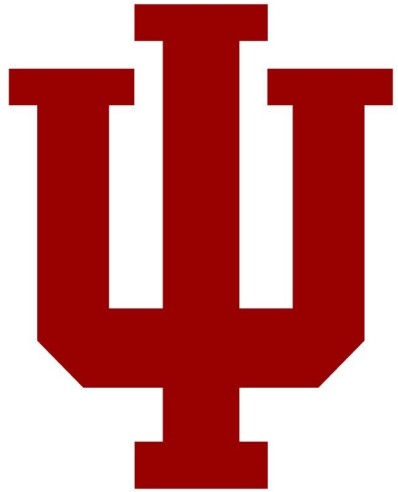
13: Real DMA

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



Announcements

P4 → Popcount (Python) MMIO ~ 10 min

- P5 is due Today! → Popcount (C) MMIO
- P6 out → Popcount (Python) DMA
  - AG not up yet?

P7 → Popcount (C) DMA ~ 0.5 seconds

- No Class Wednesday, Oct 26th.

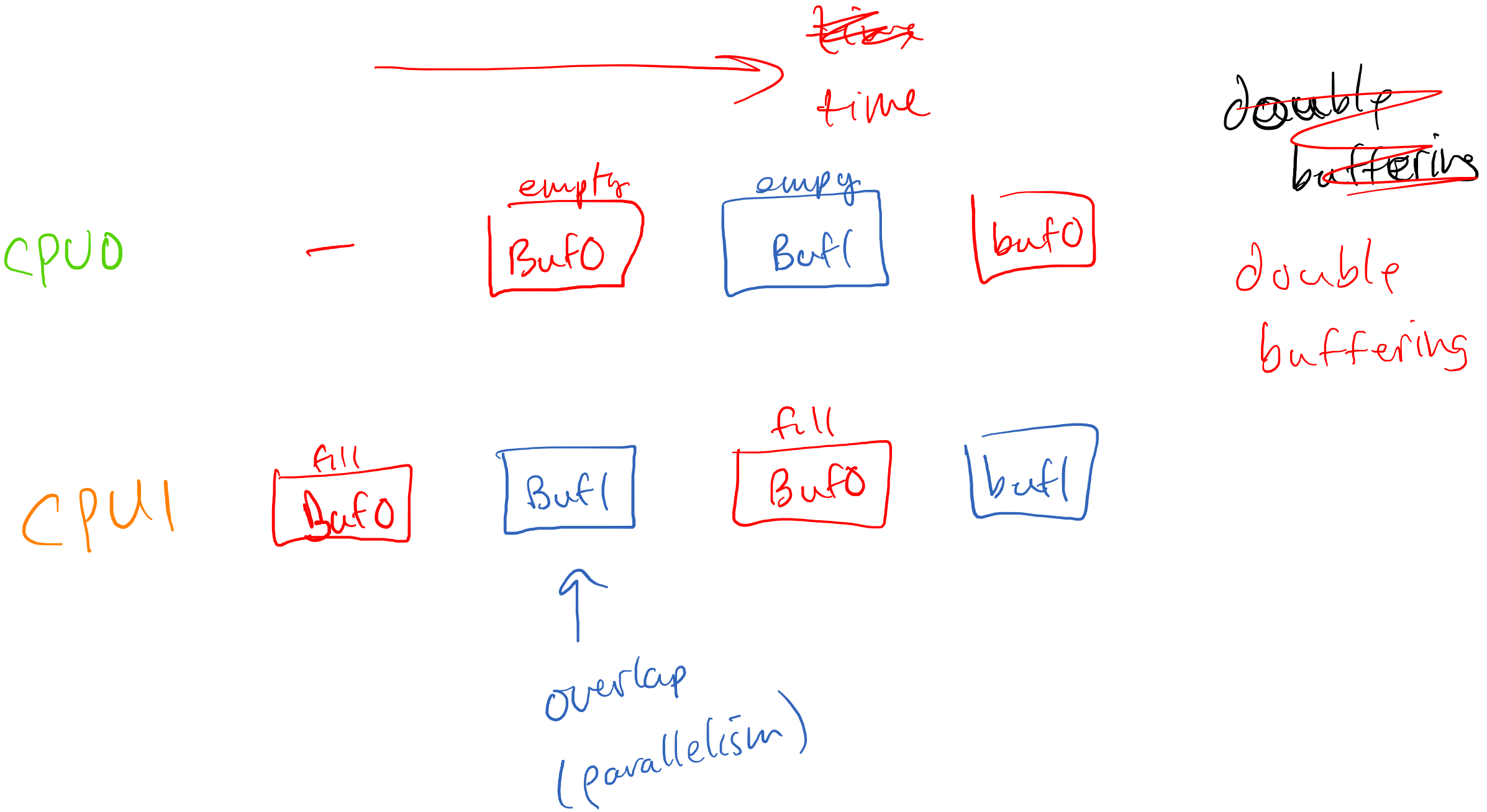
DMA: a mini-CPU that does this:

```
void copy (uint32_t * source,
                uint32_t * dest,
                uint32_t size)
{
   register uint32_t reg;

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

        reg = *source;

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

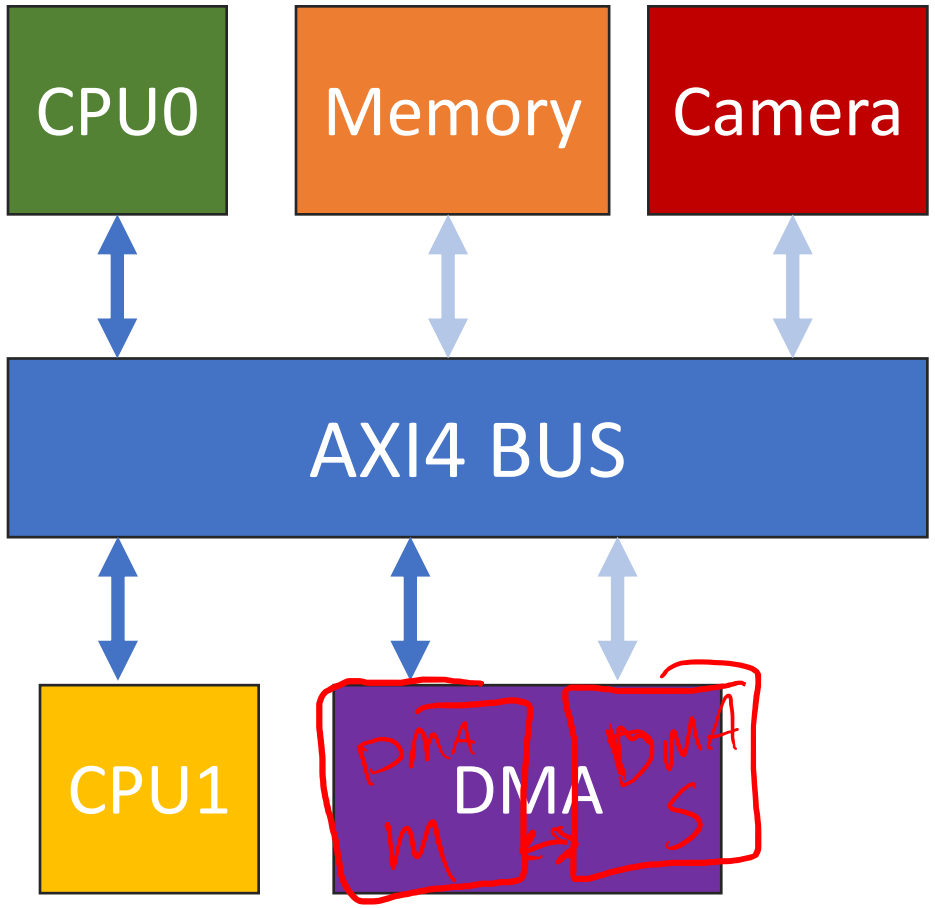
Using DMA from C:

double buffering

```
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();
