6.5930/1

Hardware Architectures for Deep Learning

Mapping to Hardware

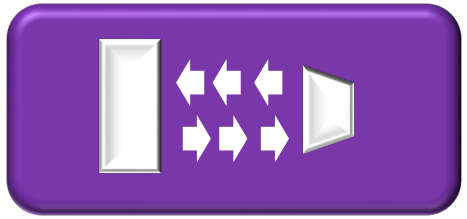
March 18, 2024

Joel Emer and Vivienne Sze

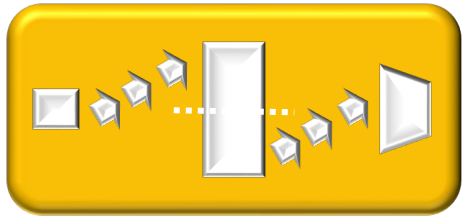
Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Data Orchestration

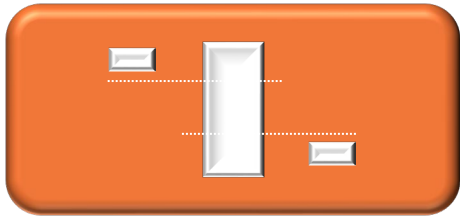
Guiding Principles for Data Orchestration



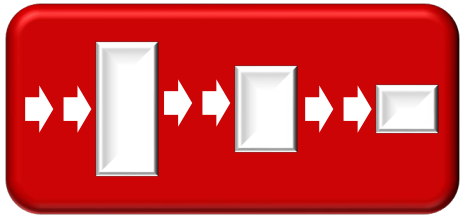
Efficient reuse – small storage physically close to consuming units for reused data



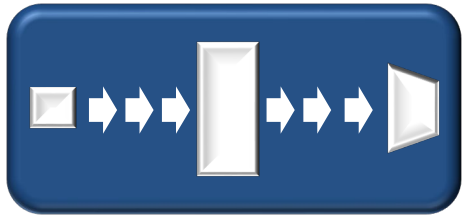
Delivery/use overlap - Next tile should be available when current is done (e.g., double-buffering)



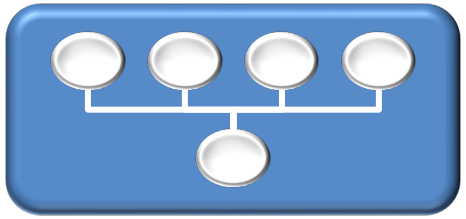
Precise synchronization – Only wait for exactly data you need, respond quickly (e.g., no barriers or remote polling)



Storage usage efficiency – Minimize idle storage waiting for long round trip latency



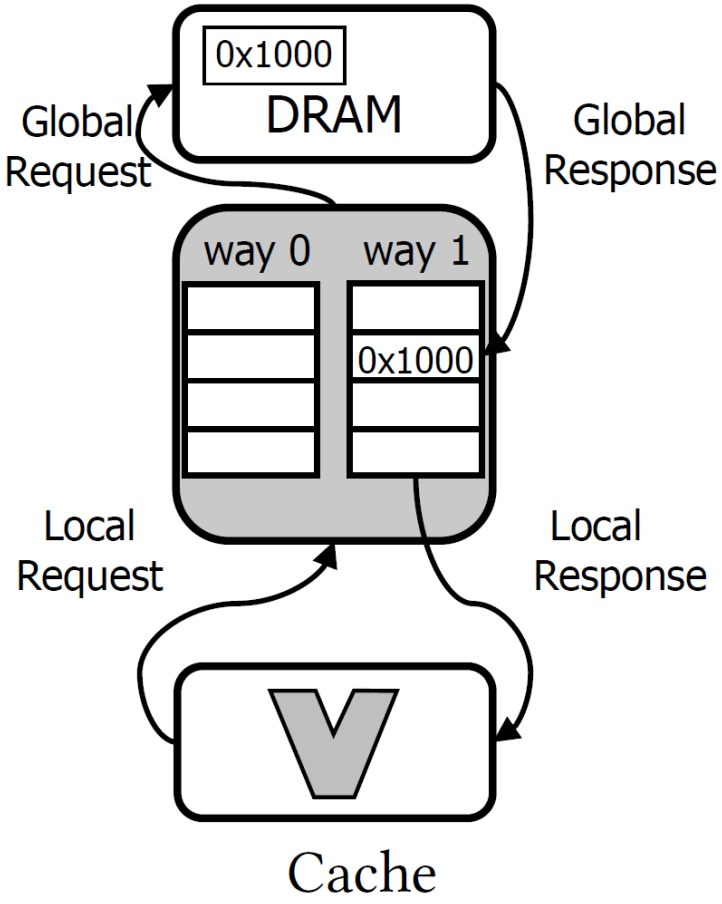
Bandwidth efficiency - Maximize delivery rate by controlling outstanding requests



