

# 18: Parallelism I

ENGR 315: Hardware/Software CoDesign

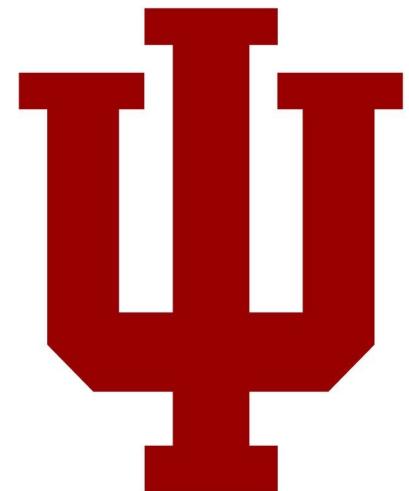
Andrew Lukefahr

Indiana University

Some material taken from:

[https://github.com/trekhleb/homemade-machine-learning/tree/master/homemade/neural\\_network](https://github.com/trekhleb/homemade-machine-learning/tree/master/homemade/neural_network)

<http://cs231n.github.io/neural-networks-1/>



# Announcements

- P7: Due Friday.
  - New SD Card / Linux Image
- Exam: Nov 7th
- P8: Coming Soon.

# P7 – DMA from C

A screenshot of a search results page from a web browser. The search bar at the top contains the query "xilinx dma ip". Below the search bar are navigation links: "All" (highlighted in blue), "News", "Shopping", "Images", "Videos", "More", and "Tools". A status message indicates "About 294,000 results (0.50 seconds)". The first result is a link to the "AXI DMA v7.1 LogiCORE IP Product Guide - Xilinx" document, which was published on Jun 14, 2019. The document provides high-bandwidth direct memory access between AXI4 memory mapped and AXI4-Stream IP cores, spanning 97 pages. The URL for the document is [https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_dma/v7\\_1/pg021\\_axi\\_dma.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf).

[https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_dma/v7\\_1/pg021\\_axi\\_dma.pdf](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)**

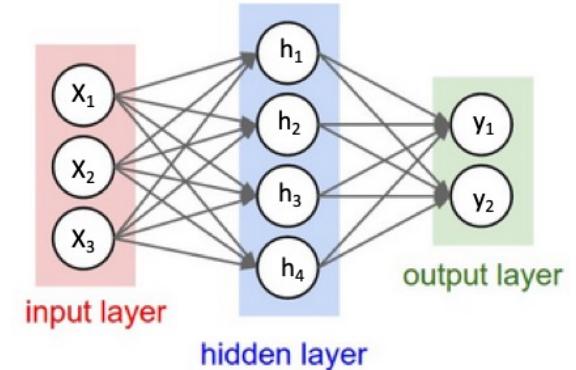
# P7 – DMA from C

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. Wait until MM2S\_DMASR.Idle==1 for completion

# P8+ are part of the “Design Challenge”

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

# Why focus on Dot Product?



```
# 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)
```

# Matrix Multiplication (Dot Product)

inputs

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

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

(Answer)

