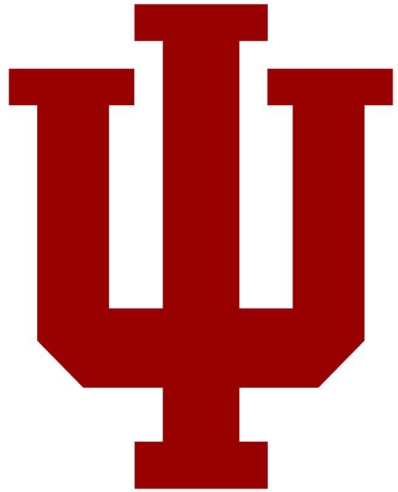
11: Multi-Master Buses

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



Announcements

- P3 is due Wednesday
- P4 is out.
  - Expect some revisions
  - Use P3 bitstream / hwh files
- P5 is out.

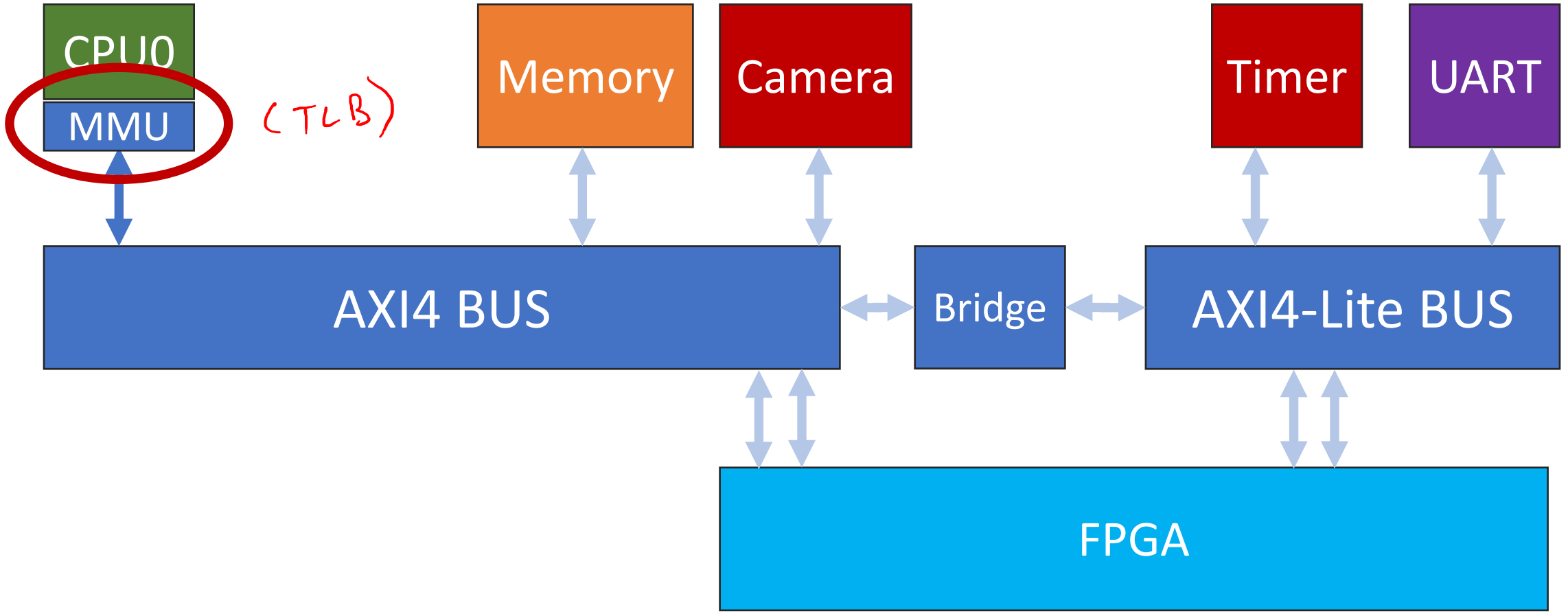
PG "commits"

Exam Planning

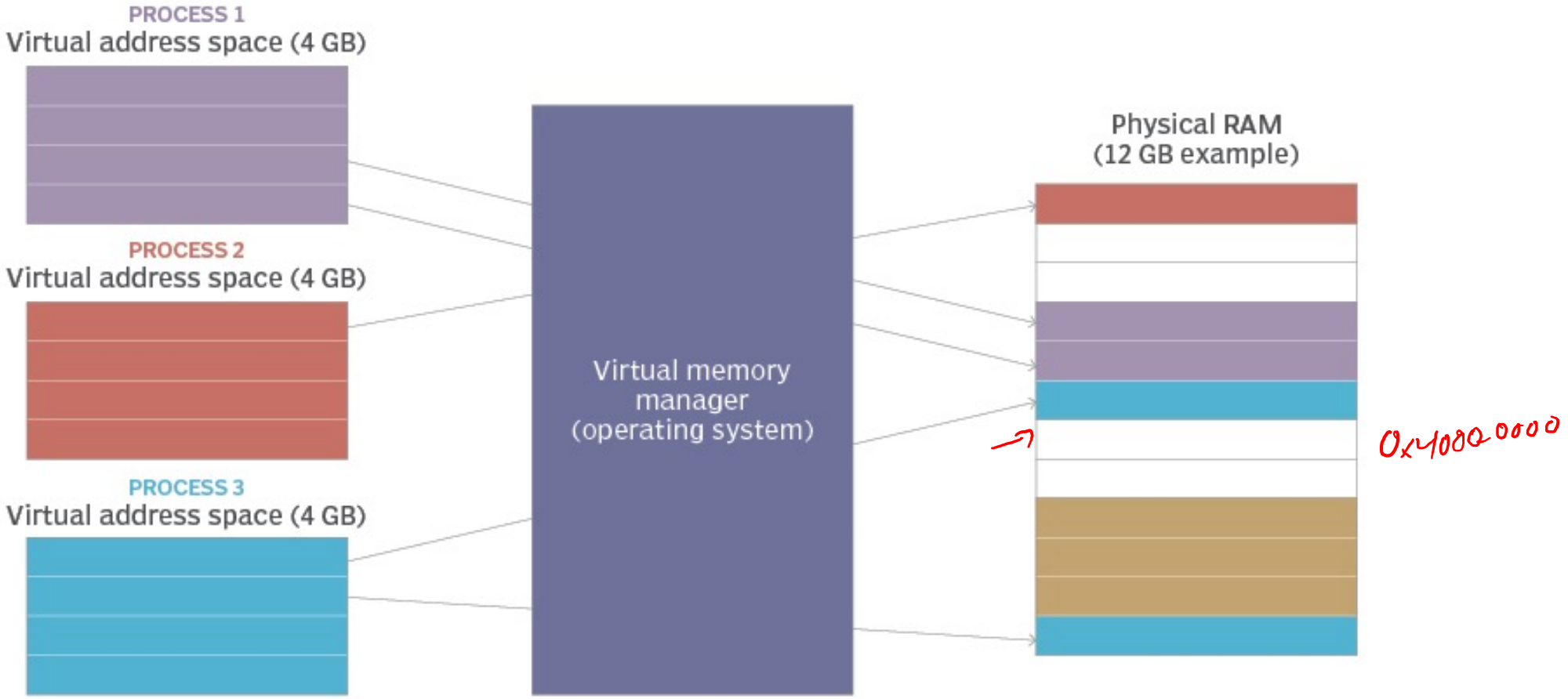
3/15	Tuesday	–	SPRING BREAK	
3/17	Thursday	–	SPRING BREAK	P7 Dot (V)
3/22	Tuesday	17	Review	
3/24	Thursday	18	~~Exam~~ Review	P7 Dot (V)
3/29	Tuesday	19	~~Linux Kernel I~~ Exam	
3/31	Thursday	20	Linux Kernel II	PX Accel Dot (V)
4/05	Tuesday	21	Linux Kernel III	

