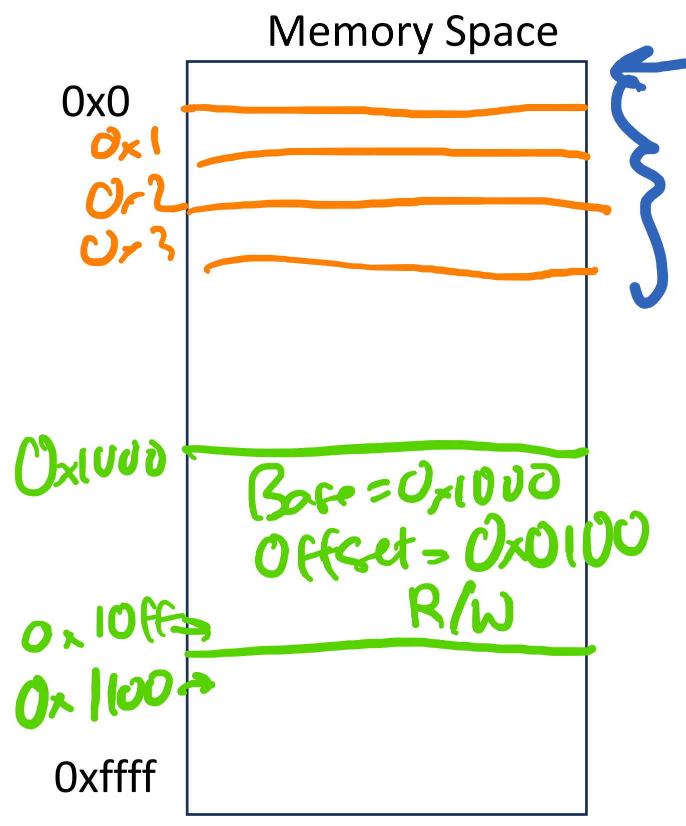
- Rejects load + stores that are "unauthorized"
- Translates addresses (Later)



$$\begin{tikzcd}
 & & \text{prog} \arrow[dl] \arrow[dr] & & & \\
\text{read-only} \arrow[r] & \underline{\text{code}} & & \text{data} \arrow[dl] \arrow[dr] & & \\
 & & \text{r/w} & & \underline{\text{const}} & \text{read-only}
\end{tikzcd}$$

TLB → translation lookaside buffer

## MMUs track the following things



0x0

- **BASE ADDRESS**: the start of a memory region that is allowed through the MMU

= 0x4

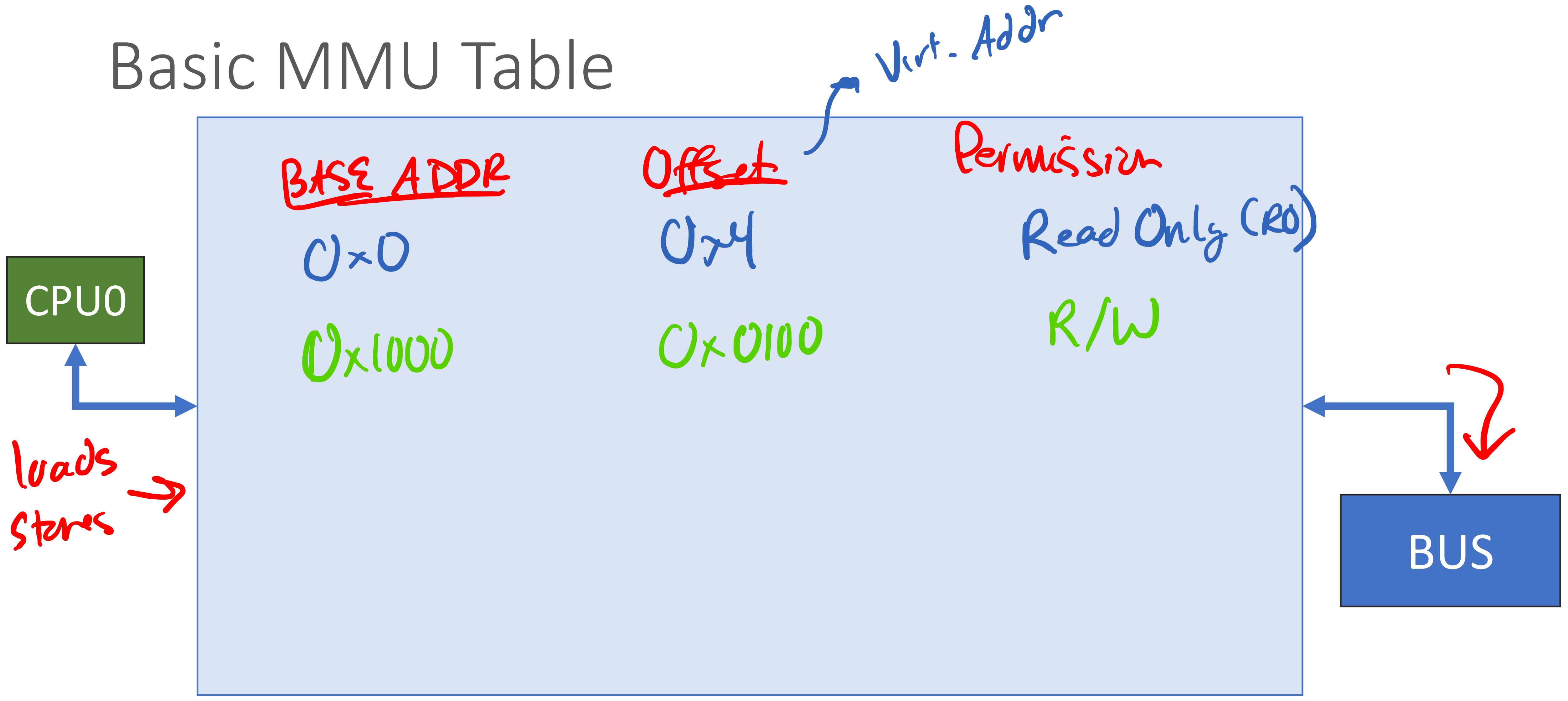
- **OFFSET**: the size of a memory region that is allowed through the MMU

