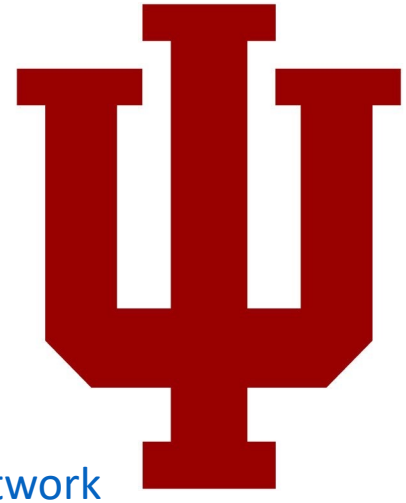
18: Hardware Acceleration I

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

- P4 is due TOMORROW (not today)
  - Bitstream / hwh files added
  - Password: ‘iuxilinx’
- P5 is out
- Exam in 3 weeks

P4: volatile

P5: 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

Example DMA

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

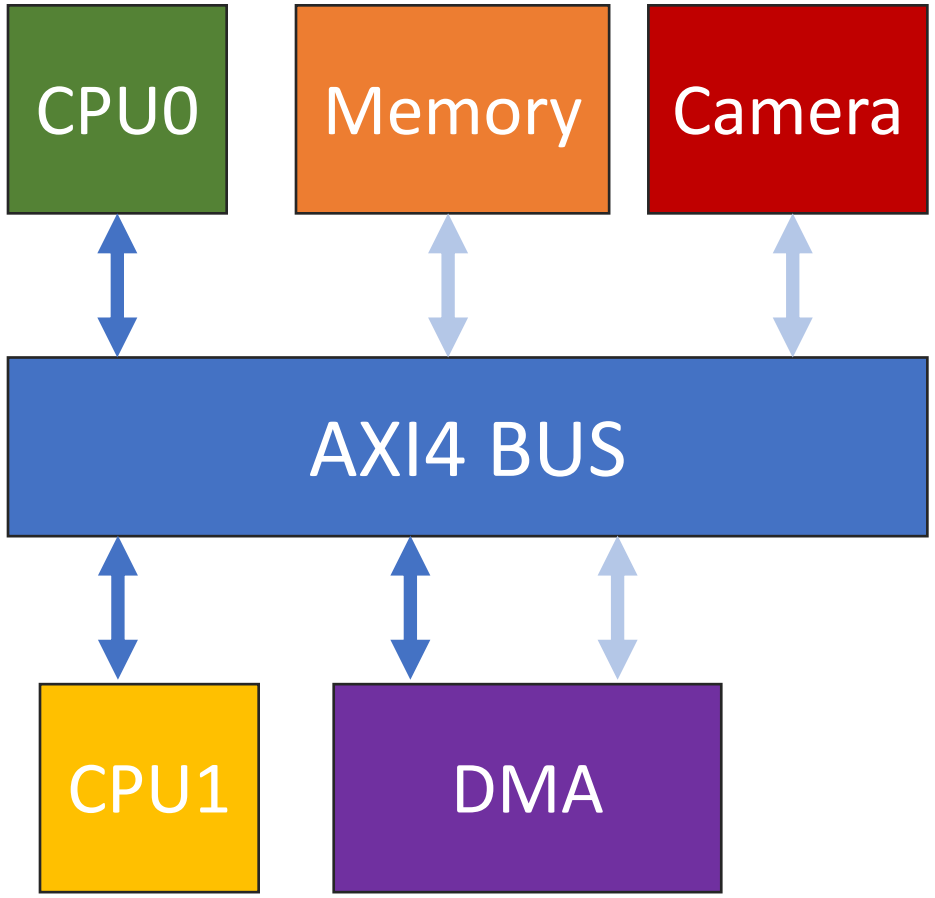
```
import numpy as np
from pynq import allocate
from pynq import Overlay

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

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