    }
}
```


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

DMA has 2 Memory Interfaces



- Interface 1: Memory Copy
  - Data Interface
  - Fast
  - Master
- Interface 2: Tell DMA what to copy
  - Control Interface
  - Slower
  - Slave

DMA V1.0 Interface

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

All MMIO Registers

CPU store order

Control - 0x0400
R/W

31-1	0
Reserved	Start

4

Status - 0x0404
RO

31-1	0
Reserved	Done

Source - 0x0408
R/W

31-0
DMA Source Address

1-3

Destination - 0x040C
R/W

31-1
DMA Destination Address

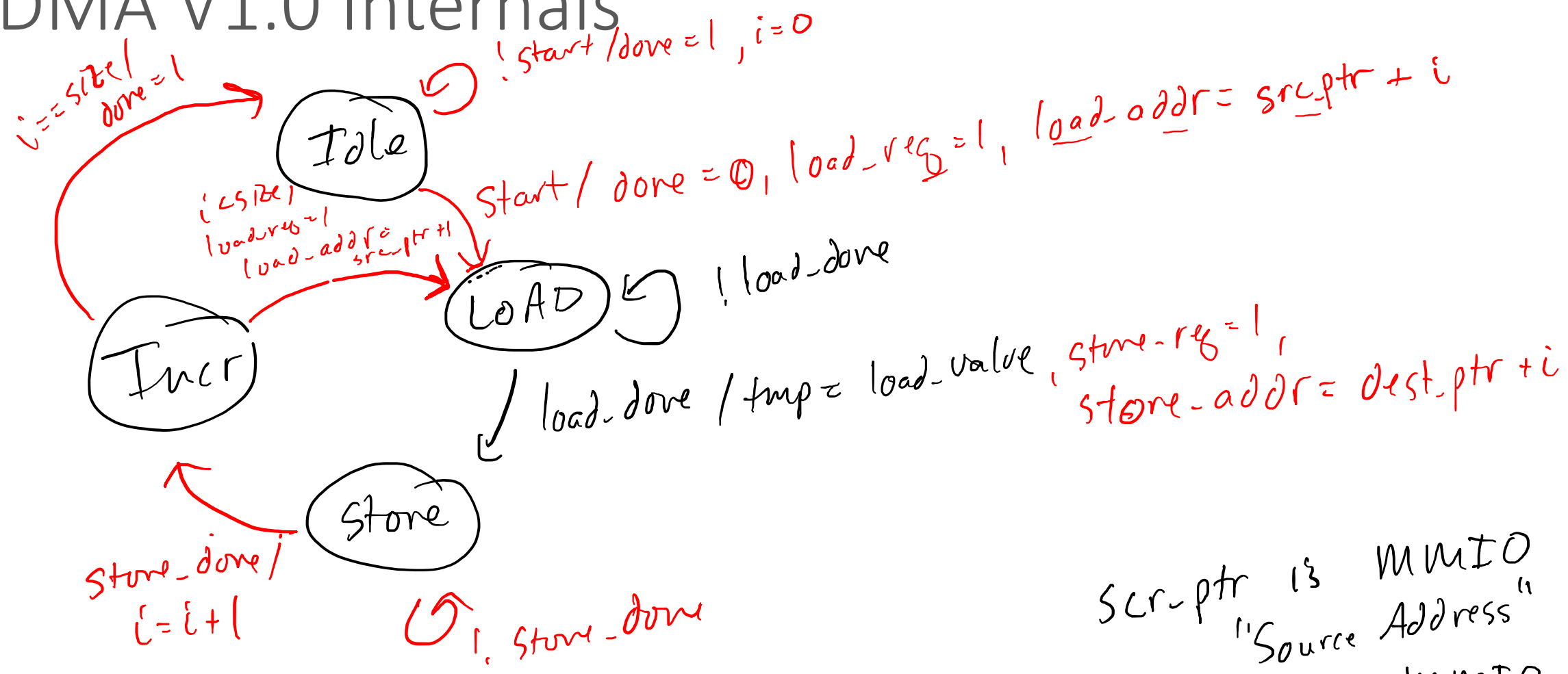
1-3

Size - 0x0410
R/W

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

1-3

DMA V1.0 Internals



tmp = temporary value

scr_ptr is MMIO "Source Address"

dest_ptr is MMIO "Destination Address"