if objections: please let me know by Tues 2/22

Machine Model, V3: MMUs

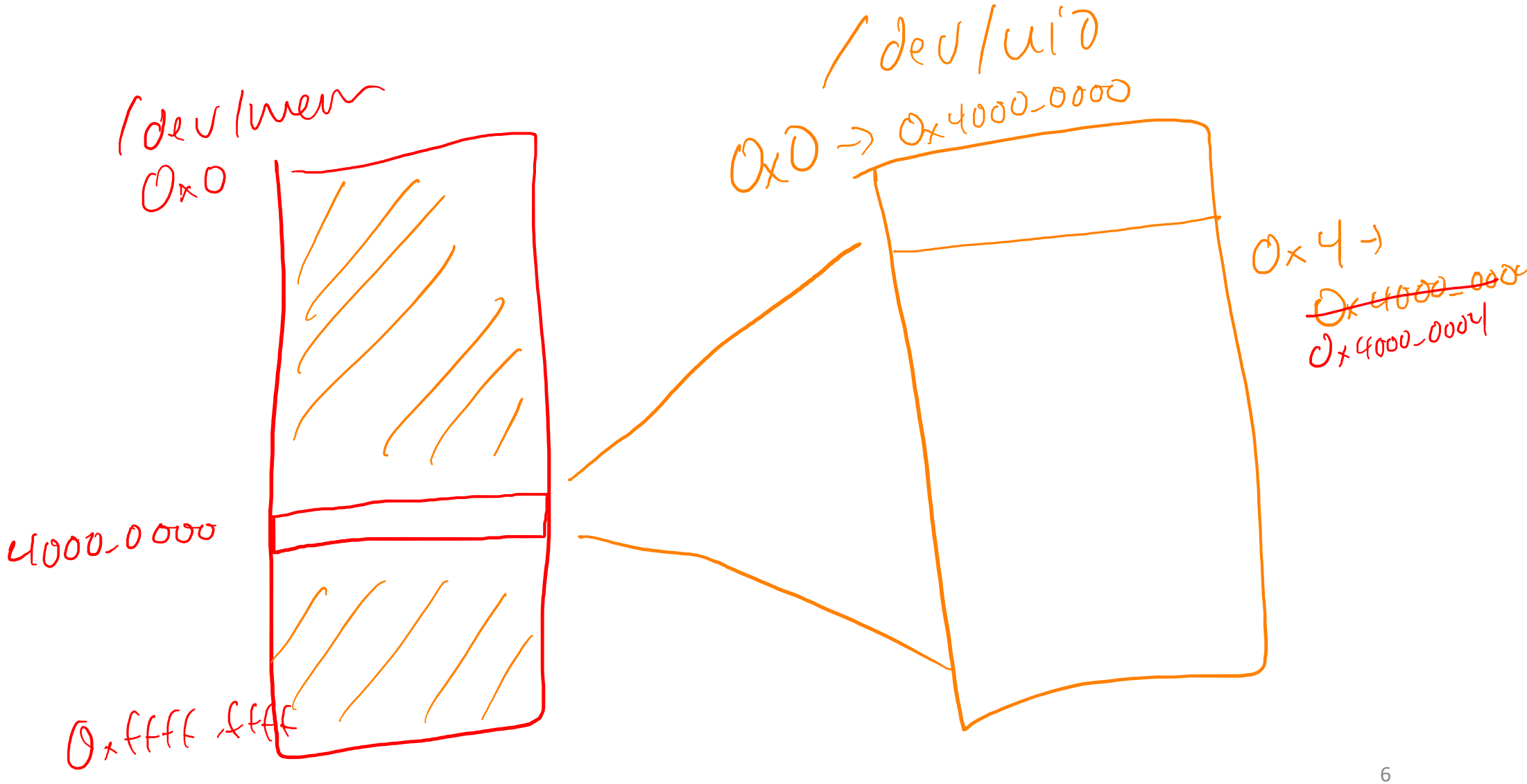


OS (Linux) mains full Virtual->Physical Mappings

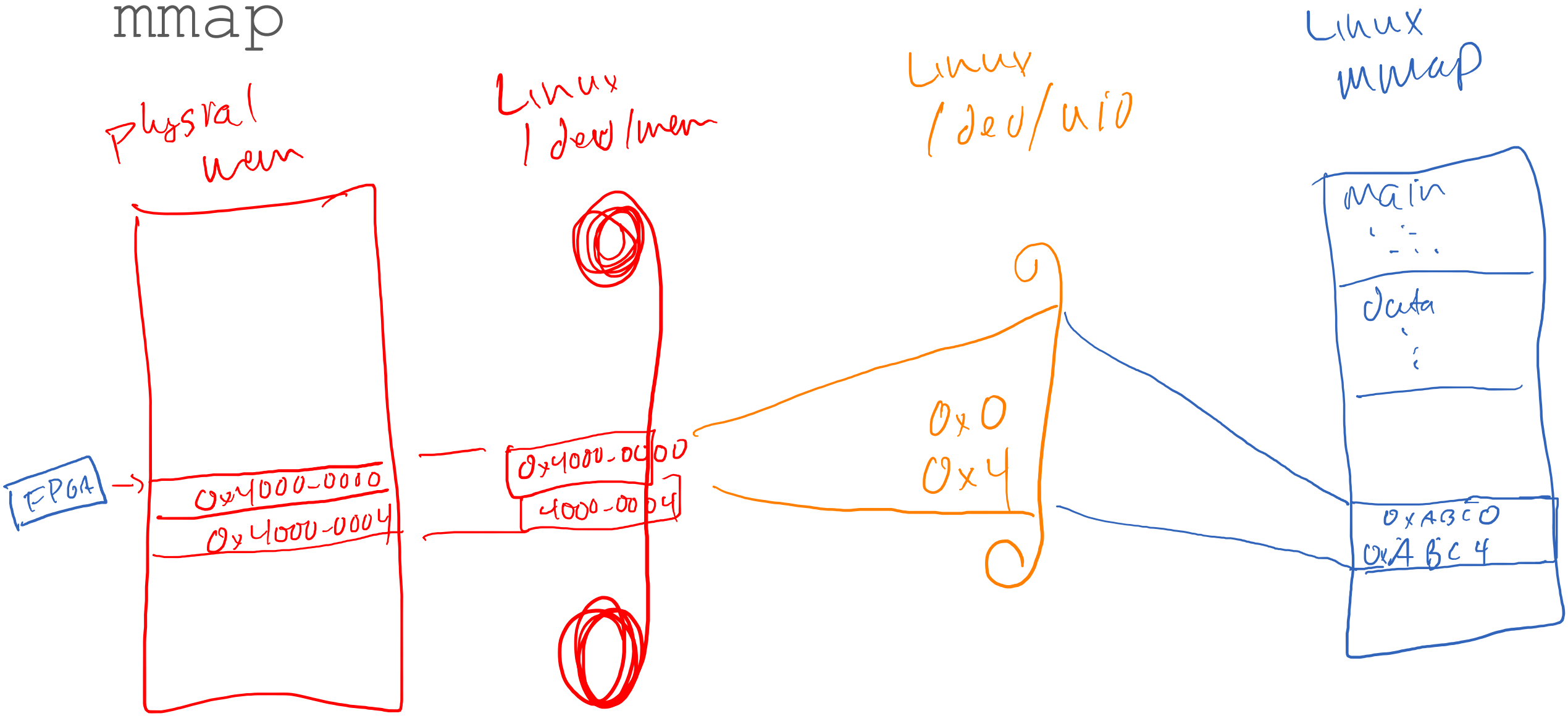


https://www.techtarget.com/searchstorage/definition/virtual-memory

/dev/uio


/dev/mem
0x0
4000_0000
0xffff_ffff
/dev/uio
0x0 → 0x4000_0000
0x4 →
0x4000_0004

mmap


Physical mem
Linux /dev/mem
Linux /dev/uio
Linux mmap
FPGA
0x4000-0010
0x4000-0004
0x4000-0000
4000-0004
0x0
0x4
main
Data
0xABC0
0xABC4

Userspace EMA w/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; }
```

```
    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;
}
```

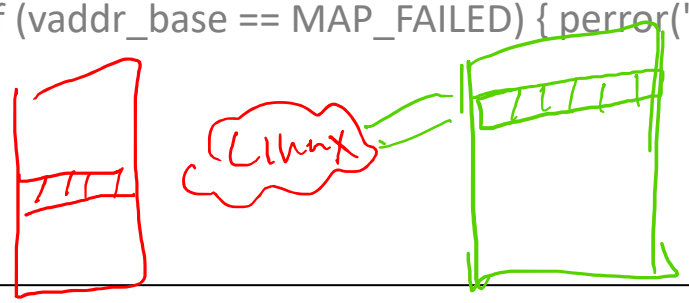
Userspace EMA w/C

```
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; }
```

```
    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

```
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; }
```




```
    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

```
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; }
```

```
    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

```
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; }
```

```
    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;
}
```

