11: Direct Memory Access (DMA)

DMA

Engr 315: Hardware / Software Codesign

Andrew Lukefahr

Indiana University

Announcements

→ next yr:
rename
DMA popcount

- P3 demos due Friday

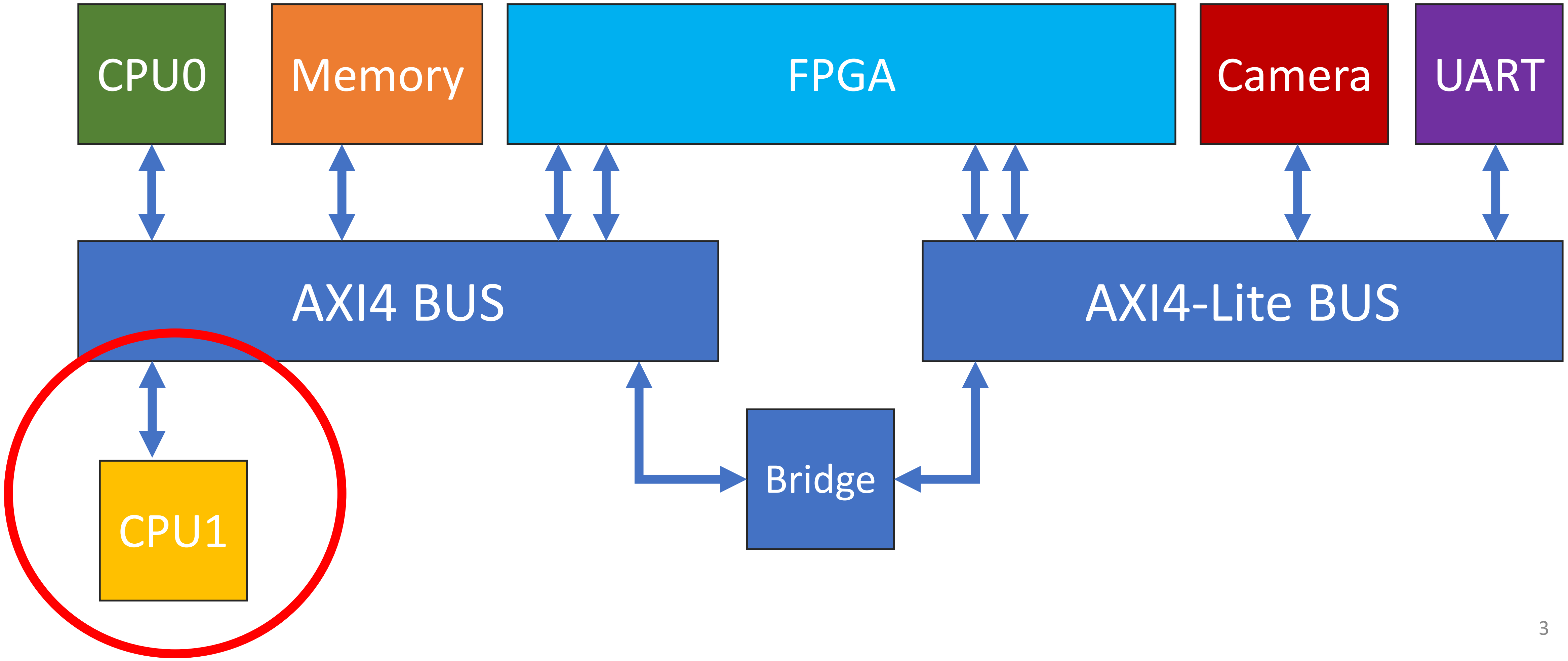
- P4 is out
  - Hopefully ~~Expect some~~ no revisions
  - Bitstream / hwh files added ↙
  - Password: 'iuxilinx' ←

user: xilinx
passwd: iuxilinx

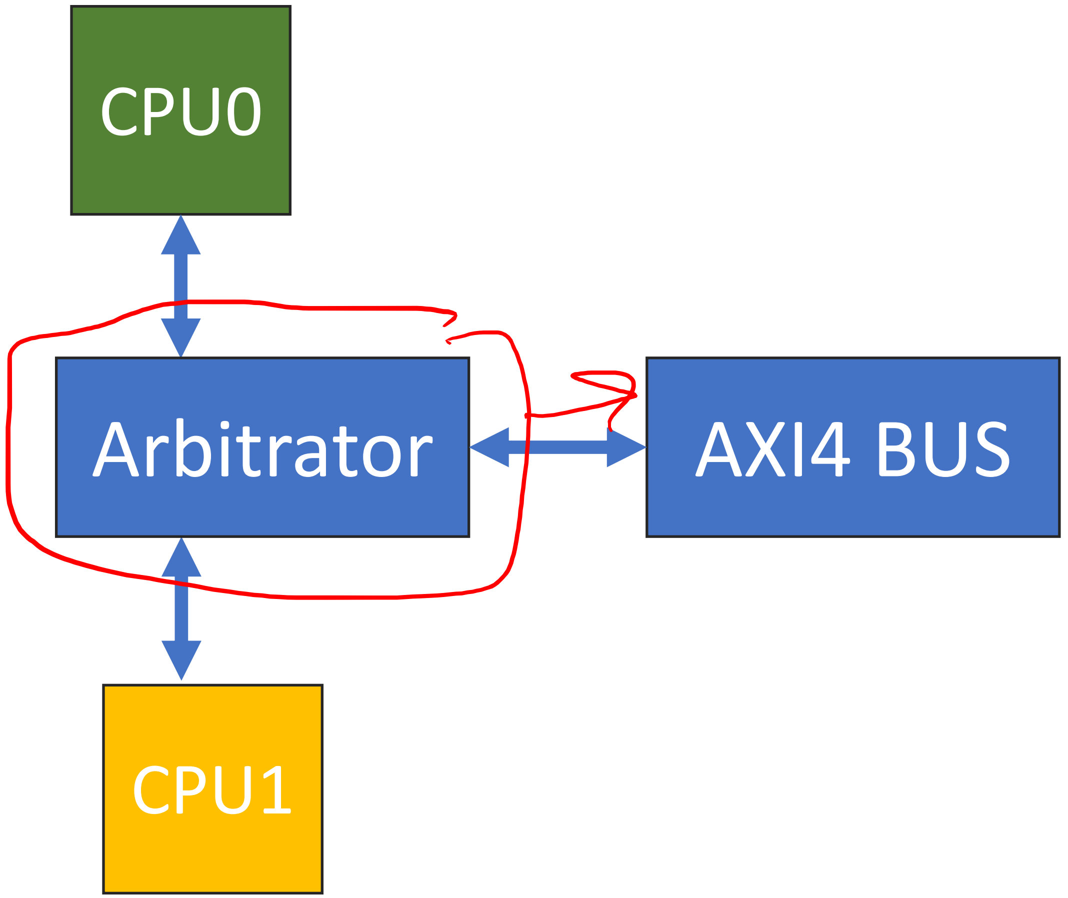
- P5 is out
  ↳ DMA (Verilog)

