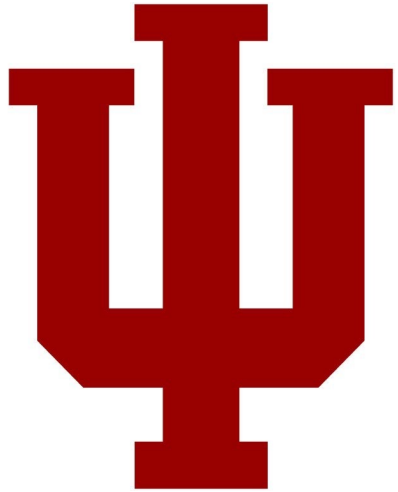
More code PS

17: Pipelining II

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/

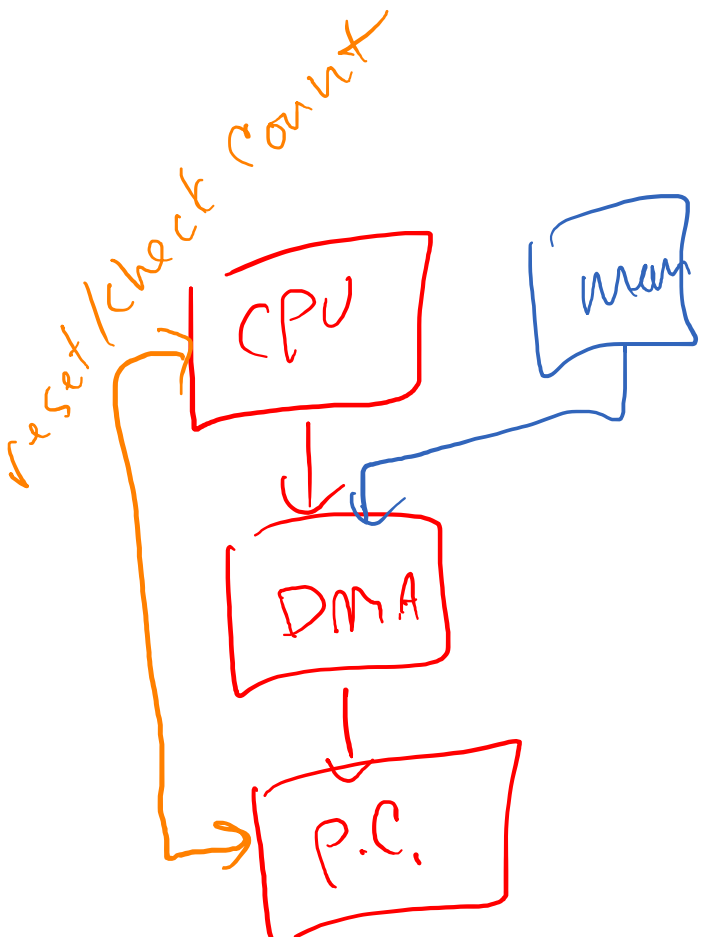
Announcements

- P6 out. Due Wednesday.
- No Class on Wednesday (Oct 26th)
- Exam Nov 6th

TA session then

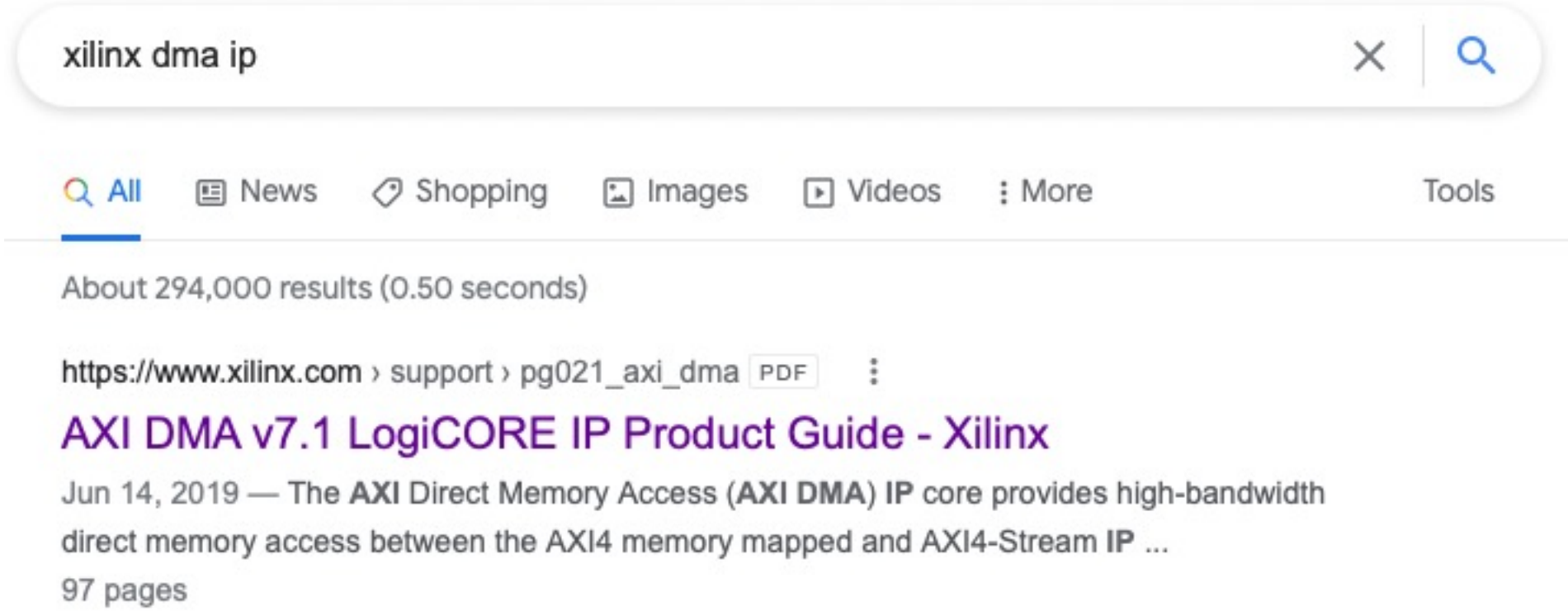
Review - Prior Wed Nov. 1st - mon

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

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

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.