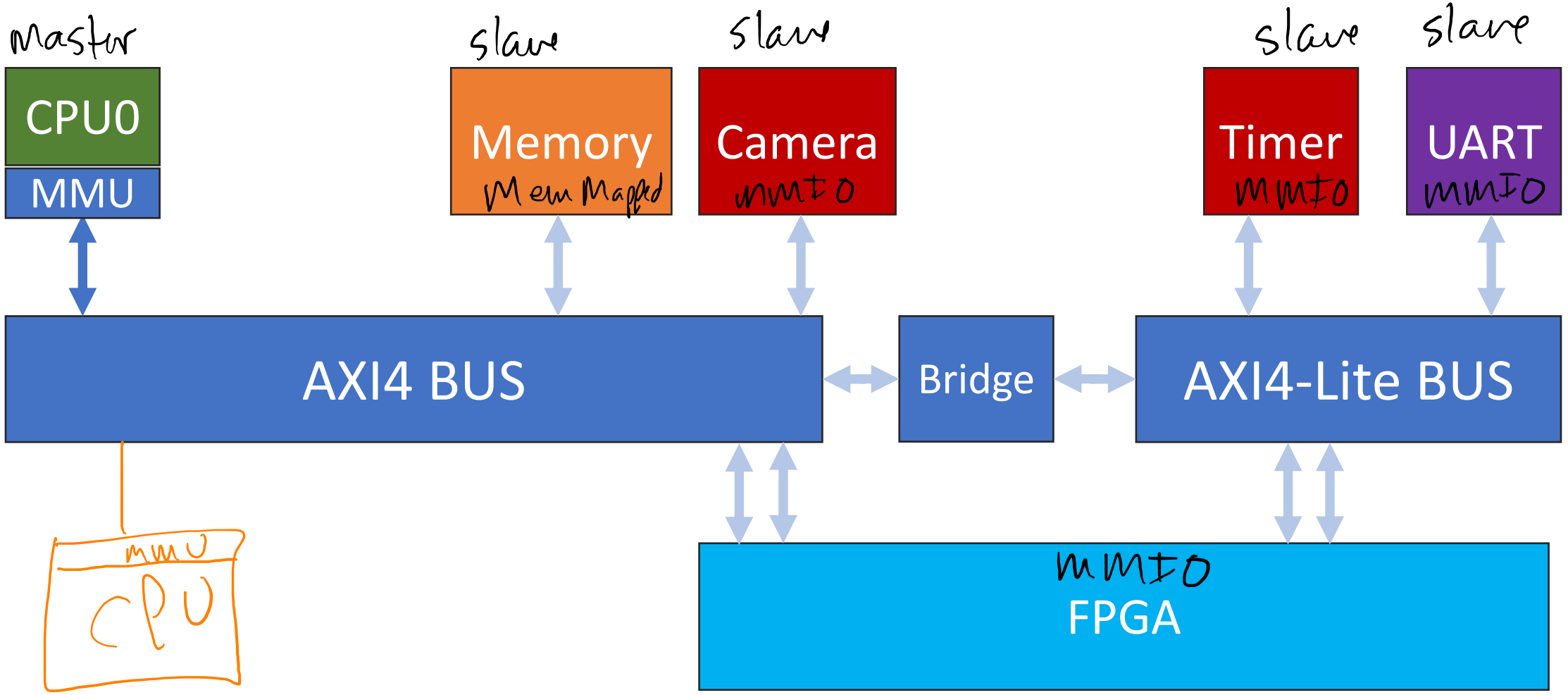
Complete EMA

```
#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; }

    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;
}
```

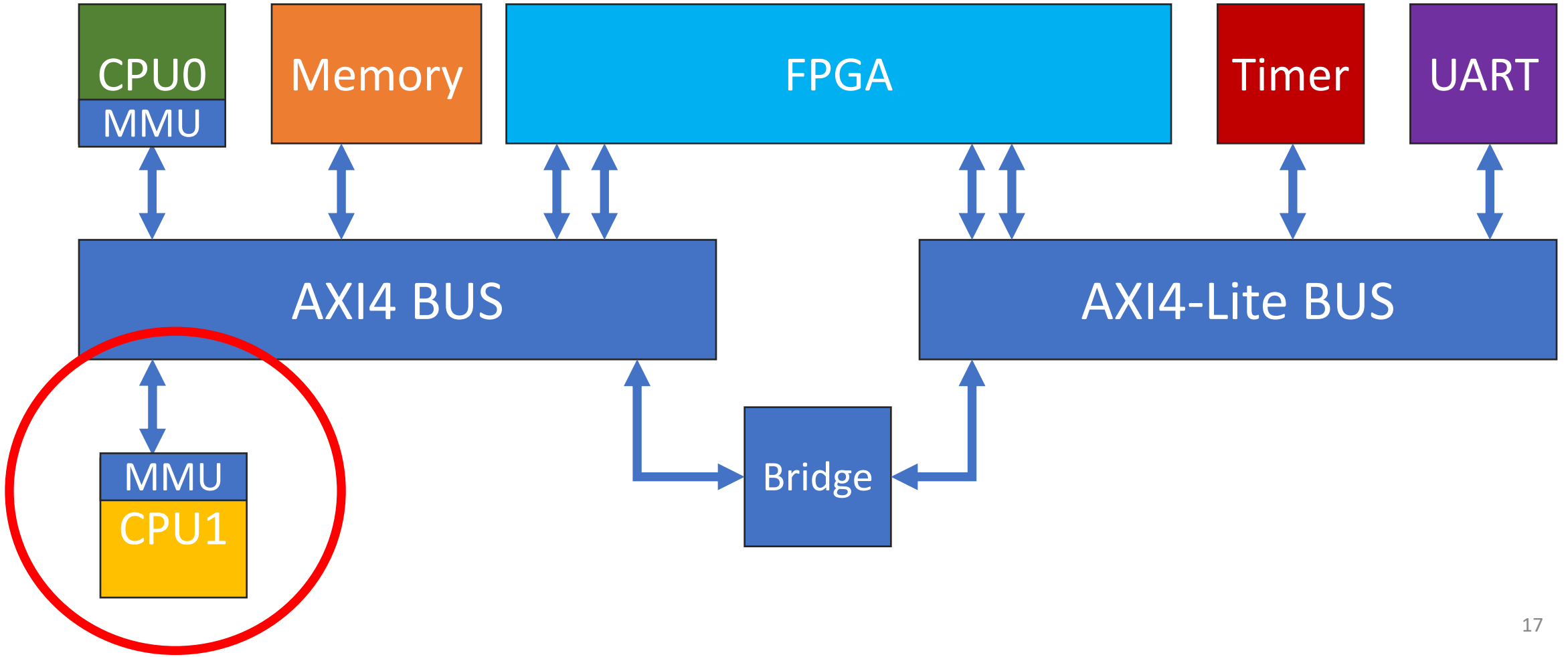
Demo Time

master: issue request
slave: fulfil request

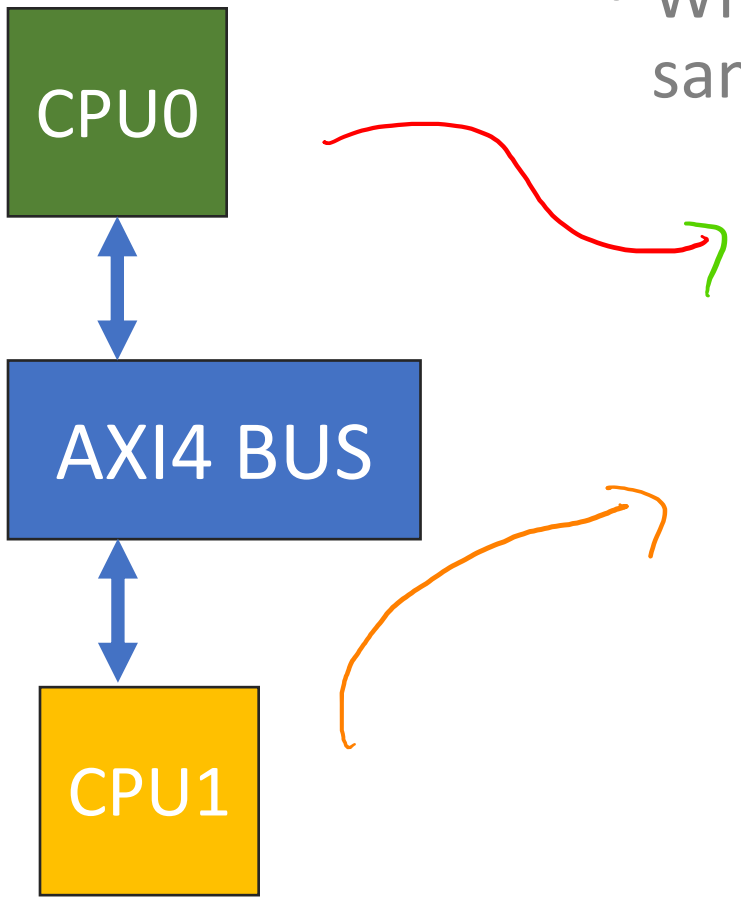
Which are AXI4 ”Masters”?



