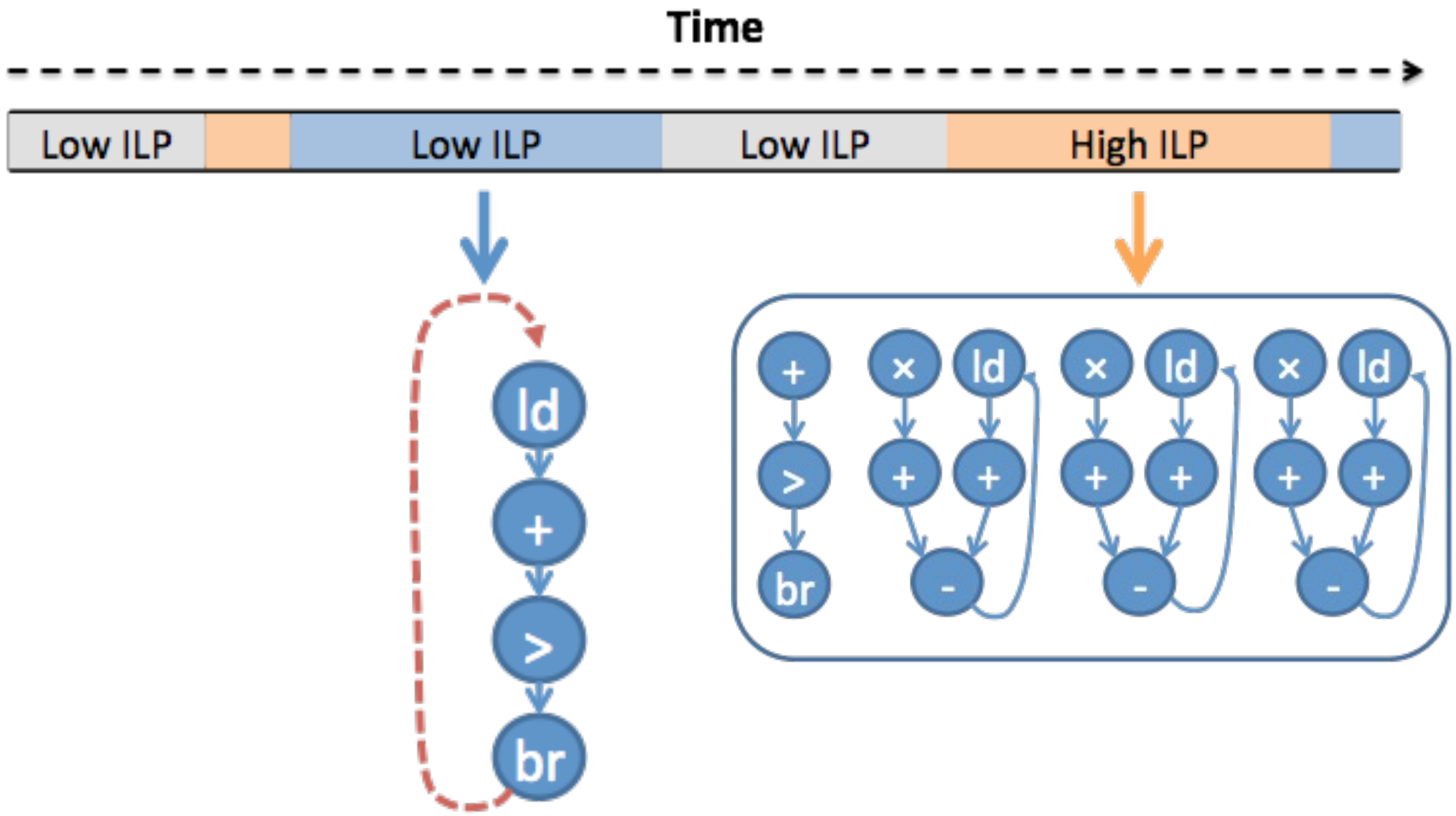
CMPT 450/750: Computer Architecture
Fall 2024
Domain-Specific Architecture II
How did we get here ?
What are they ?

Alaa Alameldeen & Arrvindh Shriraman

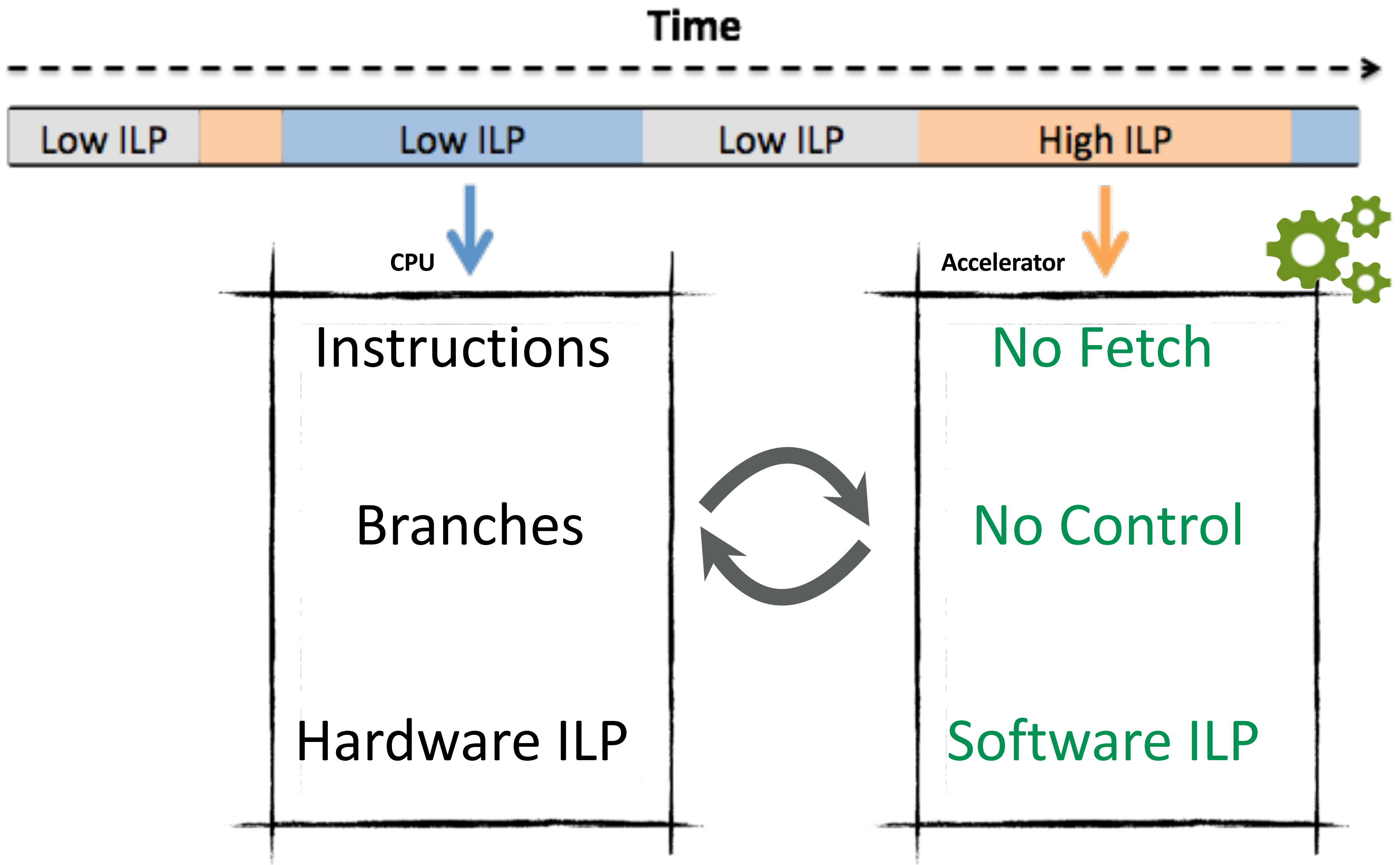
Recall: ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level
- ISA: Specifies how the programmer sees the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order
- Microarchitecture: How the underlying implementation actually executes instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
    - Programmer should see the order specified by the ISA

What are Accelerators?



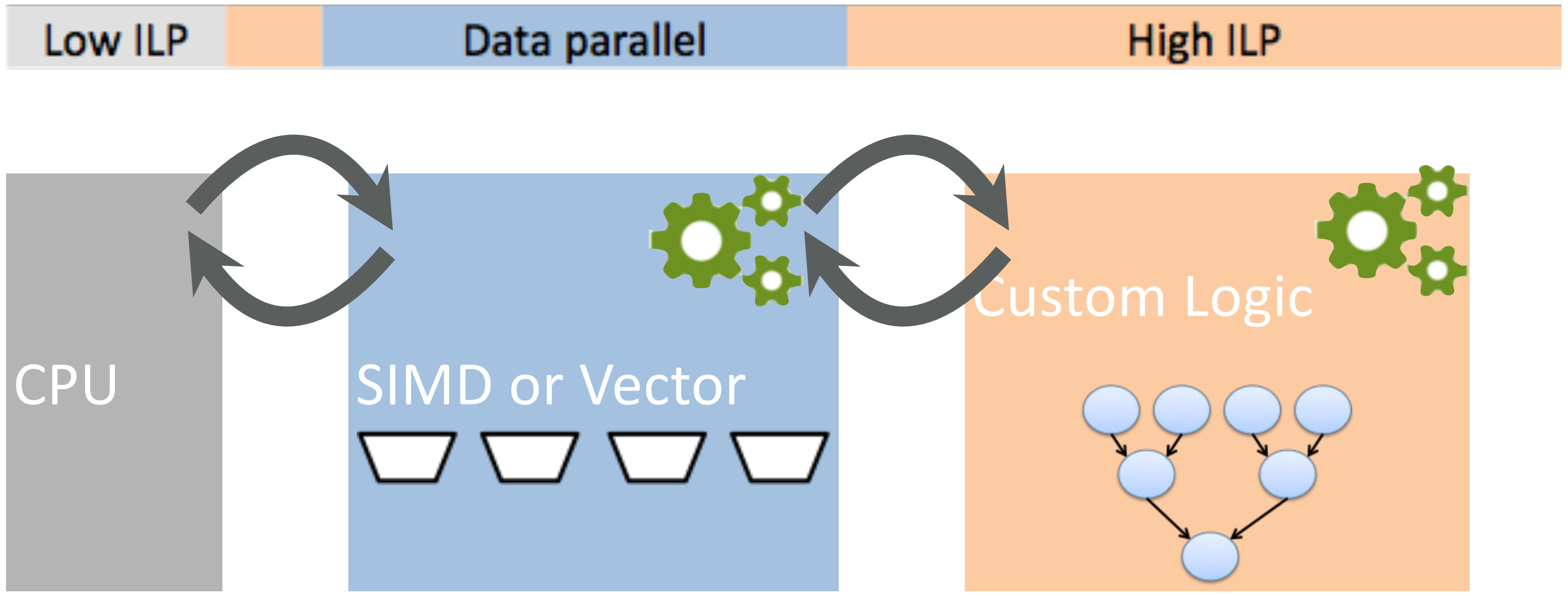
What are Accelerators?



[Intel Harp, IBM CAPI, ARM Big-Little, BERET, DYSER, CCORE]

Accelerator Execution

- Hope! Large acceleratable program regions

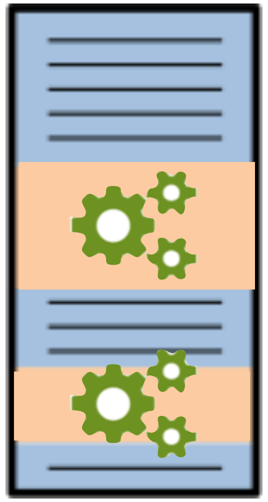


Why does it work?

- **Applications execute in phases**
- **Applications follow 90-10 rule**
  - 10% of code-region contributes to 90% of run time
- ***Creating specialization for such code-regions amortizes the overheads***
  - Removing instructions from main pipeline
    - Less use of Instruction Queue, ROB, Register File
    - Effectively larger instruction window
  - Decoupled Execution
    - Concurrency between main processor and CGRA
    - Many FUs -> High Potential ILP
  - Benefits of Vectorization
    - Fewer memory access instructions
    - Explicit pipelining of CGRA

How can software help accelerators?

- Challenge 1: Find acceleratable programs regions

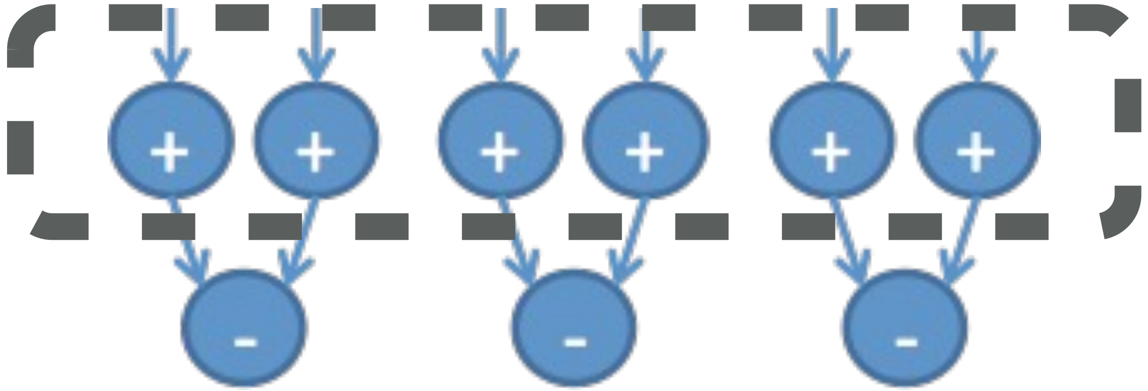


Control not supported (need SW help)

Mem. ops not supported (self prophecy?)

- Challenge 2: Identifying accelerator types

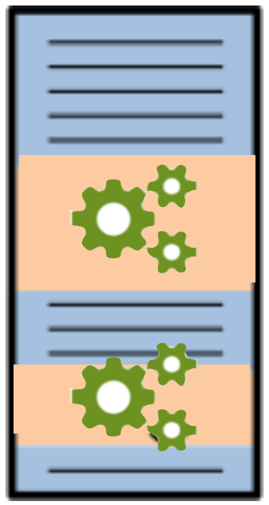
SIMD

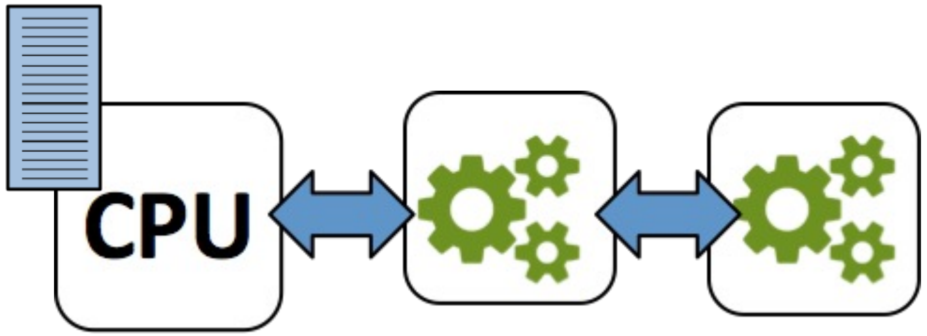


How can software define accelerators?

SFU

- Challenge 3: How to compose accelerators?





Accelerator Granularity

SFU

FPGA	Algorithm
GPUs	Threads
Onchip-FPGA	Extended Basic Blocks
Loop Accelerators	Program loops
SIMD	Instructions

Types of Accelerators?

SFU

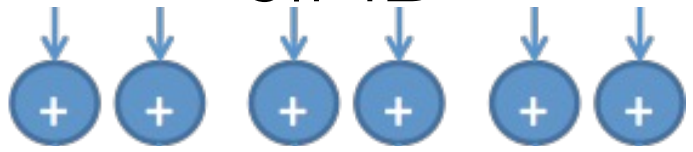
Control regularity

Memory regularity

Manycore

Data Parallel region

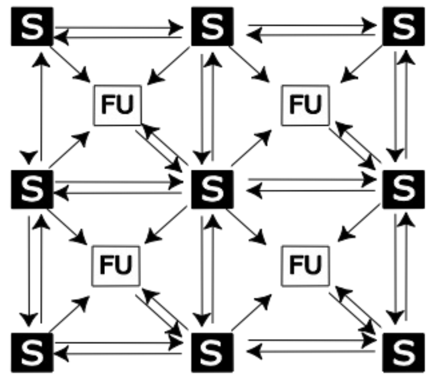
SIMD



Irregular

OOO CPU

Dynamic ILP region



Achieving ASIC Efficiencies: Getting to 500x

Need basic ops that are extremely low-energy

- Function units have overheads over raw operations
- 8-16 bit operations have energy of sub pJ
  - Function unit energy for RISC was around 5pJ

And then don't mess it up

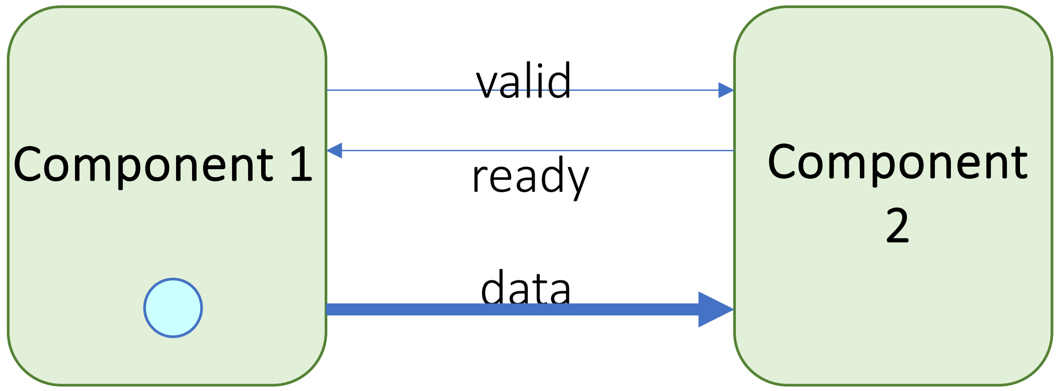
- "No" communication energy / op
  - This includes register and memory fetch
- Merging of many simple operations into mega ops
  - Eliminate the need to store / communicate intermediate results

Understanding Sources of Inefficiency in General-Purpose Chips, Hameed et al., ISCA 2010

Domain Specific Architecture = Compiler-Driven Spatial Hardware

Dataflow Execution

- Implement **dynamic scheduling**
- Every component communicates via a pair of handshake signals
- The data is propagated from component to component as soon as dependencies are resolved; fire when sources are ready



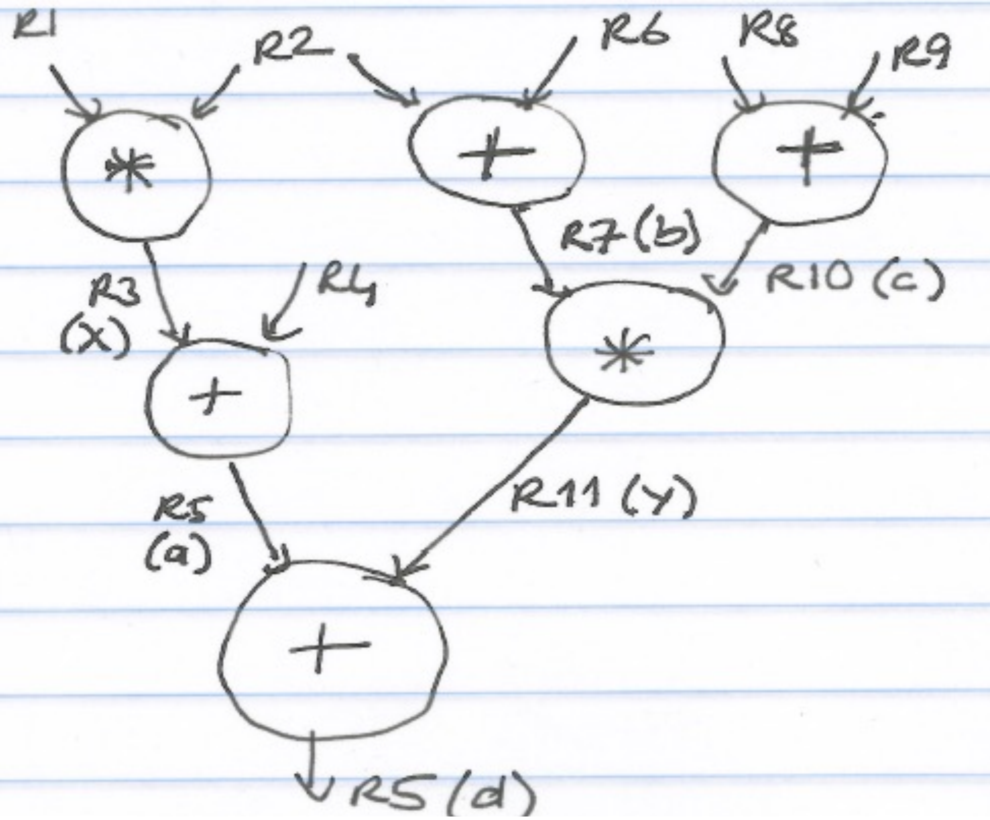
Recall: Dataflow Graph

```
MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)
```

Dataflow graph

Nodes: operations performed by the instruction

Arcs: tags in Tomasulo's algorithm



We can "easily" reverse-engineer the dataflow graph of the executing code!

Compiler Demo

Dataflow Execution Model

- Dataflow by nature has write-once semantics
- Each arc (token) represents a data value
- An arc (token) gets transformed by a dataflow node into a new arc (token)
  No persistent state…

Eliminates per instruction overheads

  No fetch, decode etc.,

  No expensive register reads etc.,

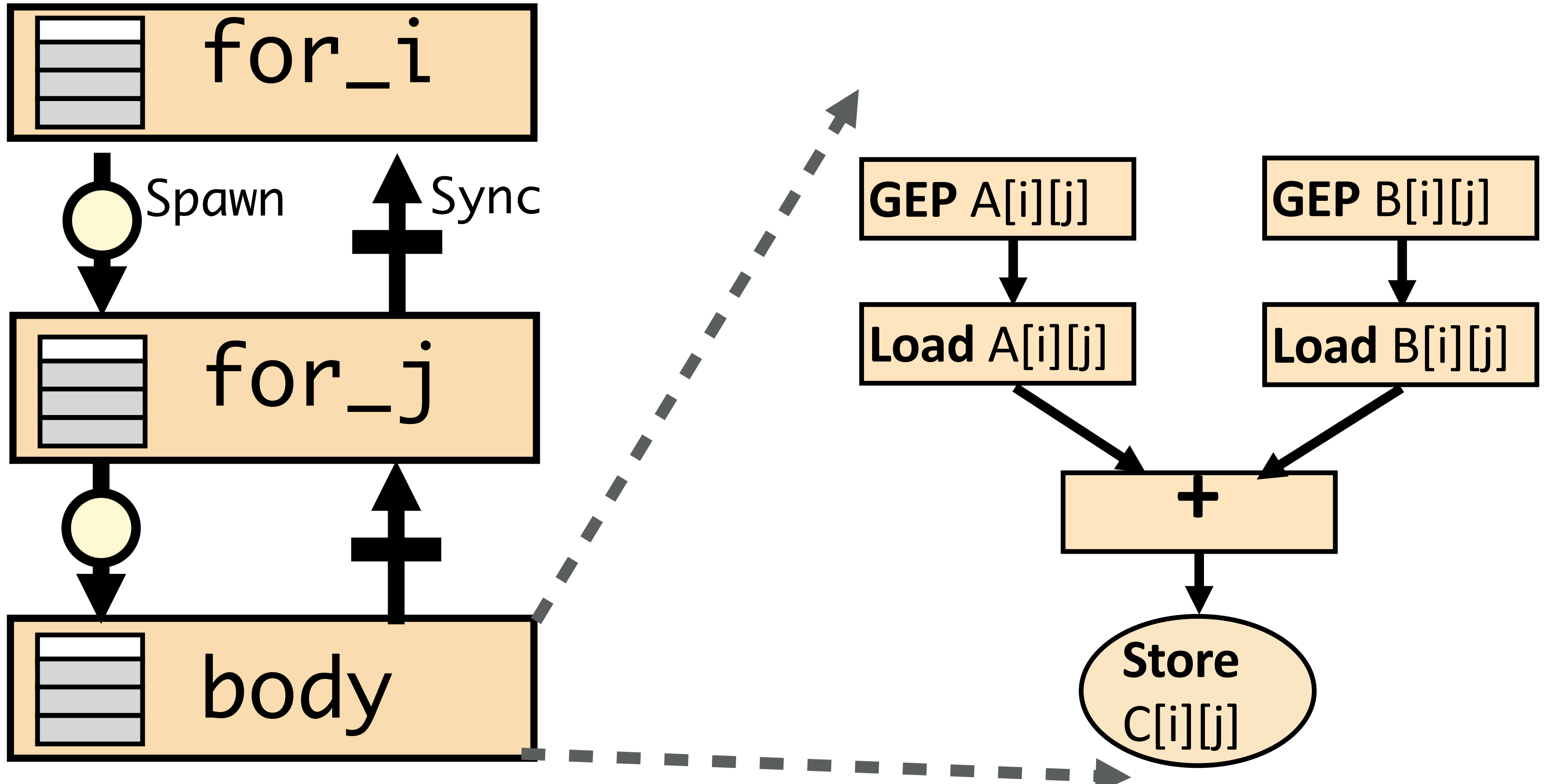
High performance itself leads to energy savings

  No additional power-hungry structures

Hierarchical Data & Control

```
parallel_for(i = 0 until n)
  parallel_for(j = 0 until n)
    c[i][j] = a[i][j] + b[i][j];
```

Hierarchical Data + Control Dynamic Graph



Loop Unrolling to Eliminate Branches

```
for (int i = 0; i < N; i++){

  A[i] = A[i] + B[i];

}
```

```
for (int i = 0; i < N; i+=4){

  A[i]   = A[i]   + B[i];
  A[i+1] = A[i+1] + B[i+1];
  A[i+2] = A[i+2] + B[i+2];
  A[i+3] = A[i+3] + B[i+3];

}
```

- Idea: Replicate loop body multiple times within an iteration

+ Reduces loop maintenance overhead
  - Induction variable increment or loop condition test

+ Enlarges basic block (and analysis scope)
  - Enables code optimization and scheduling opportunities

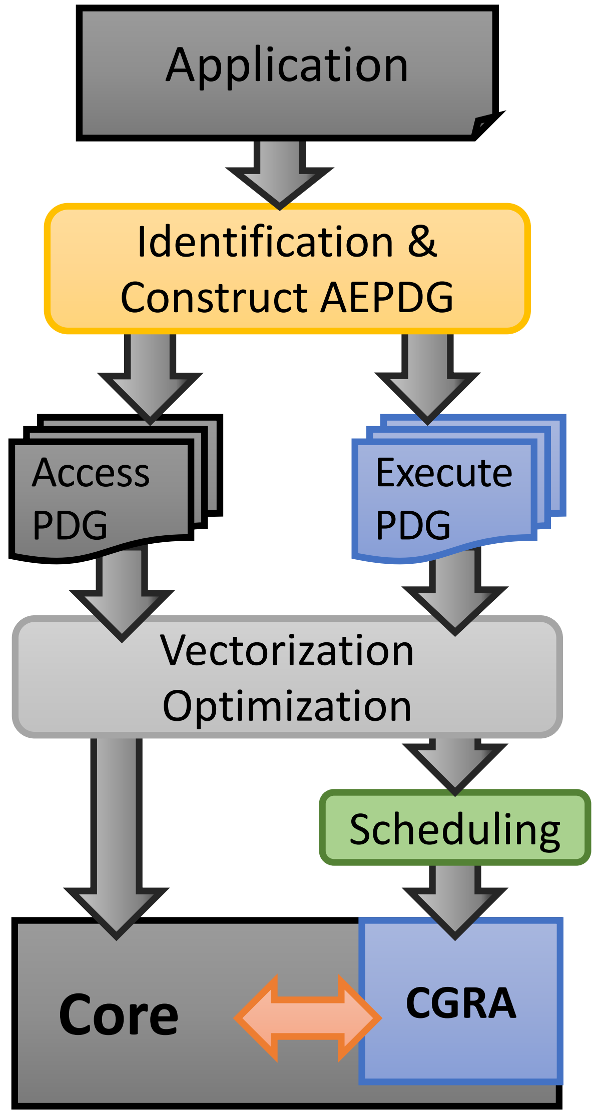
-- What if iteration count not a multiple of unroll factor? (need extra code to detect this)