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

# Floating-Point Multiply-Accumulate (FMAC)

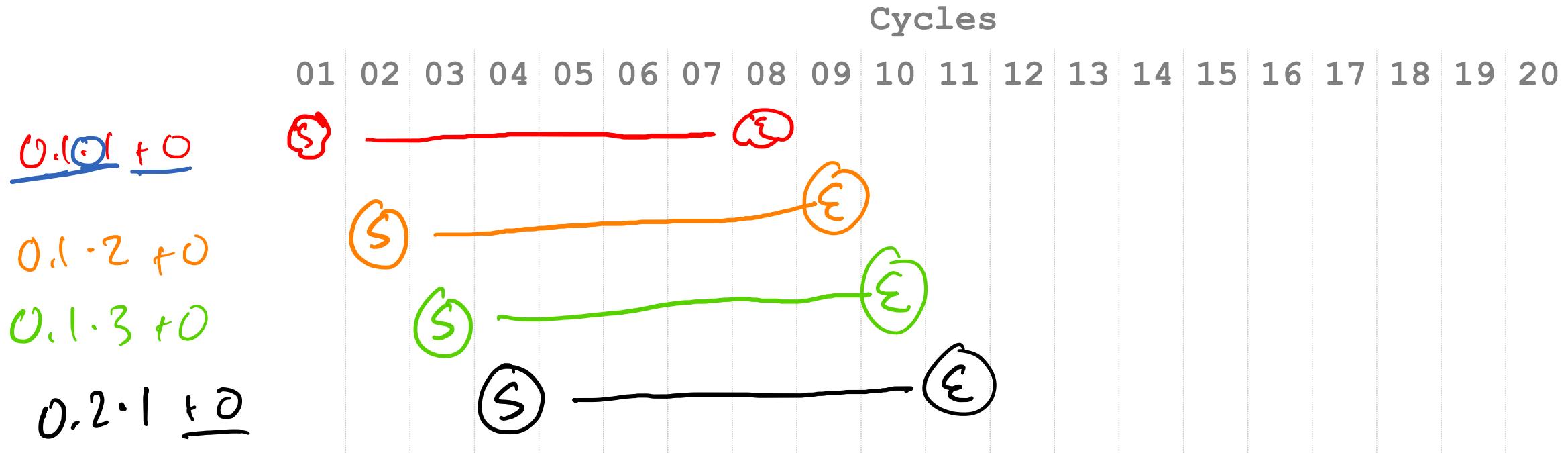
- Math:  $a * b + c$

# Floating-Point math takes 8 cycles.

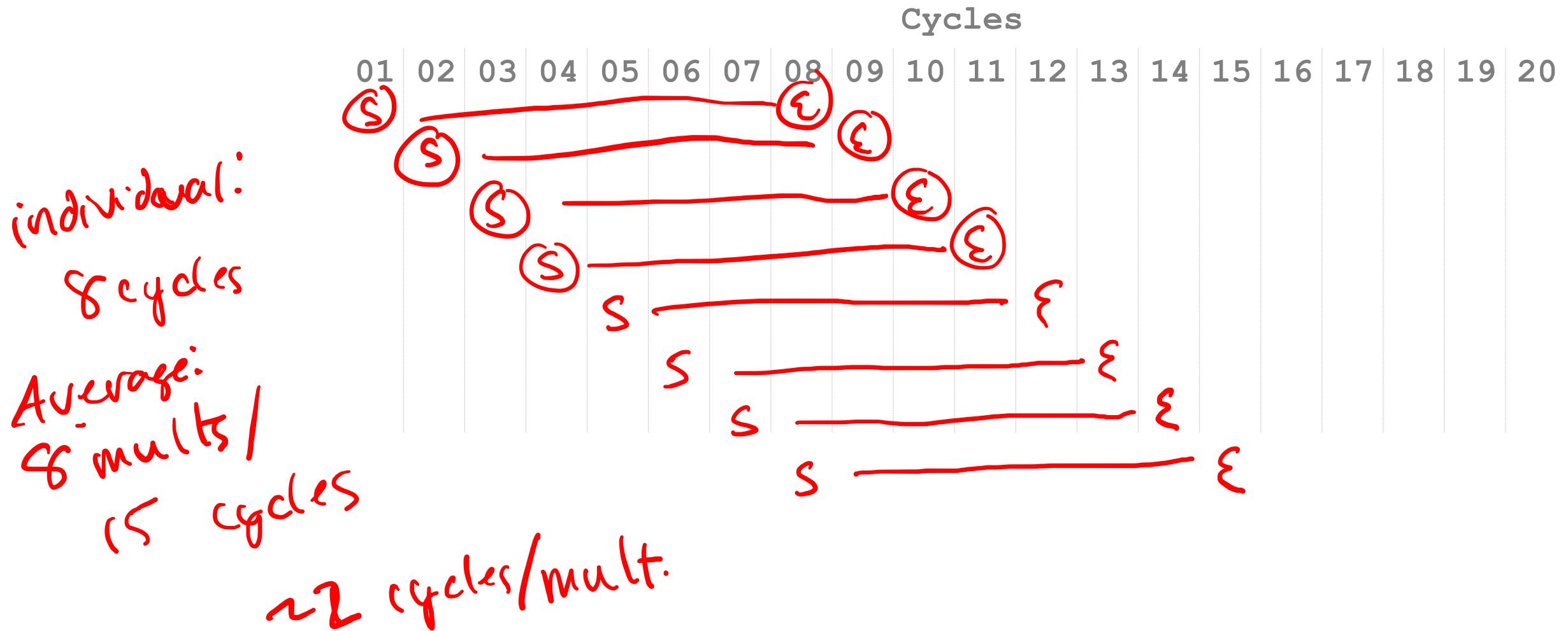
- Floating-Point is complicated.
- 8 cycles of complicated.
- How do we work around an 8 cycle latency?
- Pipelining!