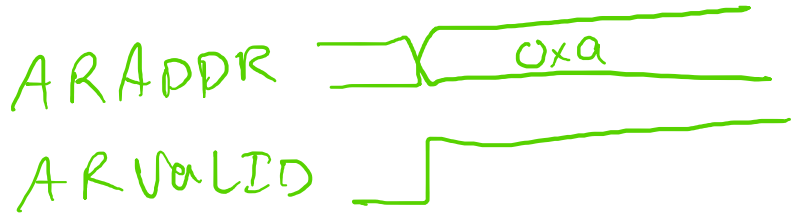
Multiple Masters

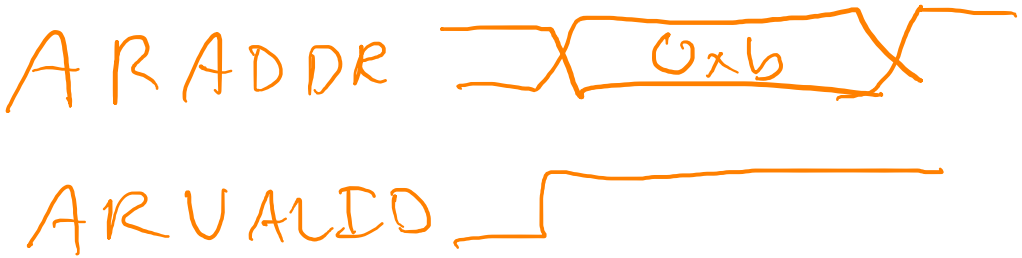


Multi-Master Buses

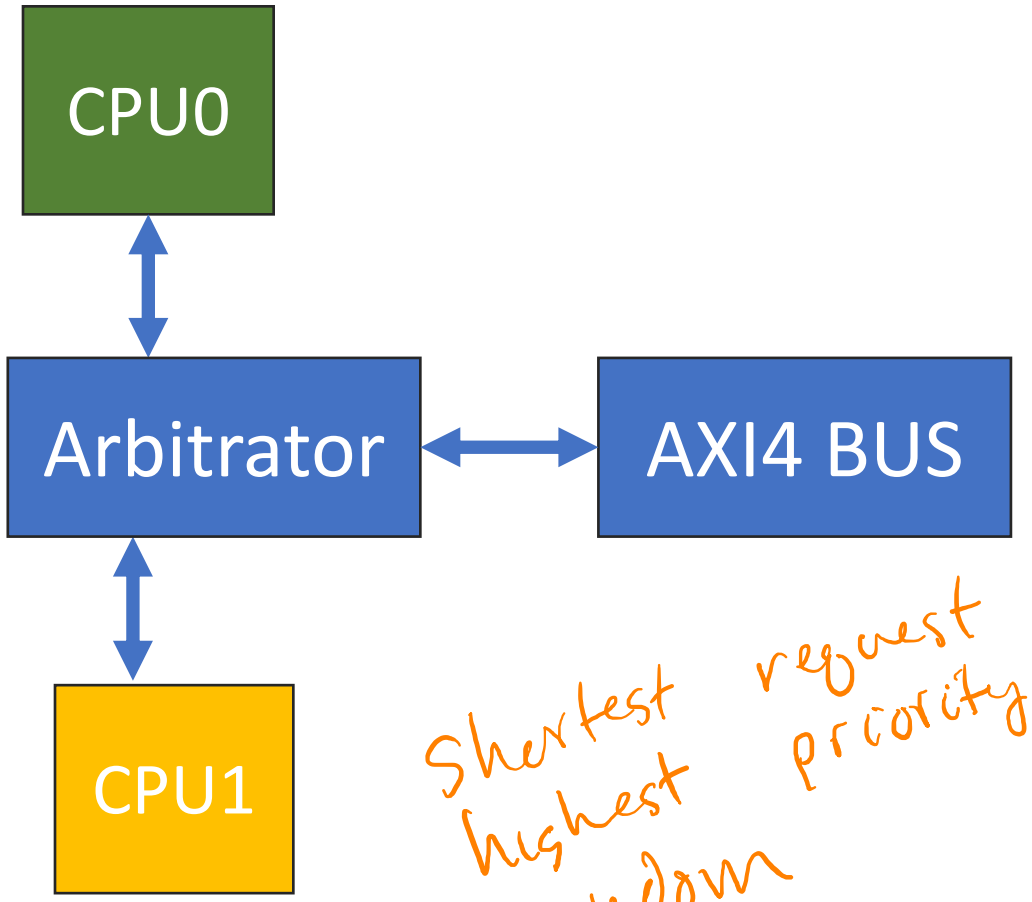


- What happens if both request a transaction at the same time?





An Arbitrator selects who gets to use the bus



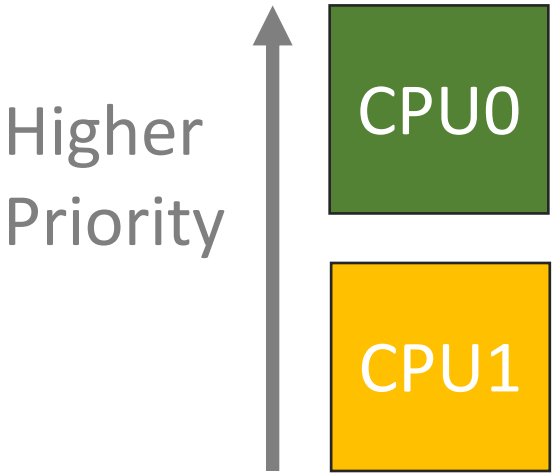
- What happens if both request a transaction at the same time?
- Arbitration: Pick a winner!
- What Arbitration scheme to use?

