PS -> pushed to Monday
demo Wed

13: DMA

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University

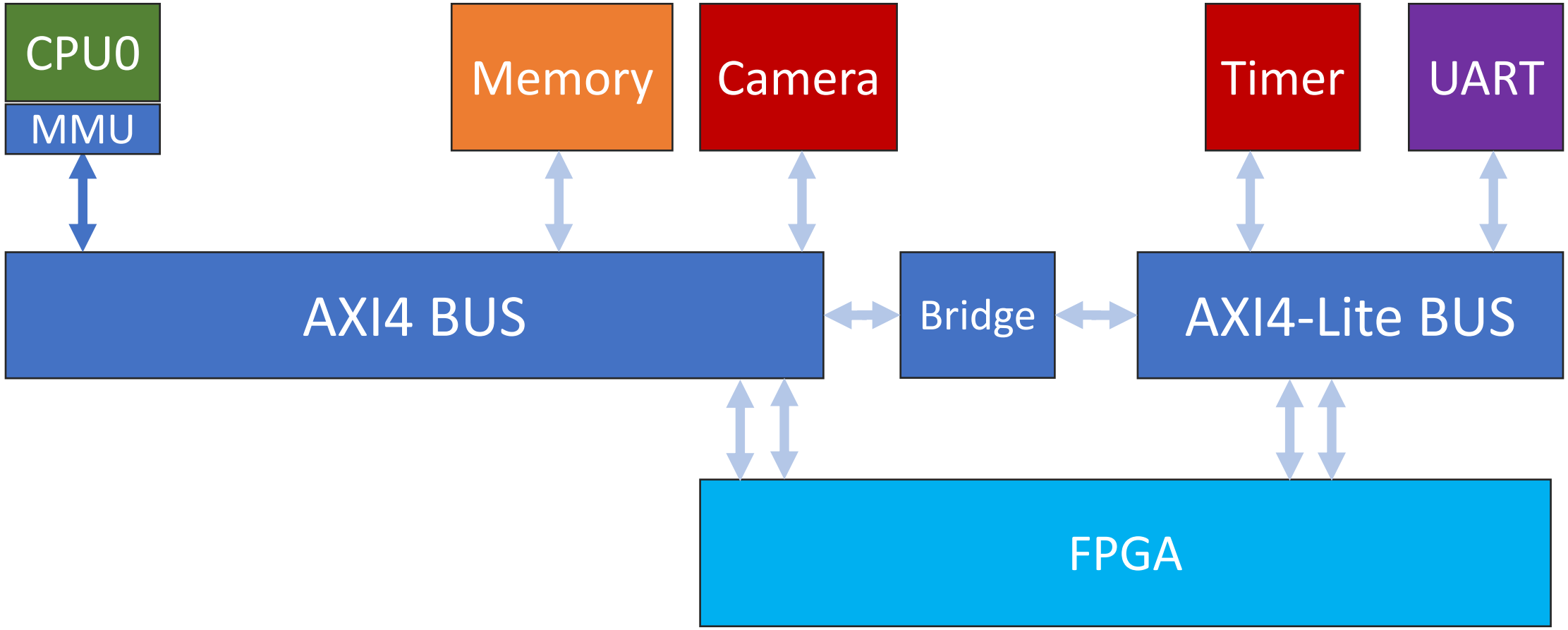
Announcements

- P5 is out! → extended

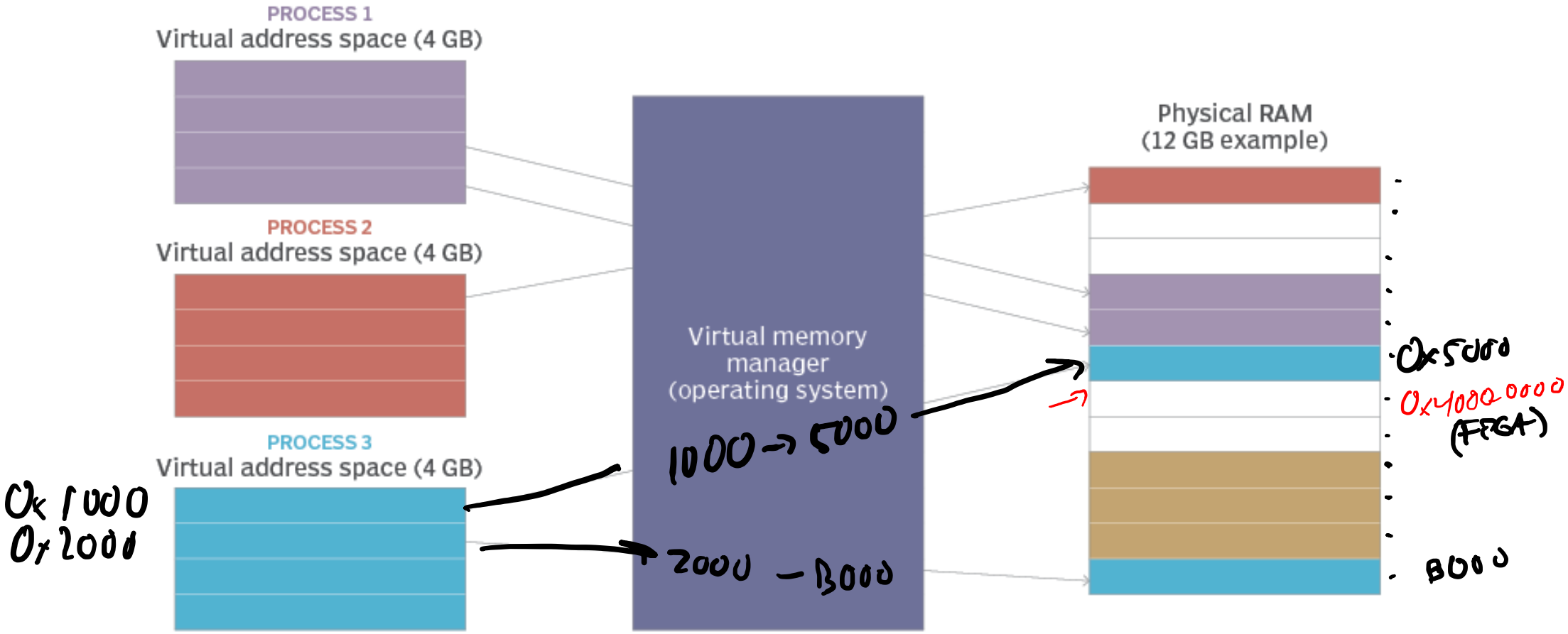
Exam Planning

11/01	Wednesday	18	Review	–
11/06	Monday	19	Exam	

Machine Model, V3: MMUs

