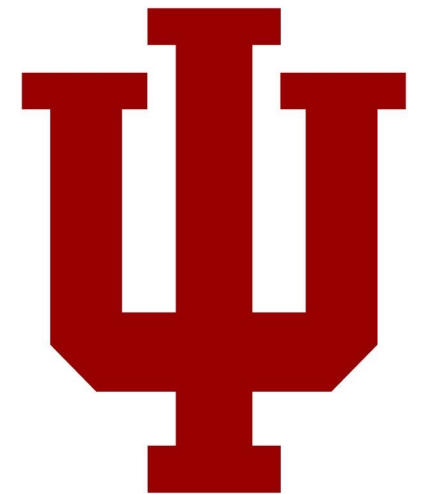# 11: Multi-Master Buses

Engr 315: Hardware / Software Codesign
Andrew Lukefahr
Indiana University

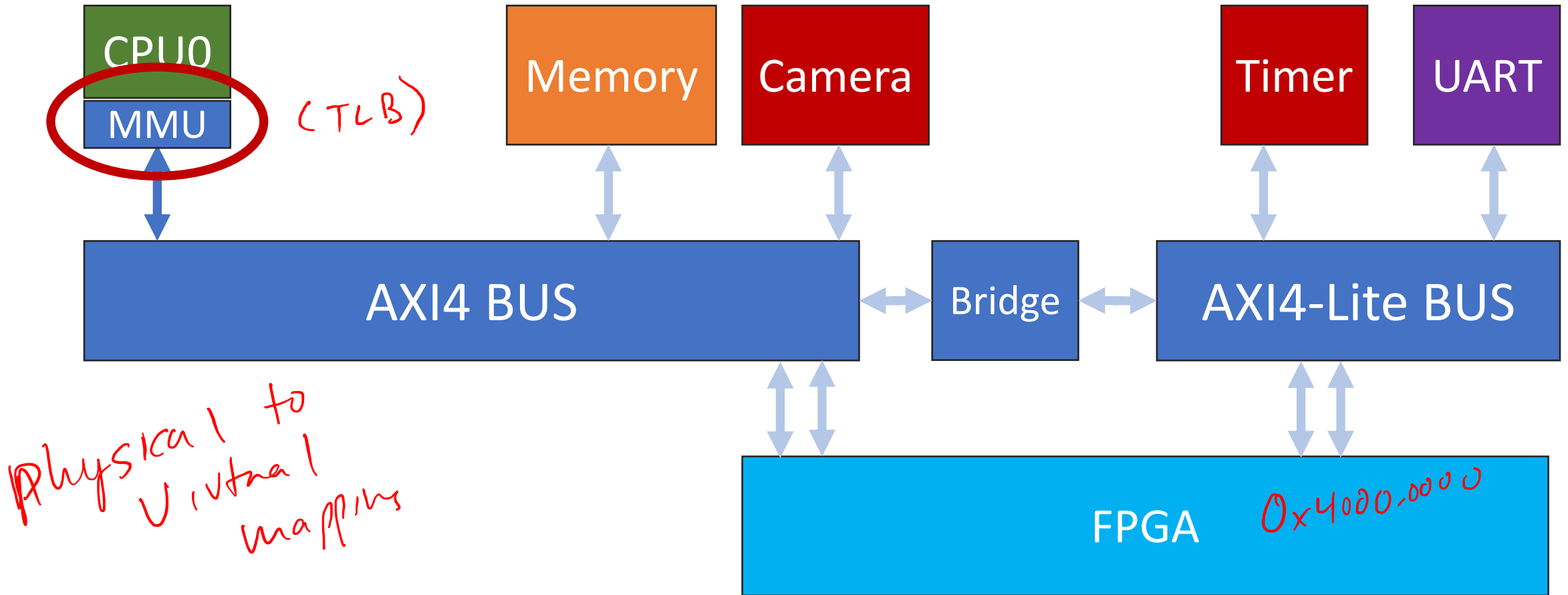# Announcements

- P5 is out!
  - Due Monday after break
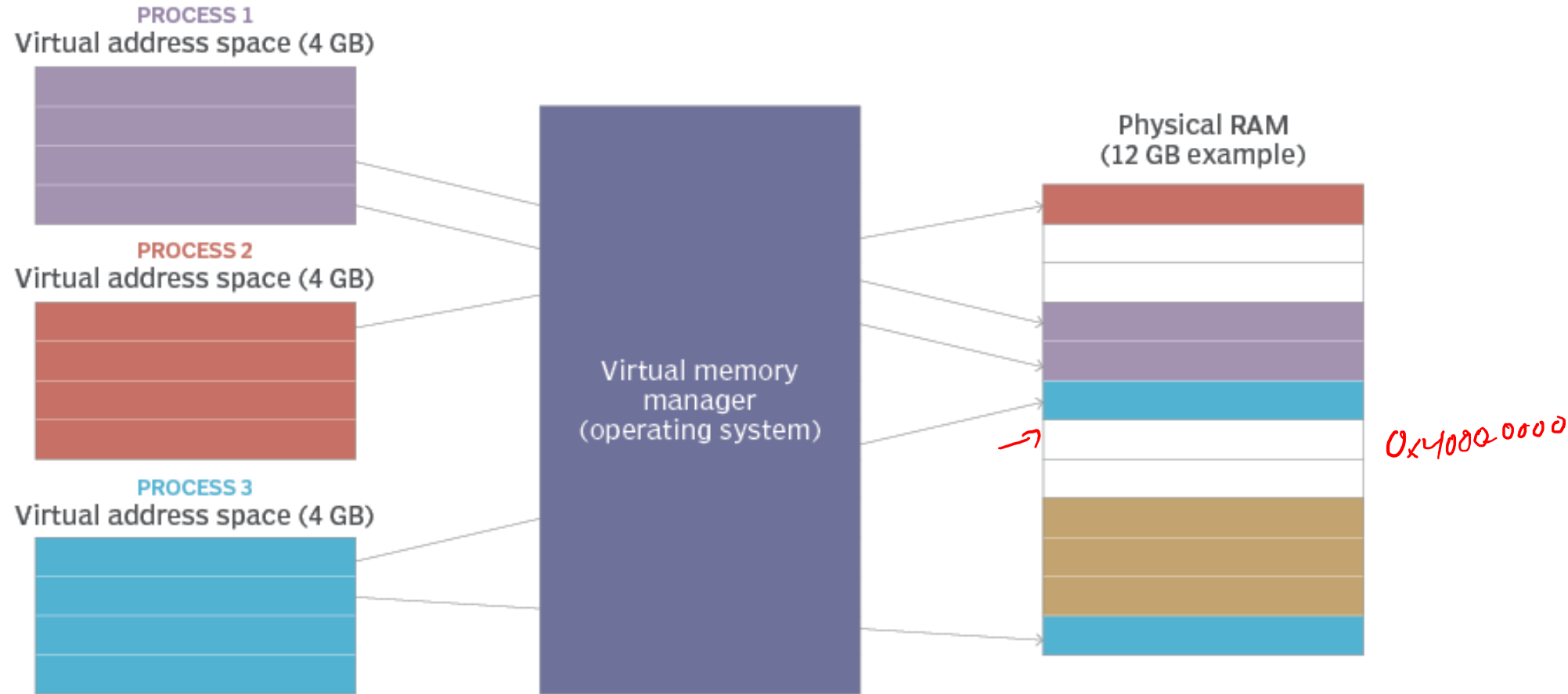
Exam Planning

