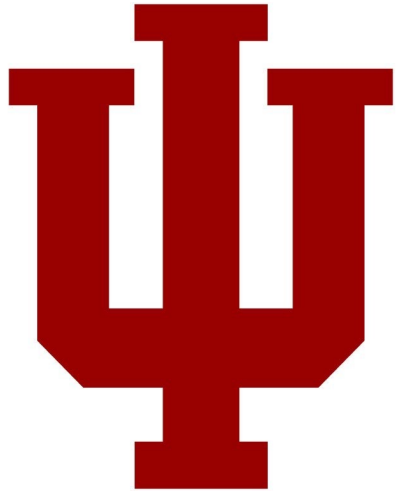
17: Hardware Acceleration IV

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



Announcements

- Exams almost done grading.
- P7 is out → Dot Product ≤ 300 cycles
- P8 is coming. → Dot Product how fast can you go

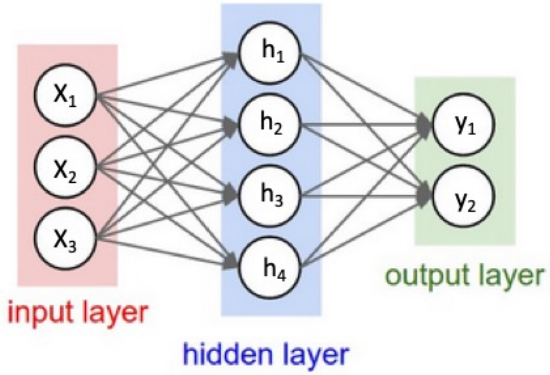
Project 7: Accelerate Dot Product

- Given: Slow Dot-Product in Hardware
- Your Task: Accelerate it!
- How? Pipelining + Parallelism (later)
- Groups of 2 allowed.

Project 8: Accelerate Dot Product More

- Your Task: Accelerate P7
- How? Pipelining + Parallelism
- Same groups of 2 allowed.
- Additional documentation requirements

Why 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)

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

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

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

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

Alternative Dot Computations

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

$[0 \quad 0 \quad 0]$ ← temp result

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

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

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

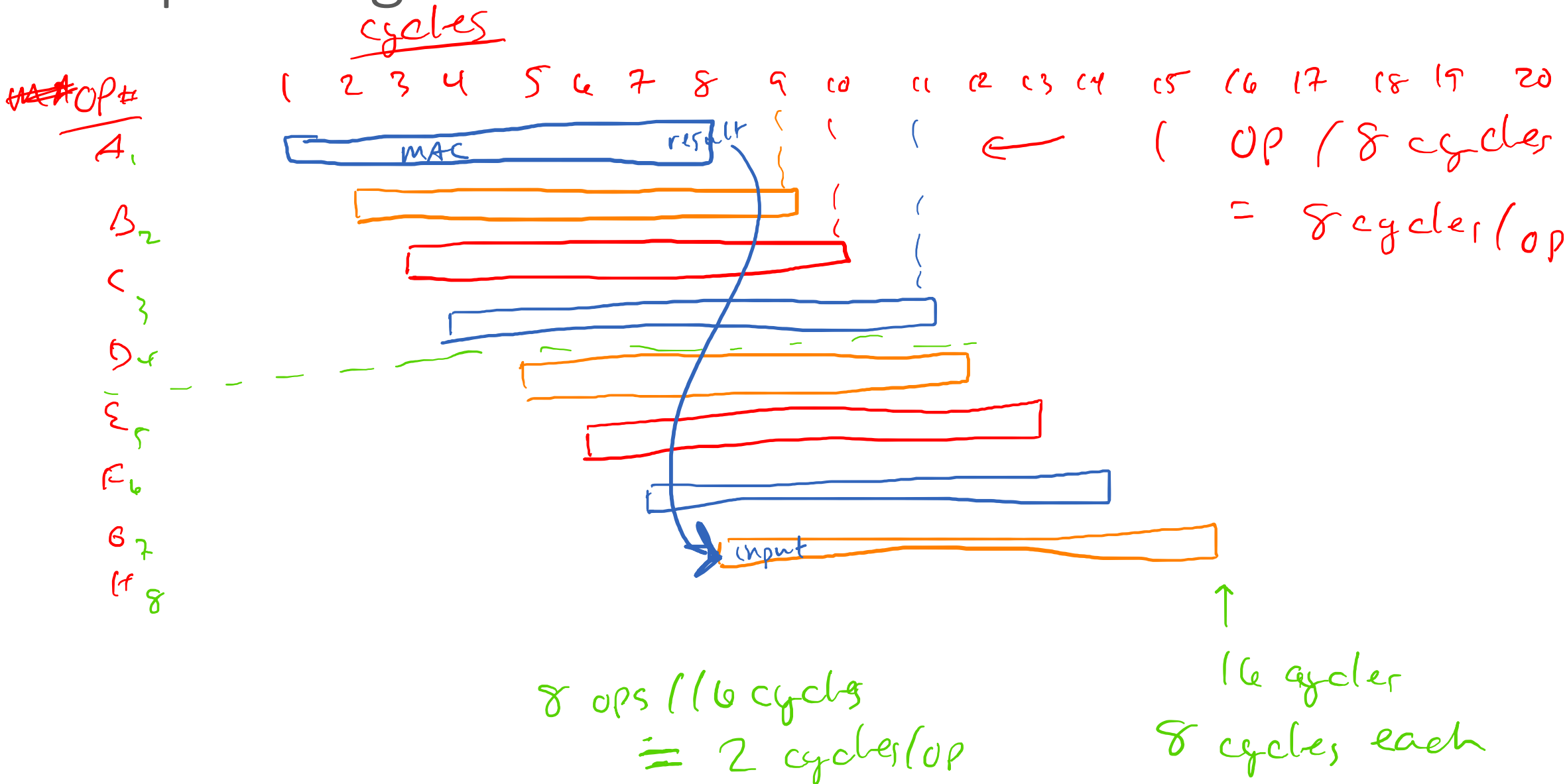
$[0.8 \quad 1.0 \quad 1.2] + [0.1 \quad 0.2 \quad 0.3] = [0.9 \quad 1.2 \quad 1.5]$