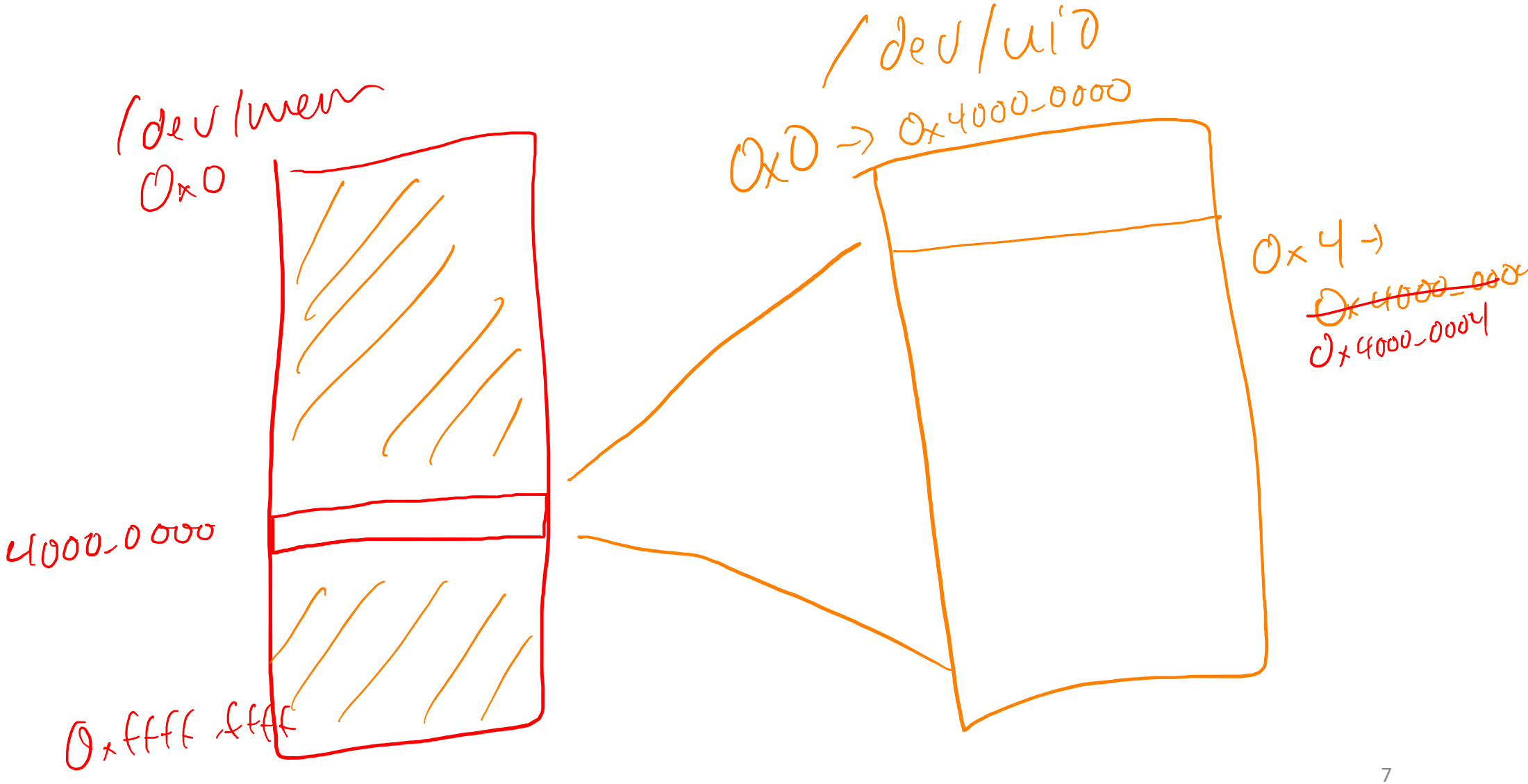


OS (Linux) mains full Virtual->Physical Mappings

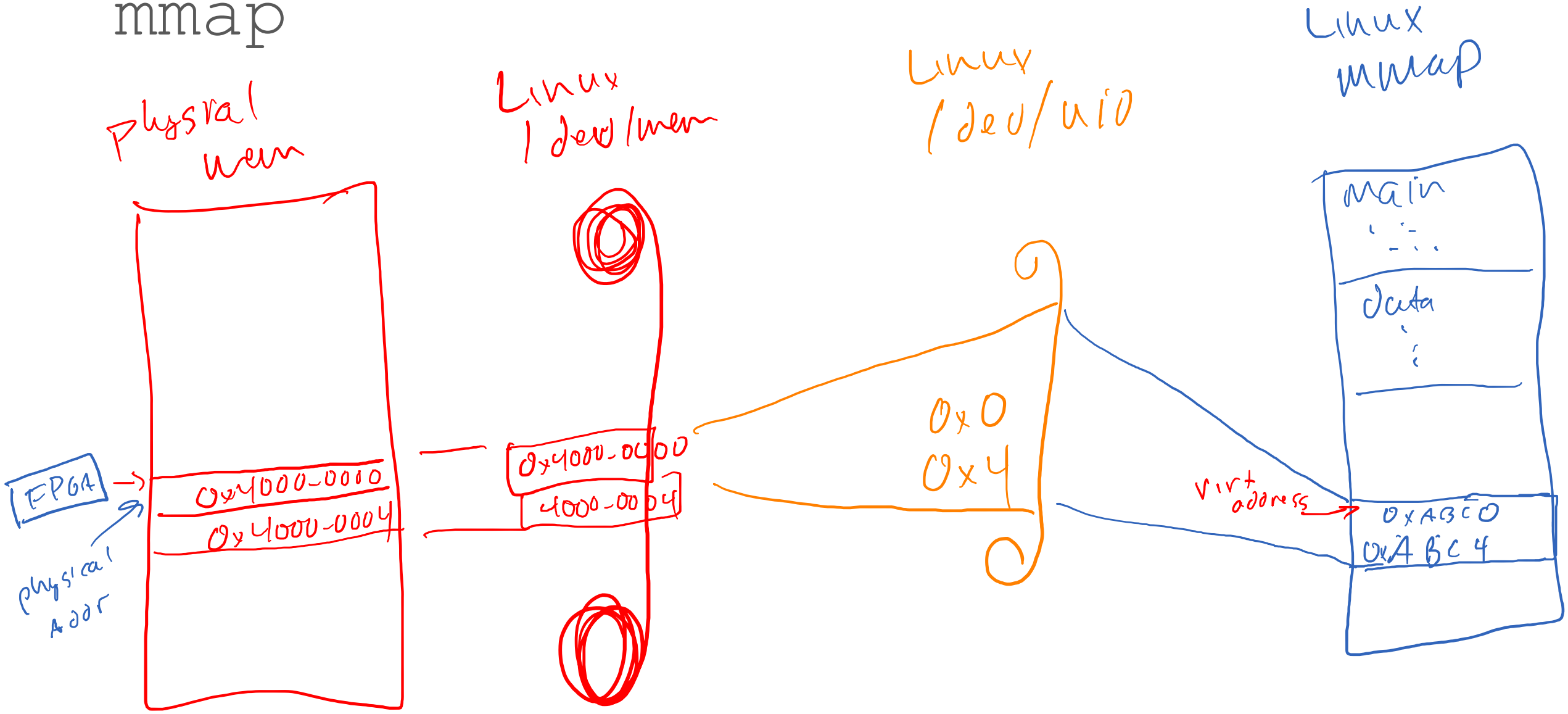


https://www.techtarget.com/searchstorage/definition/virtual-memory

/dev/uio


/dev/mem
0x0
4000_0000
0xffff_ffff
/dev/uio
0x0 -> 0x4000_0000
0x4 ->
~~0x4000_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
virt address
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; }
```

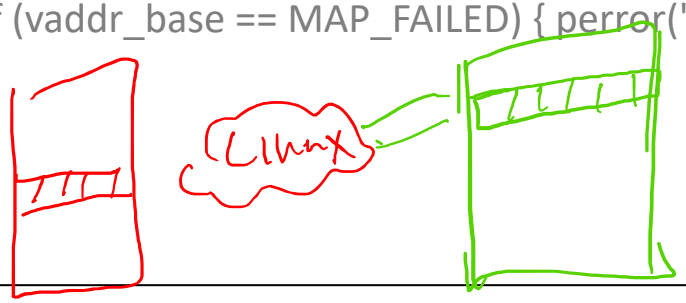
```
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    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);
```

4000 0000 w/ different name