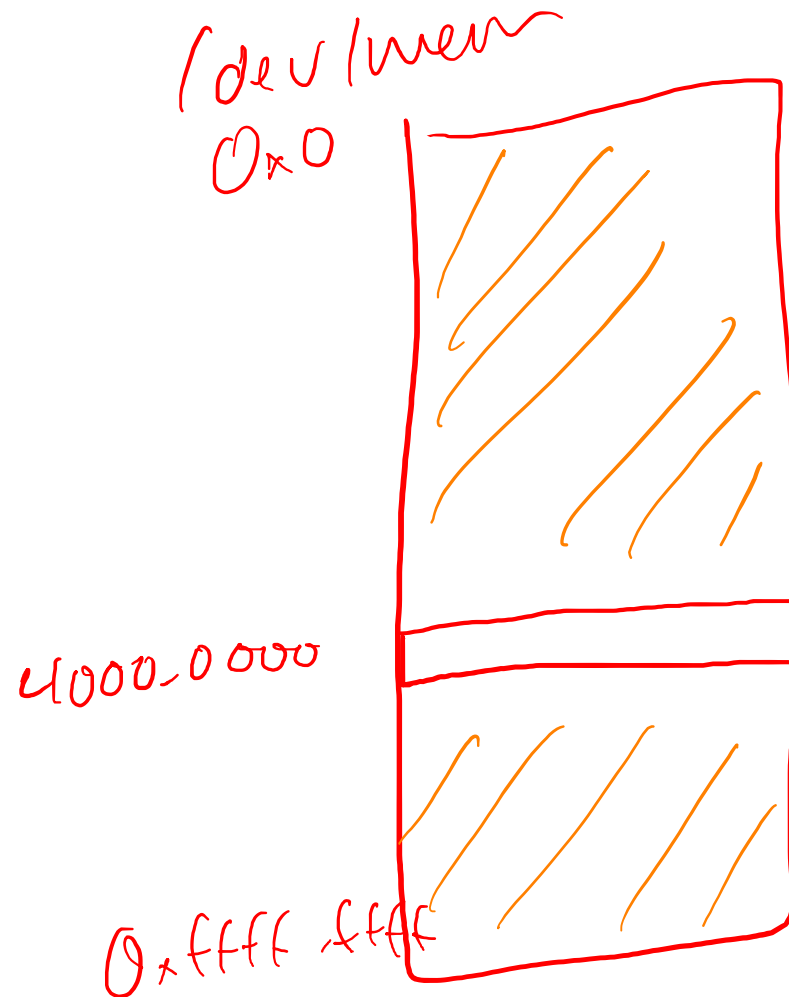| 10/26	Wednesday	–	Parallel Hardware III	P7 Linux DMA (C)
10/31	Monday	17	Parallel Hardware IV	
11/02	Wednesday	18	Review	–
11/07	Monday	19	Exam	

Machine Model, V3: MMUs

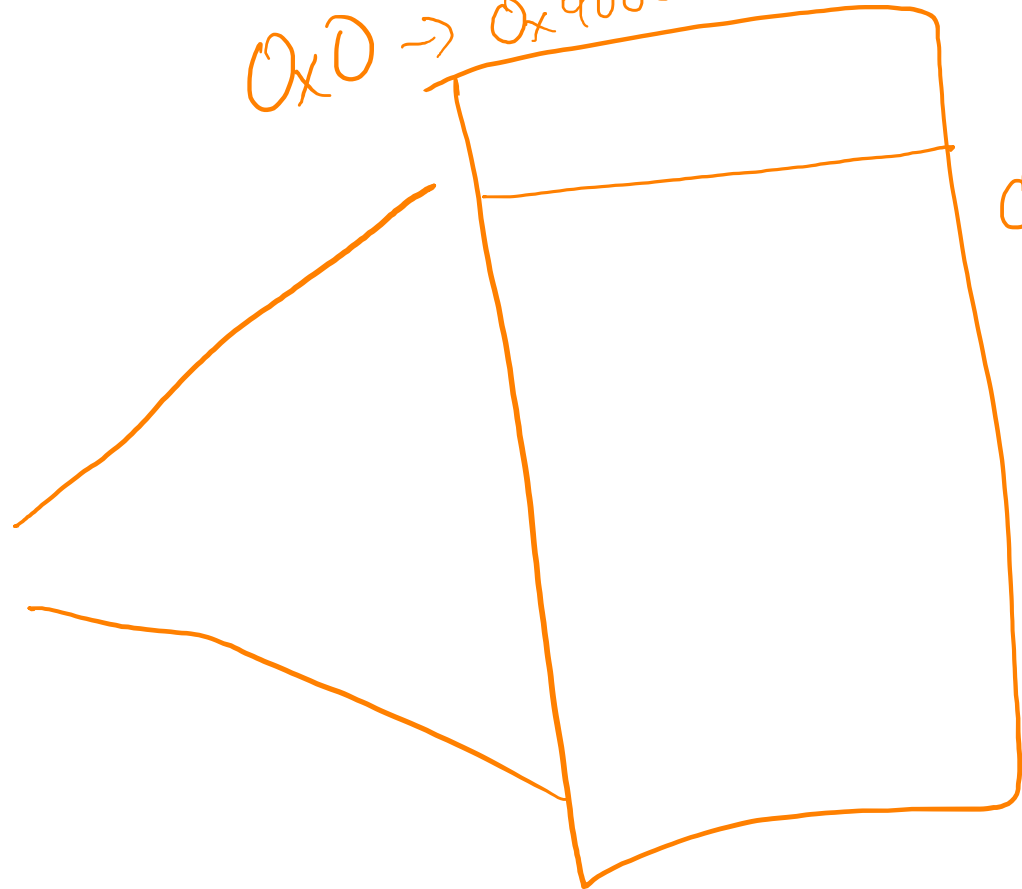MMU: Memory Management unit
TLB: Translation Look-aside buffer

CPU0
MMU (TLB)

Memory

Camera

Timer

UART

AXI4 BUS

Bridge

AXI4-Lite BUS

Physical to Virtual mapping

FPGA 0x4000.0000

OS (Linux) mains full Virtual->Physical Mappings

PROCESS 1
Virtual address space (4 GB)

PROCESS 2
Virtual address space (4 GB)

PROCESS 3
Virtual address space (4 GB)

Virtual memory manager (operating system)

Physical RAM (12 GB example)

0x4000 0000

/dev/uio

/dev/mem
0x0

4000_0000

0xffff_ffff

/dev/uio

0x0 → 0x4000_0000

0x4 →
~~0x4000_000~~
0x4000_0004

mmap

Physical mem

Linux /dev/mem

Linux /dev/uio

Linux mmap

FPGA →

0x4000-0000
0x4000-0004

Physical Addr

0x4000-0000
4000-0004

0x0
0x4

main
.
.
.

Data
:

virt address →

0xABC0
0xABC4

Userspace EMA w/C

```c
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

    // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
            MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```c
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Userspace EMA w/C

```c
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

    // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
            MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```c
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```
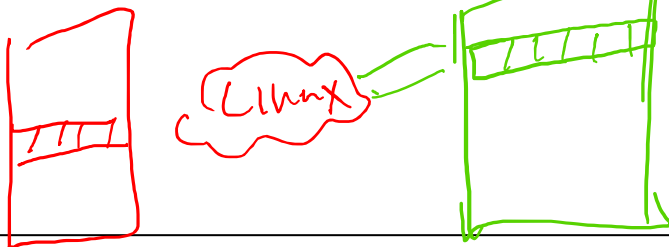
Userspace EMA w/C

```c
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

    // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
            MAP_SHARED, dev_mem_fd, 0x0);
```

4000 0000   w/ different name

```c
    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

Linux

```c
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }


    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Userspace EMA w/C

Virtual address base (handwritten annotation)

```c
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

    // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
            MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```c
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Userspace EMA w/C