Pipelining

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

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

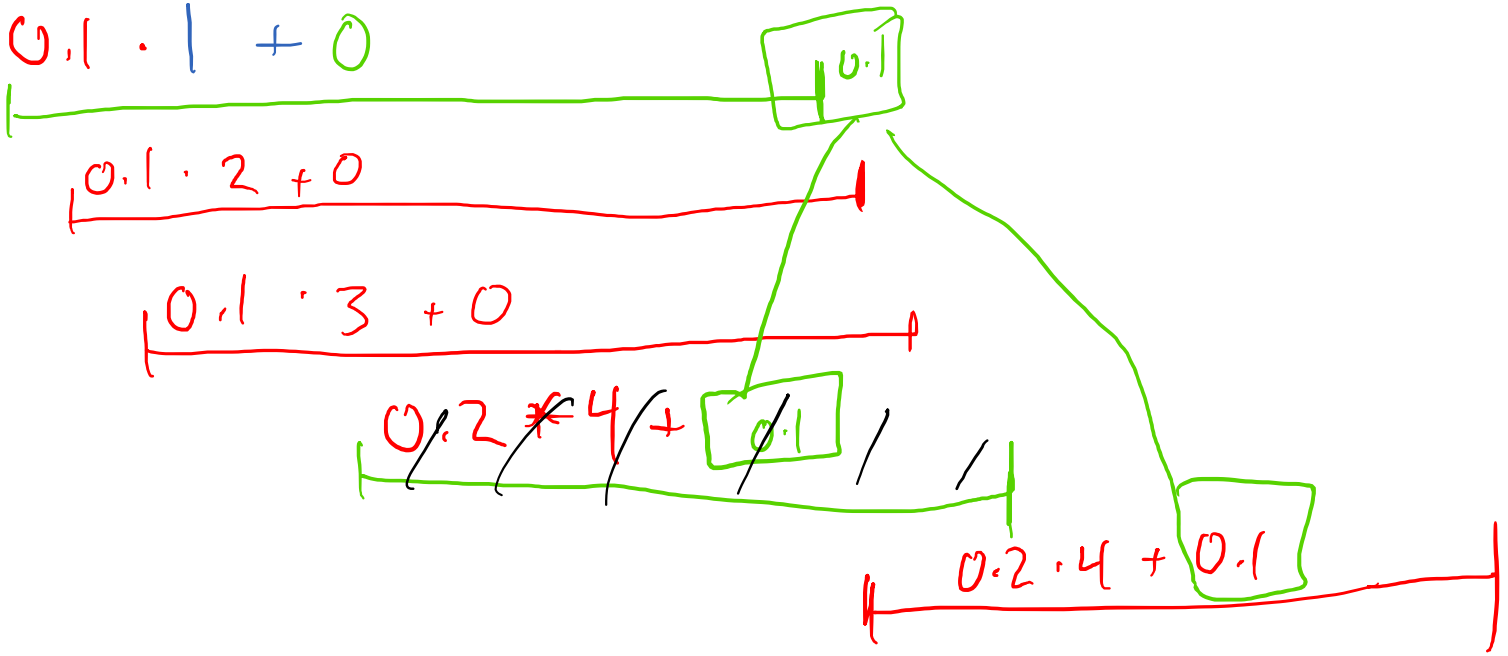
Pipelining


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

Pipelined Dot Computations

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

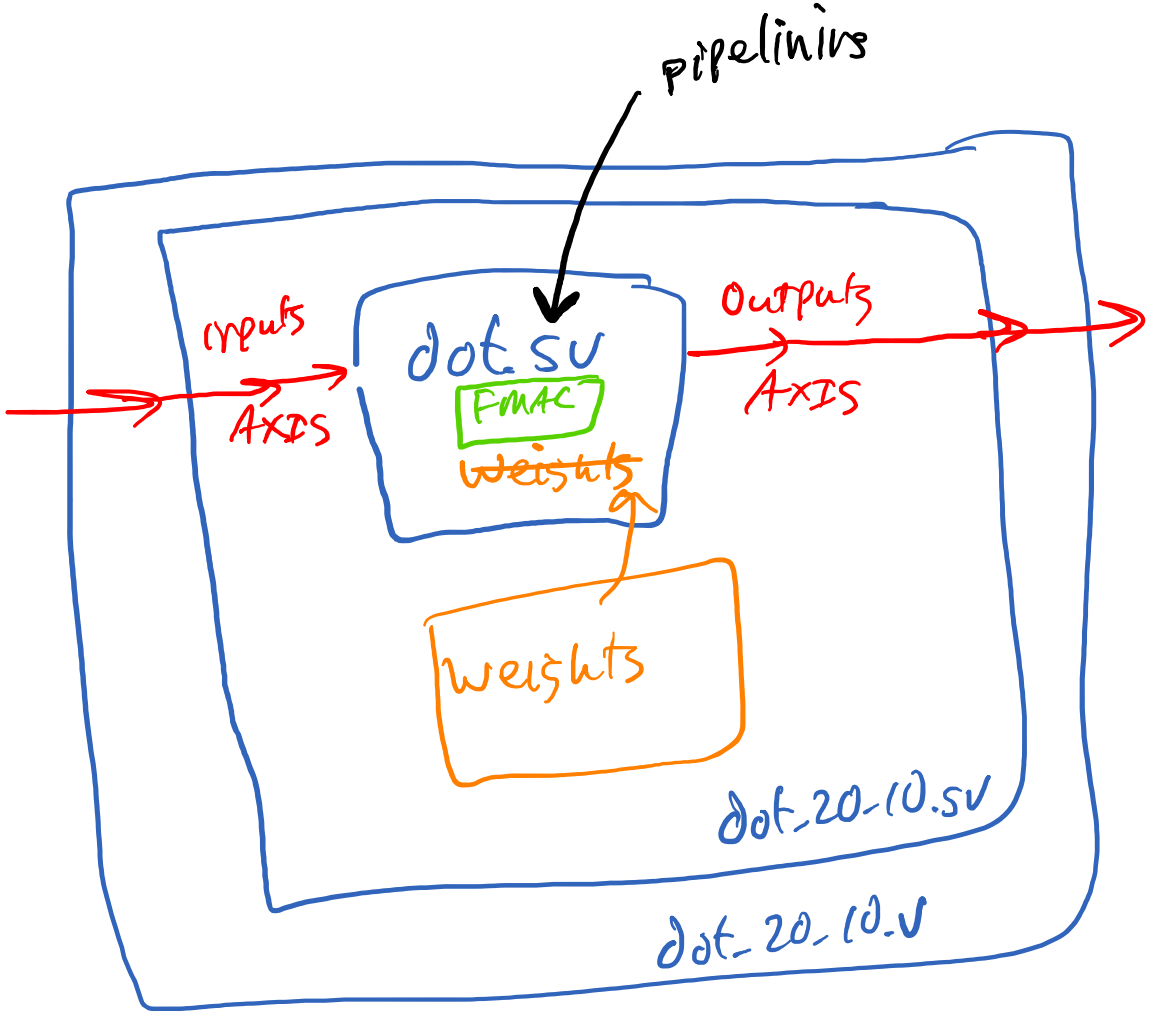
time →



Latency on Pipelined FMAC

- Pro Tip: Stall at the end of a row.
- Drain the FMAC pipeline.

Dot Data flow



[0.1 ,0.2] [3 4 5; 6 7 8]

Verilog Parameters

- Parameters are Verilog constructs that allow a module to be **reused with a different specification**.

```
module adder #(parameter BITS = 2) (
     input [BITS-1:0] a,
     inputs [BITS-1:0] b,
     outputs [BITS-1:0 c) );
     assign c = a + b;
endmodule
```

Verilog Parameters

```
module adder #(parameter BITS = 2) (
       input [BITS-1:0] a,
     inputs [BITS-1:0] b,
     outputs [BITS-1:0 c) );
       assign c = a + b;
endmodule
```

```
// 2-bit adder
adder add2 (a2, b2, c2);
//another 2-bit adder
adder #(BITS=2) add2b (a2b,b2b,c2b);
//a 3-bit adder
adder #(BITS=3) add3 (a3, b3, c3);

Adder #(BITS=32) add32 (a32, b32, c32);
```

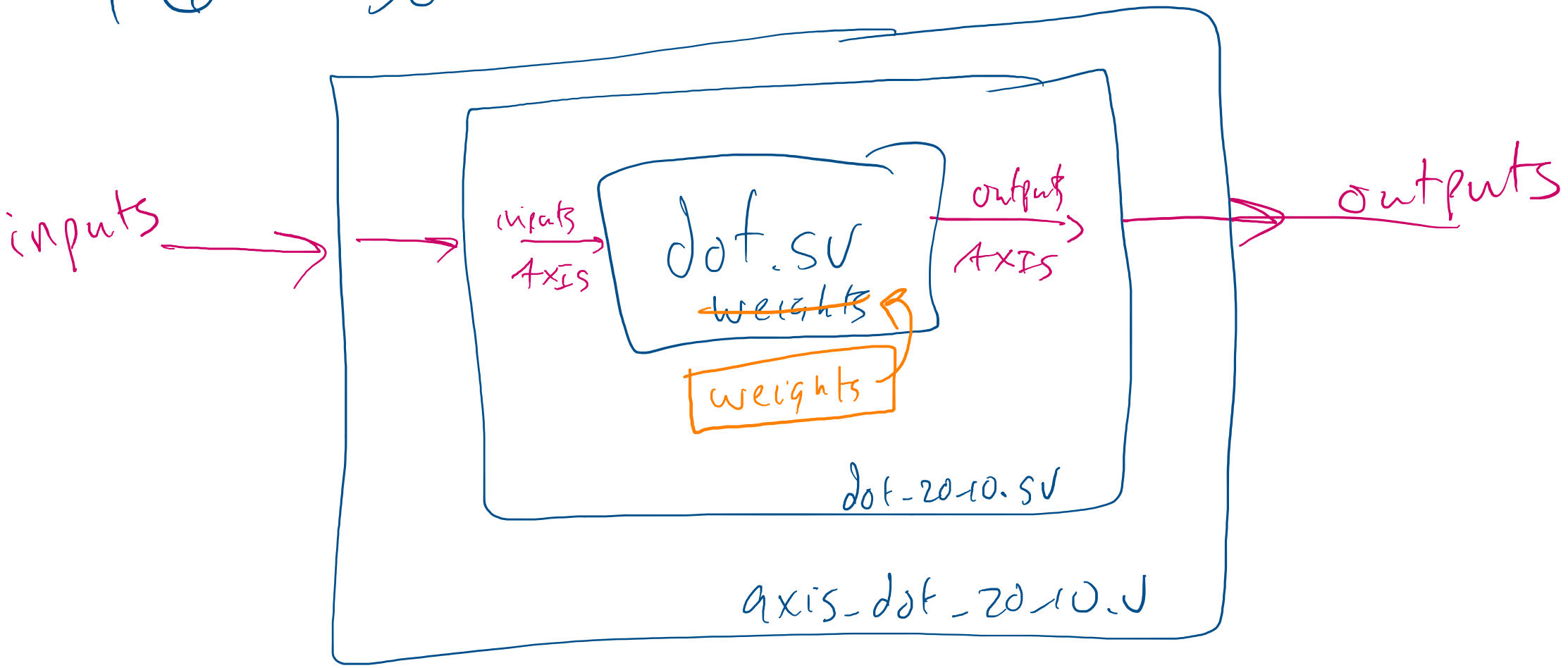
Verilog Parameters in Dot.sv

```
module dot #(
parameter ROWS = 3,
parameter COLS = 4,
parameter [31:0] weights [0:ROWS-1] [0:COLS-1] = '{
    '{$shortrealtobits(1.0),$shortrealtobits(2.0),$shortrealtobits(3.0),$shortrealtobits(4.0)},
    '{$shortrealtobits(5.0),$shortrealtobits(6.0),$shortrealtobits(7.0),$shortrealtobits(8.0)},
    '{$shortrealtobits(9.0),$shortrealtobits(10.0),$shortrealtobits(11.0),$shortrealtobits(12.0)}
}
    )(
    input clk, rst,
    // Incomming Matrix AXI4-Stream
    input [31:0]                    INPUT_AXIS_TDATA,
    input                           INPUT_AXIS_TLAST,
    input                           INPUT_AXIS_TVALID,
    output reg                      INPUT_AXIS_TREADY,
    // Outgoing Vector AXI4-Stream
    output reg [31:0]               OUTPUT_AXIS_TDATA,
    output reg                      OUTPUT_AXIS_TLAST,
    output reg                      OUTPUT_AXIS_TVALID,
    input                           OUTPUT_AXIS_TREADY  );
```

