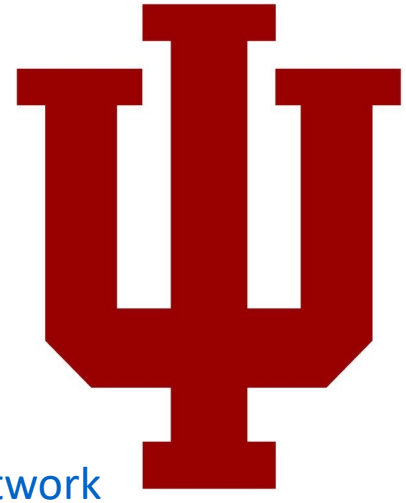
15: DMA III

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



Some material taken from:
https://github.com/trekhleb/homemade-machine-learning/tree/master/homemade/neural_network
http://cs231n.github.io/neural-networks-1/

Announcements

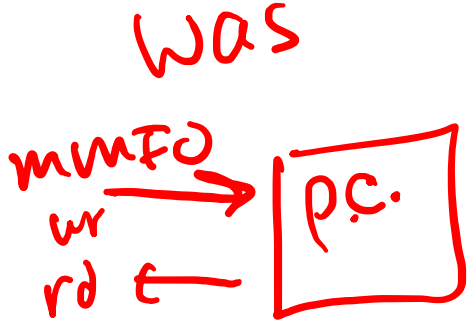
- P6 due week from Wed
  - ~~AG is new...~~
  - P7: new AG
- No Class Wednesday, Oct 25th.
  - demo due Friday after

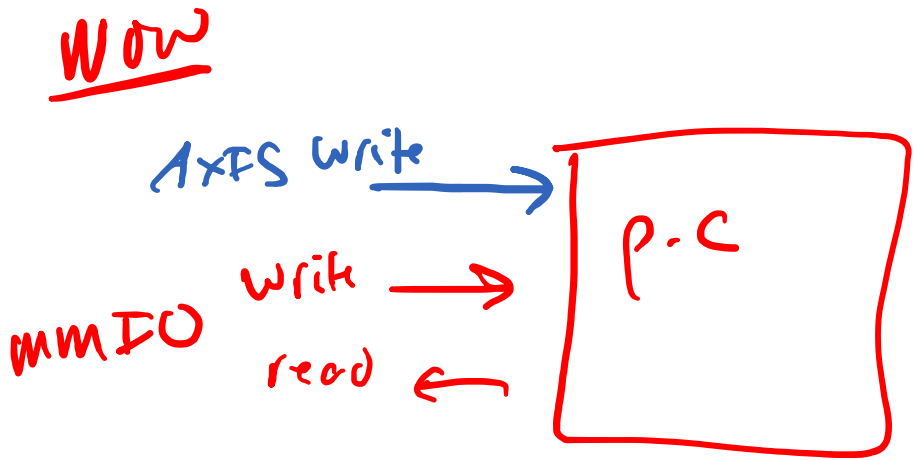
Exam Planning

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

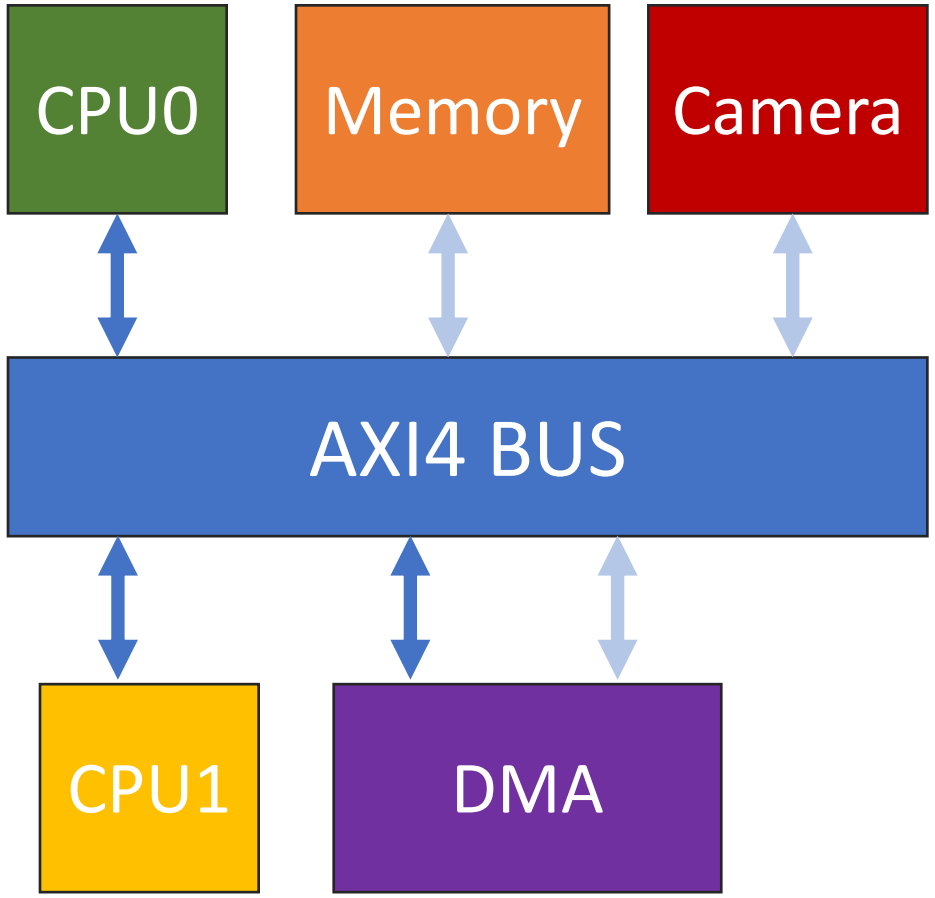
P6: Adds DMA + AXI-Stream to Popcount

- DMA
  - Add DMA engine to move data via AXI4-Full to AXI-Stream interface
- Popcount.sv:
  - Add AXI-Stream Interface
  - Keep AXI4-Lite Interface to read result