```c
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

    // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
            MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```
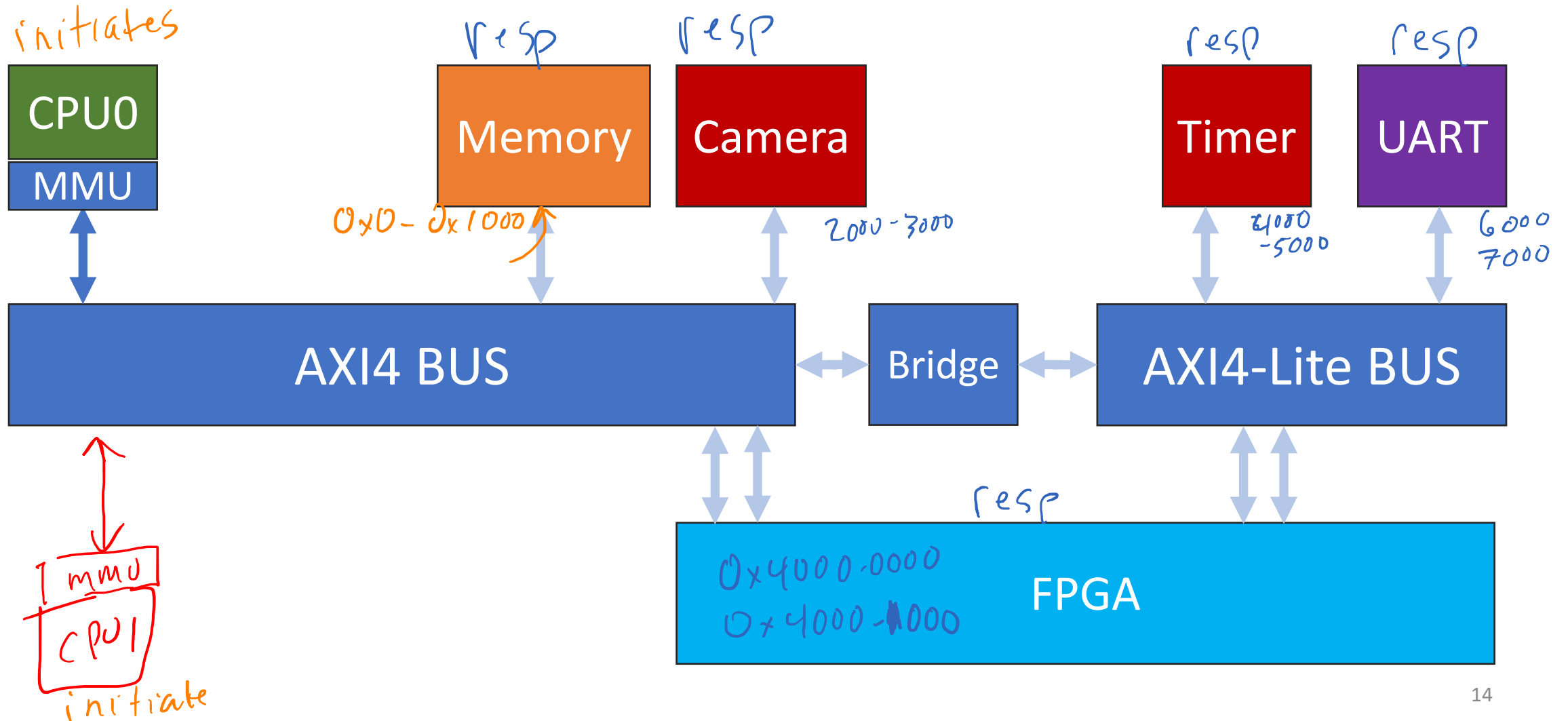
```c
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store to 0x4000_0000
        tmp = *ema_reg; //mmio load from 0x4000_0000
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Complete EMA

```c
#include <fcntl.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

    // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
            MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```c
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;