Verilog Parameters in dot_20_10.sv

```
dot #(
        .ROWS(ROWS),
        .COLS(COLS),
        .weights(weights)
    ) dot0 (
        // AXI4-Stream Interface
        .clk(clk),
        .rst(rst),
        .INPUT_AXIS_TDATA(INPUT_AXIS_TDATA),
        .INPUT_AXIS_TLAST(INPUT_AXIS_TLAST),
        .INPUT_AXIS_TVALID(INPUT_AXIS_TVALID),
        .INPUT_AXIS_TREADY(INPUT_AXIS_TREADY),
        .OUTPUT_AXIS_TDATA(OUTPUT_AXIS_TDATA),
        .OUTPUT_AXIS_TLAST(OUTPUT_AXIS_TLAST),
        .OUTPUT_AXIS_TVALID(OUTPUT_AXIS_TVALID),
        .OUTPUT_AXIS_TREADY(OUTPUT_AXIS_TREADY)
    );
```

P6 Dot Data flow



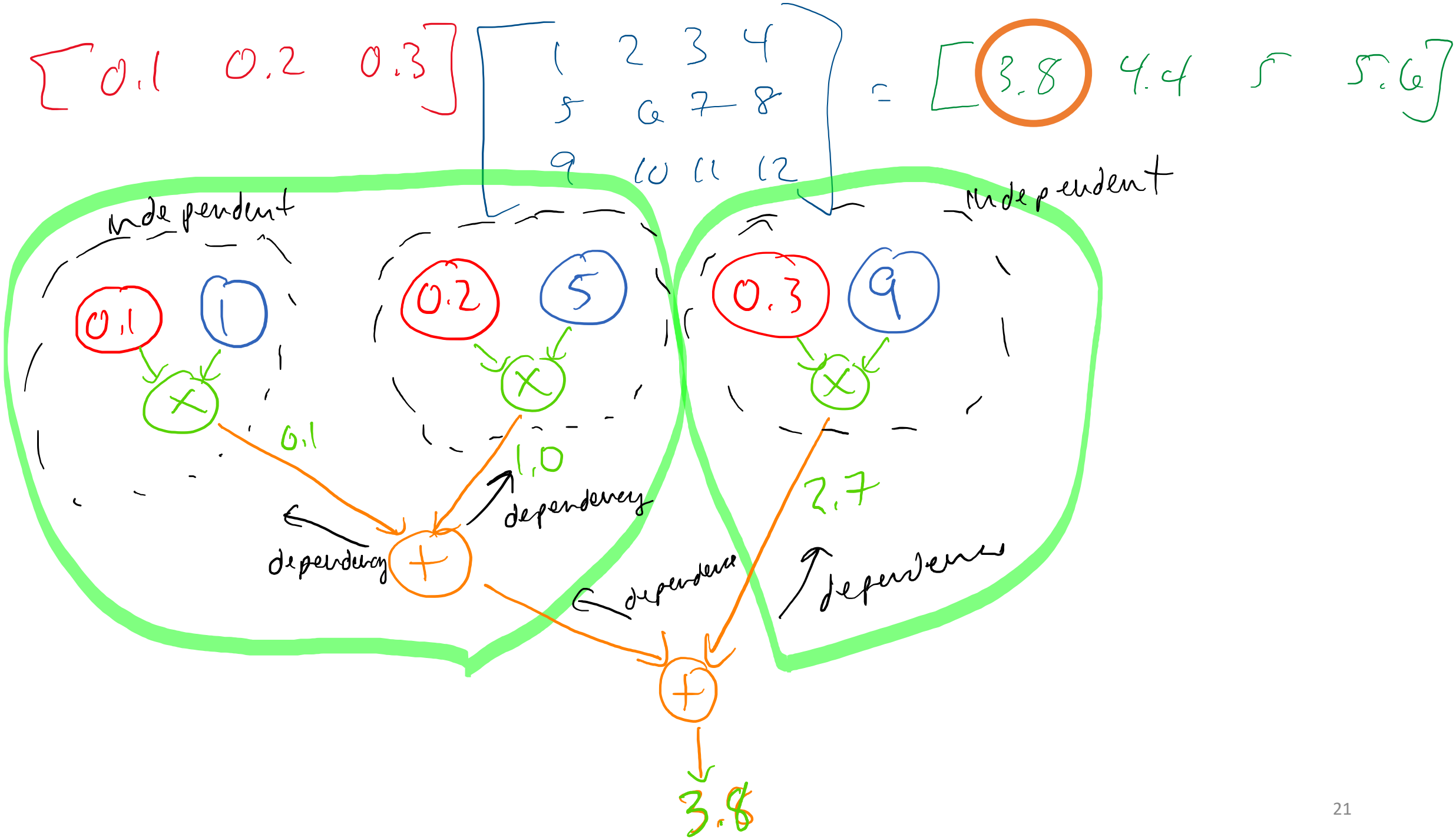
Data flow graph

Fixme: add 0.4

[0.1 0.2 0.3 0.4]