shortest
highest
random
request priority
round-robin

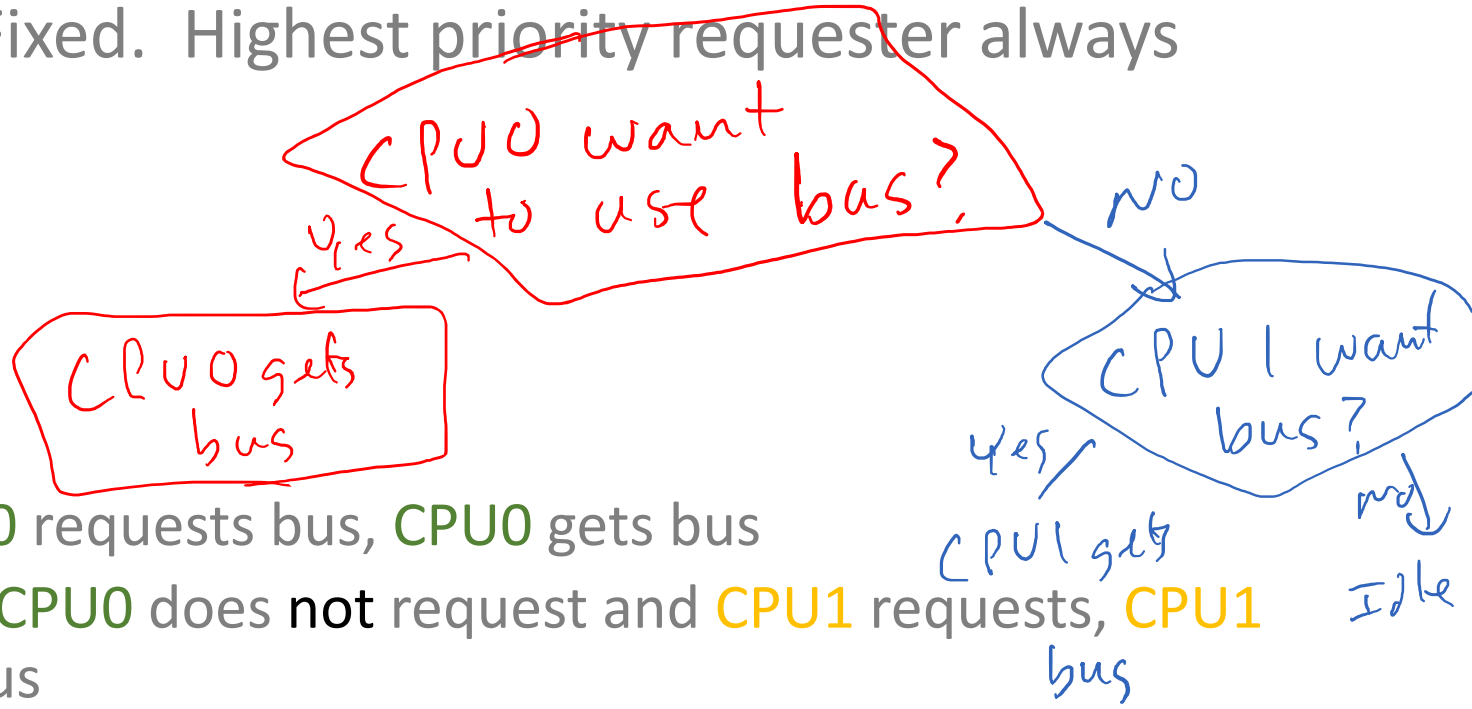
Arbitration Options

- Highest-Priority First
- Round Robin

Highest-Priority First

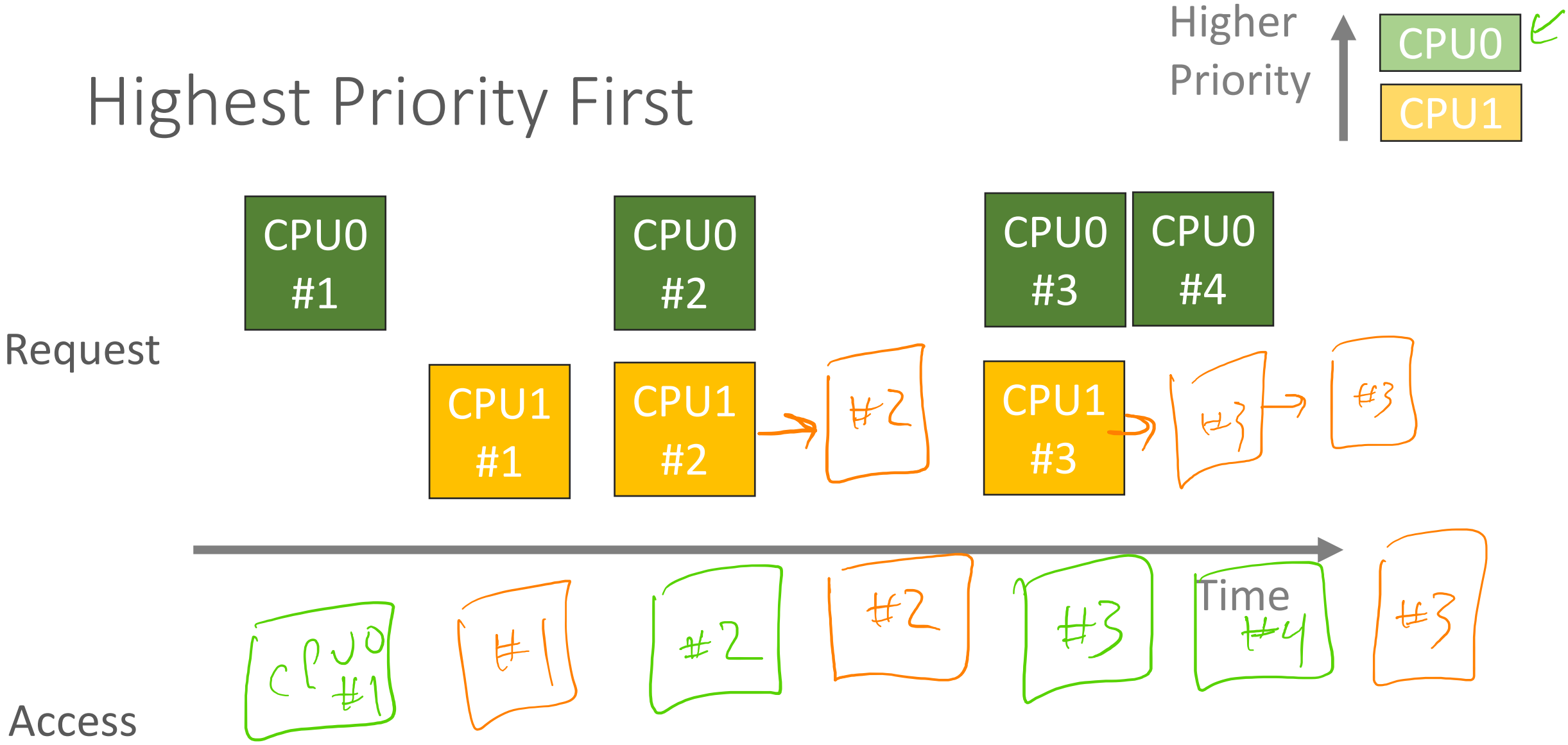


- Priority Fixed. Highest priority requester always wins.

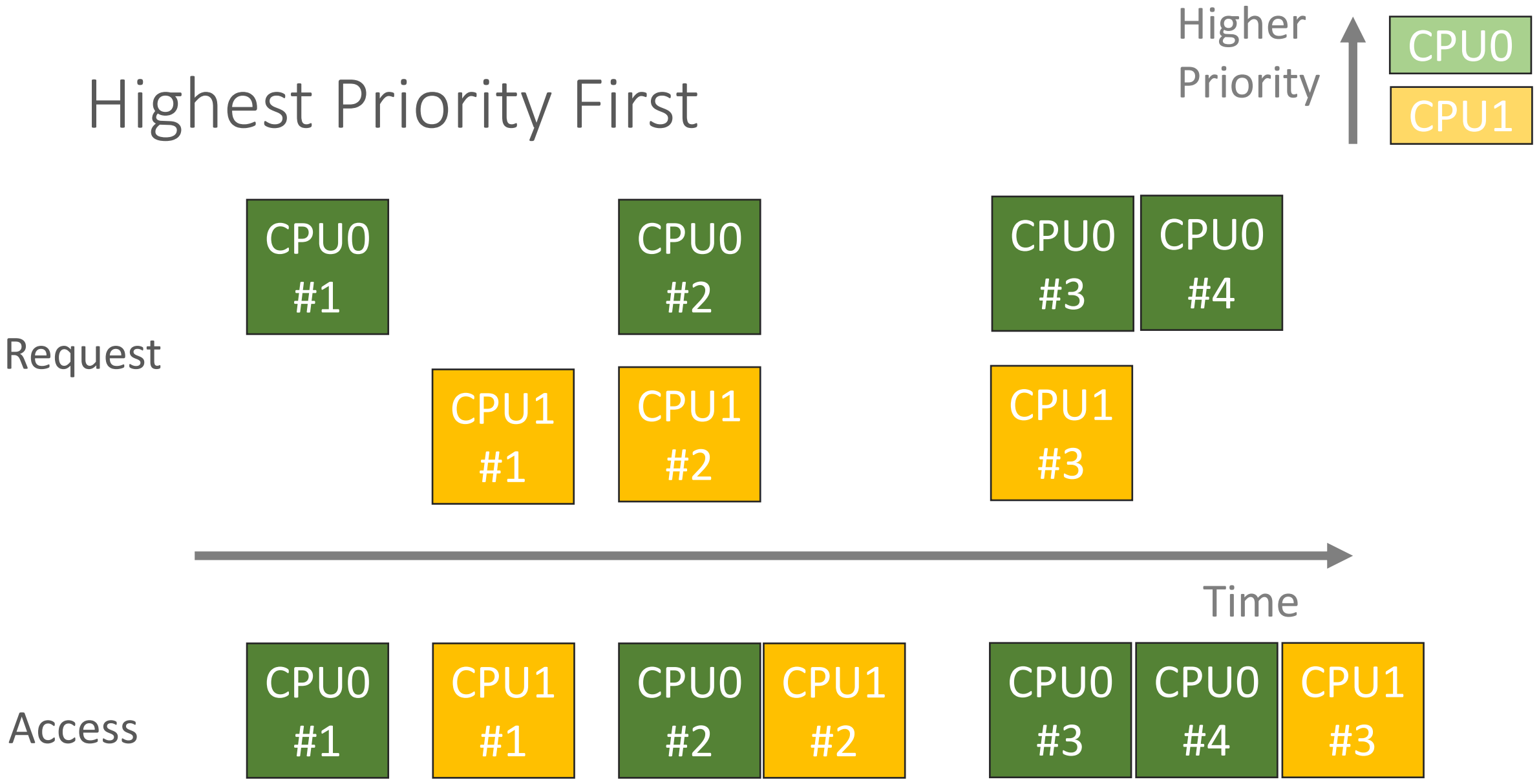


- Rules:
  - If CPU0 requests bus, CPU0 gets bus
  - Else if CPU0 does not request and CPU1 requests, CPU1 gets bus
  - Else bus idle

Highest Priority First



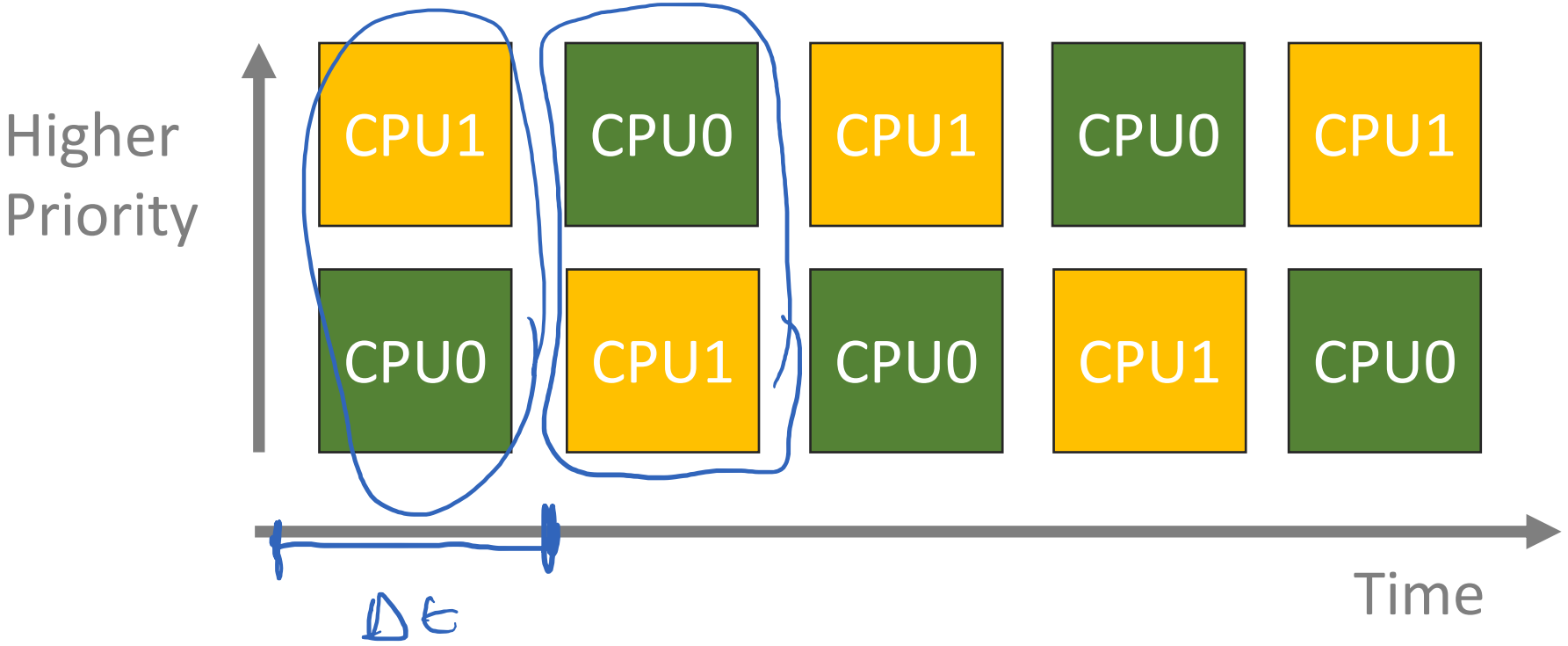
Highest Priority First



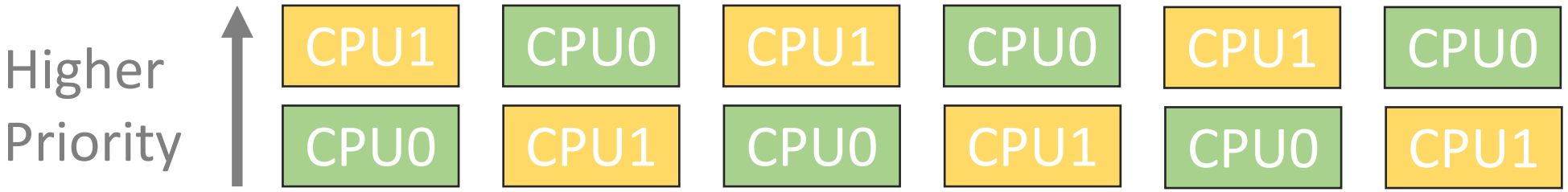
Arbitrator Circuit?