}
```
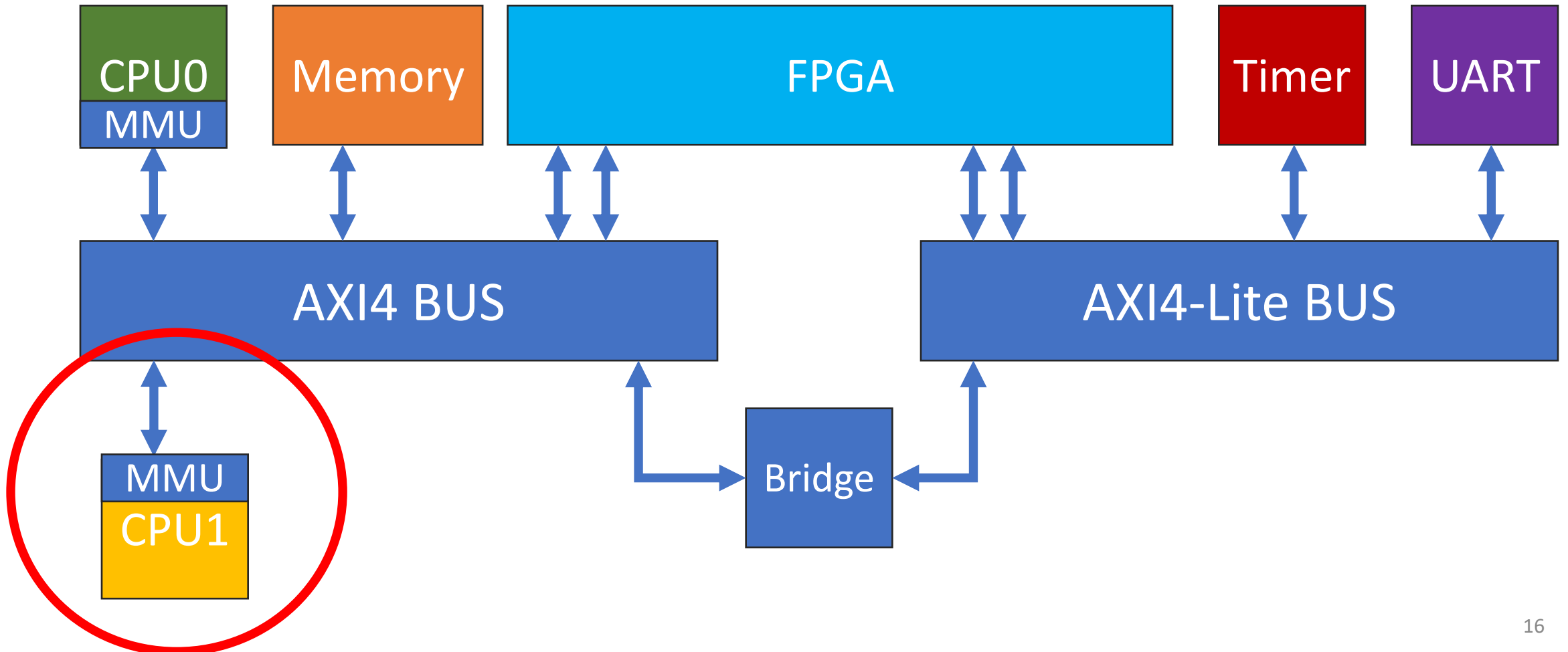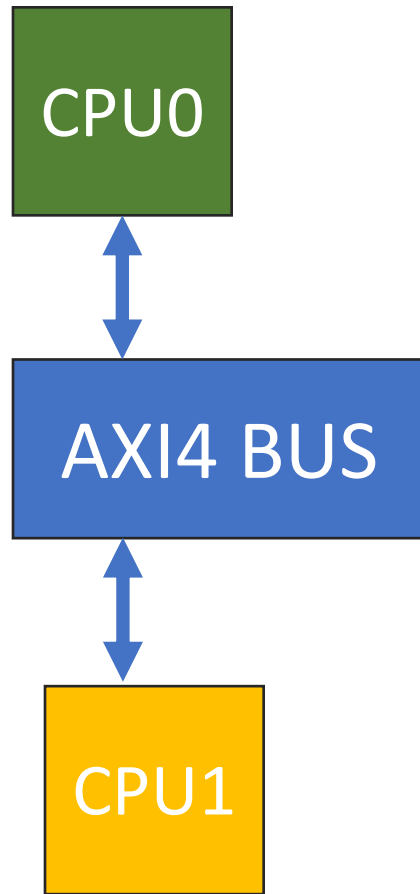
Which are AXI4 "Masters"?

initiates

resp   resp

CPU0

MMU

Memory   Camera

0x0 - 0x1000   2000-3000

resp   resp

Timer   UART

4000-5000   6000-7000

AXI4 BUS   Bridge   AXI4-Lite BUS

MMU

CPU1

initiate

resp

0x4000-0000
0x4000-4000   FPGA

14

Multiple Masters

Multi-Master Buses

CPU0

AXI4 BUS

CPU1

- What happens if both request a transaction at the same time?

An Arbitrator selects who gets to use the bus

CPU0

Arbitrator ←→ AXI4 BUS

CPU1

shortest-first
longest first
least-reciently used
random
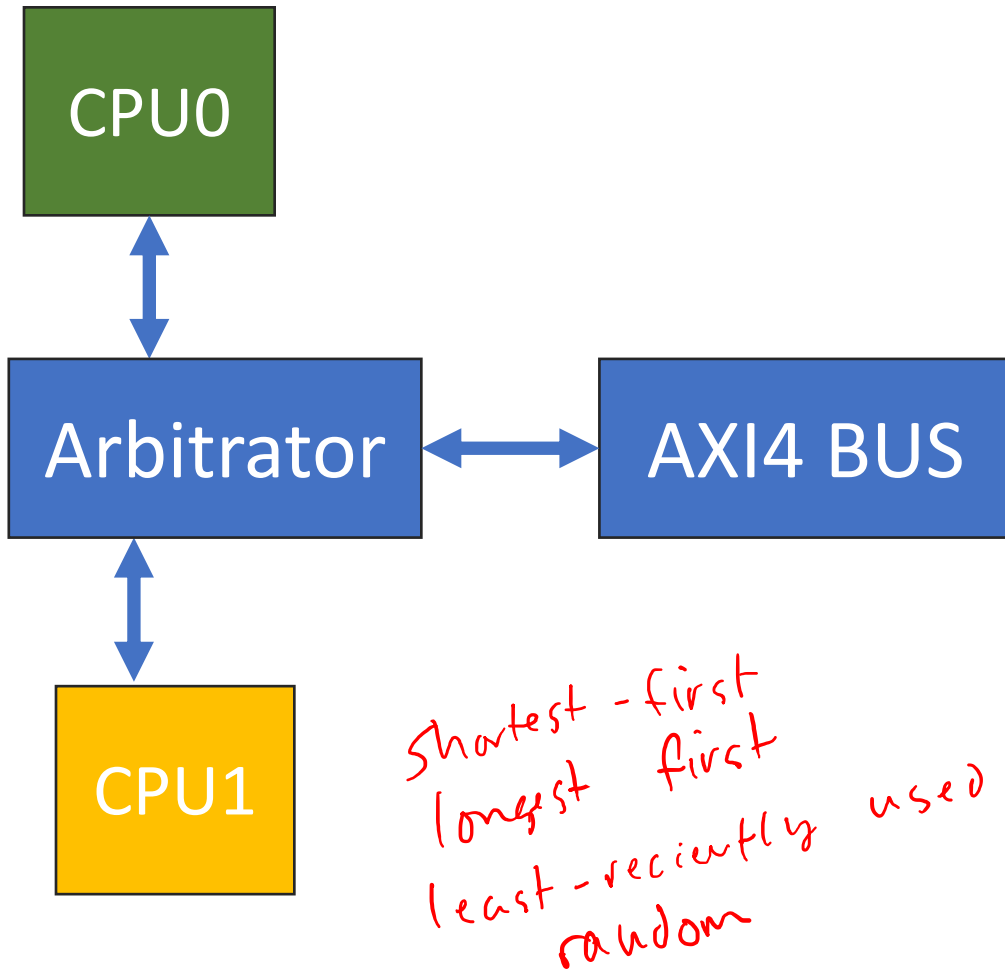
- What happens if both request a transaction at the same time?

- **Arbitration**: Pick a winner!

- What Arbitration scheme to use?