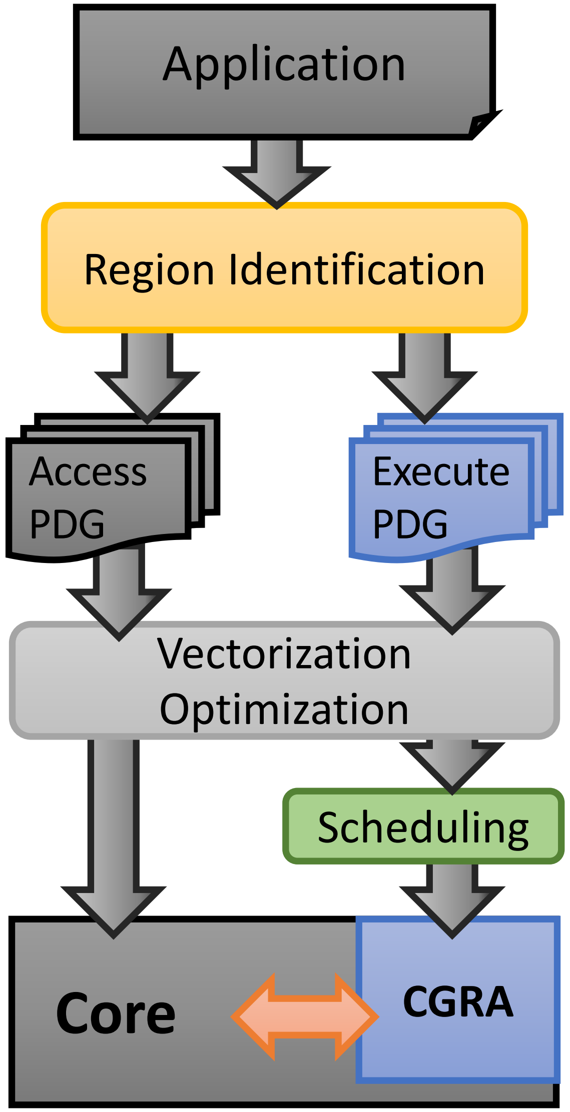
-- Increases code size

Compilation Tasks

SFU

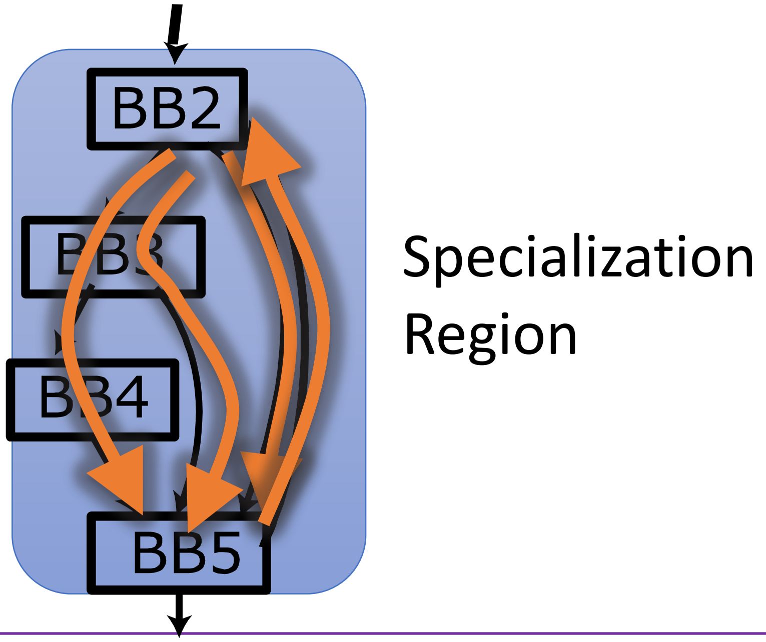
- Identify code-regions/loops to specialize
- Construct AEPDG
  - Access PDG
  - Execute PDG
- Perform Vectorization/ Optimizations
- Schedule
  - Execute PDG to CGRA
  - Access PDG to core

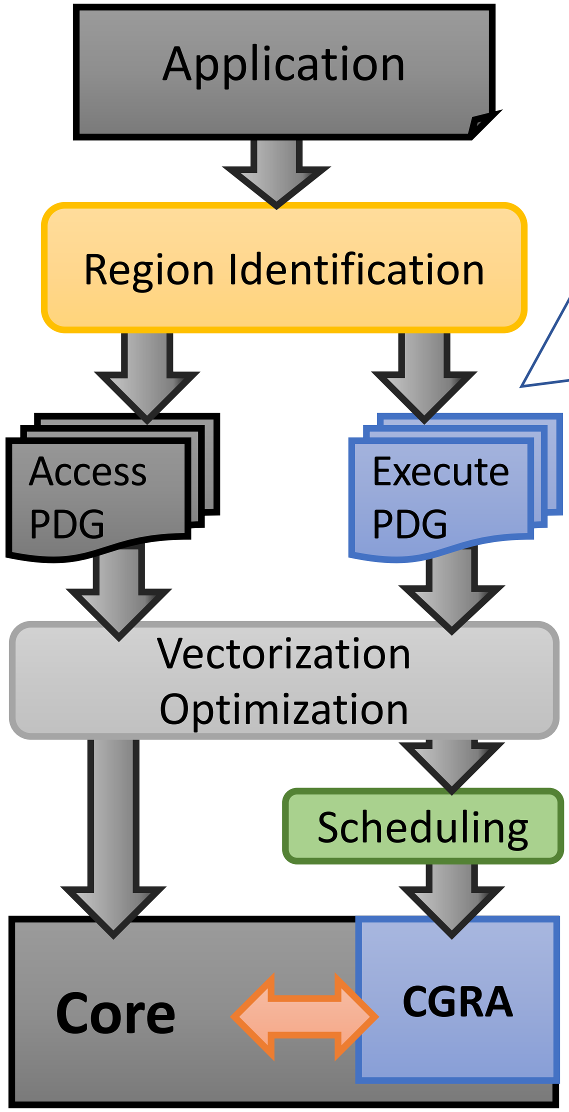




Region Identification

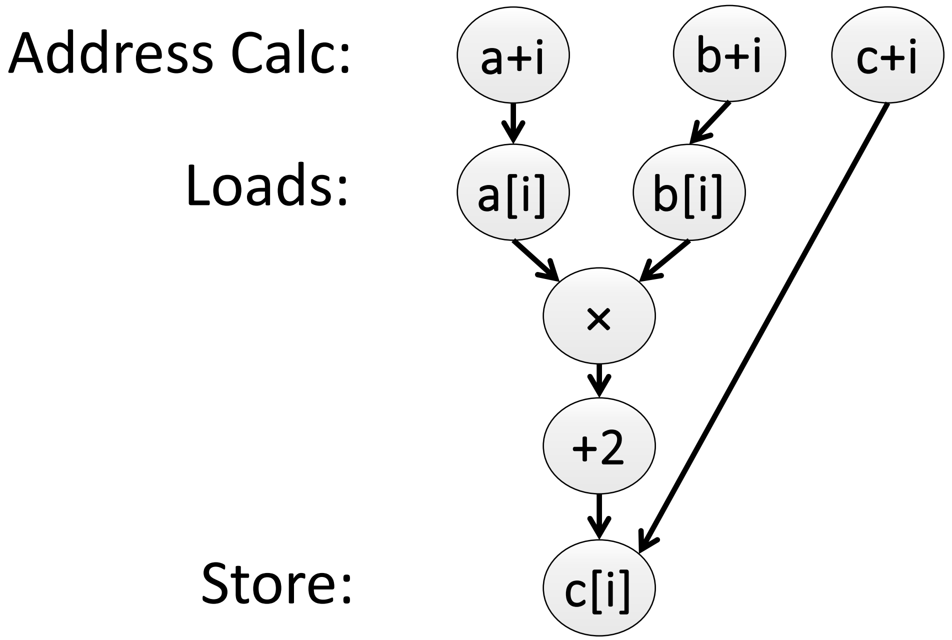
- Identify code-regions to specialize
  - Path Profiling
  - Utilize Loops
- Need Single-Entry / Single Exit Region



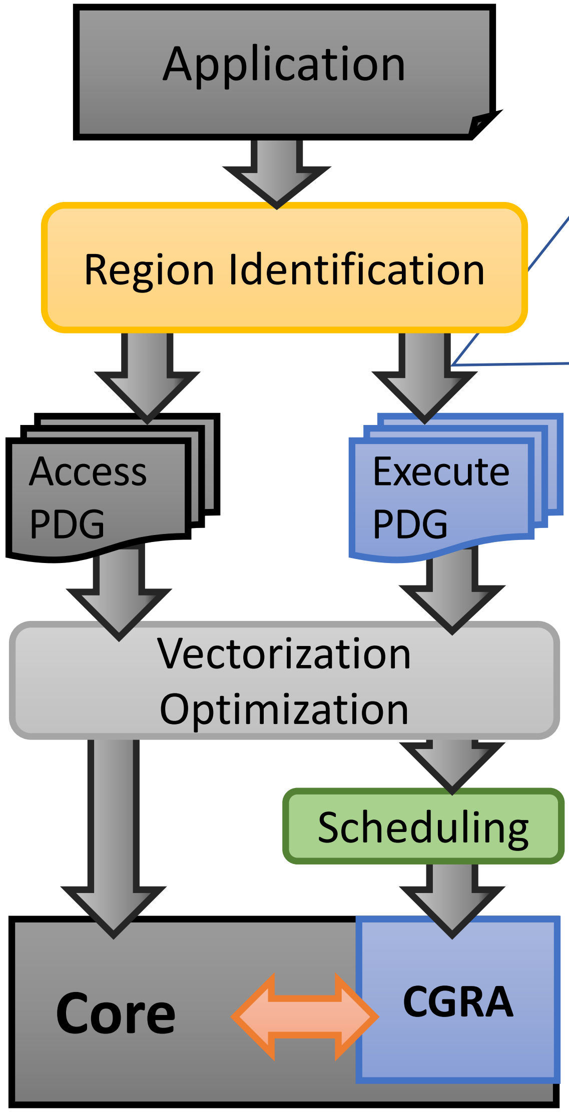


Construct AEPDG

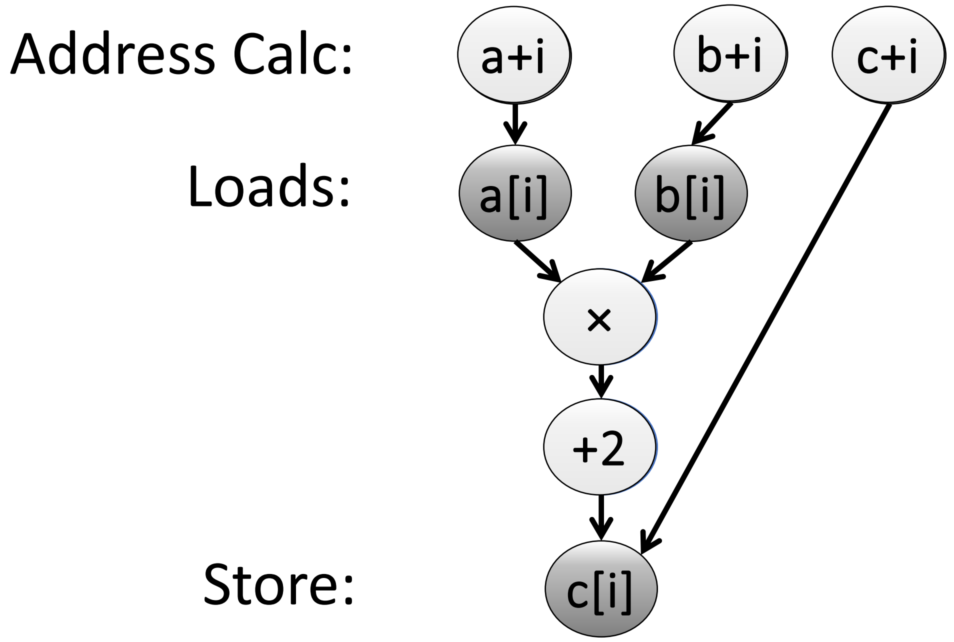
- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.

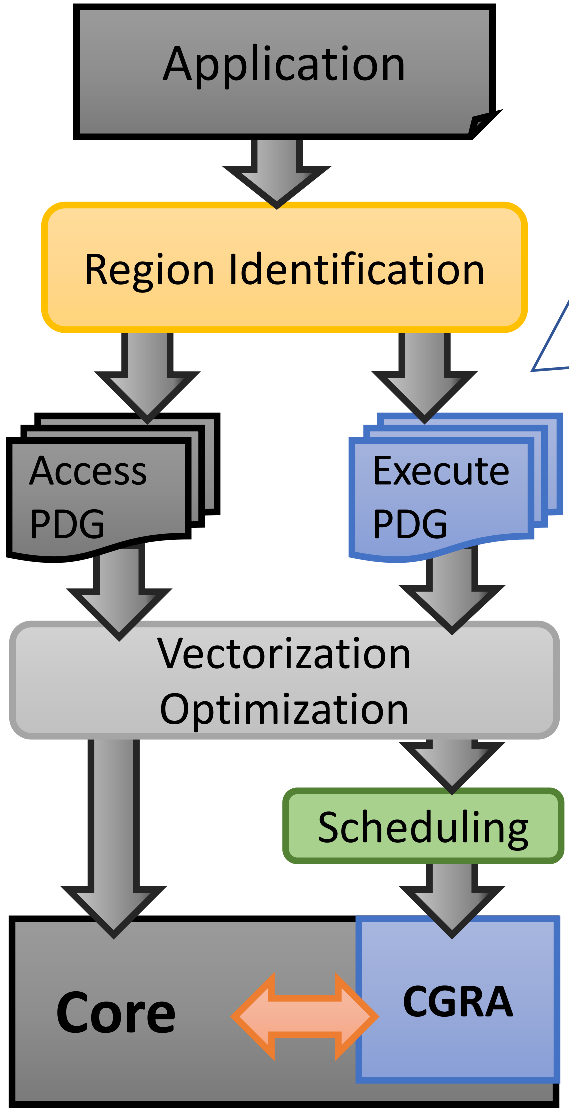


Construct AEPDG



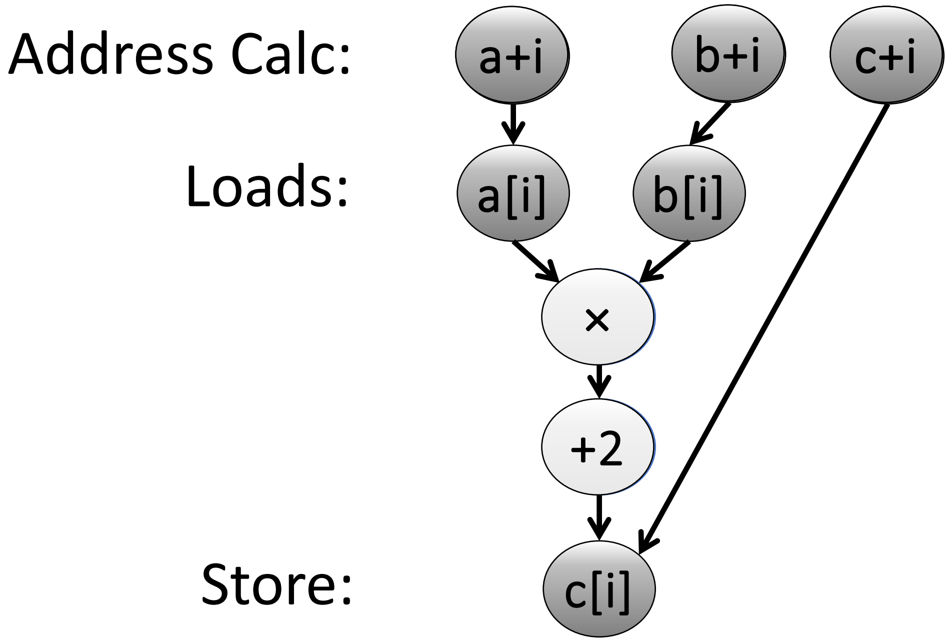
- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.

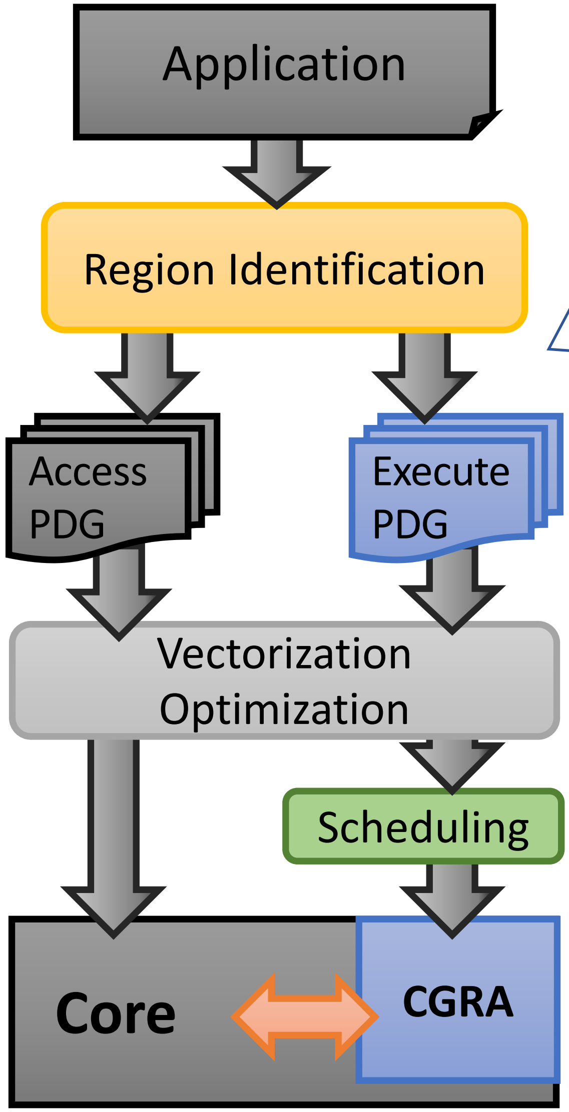




Construct AEPDG

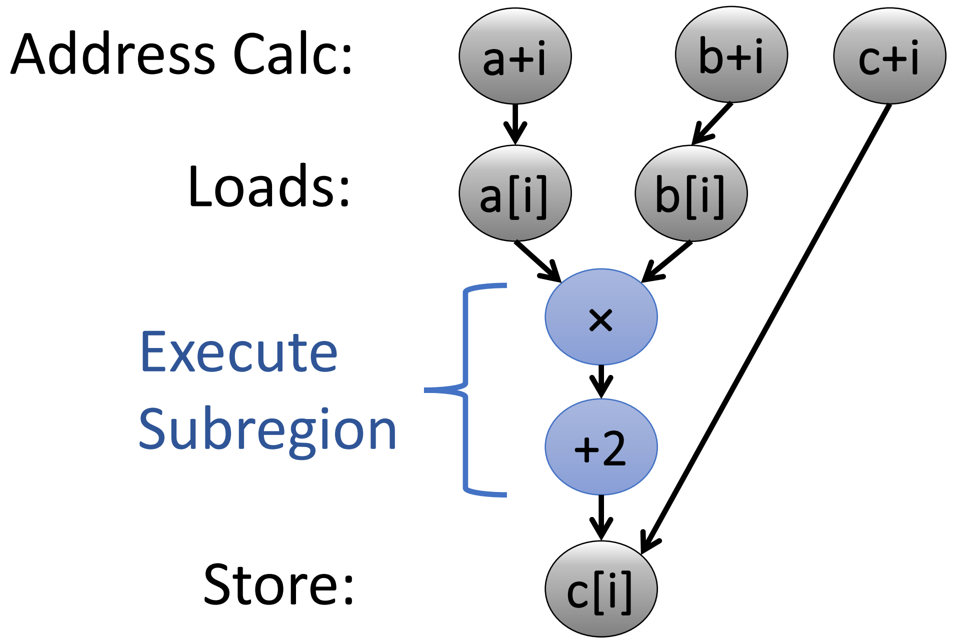
- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.



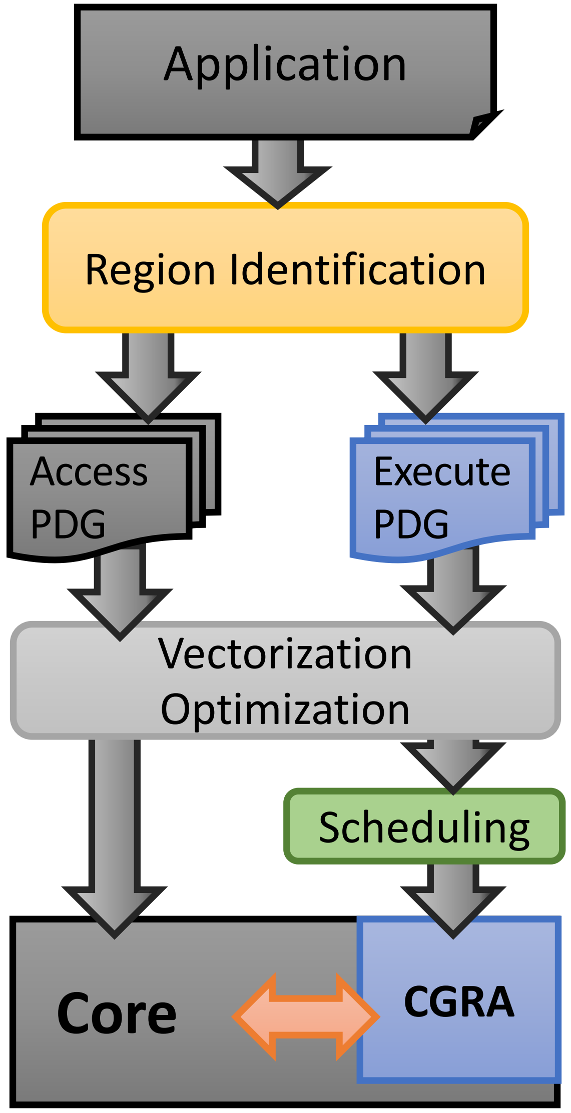


Construct AEPDG

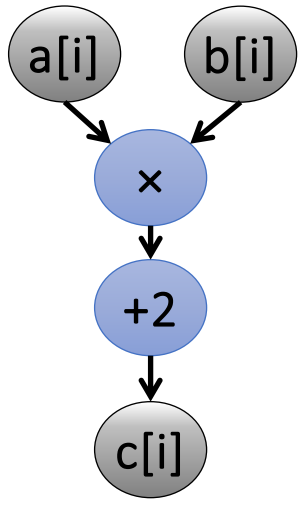
- Build Program Dependence Graph
- Separate memory access from computation.
- Loads/Stores and all dependent computation are access.

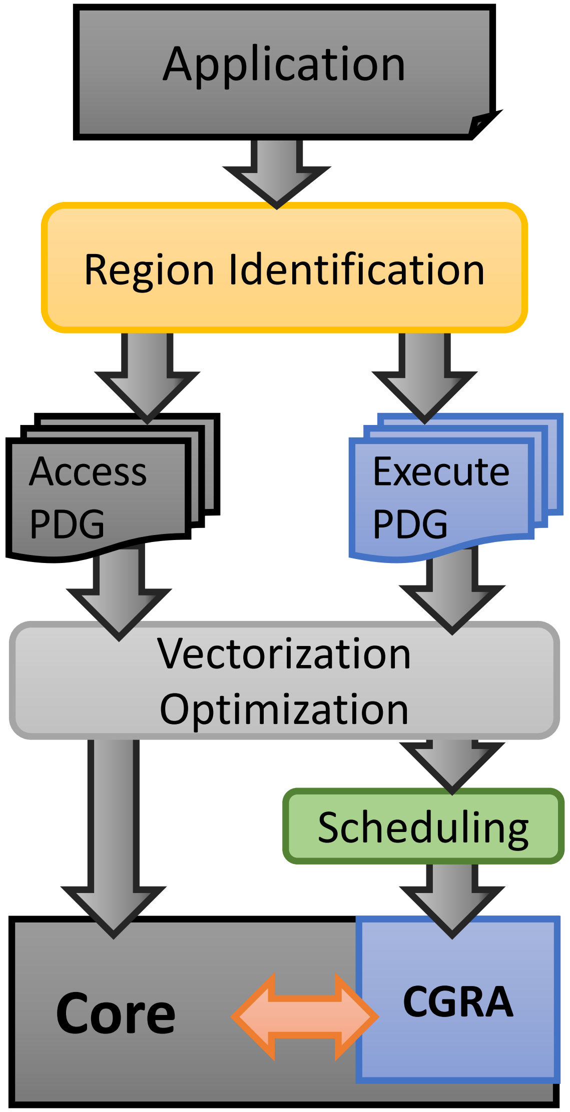


Vectorization



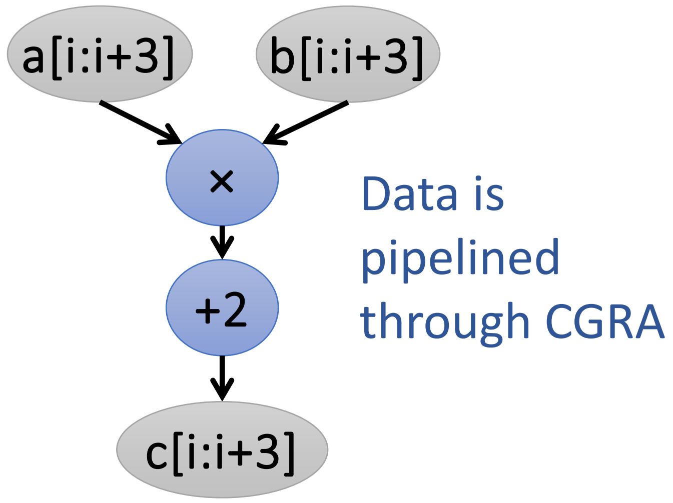
- Similar to SIMD Techniques, loops must have:
  - Independent Iterations
  - Must be no Store/Load Aliasing
- Memory Access: No gather/scatter
- Perform Loop Control
  - Modify trip count/peel scalar loop