Cross-unit use – amortize data access and communication

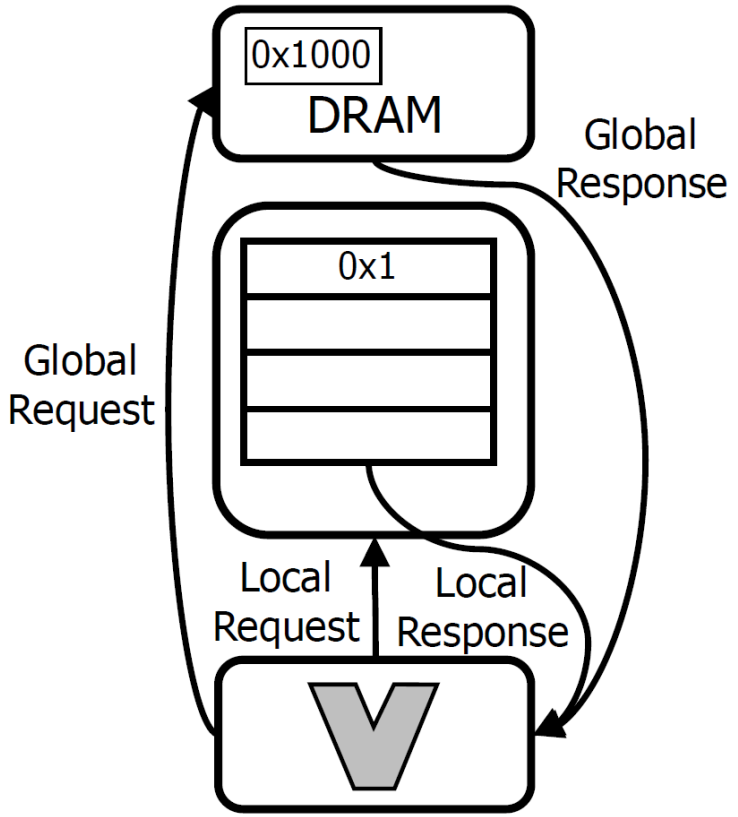
Approaches: Implicit versus Explicit

Implicit:



Cache

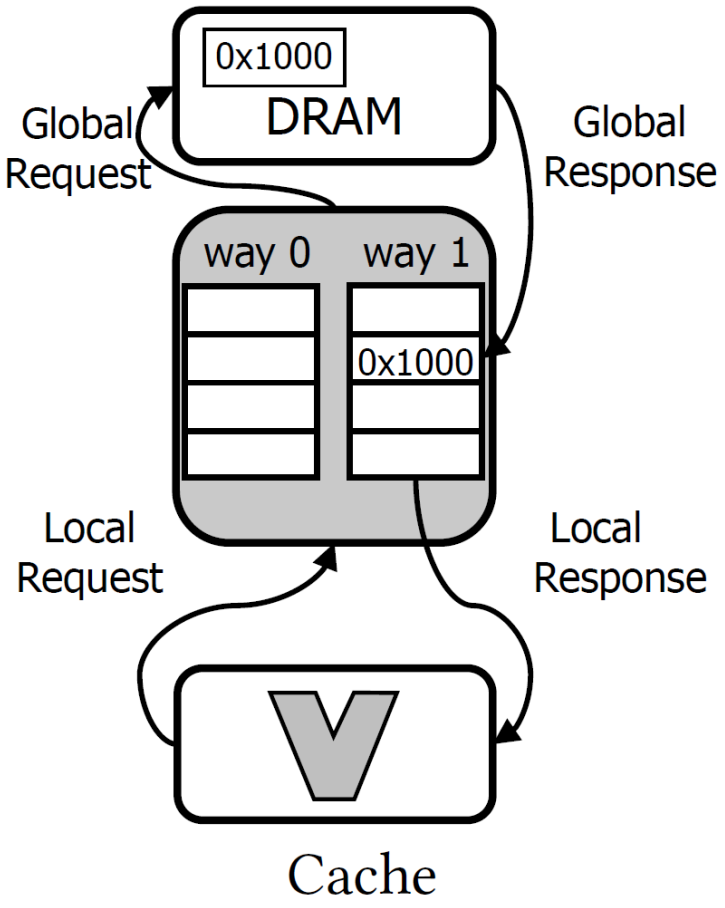
Explicit:



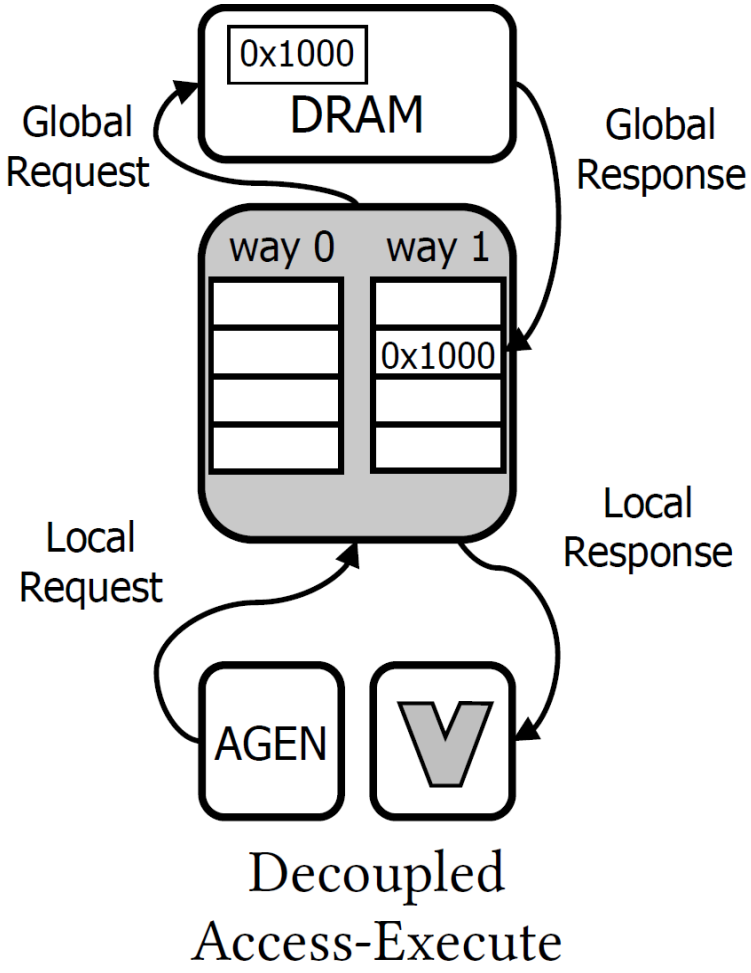
Scratchpad (e.g., GPU shared memory)