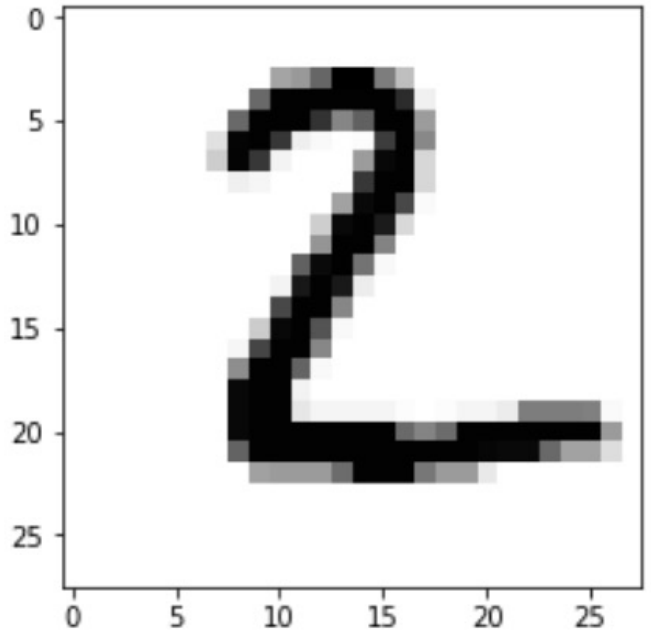
5. Wait until MM2S_DMASR.Idle==1 for completion

P8+ Accelerate Machine Learning

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

Simple Neural Network

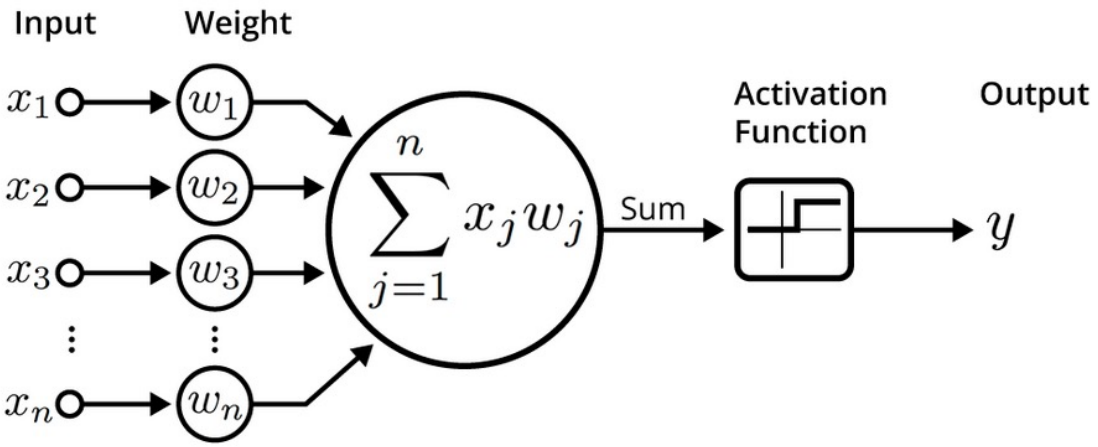
```
================================
Index: 0
Image:
```




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

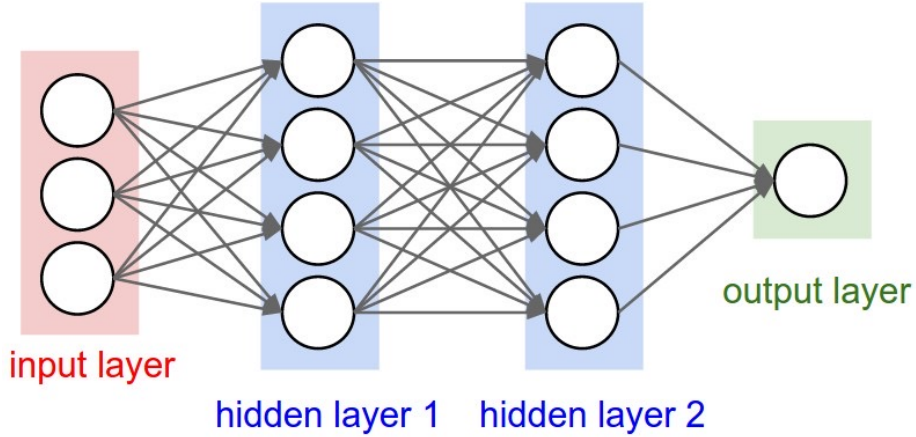
- Takes in image of number
- Returns integer value
- How? artificial neural network