```
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]
```

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

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)

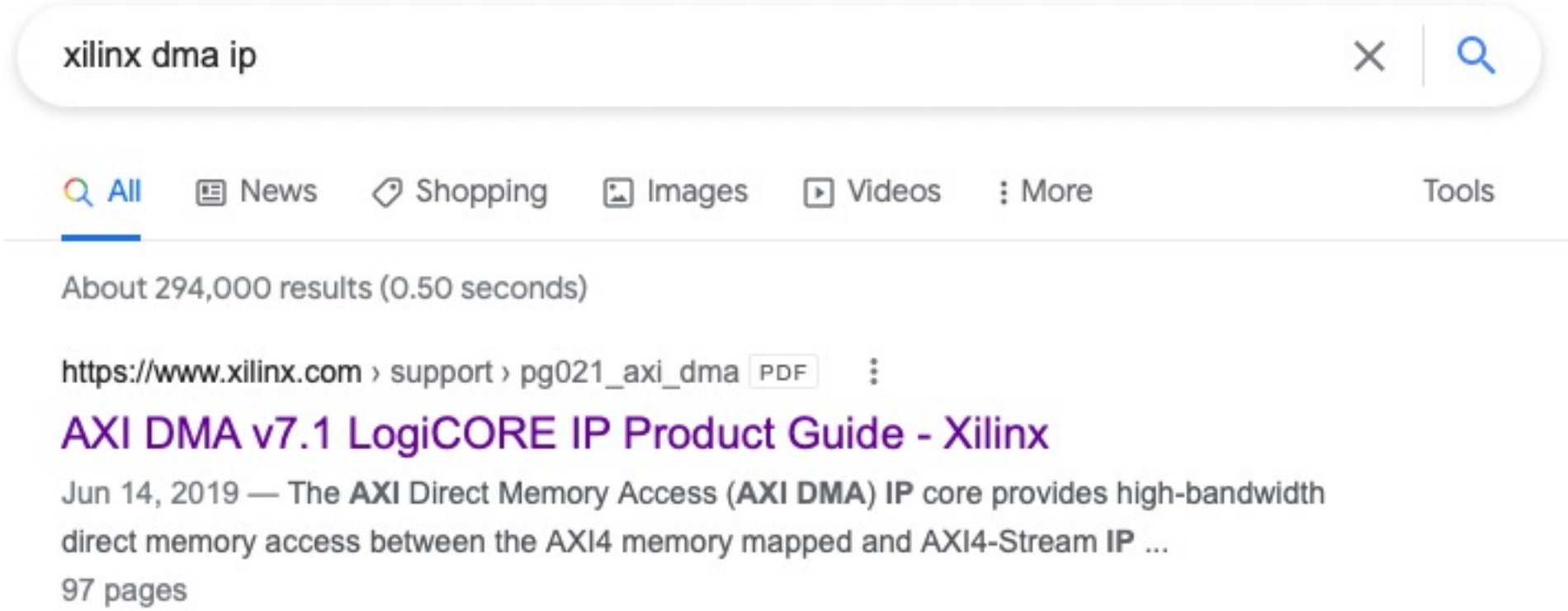
1-3

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){;}
}
```

P6 – DMA from C



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

P6 – DMA from C

Programming Sequence

Direct Register Mode (Simple DMA)

P6 – 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.

P6 – 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

P6 – 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.

P6 – 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.

P6 – 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

P7+ Accelerate Machine Learning

- Goal: Accelerate reference neural network
- Harder, more open-ended projects
- Groups of 2 allowed.

Accelerate Machine Learning

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

Grade: Based on overall speedup!

Project 6: Dot Product

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

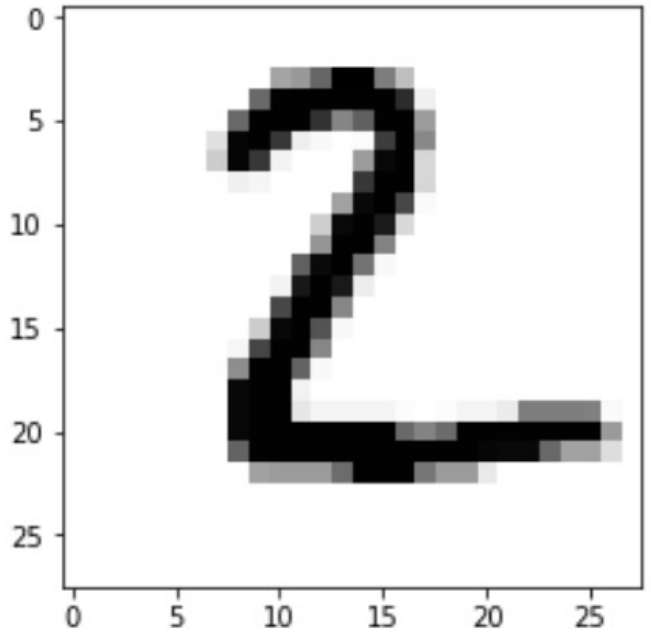
What Number is this?

Raw data, in hex

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 74 7d ab ff ff 96 5d 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 a9 fd fd fd fd fd fd da 1e 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 a9 fd fd fd d5 8e b0 fd fd 7a 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 34 fa fd d2 20 0c 00 06 ce fd 8c 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 4d fb d2 19 00 00 00 7a f8 fd 41 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 1f 12 00 00 00 00 d1 fd fd 41 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 75 f7 fd c6 0a 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 4c f7 fd e7 3f 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 80 fd fd 90 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 b0 f6 fd 9f 0c 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 19 ea fd e9 23 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 c6 fd fd 8d 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 4e f8 fd bd 0c 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 13 c8 fd fd 8d 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 86 fd fd ad 0c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 f8 fd fd 19 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 f8 fd fd 2b 14 14 14 14 05 00 05 14 14 25 96 96 96 93 0a 00
00 00 00 00 00 00 00 00 f8 fd fd fd fd fd fd fd a8 8f a6 fd fd fd fd fd fd fd 7b 00
00 00 00 00 00 00 00 00 ae fd fd fd fd fd fd fd fd fd fd fd f9 f7 f7 a9 75 75 39 00
00 00 00 00 00 00 00 00 00 76 7b 7b 7b a6 fd fd fd 9b 7b 7b 29 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Simple Neural Network