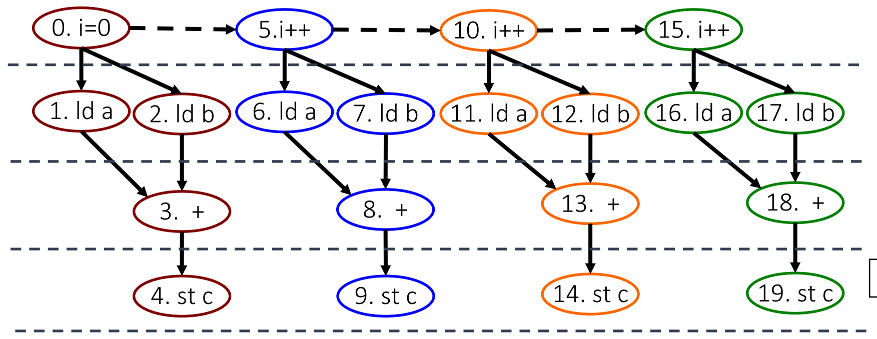
Vectorization

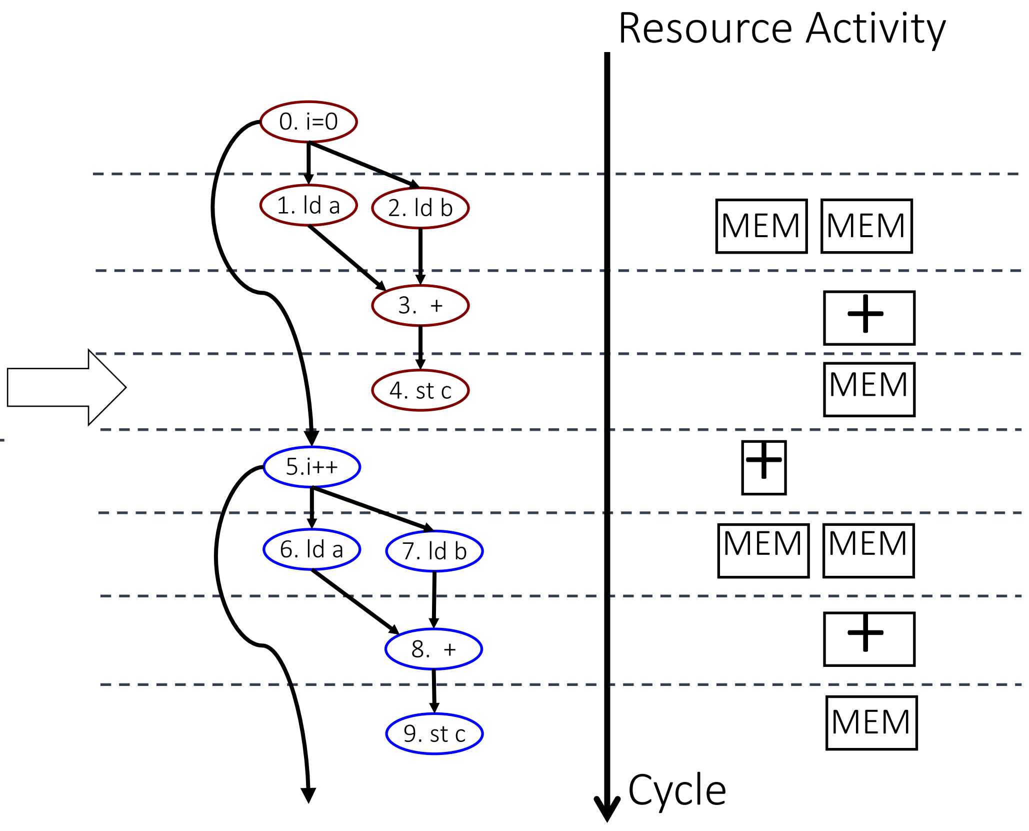
- Similar to SIMD Techniques, loops must have:
  - Independent Iterations
  - Must be no Store/Load Aliasing
- Memory Access: No gather/scatter
- Perform Loop Control
  - Modify trip count/peel scalar loop



Scheduling 101

Idealistic DDDG



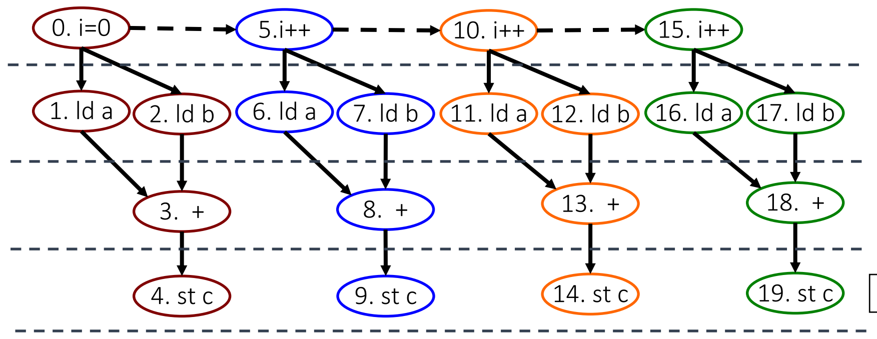


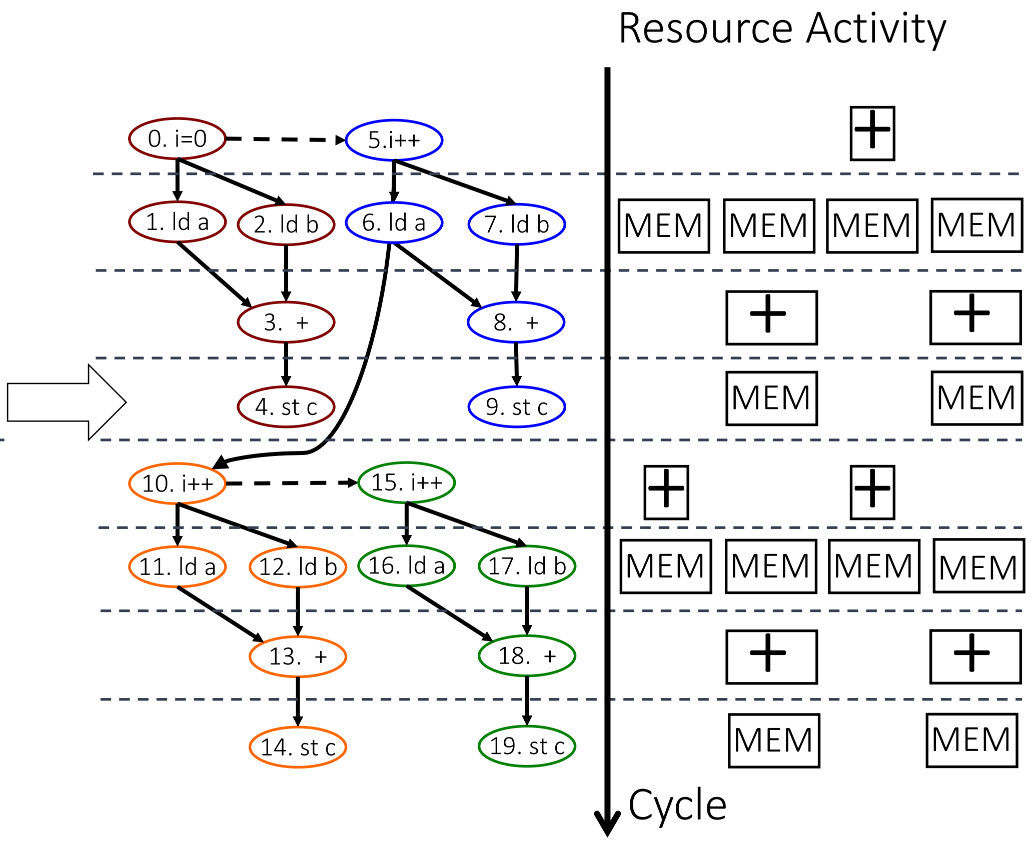
Acc Design Parameters:

- ✓ Memory BW <= **2**
- ✓ **1** Adder

Scheduling 101

Spatio-Temporal Dataflow

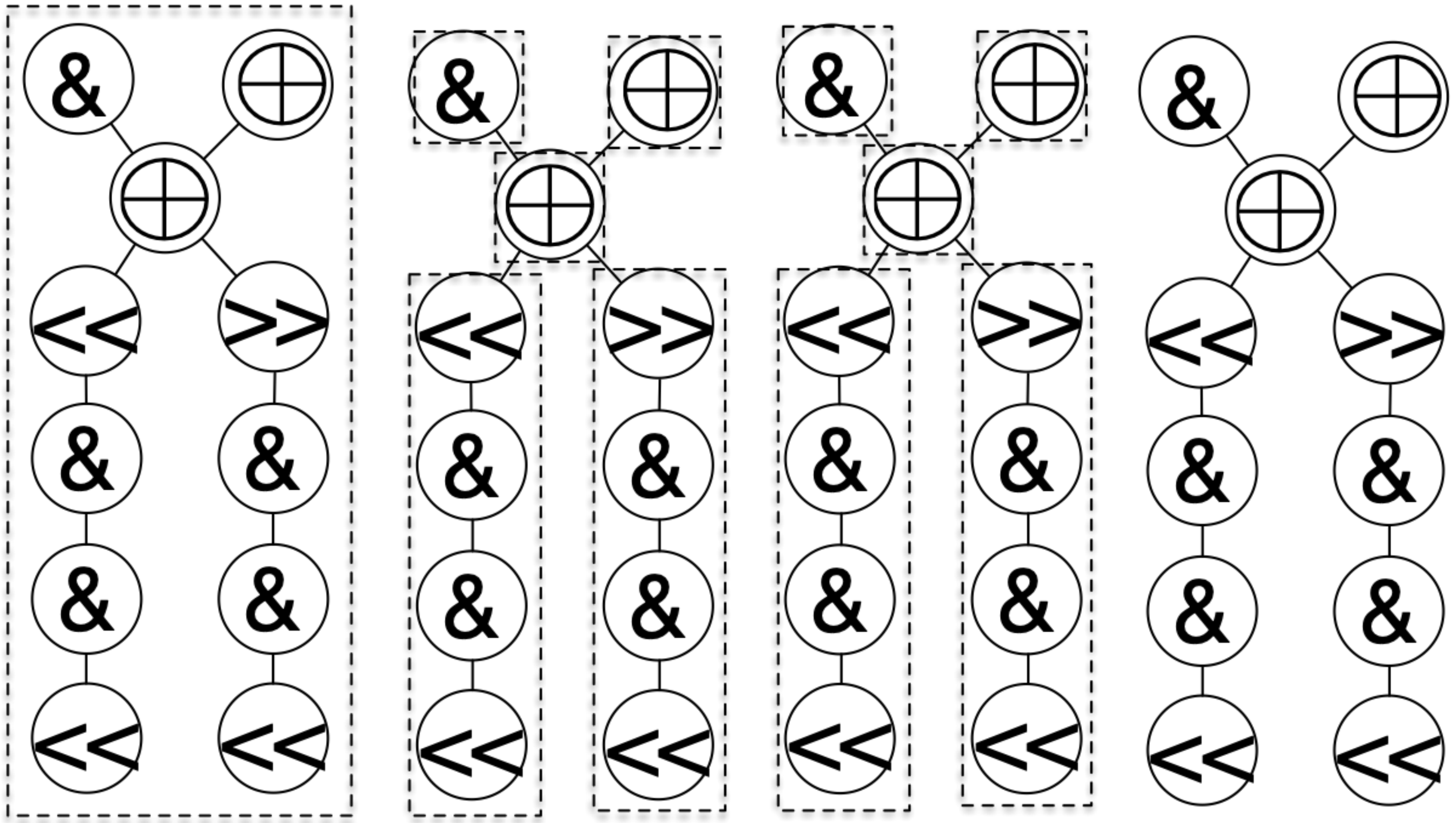




Acc Design Parameters:

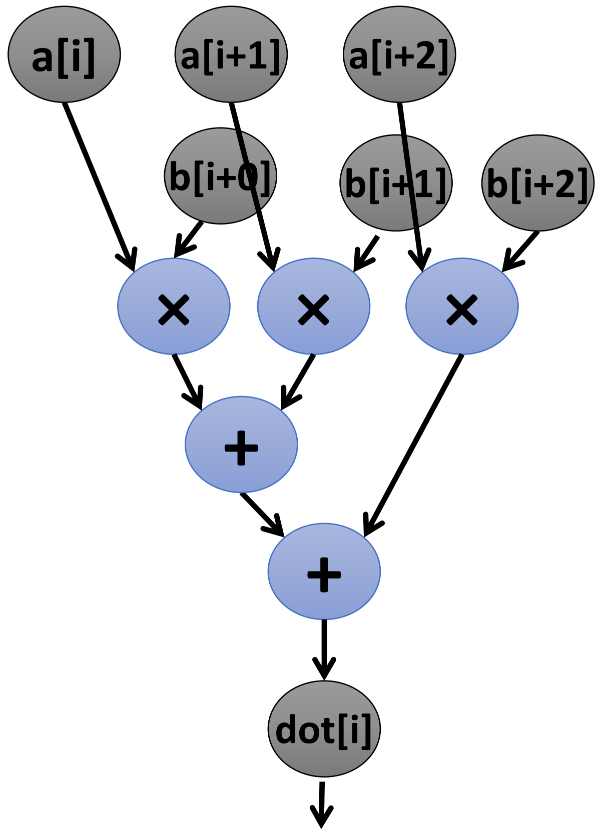
- ✓ Memory BW <= **4**
- ✓ **2** Adders

Scheduling 101



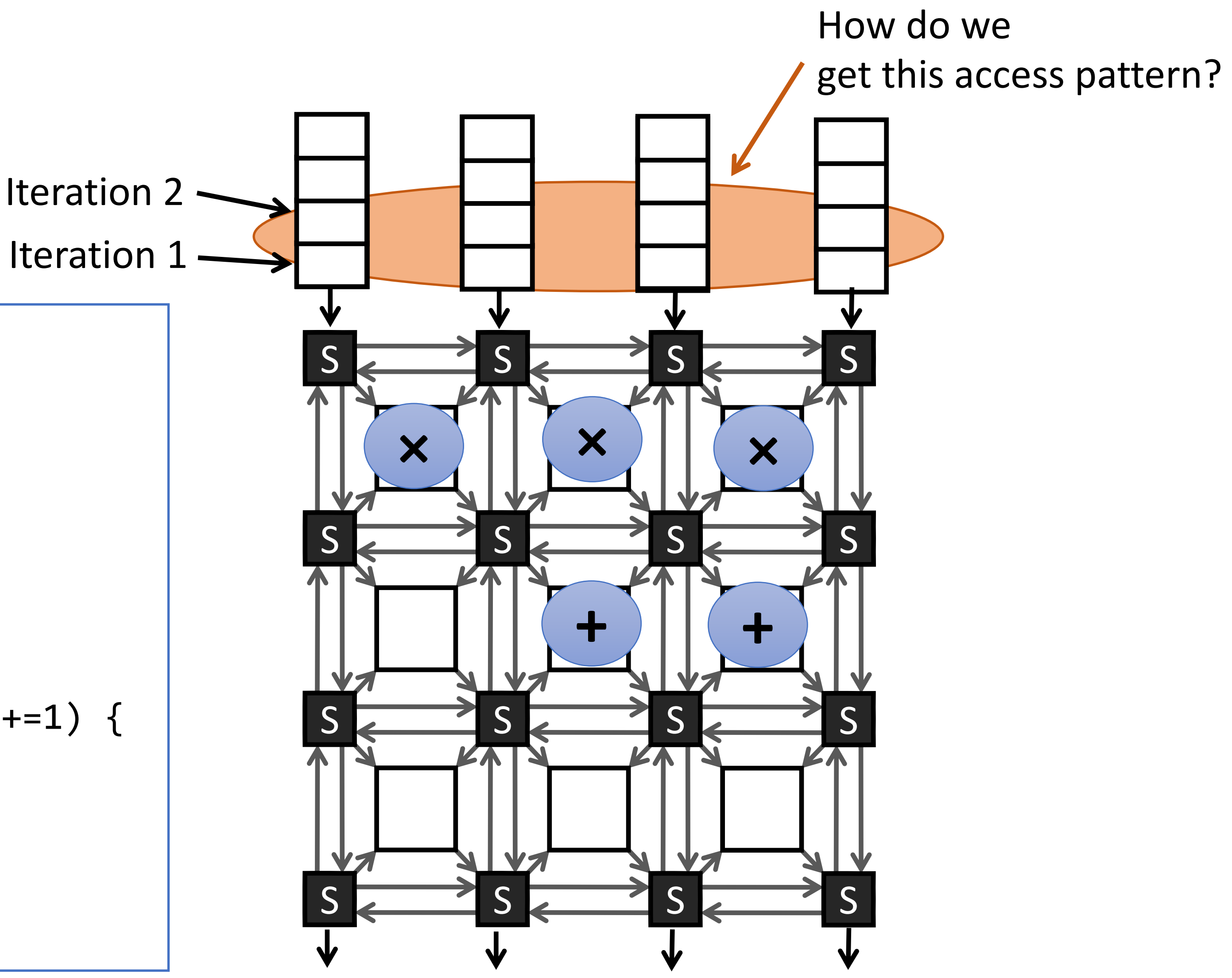
CGRA Vector Interface

```
struct vec {
  float x, y, z;
  float q;
}
vec A[], B[];
float *a = A, *b = B;
float dot[];
for(int i =0; i < LEN; i+=1) {
  dot[i]=A[i].x*B[i].x
        +A[i].y*B[i].y
        +A[i].z*B[i].z;
}
```

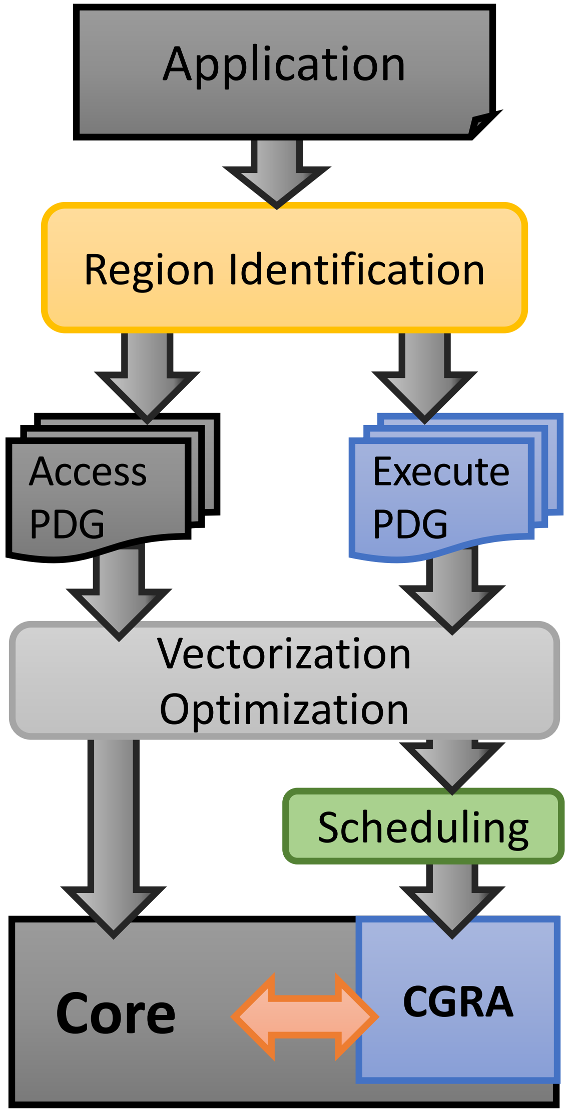


CGRA Vector Interface

```
struct vec {
  float x, y, z;
  float q;
}
vec A[], B[];
float *a = A, *b = B;
float dot[];
for(int i =0; i < LEN; i+=1) {
  dot[i]=A[i].x*B[i].x
        +A[i].y*B[i].y
        +A[i].z*B[i].z;
}
```

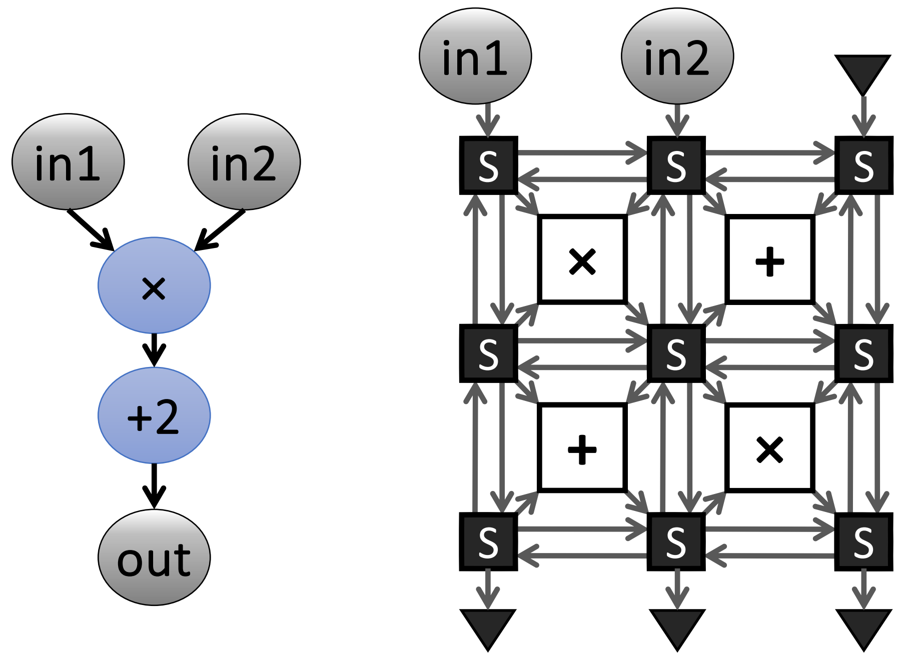


Ports shown only for a[]



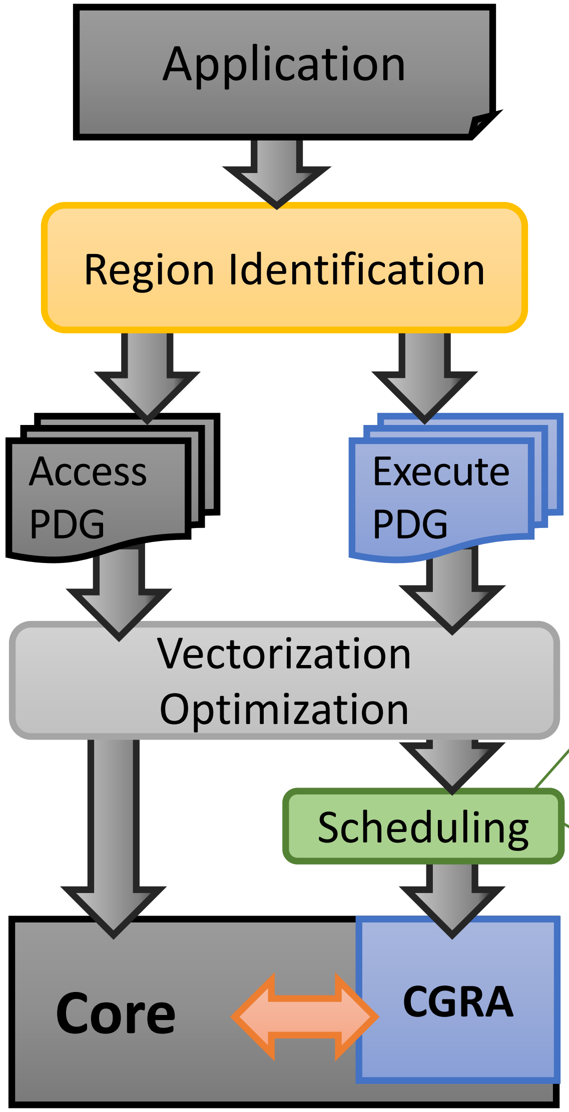
Scheduling

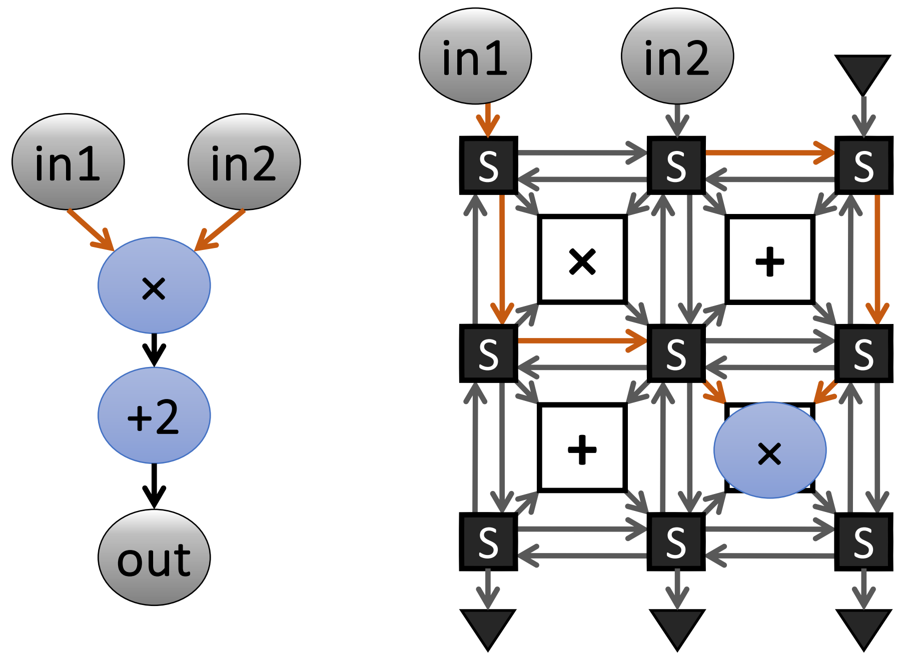
- Map Execute Subregion
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

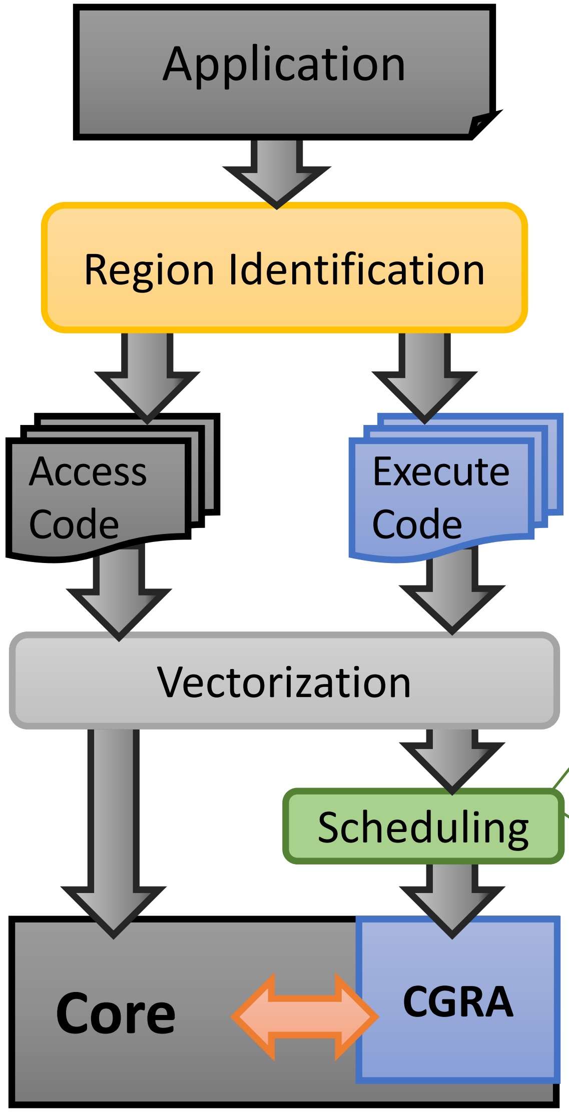


Scheduling

- Map Execute Subregion to CGRA
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