Python Neuron

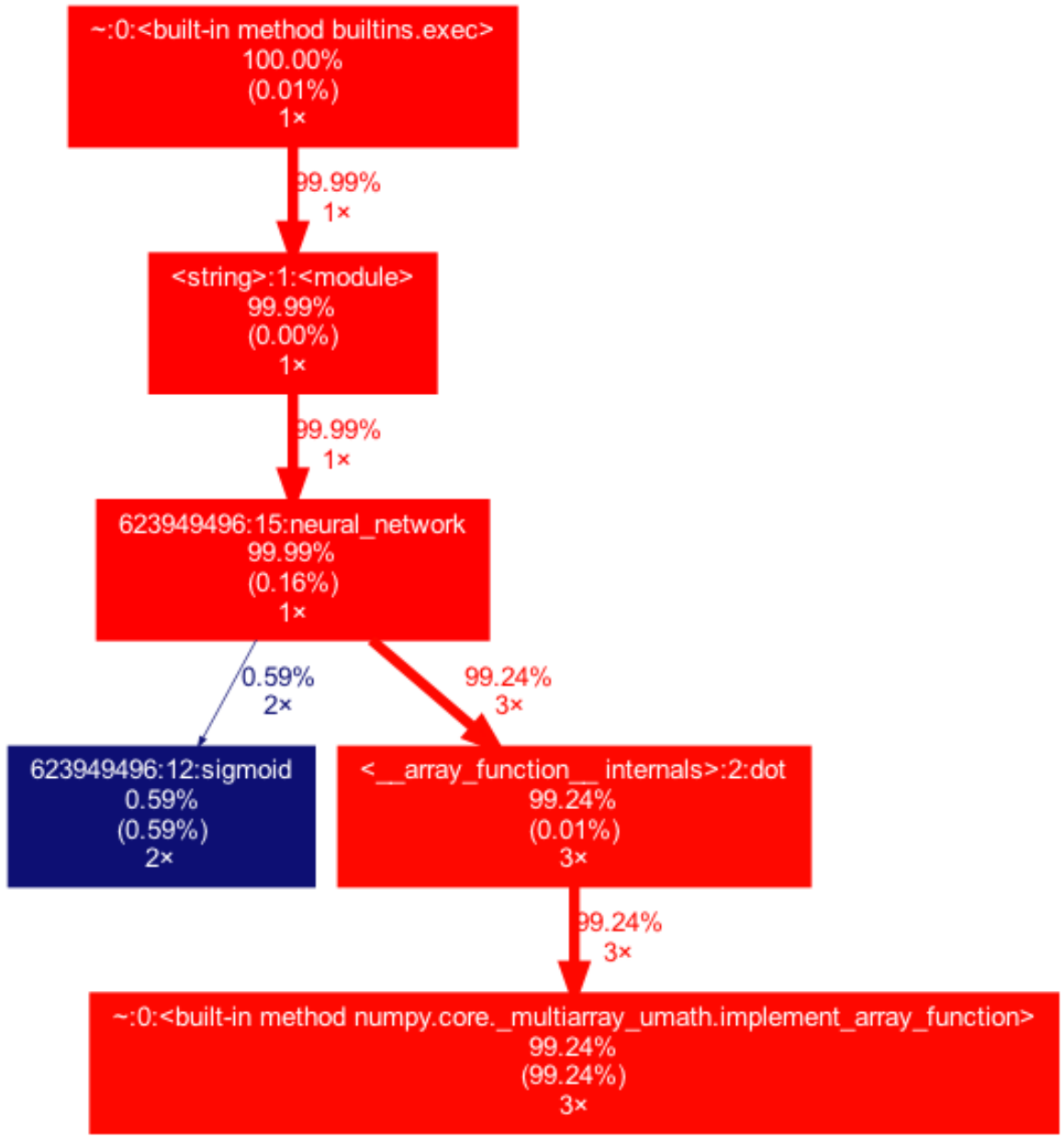



```
class Neuron(object):
  # ...
  def forward(self, inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return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

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)
```


~:0:<built-in method builtins.exec>
100.00%
(0.01%)
1×
99.99%
1×
<string>:1:<module>
99.99%
(0.00%)
1×
99.99%
1×
623949496:15:neural_network
99.99%
(0.16%)
1×
0.59%
2×
99.24%
3×
623949496:12:sigmoid
0.59%
(0.59%)
2×
<__array_function__ internals>:2:dot
99.24%
(0.01%)
3×
99.24%
3×
~:0:<built-in method numpy.core._multiarray_umath.implement_array_function>
99.24%
(99.24%)
3×

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)

inputs

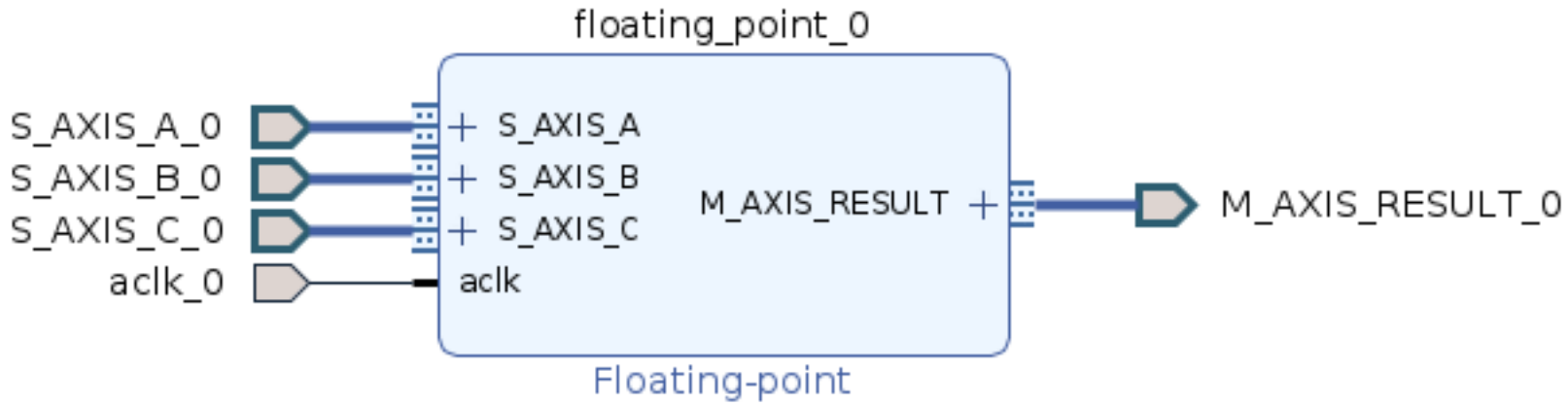
$$\begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} =$$

weights

Floating-Point Multiply-Accumulate (FMAC)