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

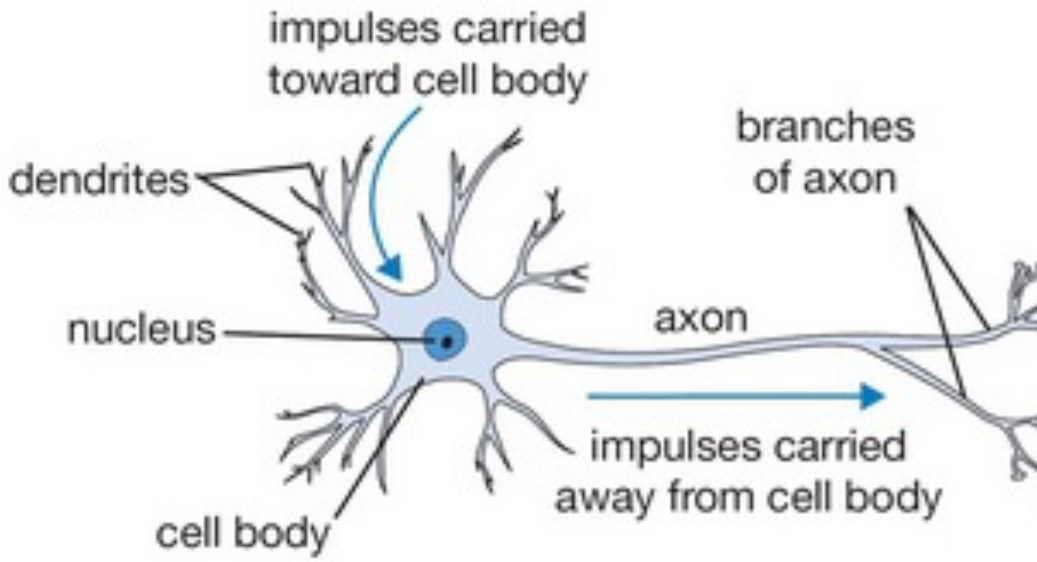



```
ML Classification Result: 2
Real Value: 2
Correct Result: True
================================
```