P6 → DMA (C)

Review: Multi-Master Buses



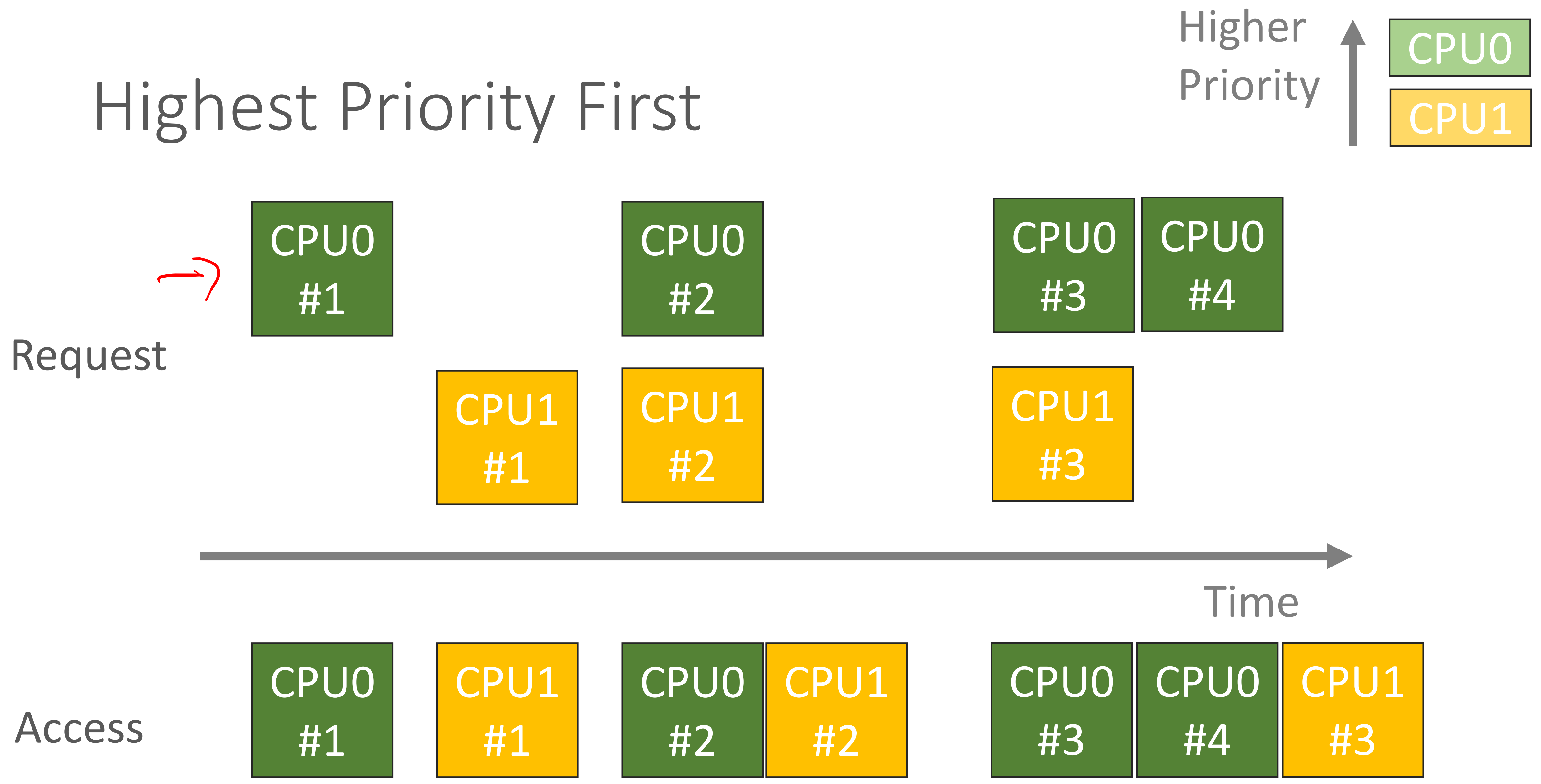
An Arbitrator selects who gets to use the bus



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

- **Arbitration:**
    - Fixed-Priority $\rightarrow$ fast
    - Round Robin $\rightarrow$ fair
    - Many more…