# FMAC Pipelining

0.1 0.2 [ 123 ]  
[ 456 ]



# FMAC Pipelining



# Latency vs. Throughput

- **Latency:** How long does an individual operation take to complete?

individual mult: 8 cycles

- **Throughput:** How many operations can you complete per second (or per cycle)?

Average output per cycles:

~ 1 mult / cycle

# Multiply-Accumulate Dot Computations

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

round 1

$$\begin{bmatrix} 0.1 \cdot 1 & 0.1 \cdot 2 & 0.1 \cdot 3 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0.1 & 0.2 & 0.3 \end{bmatrix}$$

round 2

$$0.2 \cdot 4 & 0.2 \cdot 5 & 0.2 \cdot 6 + \begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} = \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$

# Multiply-Accumulate Dot Computations

Step 0:

$$\begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$
$$= \begin{bmatrix} 0^{\sim} & 0^{\sim} & 0^{\sim} \end{bmatrix}$$
$$\begin{array}{c} 0.1 \cdot 1 + 0 \\ 0.1 \cdot 2 + 0 \\ 0.1 \cdot 3 + 0 \end{array} \rightarrow \boxed{[0.1]} \quad \begin{array}{c} 0.2 \cdot 4 + 0 \\ 0.2 \cdot 5 + 0.2 \\ 0.2 \cdot 6 + 0.3 \end{array} \rightarrow \boxed{[0.9 \quad 1.2 \quad 1.5]}$$

Input  
[[0.1 0.2 0.3]]

Weights  
[1. 2. 3. 4.]  
[5. 6. 7. 8.]  
[ 9. 10. 11. 12.]

Output  
= [3.8000002 4.4  
5. 5.6000004]  
[0.1 0.2 0.3 0.4]