```
    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

virtual address base

```
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

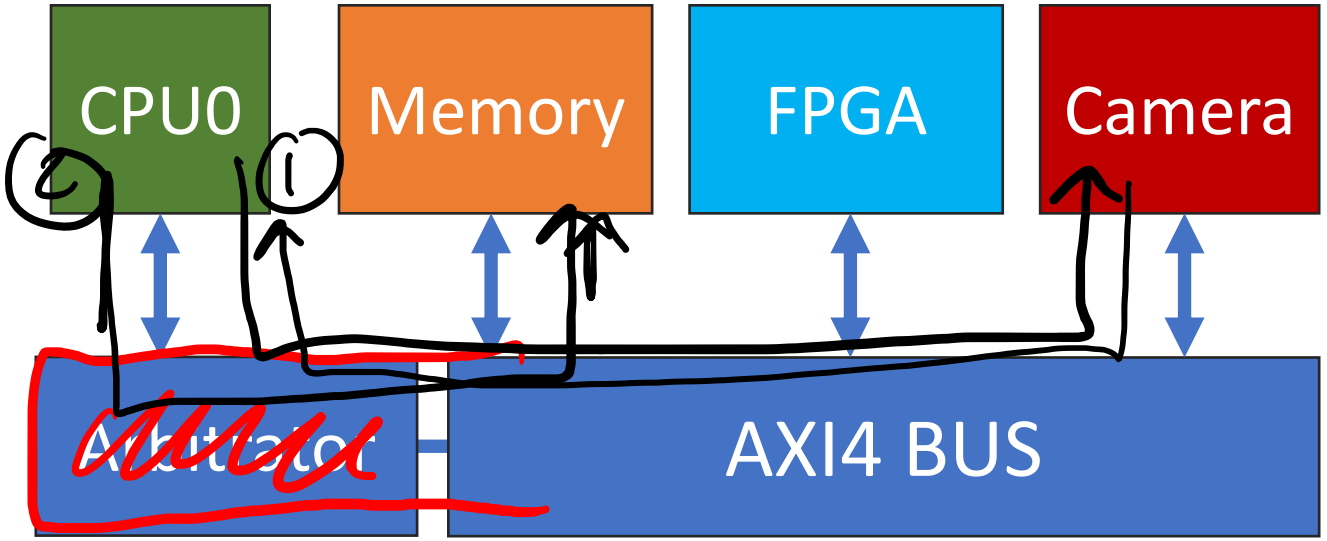
```
int main (){

    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;
	}
```

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



1. Load from Camera to CPU

2. Store to memory

A: The CPU copies data from Camera to Memory

i=2

x = i++; → x=2, i=3

x = ++i → x=3, i=3

include " fancy-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(;;){
    copy-image(camera, buf, BUF_SIZE);
    fancy-face-detect(buf)
}

}
```

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

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 () {
    //...
    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 else can the CPU do while copying data?

CPU → runs @ 1 GHz = f

$P = \frac{1}{1\,GHz} = \frac{1}{10^9\,Hz} = 10^{-9}\,s$

∴ 1 ns/op

= 10^9 operations / second

What else can the CPU do while copying data?

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

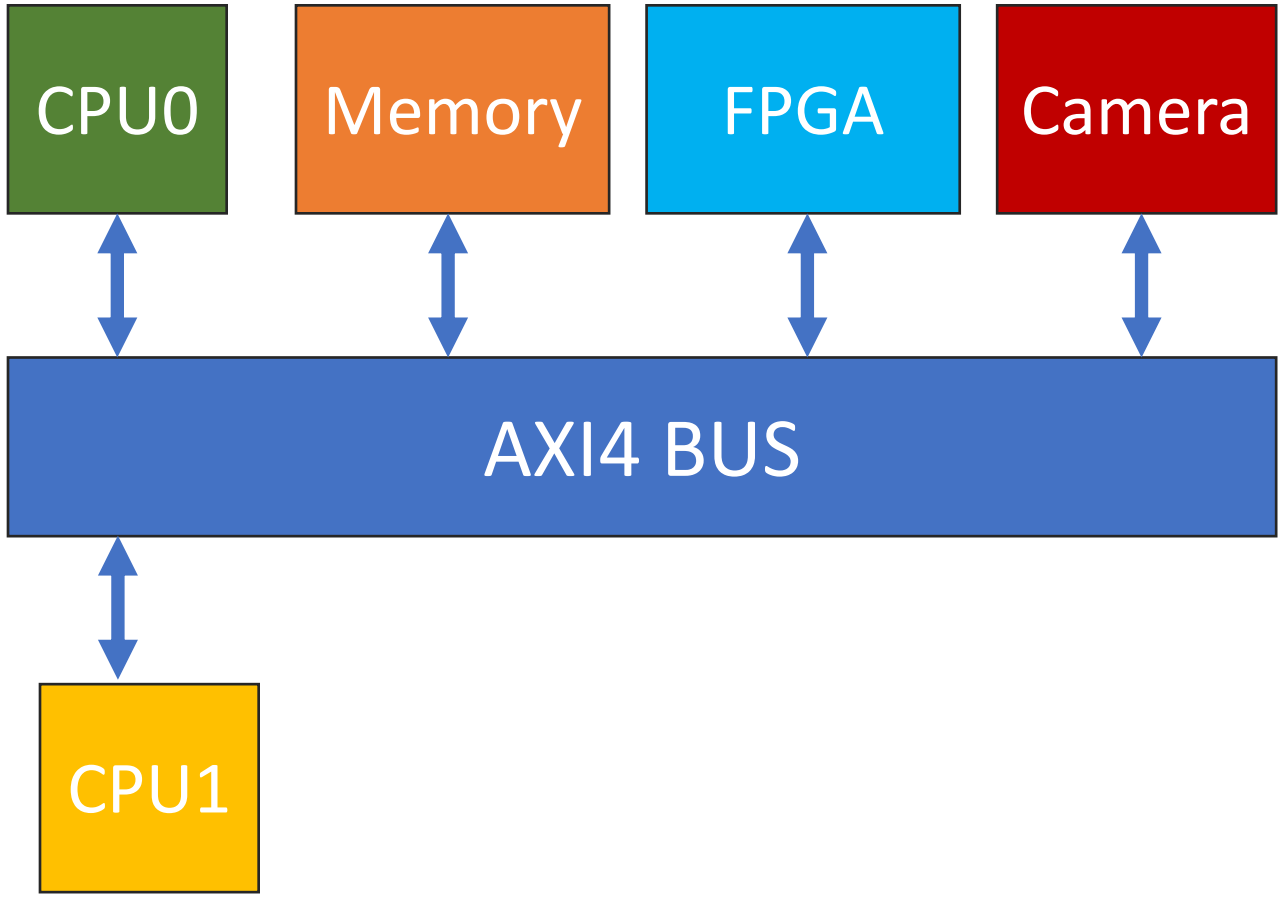
What about Ethernet?