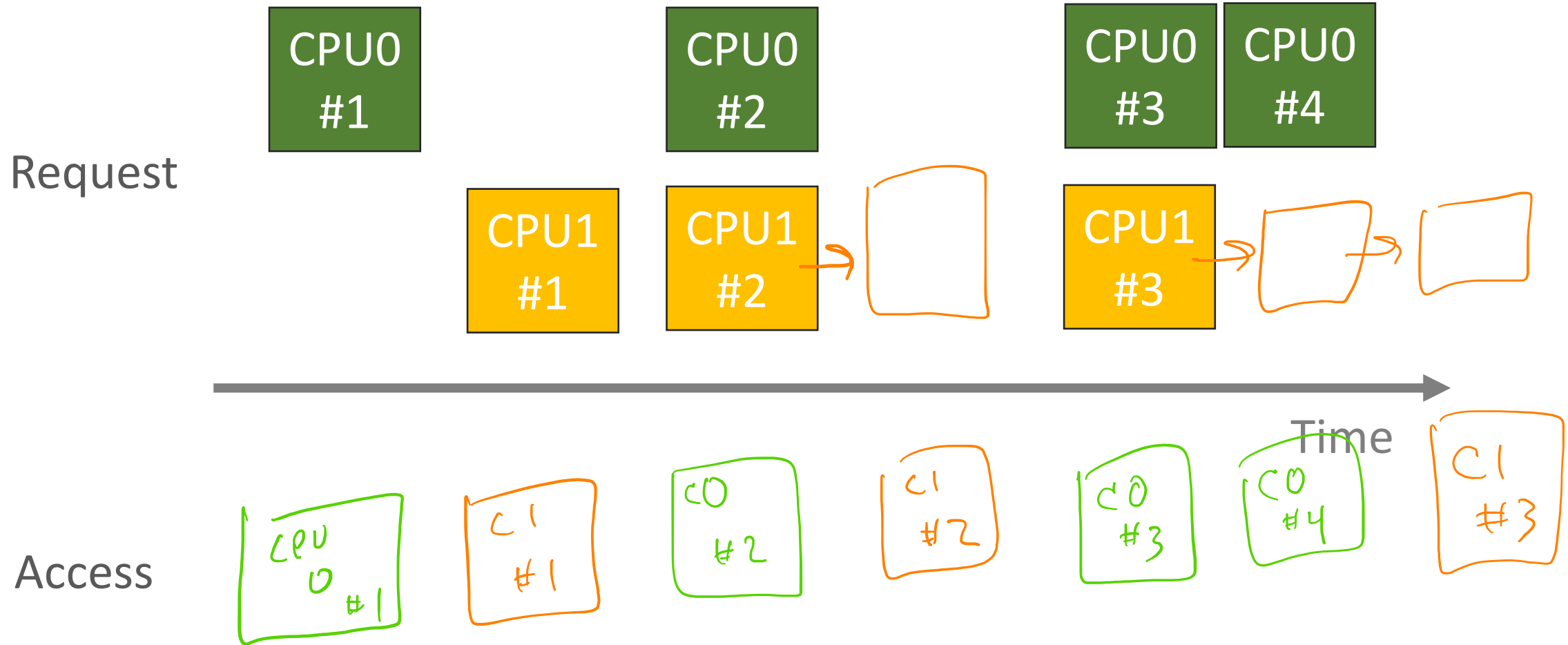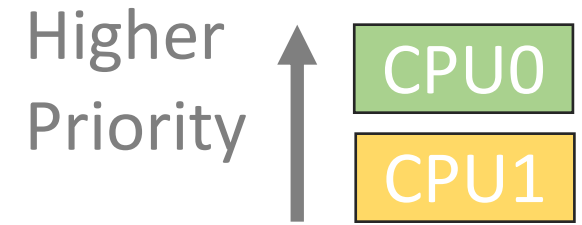 *highest priority*
  round-robin

19

Arbitration Options in E315

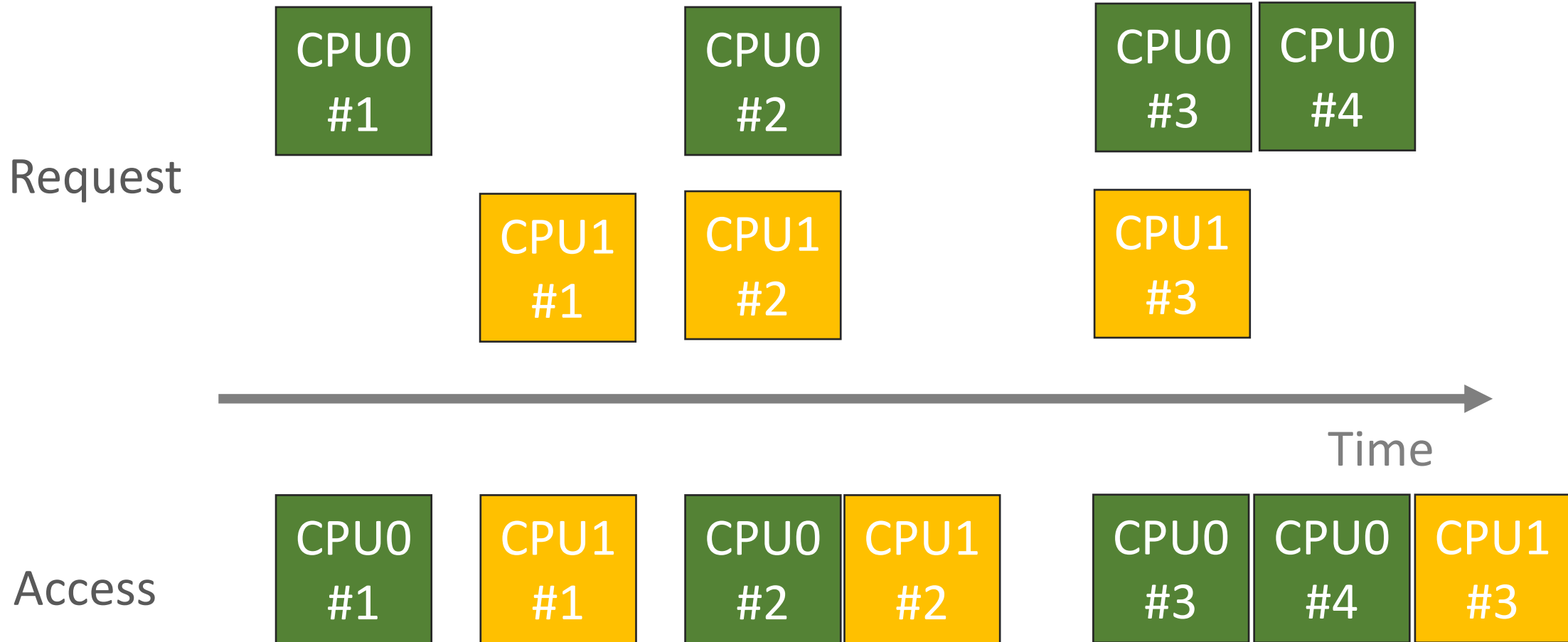- Highest-Priority First
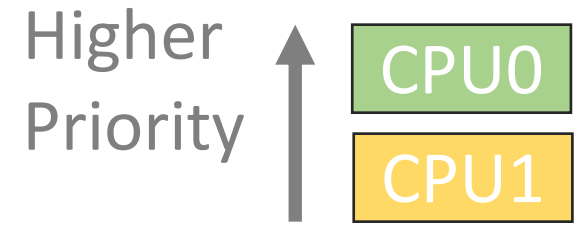
- Round Robin

Highest-Priority First
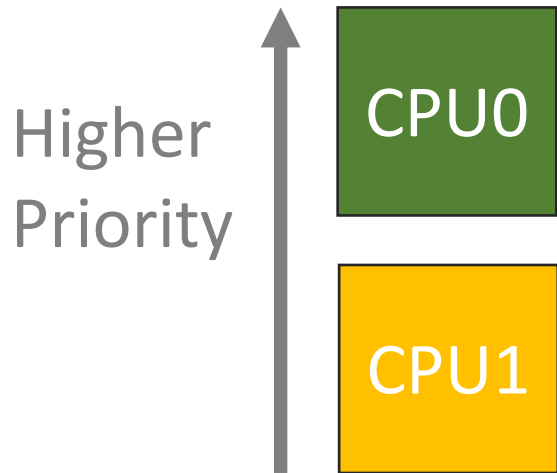
Higher Priority

- Priority Fixed. Highest priority requester always wins.

- Rules:
  - If CPU0 requests bus, CPU0 gets bus
  - Else if CPU0 does **not** request and CPU1 requests, CPU1 gets bus
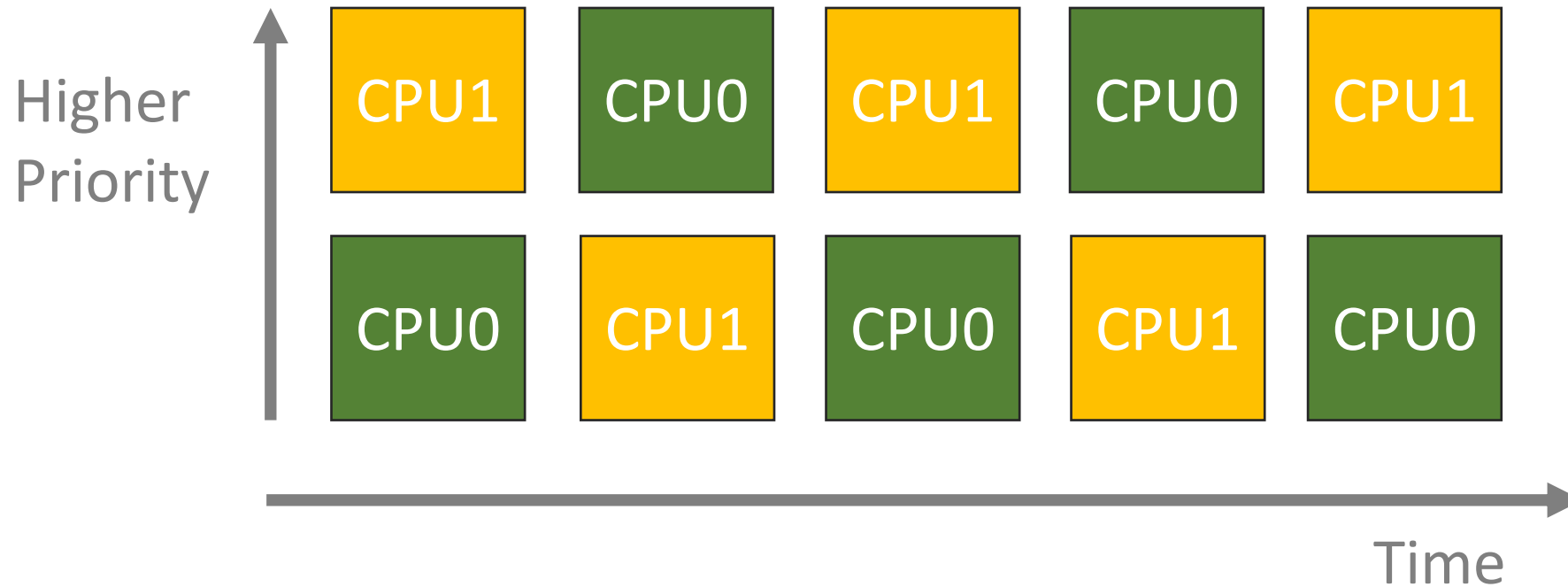 - Else bus idle
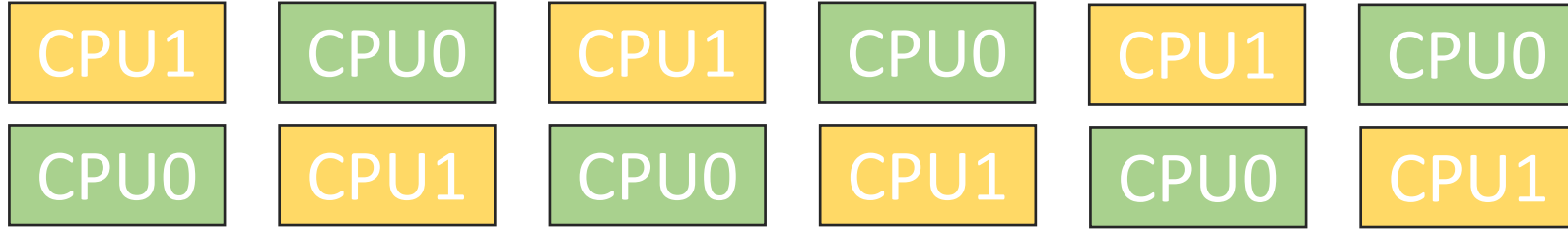
Highest Priority First

Arbitrator Circuit?

Skip

Round Robin

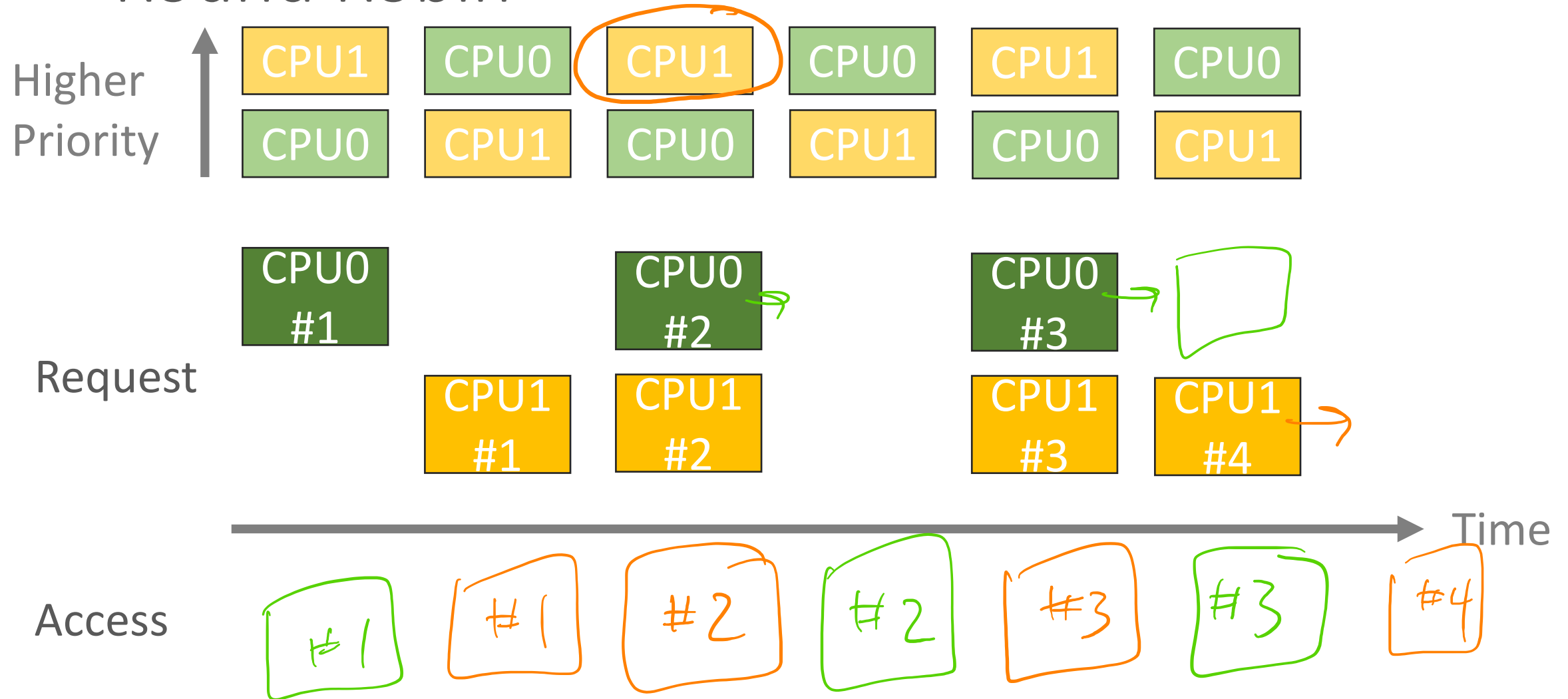- Priority updates every cycle. Everyone get's equal access to highest priority