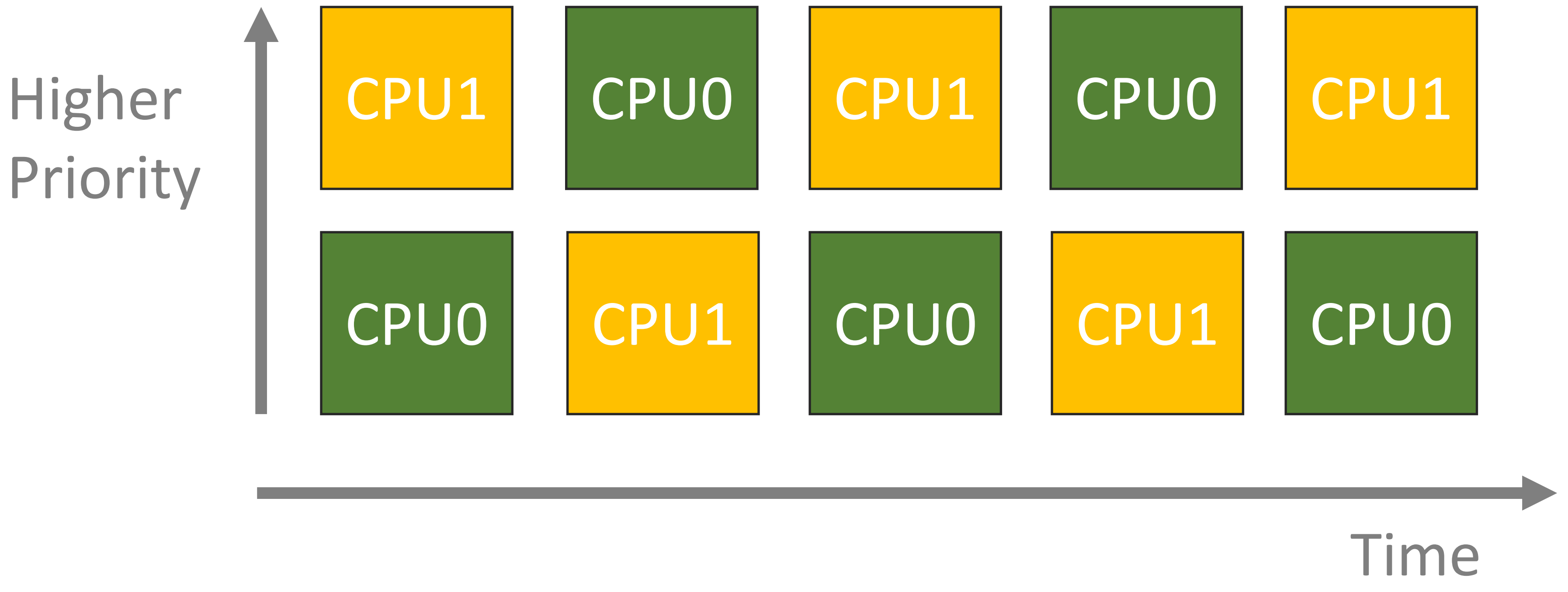
Highest Priority First



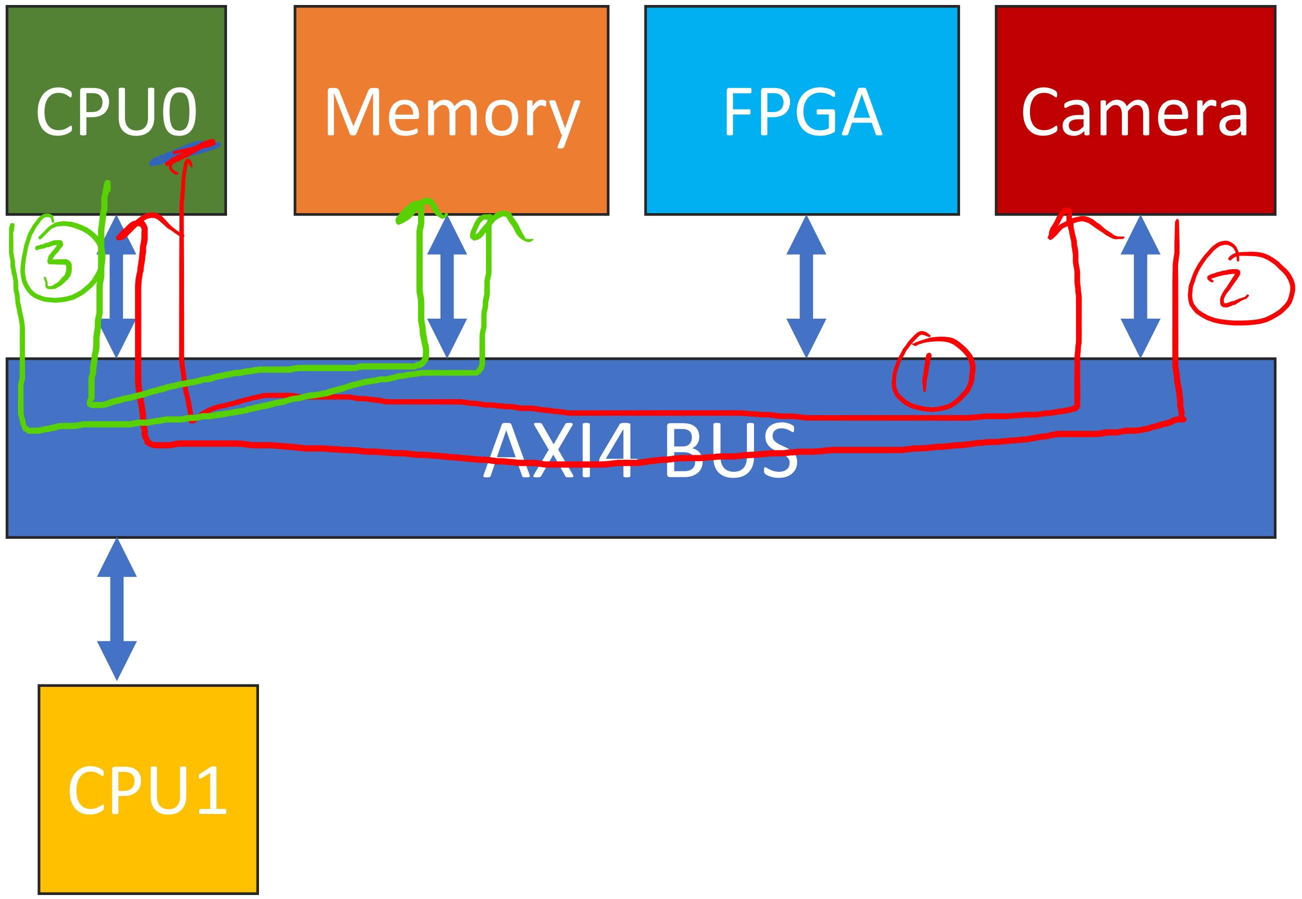
Round Robin

Add ~~a~~ example

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



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



A: The CPU copies data from Camera to Memory

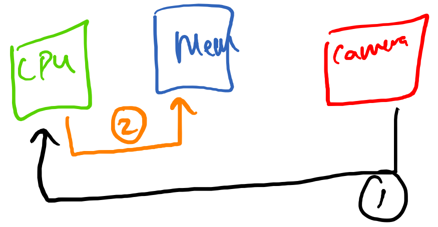
```
#define CAMERA_MMIO_ADDR 0x40000004
volatile uint32_t * camera =
        (uint32_t *)(CAMERA_MMIO_ADDR);
#define BUF_SIZE 1024;
uint32_t buf[BUF_SIZE];

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){
  (CPU Reg)     (camera)
       reg = *from;
   (memory)     (CPU Reg)
       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;
                                ③
    for (int i = 0; i < size; ++i){
    ① reg = *from;
    ② to[i] = reg;
    }④
}
```



① load
② store
③ incriment
④ branch

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?