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

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

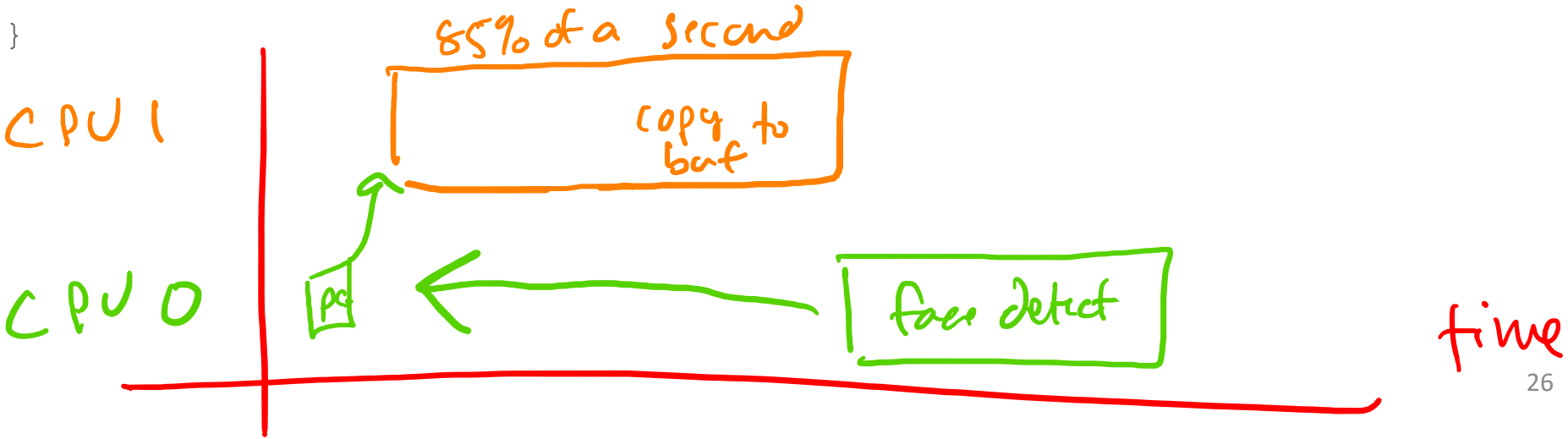
- **Nothing. ~100% of CPU required?**

What if we do the copy on a new CPU, CPU1?



What if we do the copy on a new CPU, CPU1?

```
int main () {

    while (true){

        ask_cpu1_to_copy_image(camera, buf, BUF_SIZE);


 CPU 0 detect_face(buf);
    }
}
```