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

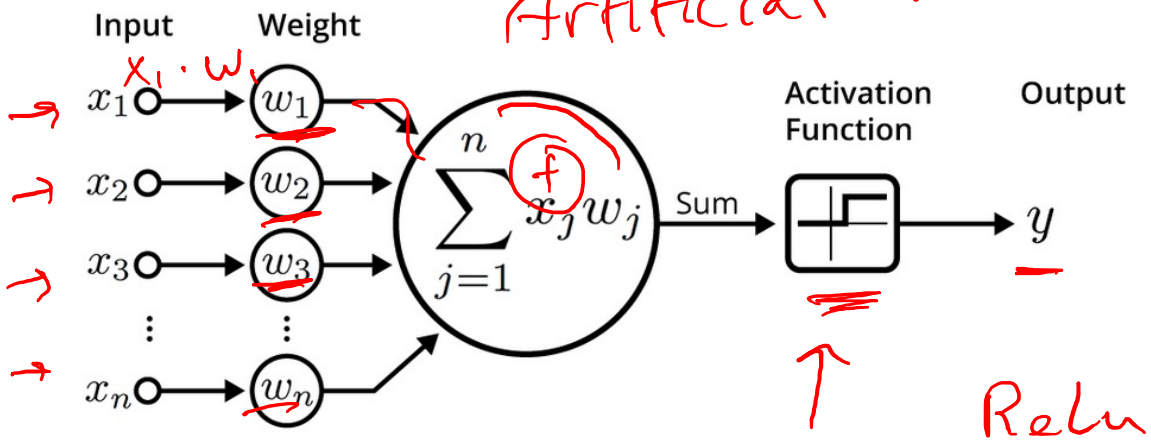
Fully-Connected Neural Networks

The Biological Neuron

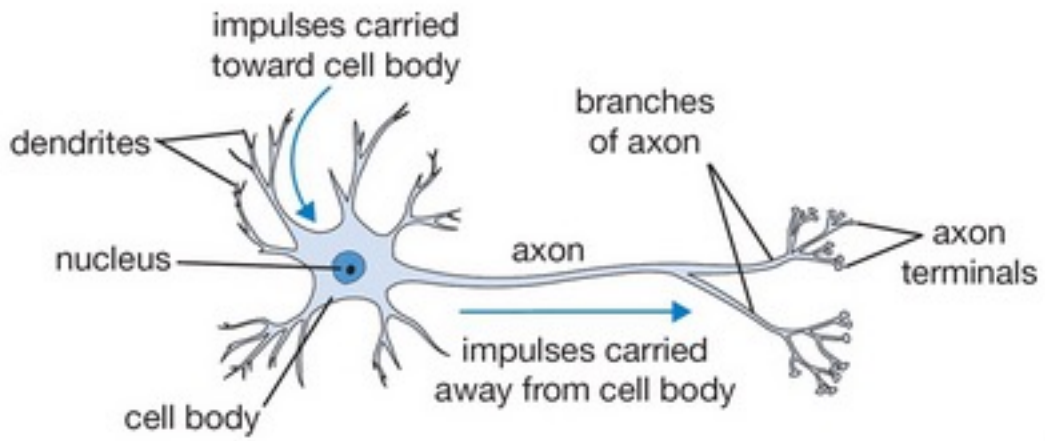


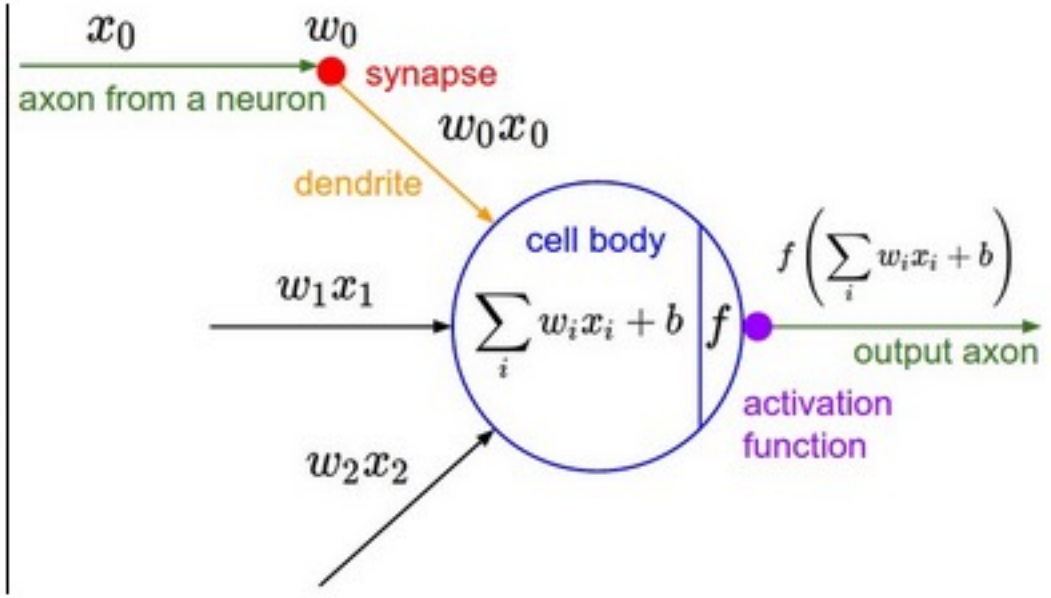
The Math Neuron

The Math Neuron

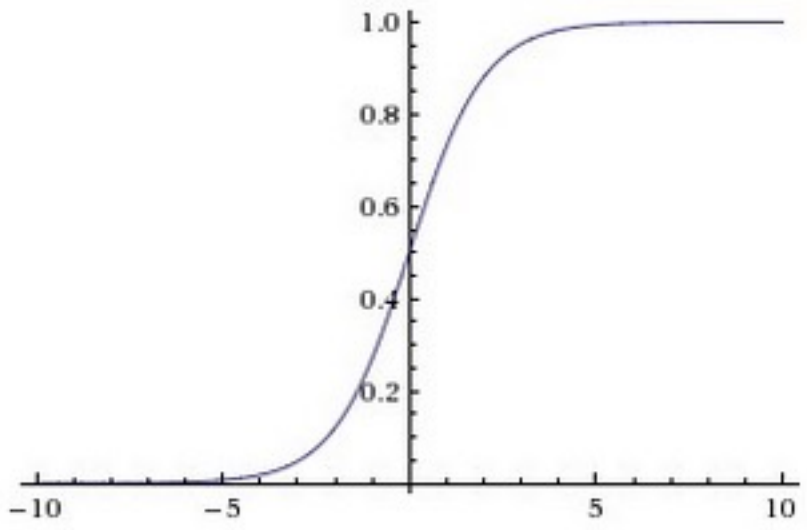


An illustration of an artificial neuron. Source: Becoming Human.