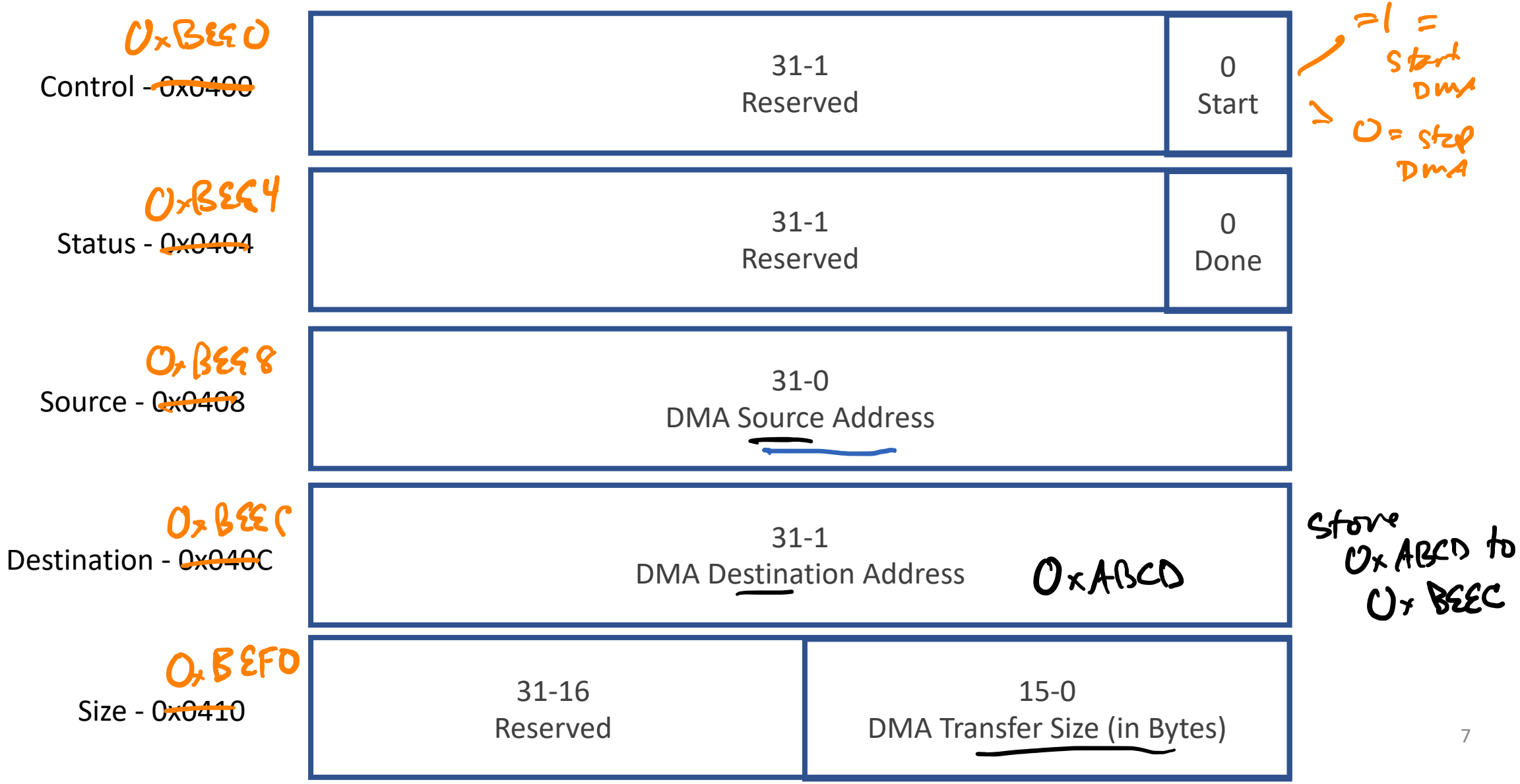


DMA has 2 Memory Interfaces



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

All MMIO Registers



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){;}
}
```

Fixme: `volatile` keyword omitted!

Xilinx CDMA

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

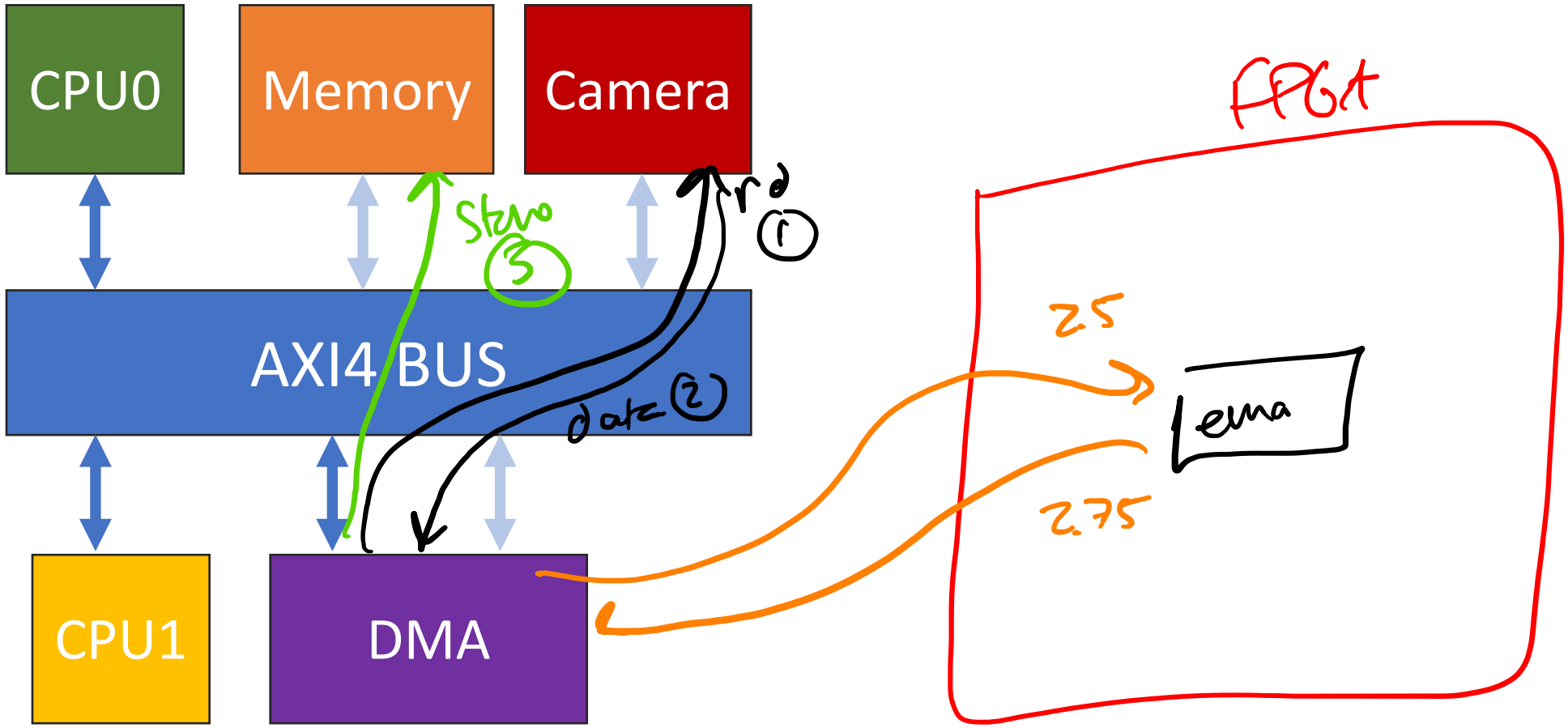
Programming Sequence

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.

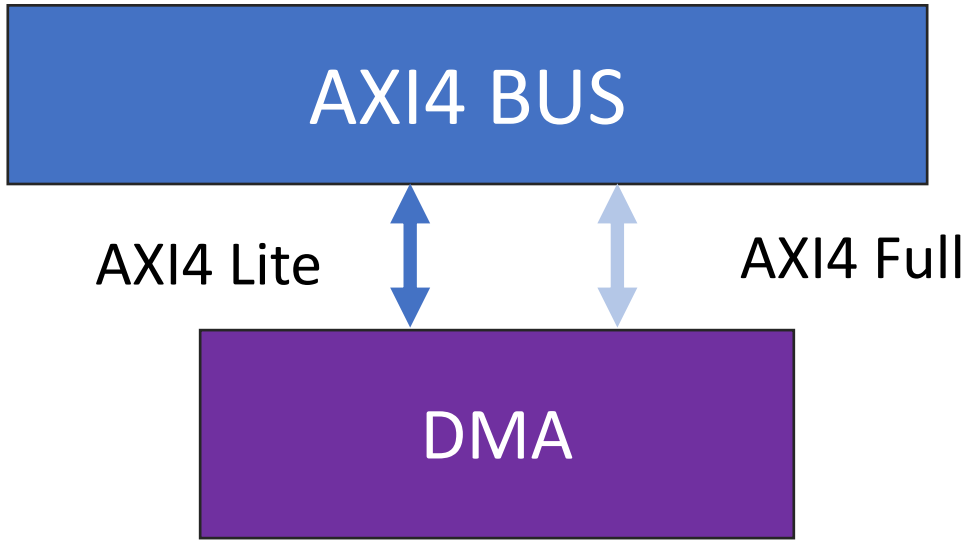
These basic steps describe how to set up and initiate a CDMA transfer in simple operation mode.

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.

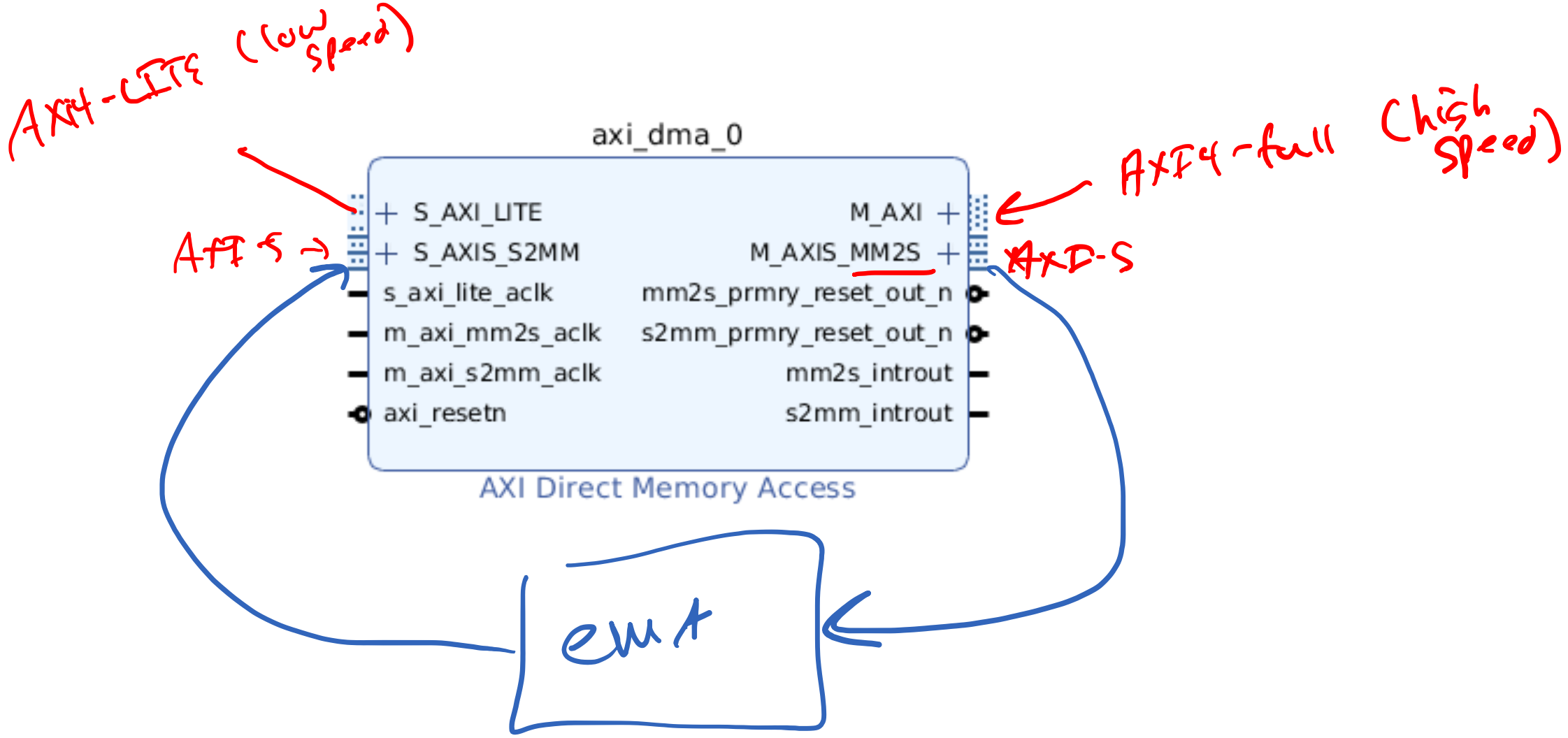
Xilinx DMAs do more than load + store



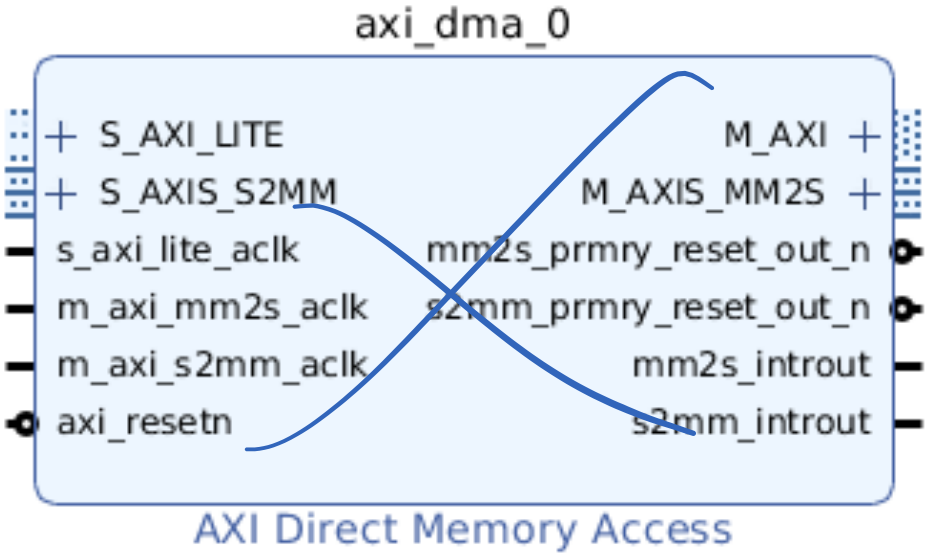