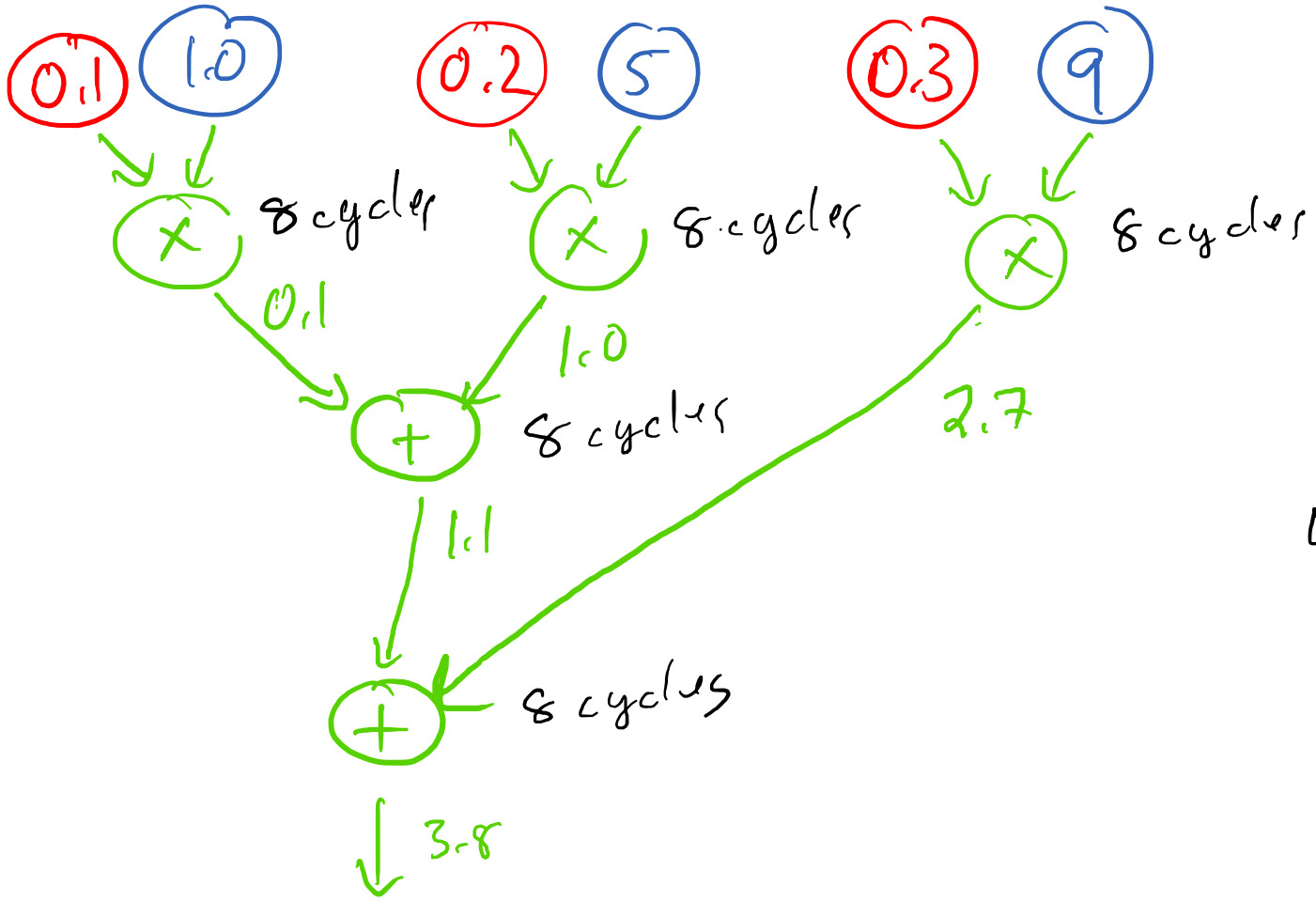
- A data-flow graph is **a collection of arcs and nodes in which the nodes are either places where variables are assigned or used**, and the arcs show the relationship between the places where a variable is assigned and where the assigned value is subsequently used.

[ref]


[0.1 0.2 0.3]
1 2 3 4
5 6 7 8
9 10 11 12
= [3.8 4.4 5 5.6]
independent
independent
0.1
1
0.2
5
0.3
9
0.1
1.0
2.7
dependency
dependency
dependency
dependency
3.8

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

40 cycles

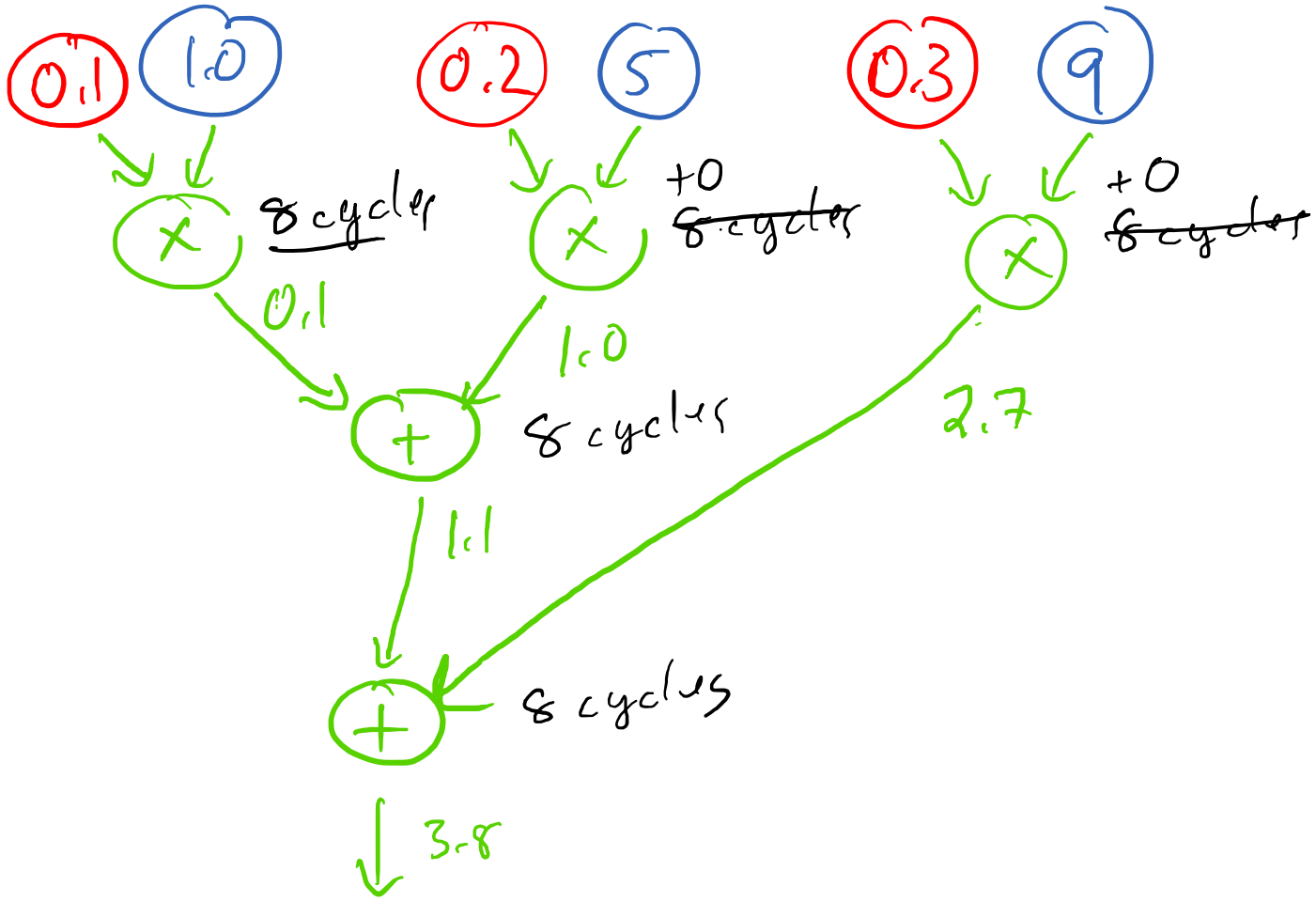


8 · 5 = 40 cycles

40 cycles · 4 #'s =

160 cycles

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



3x (multipliers or Adders)