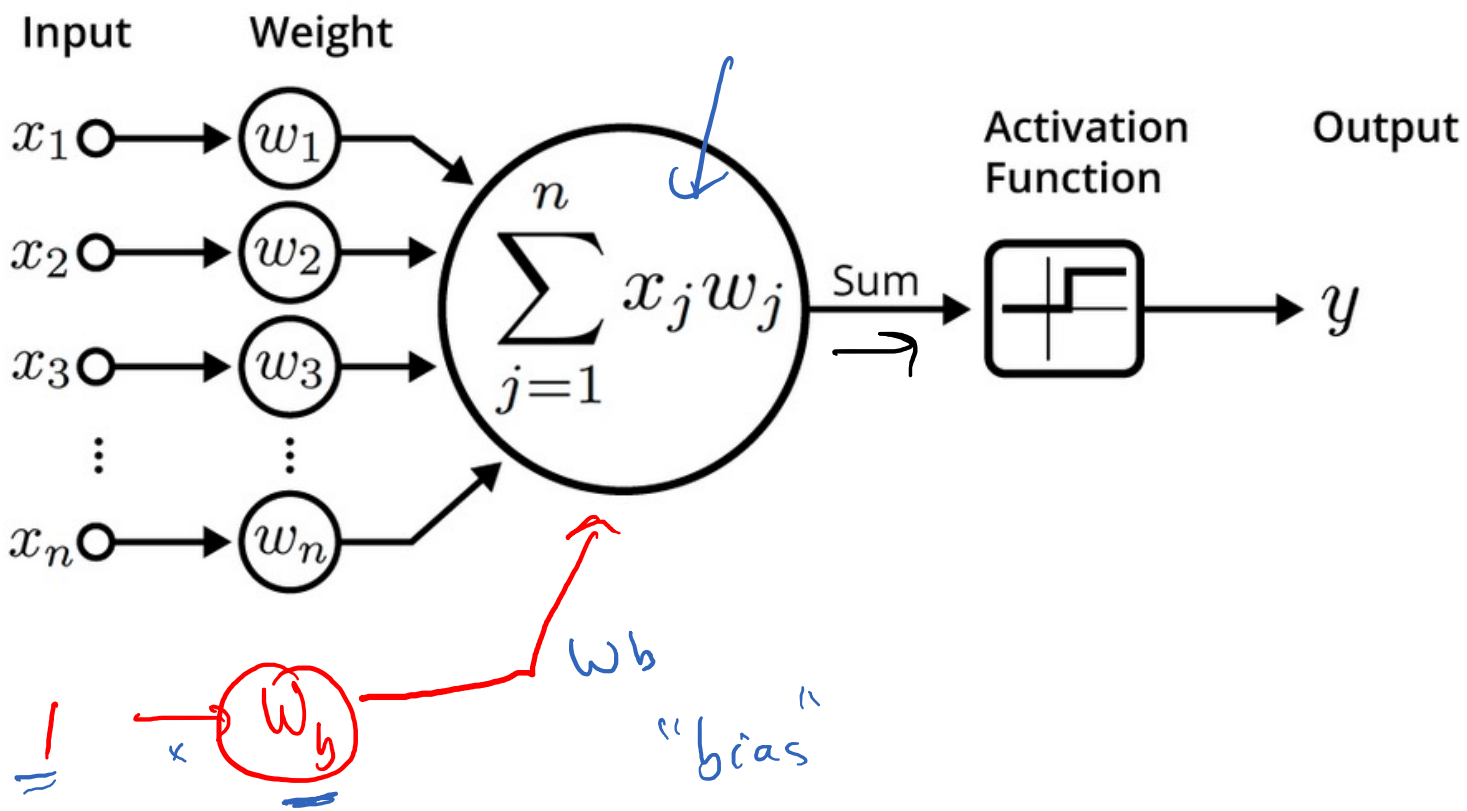
Activation Function: Sigmoid



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