## Dependencies

$$0.1 \cdot 1 + 0$$

$$0.1 \cdot 2 + 0$$

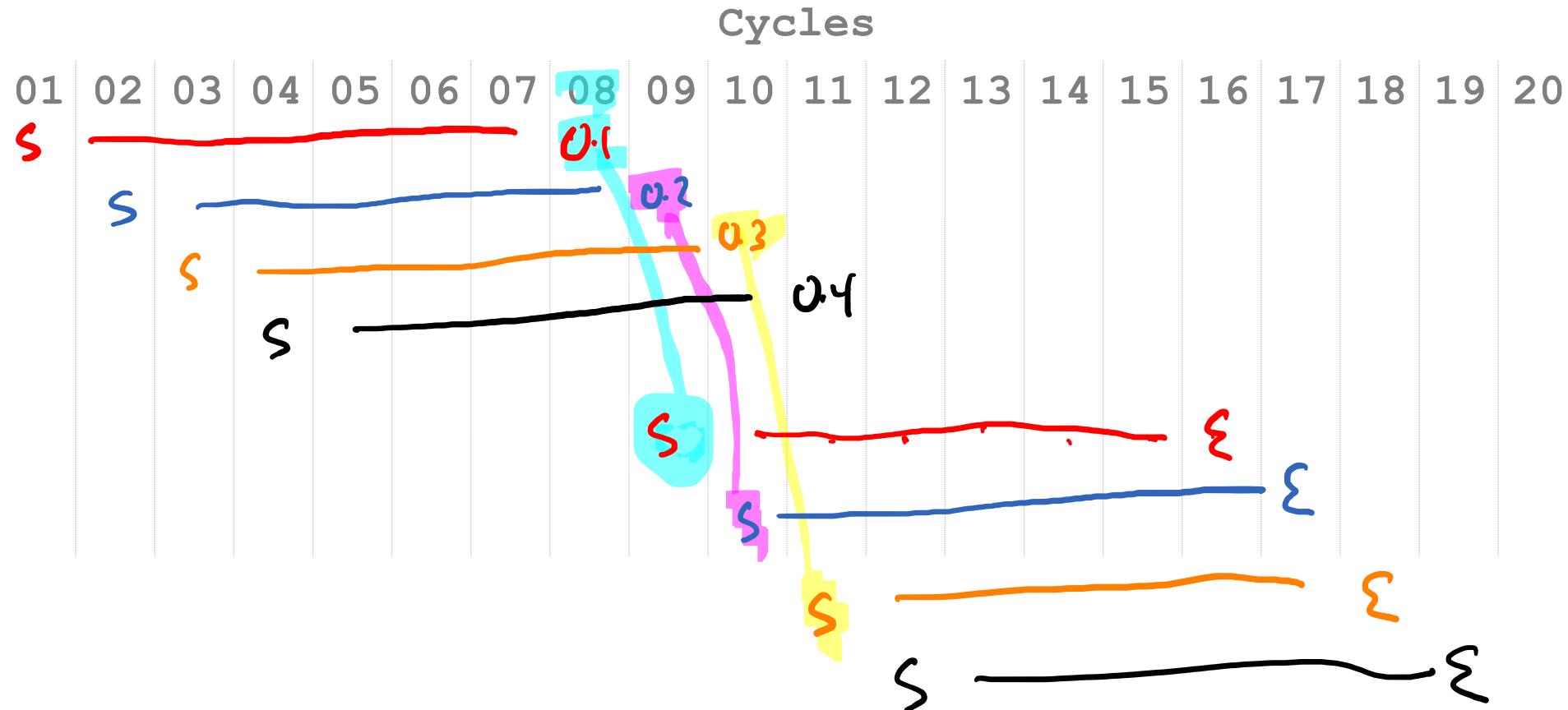
$$0.1 \cdot 3 + 0$$

$$0.1 \cdot 4 + 0$$

$$0.2 \cdot 5 + 0.1$$

$$0.2 \cdot 6 + 0.2$$

$$0.2 \cdot 7 + 0.3$$



# Latency on Pipelined FMAC

- Solution: Stall at the end of a row.
  - “Drain” the pipeline.
- 
- Restart new row
  - “Refill” the pipeline

How many cycles should this take?

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

# Data flow graph

- A data-flow graph is a **collection of arcs and nodes** in which the nodes are either **places where variables are assigned or used**, and the arcs show the relationship between the places where a variable is assigned and where the assigned value is subsequently used.

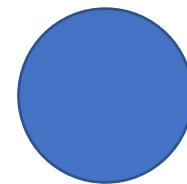
[\[ref\]](#)

# Data Flow Graph

- Illustrates the dependencies between inputs and operations to produce an output