8 + 0 + 0

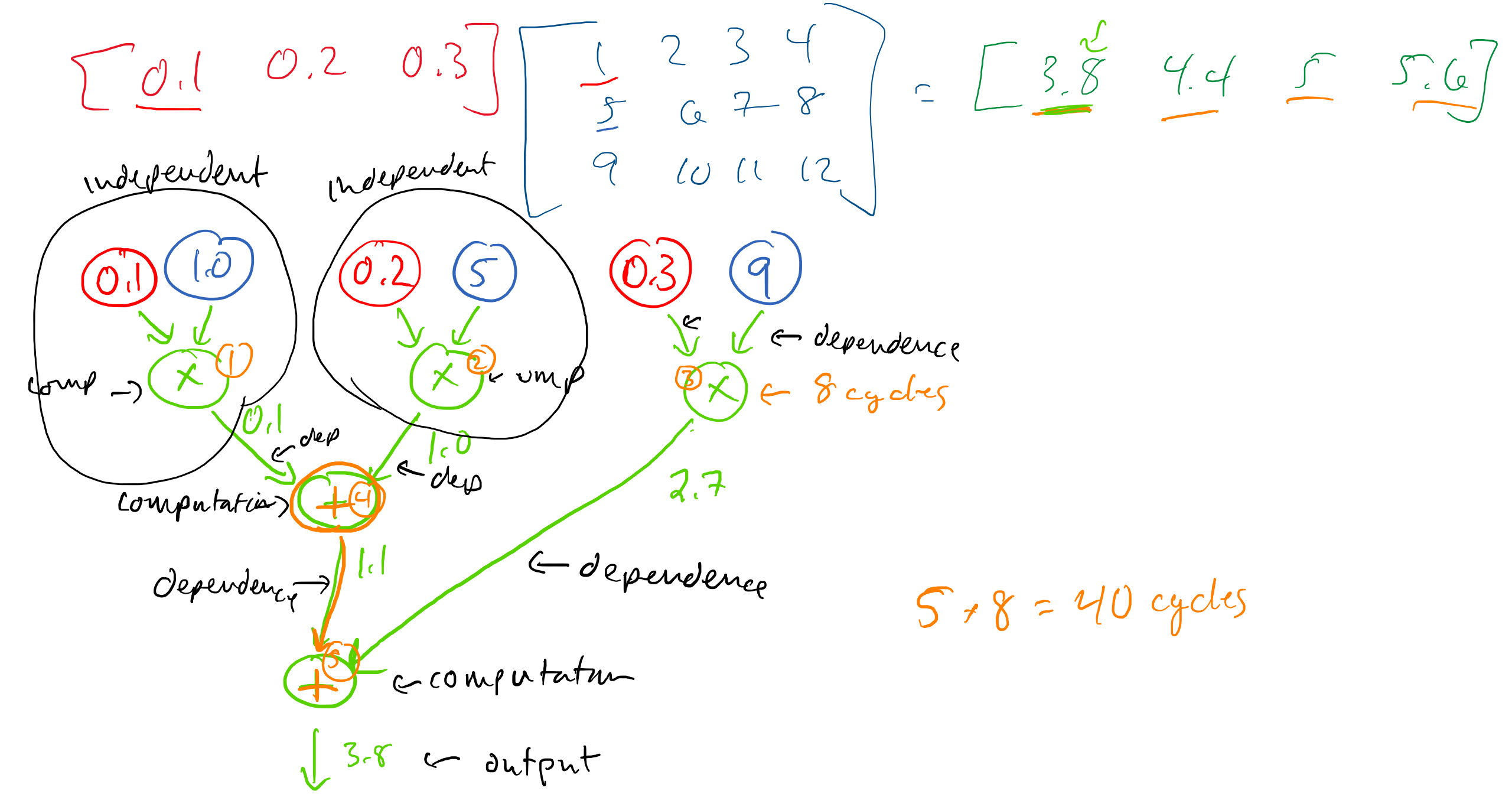
+ 8

+ 8

---

24 cycles / #

24 · 4 = 96 cycles


[0.1 0.2 0.3]
[1 2 3 4; 5 6 7 8; 9 10 11 12]
= [3.8 4.4 5 5.6]
independent
independent
0.1
1.0
0.2
5
0.3
9
comp
comp
dependence
8 cycles
0.1
dep
1.0
dep
computation
2.7
1.1
dependency
dependence
computation
3.8
output
5 * 8 = 40 cycles