Xilinx DMAs allow you to **program** what the DMA does



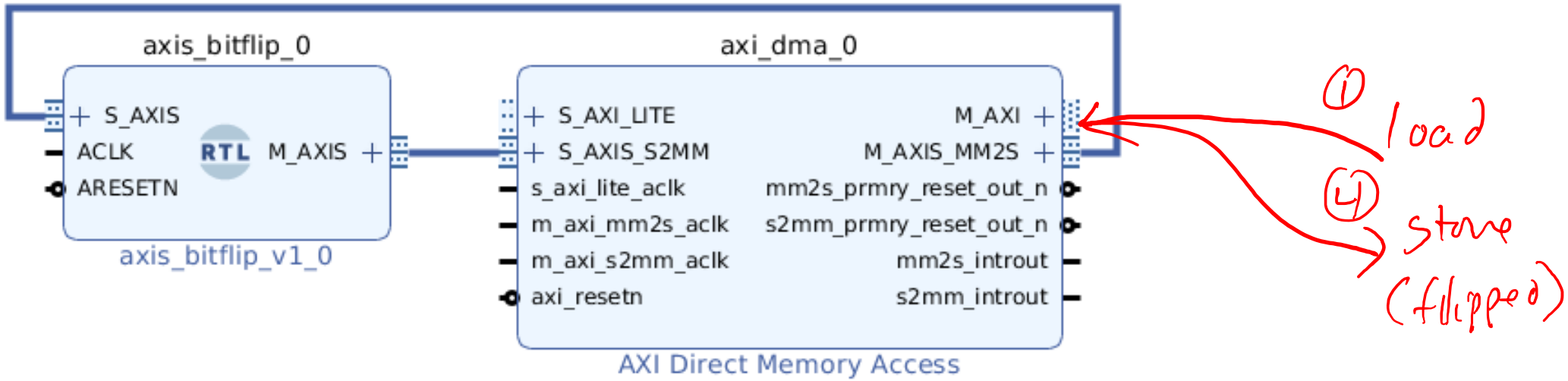
Xilinx Programmable DMA



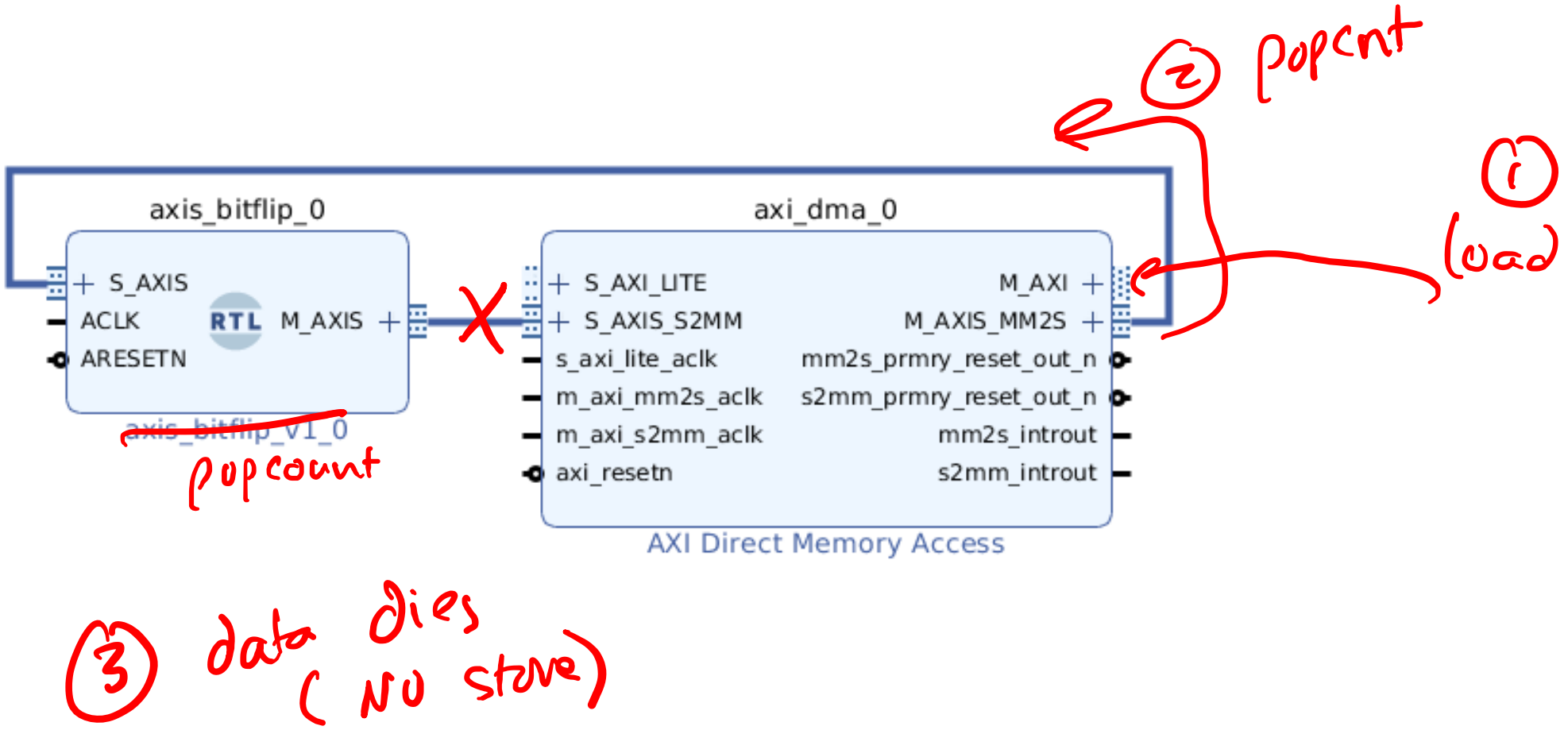
Can we make this into a regular DMA?



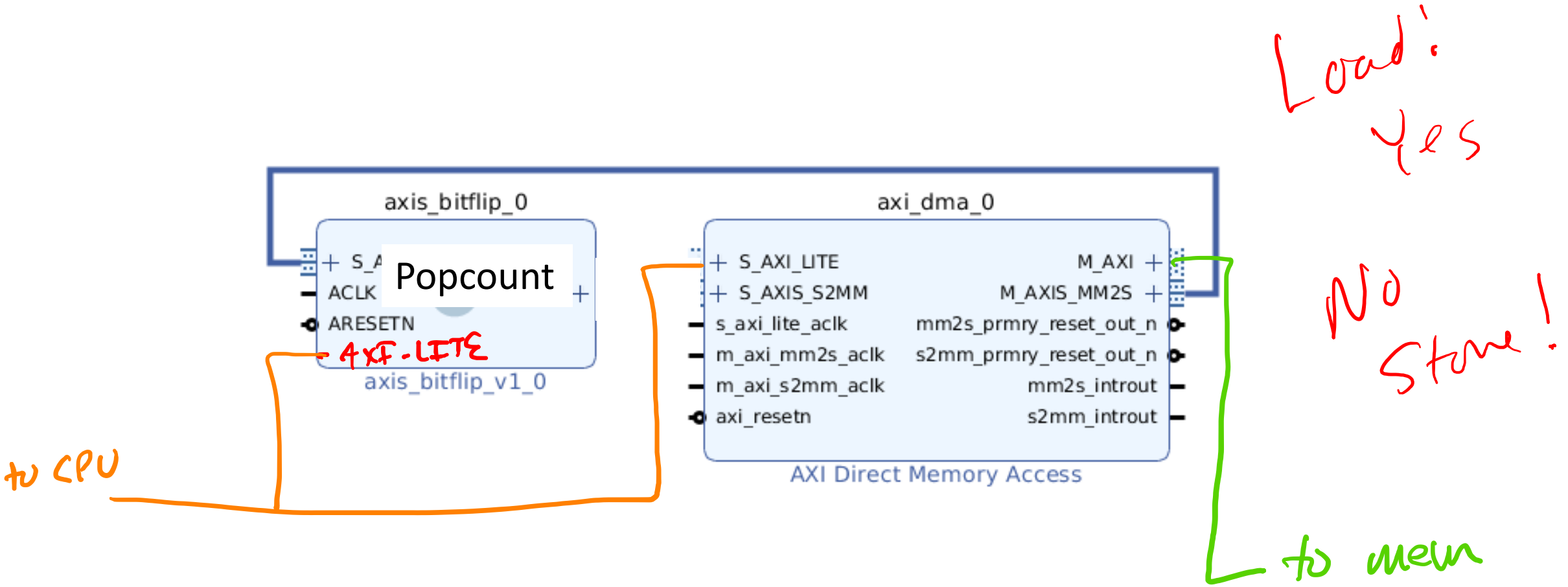
DMA that flips all the bits in the copy



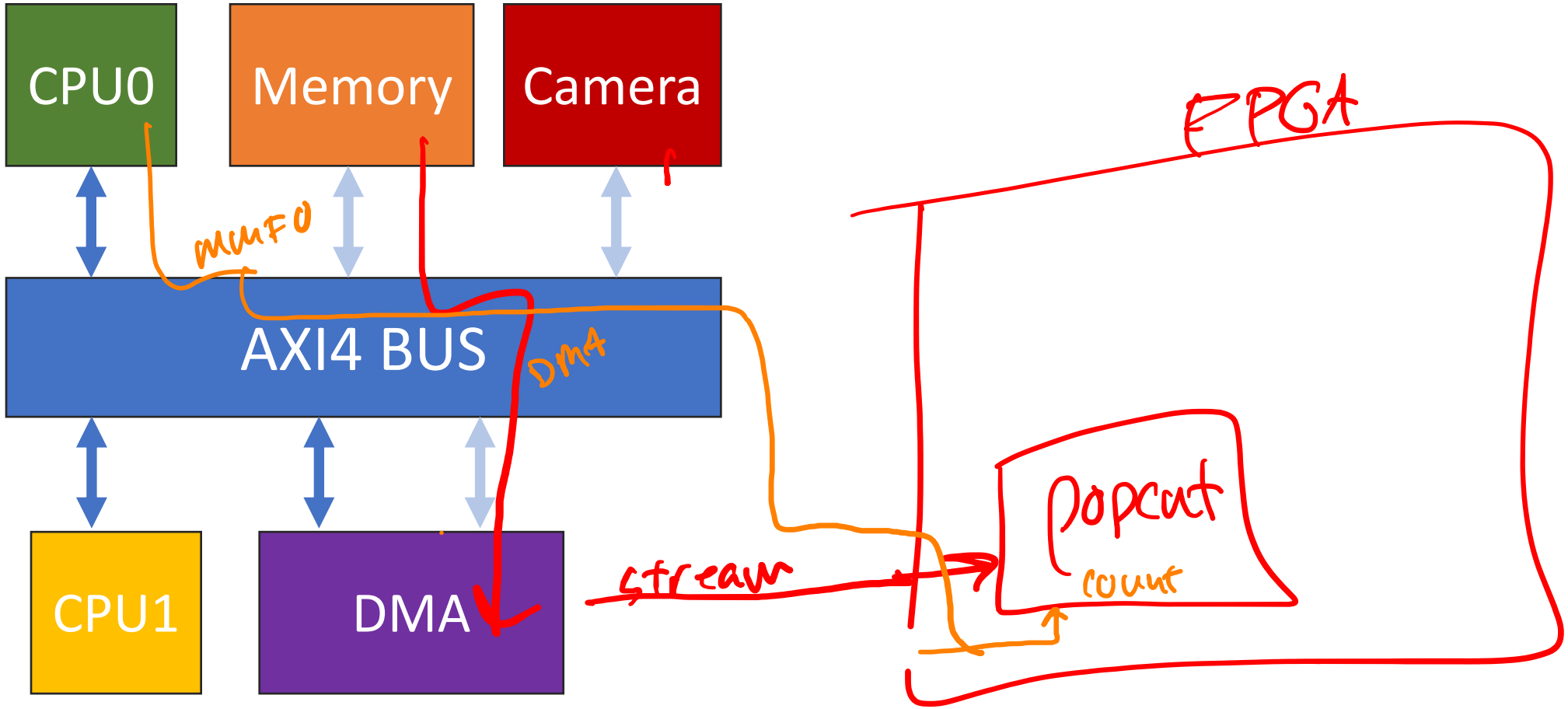
DMA for Popcount



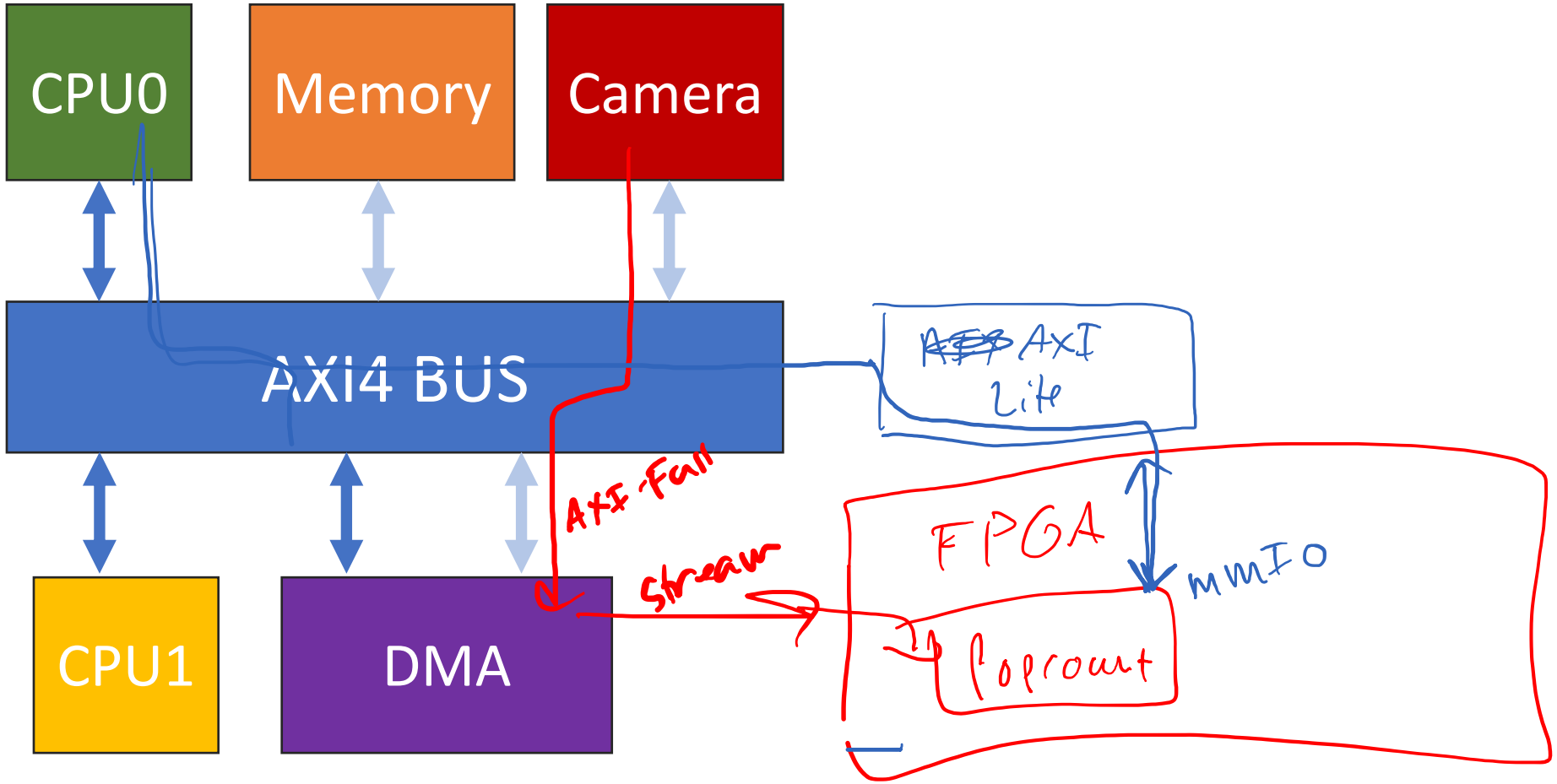
We don’t need the write portion.



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



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



P6 Example DMA

https://pynq.readthedocs.io/en/v2.6.1/pynq_libraries/dma.html

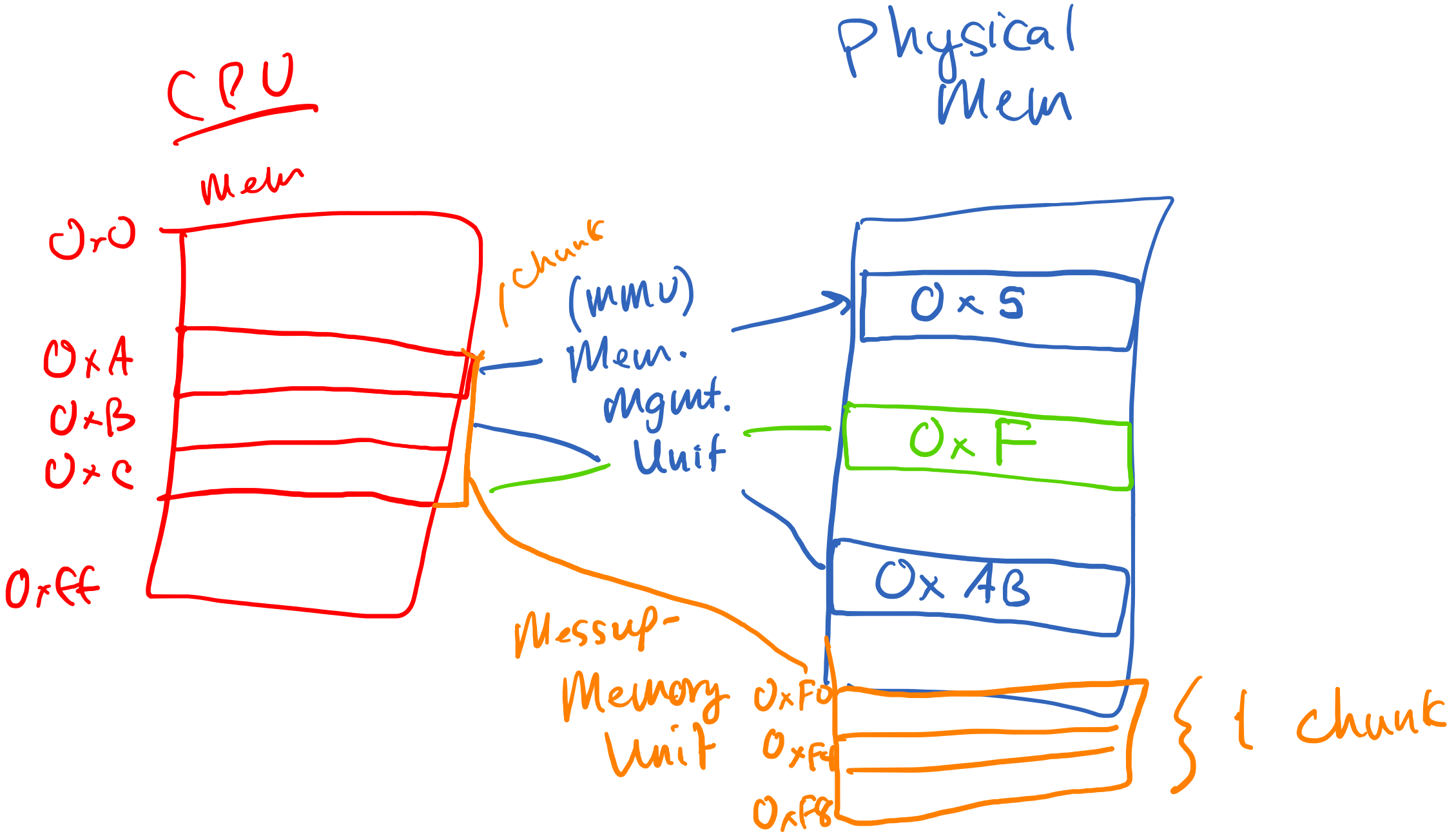
```
import numpy as np
from pynq import allocate
from pynq import Overlay