Inputs



Operations



Dependencies

# Data Flow Graph

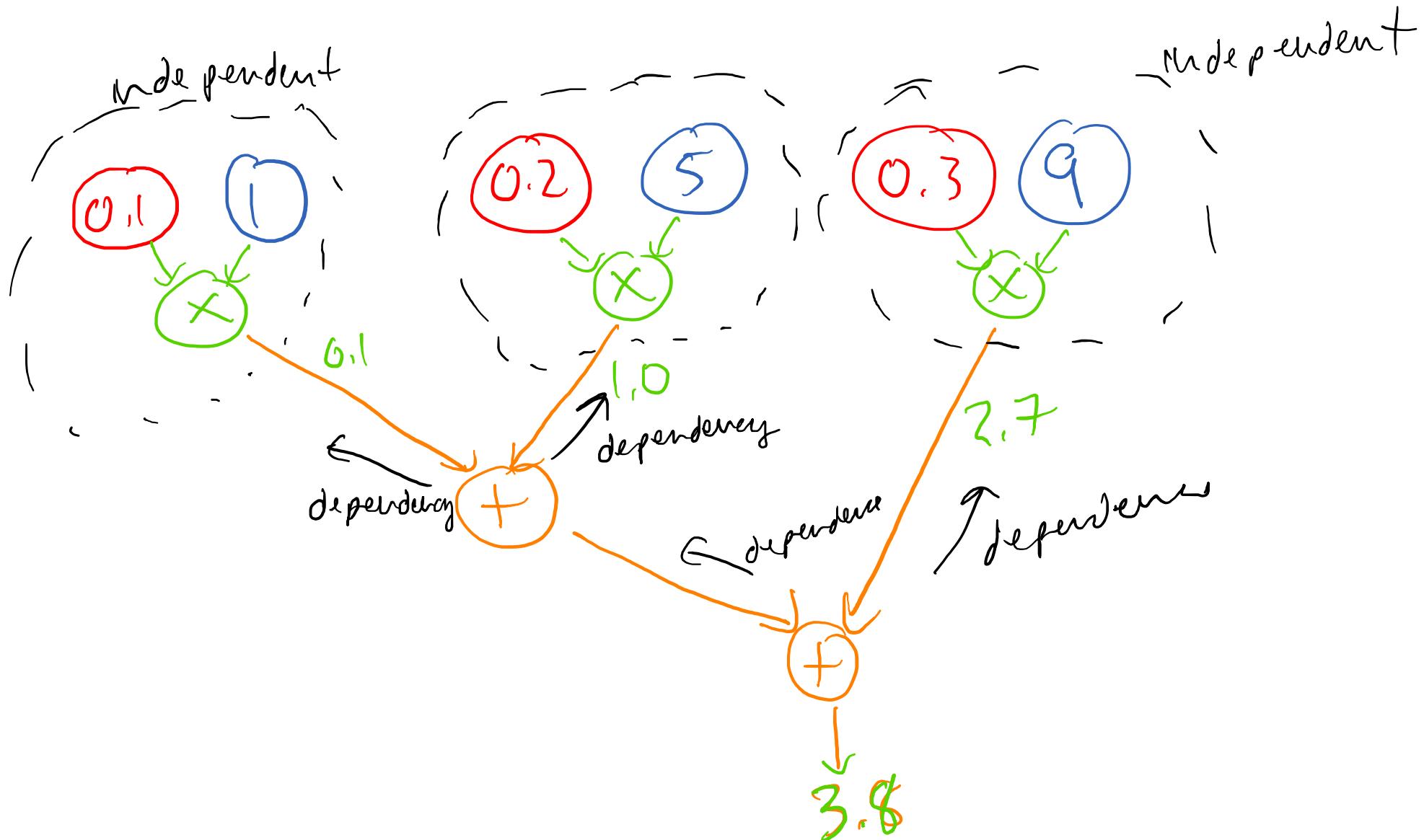
$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 5 \\ 5 \\ 9 \end{bmatrix} = \begin{bmatrix} 3.8 \end{bmatrix}$$

# Finding Parallelism

- Find some computation that doesn't depend on other computation's results.
- No arrows between blocks.
- Shared Inputs are OK.