Round Robin

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



Round Robin

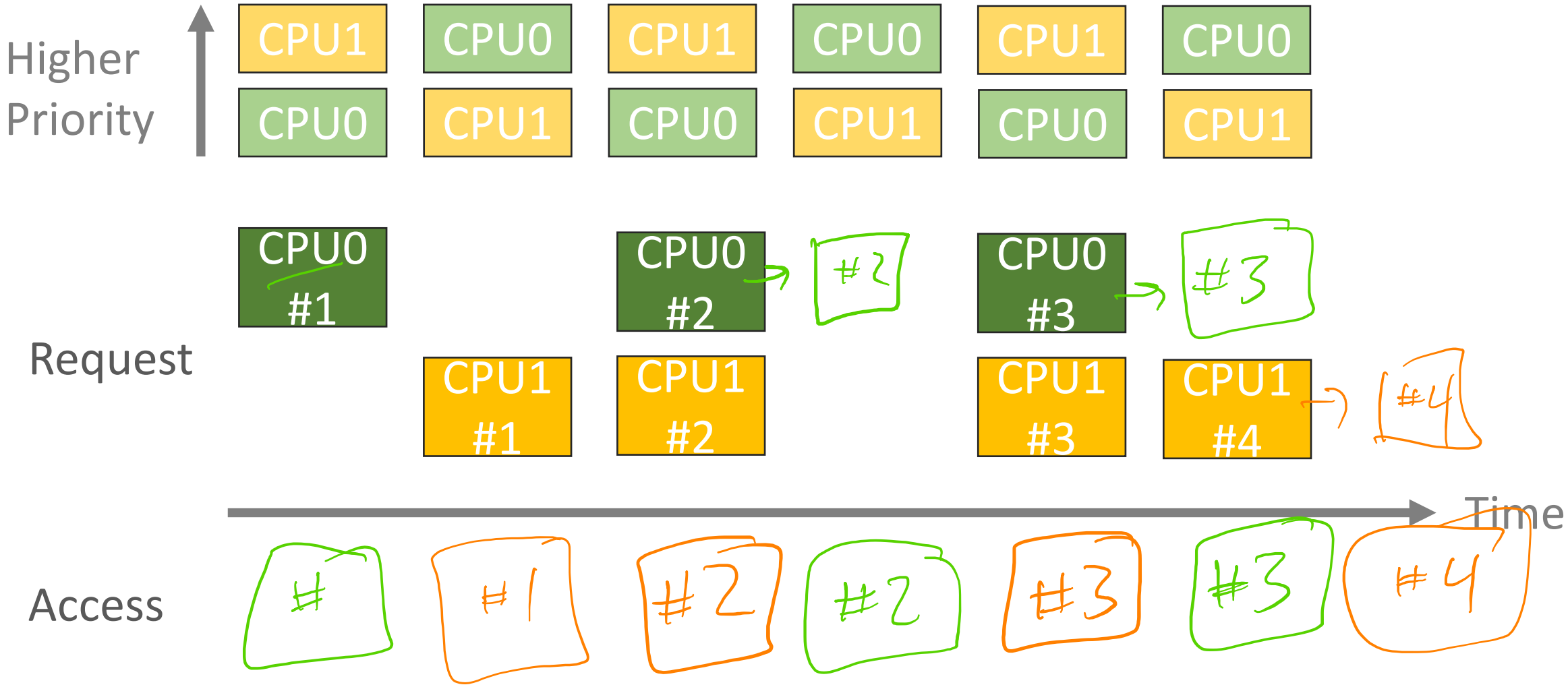


Request

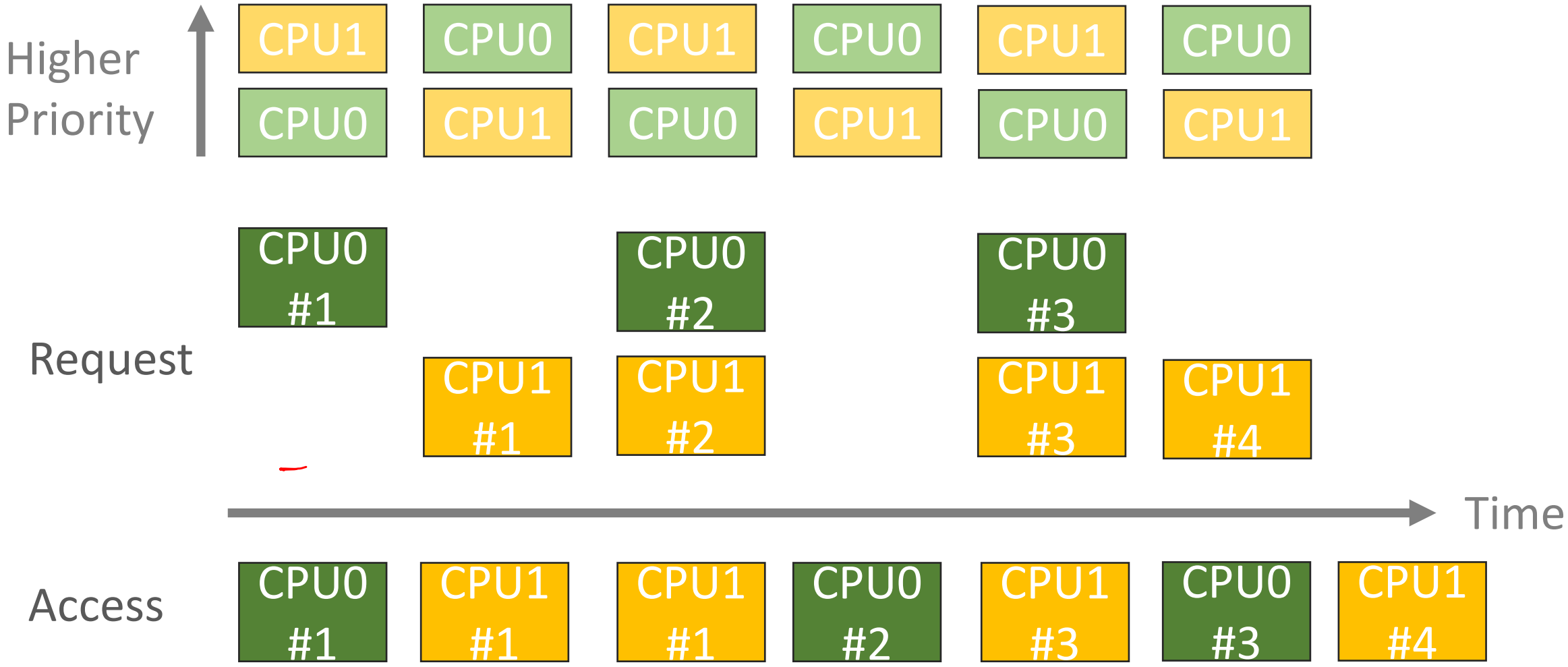
Time

Access

Round Robin

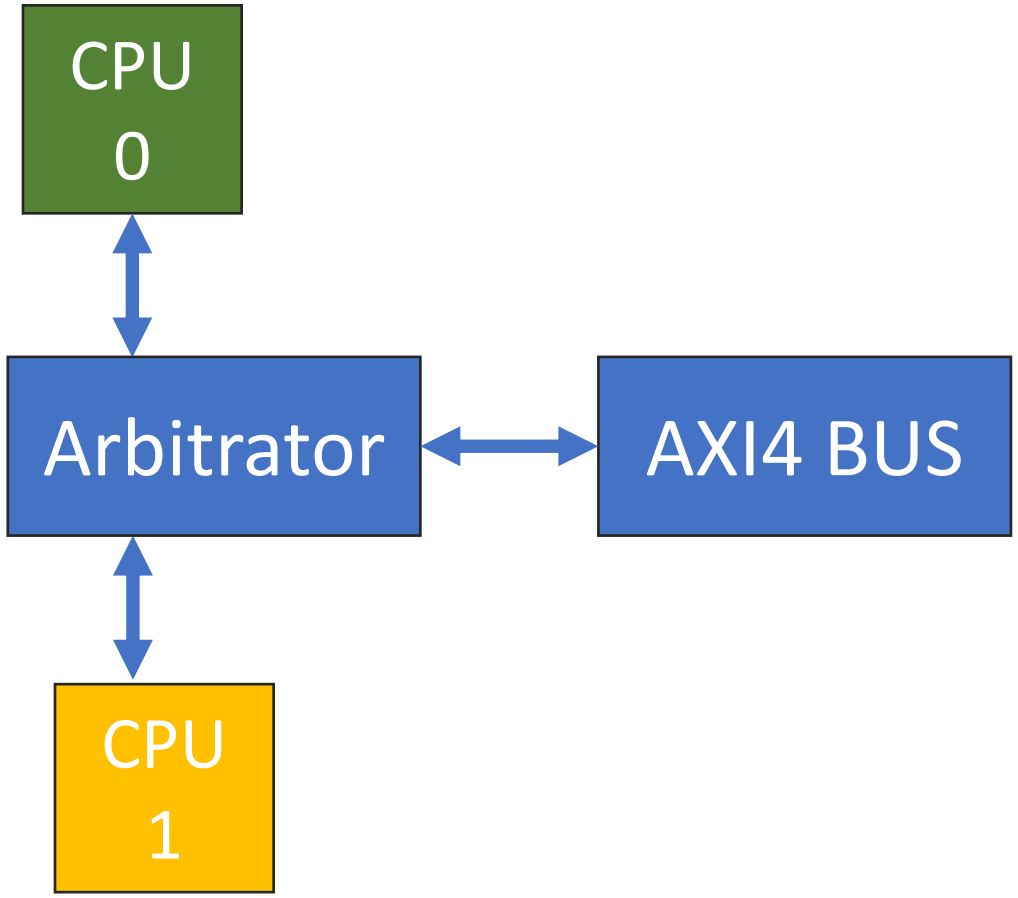

Higher Priority
CPU1
CPU0
CPU1
CPU0
CPU1
CPU0
CPU0
CPU1
CPU0
CPU1
CPU0
CPU1
Request
CPU0 #1
CPU0 #2
#2
CPU0 #3
#3
CPU1 #1
CPU1 #2
CPU1 #3
CPU1 #4
#4
Time
Access
#
#1
#2
#2
#3
#3
#4

Round Robin


Higher Priority
CPU1
CPU0
CPU1
CPU0
CPU1
CPU0
CPU0
CPU1
CPU0
CPU1
CPU0
CPU1
Request
CPU0 #1
CPU0 #2
CPU0 #3
CPU1 #1
CPU1 #2
CPU1 #3
CPU1 #4
Time
Access
CPU0 #1
CPU1 #1
CPU1 #1
CPU0 #2
CPU1 #3
CPU0 #3
CPU1 #4

Other Arbitration ideas?

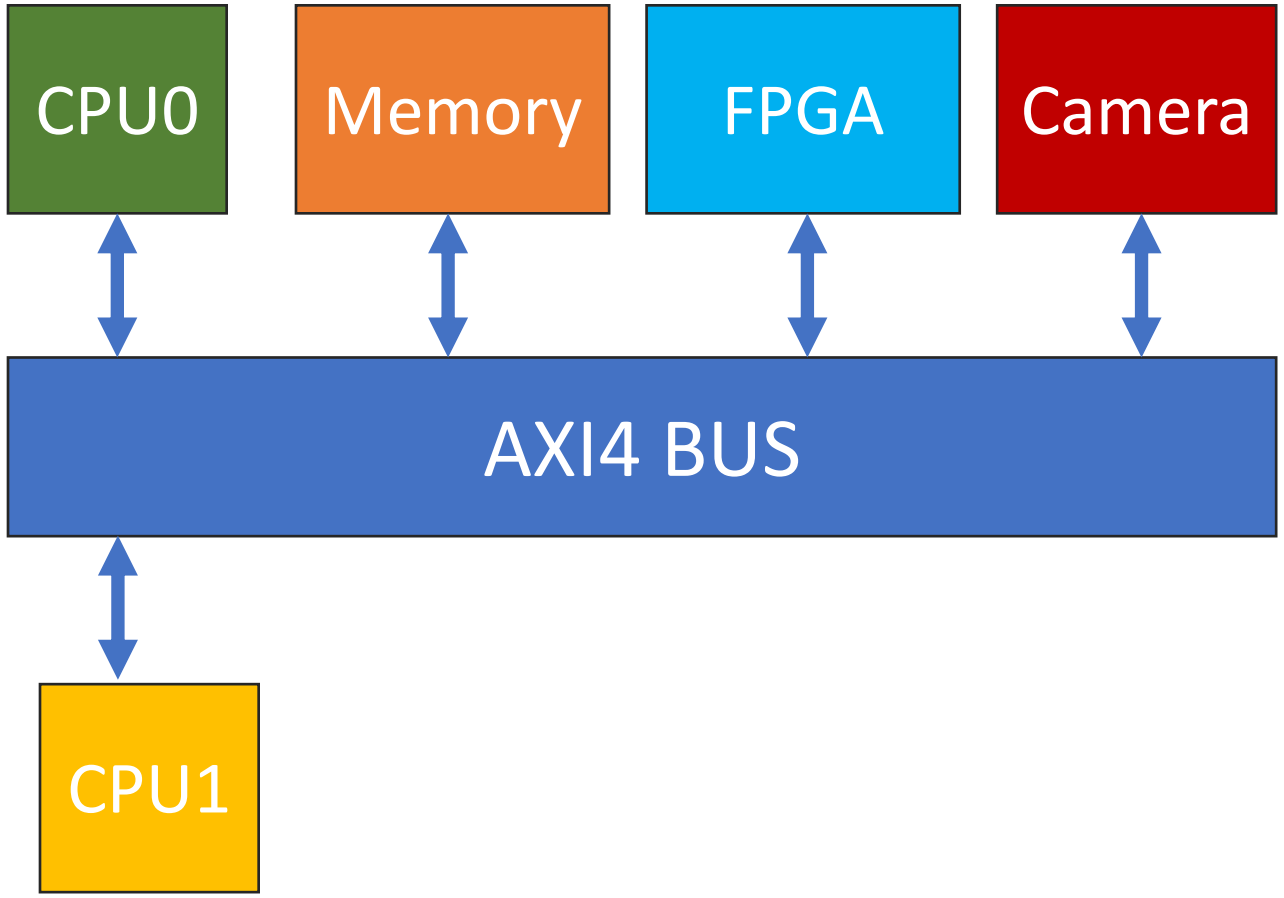
An Arbitrator selects who gets to use the bus