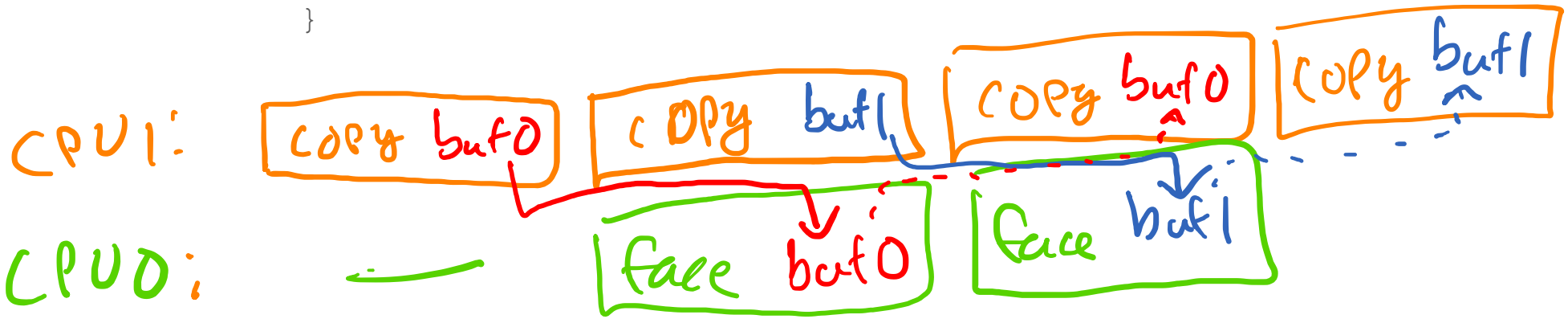
What if we do the copy on CPU1?

buf0: Red
buf1: Blue
CPU0: Green
CPU1: Orange

int [buf] buf0;
int [buf] buf1;

```
int main () {

    while (true){

        ask_cpu1_to_copy_image(camera, ~~buf~~ buf0 BUF_SIZE);



        detect_face(buf);
    }
}
```

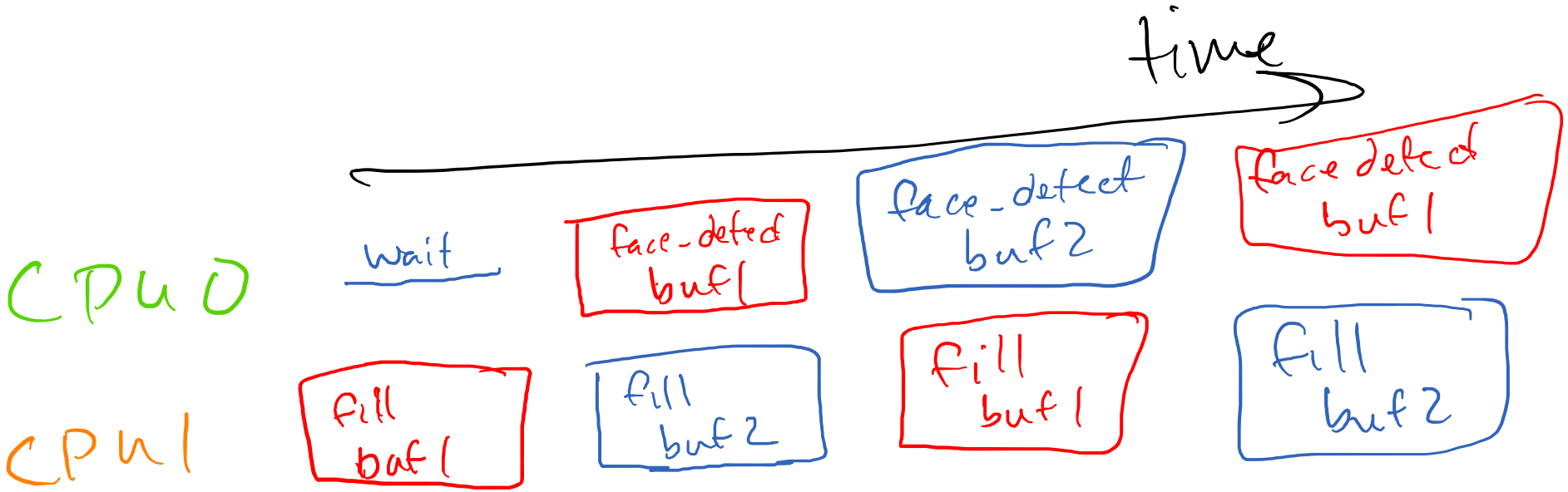
double buffering



Double Buffering explained.

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-detect buf1
face-detect buf2
face detect 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