overlay = Overlay('example.bit')
dma = overlay.axi_dma

input_buffer = allocate(shape=(5,), dtype=np.uint32)
output_buffer = allocate(shape=(5,), dtype=np.uint32)
```

load ~~DMA~~ bitstream
& find dma in bitstream
special allocate

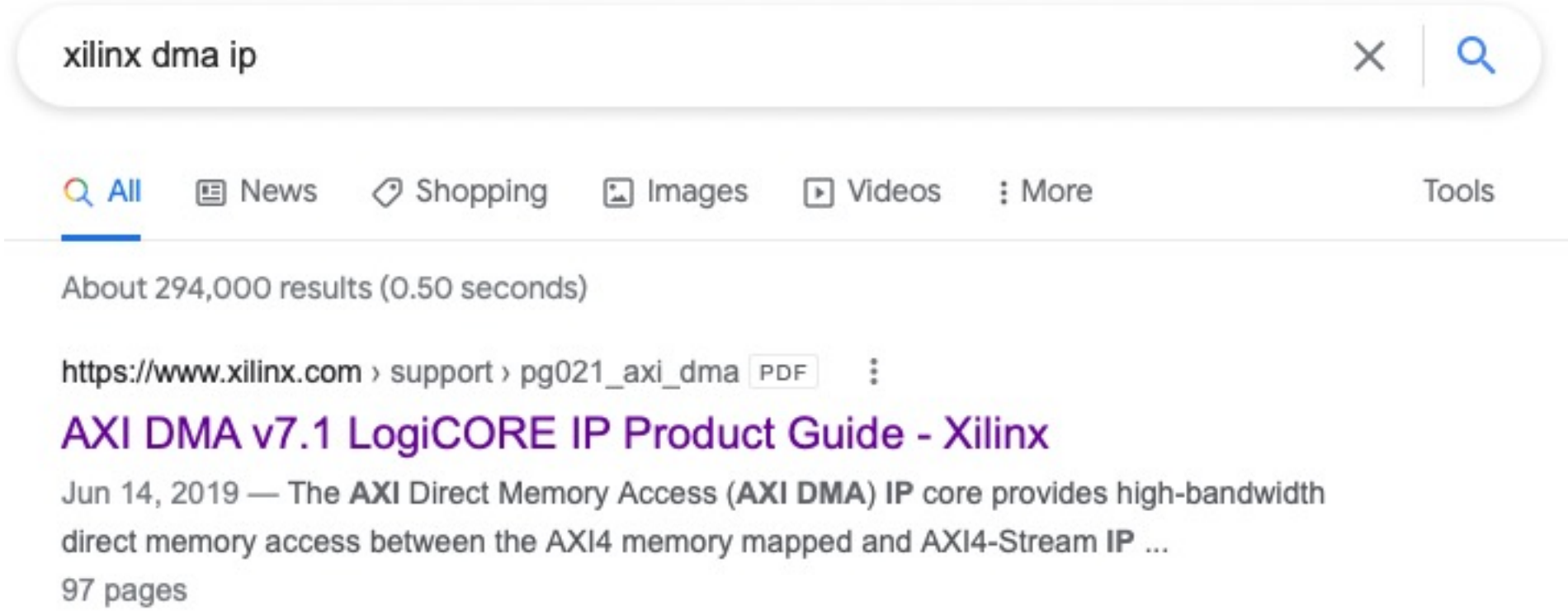
```
for i in range(5):
   input_buffer[i] = i
```