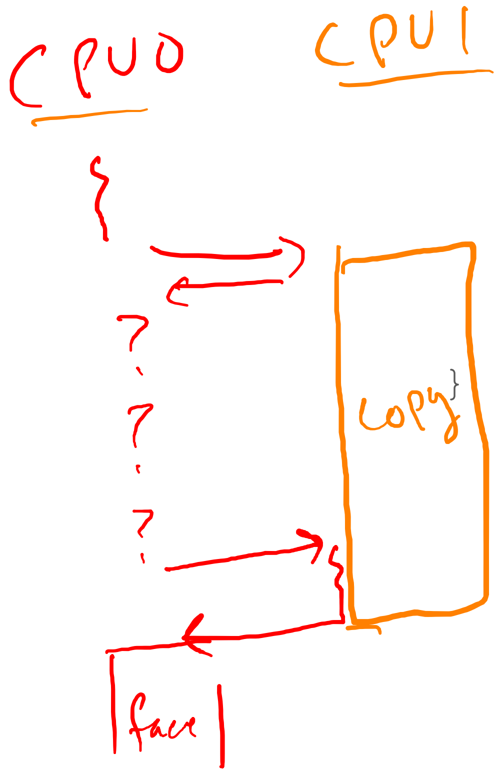
- CPU can do 1B instructions/second. (1GHz)
- 4 Instructions per loop
  - 1 load, 1 store, 1 increment, 1 branch
- 250M copies/second

- 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?

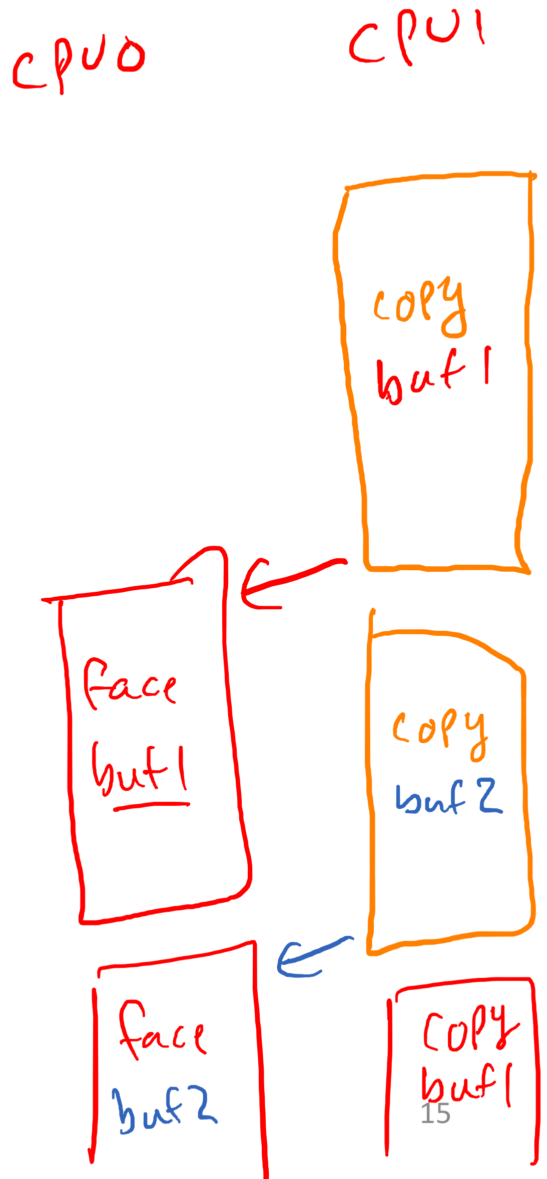
```
int main () {

    while (true){
                   start
        ask_cpu1_to^copy_image(camera, buf, BUF_SIZE);
        wait_for_cpu1_to_finish();

        detect_face(buf);
    }
}
```



What if we do the copy on CPU1?

```
int main () {

    while (true){

        ask_cpu1_to_copy_image(camera, buf1, BUF_SIZE);

        wait_for_cpu1_done();

        detect_face(buf1);
    }
}
```

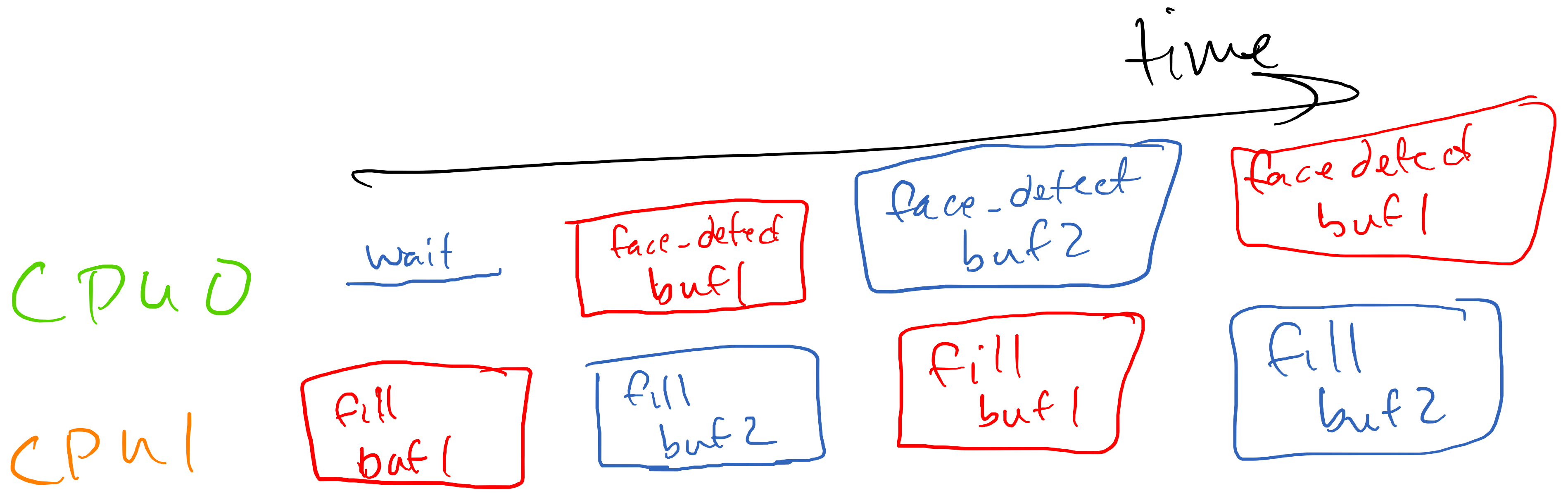
Queue of 1



Double-Buffering

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 buf 1
face_detect buf 2
face detect buf 1
fill buf 1
fill buf 2
fill buf 1
fill buf 2