- Math: a * b + c

Floating-Point Multiply in Hardware

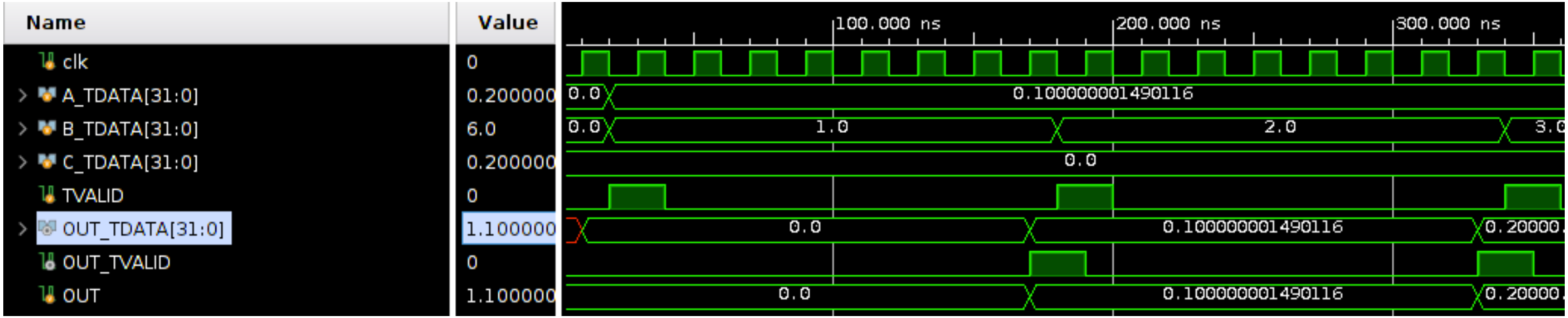


- result = a * b + c

Floating-Point math takes 8 cycles.

- Floating-Point is complicated.
- 8 cycles of complicated.

Demo Time

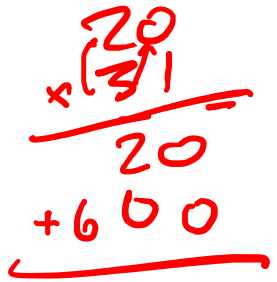

Name
Value
clk
A_TDATA[31:0]
B_TDATA[31:0]
C_TDATA[31:0]
TVALID
OUT_TDATA[31:0]
OUT_TVALID
OUT
0
0.200000
6.0
0.200000
0
1.100000
0
1.100000
100.000 ns
200.000 ns
300.000 ns
0.0
0.100000001490116
1.0
2.0
3.0
0.20000

Floating-Point math takes 8 cycles.

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

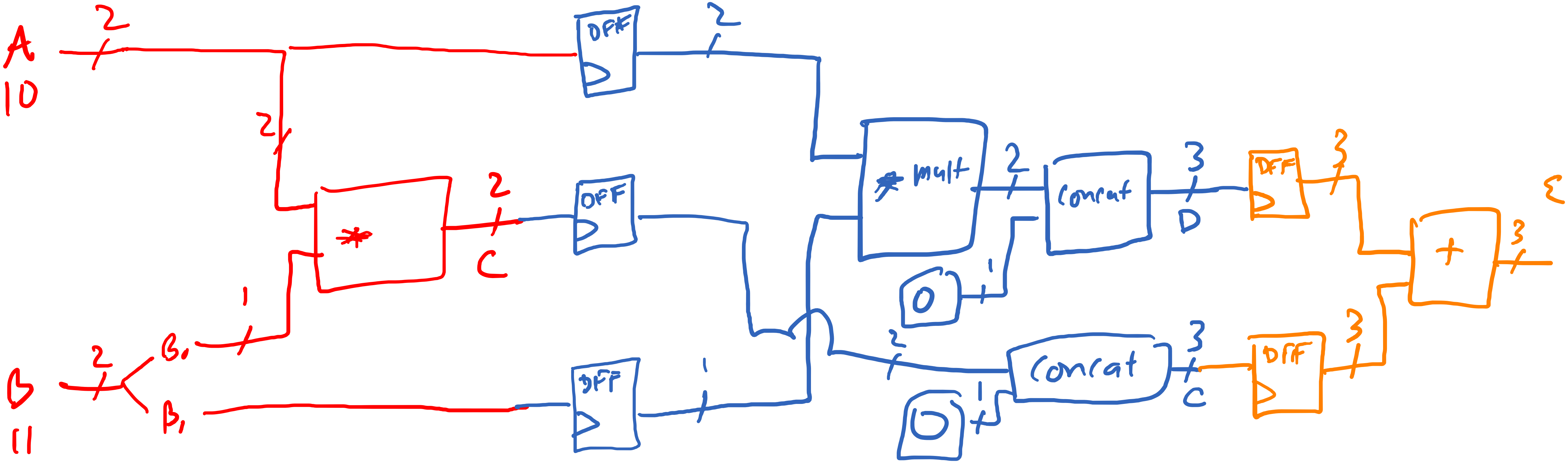
How does Pipelining work in Math?