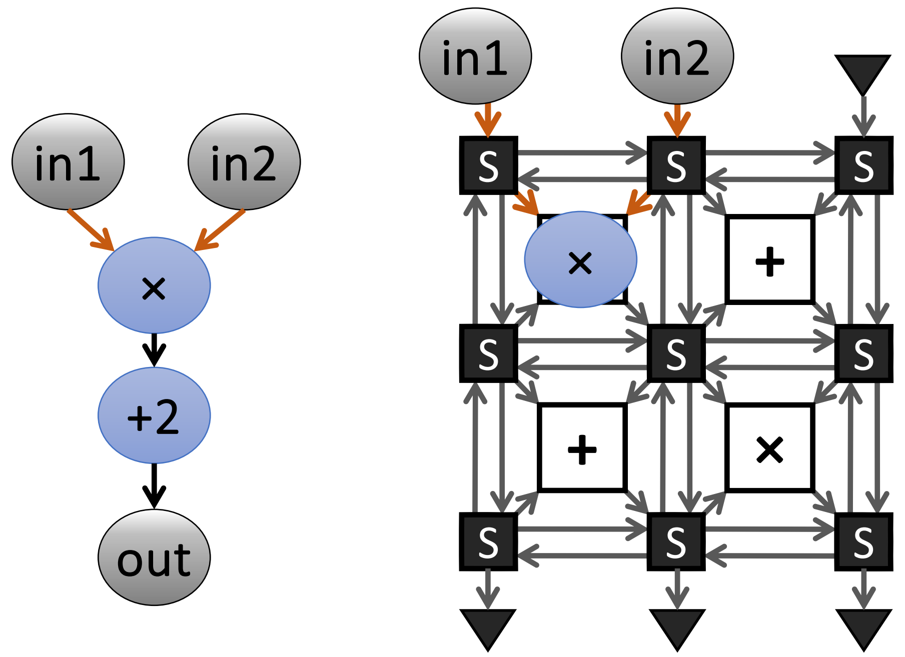


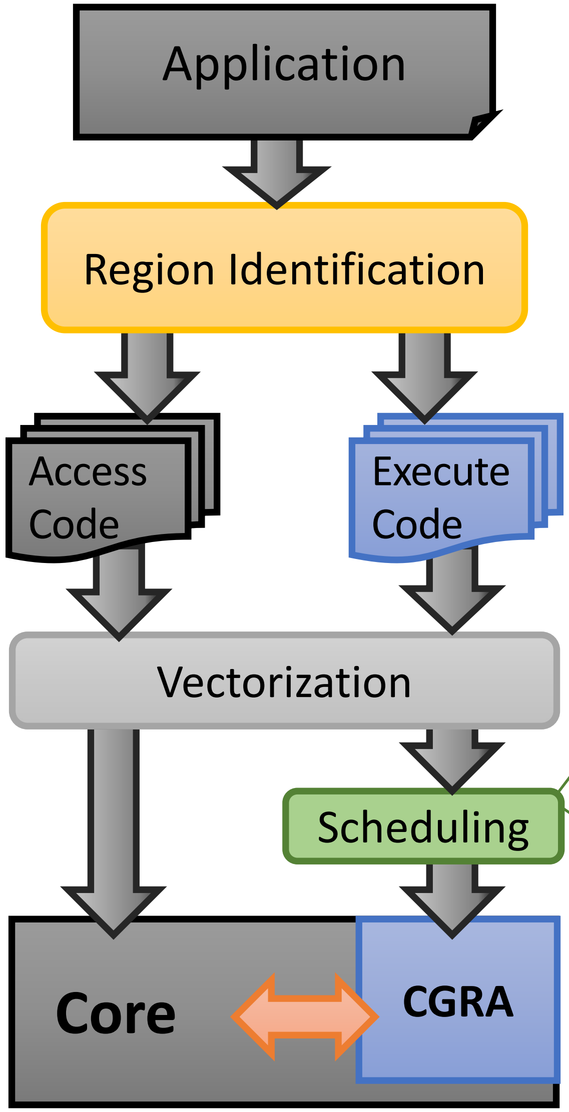




Scheduling

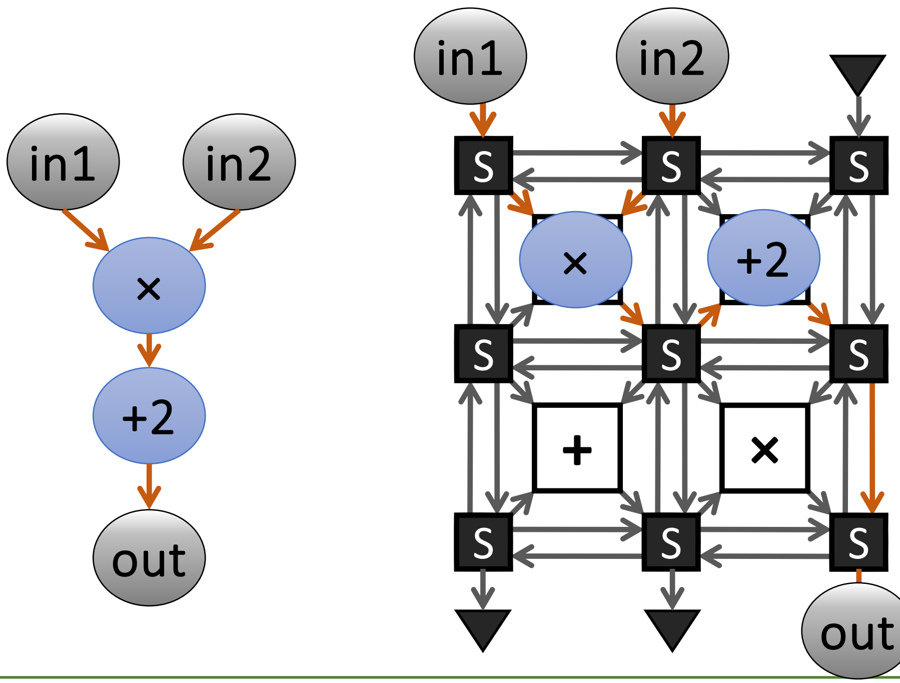
- Map Execute Subregion
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

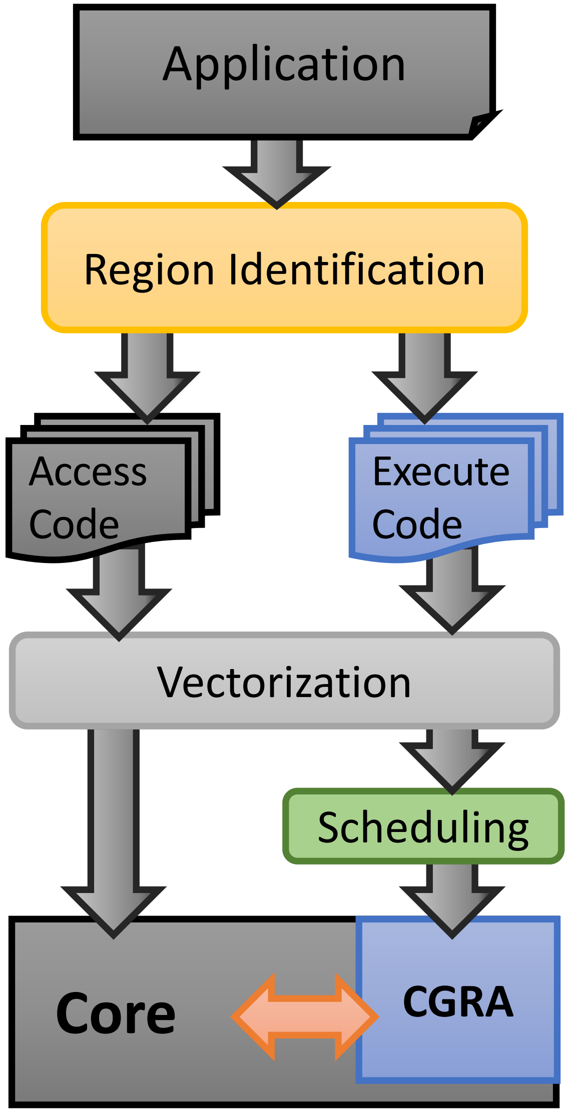




Scheduling

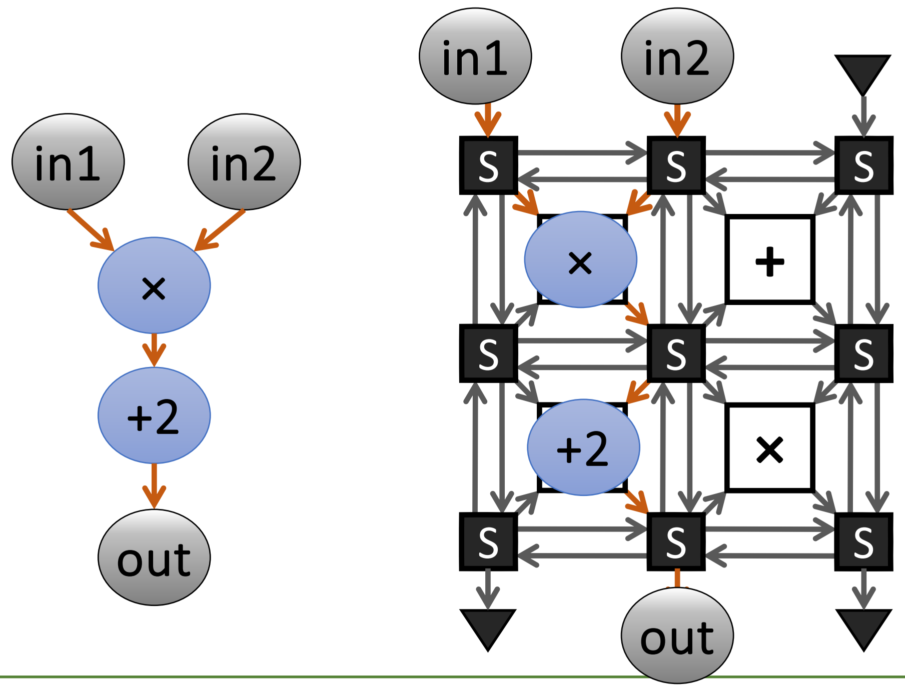
- Map Execute Subregion
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes





Scheduling

- Map Execute Subregion to CGRA
  - Sort nodes in data flow order
  - Greedily place each node to minimize the total routes

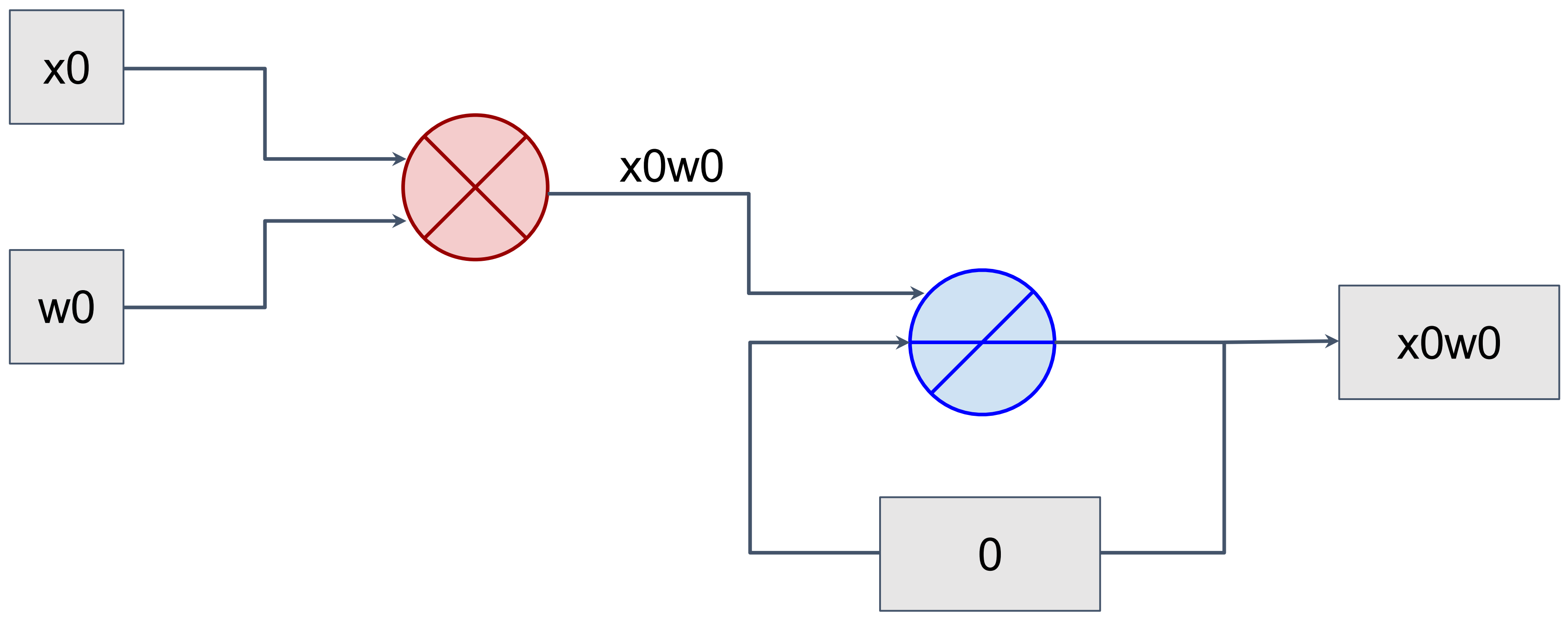


Outline

1. PE Microarchitecture
    a. Parallelization
    b. Pipelining
    c. Interleaving
    d. Arithmetic

2. On-Chip Memory
    b. Basics
    c. Banking

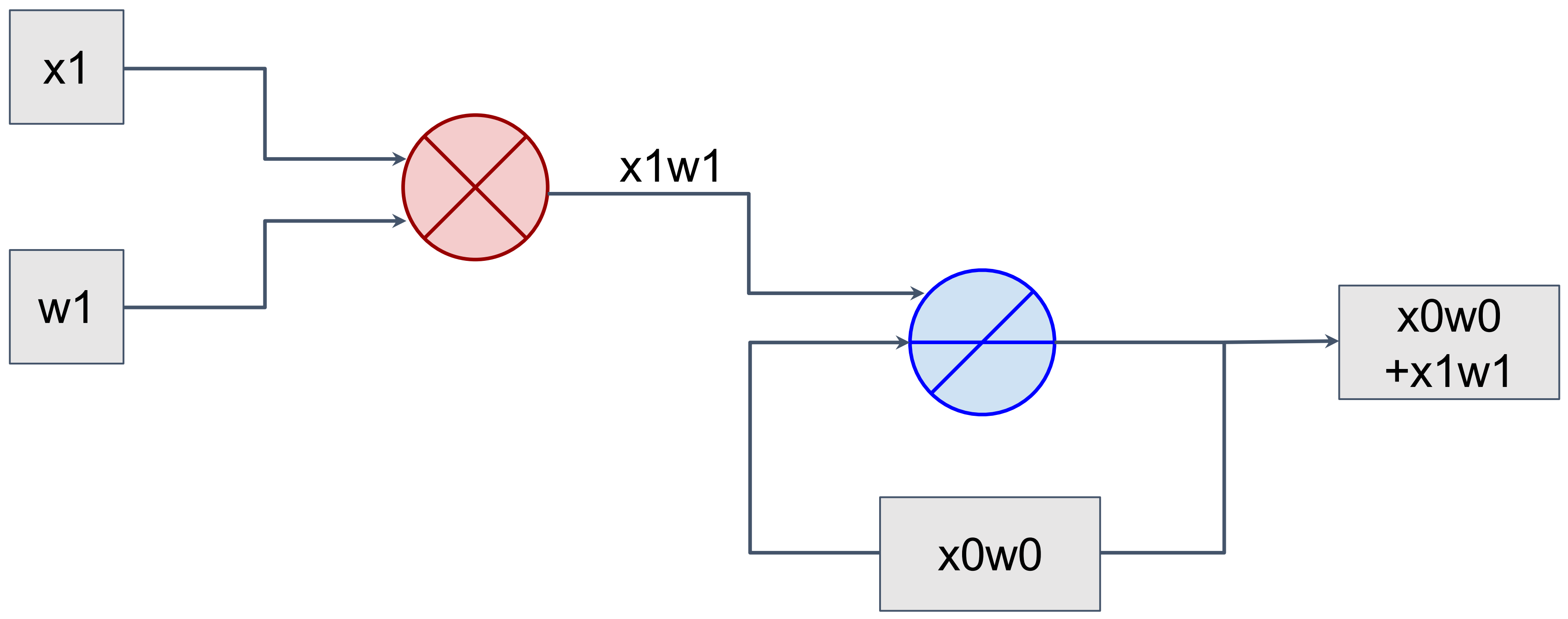
Processing Element (PE)



$$\begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \dots \\ w_n \end{pmatrix}$$

$$y = x \,.\, w = x_0w_0 + x_1w_1 + \dots + x_nw_n$$

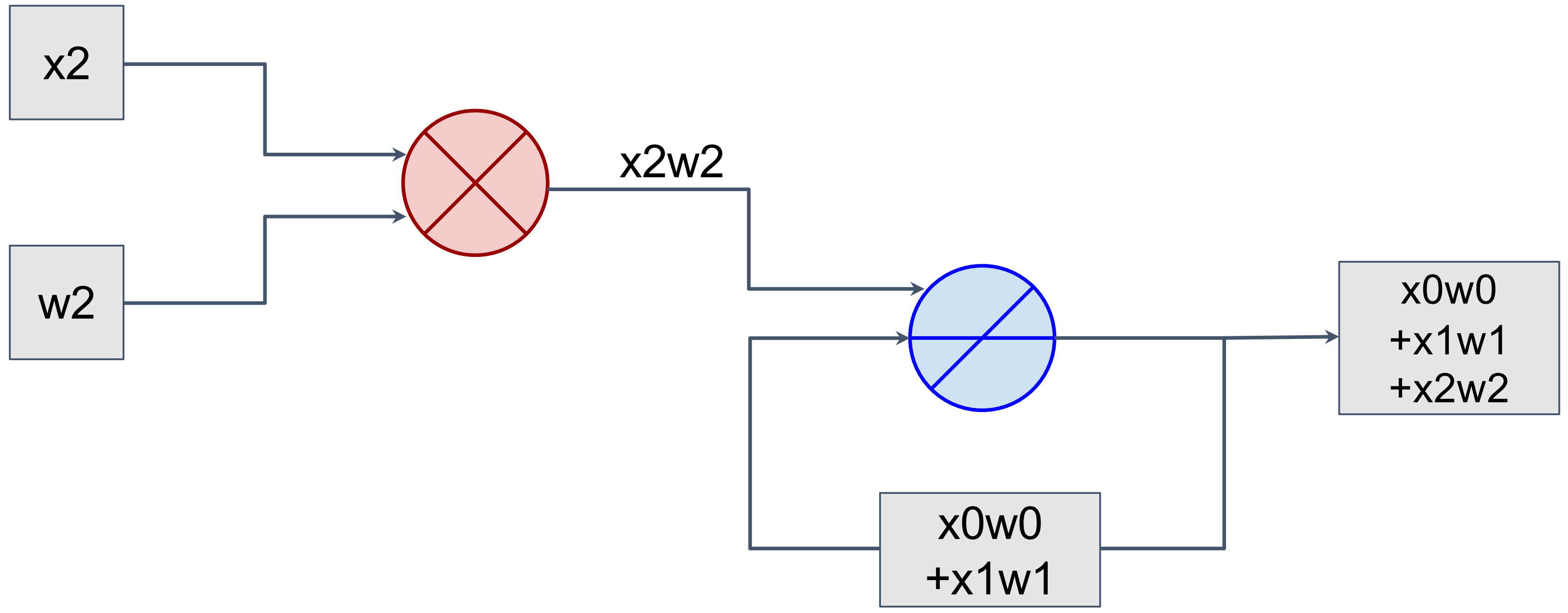
Processing Element (PE)



$$\begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \dots \\ w_n \end{pmatrix}$$

$y = x \, . \, w = x_0w_0 + x_1w_1 + \ldots + x_nw_n$

Processing Element (PE)

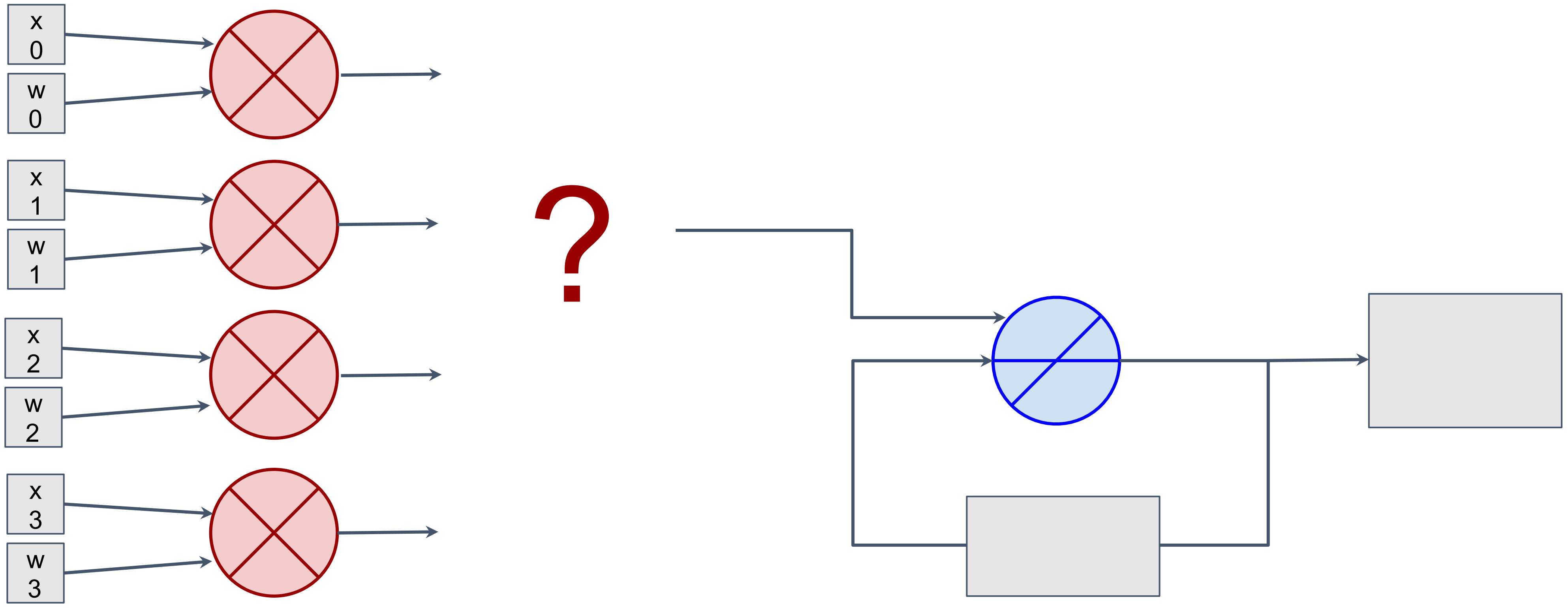


$$\begin{pmatrix} x0 \\ x1 \\ \dots \\ x_n \end{pmatrix} \begin{pmatrix} w0 \\ w1 \\ \dots \\ w_n \end{pmatrix}$$

y = x . w = x0w0 + x1w1 + … + xnwn

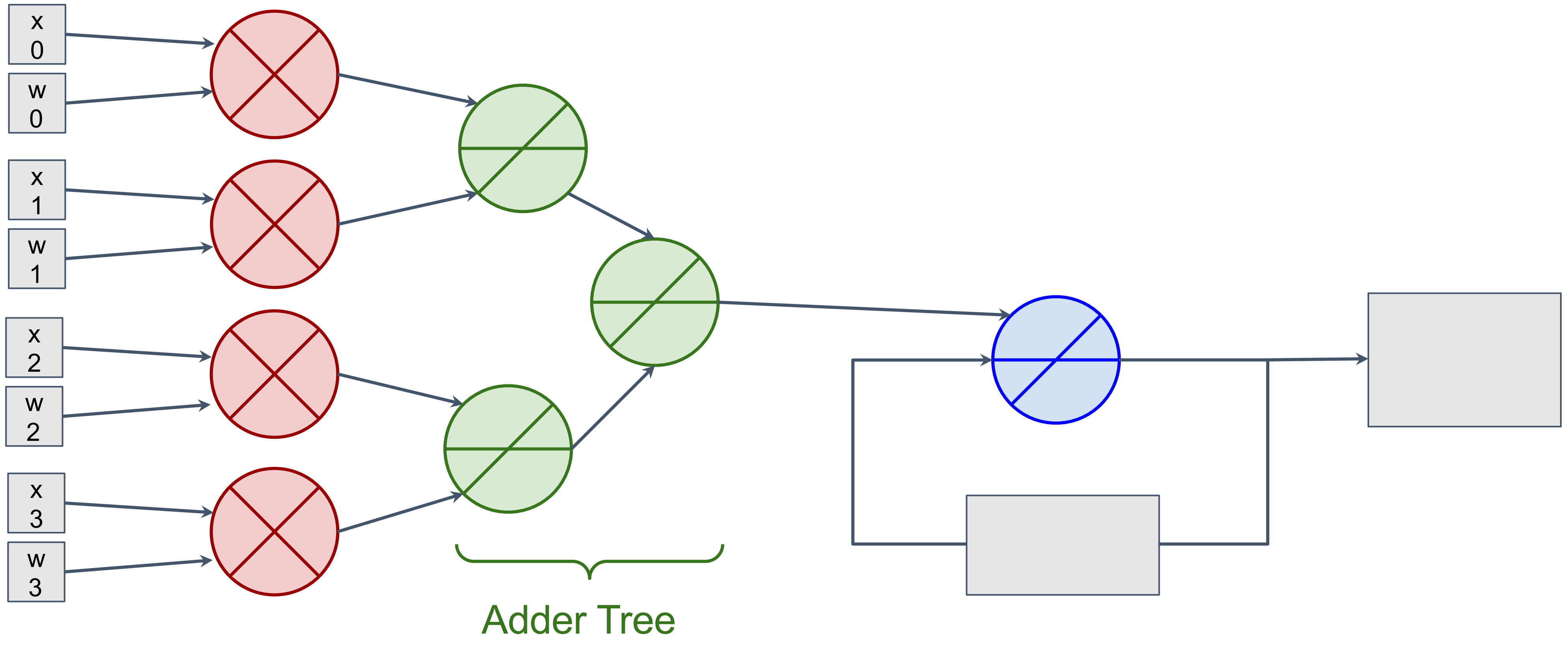
SFU

Parallelization (or Vectorization)



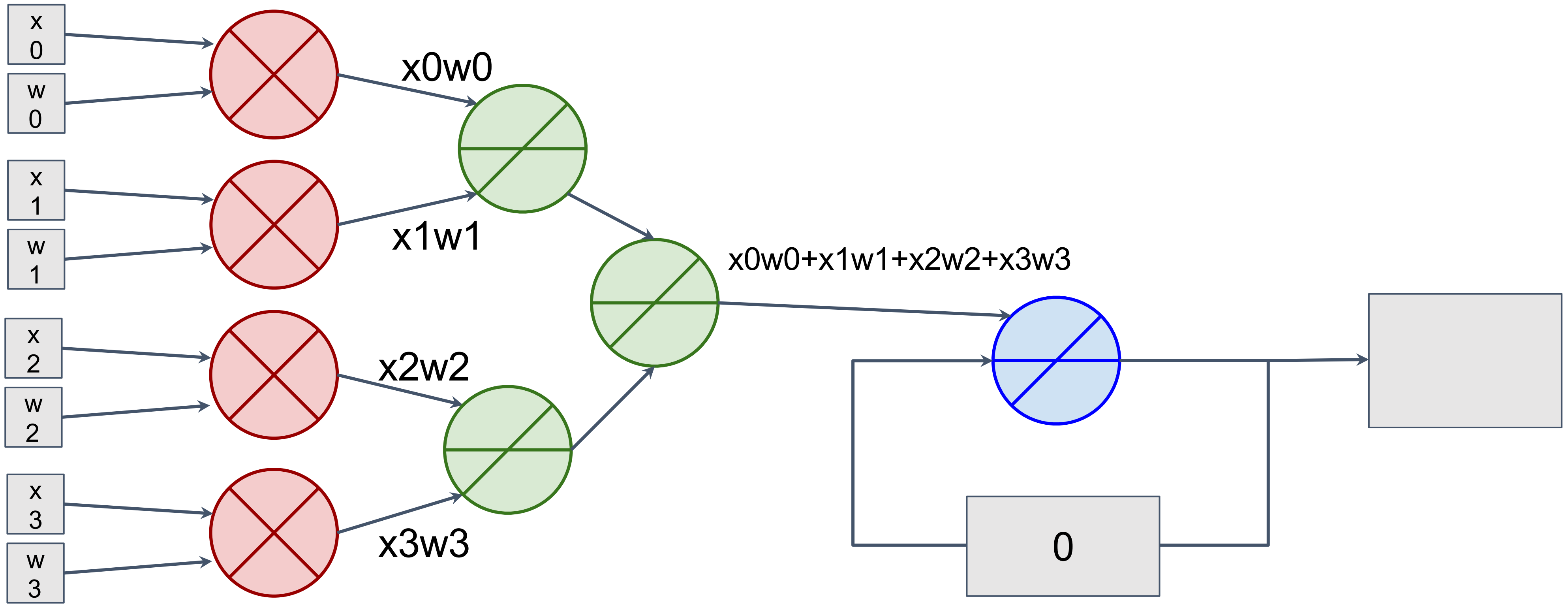
y = x . w = x0w0 + x1w1 + … + xnwn

Parallelization (or Vectorization)