Adding Bias

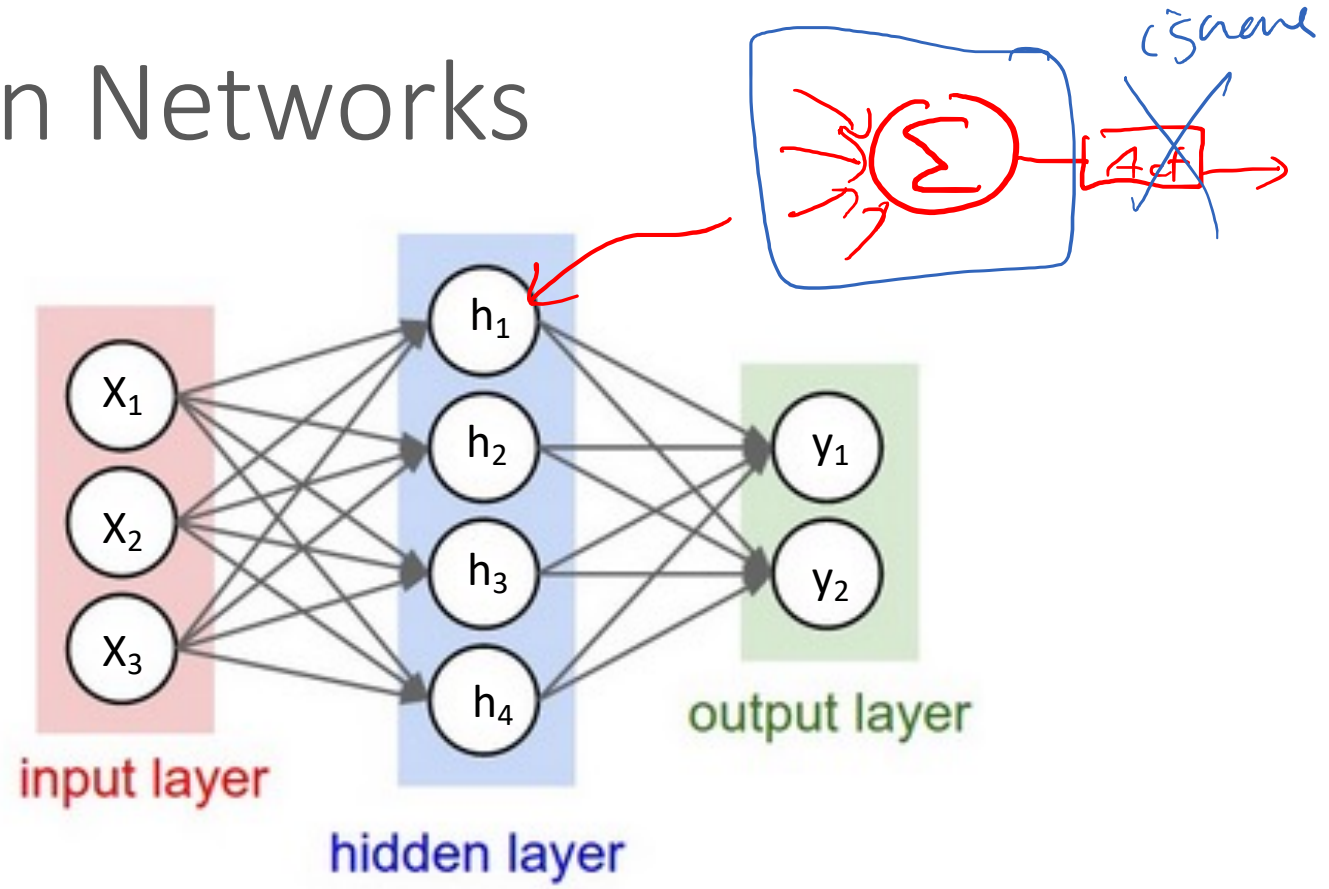
$$\sum_{j=1}^{n} x_j w_j + b \quad (\text{or } w_b)$$