$$\begin{array}{r} 2 \\ \times\ 3 \\ \hline 6 \end{array} \Rightarrow \begin{array}{r} 10 \\ \times\ 11 \\ \hline 110 \end{array} \Rightarrow \begin{array}{r} 10 \\ \times\ 1 \\ \hline 10 \end{array} + \begin{array}{r} 10 \\ \times\ 10 \\ \hline 100 \end{array} = 110$$



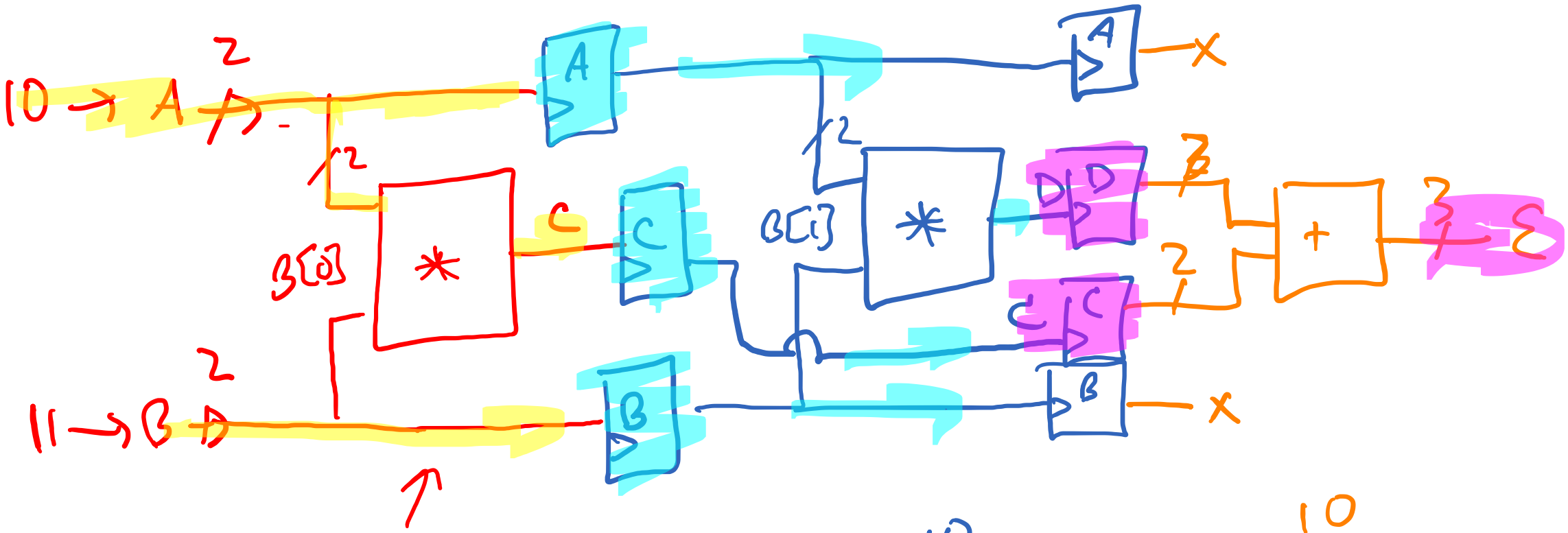
How does pipelining work in circuits?

2		10		10		10		
x 3	=>	x 11	=>	x 1	+	x 10		
6		110		010	+	100	=	110



Pipelining in hardware

```
  2        10        10
x 3   =>  x 11  =>   x 1   +  x 10
  6        110       10    + 100  = 110
```