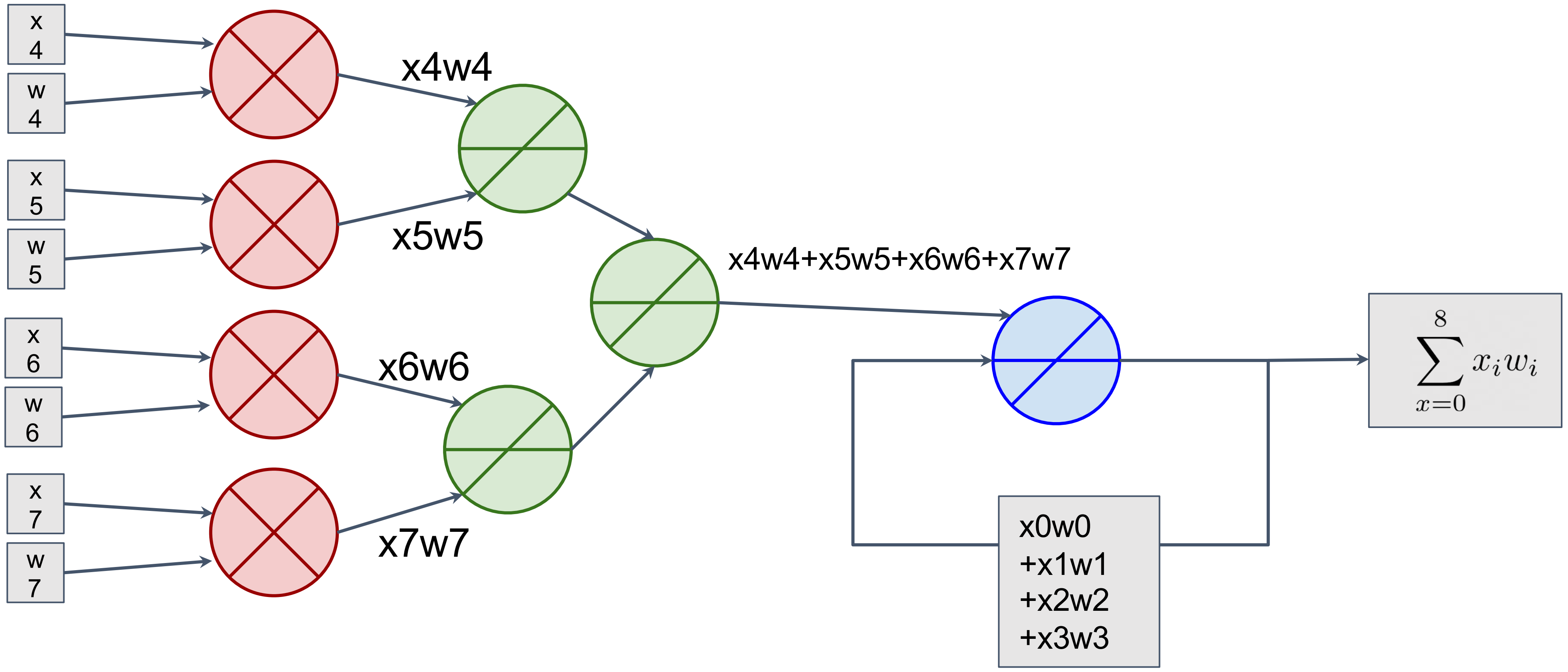
y = x . w = x0w0 + x1w1 + … + xnwn

Parallelization (or Vectorization)



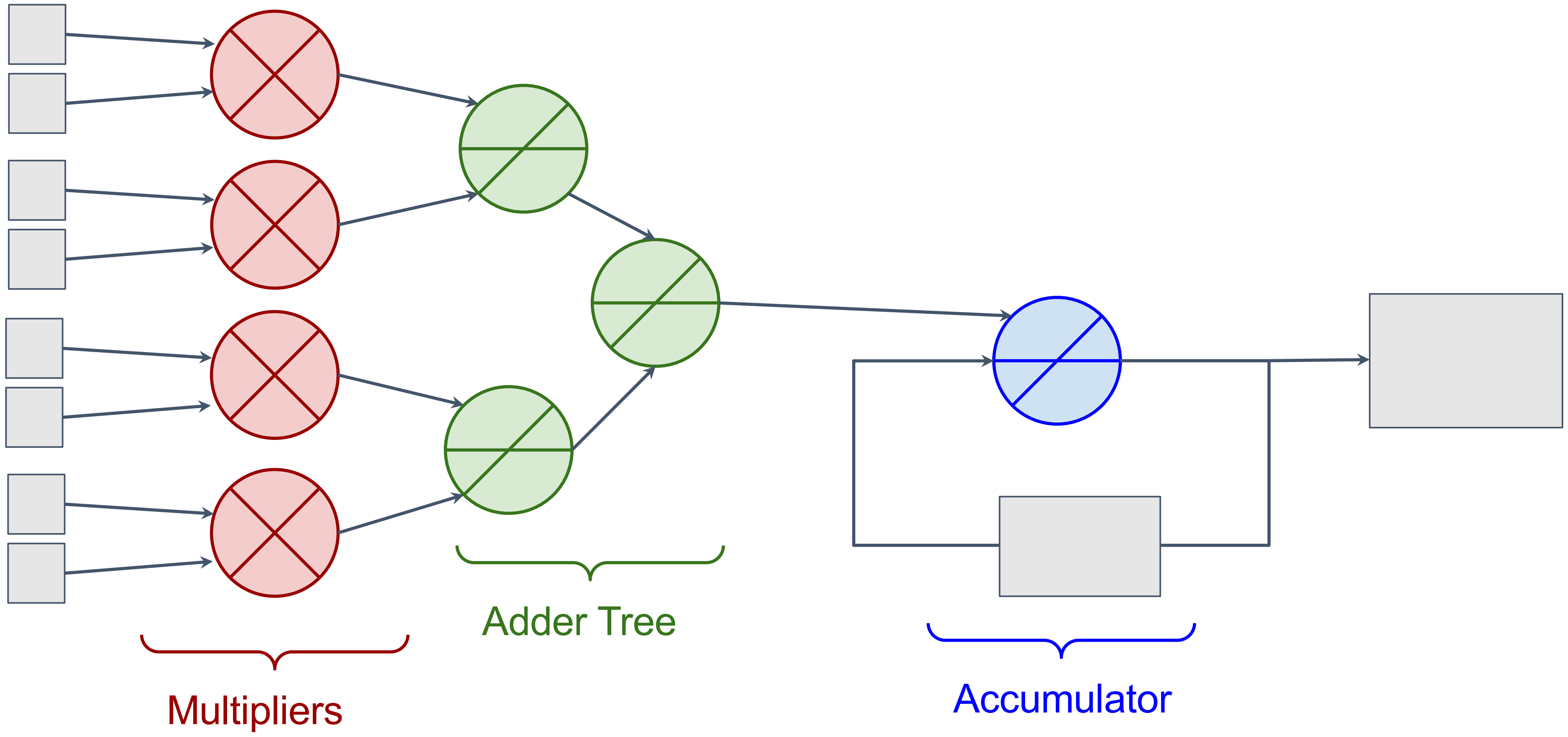
y = x . w = x0w0 + x1w1 + … + xnwn

Parallelization (or Vectorization)



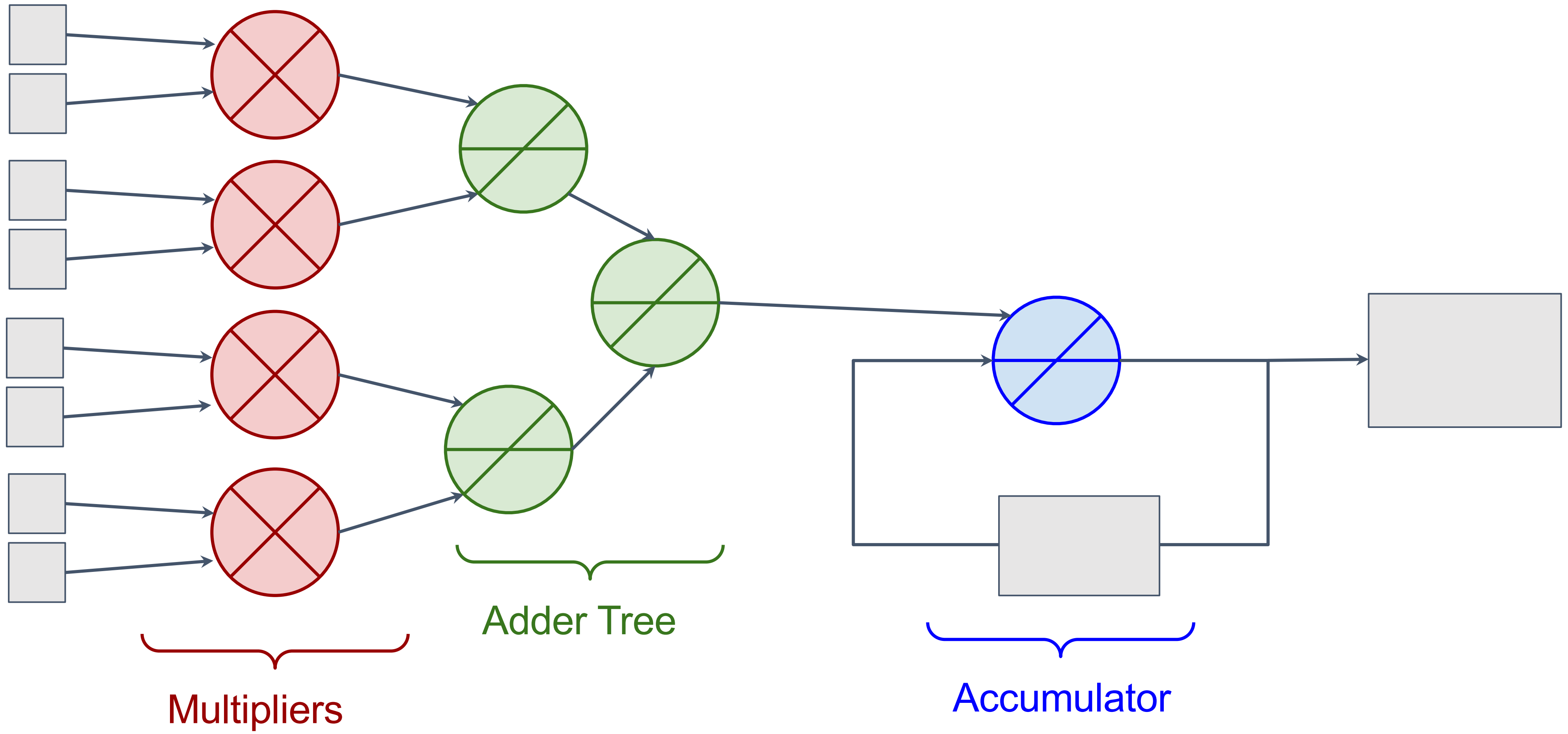
y = x . w = x0w0 + x1w1 + … + xnwn

Parallelization (or Vectorization)



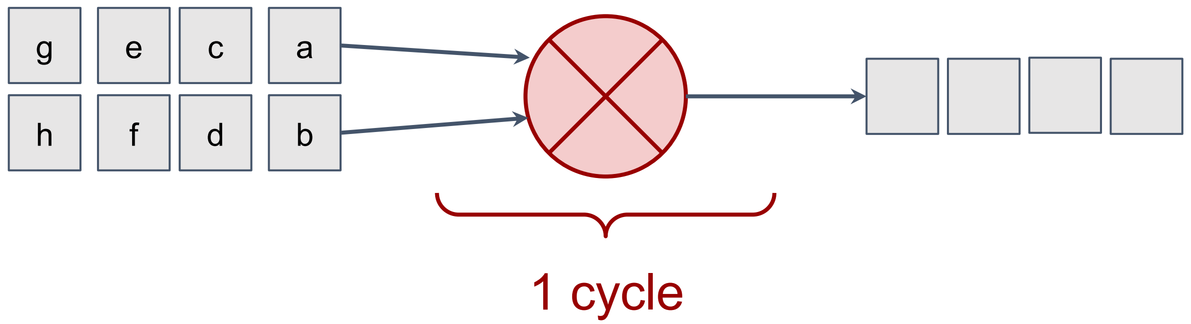
Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop



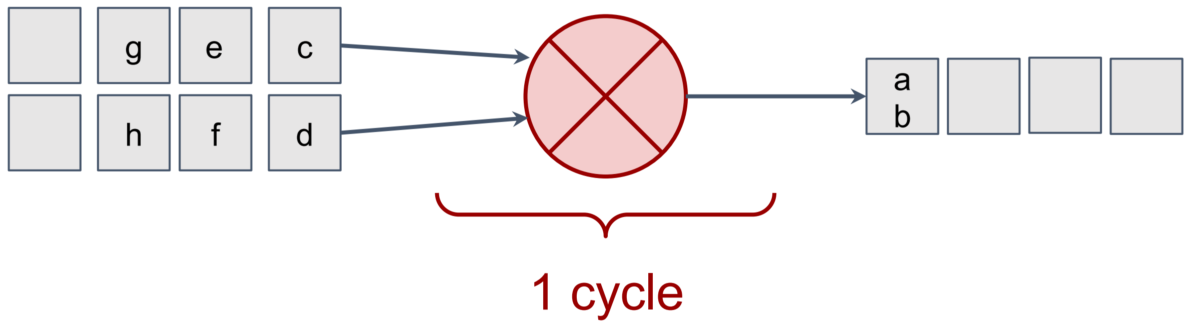
Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop



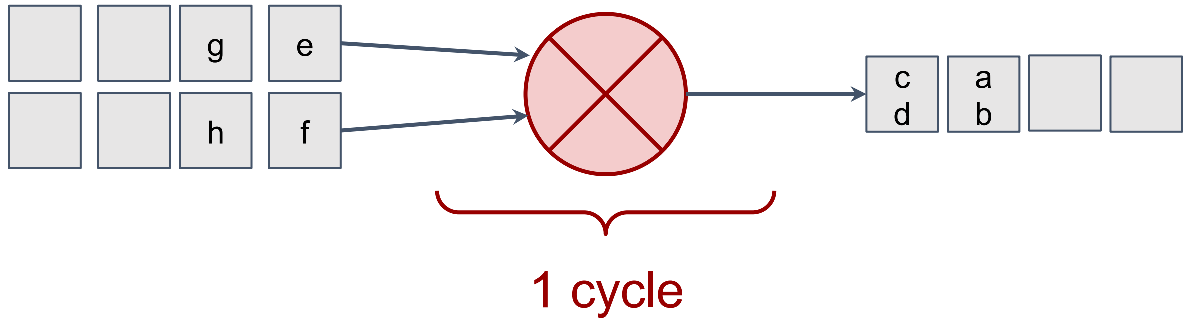
Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop



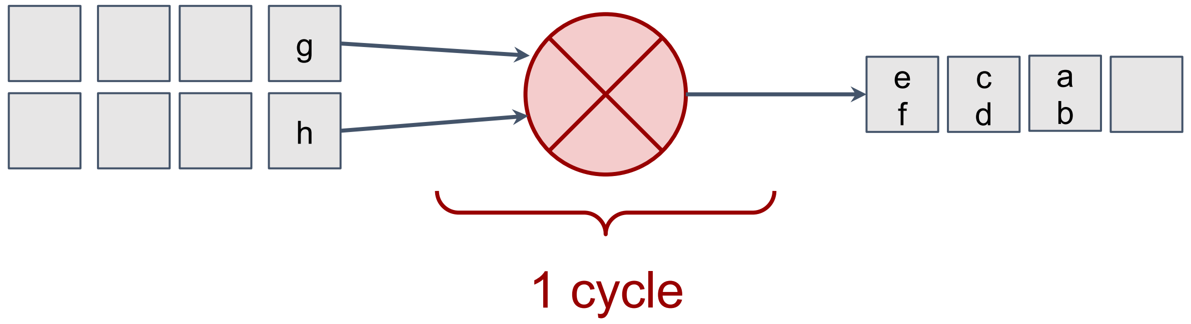
Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop



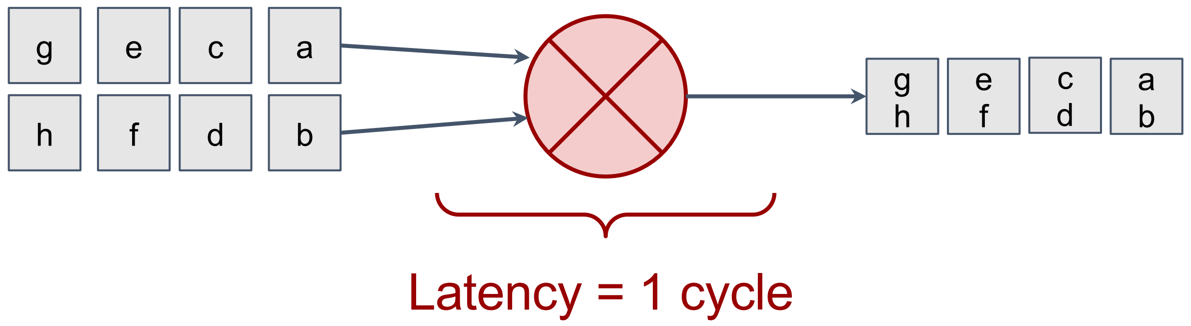
Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop



Pipelining

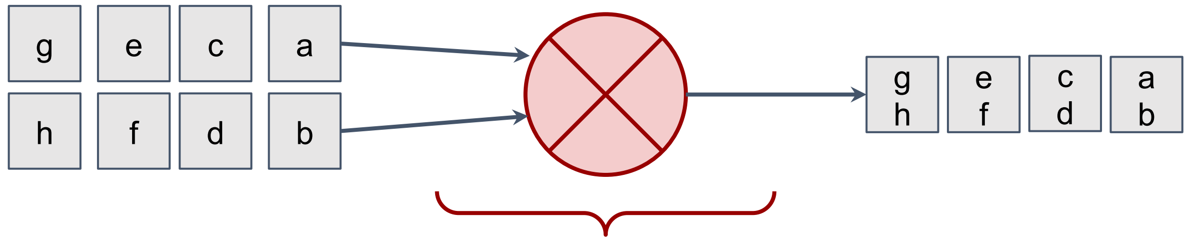
Initiation Interval: How often I can start the computation of a new element of a loop



What is my throughput? **1 op/cycle**

Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop

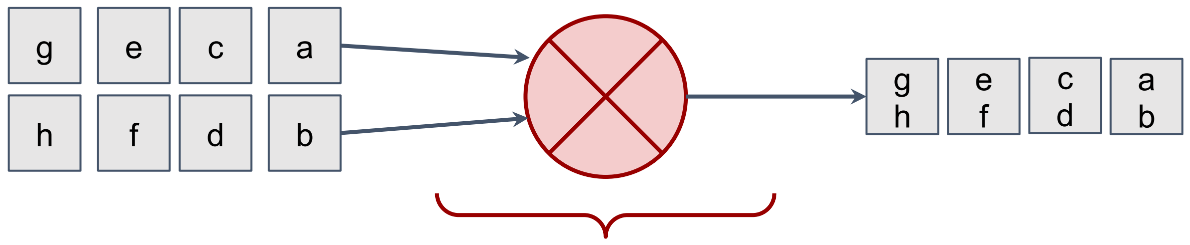


Latency = 5 cycles

Now, what is my throughput? **1 op/cycle**
if fully pipelined

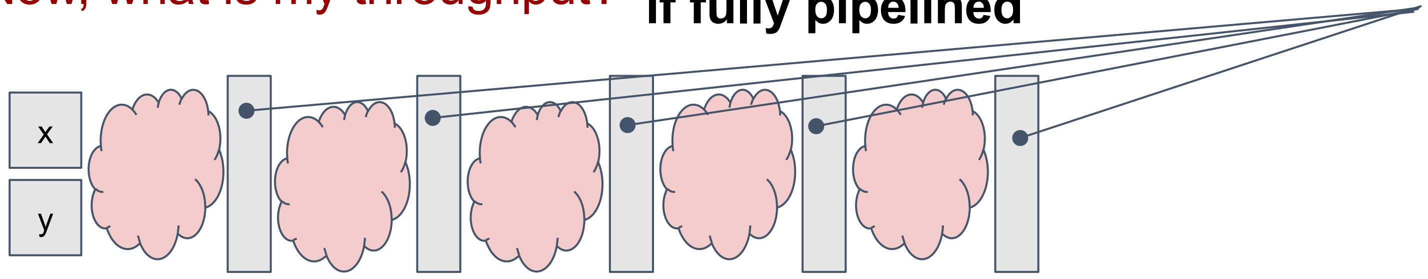
Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop



Latency = 5 cycles

Now, what is my throughput? **1 op/cycle if fully pipelined**

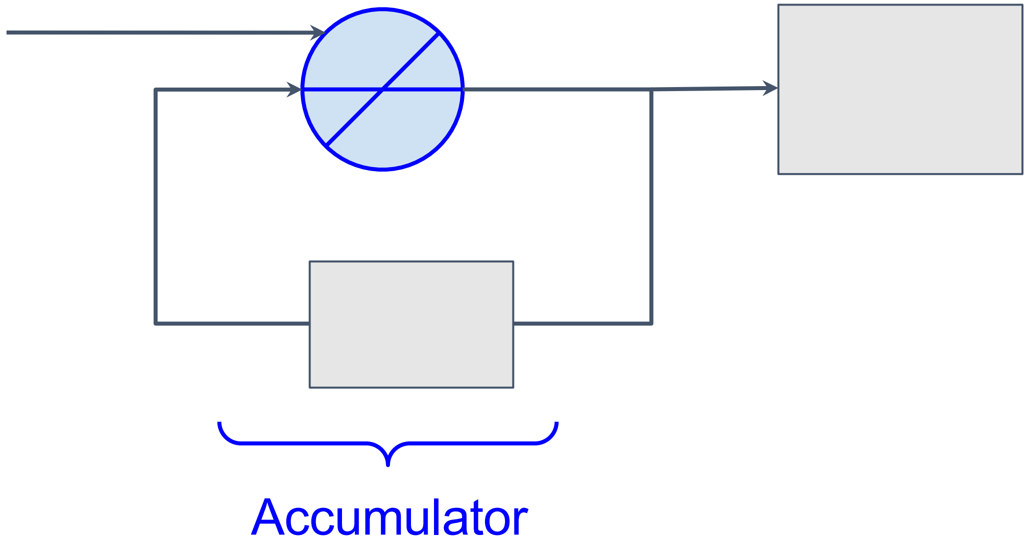


Space to store intermediate results

Allows you to start a new op per cycle II=1

Pipelining

Initiation Interval: How often I can start the computation of a new element of a loop

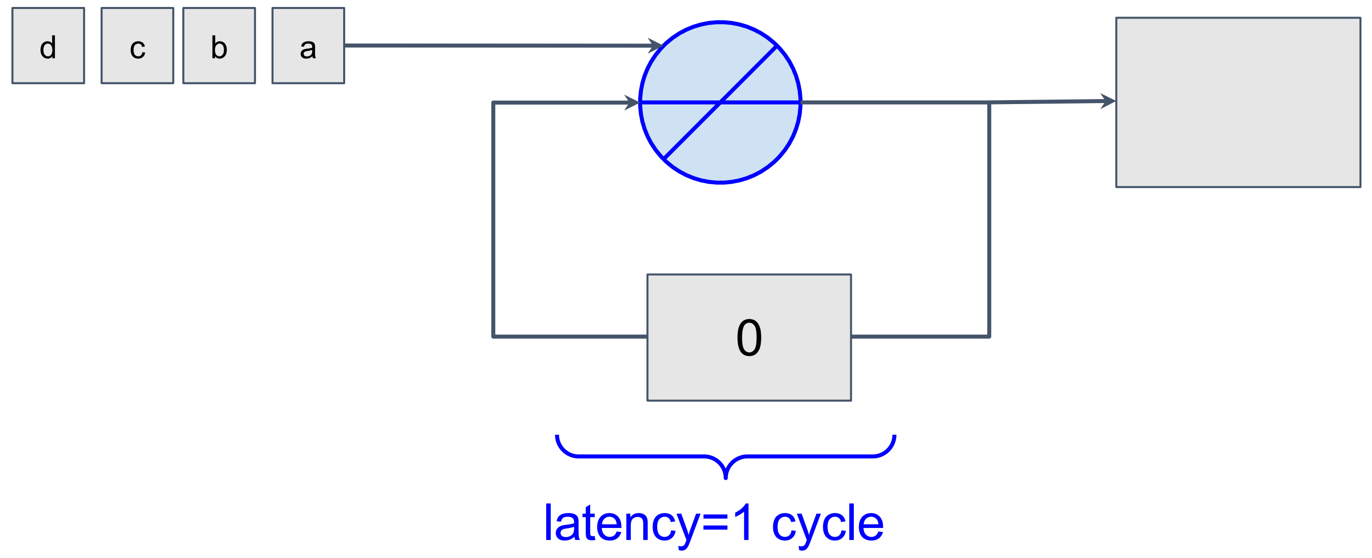


What about accumulators?

Different because they have a data dependency

Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop

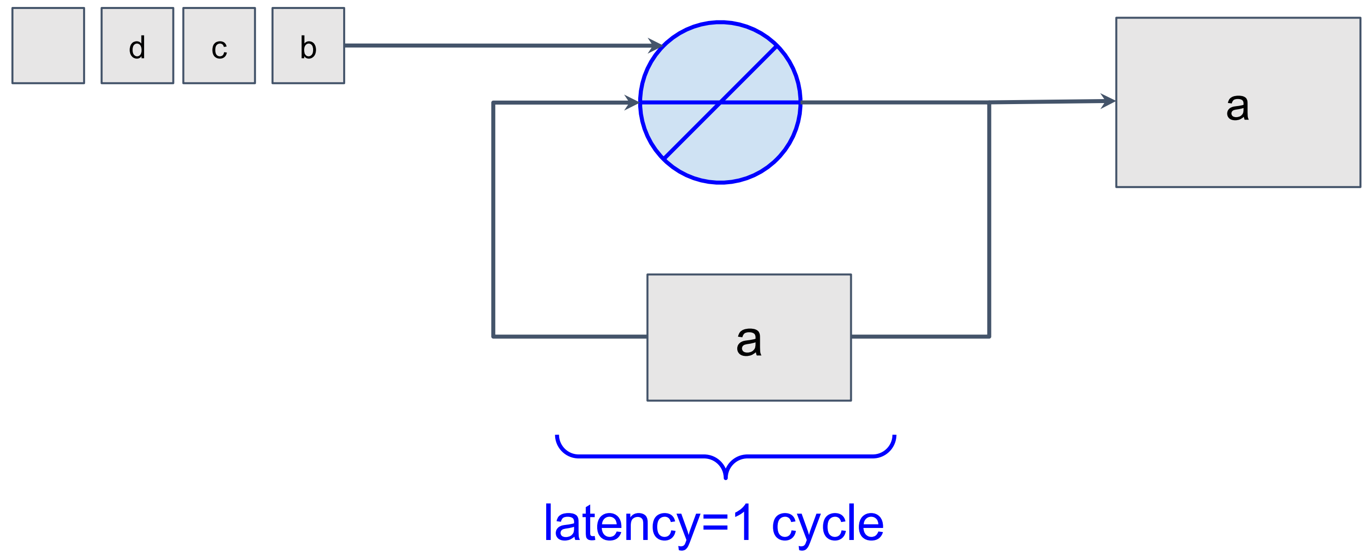


What about accumulators?