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 * 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){;}
}
```

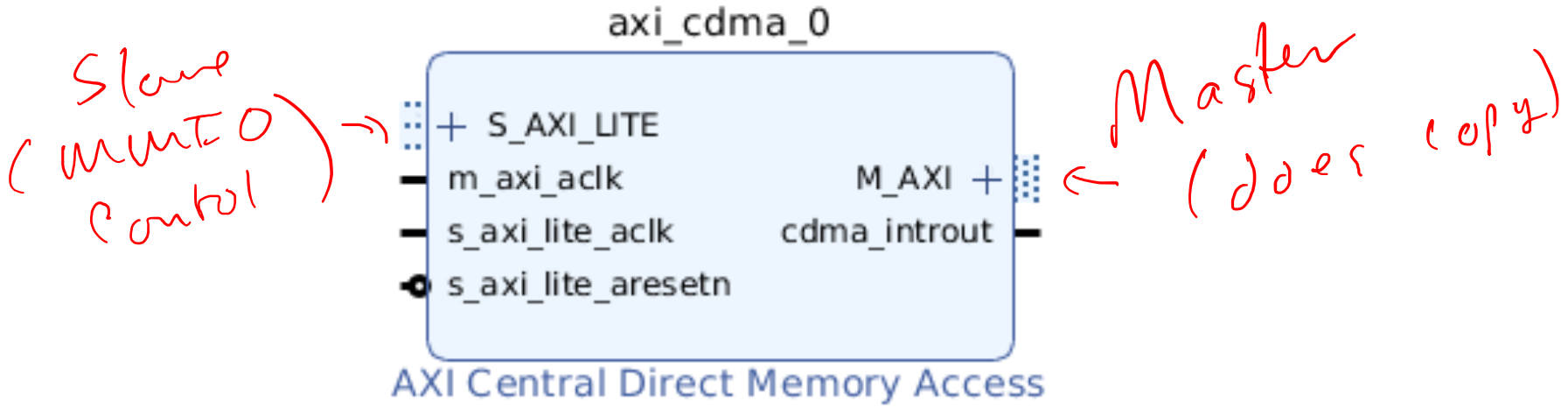
Using DMA from the CPU:

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

```
void dma_start_copy (   uint32_t * 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!

DMA in Xilinx



Xilinx terminology:

- Central Direct Memory Access : CMDA
  - Standard DMA
- Direct Memory Access (DMA)
  - Programmable DMA!

Real DMA

Register Address Map

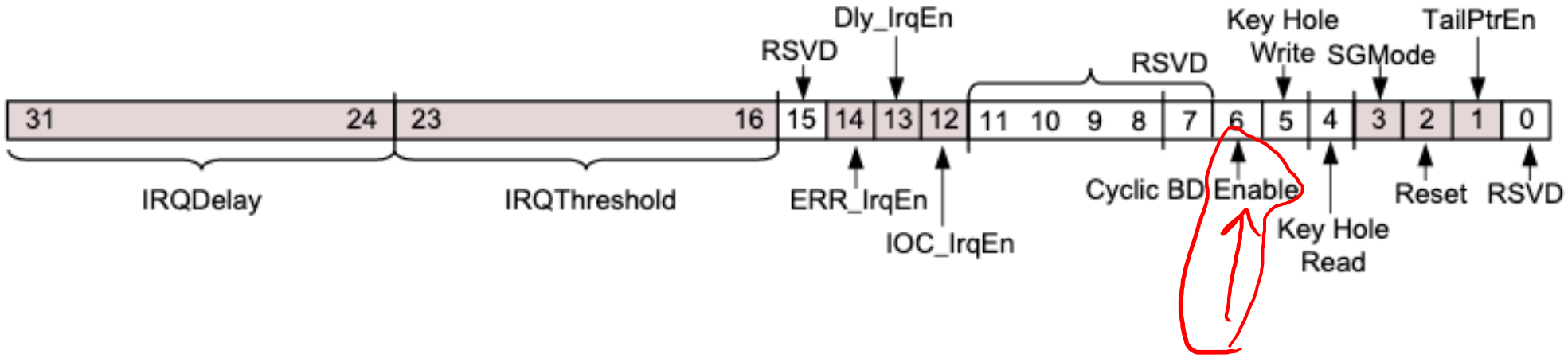
Table 2-6: **AXI CDMA Register Summary**

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

Register Details

CDMACR (CDMA Control – Offset 00h)

This register provides software application control of the AXI CDMA.