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

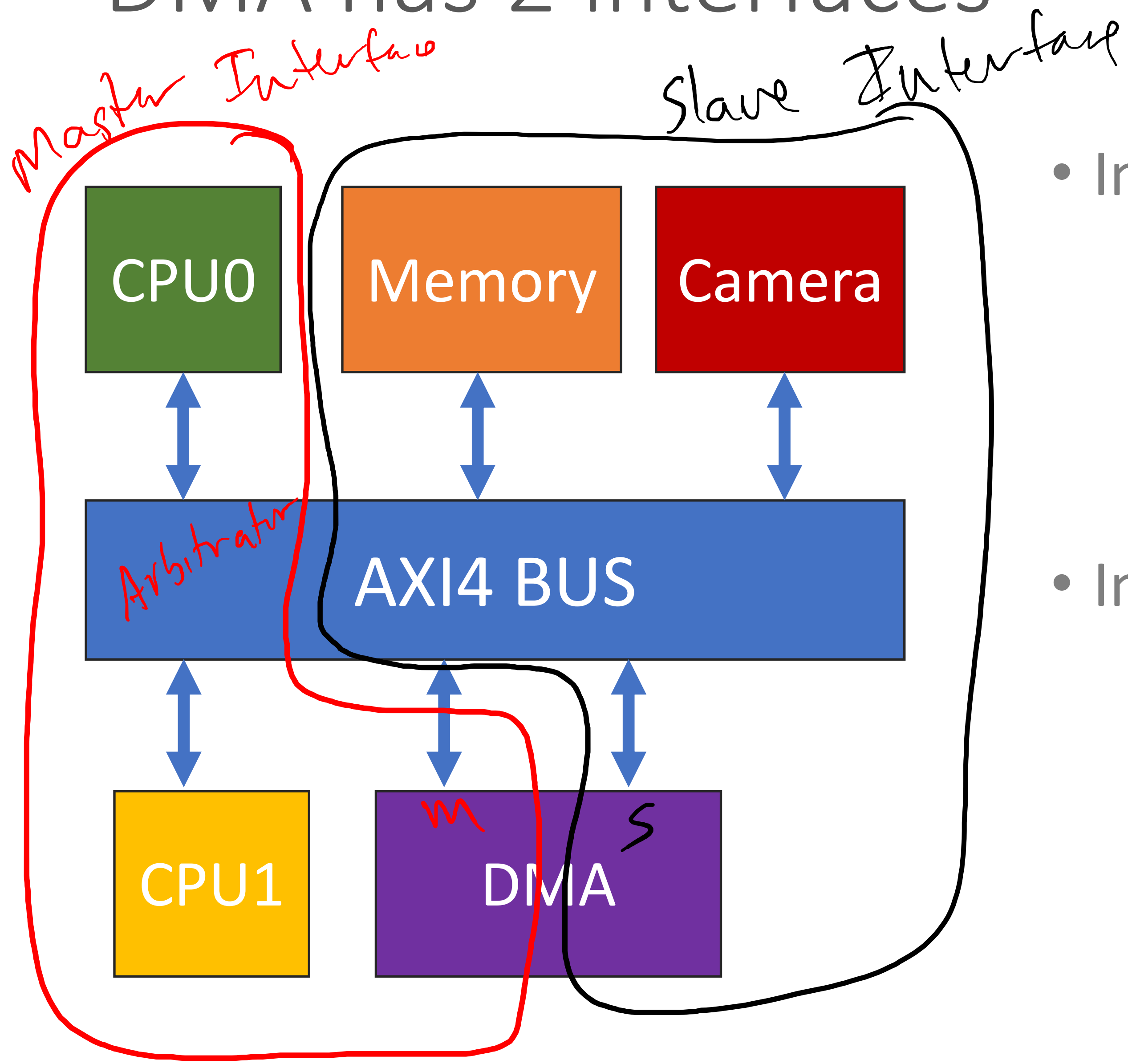
- A mini-CPU that does copy for you:

```
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;
    }
}
```

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();
    }
}
```

DMA has 2 interfaces



- 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?

source memory Address

destination memory Address

how big?

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:

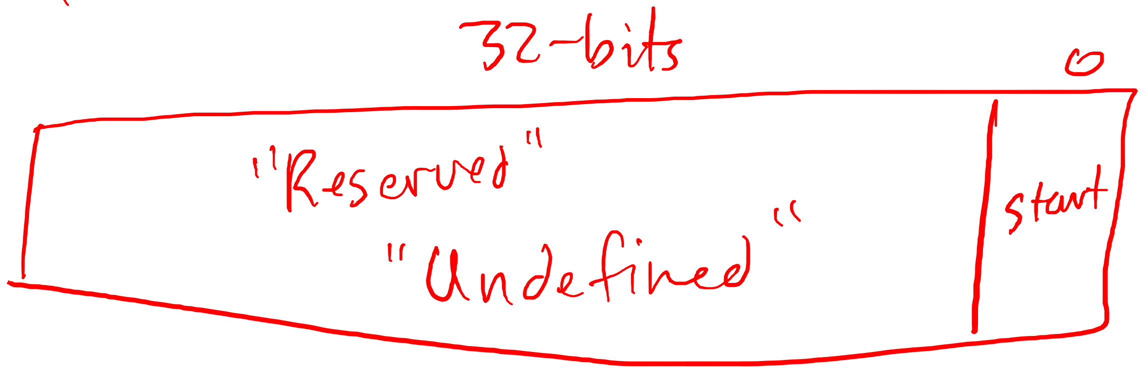
```
void dma_copy (uint32_t * from,
               uint32_t * to,
               uint32_t size)
{
   register uint32_t reg;