Different because they have a data dependency

Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop

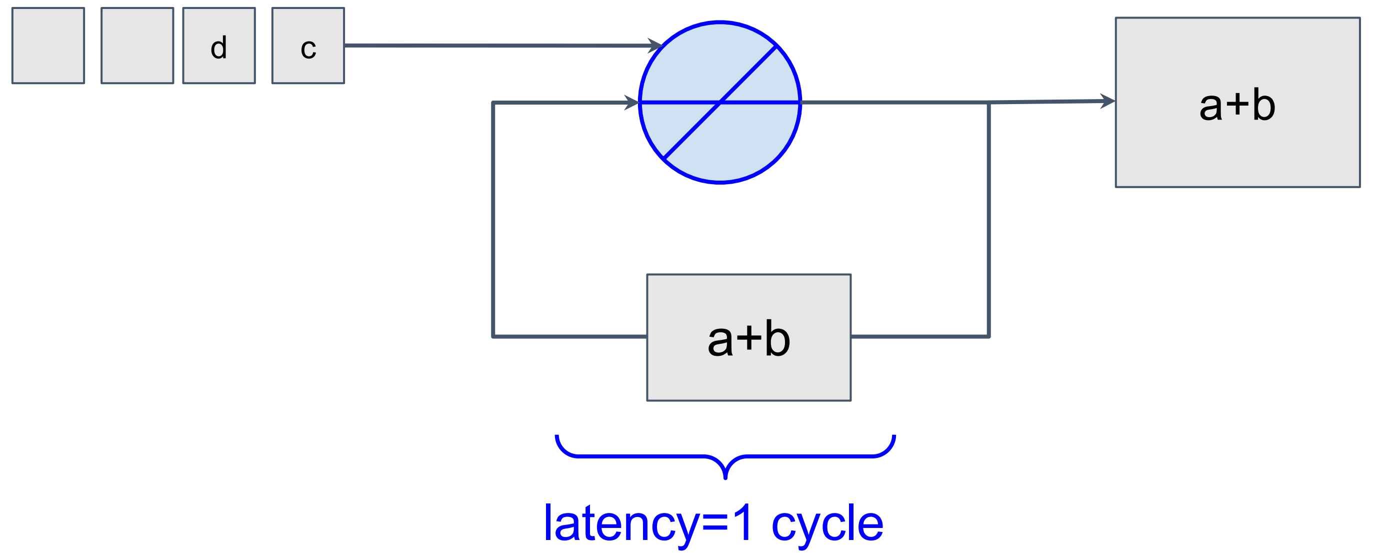


What about accumulators?

Different because they have a data dependency

Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop

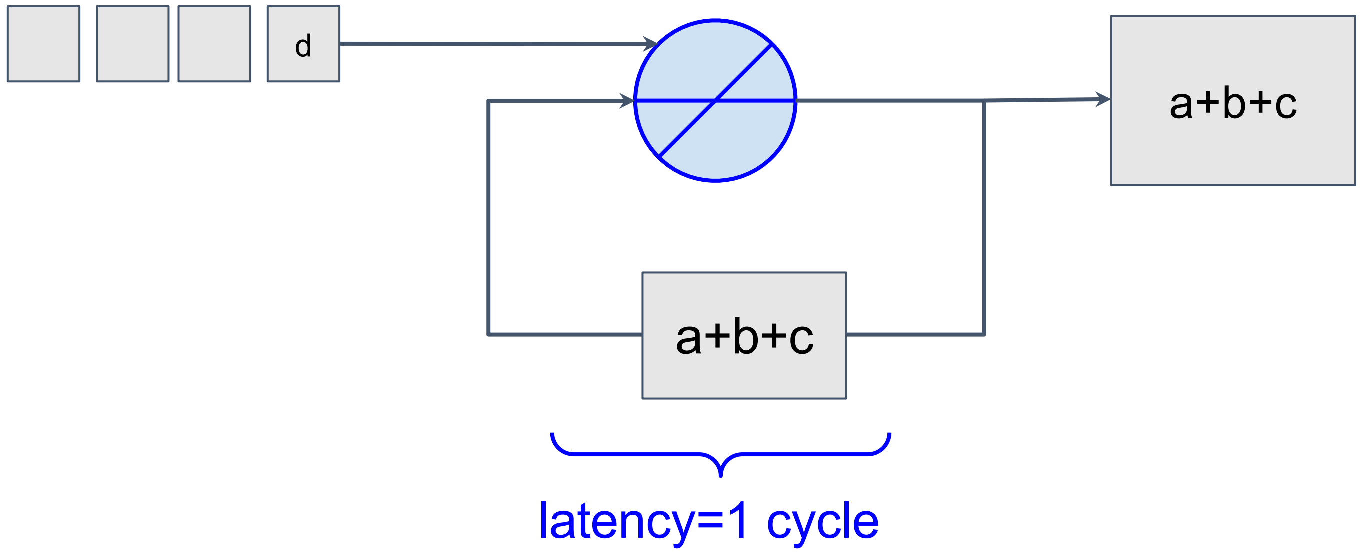


What about accumulators?

Different because they have a data dependency

Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop

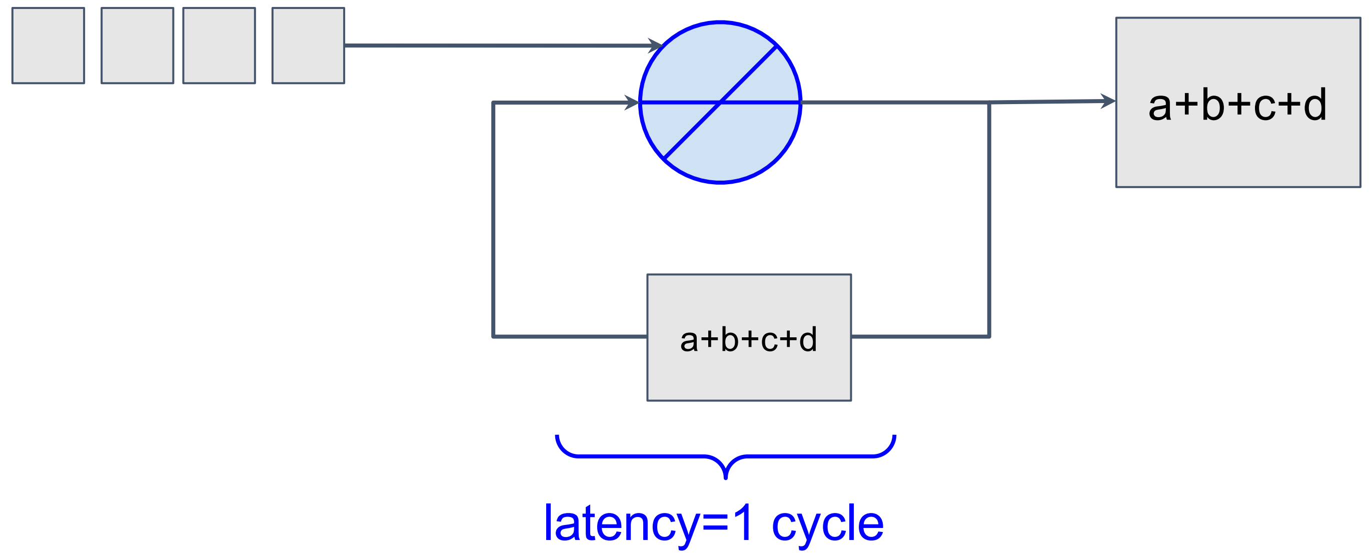


What about accumulators?

Different because they have a data dependency

Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



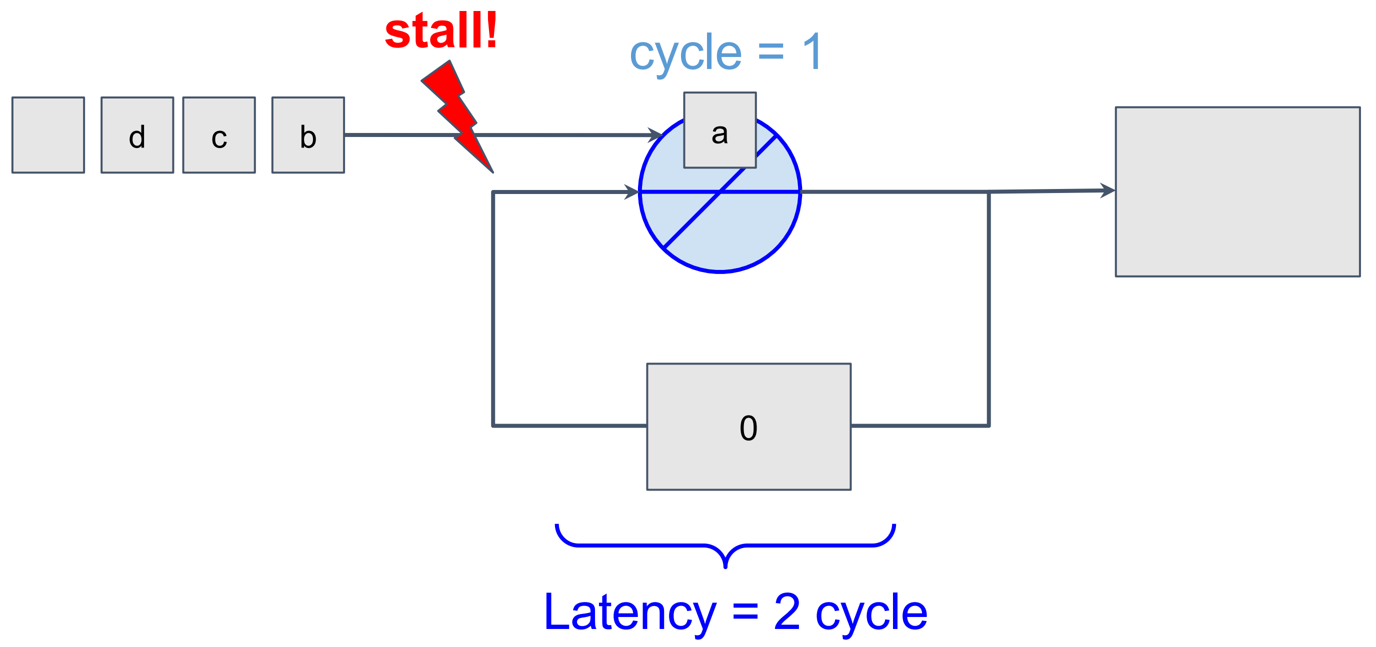
What about accumulators?

Different because they have a data dependency

What is my throughput? **1 op/cycle**

Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



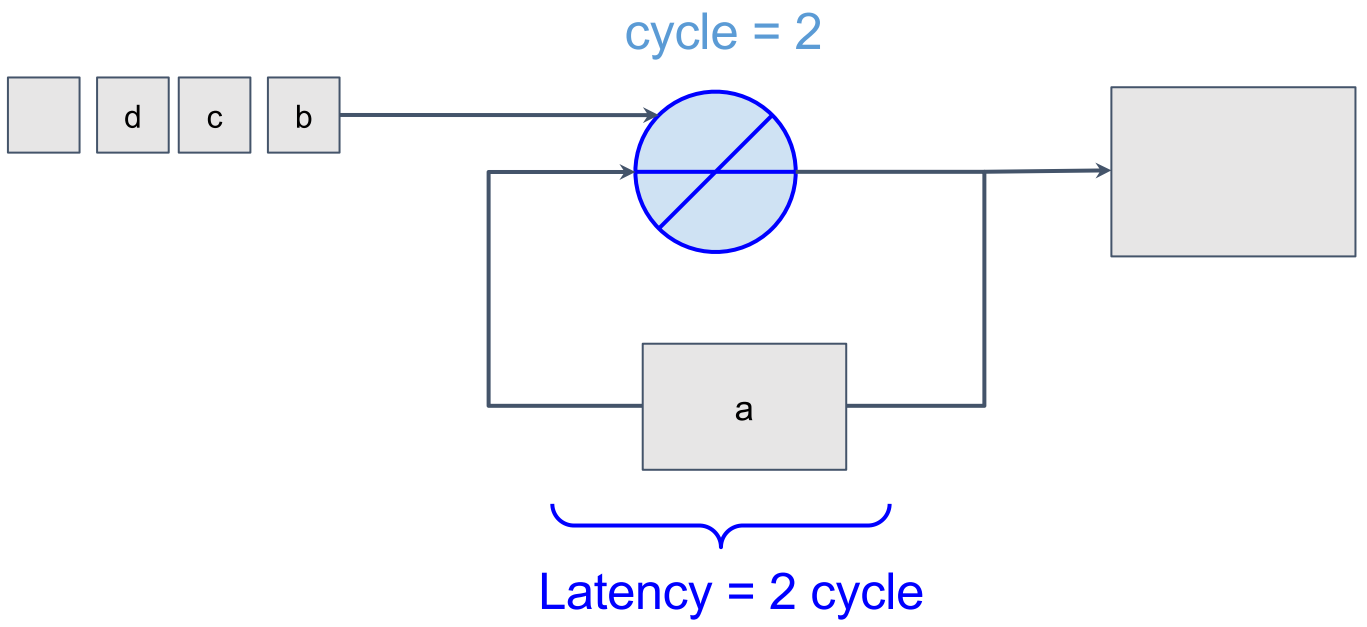
What about accumulators?

Different because they have a data dependency

Now, what is my throughput?

Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



What about accumulators?

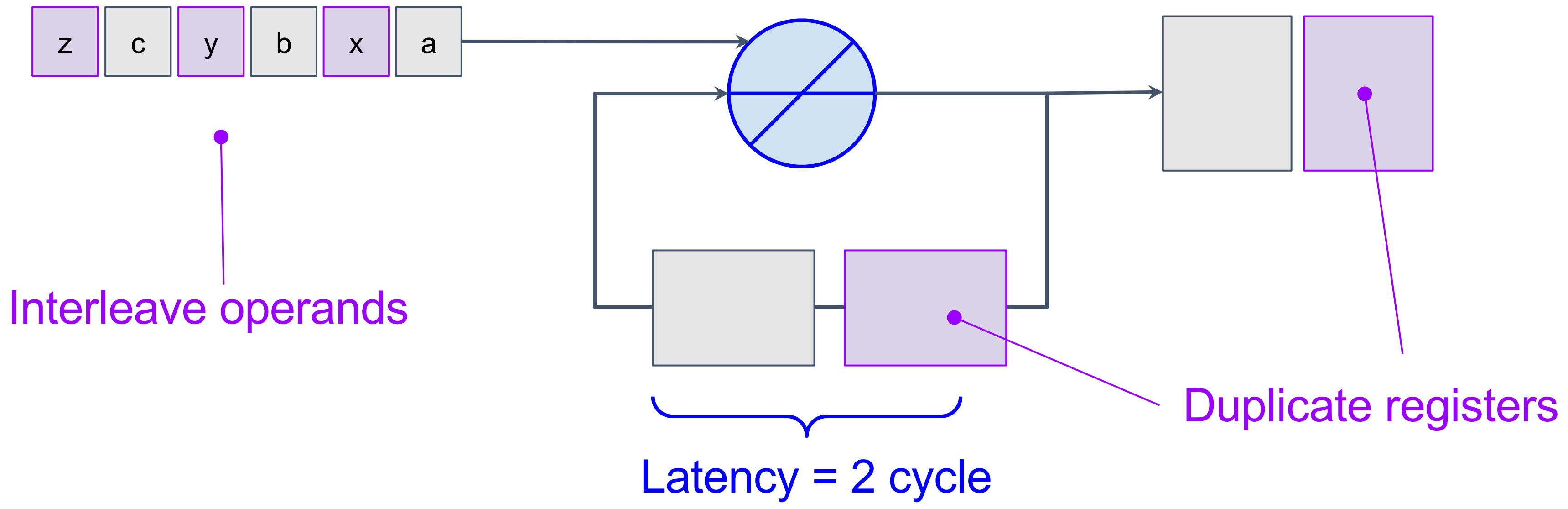
Different because they have a data dependency

Now, what is my throughput? **0.5 ops/cycle – data dependency stalls my pipeline!**

Note: II = 2

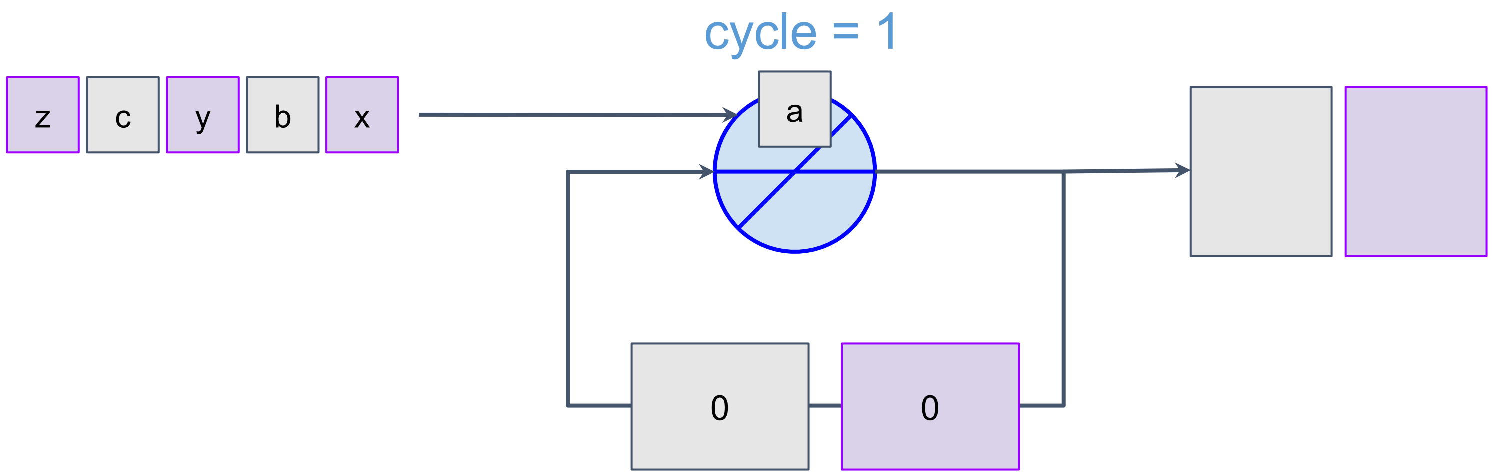
Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



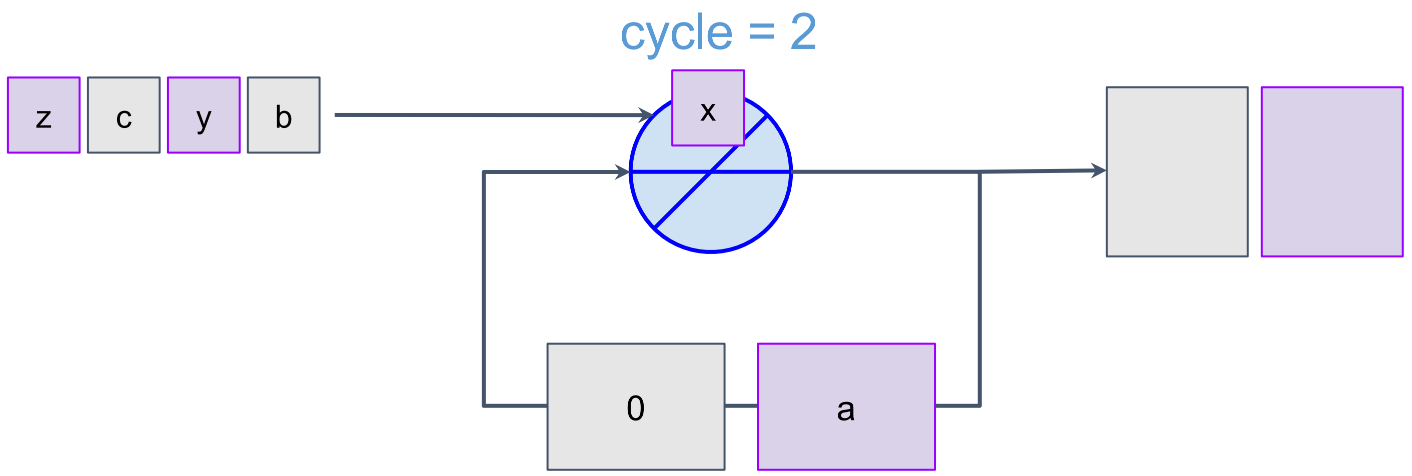
Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



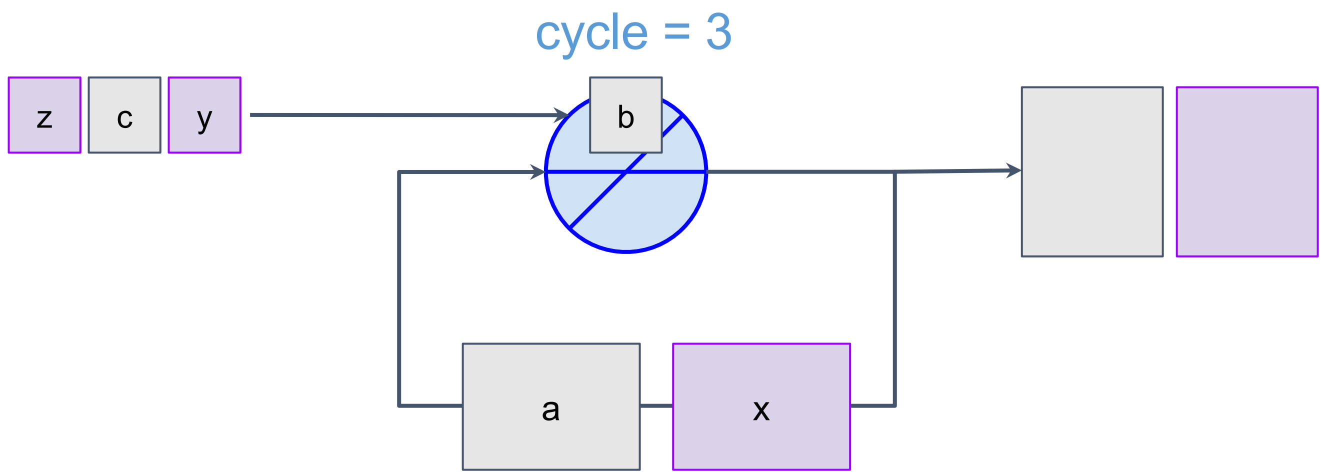
Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



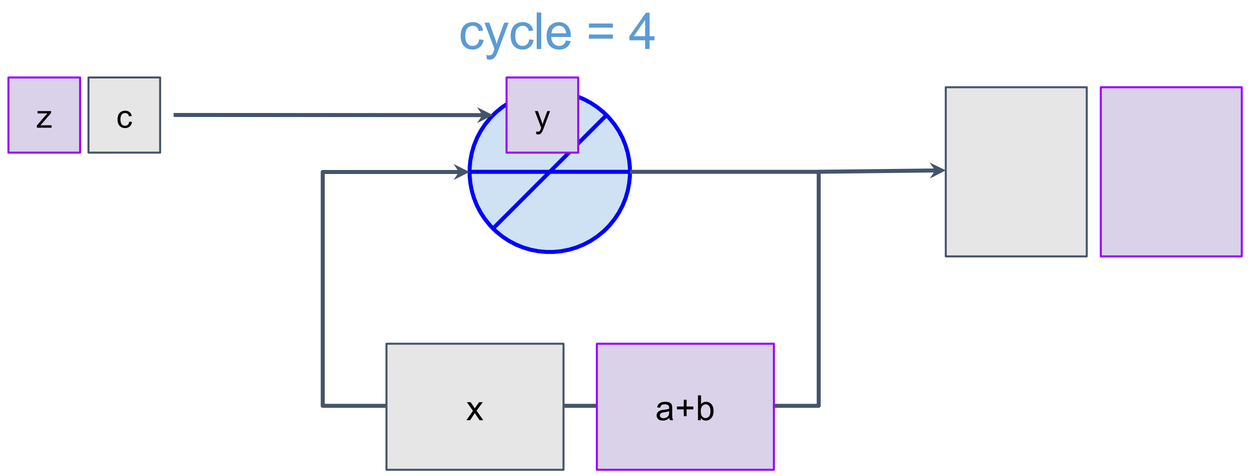
Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



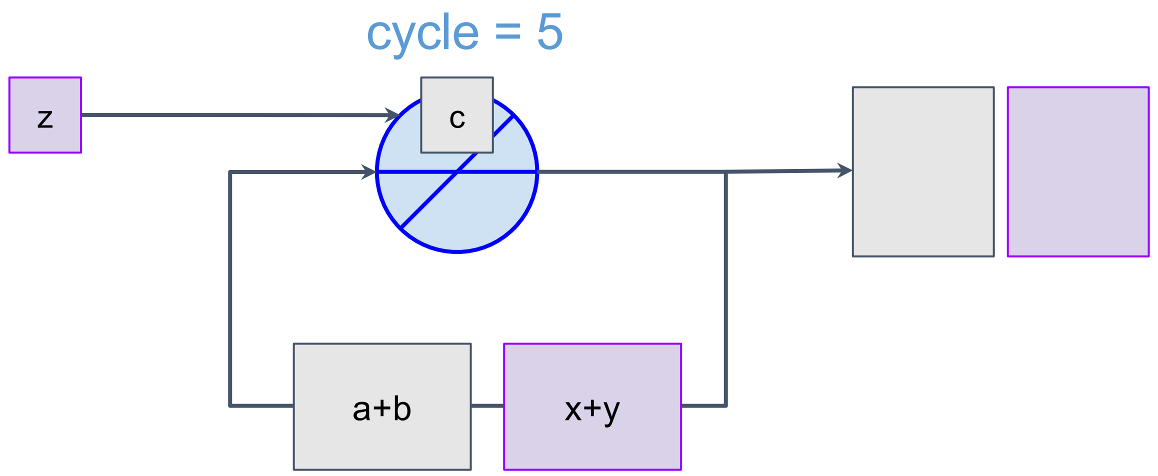
Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



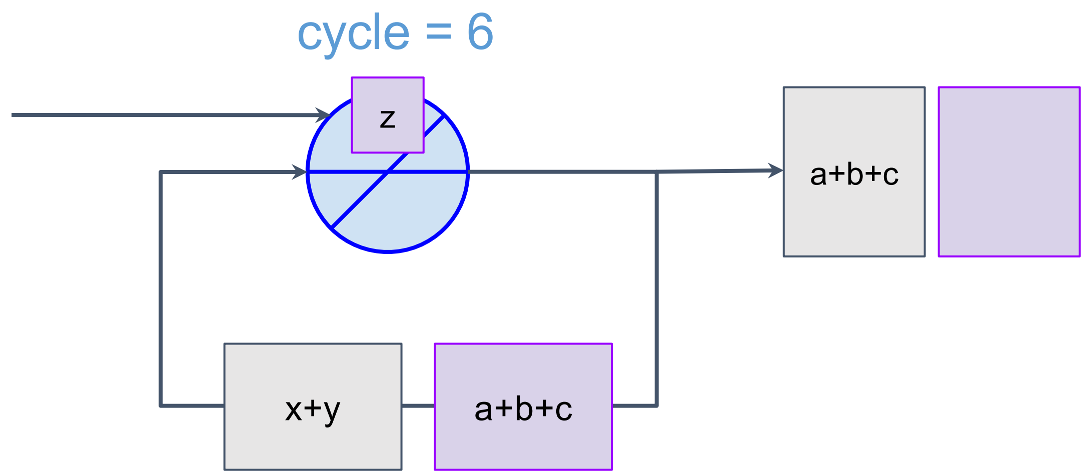
Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



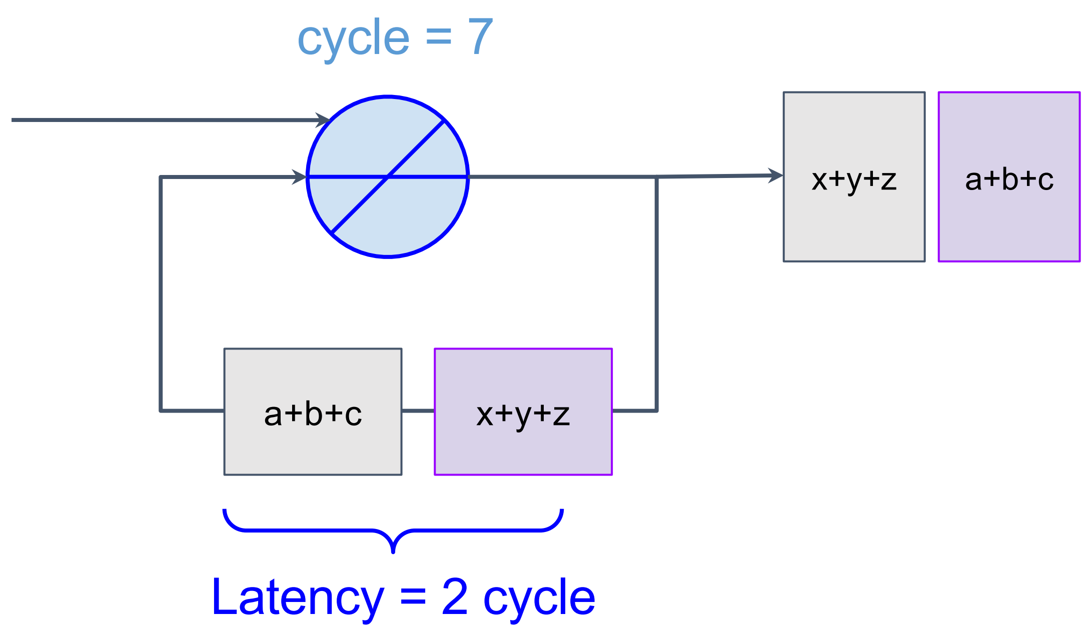
Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop



Interleaving

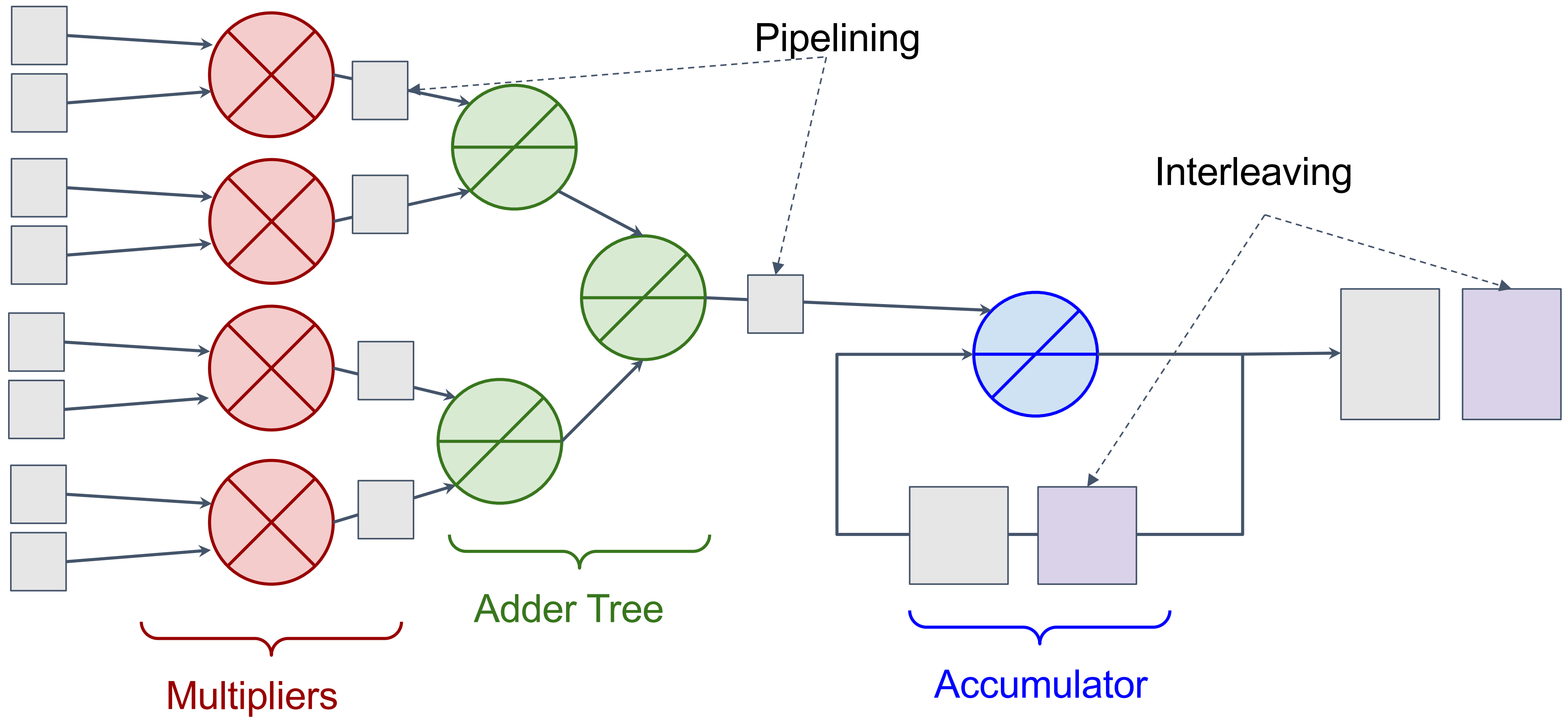
Initiation Interval: How often I can start the computation of a new element of a loop



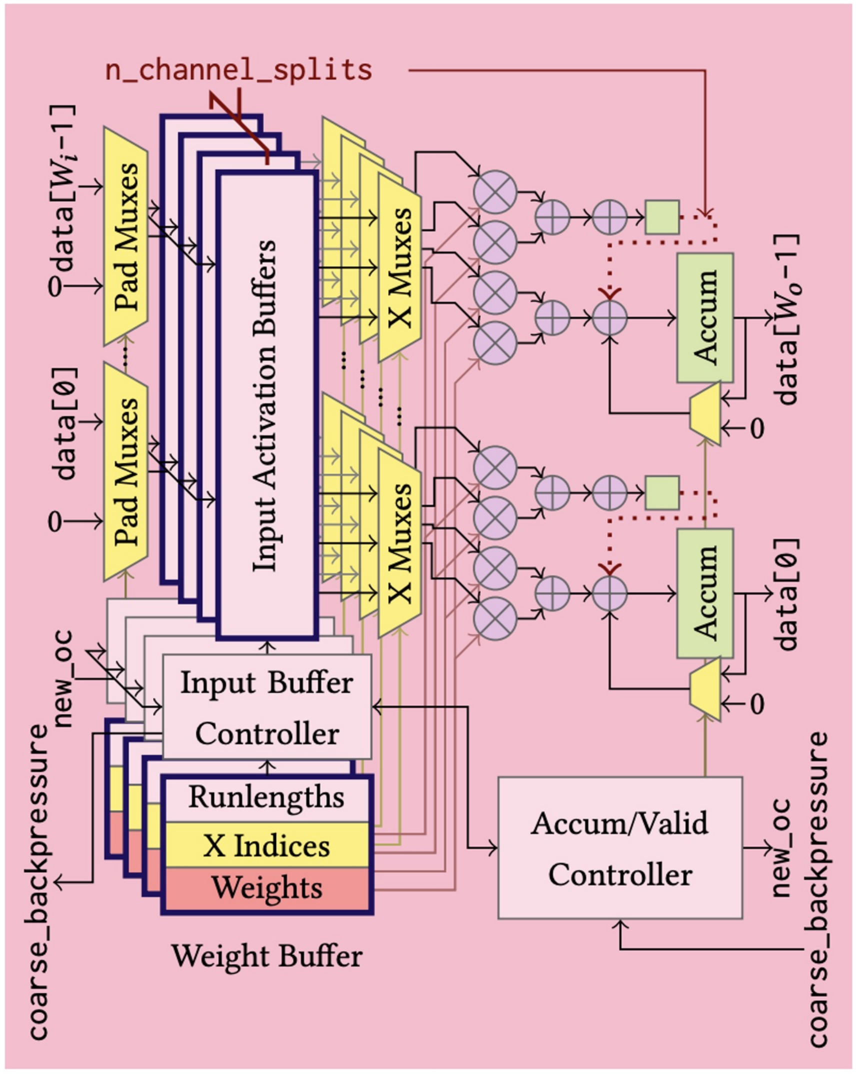
Now, what is my throughput? **1 op/cycle** Note: II = 1

Vectorization, Pipelining and Interleaving

Initiation Interval: How often I can start the computation of a new element of a loop

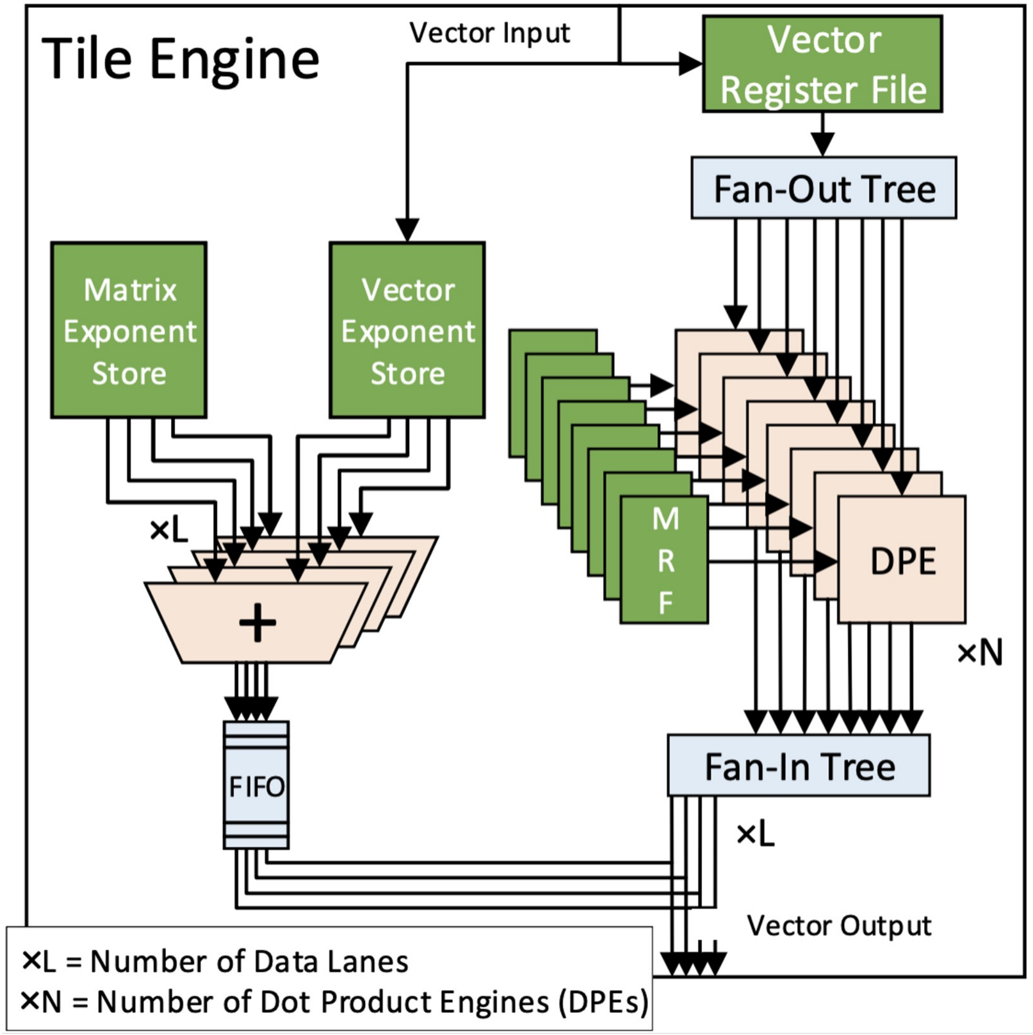


PEs in the wild



Hall and Betz: HPIPE

Chung et al. Brainwave



Outline

1. PE Microarchitecture

a. Parallelization

b. Pipelining

c. Interleaving

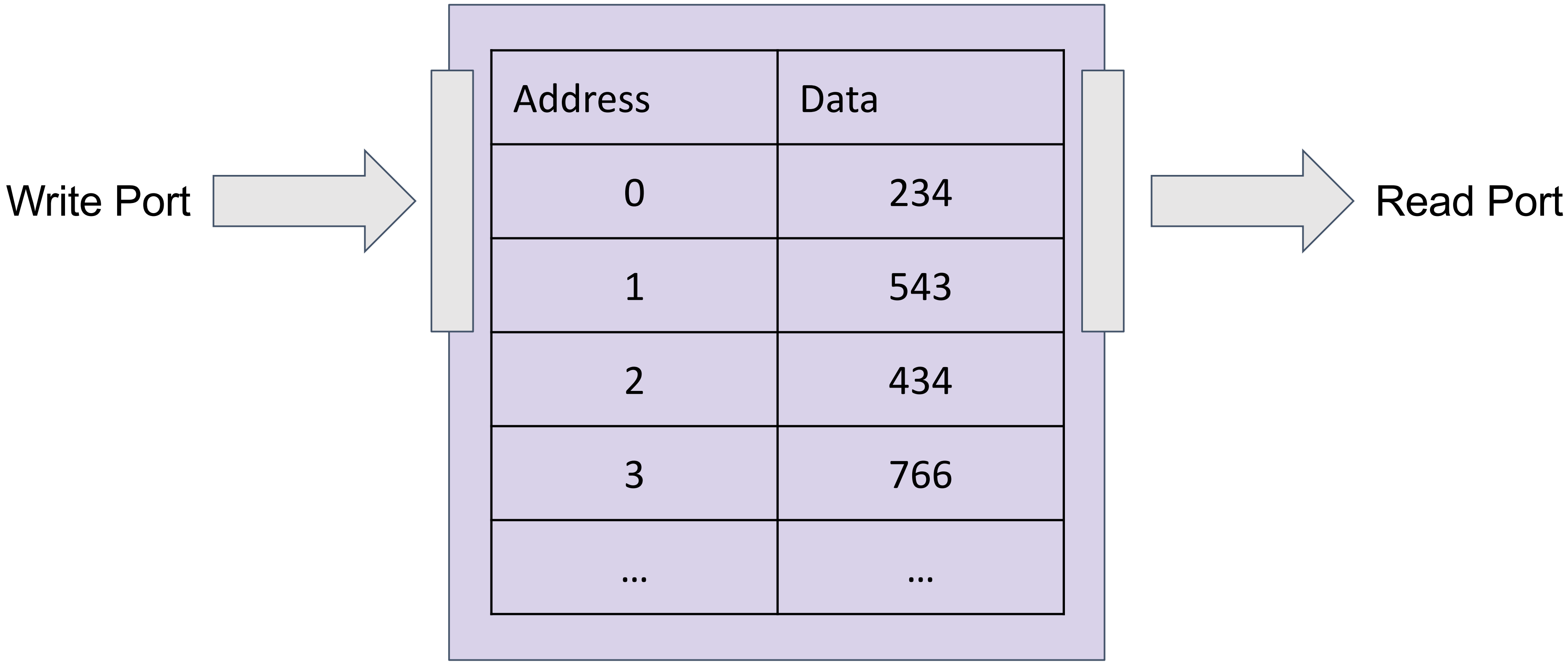
d. Arithmetic

2. On-Chip Memory

b. Basics

c. Banking

On-Chip Memory (SRAM)



Address	Data
0	234
1	543
2	434
3	766
…	…

QUESTION

What decides the bit-width of the addresses?

1. The width of the data
2. The number of data entries
3. The address bus size
4. The data bus size

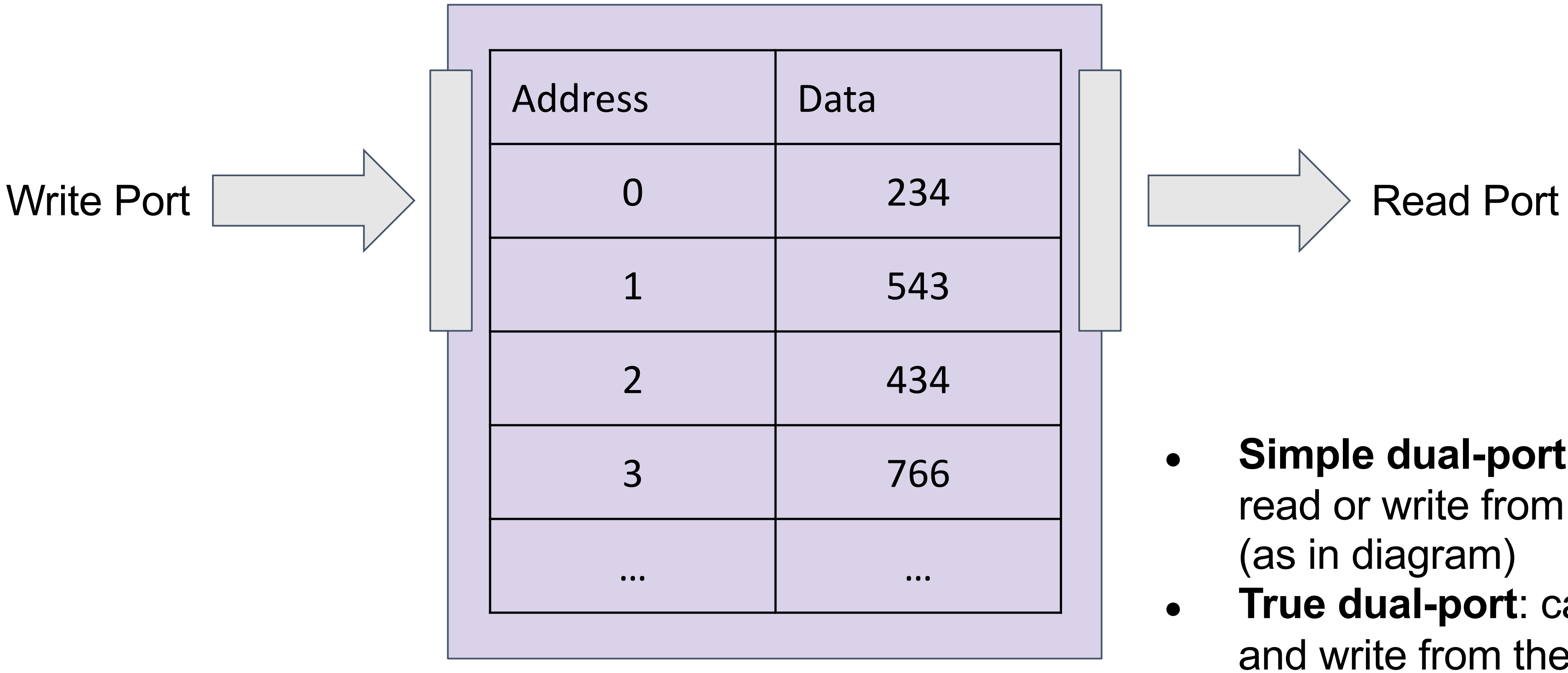
QUESTION

What decides the bit-width of the addresses?

1. ~~The width of the data~~
2. The number of data entries
3. ~~The address bus size~~
4. ~~The data bus size~~

Has nothing to do with data. Need b bits to represent 2^b entries

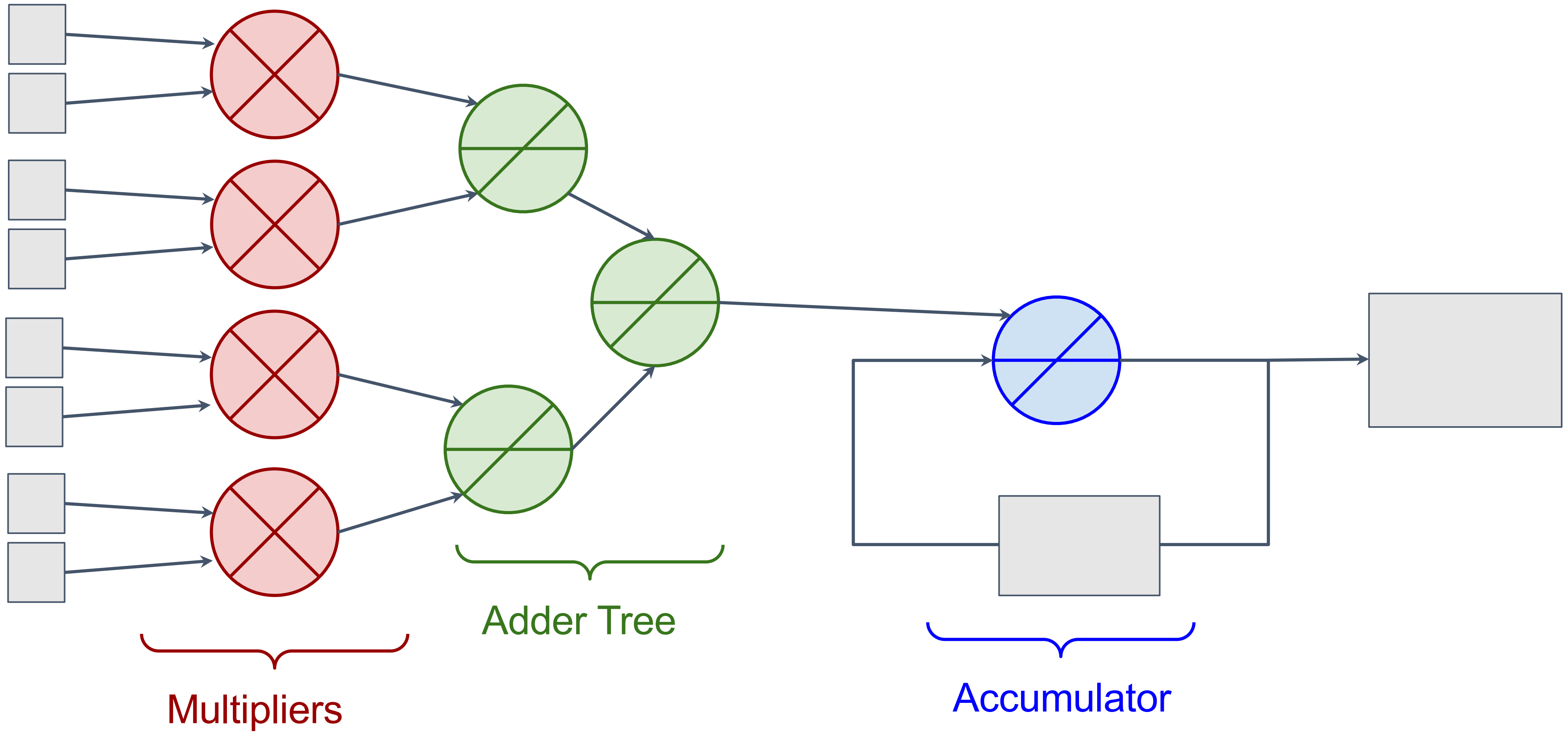
On-Chip Memory (SRAM)



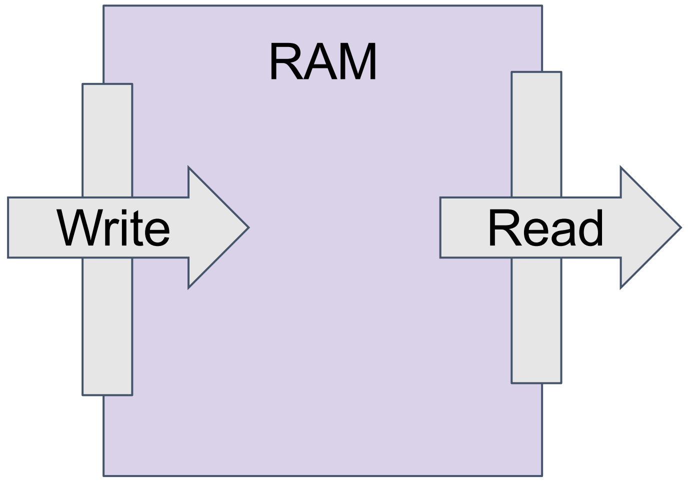
Address	Data
0	234
1	543
2	434
3	766
…	…

- **Simple dual-port**: can either read or write from one port (as in diagram)
- **True dual-port**: can both read and write from the same port

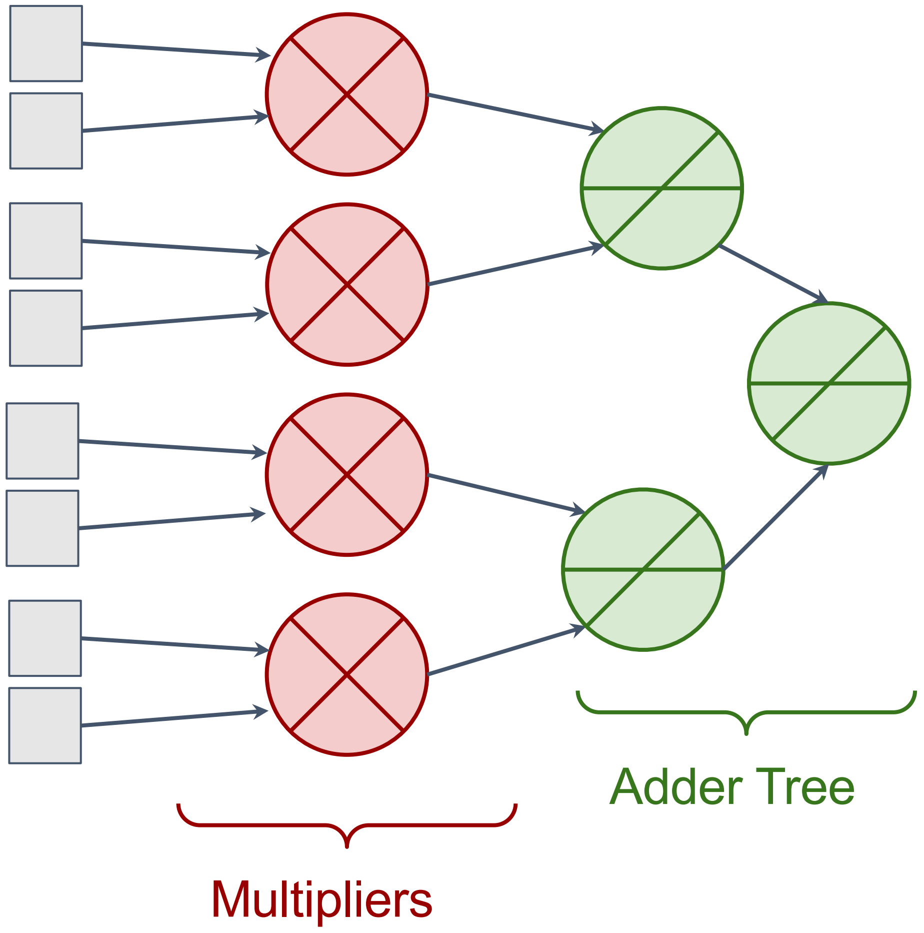
Connect RAM to MAC



Connect RAM to MAC

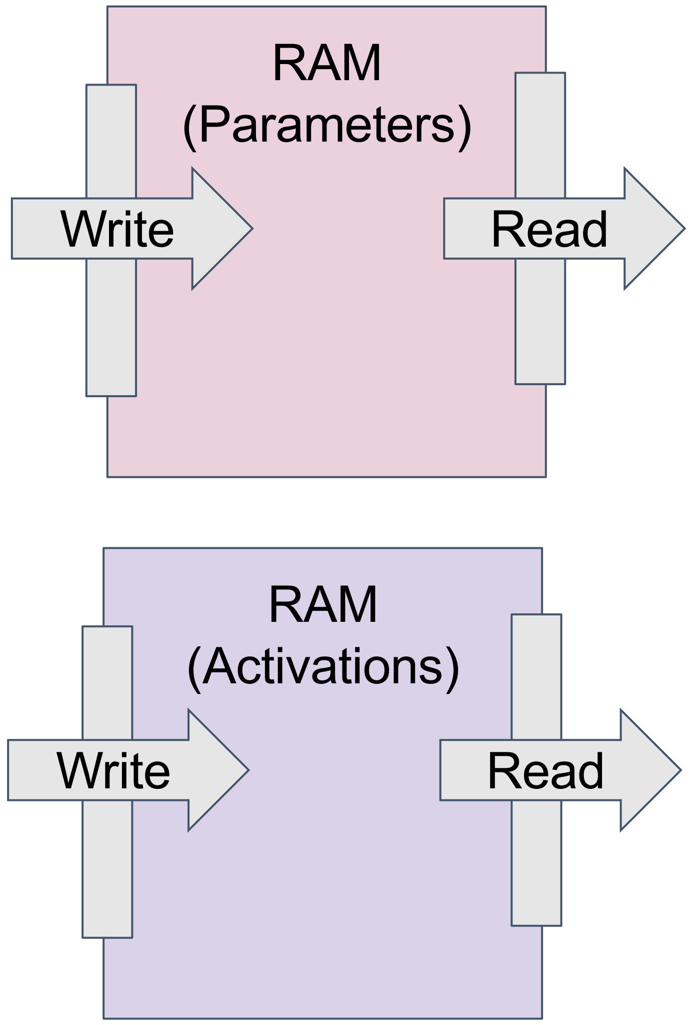


?