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

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

$0.1 \cdot 1 + 0 = 0.1$ ← +8 cycles

→ $0.2 \cdot 5 + 0.1 = 1.1$ ← +8 cycles

$0.3 \cdot 9 + 1.1 = 3.8$ ← +8 cycles

→ result: 3.8 ← 8 + 8 + 8 = 24 cycles

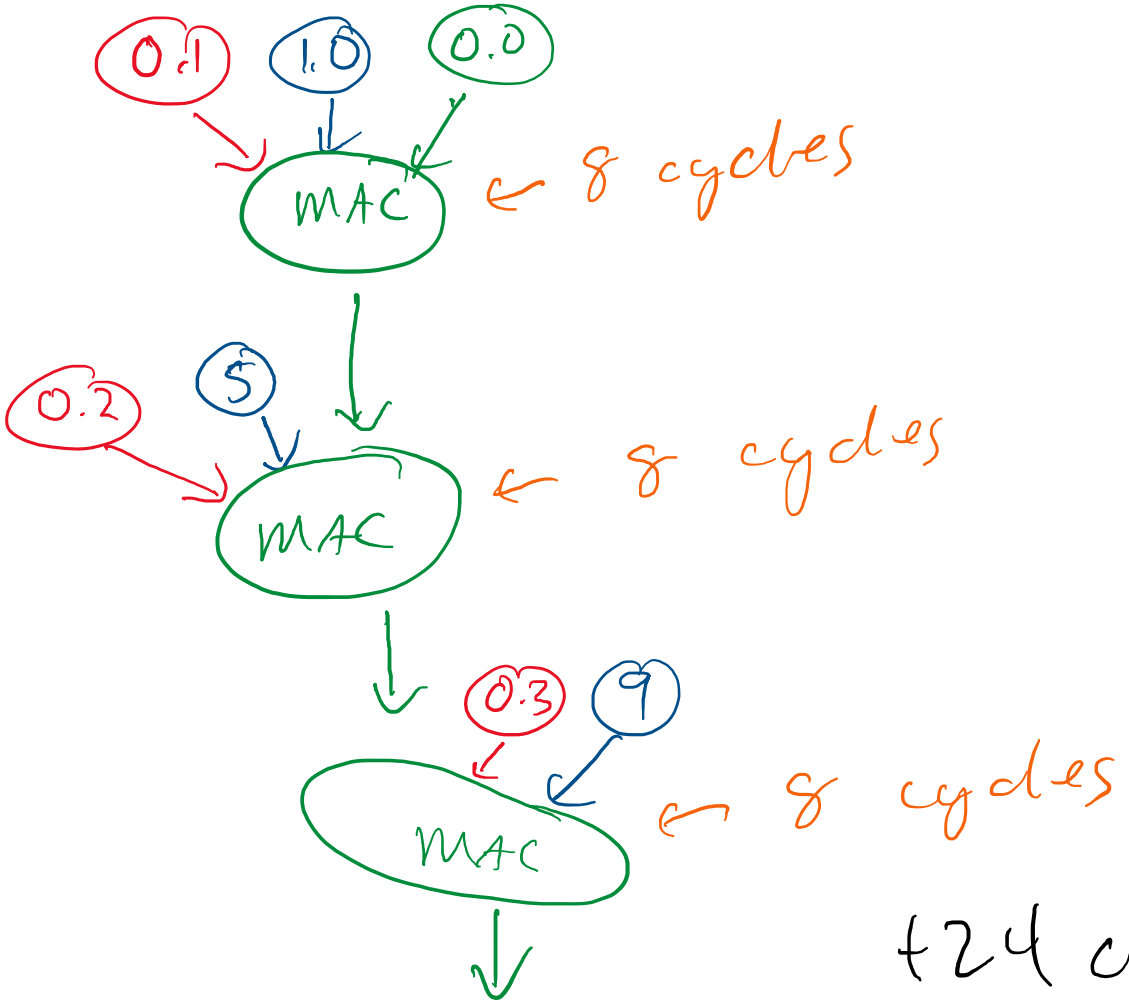
4.4 ← 24 cycles

5 ← 24

5.6 ← 24

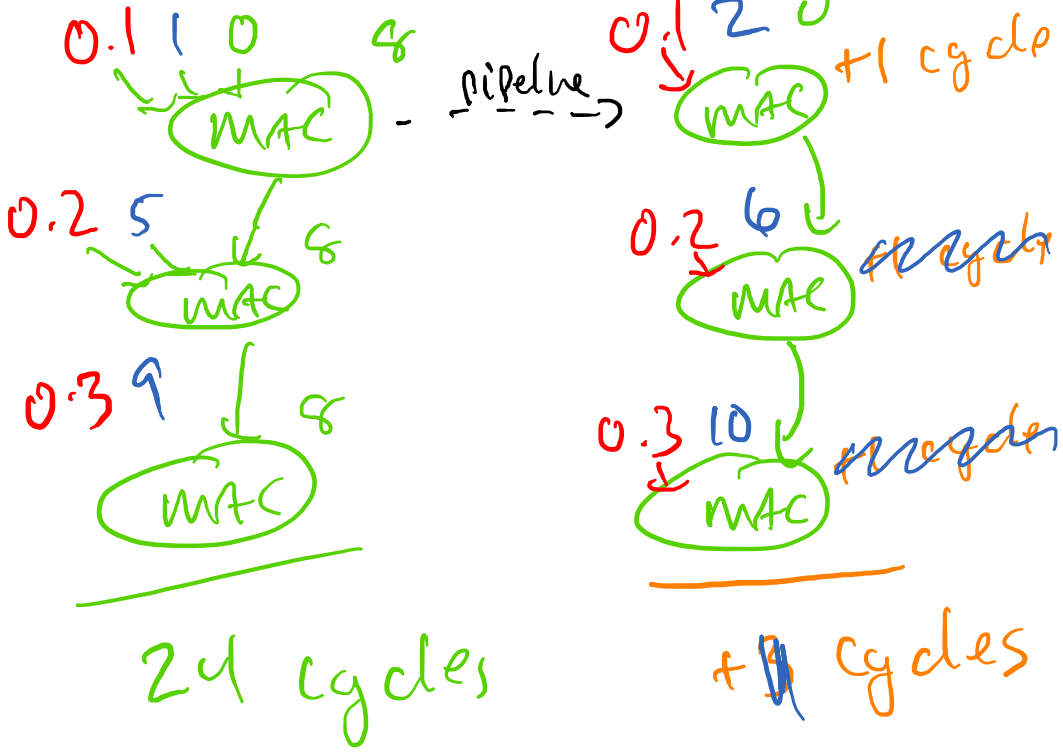
96 cycles

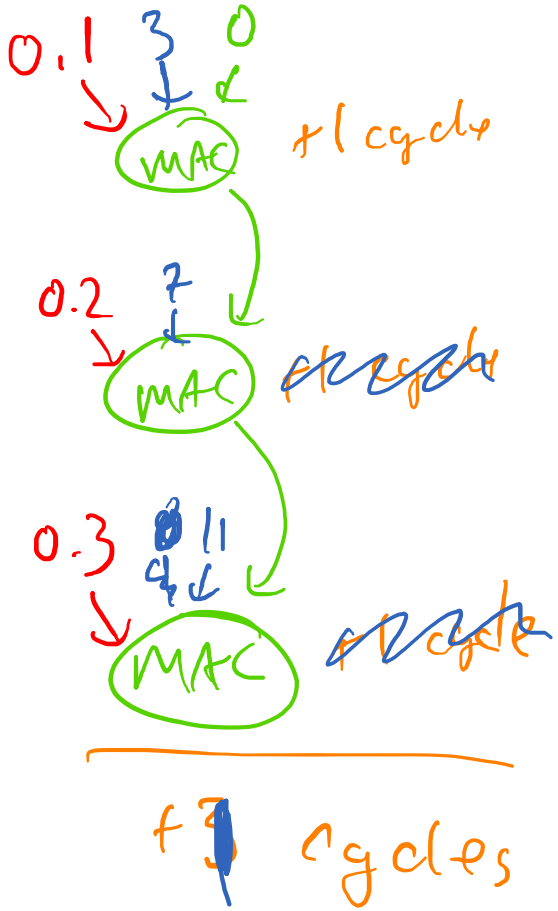
$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

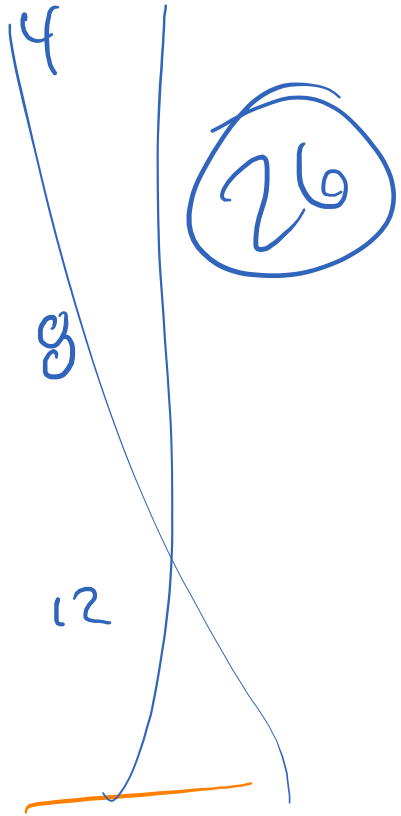


+24 cycles

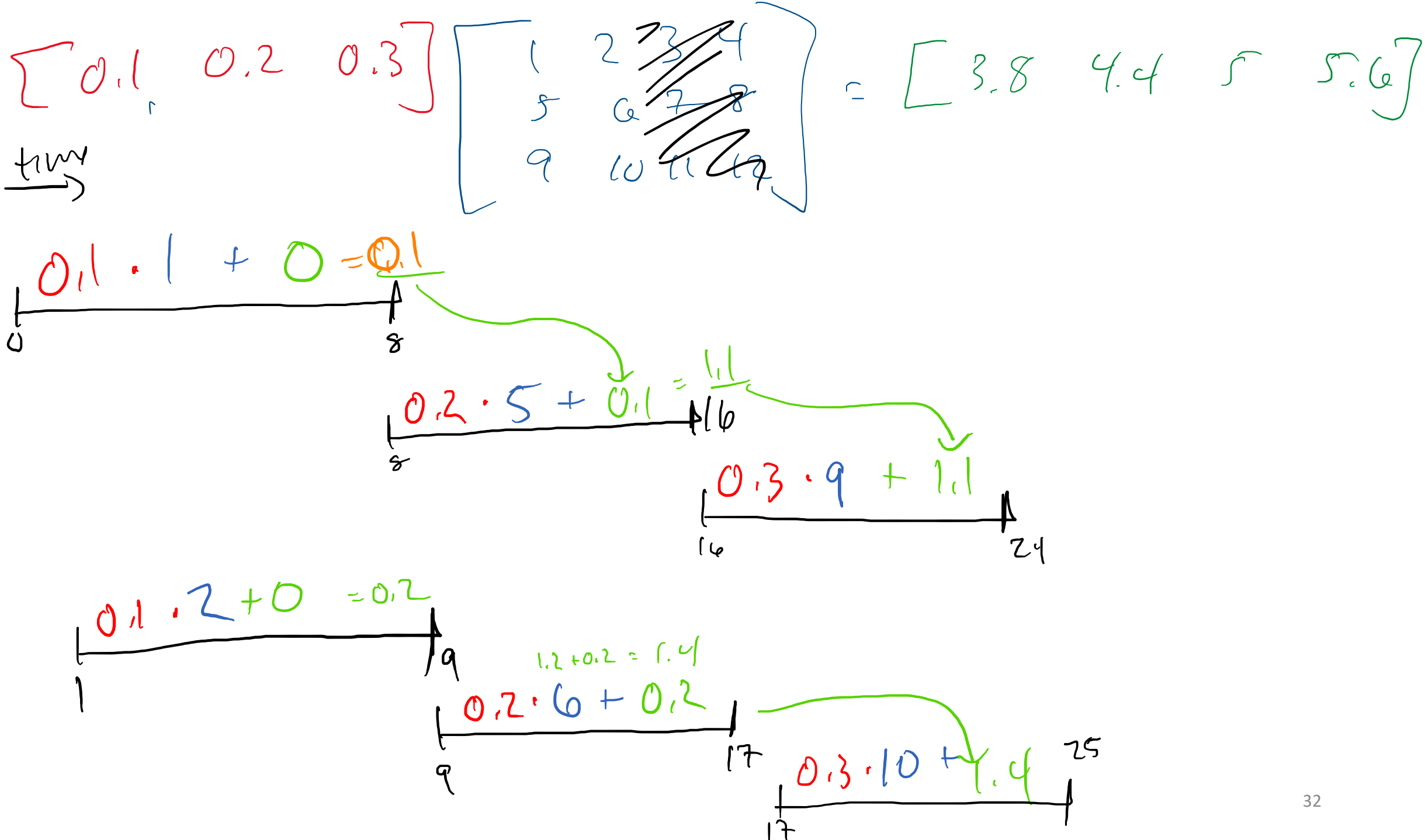
$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$







24 + 3 + 3 = 32 cycles


[0.1 0.2 0.3]
[1 2 3 4
5 6 7 8
9 10 11 12]
= [3.8 4.4 5 5.6]
time
0.1 · 1 + 0 = 0.1
0
8
0.2 · 5 + 0.1 = 1.1
8
16
0.3 · 9 + 1.1
16
24
0.1 · 2 + 0 = 0.2
1
9
1.2 + 0.2 = 1.4
0.2 · 6 + 0.2
9
17
0.3 · 10 + 1.4
17
25

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

Unpipelined:
8 cycles / MAC
X 4 MACs / input
X 3 inputs
---
96 cycles

pipelined:
8 cycles for 1st MAC
and +1 cycle for 2nd+ MAC
X 4 MACs / input
---
8 + 1 + 1 + 1 = 11 cycles

11 cycles / input
X 3 inputs
---
33 cycles

Hardware Parallelism

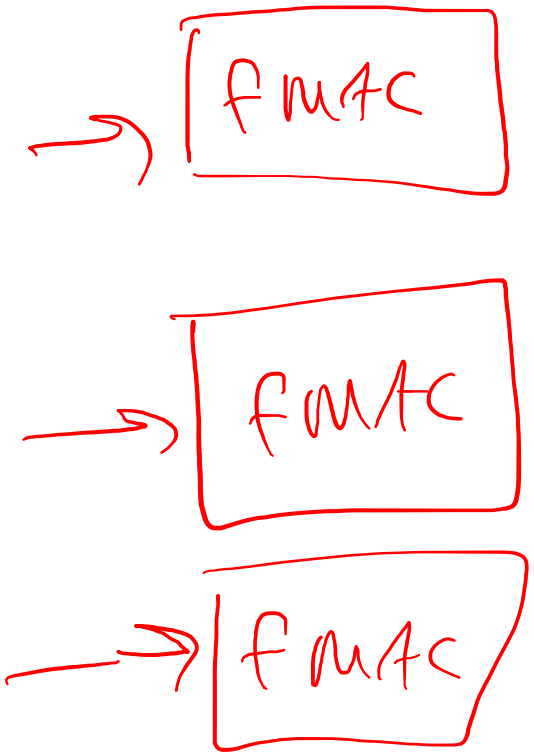
- CPU: 1 Floating-Point Unit
- FPGA? 10 Floating-Point Units?
  20 ?
  100 ?