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

      reg = *from; //load

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

MyDMA MMIO Interface

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

MMIO Control Register

@ 0x0400

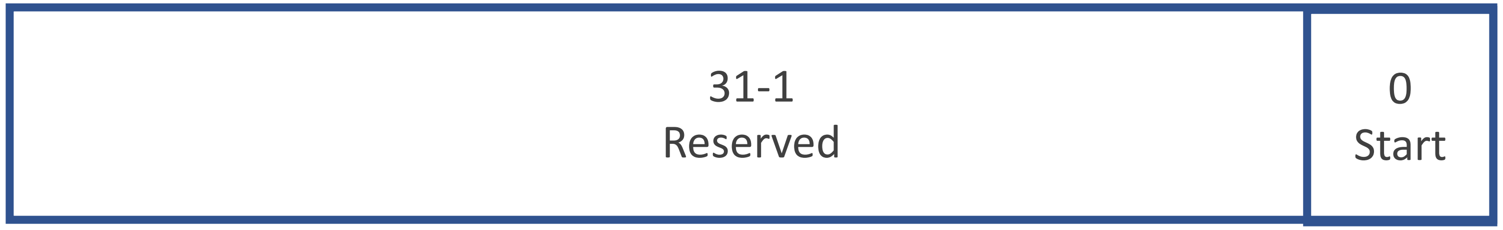


CPU-land

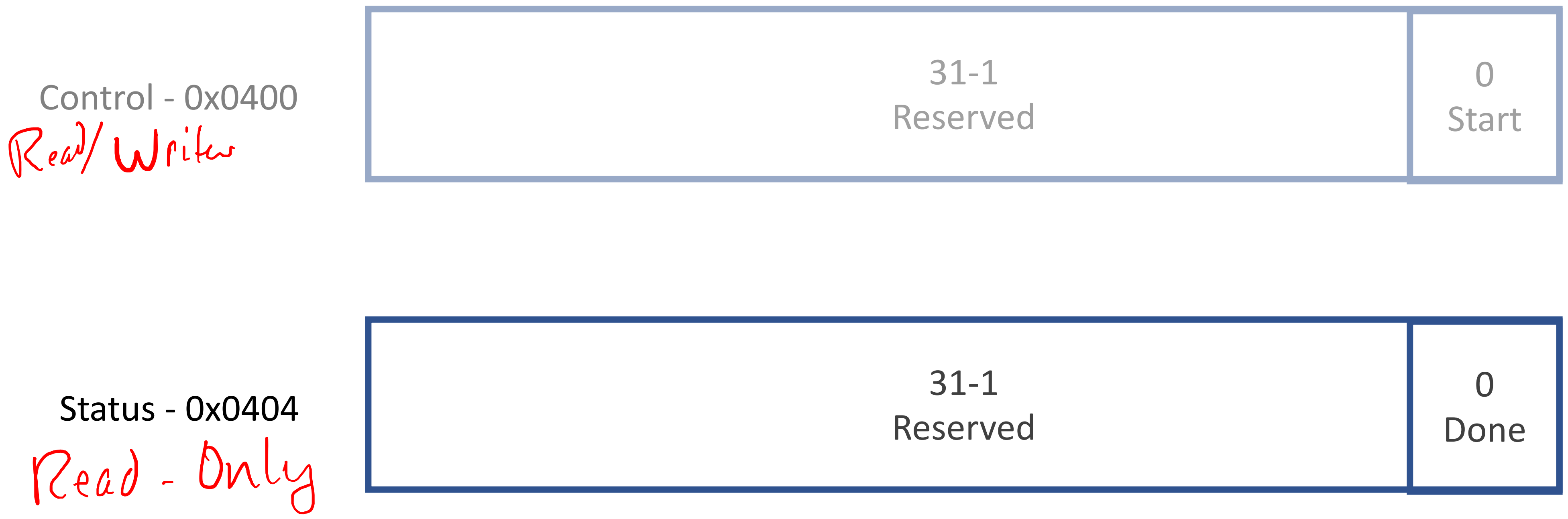
```
(*MyMMIO_Control_Reg) = 0x1; // Start the DMA
```