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
```

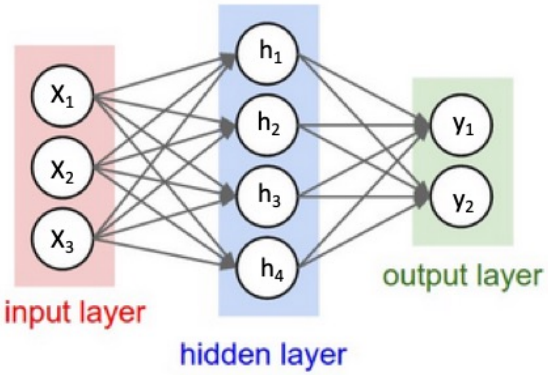
Multi-Neuron Networks



Input Layer: Just input values

Output Layers: No Activation Function

Let’s focus on just the Matrix for now.



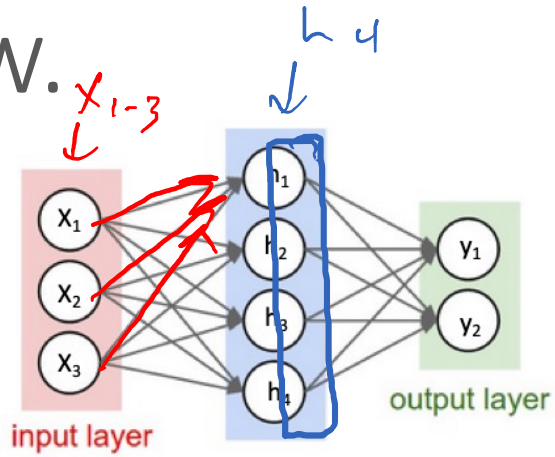
vector dot product Matrix Multiplication

Let's focus on just the Matrix for now.

inputs

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

weights



outputs

$$\begin{bmatrix} h_1 & h_2 & h_3 & h_4 \end{bmatrix}$$

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

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

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

$$h_4 = x_1 \cdot w_{41} + x_2 \cdot w_{42} + x_3 \cdot w_{43}$$

Matrix Multiplication (Dot Product)

$$\underbrace{\begin{bmatrix} i_0 & i_1 \end{bmatrix}}_{\text{inputs}} \times \underbrace{\begin{bmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \end{bmatrix}}_{\text{weights}} = \underbrace{\begin{bmatrix} o_0 & o_1 & o_2 \end{bmatrix}}_{\text{outputs}}$$

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

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

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

Matrix Multiplication (Dot Product)

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

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

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