Approaches: Coupled versus Decoupled

Implicit + Coupled

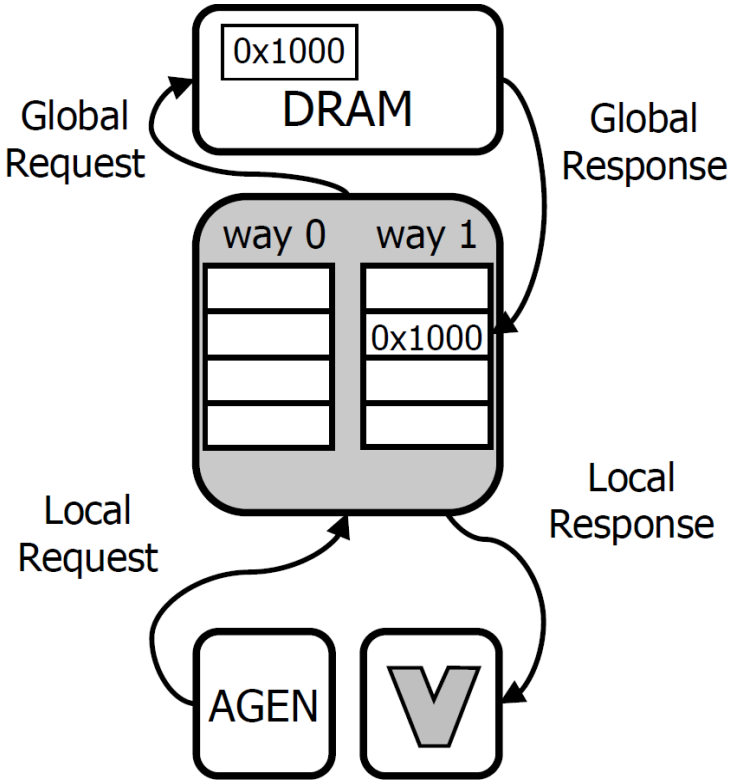


Implicit + *Decoupled*



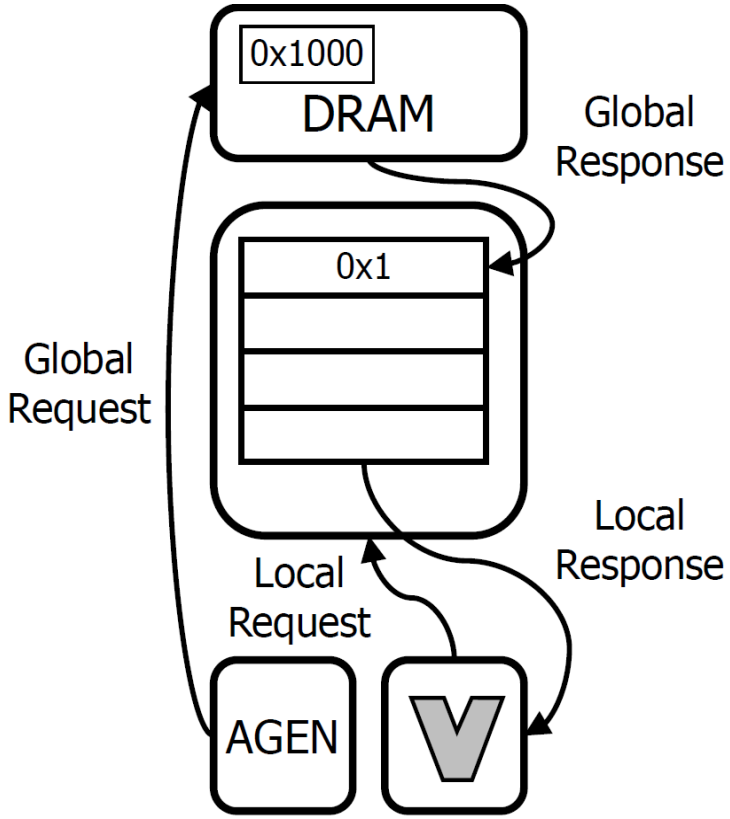
Explicit Decoupled Data Orchestration

Implicit + *Decoupled*



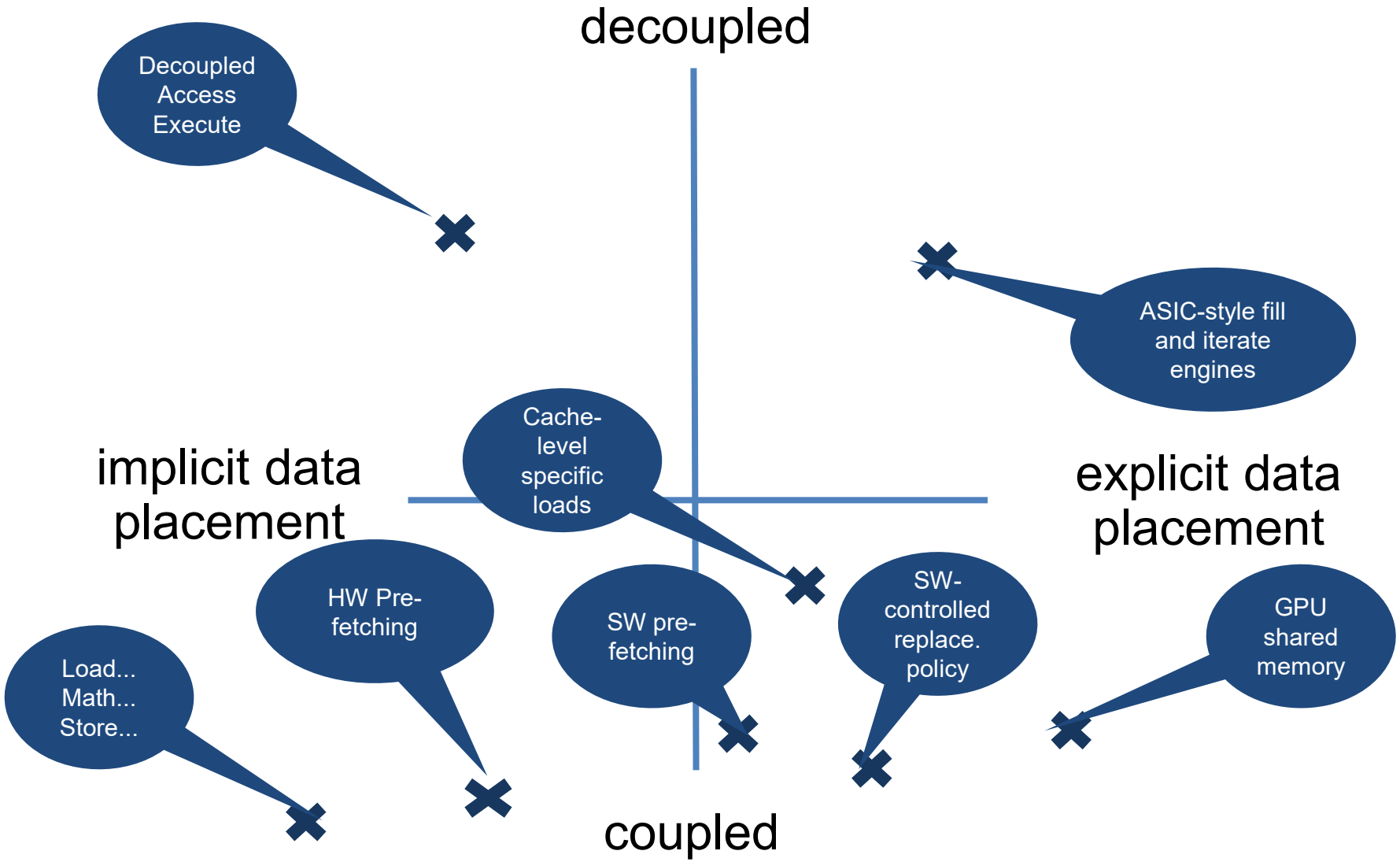
Decoupled Access-Execute

Explicit + Decoupled



DMA Engines (EDDO - *this work*)