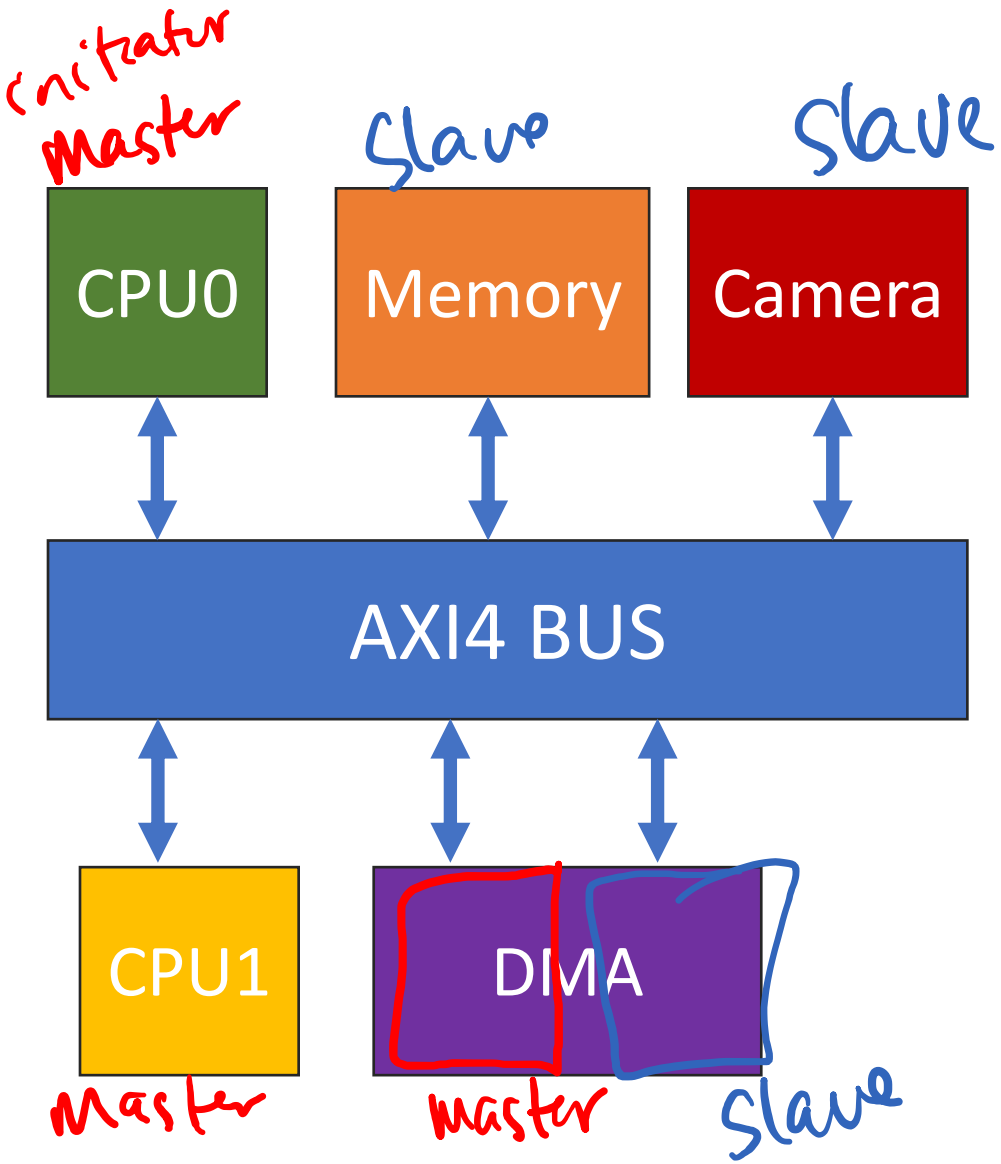
- **A mini-CPU that (only) does copy for you:**

```
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;
    }
}
```

Copy with DMA

```
int main () {

    dma_start_copy (camera, buf1, BUF_SIZE);
    dma_wait_for_done();

    while (true){
        dma_start_copy (camera, buf2, BUF_SIZE);
        detect_face(buf1);
        dma_wait_for_done();

        dma_start_copy (camera, buf1, BUF_SIZE);
        detect_face(buf2);
        dma_wait_for_done();
    }
}
```

DMA has 2 interfaces


initiator
Master
Slave
Slave
CPU0
Memory
Camera
AXI4 BUS
CPU1
DMA
Master
Master
Slave

- Interface 1: Copy Memory
  - Data-Intensive Interface
  - AXI4 Master
  - Initiates Loads / Stores
- Interface 2: Tell DMA <u>what</u> to copy
  - Control Interface
  - AXI4 Slave
  - Responds to Loads/Stores

What's needed to do this in Hardware?

start

```
void dma_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;
    }
}
```

Done

Hardware Needs:

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

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

        reg = *from; //load

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

- AXI4 Master Interface
  - Actual Loads + Stores

- AXI4 Slave Interface
- 5 MMIO registers
  - Control (Start)
  - Status (Done)
  - Source (From)
  - Destination (To)
  - Size (in Bytes)

MyDMA MMIO Interface

- 0x0400: Control Register
- 0x0404: Status Register
- 0x0408: Source Address
- 0x040C: Destination Address
- 0x0410: Transfer Size in Bytes