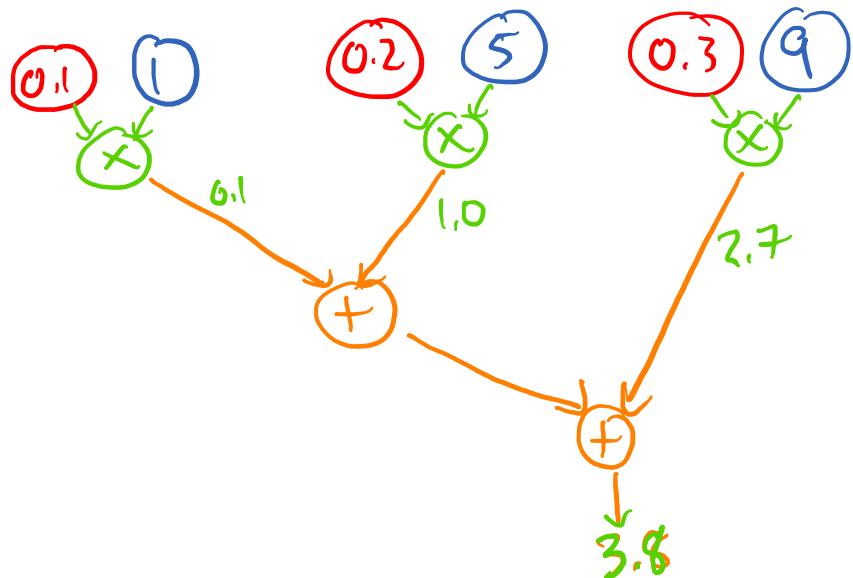
$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 9 \end{bmatrix} = [3.8]$$

# How many cycles should this take?



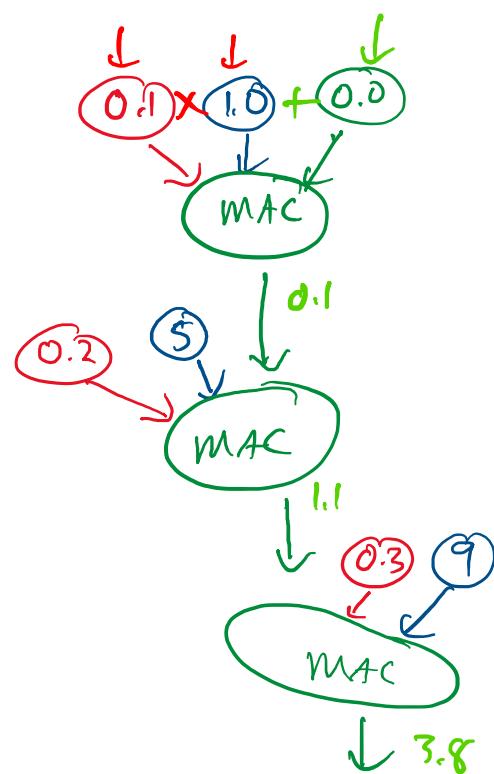
# How many cycles should this take?

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \times \begin{bmatrix} 5 \\ 5 \\ 9 \end{bmatrix} = \begin{bmatrix} 3.8 \end{bmatrix}$$