```
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
```

```
Output buffer will contain: [0 1 2 3 4]
```


CPU
Mem
0x0
0xA
0xB
0xC
0xFF
1 chunk
(mmu)
Mem.
Mgmt.
Unit
Physical
Mem
0x5
0xF
0xAB
Messup-
Memory
Unit
0xF0
0xF4
0xF8
} 1 chunk

P7 – DMA from C



https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

P7 – DMA from C

Programming Sequence

Direct Register Mode (Simple DMA)

P7 – DMA from C

0: turn on

1. Start the MM2S channel running by setting the run/stop bit to 1 (MM2S_DMACR.RS = 1). The halted bit (DMASR.Halted) should deassert indicating the MM2S channel is running.

P7 – DMA from C

1. Start the MM2S channel running by setting the run/stop bit to 1 (MM2S_DMACR.RS = 1). The halted bit (DMASR.Halted) should deassert indicating the MM2S channel is running.
2. Skip (No Interrupts)

P7 – DMA from C

1. Start the MM2S channel running by setting the run/stop bit to 1 (MM2S_DMACR.RS = 1). The halted bit (DMASR.Halted) should deassert indicating the MM2S channel is running.

2. Skip

3. Write a valid source address to the MM2S_SA register.

P7 – DMA from C

1. Start the MM2S channel running by setting the run/stop bit to 1 (MM2S_DMACR.RS = 1). The halted bit (DMASR.Halted) should deassert indicating the MM2S channel is running.

2. Skip

3. Write a valid source address to the MM2S_SA register.

4. Write the number of bytes to transfer in the MM2S_LENGTH register.

   The MM2S_LENGTH register must be written last.

P7 – DMA from C

1. Start the MM2S channel running by setting the run/stop bit to 1 (MM2S_DMACR.RS = 1). The halted bit (DMASR.Halted) should deassert indicating the MM2S channel is running.
2. Skip
3. Write a valid source address to the MM2S_SA register.
4. Write the number of bytes to transfer in the MM2S_LENGTH register.

   The MM2S_LENGTH register must be written last.

5. Poll MM2S_DMASR.Idle bit for completion

↳ while loop

Now on to Pipelining / Parallelization

P8+ Accelerate Machine Learning

- Goal: Accelerate reference neural network
- Harder, more open-ended projects

ML
221, 222
Neural Nets

Accelerate Machine Learning

- Given: Python starter code to classify digits
- Your Task: Accelerate it!
- How: Up to you!

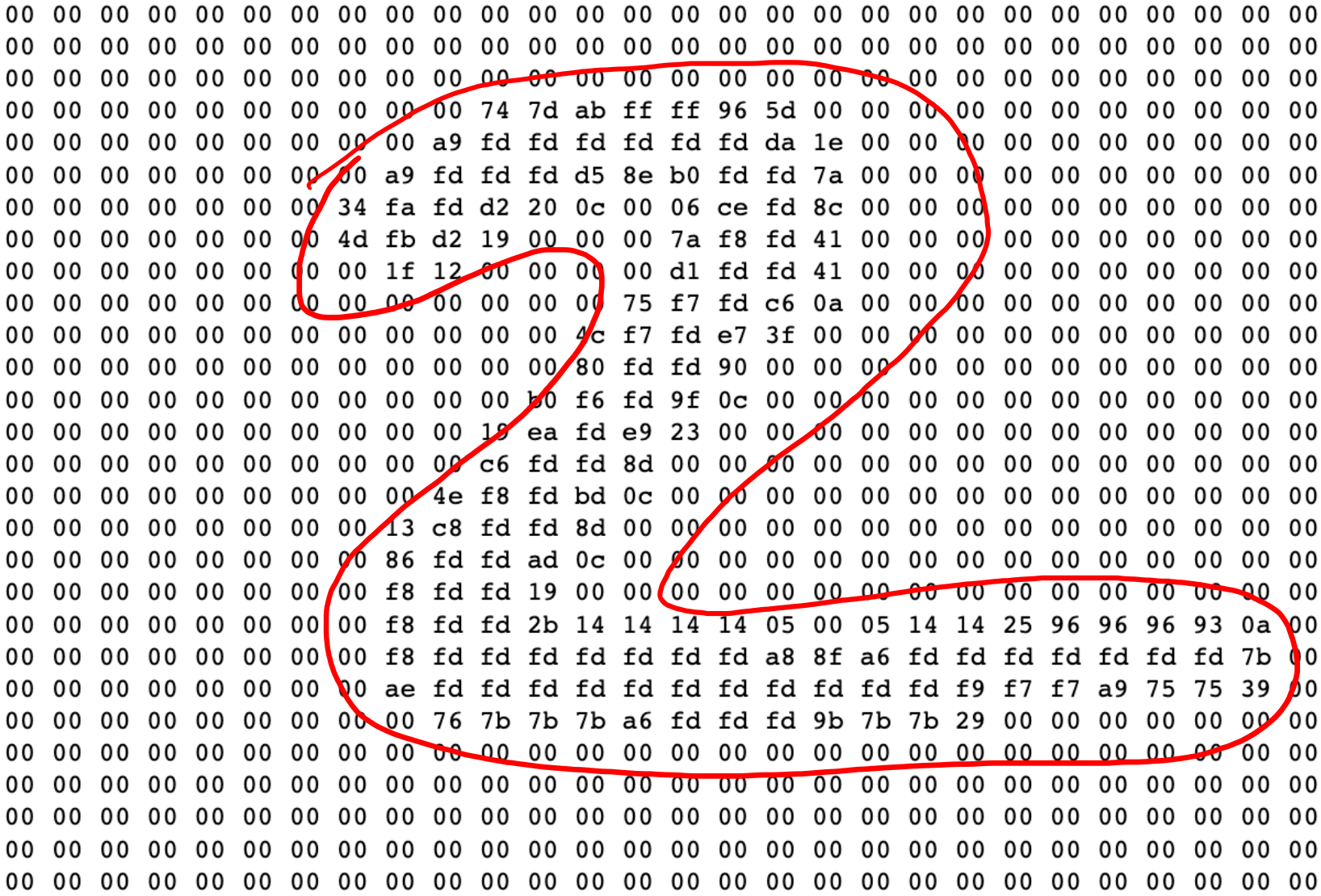
Grade: Based on overall speedup!

Project 8: Dot Product

- Given: Slow Dot-Product in Hardware
- Your Task: Accelerate it!
- How? Pipelining + Parallelism
- Why? Project X

What Number is this?

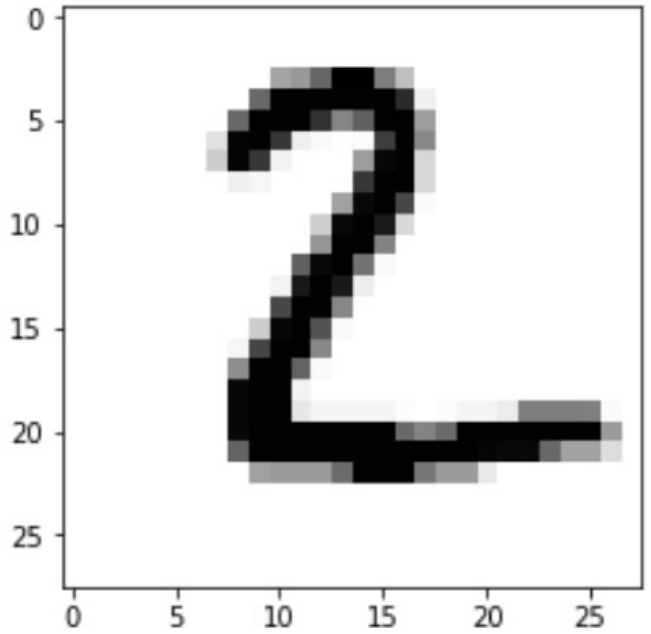
Raw data, in hex



Simple Neural Network

MNIST

```
================================
Index: 0
Image:
```

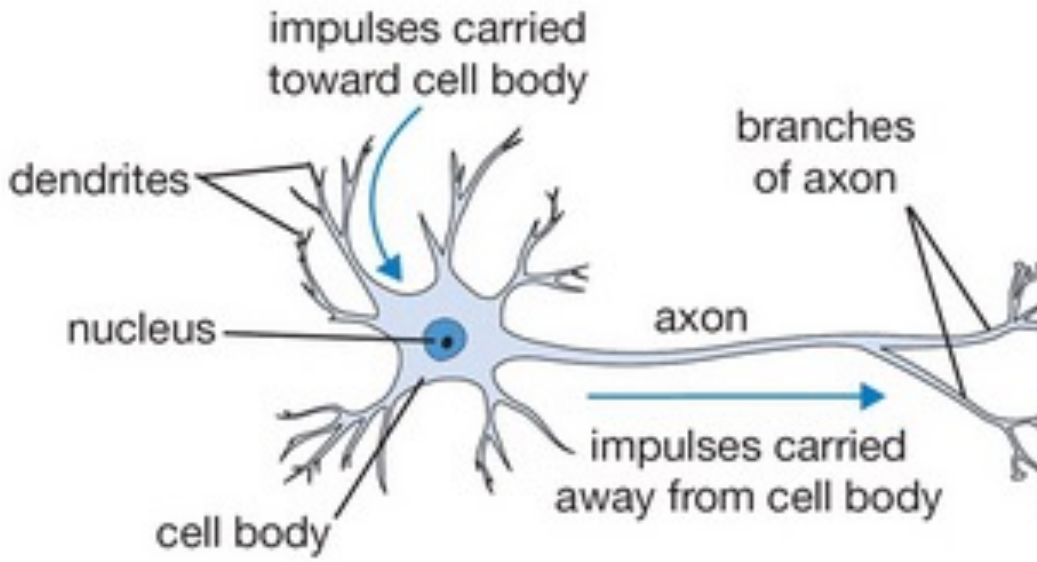