MMIO Control Register

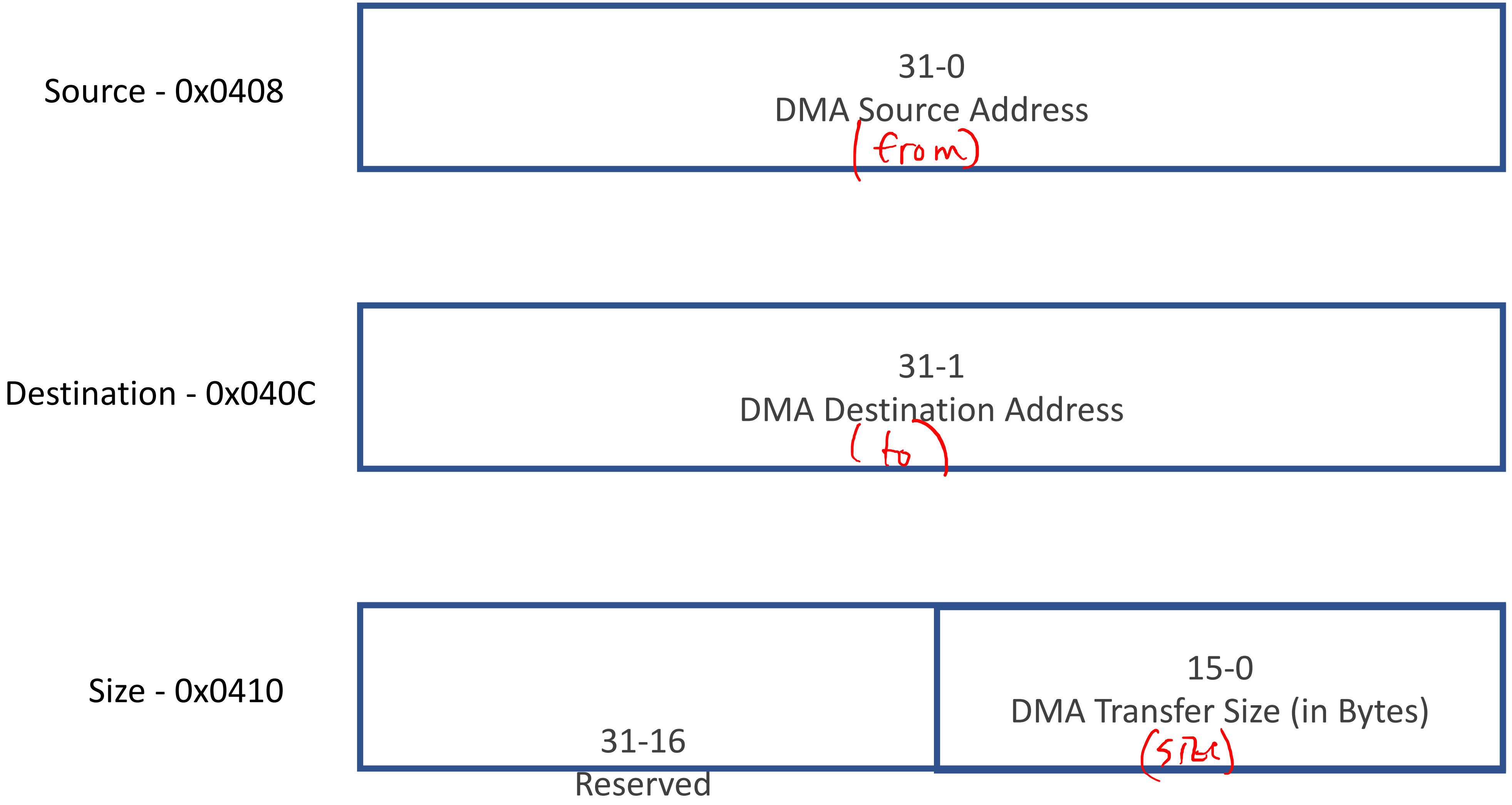
Control - 0x0400



MMIO Status Register



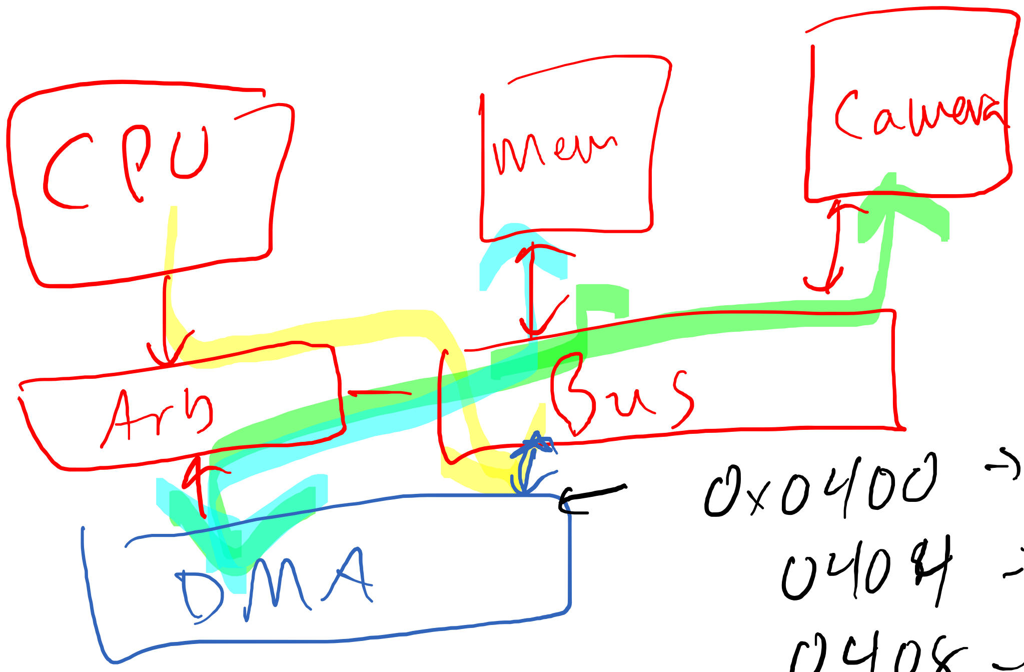
MMIO Data Registers



All MMIO Registers

				CPU store order
Control - 0x0400	R/W	31-1 Reserved	0 Start	$\underline{\underline{4}}$
Status - 0x0404	RO	31-1 Reserved	0 Done	
Source - 0x0408	R/W	31-0 DMA Source Address		1-3
Destination - 0x040C	R/W	31-1 DMA Destination Address		1-3
Size - 0x0410	R/W	31-16 Reserved	15-0 DMA Transfer Size (in Bytes)	1-3

MyDMA Interface



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

MyDMA Internals

DMA V1.0 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

0x0400: Control Register

```
void dma_copy ( uint32_t * ~~src~~ from,
                uint32_t * ~~dest~~ to,
                uint32_t size){
    ~~register uint32_t reg;~~
    ~~for (int i = 0; i < size; ++i){~~
        ~~reg = *from; //load~~
        ~~to[i] = reg; //store~~
    ~~}~~

    //code me!
}
```

```
*((volatile uint32t *)(0x0408)) = from;
*((   "        "      (0x040C)) = to
*(    "        "      (0x0410)) = size
*(volatile uint32_t *)(0x0400) = 0x1;
while ( (*(volatile uint32_t *)(0x0404) & 0x1) != 1)
```

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

Using DMA from the CPU:

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

     *((volatile uint32_t *)(0x0408))=src;
     *((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){;}
}
```

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

Using DMA from the CPU:

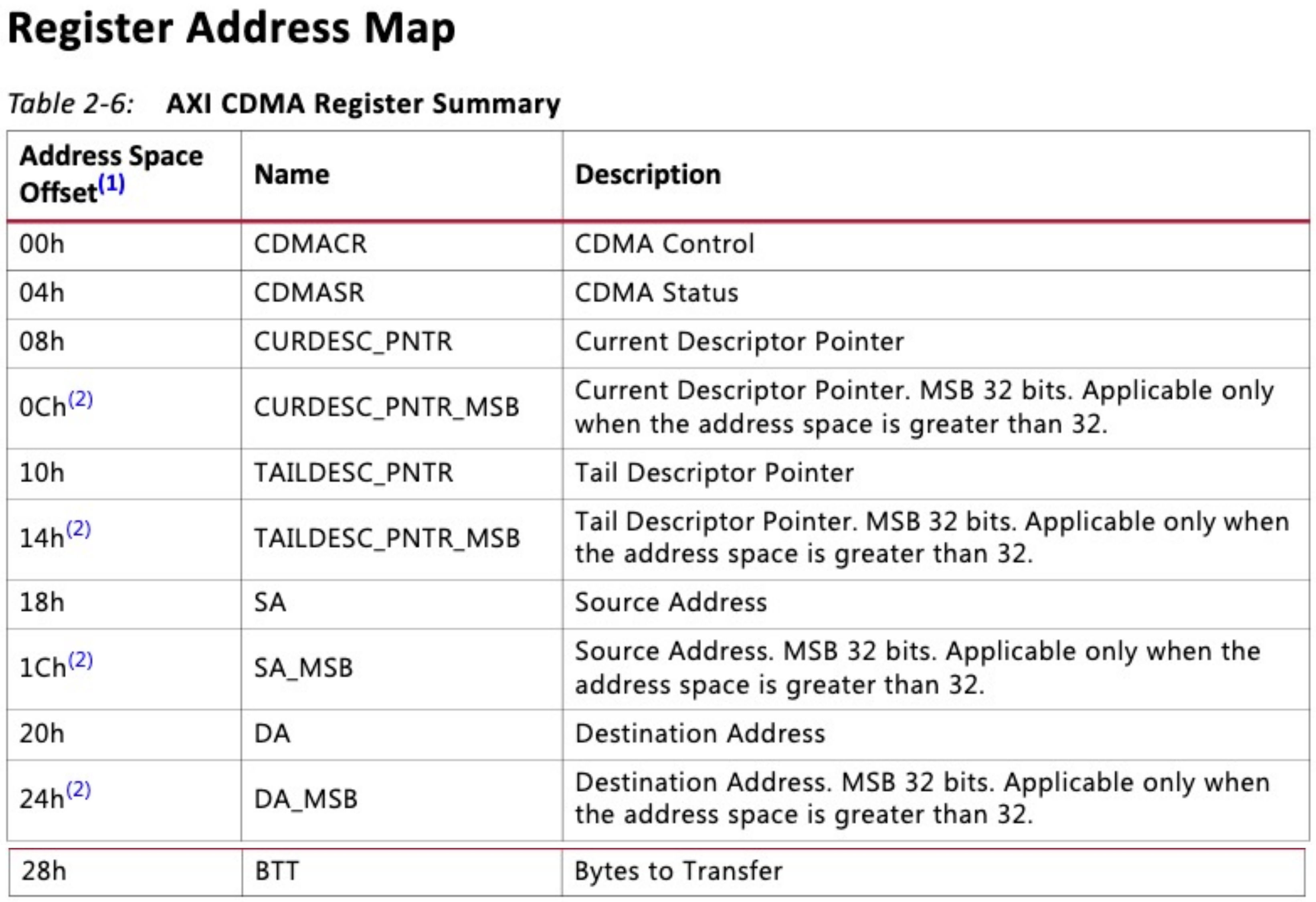
```
void dma_start_copy (   uint32_t * src,
                        uint32_t * dest,
                        uint32_t size){

        *((volatile uint32_t *)(0x0408))=src;
        *((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();
    }
}
```

Real DMA



Register Address Map

Table 2-6: **AXI CDMA Register Summary**

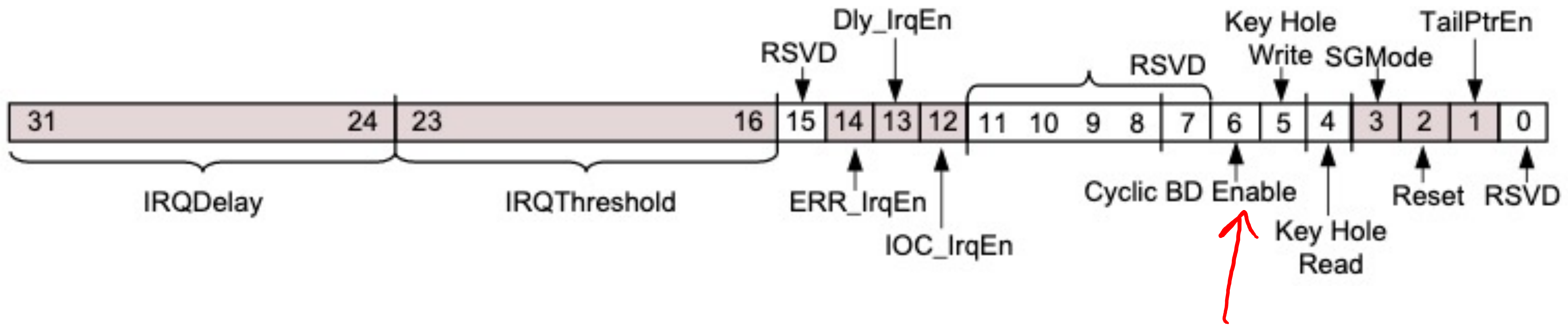
Address Space Offset[1]	Name	Description
00h	CDMACR	CDMA Control
04h	CDMASR	CDMA Status
08h	CURDESC_PNTR	Current Descriptor Pointer
0Ch[2]	CURDESC_PNTR_MSB	Current Descriptor Pointer. MSB 32 bits. Applicable only when the address space is greater than 32.
10h	TAILDESC_PNTR	Tail Descriptor Pointer
14h[2]	TAILDESC_PNTR_MSB	Tail Descriptor Pointer. MSB 32 bits. Applicable only when the address space is greater than 32.
18h	SA	Source Address
1Ch[2]	SA_MSB	Source Address. MSB 32 bits. Applicable only when the address space is greater than 32.
20h	DA	Destination Address
24h[2]	DA_MSB	Destination Address. MSB 32 bits. Applicable only when the address space is greater than 32.
28h	BTT	Bytes to Transfer

Real DMA

Register Details

CDMACR (CDMA Control – Offset 00h)

This register provides software application control of the AXI CDMA.



Other DMA tweaks

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

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

        reg = (inc_from ? *from[i] : *from);

        if (inc_to)  to[i] = reg;
        else         to = reg;
    }
}
```

Other DMA tweaks

- Interrupts (not in E315)
- Repeat the transfer?

References

- https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf

11: Direct Memory Access (DMA)

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University