# More Data Flow

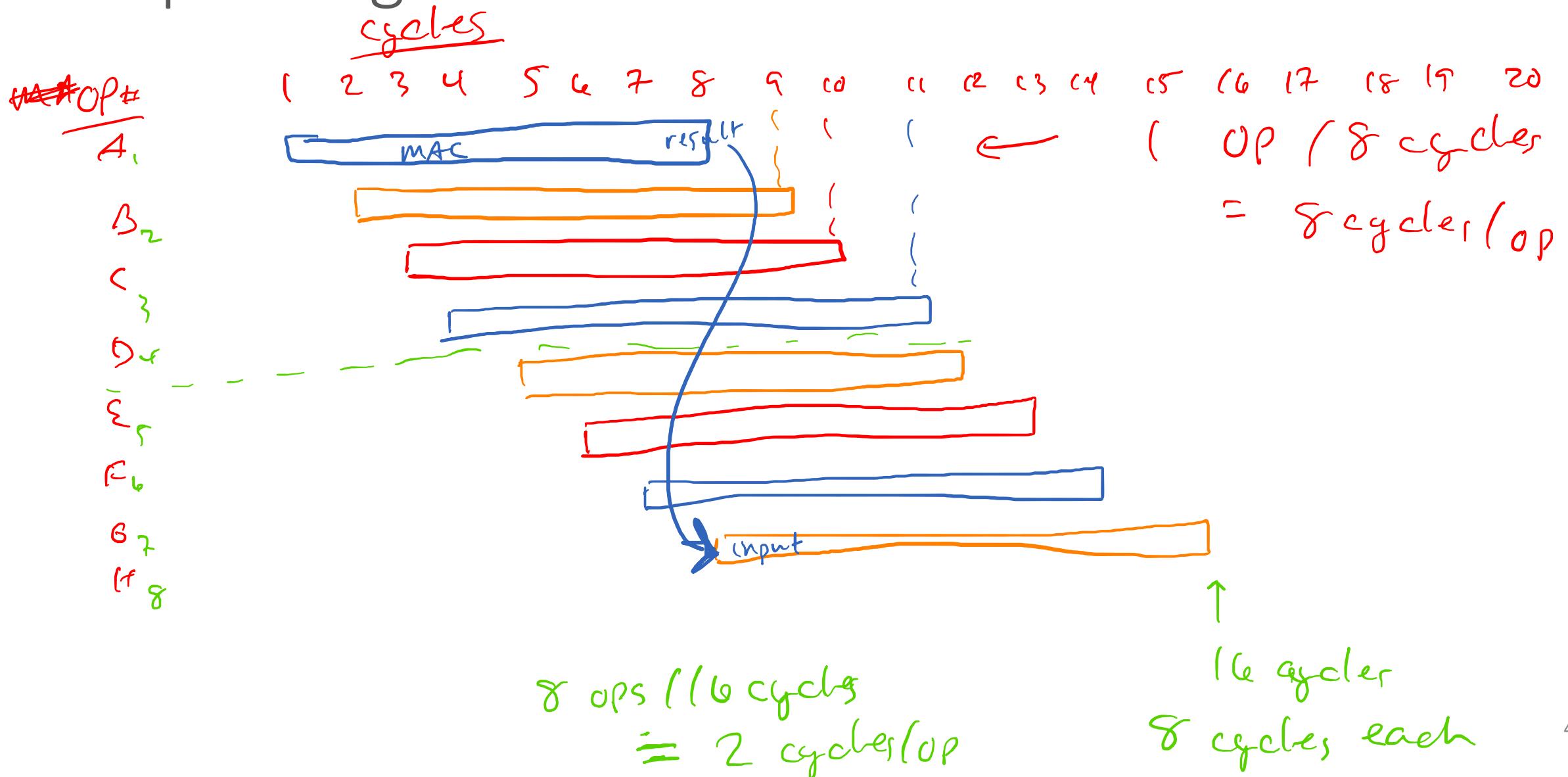
$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5.4 & 5.6 \end{bmatrix}$$