- 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

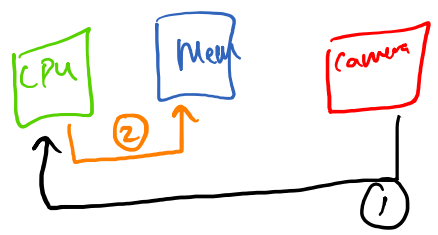
```
#define CAMERA_MMIO_ADDR 0x40000004
volatile uint32_t * camera =
        (uint32_t *)(CAMERA_MMIO_ADDR);
#define BUF_SIZE 1024;
uint32_t buf[BUF_SIZE];

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)
{
   register uint32_t reg;

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

        reg = *from;

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

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);
    }
}
```

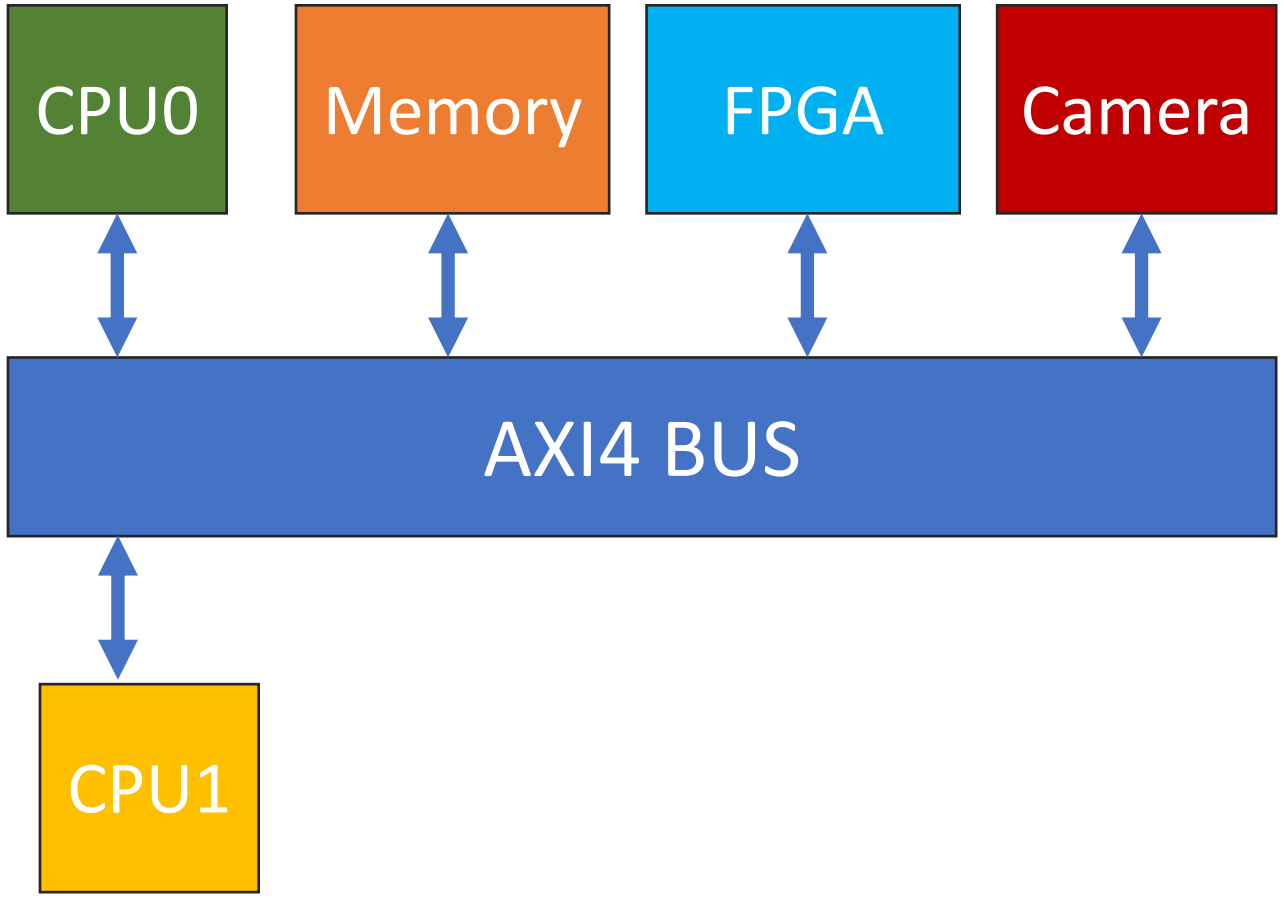
```
void copy_image (uint32_t * from,
                 uint32_t * to,
                 uint32_t size)
{
    register uint32_t reg;

    for (int i = 0; i < size; ++i){
        reg = *from;
        to[i] = reg;
    }
}
```