Higher Priority

| CPU1 | CPU0 | CPU1 | CPU0 | CPU1 |
| CPU0 | CPU1 | CPU0 | CPU1 | CPU0 |

Time

Round Robin

Higher
Priority

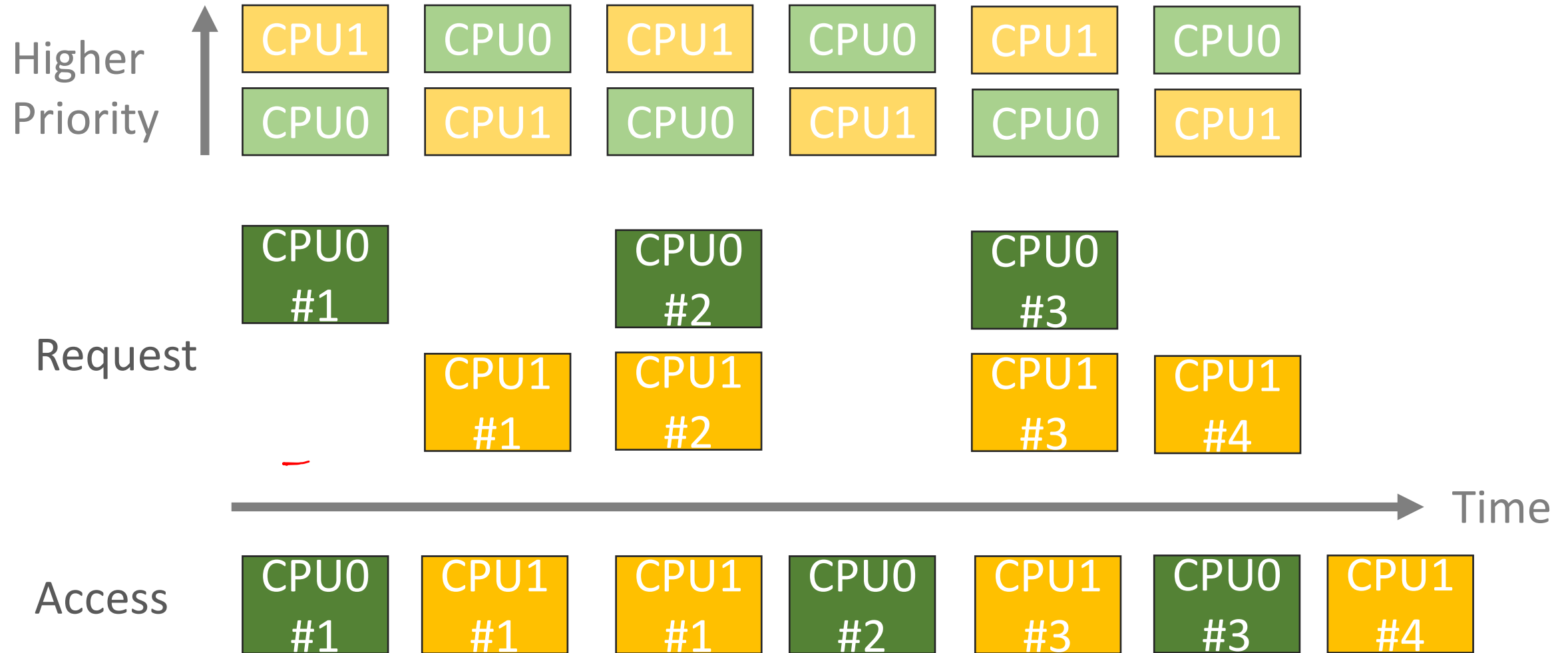| CPU1 | CPU0 | CPU1 | CPU0 | CPU1 | CPU0 |
| CPU0 | CPU1 | CPU0 | CPU1 | CPU0 | CPU1 |

Request

Time
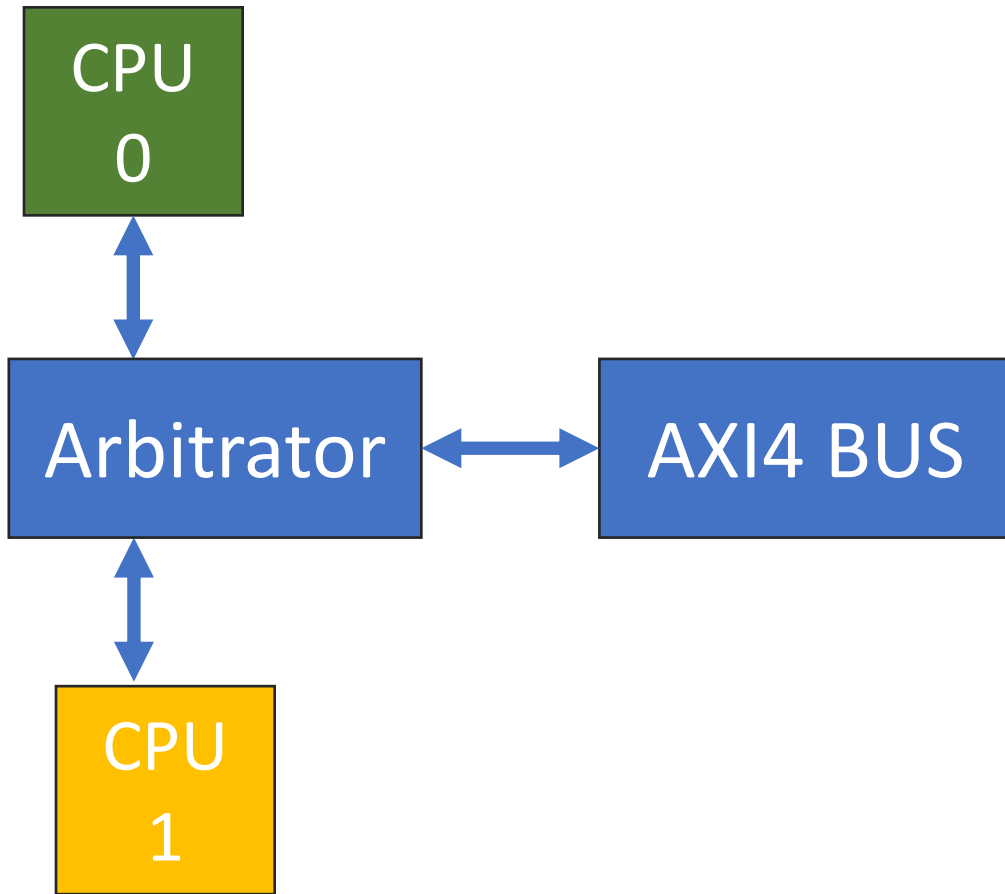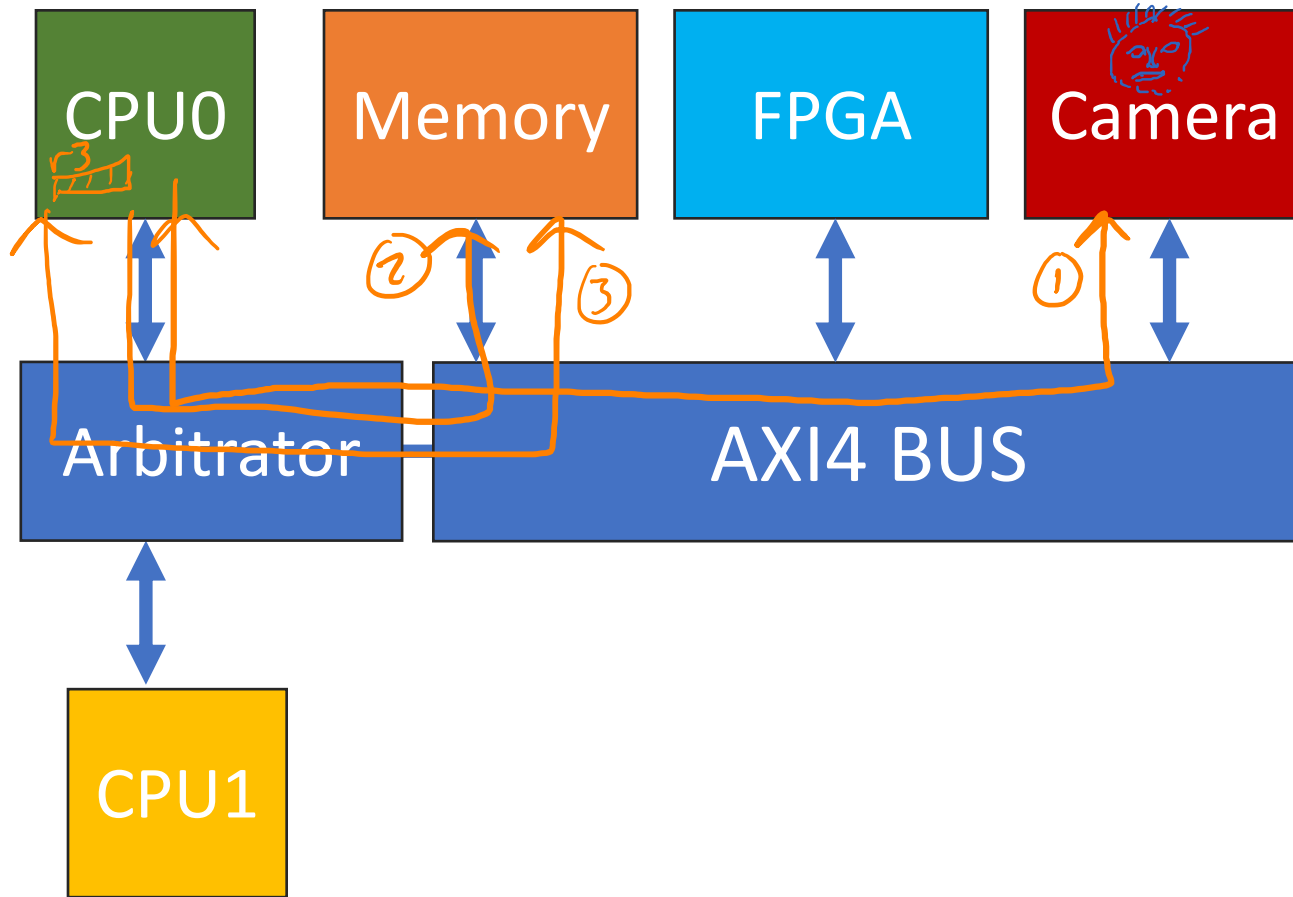
Access

Round Robin

Round Robin

Other Arbitration ideas?

An Arbitrator selects who gets to use the bus

- What happens if both request a transaction at the same time?

- Arbitration:
  - Fixed-Priority
  - Round Robin
  - Many more...

Q: How do I move data between the Camera and Memory?

A: The CPU copies data from Camera to Memory

#include "Magic_face_detect.h"