# Pipelining

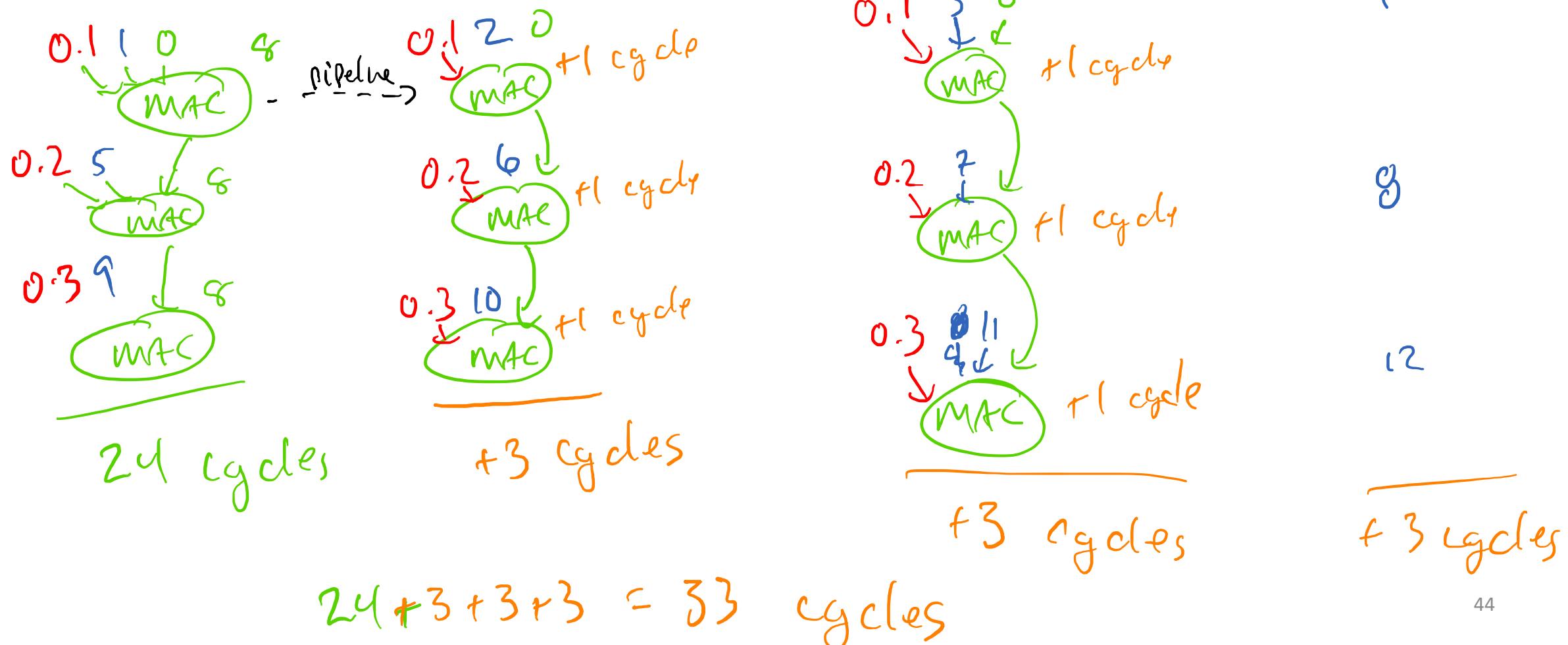
- FMAC takes 8 cycles for 1 value
  - But can accept a new value every cycle.
- Latency: 8 cycles / value ↵
  - Throughput: 1 value / cycle

# Pipelining



$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$



$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

One Output

All 4 Outputs

Multiply & Add separate:

MAC

Pipelined MAC

Maximum Parallel:

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

	<u>One Output</u>	<u>All 4 Outputs</u>
Multiply & Add separate:	40 cycles	160 cycles
MAC	24 cycles	96 cycles
Pipelined MAC	24 cycles (+1 for extra rows)	33 cycles
Maximum Parallel:	24 cycles	24 cycles

# Next Time: Divide and Conquer

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

$$0.1 \cdot \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix}$$

$$0.2 \cdot \begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix} = \begin{bmatrix} 1.1 & 1.4 & 1.7 & 2.0 \end{bmatrix}$$

$$0.3 \cdot \begin{bmatrix} 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 1.1 & 1.4 & 1.7 & 2.0 \end{bmatrix} = \underbrace{\begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}}$$

# Can we parallelize Dot?

```
# 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
```

# Can we parallelize Dot?

```
# 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
```

```
def par_pydot(inputs, weights):
    par_inputs = [inputs[:,::2], inputs[:,1::2]]
    par_weights = [weights[::2,:,:], weights[1::2,:,:]]

    par_outputs = [pydot(par_inputs[0], par_weights[0]),
                  pydot(par_inputs[1], par_weights[1])]

    outputs = par_outputs[0] + par_outputs[1]
    return outputs
```

# Can we parallelize Dot?

```
# 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
```

```
def par_pydot(inputs, weights):
    par_inputs = [inputs[:,::2], inputs[:,1::2]]
    par_weights = [weights[::2,:,:], weights[1::2,:,:]]

    → par_outputs = [pydot(par_inputs[0], par_weights[0]),
                     pydot(par_inputs[1], par_weights[1])]

    outputs = par_outputs[0] + par_outputs[1]
    return outputs
```