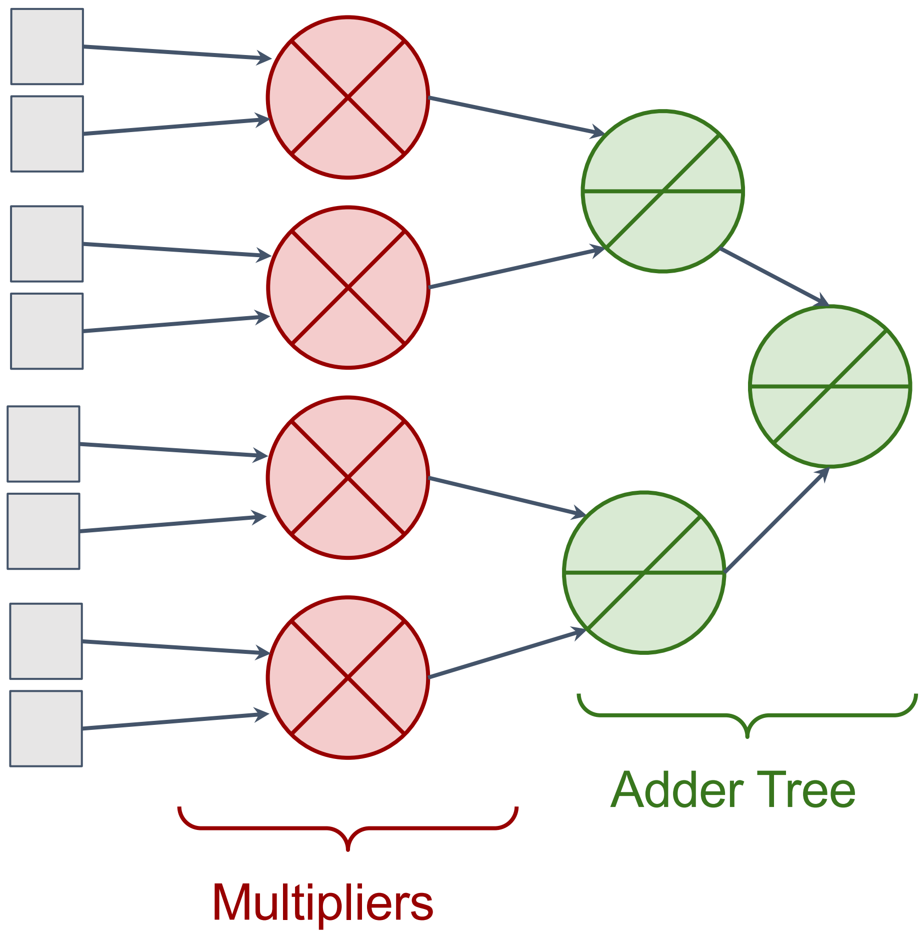


Connect RAM to MAC

Use different memories for the two operands

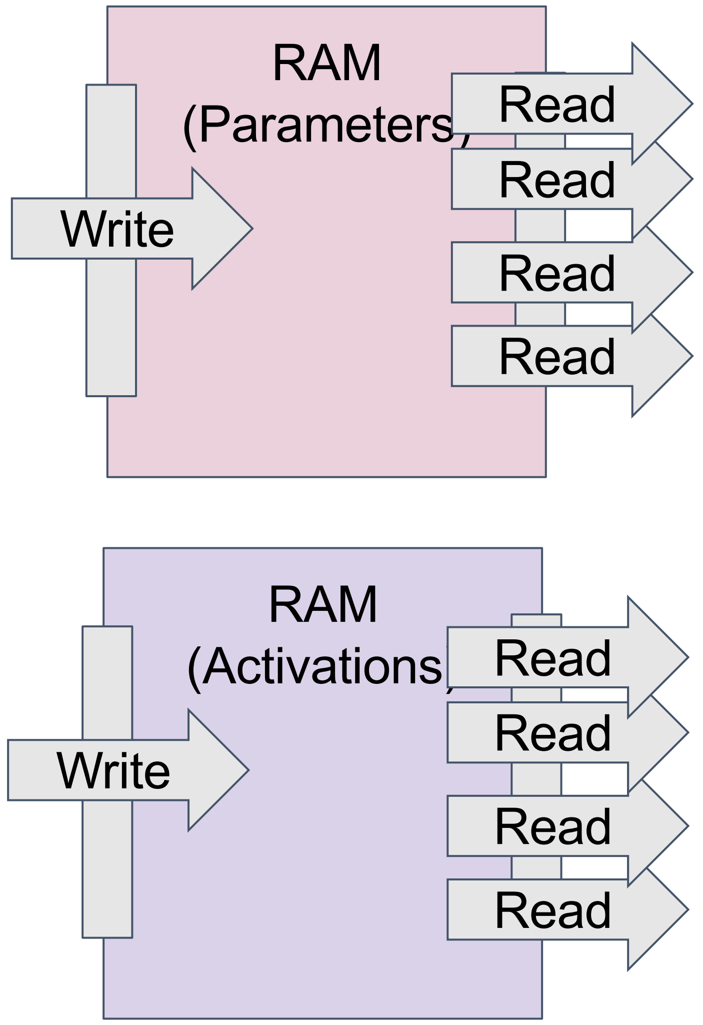


?

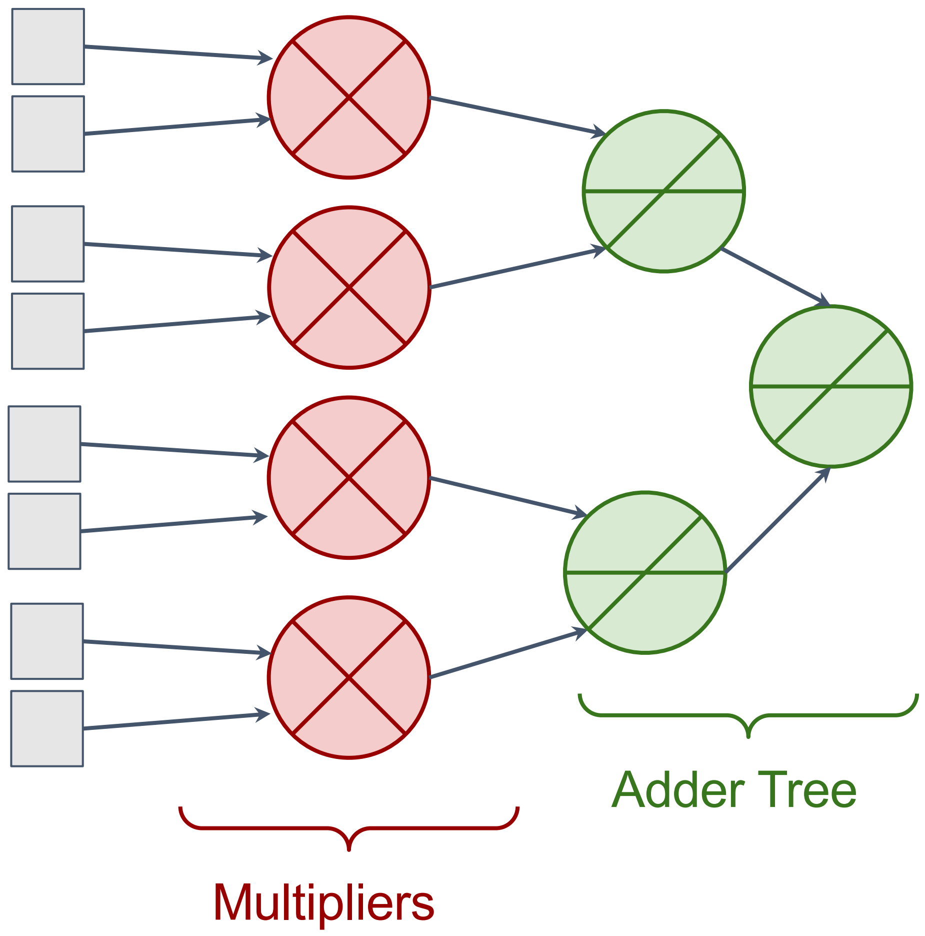


Connect RAM to MAC

Duplicate number of ports to read 4 elements per cycle

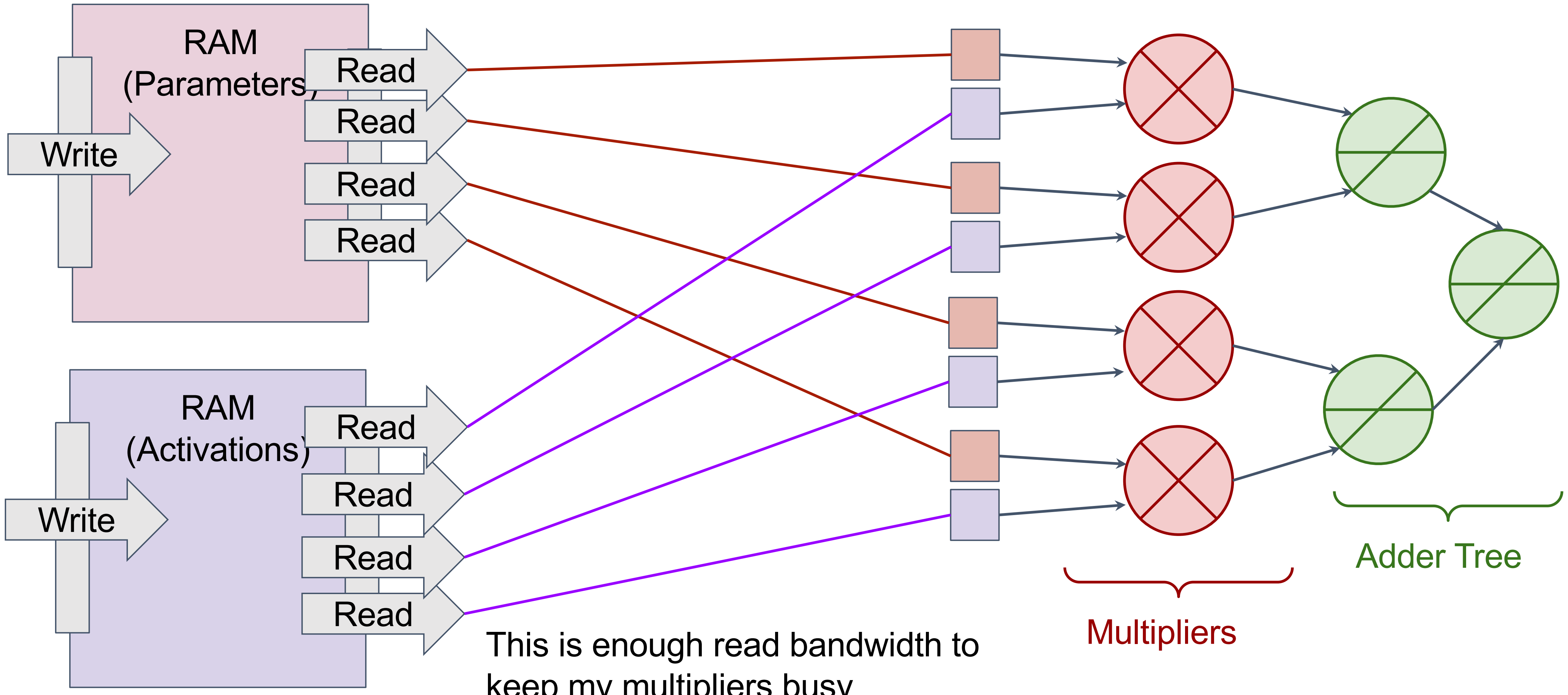


?



Duplicate number of ports to read 4 elements per cycle

Connect RAM to MAC



This is enough read bandwidth to keep my multipliers busy

SFU

QUESTION

What is wrong with adding many read ports to SRAM?

1. SRAM will be slow
2. SRAM will be large
3. SRAM will be power-hungry
4. Nothing is wrong, it’s fine

QUESTION

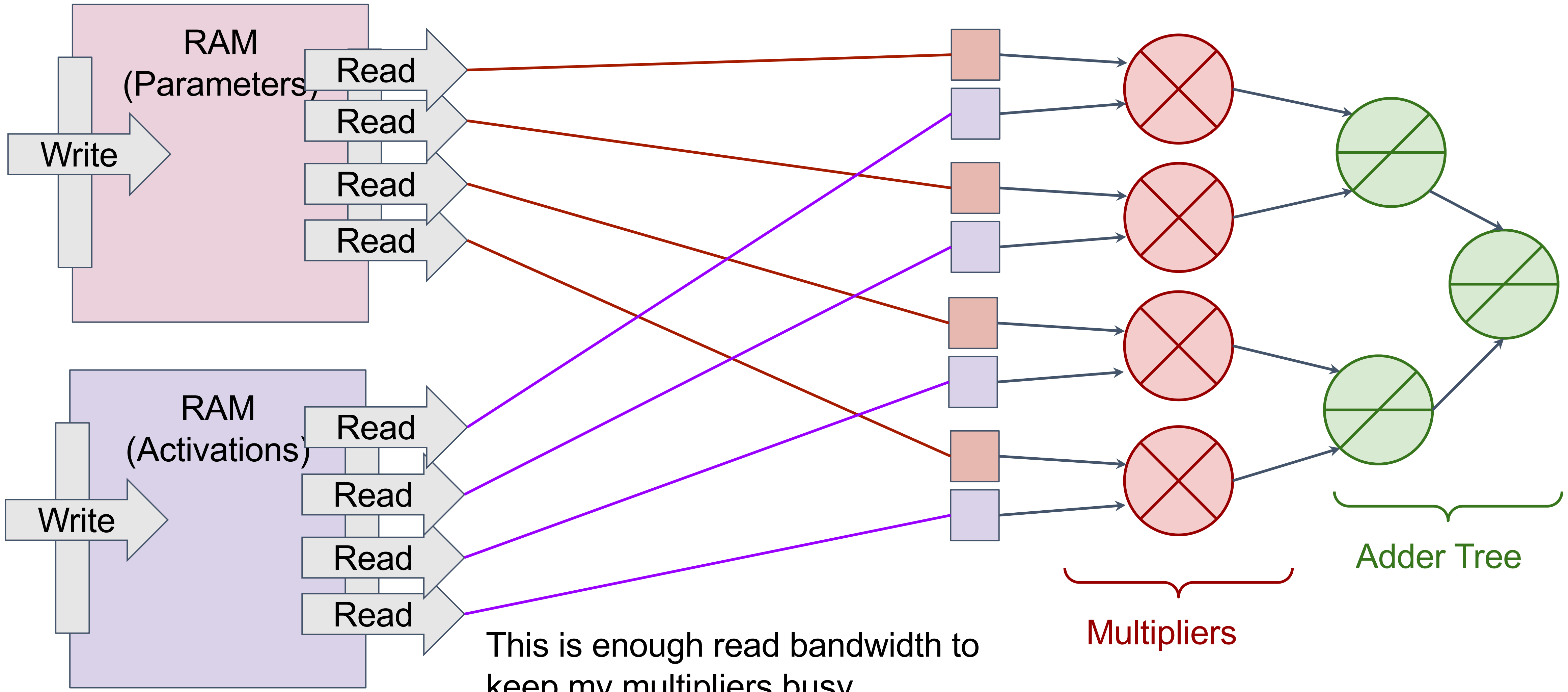
What is wrong with adding many read ports to SRAM?

1. SRAM will be slow
2. SRAM will be large
3. SRAM will be power-hungry
4. ~~Nothing is wrong, it’s fine~~

Circuitry to support multiple concurrent reads to the same SRAM cells is expensive

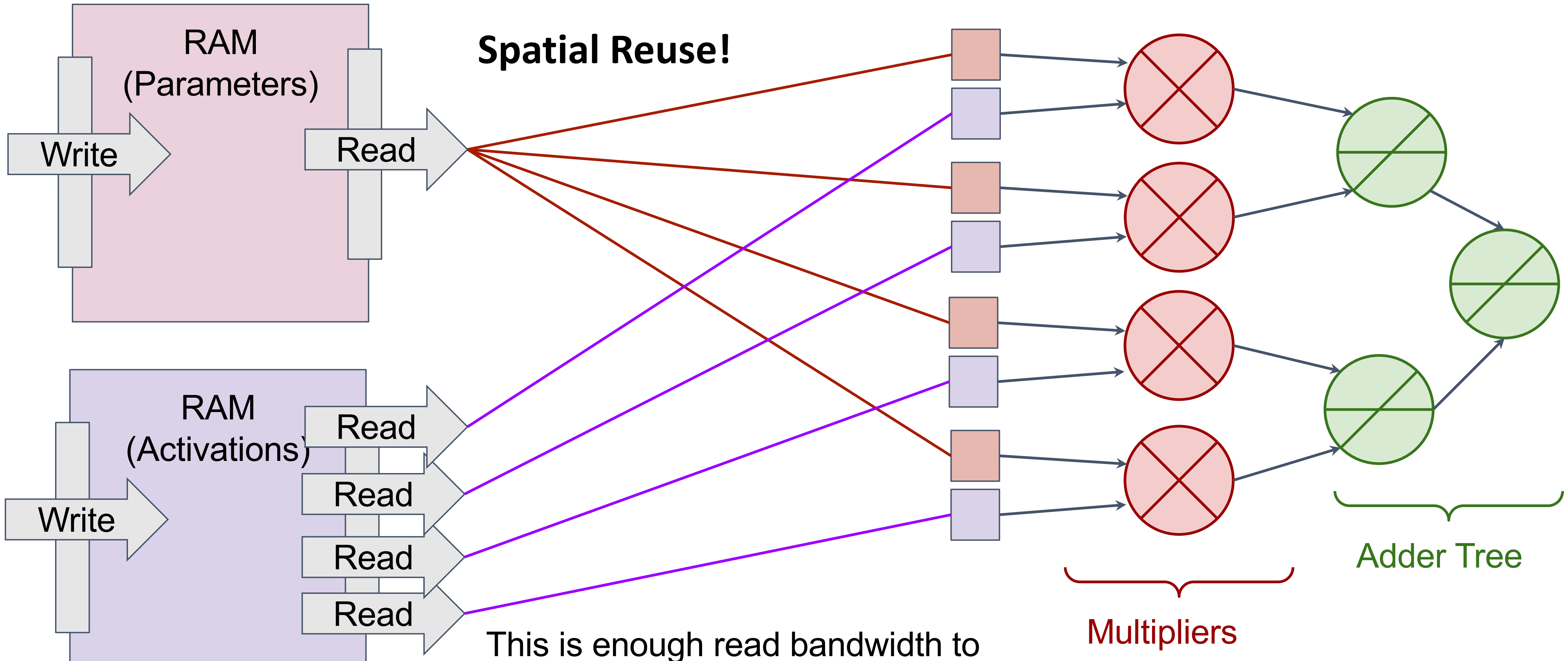
Duplicate number of ports to read 4 elements per cycle

Connect RAM to MAC



This is enough read bandwidth to keep my multipliers busy

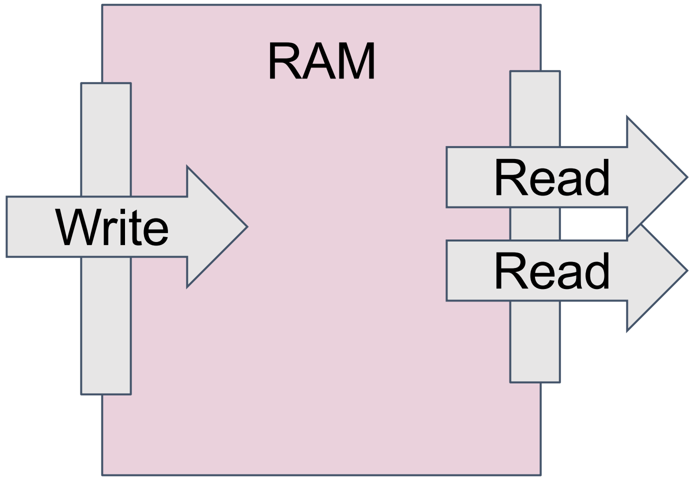
Connect RAM to MAC



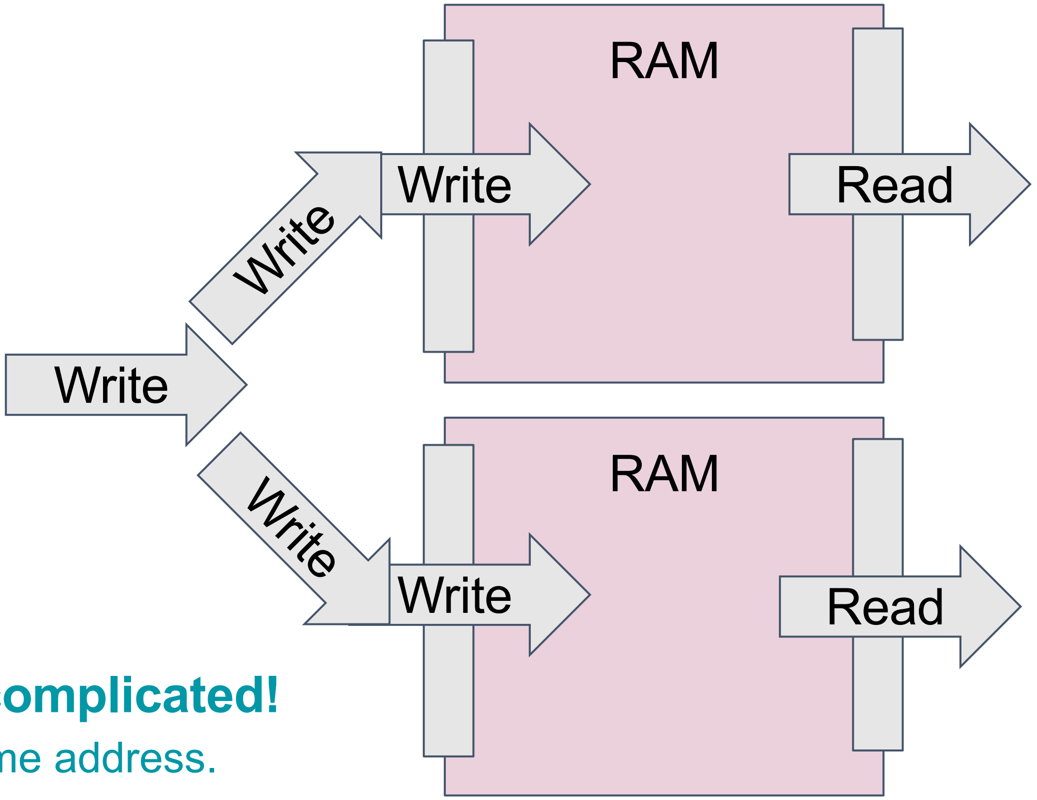
This is enough read bandwidth to keep my multipliers busy

Multiported Memories

ASICs: Adding more ports increases area/power and delay in the SRAM circuitry



FPGAs: You need to duplicate your memories!



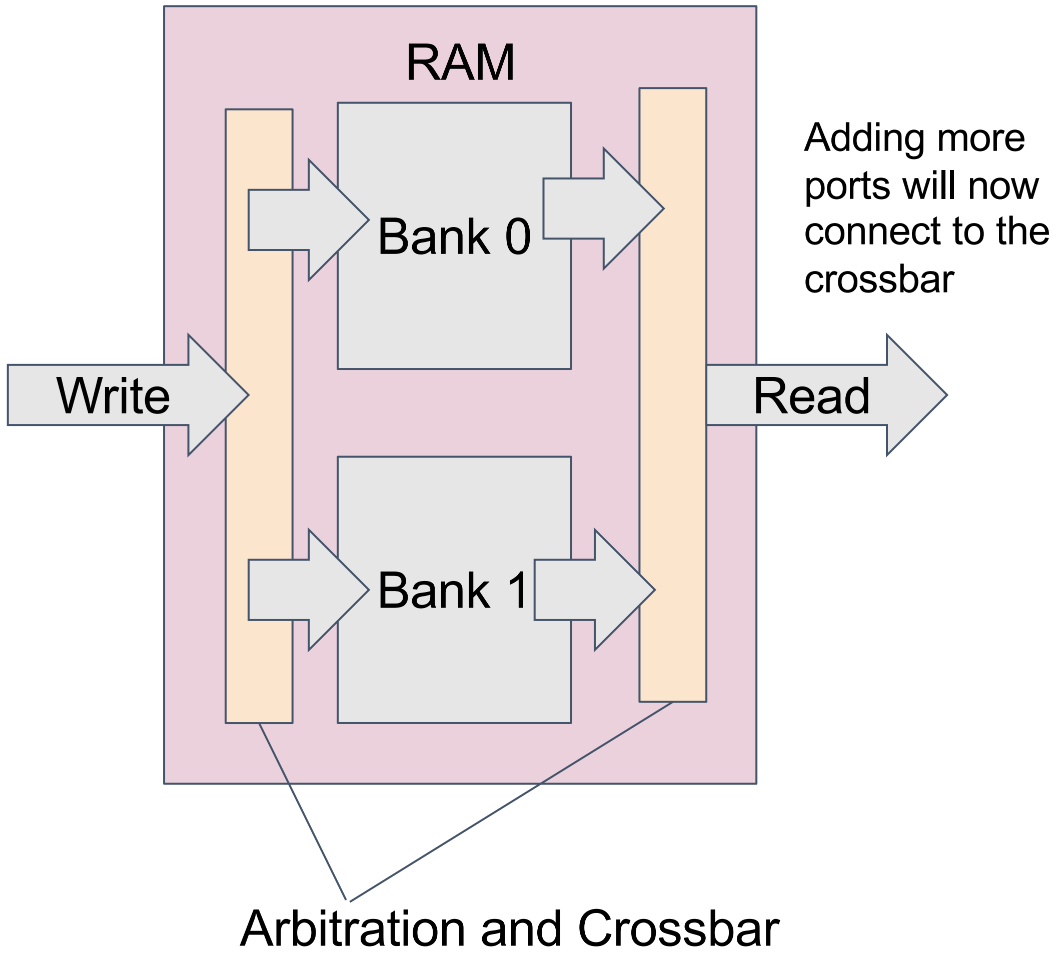
NOTE: Multiple write ports are even more complicated!

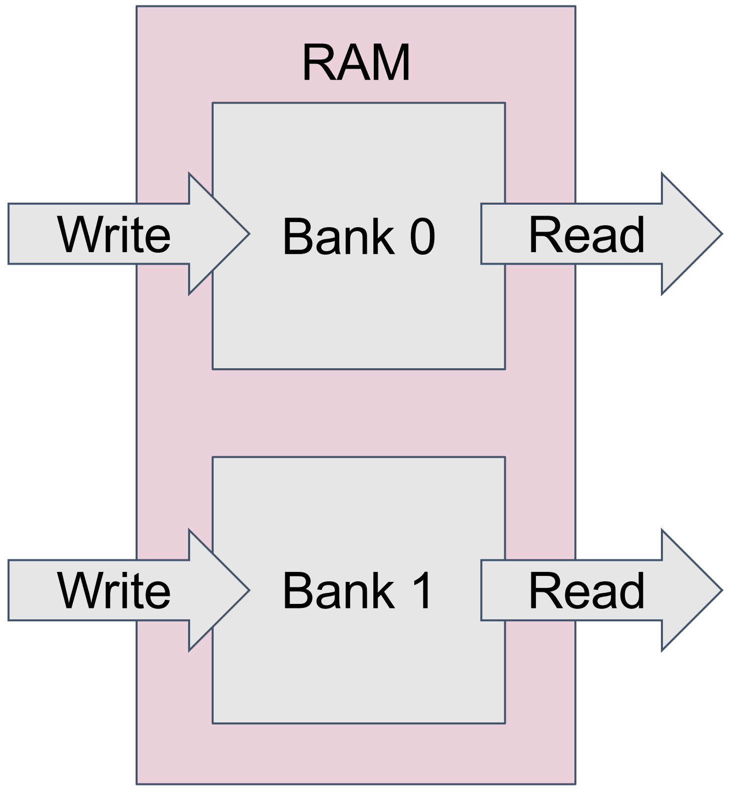
Need to decide what to do when two ports access same address.

Rule of Thumb

"Use small fast memory together large slow memory to provide illusion of large fast memory" - John Wawrzynek and Krste Asanovic

Memory Banking





Explicitly-managed banks are common in on-chip memories

Outline

1. PE Microarchitecture

a. Parallelization
b. Pipelining
c. Interleaving
d. Arithmetic

2. On-Chip Memory

b. Basics
c. Banking