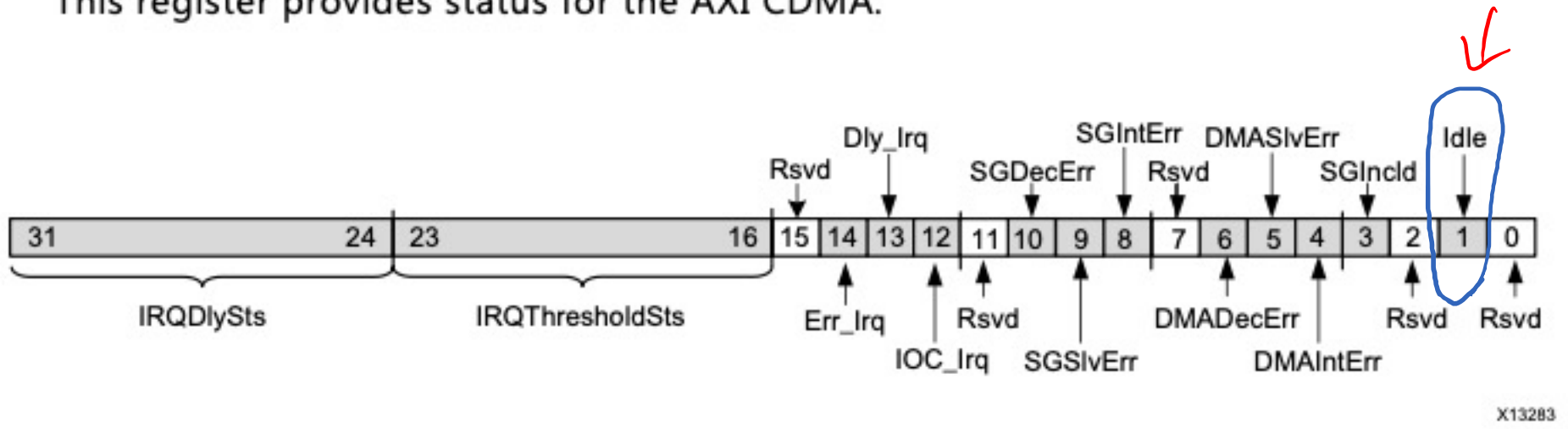


means: "turn on"

not "start running"

CDMASR (CDMA Status – Offset 04h)

This register provides status for the AXI CDMA.



Programming Sequence

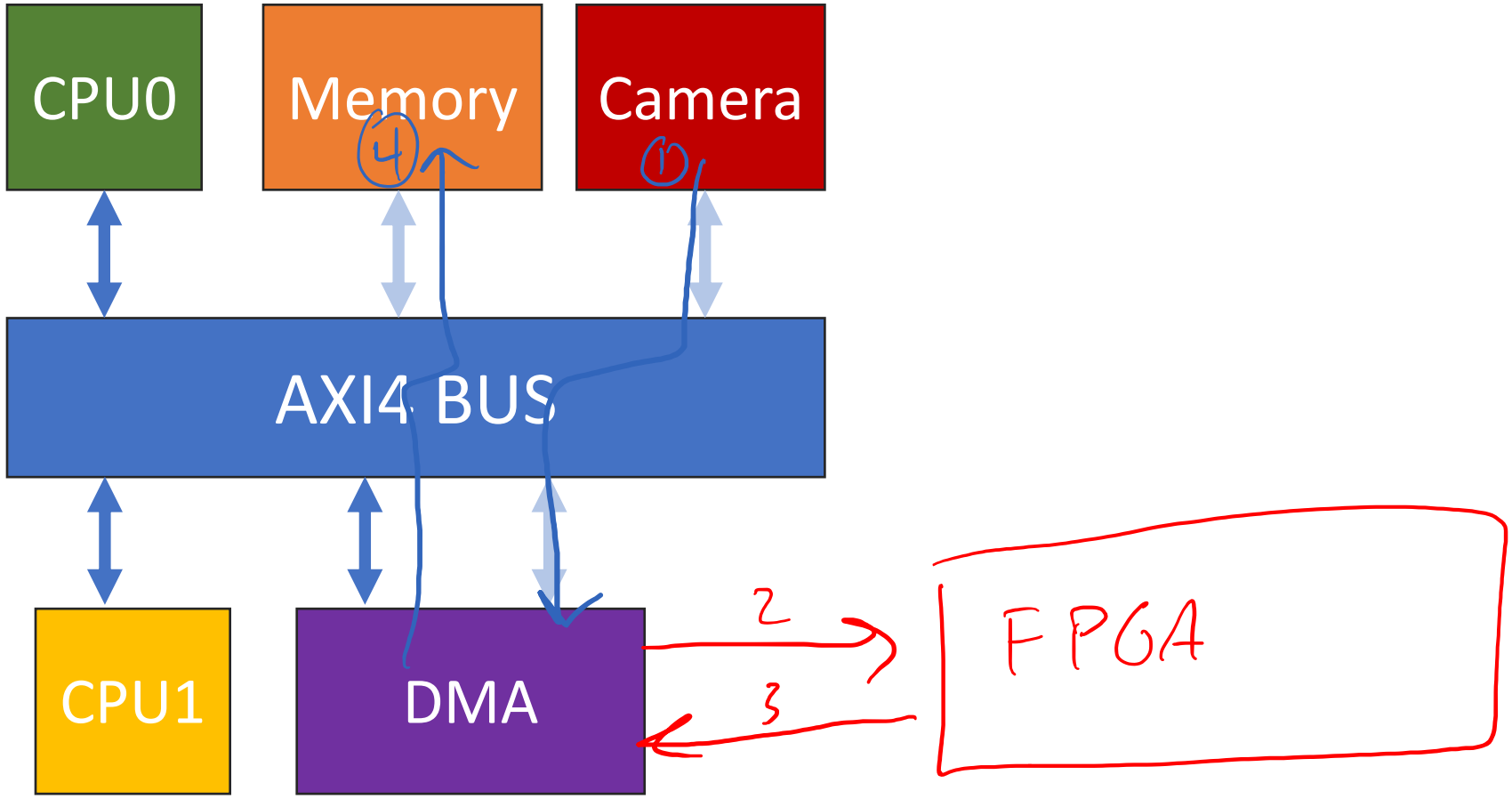
Simple DMA mode is the basic mode of operation for the CDMA when Scatter Gather is excluded. In this mode, the CDMA executes one programmed DMA command and then stops. This requires the CDMA registers to be set up by an external AXI4 Master for each DMA operation required.