```
#define CAMERA_MMIO_ADDR 0x40000004

volatile uint32_t * camera =

        (uint32_t *)(CAMERA_MMIO_ADDR);

#define BUF_SIZE 1024;

uint32_t buf[BUF_SIZE];


int main () {

    for (i; ) {
        copy_image (camera, buf, 1024);
        detect_face (buf);
    }

}
```

```
void copy_image (uint32_t * from,
                 uint32_t * to,
                 uint32_t size)
{
    for ( i = 0; i < size, ++i) {
        to [i] = *(from )

    }

}
```

A: The CPU copies data from Camera to Memory

```
#define CAMERA_MMIO_ADDR 0x40000004

volatile uint32_t * camera =
        (uint32_t *)(CAMERA_MMIO_ADDR);

#define BUF_SIZE 1024;
uint32_t buf[BUF_SIZE];

int main () {
    //...
    while (true){
        copy_image(camera, buf, BUF_SIZE);
        detect_face(buf);
    }
}
```
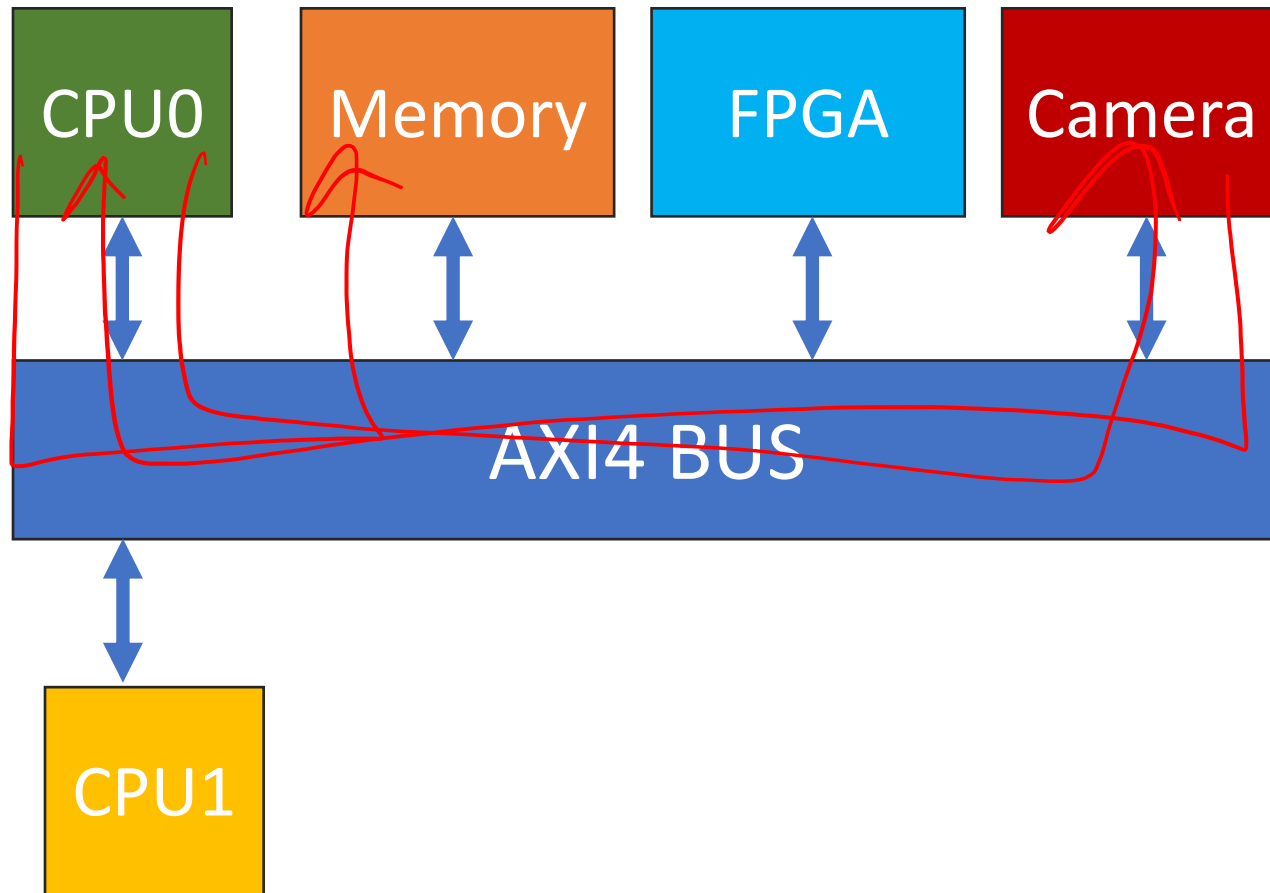
```
void copy_image (uint32_t * from,
                 uint32_t * to,
                 uint32_t size)
{
    volatile
    register uint32_t reg;

    for (int i = 0; i < size; ++i){

        reg = *from;

        to[i] = reg;
    }
}
```

What bus transactions result?

What else can the CPU do while copying data?

$$\underset{freq}{1 \text{ GHz}} \Rightarrow \quad | \quad 10^{-9} \text{ seconds} \quad (period)$$

$$| \quad 10^{9} \text{ cycles/second}$$

$$\sim | \text{ Billion operations/second}$$

What else can the CPU do while copying data?

- CPU can do 1B instructions/second. (1GHz)
- 4 Instructions per ~~loop~~ *copy loop*
  - 1 load, 1 store, 1 increment, 1 branch
- 250M copies/second

4K Video: 1697 Mbps* = 212 MB / second

~85% CPU utilization for Copy!

What about Ethernet?

- CPU can do 1B instructions/second. (1GHz)
- 4 Instructions per loop
  - 1 load, 1 store, 1 increment, 1 branch
- 250M copies/second

- 1Gbps Ethernet:
- 1 Gbps Receive + 1Gbps Transmit = 2 Gbps
- 2Gbps = 250MB/second

- **Nothing. ~100% of CPU required?**

What if we do the copy on CPU1?