```
ML Classification Result: 2
Real Value: 2
Correct Result: True
================================
```

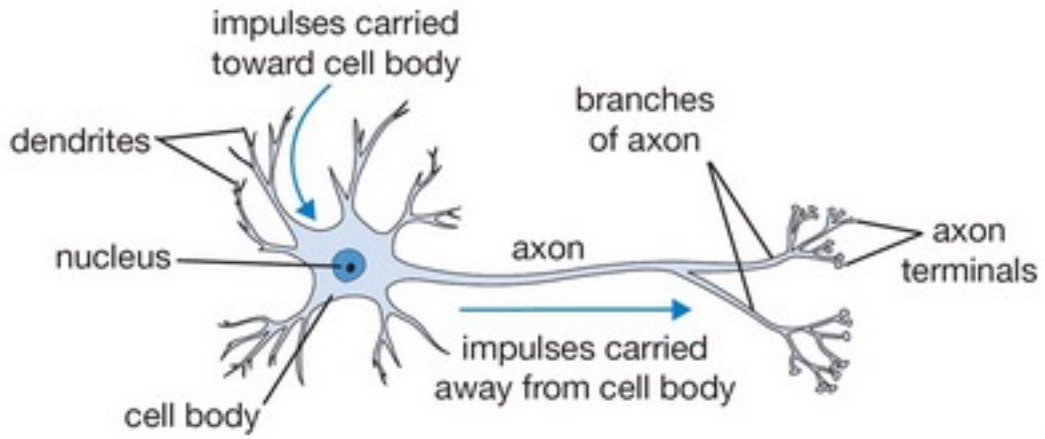
- Takes in image of number
- Returns integer value
- How? artificial neural network

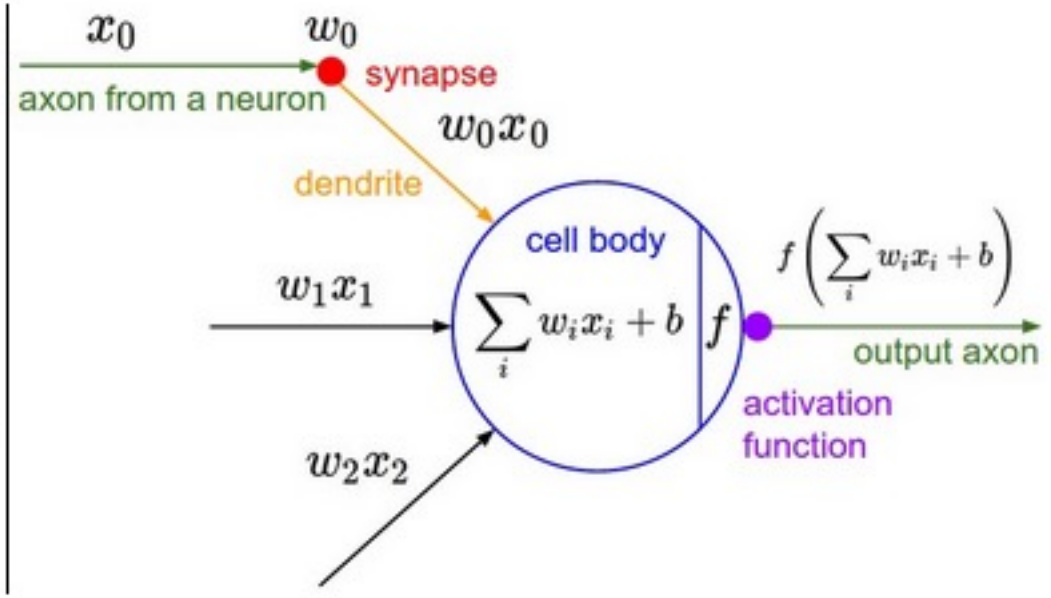
Fully-Connected Neural Networks

The Biological Neuron

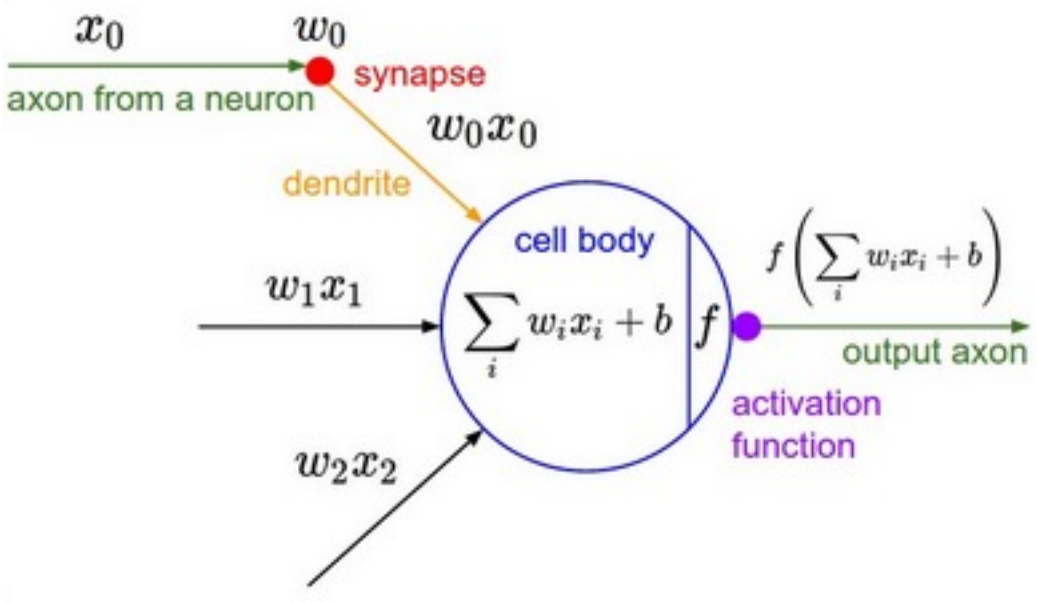


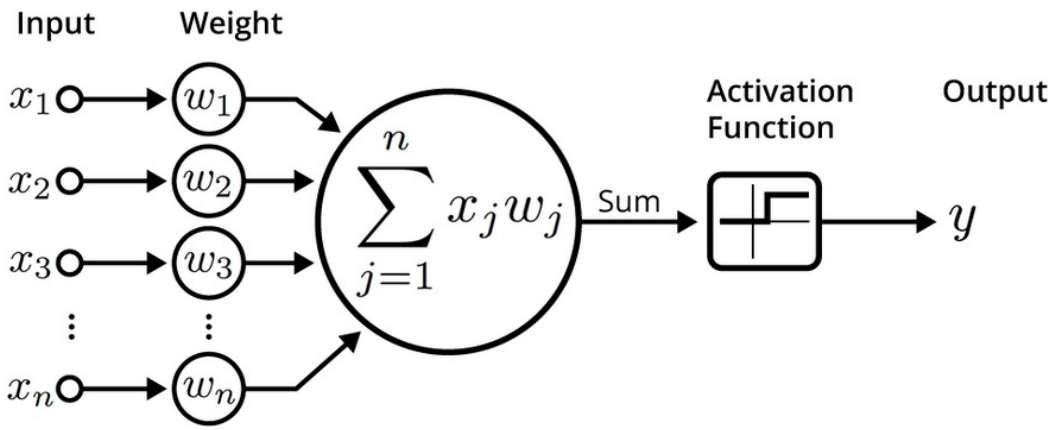
The Math Neuron





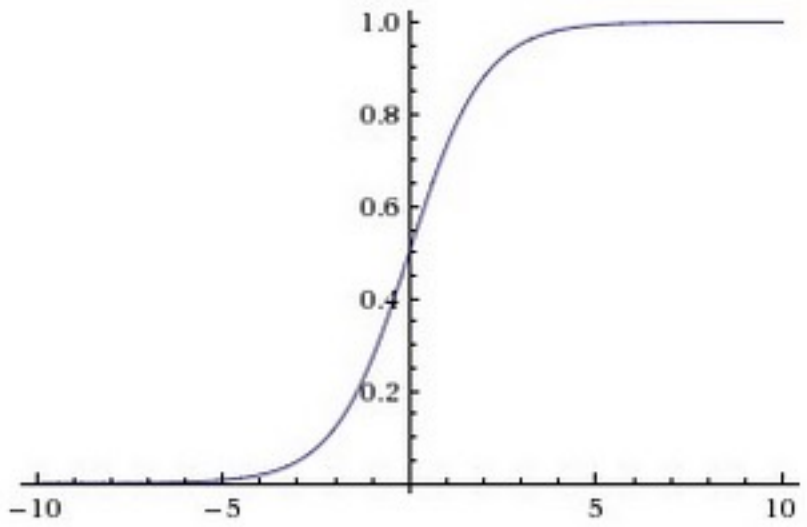
The Math Neuron





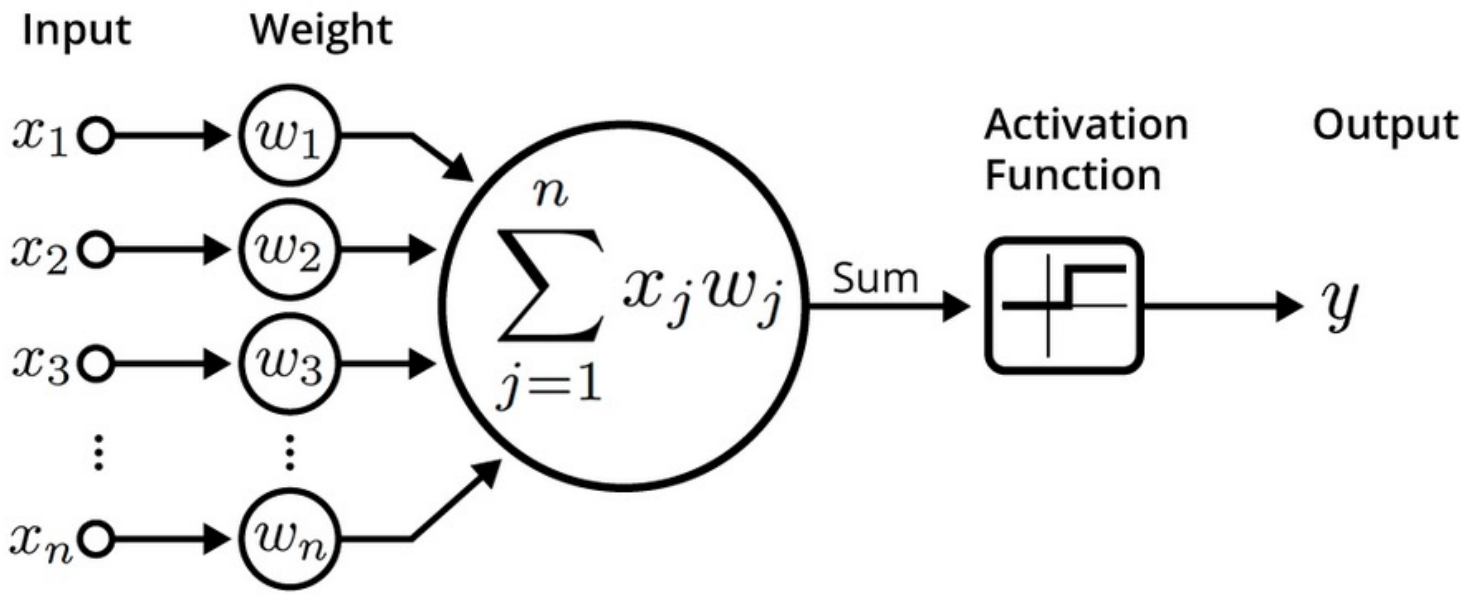
An illustration of an artificial neuron. Source: Becoming Human.

Activation Function: Sigmoid

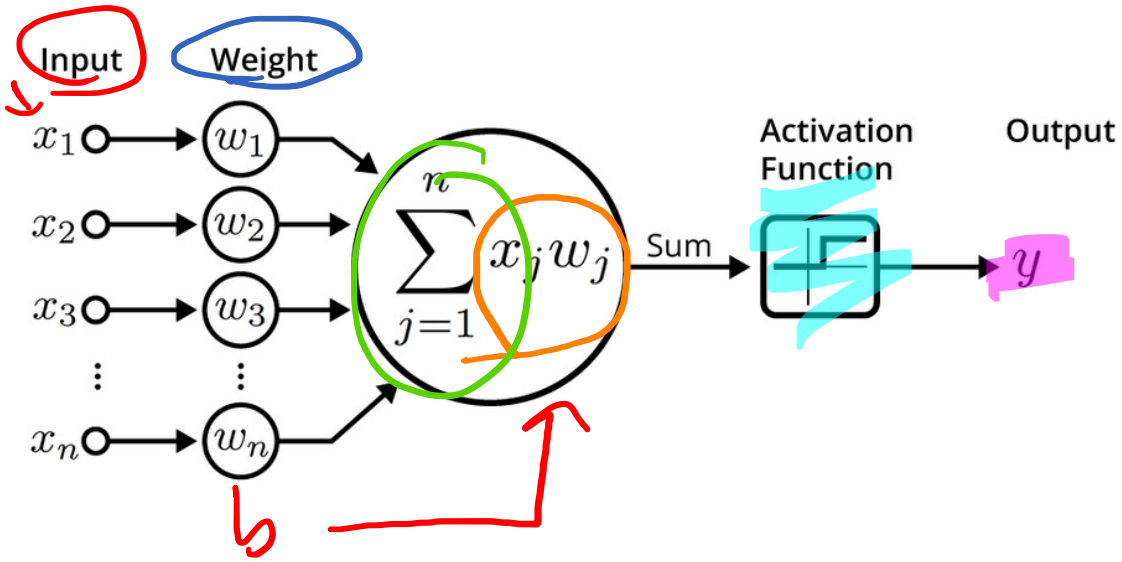


$$\sigma(x)=1/(1+e^{-x})$$

Adding Bias


Input
Weight
x_1
x_2
x_3
x_n
w_1
w_2
w_3
w_n
$\sum_{j=1}^{n} x_j w_j$
Sum
Activation Function
Output
y

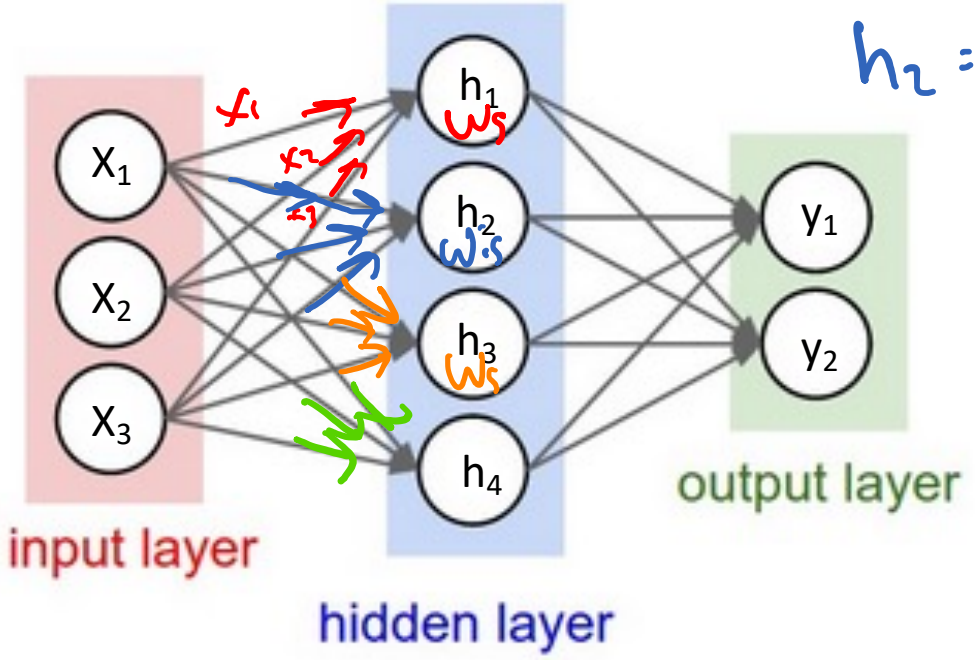
Python Neuron