These basic steps describe how to set up and initiate a CDMA transfer in simple operation mode.

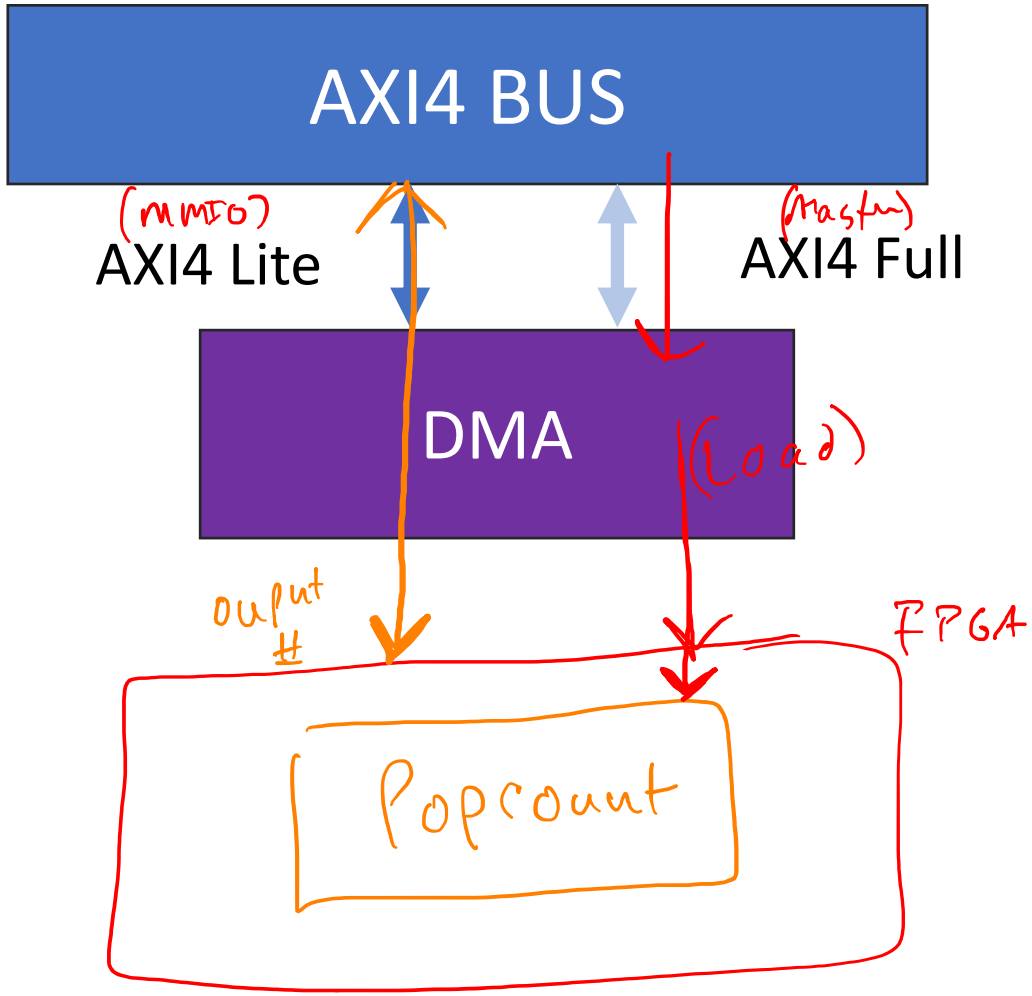
0: Enable DMA: CDMACR.ENABLE = 1

1. Verify CDMASR.IDLE = 1.
2. Program the CDMACR.IOC_IrqEn bit to the desired state for interrupt generation on transfer completion. Also set the error interrupt enable (CDMACR.ERR_IrqEn), if so desired.
3. Write the desired transfer source address to the Source Address (SA) register. The transfer data at the source address must be valid and ready for transfer. If the address space selected is more than 32, write the SA_MSB register also.
4. Write the desired transfer destination address to the Destination Address (DA) register. If the address space selected is more than 32, then write the DA_MSB register also.
5. Write the number of bytes to transfer to the CDMA Bytes to Transfer (BTT) register. Up to 8,388,607 bytes can be specified for a single transfer (unless DataMover Lite is being used). Writing to the BTT register also starts the transfer.
6. Either poll the CDMASR.IDLE bit for assertion (CDMASR.IDLE = 1) or wait for the CDMA to generate an output interrupt (assumes CDMACR.IOC_IrqEn = 1).