```
int main () {

    while (true){

        ask_cpu1_to_copy_image(camera, buf, BUF_SIZE);


        detect_face(buf);
    }
}
```
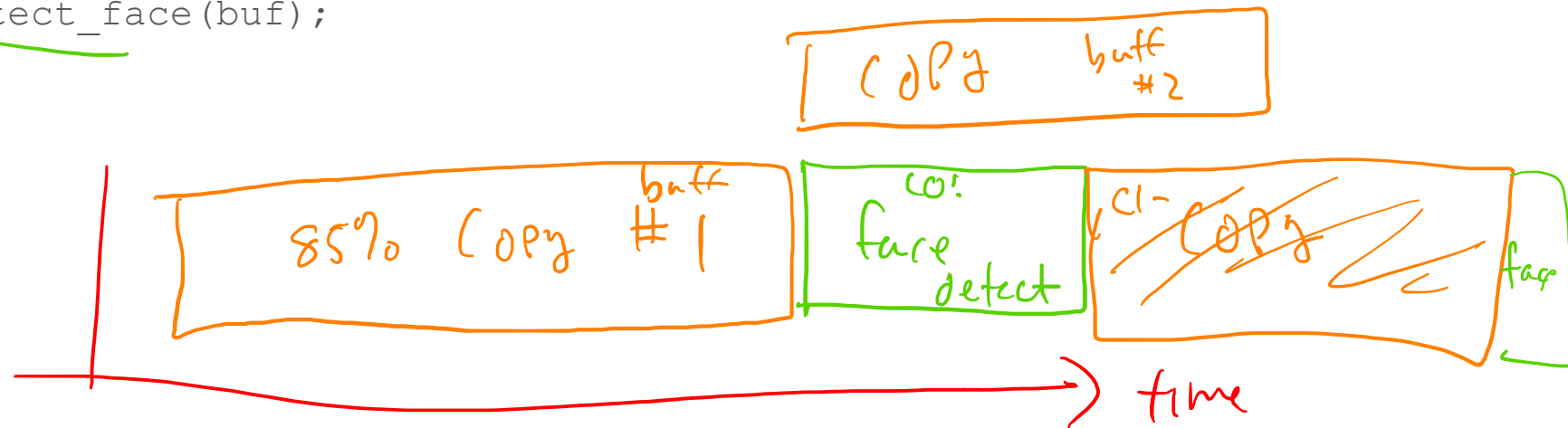
What if we do the copy on CPU1?

int [size] buf1;

int [size] buf2;

```
int main () {
        ask_ ... image (buf2)
    while (true){
```

buf1

```
        ask_cpu1_to_copy_image(camera, ~~buf~~, BUF_SIZE);
```

detect_face (buf2);

ask_cpu1_to_copy_image (camera, buf2, BUF_SIZE)

buf1

```
        detect_face(~~buf~~);
    }

}
```

What if we do the copy on CPU1?

```
int main () {

    while (true){

        ask_cpu1_to_copy_image(camera, buf, BUF_SIZE);

        wait_for_cpu1_done();

        detect_face(buf);
    }
}
```

time

double buffering

CPU0

empty
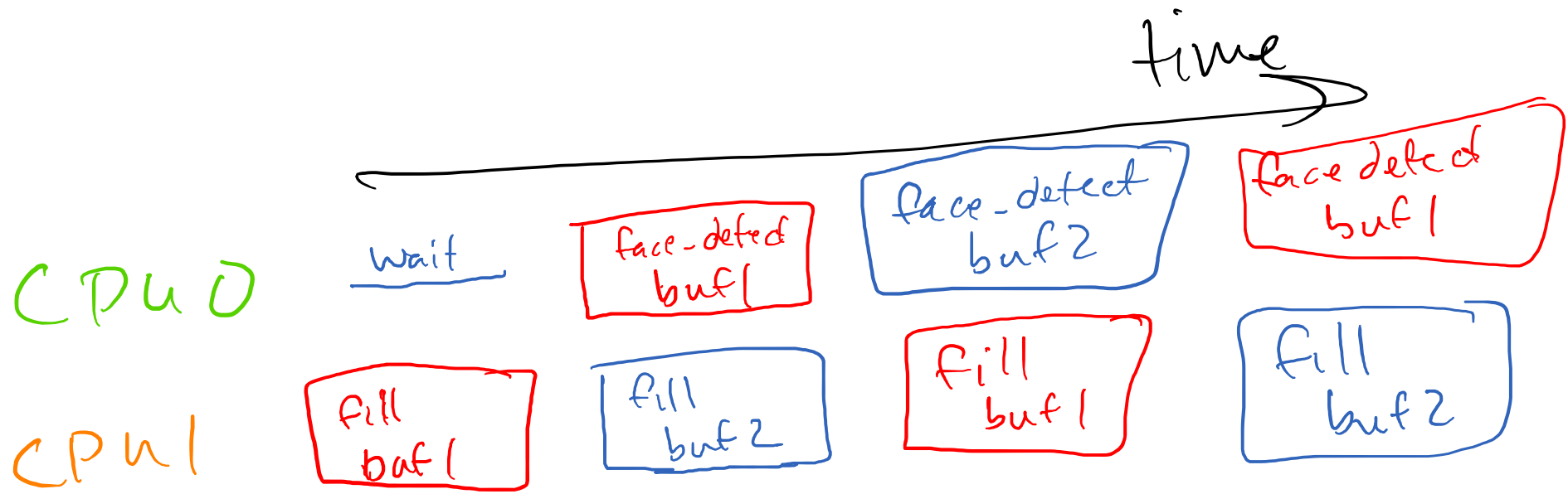Buf0

empty
Buf1

buf0

CPU1

fill
Buf0

Buf1

fill
Buf0

buf1

↑
overlap
(parallelism)

51

Copy on CPU1, Version 2.

```
int main () {

    ask_cpu1_to_copy_image(camera, buf1, BUF_SIZE);
    wait_for_cpu1_done();

    while (true){
        ask_cpu1_to_copy_image(camera, buf2, BUF_SIZE);
        detect_face(buf1);
        wait_for_cpu1_done();

        ask_cpu1_to_copy_image(camera, buf1, BUF_SIZE);
        detect_face(buf2);
        wait_for_cpu1_done();
    }

}
```

time

CPU 0

Wait | face-detect buf1 | face-detect buf2 | face-detect buf1

CPU 1

fill buf1 | fill buf2 | fill buf1 | fill buf2

54

Why are we wasting an entire CPU for this?

```
void copy_image (uint32_t * from,
                 uint32_t * to,
                 uint32_t size)
{
    register uint32_t reg;


     for (int i = 0; i < size; ++i){


         reg = *from;


         to[i] = reg;
     }

}
```

DMA: Direct Memory Access

- A mini-CPU that does copy for you:

```
void copy (uint32_t * from,
           uint32_t * to,
           uint32_t size)
{
    register uint32_t reg;

    for (int i = 0; i < size; ++i){

        reg = *from;

        to[i] = reg;
    }
}
```

DMA

```
int main () {

    dma_copy_image(camera, buf1, BUF_SIZE);
    wait_for_dma_done();

    while (true){
        dma_copy_image(camera, buf2, BUF_SIZE);
        detect_face(buf1);
        wait_for_dma_done();

        dma_copy_image(camera, buf1, BUF_SIZE);
        detect_face(buf2);
        wait_for_dma_done();
    }

}
```

57

Next Time:

- Direct Memory Access

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

- https://lauri.võsandi.com/hdl/zynq/axi-stream.html

11: Multi-Master Buses

Engr 315: Hardware / Software Codesign
Andrew Lukefahr
Indiana University