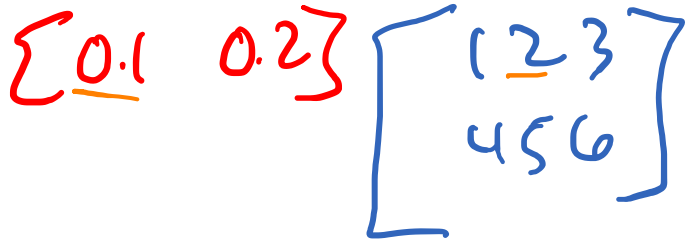


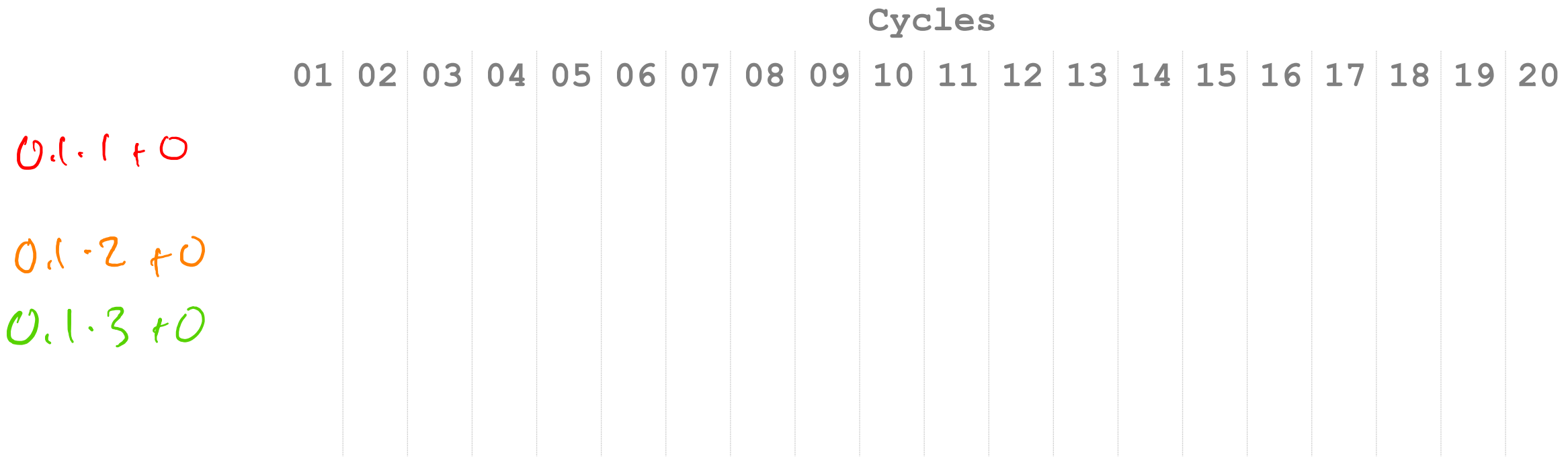
(3) 10 × 1 = C = 10

10 × 10 = D = 100

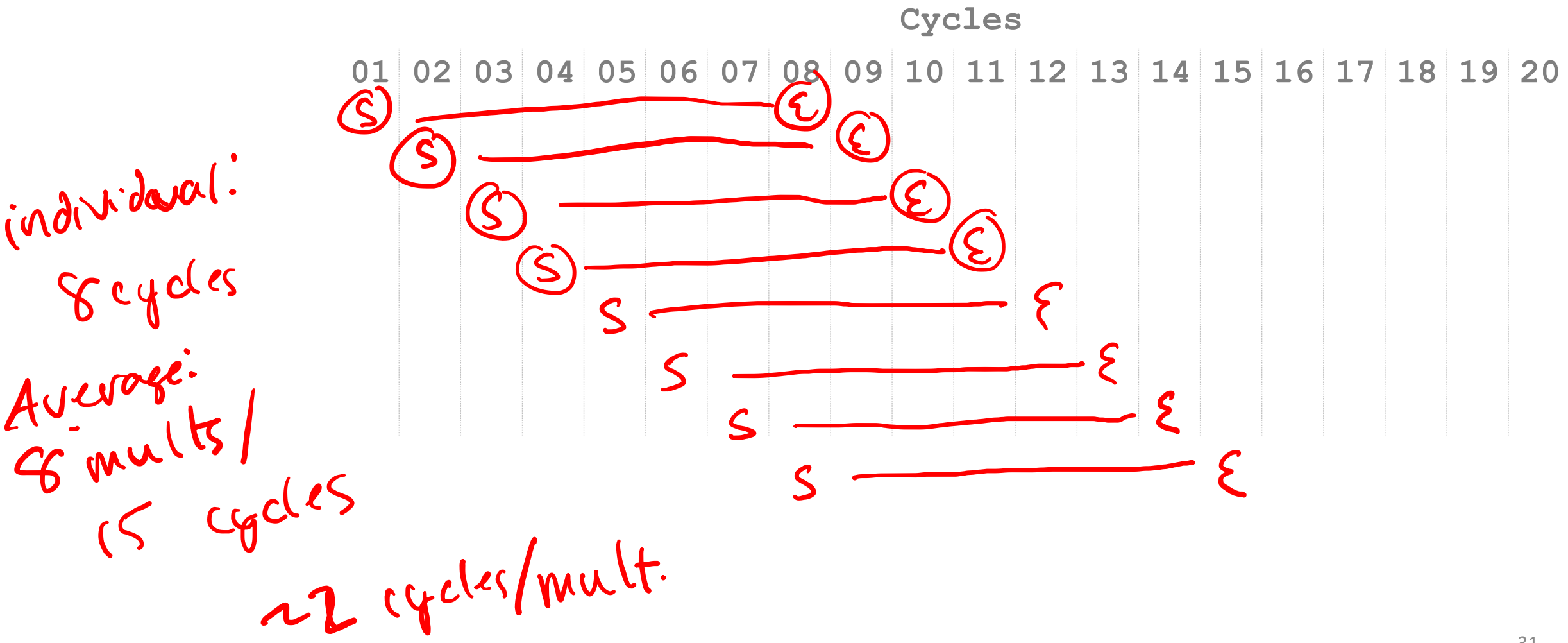
10 + 100 = E = 110

FMAC Pipelining





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

Pipelining

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

- What is Latency: 8
- What is Throughput: ~1/cycle