7. If interrupt based, determine the interrupt source (transfer completed or an error has occurred).
8. Clear the CDMASR.IOC_Irq bit by writing a 1 to the DMASR.IOC_Irq bit position.
9. Ready for another transfer. Go back to step 1.

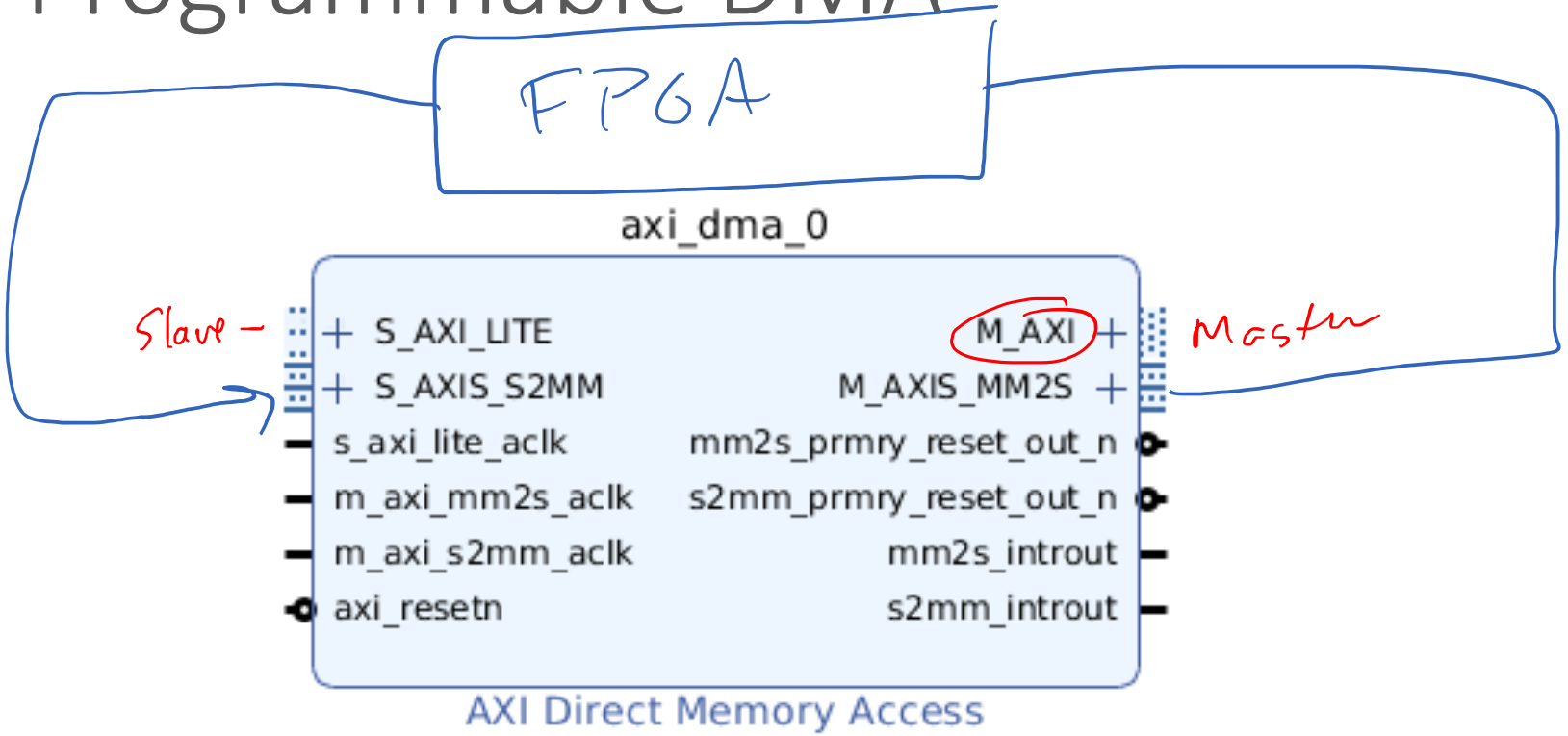