Finding Parallelism

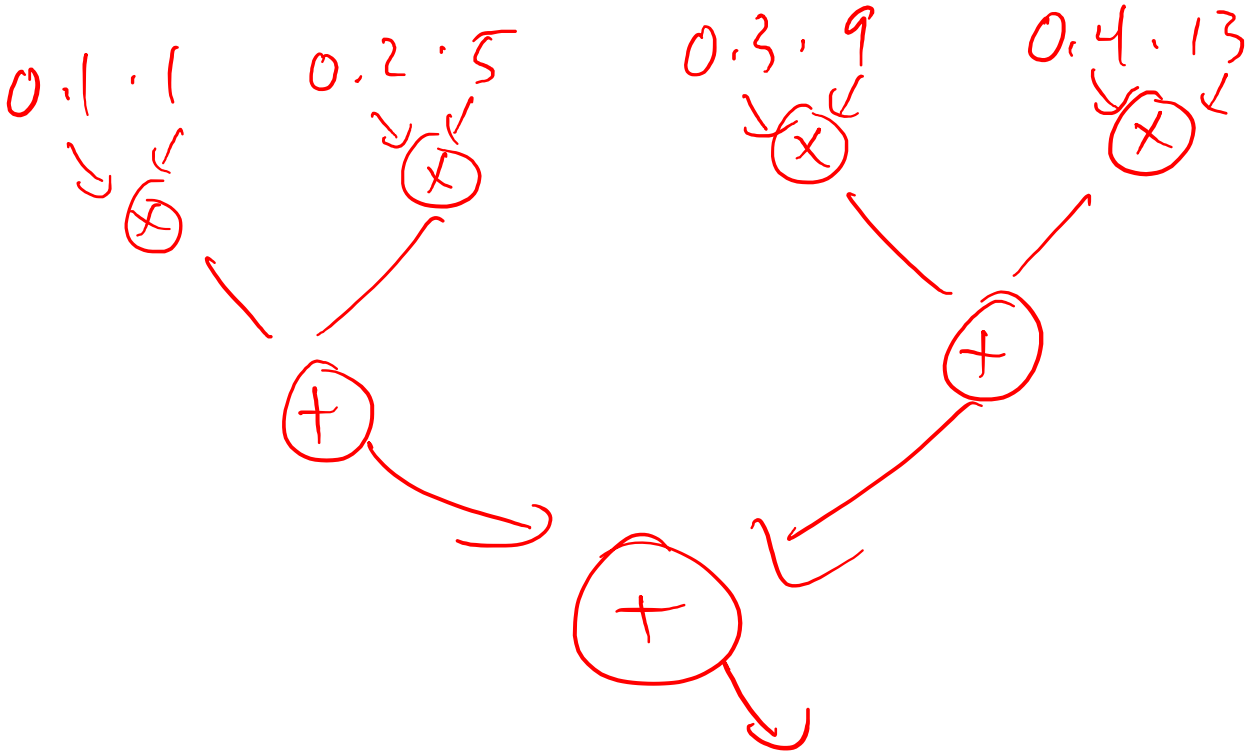
- Some some computation that doesn’t depend on other computation’s results
- Shared Inputs are OK.

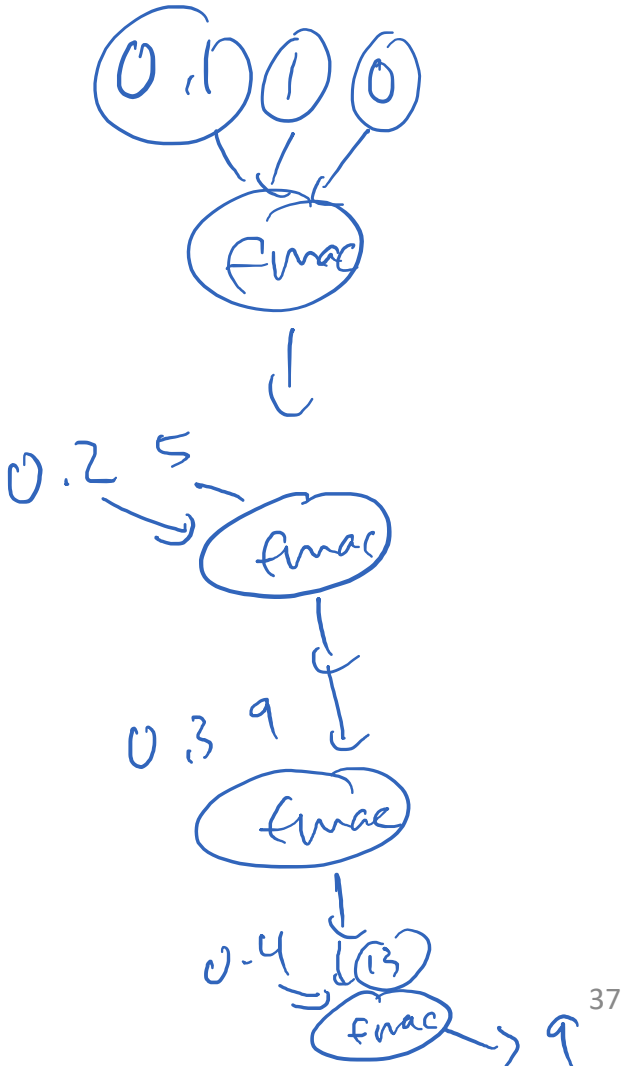
What can we do with 2+ FMACs?

dot



$$\begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} = \begin{bmatrix} 9 & 10 & 11 & 12 \end{bmatrix}$$





$$\begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} = \begin{bmatrix} 9 & 10 & 11 & 12 \end{bmatrix}$$

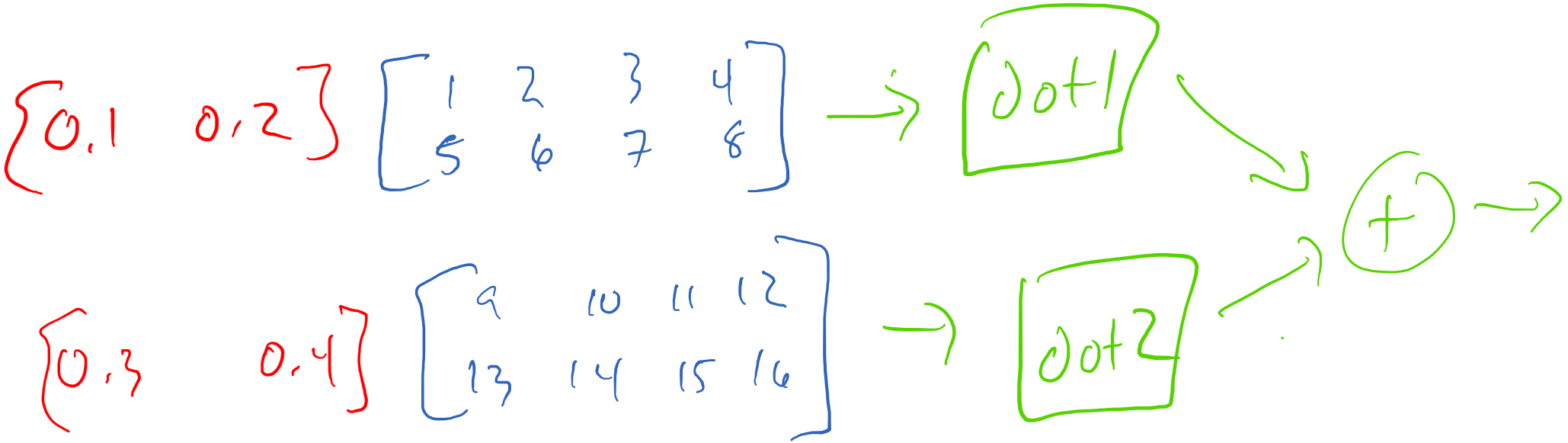
0.1 0.2 0.3 0.4
× 1 × 5 × 9 × 13
0.1 + 1 2.7 + 5.2
1.1 + 7.9
9