Recall: 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

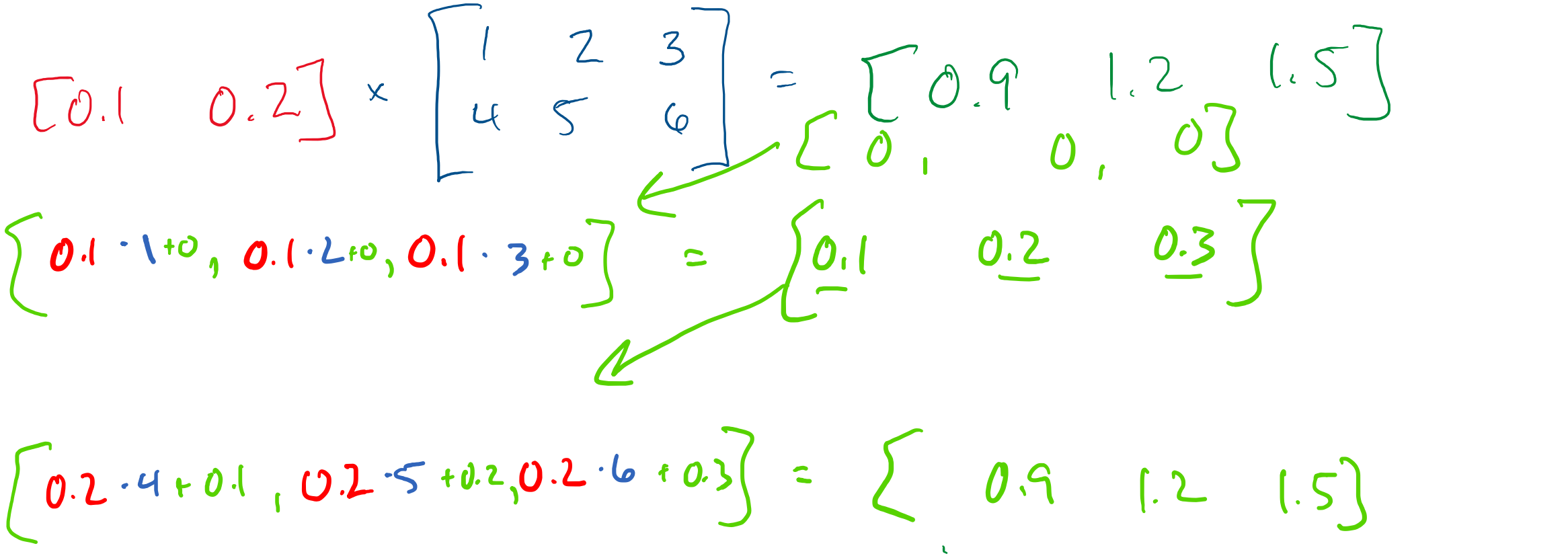
$$\begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$

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

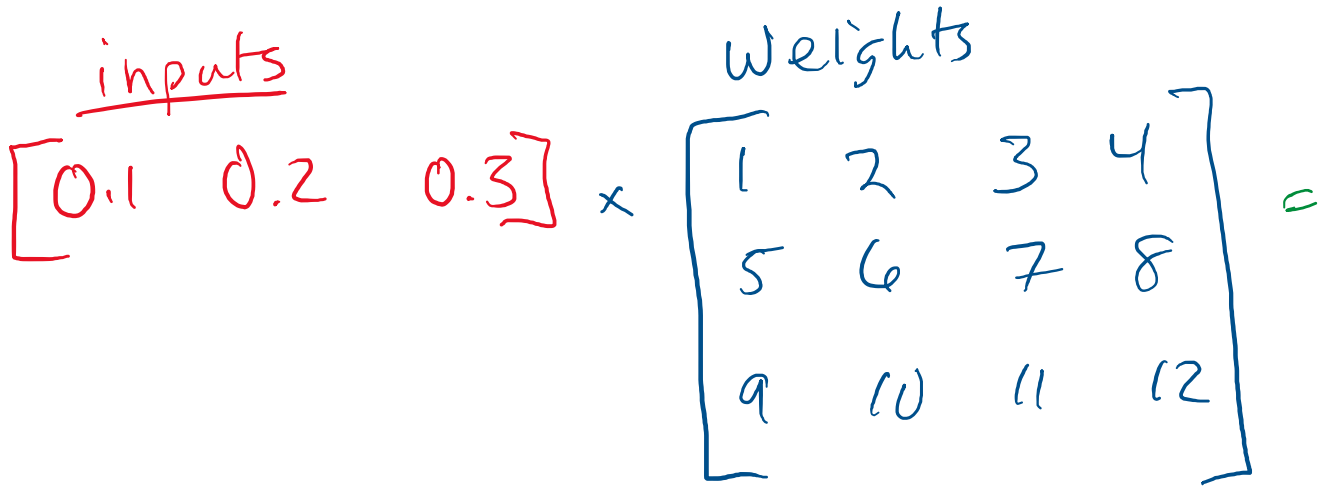
$$\begin{bmatrix} 0.2 \cdot 4 & 0.2 \cdot 5 & 0.2 \cdot 6 \end{bmatrix} = + \begin{bmatrix} 0.8 & 1.0 & 1.2 \end{bmatrix}$$