```
class Neuron(object):
  # ...
  def forward(self, inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

Profile Code!

Neuron Networks



$$h_1 = f(x_1 \cdot w_{11} + x_2 \cdot w_{12} + x_3 \cdot w_{13} + b_1)$$

$$h_2 = f(x_1 \cdot w_{21} + x_2 \cdot w_{22} + x_3 \cdot w_{23} + b_2)$$

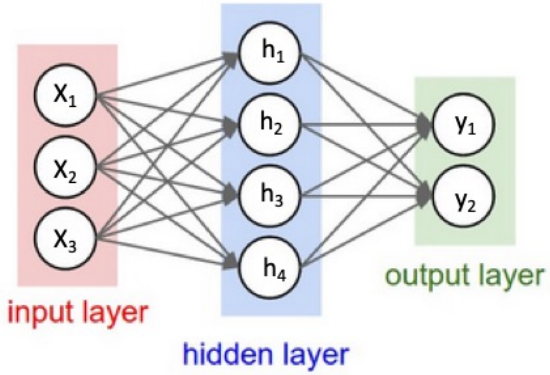
$$h_3 = f(x_1 \cdot w_{31} + x_2 \cdot w_{32} + x_3 \cdot w_{33} + b)$$

Input Layer: Just input values

Output Layers: No Activation Function

Why Dot Product?

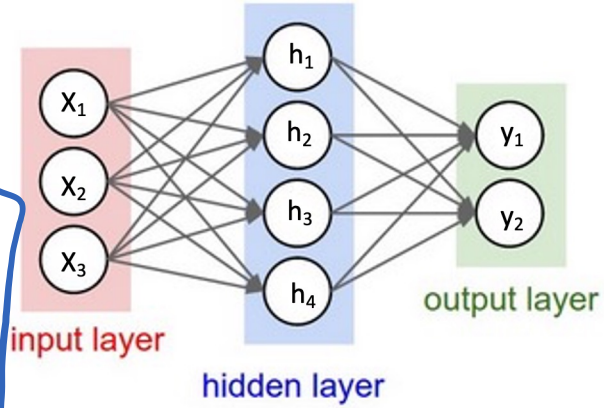
221




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Neuron Networks

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{21} & w_{31} & w_{41} \\ w_{12} & w_{22} & w_{32} & w_{42} \\ w_{13} & w_{23} & w_{33} & w_{43} \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 & h_4 \end{bmatrix}$$



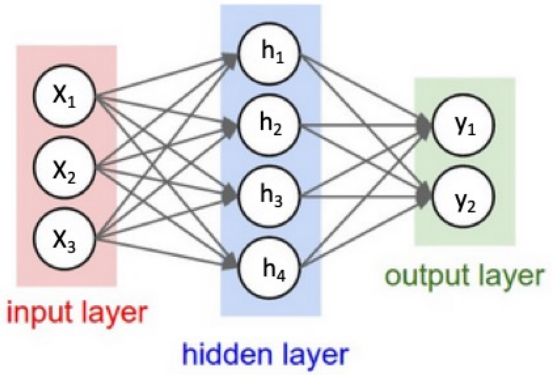
$$h_1 = x_1 \cdot w_{11} + x_2 \cdot w_{12} + x_3 \cdot w_{13}$$

$$h_2 = x_1 \cdot w_{21} + x_2 \cdot w_{22} + x_3 \cdot w_{23}$$

$$\vdots$$

Let's focus on just the Matrix for now.

$$[x_1 \; x_2 \; x_3] \cdot \begin{bmatrix} w_{11} & w_{21} & w_{31} & w_{41} \\ w_{12} & w_{22} & w_{32} & w_{42} \\ w_{13} & w_{23} & w_{33} & w_{43} \end{bmatrix}$$



$h_1 = x_1 \cdot w_{11} + x_2 \cdot w_{12} + x_3 \cdot w_{13}$

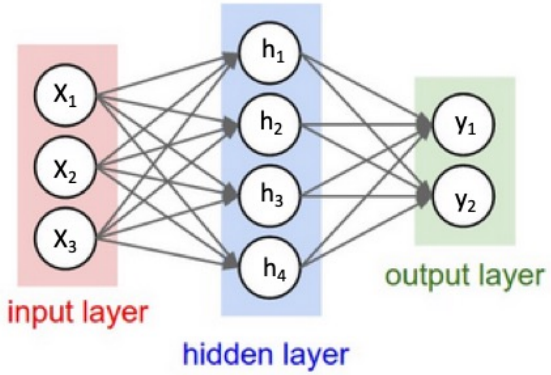
$h_2 = x_1 \cdot w_{21} + x_2 \cdot w_{22} + x_3 \cdot w_{23}$

$h_3 = x_1 \cdot w_{31} + x_2 \cdot w_{32} + x_3 \cdot w_{33}$

h_4 is similar...

true h_i = sigmoid(~ + b)

Matrix Multiplication (Dot Product)



Matrix Multiplication (Dot Product)

$$\begin{bmatrix} i_0 & i_1 \end{bmatrix} \times \begin{bmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \end{bmatrix} = \begin{bmatrix} o_0 & o_1 & o_2 \end{bmatrix}$$

$$o_0 = i_0 \cdot w_{00} + i_1 \cdot w_{01}$$

$$o_1 = i_0 \cdot w_{10} + i_1 \cdot w_{11}$$

$$o_2 = i_0 \cdot w_{20} + i_1 \cdot w_{21}$$

Matrix Multiplication (Dot Product)

FIXME: Reworked sides in Lec 17.

Matrix Multiplication (Dot Product)

inputs

$$\begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} =$$

weights

$$= \begin{bmatrix} (0.1 \times 1 + 0.2 \times 4) & (0.1 \times 2 + 0.2 \times 5) & (0.1 \times 3 + 0.2 \times 6) \end{bmatrix}$$

(Answer)

$$= \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$

Alternative Dot Computations

$$[0.1 \quad 0.2] \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = [0.9 \quad 1.2 \quad 1.5]$$

partial result

$$0.1 \cdot 1 \qquad 0.1 \cdot 2 \qquad 0.1 \cdot 3 = [0.1 \quad 0.2 \quad 0.3]$$

$$0.2 \cdot 4 \qquad 0.2 \cdot 5 \qquad 0.2 \cdot 6 = + [0.8 \quad 1.0 \quad 1.2]$$

$$[0.9 \quad 1.2 \quad 1.5]$$

Alternative Dot Computations

$$[0.1 \quad 0.2] \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = [0.9 \quad 1.2 \quad 1.5]$$

$$0.1 \cdot 1 \quad 0.1 \cdot 2 \quad 0.1 \cdot 3 = [0.1 \quad 0.2 \quad 0.3]$$

$$0.2 \cdot 4 \quad 0.2 \cdot 5 \quad 0.2 \cdot 6 = [0.8 \quad 1.0 \quad 1.2]$$

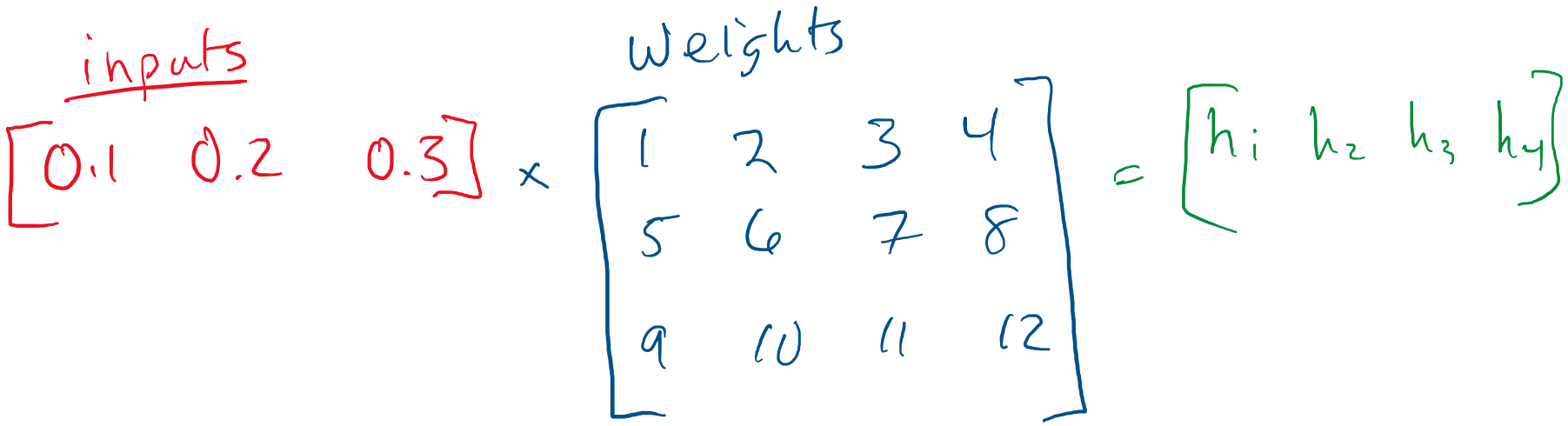
1 input @ a time?