0.1 0.2 0.3 0.4
× 2 × 6 × 10 × 14
0.2 + 1.2 3.0 + 5.6
1.4 + 8.6
10

0.1 0.2 0.3 0.4
× 3 × 7 × 11 × 15
0.3 + 1.4 3.3 + 6.0
1.7 + 9.3
11

0.1 0.2 0.3 0.4
× 4 × 8 × 12 × 16
0.4 + 1.6 3.6 + 6.4
2.0 + 10
12

$$[0.1 \quad 0.2 \quad 0.3 \quad 0.4] \cdot \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} = [9 \quad 10 \quad 11 \quad 12]$$



$[0.1 \quad 0.2 \quad 0.3 \quad 0.4] \cdot \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} = [9 \quad 10 \quad 11 \quad 12]$

$[0.1 \quad 0.3] \begin{bmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \end{bmatrix} = [2.8 \quad 3.2 \quad 3.6 \quad 4.0]$

$[0.2 \quad 0.4] \begin{bmatrix} 5 & 6 & 7 & 8 \\ 13 & 14 & 15 & 16 \end{bmatrix} = [6.2 \quad 6.6 \quad 7.4 \quad 8]$

$+ \quad [9 \quad 10 \quad 11 \quad 12]$

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
```

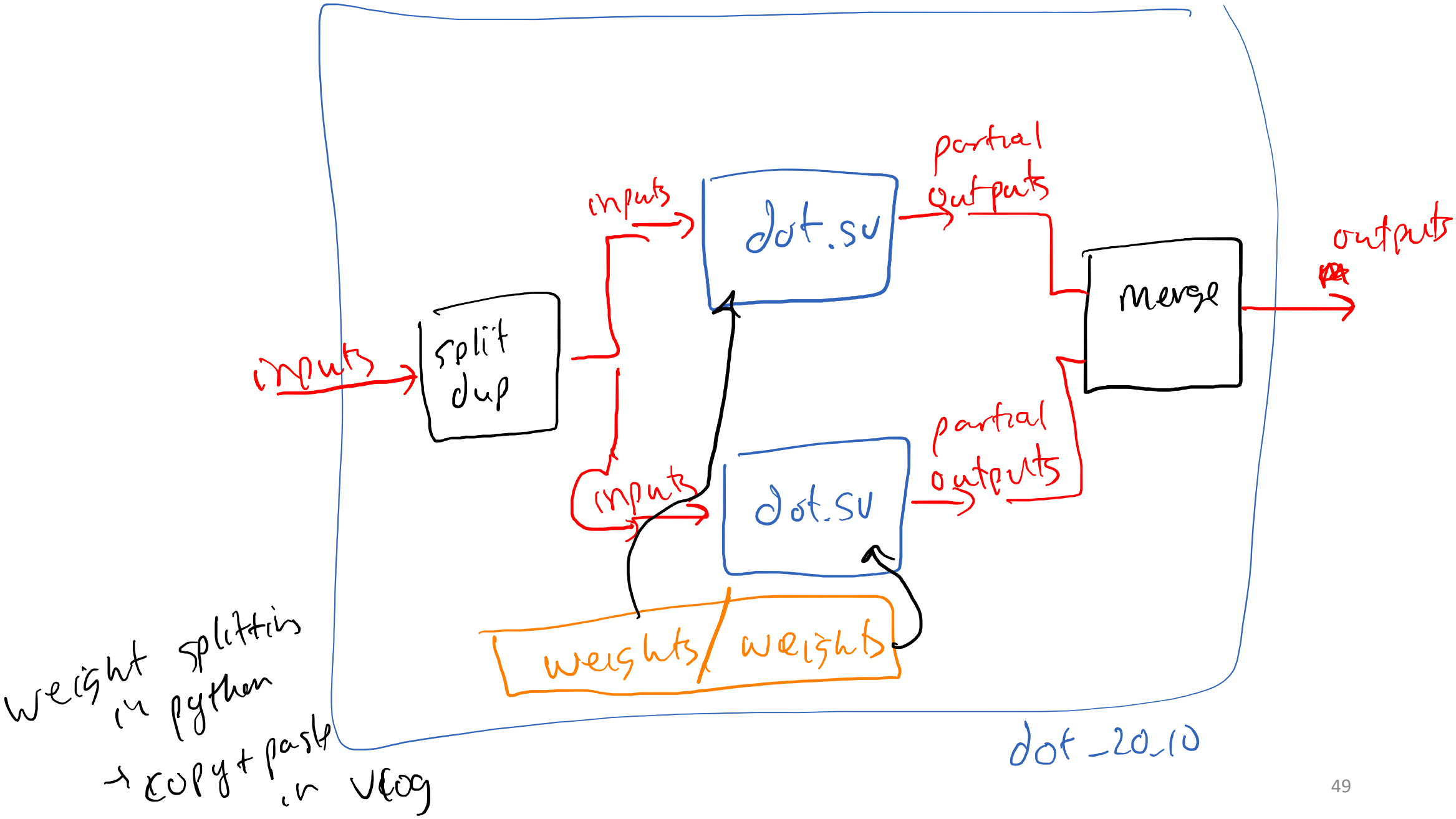
Can we parallelize Dot?

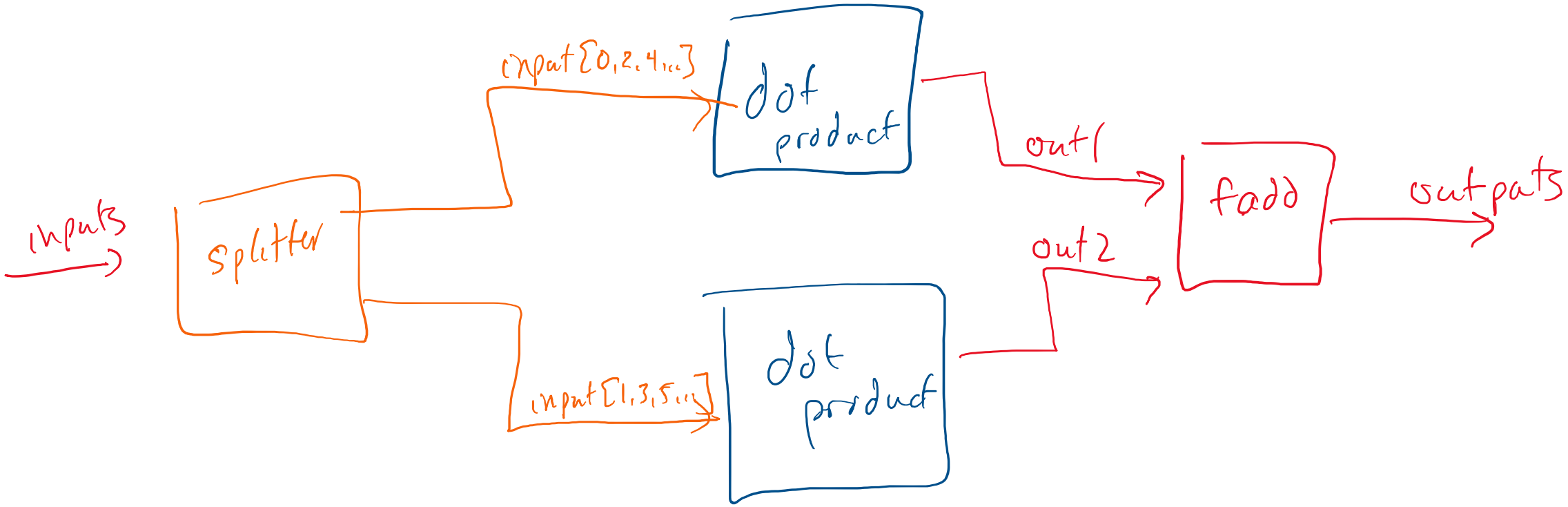
```python
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j]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


inputs
split dup
inputs
dot.sv
partial outputs
inputs
dot.sv
partial outputs
merge
outputs
weights
weights
dot_20_10
weight splitting in python
→ copy + paste in vlog


inputs
splitter
input[0,2,4,...]
input[1,3,5,...]
dot product
dot product
out1
out2
fadd
outputs

17: Hardware Acceleration IV

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University