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

Multiply-Accumulate Dot Computations



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.]
```

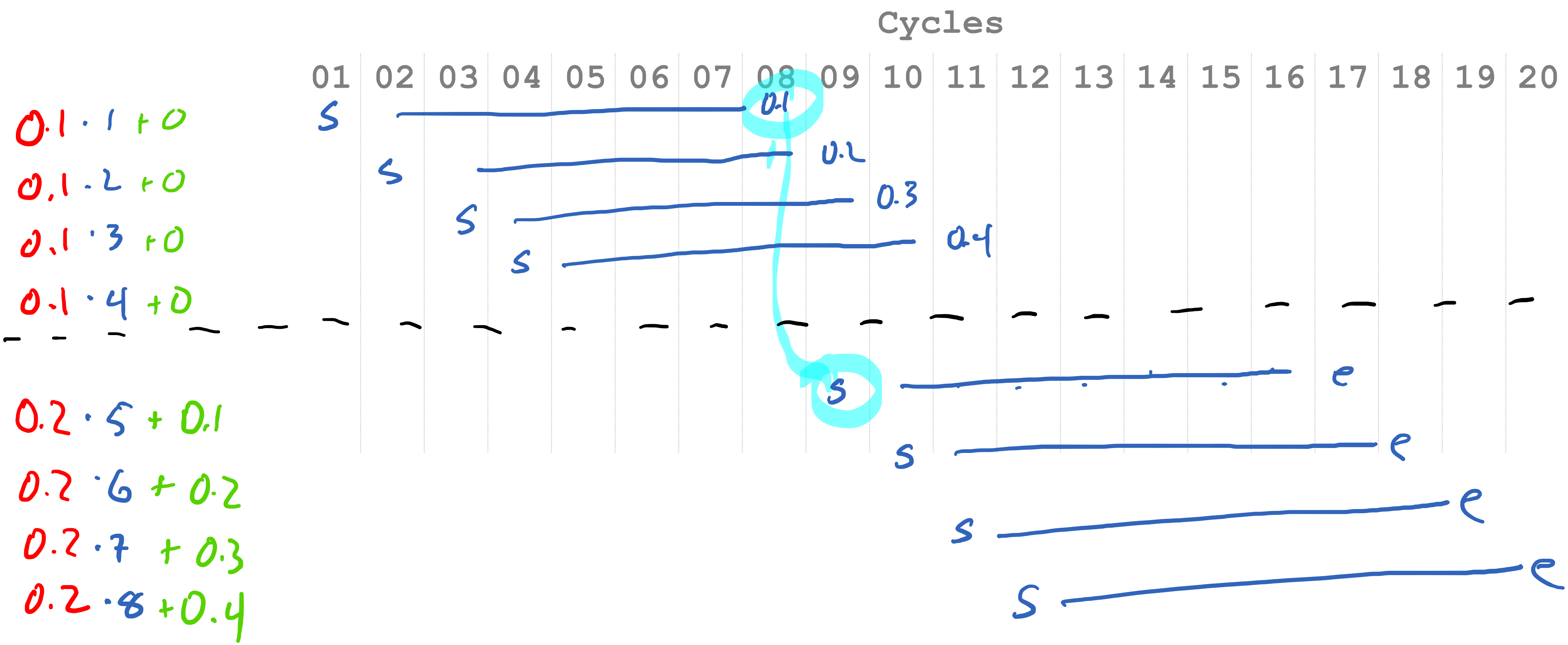
Mult-Accum Dot

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

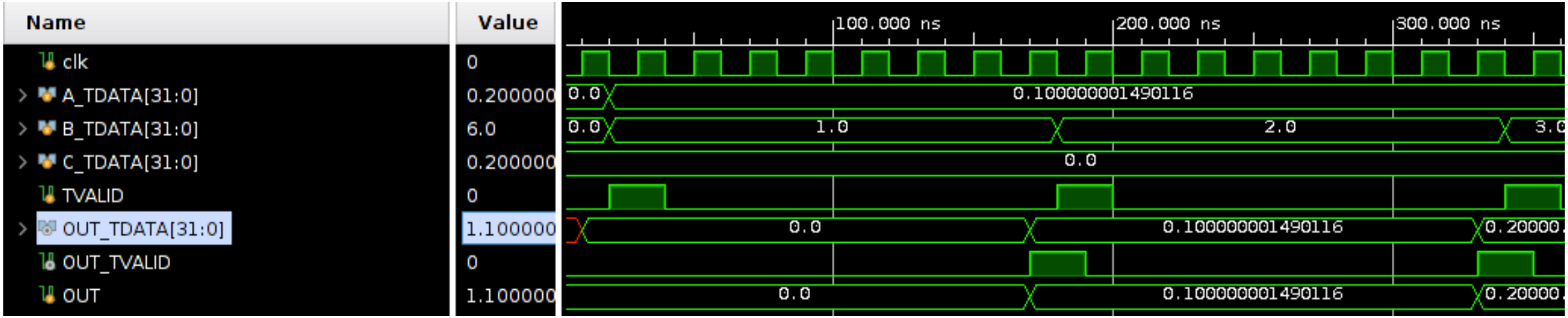
```python
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j]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

Dependencies

```
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.]
```



Demo Time


Name
Value
clk
0
A_TDATA[31:0]
0.200000
B_TDATA[31:0]
6.0
C_TDATA[31:0]
0.200000
TVALID
0
OUT_TDATA[31:0]
1.100000
OUT_TVALID
0
OUT
1.100000
100.000 ns
200.000 ns
300.000 ns
0.0
0.100000001490116
0.0
1.0
2.0
0.0
0.0
0.100000001490116
0.0
0.100000001490116

Next Time: More on Dependencies

Latency on Pipelined FMAC

- Solution: Stall at the end of a row.
- Drain the pipeline.

Hardware Parallelism

- CPU: 1 Floating-Point Unit
- FPGA? 10 Floating-Point Units?
  20 ?
  100 ?

Finding Parallelism

- Some some computation that doesn’t depend on other computation’s results
- Shared Inputs are OK.

Next Time: Can we use 2+ FMACs?

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

Stopped here!

Option 1

cycle	fMAC comp
0	0.1·1 + 0
1	0.1·2 + 0
2	0.1·3 + 0
3	0.1·4 + 0

Cycle	fMAC 1	fmac 2
0	0.1·1+0	0.1·2+0
1	0.1·3+0	0.1·3+0

Option 2

→ 0.1 *1 +0
0.1 *2 +0
0.1 *3 +0
0.1 *4 +0

0.2 * 5 +0
0.2 * 6 +0
0.2 * 7 +0
0.2 * 8 +0

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

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

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

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

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

Parallize Alternative Dot Computations?

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

$[0, 0, 0]$ ← result

$0.1 \cdot [1\ 2\ 3] + [0\ 0\ 0] =$

$[0.1\ 0.2\ 0.3] + [0\ 0\ 0] = [0.1\ 0.2\ 0.3]$ ← temp result

$0.2 \cdot [4\ 5\ 6] + [0.1\ 0.2\ 0.3] =$

$[0.8\ 1.0\ 1.2] + [0.1\ 0.2\ 0.3] = [0.9\ 1.2\ 1.5]$

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?

```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
```

```python
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
```

19: Hardware Acceleration III

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University