$$\begin{array}{l} [0.1 \quad 0.2 \quad 0.3] \\ +[0.8 \quad 1.0 \quad 1.2] \\ \hline 0.9 \quad 1.2 \quad 1.5 \end{array}$$

Alternative Dot Computations

$$\begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$

Python Time




```
weights = np.array( [[1,2,3,4],[5,6,7,8],[9,10,11,12]], dtype=np.float32)
inputs = np.array([[0.1,0.2,0.3]], dtype=np.float32)
outputs = np.dot(inputs, weights)
```

```
Input                  .    Weights                        Output
[0.1 0.2 0.3]               [1. 2. 3. 4.]          =  [3.8000002 4.4       5.        5.6000004]
                            [5. 6. 7. 8.]
                            [ 9. 10. 11. 12.]
```

```
Input                    Weights                          Output
[[0.1 0.2 0.3]]    .     [1. 2. 3. 4.]          =  [3.8000002 4.4       5.        5.6000004]
                         [5. 6. 7. 8.]
                         [ 9. 10. 11. 12.]
```

Alternative Dot

```
Inputs (Shape):
 (1, 3)
Output (Shape):
 (1, 4)
Weights (Shape):
 (3, 4)
```

```python
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j] + weights[i][j] * inputs[i]
    return outs
```

```
Input                                Weights                          Output
[[0.1 0.2 0.3]]    .    [1. 2. 3. 4.]          =  [3.8000002 4.4       5.        5.6000004]
                        [5. 6. 7. 8.]
                        [ 9. 10. 11. 12.]
```

Alternative Dot

```
Inputs (Shape):
 (1, 3)
Output (Shape):
 (1, 4)
Weights (Shape):
 (3, 4)
```

```python
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j] + weights[i][j] * inputs[i]
    return outs
```

outs = [0 0 0 0]

i=0, j=0

outs = [0.1 0 0 0]

i=0, j=1

outs = [0.1 0+0.1·2 0 0]
= [0.1 0.2 0 0]

i=0, j=2

outs = [0.1 0.2 0.3 0]

i=0, j=3

outs = [0.1 0.2 0.3 0.4]

i=1

outs = 0.1+0.2·5, 0.2+0.2·6, 0.3+0.2·7, 0.4+0.2·8

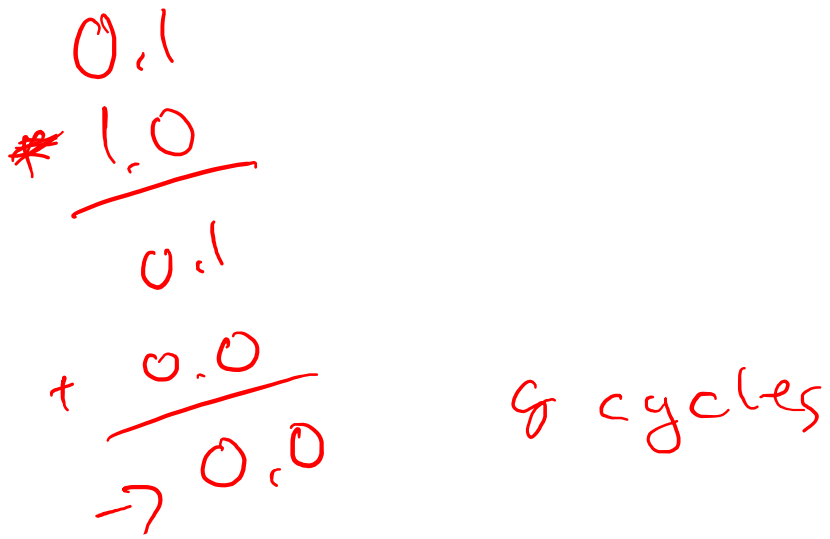
= [1.1 1.4 1.7 2.0]

i=2

outs = [3.8 4.4 5 5.6]

Floating-Point Multiply-Accumulate (FMAC)

- Math: a * b + c ← 8 cycles



Floating-Point math takes 8 cycles.

- Floating-Point is complicated.
- How do we work around an 8 cycle latency?
- Pipelining!

Pipelining

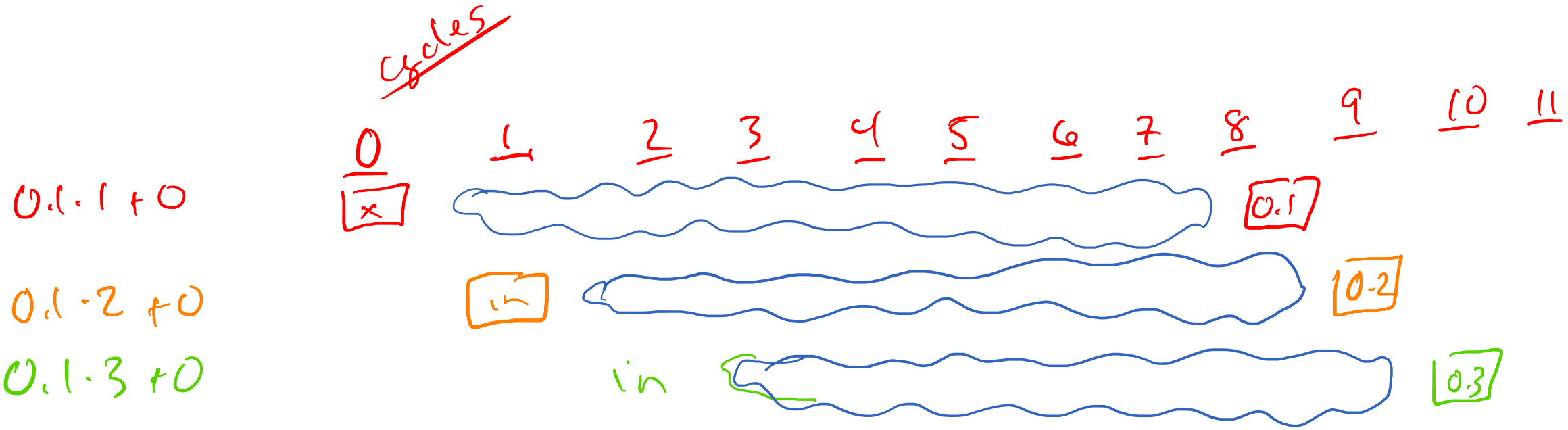
- FMAC takes 8 cycles for 1 value
- But can accept a new value every cycle.

cycles for value

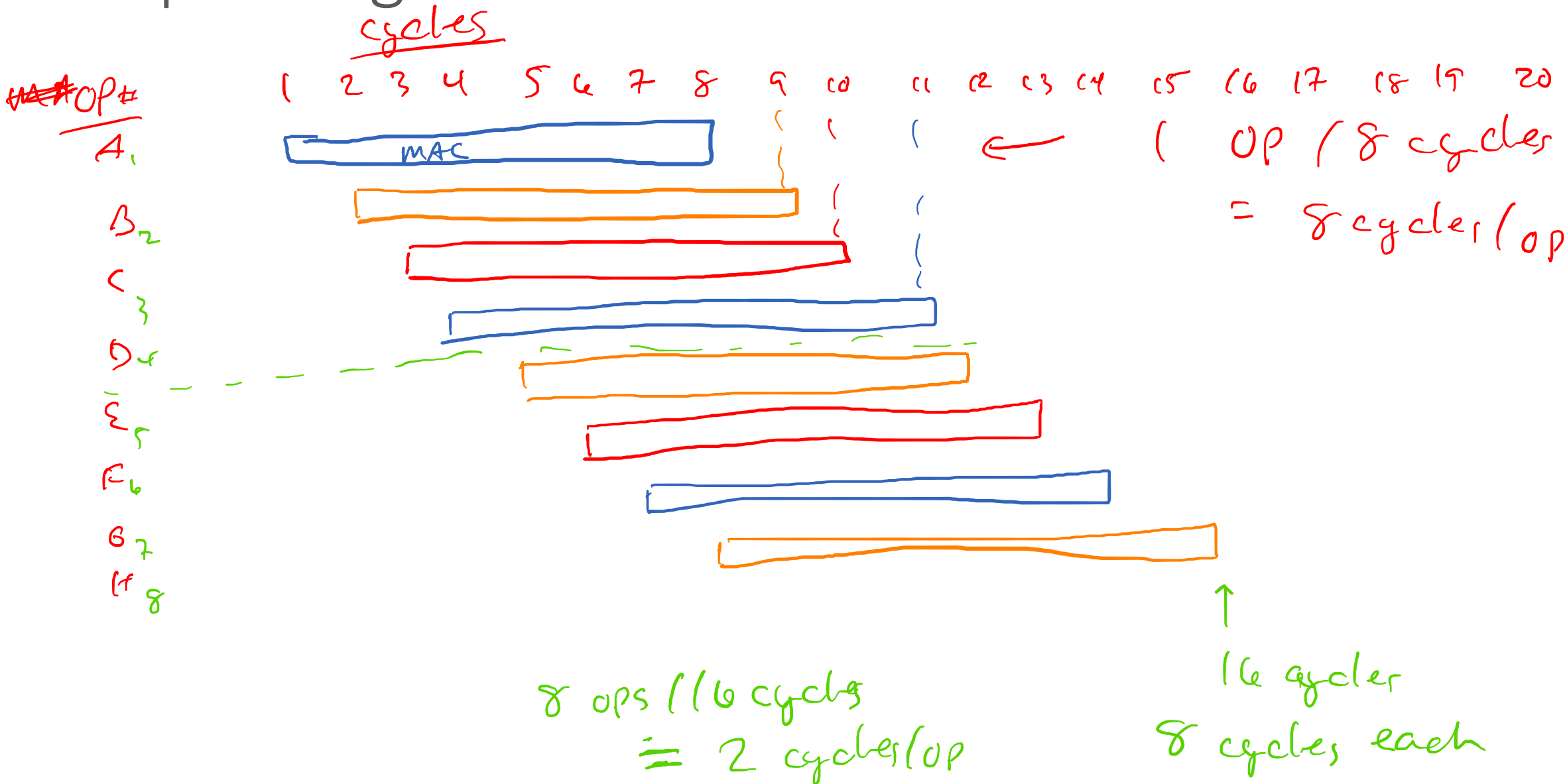
- Latency: 8 cycles / value
- Throughput: 1 value / cycle

values per cycle

Pipelining



Pipelining


cycles
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
OP#
A1
MAC
B2
C3
D4
E5
F6
G7
H8
1 OP / 8 cycles
= 8 cycles / OP
8 ops / 16 cycles
= 2 cycles / OP
16 cycles
8 cycles each

Next Time: More Hardware Parallelism

- more pipelining
- parallel hardware
  - parallelizing dot product.