MMIO Control Register

0x4000

MMIO Control Register

Control - 0x0400

31-1 Reserved	0 Start

MMIO Status Register

Control - 0x0400

Read/Write

31-1 Reserved	0 Start

Status - 0x0404

Read - Only

31-1 Reserved	0 Done

=1 DMA done

=0 = DMA not done

1st

MMIO Data Registers

store ABCD to 0x0408

Source - 0x0408

ABCD	31-0 DMA Source Address

store 1234 to

Destination - 0x040C

1234	31-0 DMA Destination Address

store 0x8

Size - 0x0410

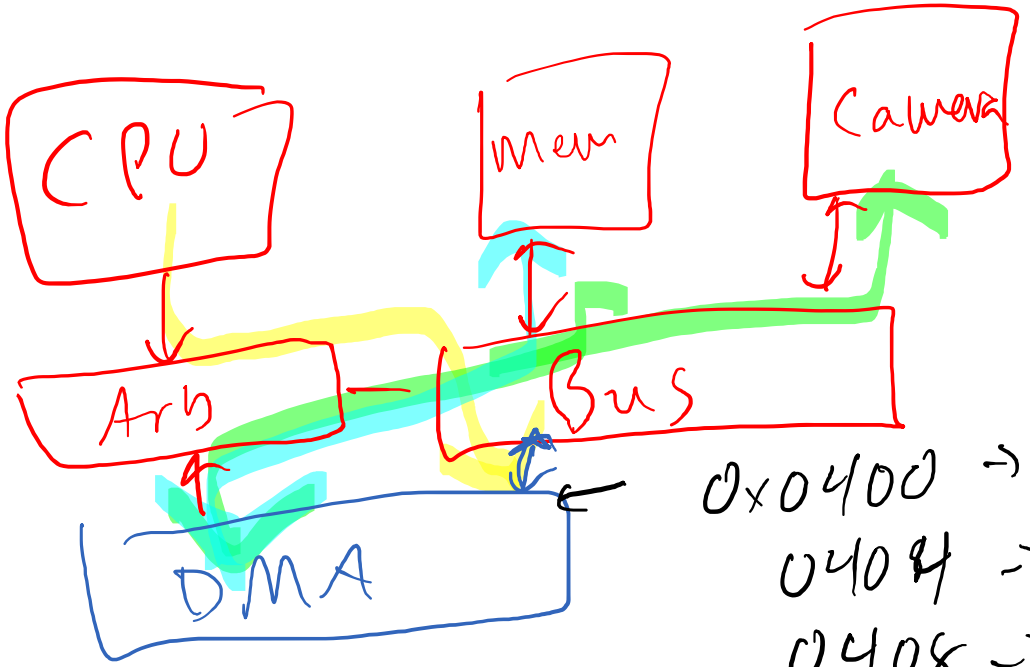
31-16 Reserved	0x8 15-0 DMA Transfer Size (in Bytes)

Stop

All MMIO Registers

Control - 0x0400	31-1 Reserved	0 Start
Status - 0x0404	31-1 Reserved	0 Done
Source - 0x0408	31-0 DMA Source Address	
Destination - 0x040C	31-1 DMA Destination Address	
Size - 0x0410	31-16 Reserved	15-0 DMA Transfer Size (in Bytes)

MyDMA Interface



0x0400 → Ctrl
0404 → Status
0408 → Source (from)
040C → Dest (to)
0410 → Size

MyDMA Internals

MyDMA Internals

- IDLE: Status[Done]=1, wait for Control[Start]
- START: Status[Done] = 0, i = 0;
- LOAD: tmp = [Source+i]
- STORE: Dest+i = tmp

Does the AXI4 Full Interface have an address?

Does the AXI4 Full Interface have an MMIO Address?

- Is pretending to be memory, or a CPU?
- Does a CPU have a memory address?
- No.
- MMIO is for SLAVE interfaces.

Using DMA from the CPU:

0x0400: Control Register
0x0404: Status Register
0x0408: Source Address
0x040C: Destination Address
0x0410: Transfer Size in Bytes

```
void dma_copy ( uint32_t * source,
                uint32_t * dest,
                uint32_t size){
```

~~register uint32_t reg;~~
~~for (int i = 0; i < size; ++i){~~
~~reg = *source; //load~~
~~dest[i] = reg; //store~~
~~}~~

```
    //code me!
}
```

Using DMA from the CPU:

0x0400: Control Register
0x0404: Status Register
0x0408: Source Address
0x040C: Destination Address
0x0410: Transfer Size in Bytes

```
void dma_copy ( uint32_t * source,
                uint32_t * dest,
                uint32_t size){

     *((volatile uint32_t *)(0x0408))= source;
     *((volatile uint32_t *)(0x040C))= dest;
     *((volatile uint32_t *)(0x0410))= size;
     *((volatile uint32_t *)(0x0400))= 0x1; //start

     //spin until copy done
     while( *((volatile uint32_t *)(0x0404)) != 0x1){;}
}
```

Using DMA from the CPU:

0x0400: Control Register
0x0404: Status Register
0x0408: Source Address
0x040C: Destination Address
0x0410: Transfer Size in Bytes

```
void dma_start_copy (   uint32_t * source,
                        uint32_t * dest,
                        uint32_t size){

        *((volatile uint32_t *)(0x0408))= source;
        *((volatile uint32_t *)(0x040C))= dest;
        *((volatile uint32_t *)(0x0410))= size;
        *((volatile uint32_t *)(0x0400))= 0x1; //start
}

void dma_wait_for_done(){
        //spin until copy done?
        while( *((uint32_t)(0x0404)) != 0x1){;}
}
```

`volatile` keyword omitted!

Using DMA from C:

```
int main () {

    dma_start_copy (camera, buf1, BUF_SIZE);
    dma_wait_for_done();

    while (true){
        dma_start_copy (camera, buf2, BUF_SIZE);
        detect_face(buf1);
        dma_wait_for_done();

        dma_start_copy (camera, buf1, BUF_SIZE);
        detect_face(buf2);
        dma_wait_for_done();
    }
}
```

11: Multi-Master Buses

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University