1 load
2 store

3 increment
4 branch

What bus transactions result?



What else can the CPU do while copying data?

CPU ⇒ 1 GHz → 1 Billion cycle / second

→ 1 Billion instructions / second

What else can the CPU do while copying data?

- CPU can do 1B instructions/second. (1GHz)
- 4 Instructions per 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

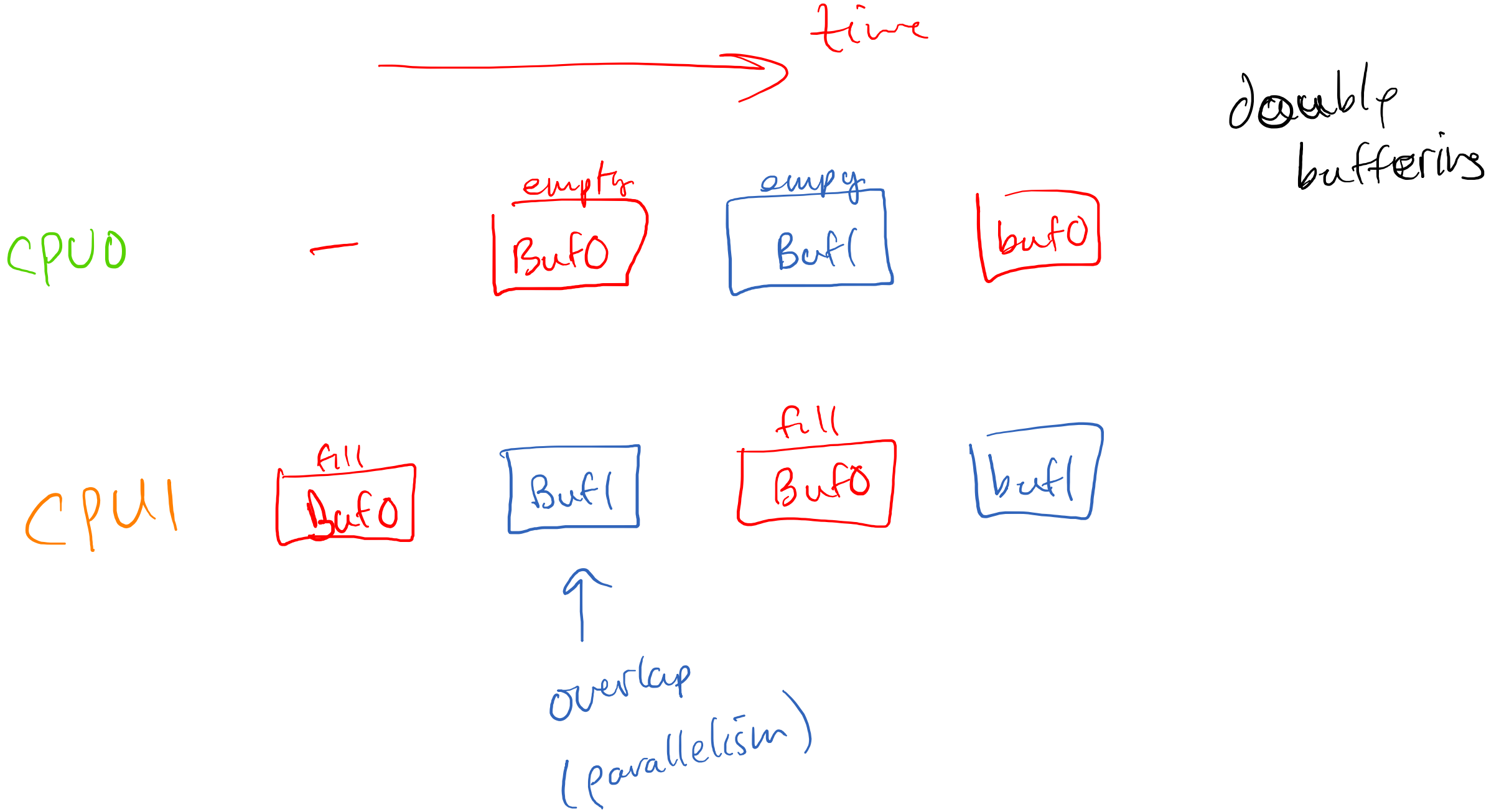
- 1Gbps Ethernet:
- 1 Gbps Receive + 1Gbps Transmit = 2 Gbps
- 2Gbps = 250MB/second

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

What if we do the copy on CPU1?

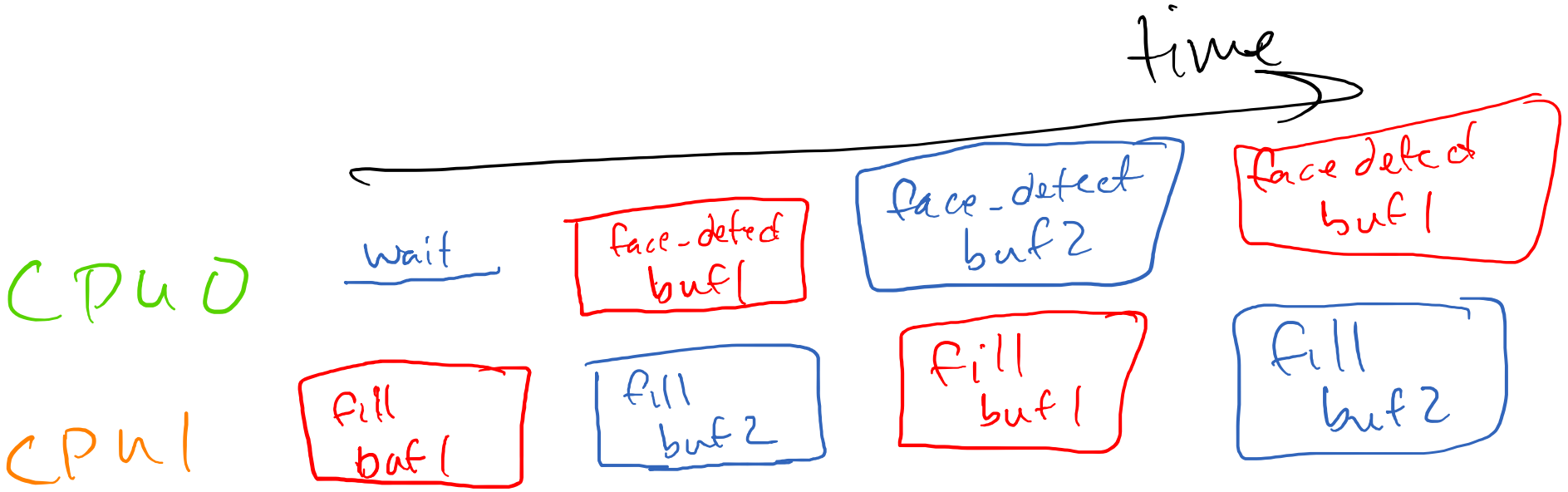
```
int main () {

    while (true){

        ask_cpu1_to_copy_image(camera, buf, BUF_SIZE);



        detect_face(buf);
    }
}
```

What if we do the copy on CPU1?


time
double bufferins
CPU0
empty
Buf0
empty
Buf1
buf0
CPU1
fill
Buf0
Buf1
fill
Buf0
buf1
overlap
(parallelism)

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
CPU 1
wait
face_defect buf1
face_defect buf2
face_defect buf1
fill buf1
fill buf2
fill buf1
fill buf2

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();
    }
}
```

Next Time:

Virt Linux → Python
Windows Vivado

- 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