→ Read-Only (RO)

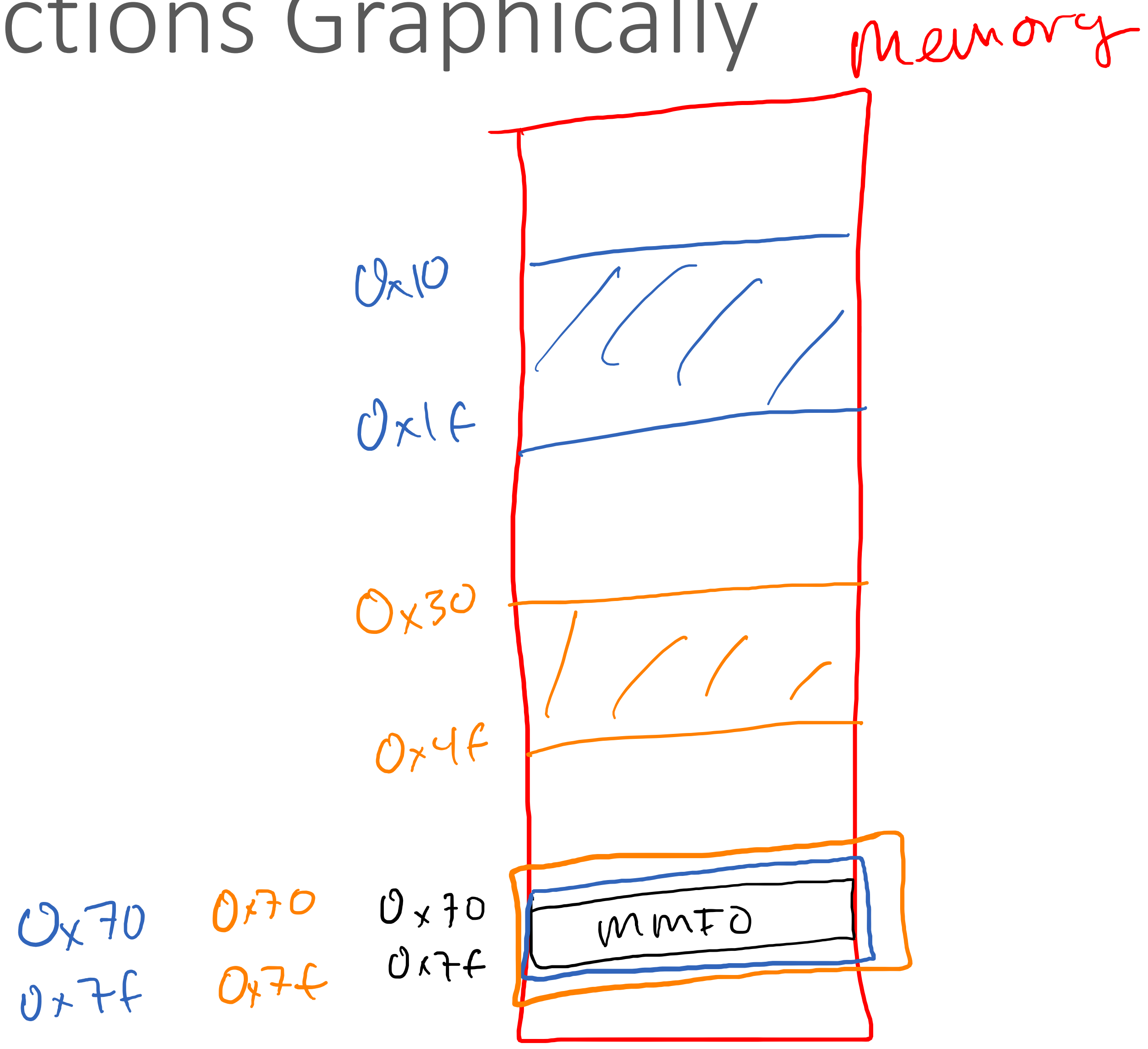
- **Permission**: the type of access that is allowed through the MMU

- Write Only (WO)
- Read/Write (R/W)

## Basic MMU Table



## Memory Protections Graphically



# Why?

- Security
  - Keep you from modifying the code
  - Keep you from executing the data

- Separate multiple applications

## References

- Zynq Book, Chapter 19 “AXI Interfacing”
- Practical Introduction to Hardware/Software Codesign
  - Chapter 10
- AMBA AXI Protocol v1.0
  - http://mazsola.iit.uni-miskolc.hu/~drdani/docs_arm/AMBAaxi.pdf

# 09: High-Performance Buses

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
*Indiana University*