DMAs copy by load + store



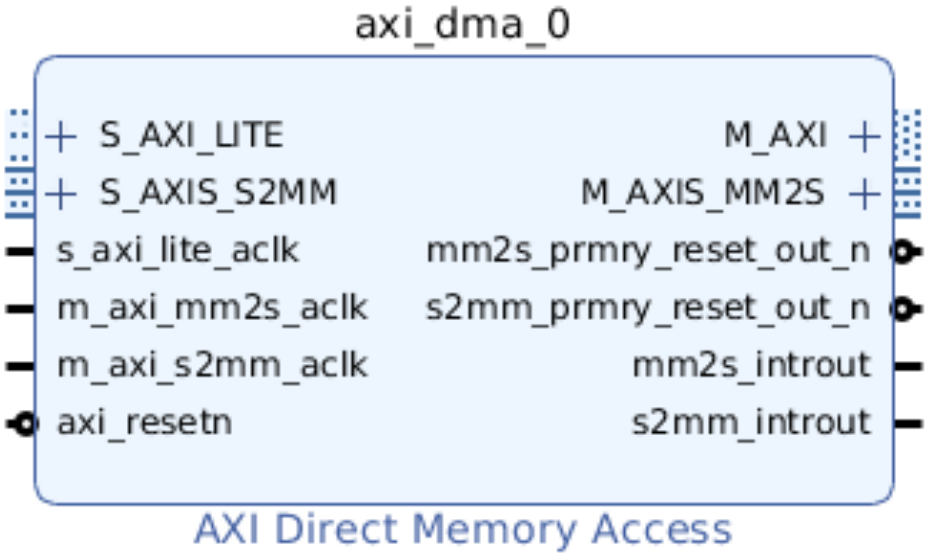
Xilinx DMAs allow you to **program** what the DMA does



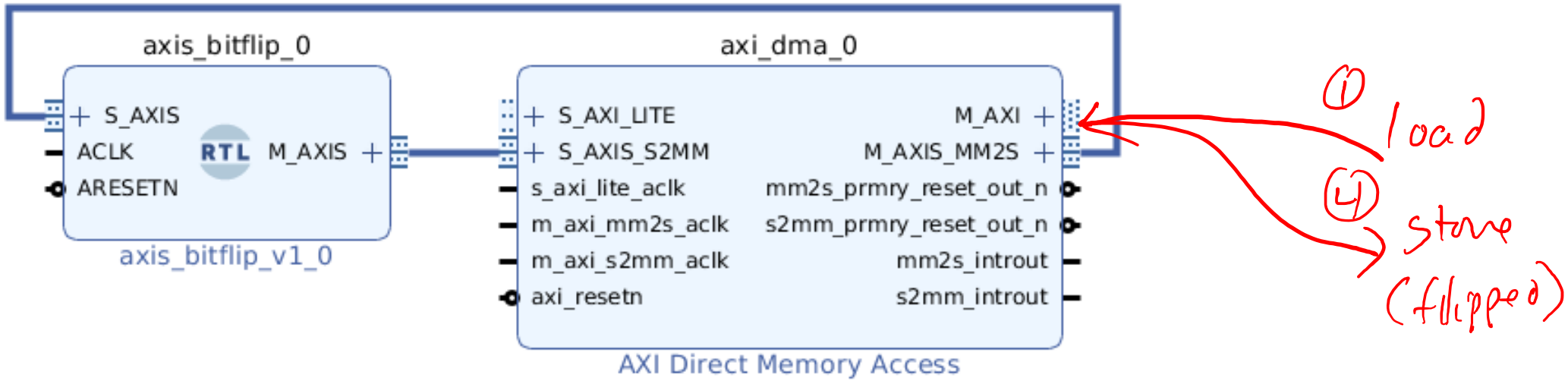
Xilinx Programmable DMA



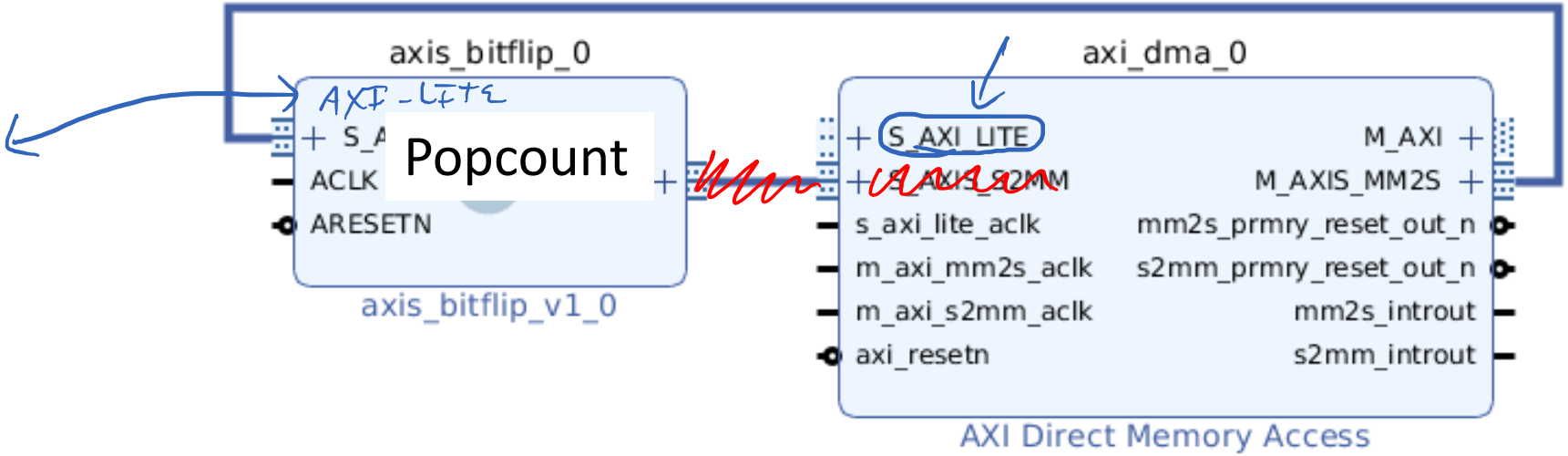
Can we make this into a regular DMA?