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

Matrix Multiplication (Dot Product)

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

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

Alternative Dot Computations

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

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

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

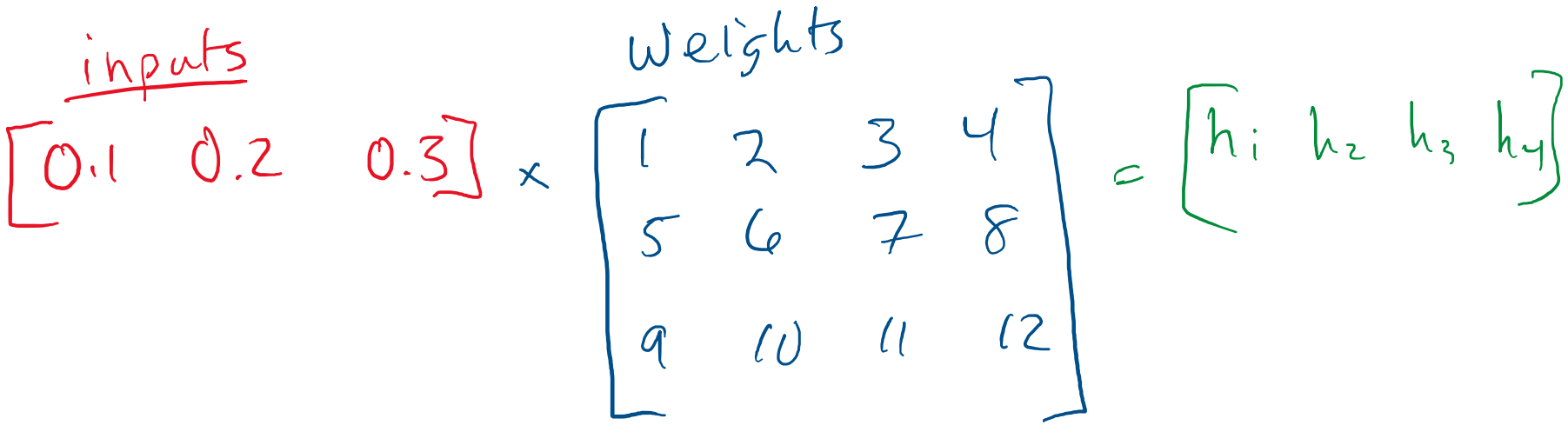
1 input @ a time?

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

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)
```

```
# 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

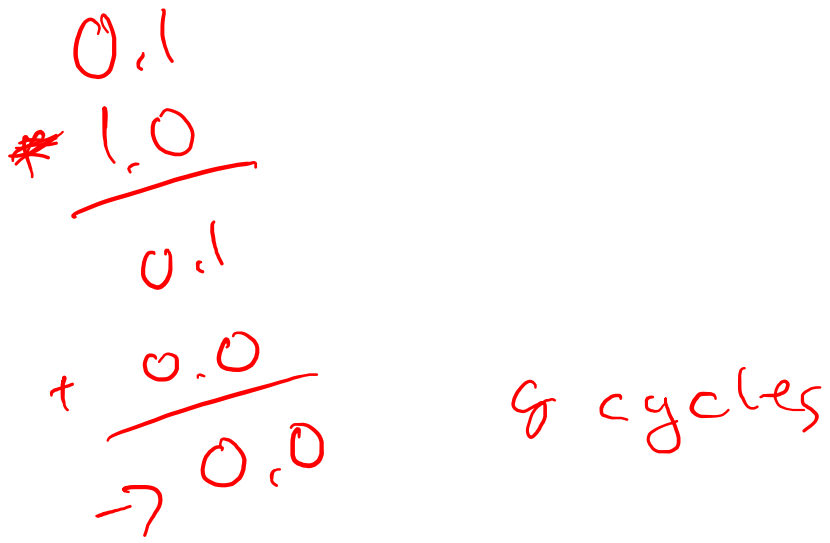
= [1.1 1.4 1.7 2.0]

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.

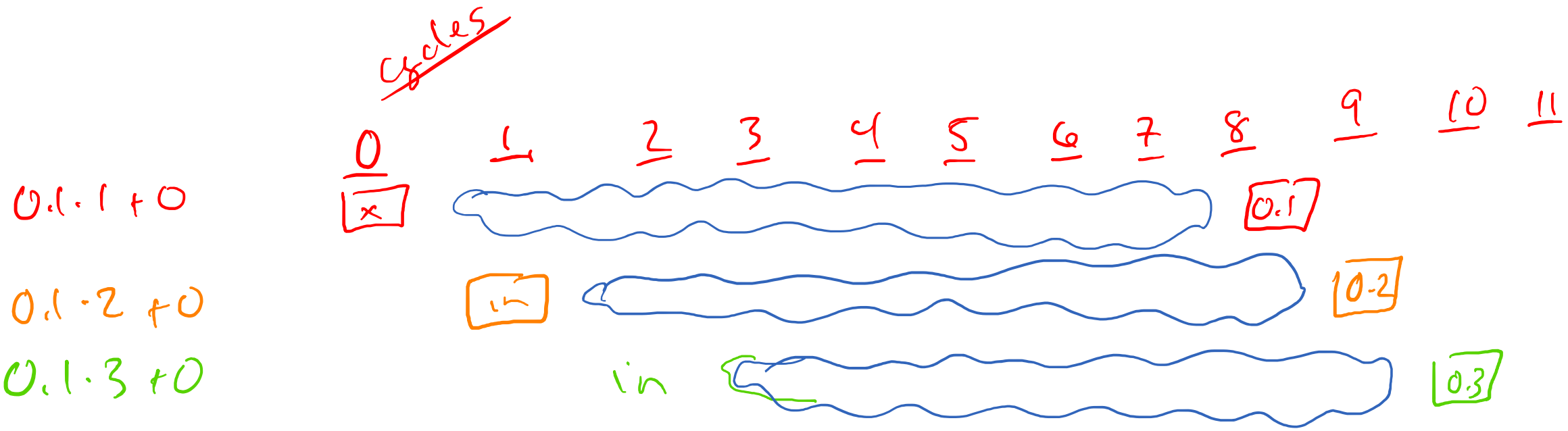
- Floating-Point is complicated.
- How do we work around an 8 cycle latency?
- Pipelining!

Pipelining

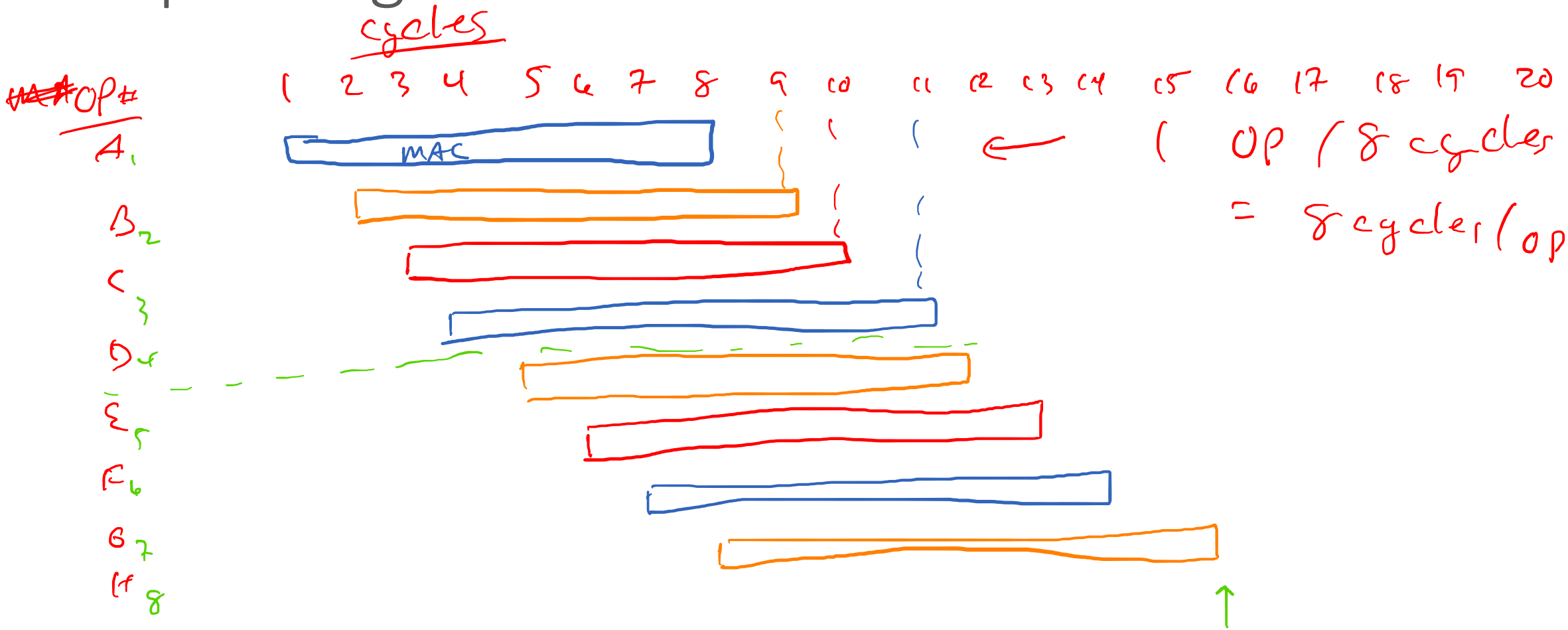
- FMAC takes 8 cycles for 1 value
- But can accept a new value every cycle.

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

Pipelining



Pipelining


cycles
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
MAC
1 OP / 8 cycles
= 8 cycles / OP
16 cycles
8 cycles each
8 ops / 16 cycles
= 2 cycles / OP

Next Time: More Hardware Parallelism

- more pipelining
- parallel hardware
  - parallelizing dot product.

18: Hardware Acceleration I

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University