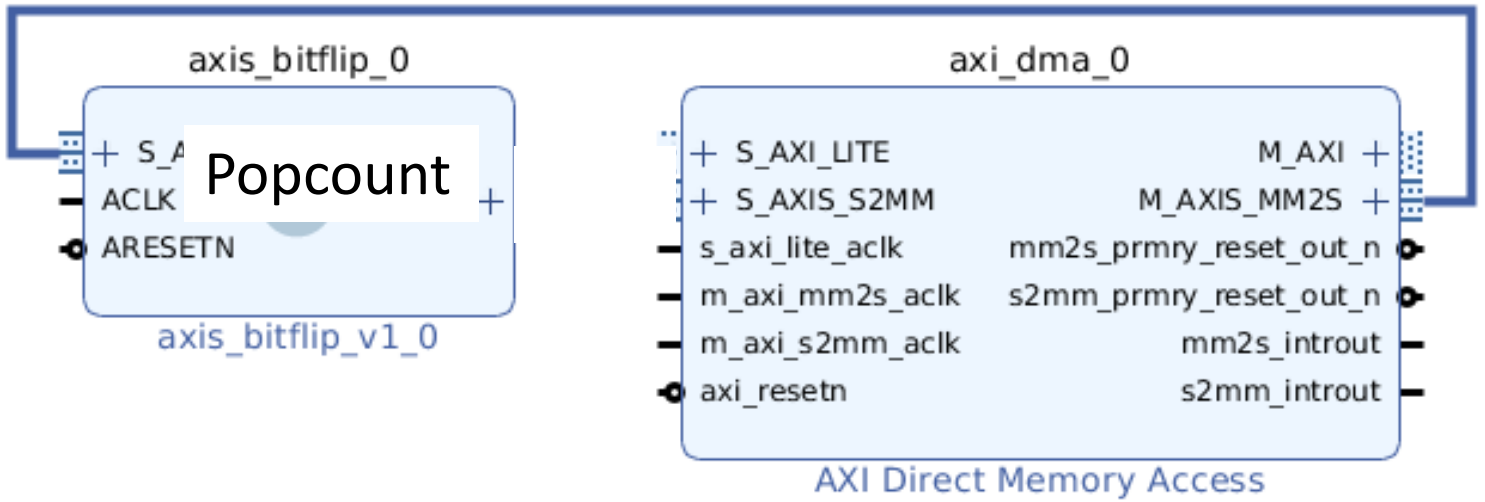
DMA that flips all the bits in the copy



DMA that flips all the bits in the copy



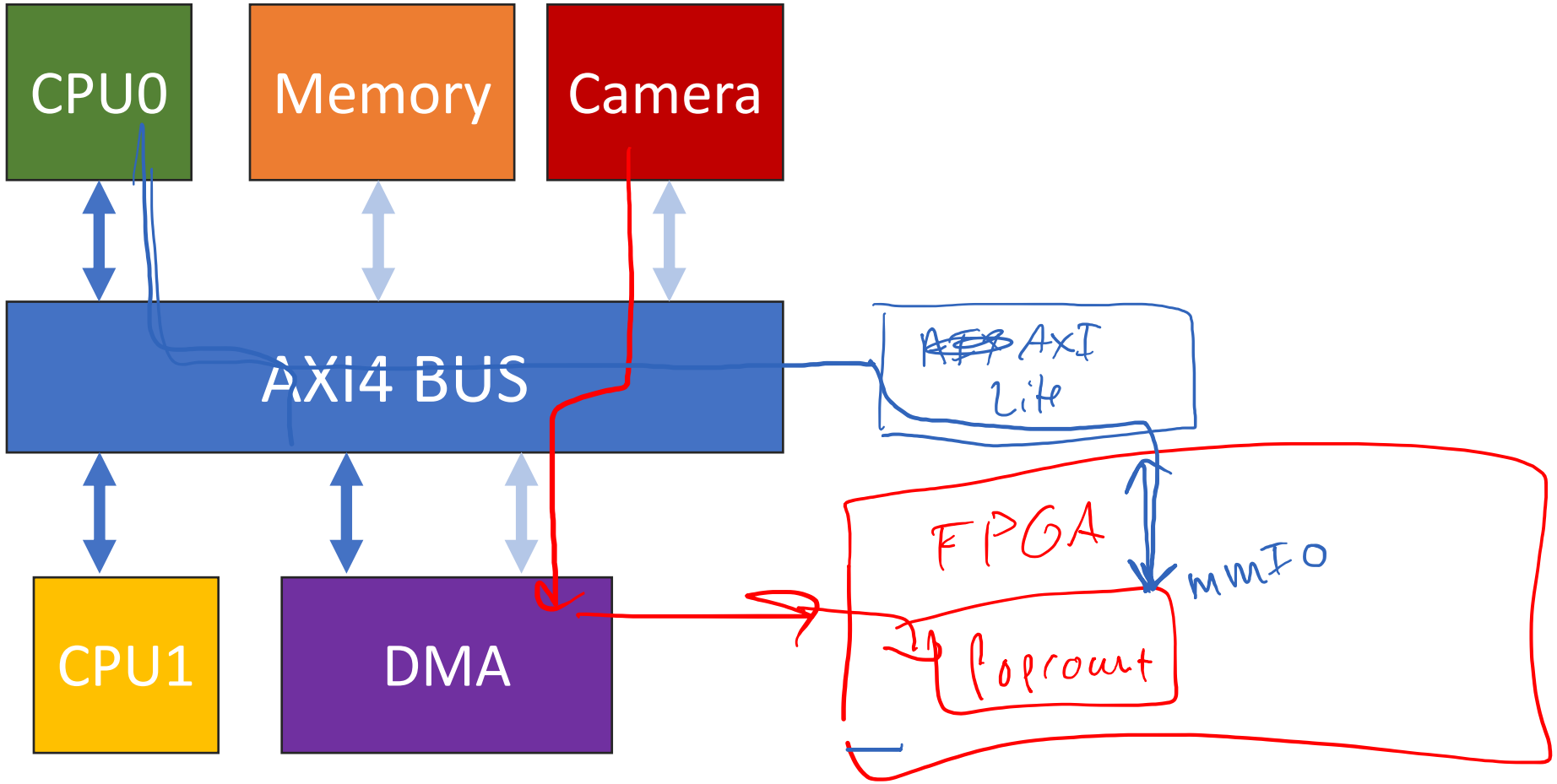
We don’t need the write portion.



Load: Yes

No Store!

With FPGAs, DMAs can copy with load + **Popcount** ~~+ store~~!



13: Real DMA

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University