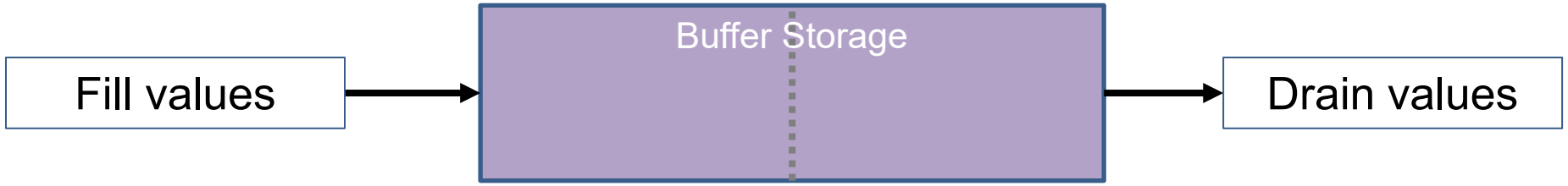
Classifying Orchestration Approaches



EDDO Strategies

Fill and use:

Double buffer

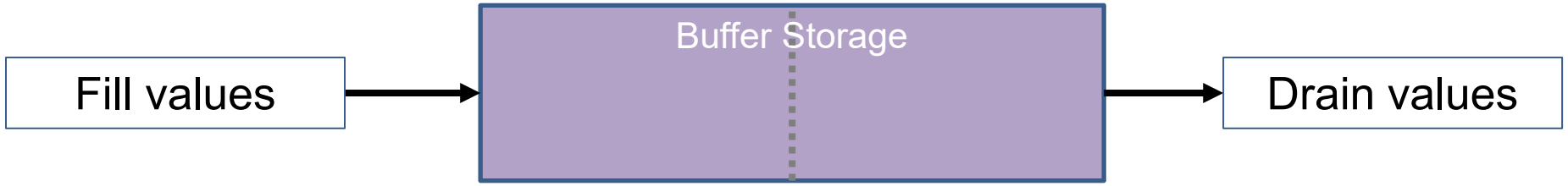


Rolling use

EDDO Strategies

Fill and use:

Double buffer

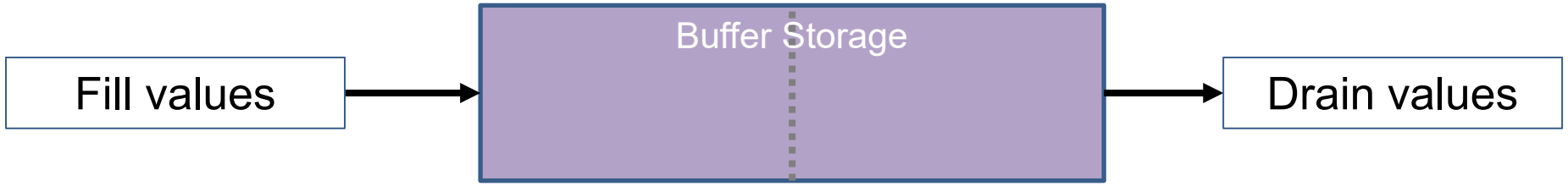


Rolling use

EDDO Strategies

Fill and use:

Double buffer

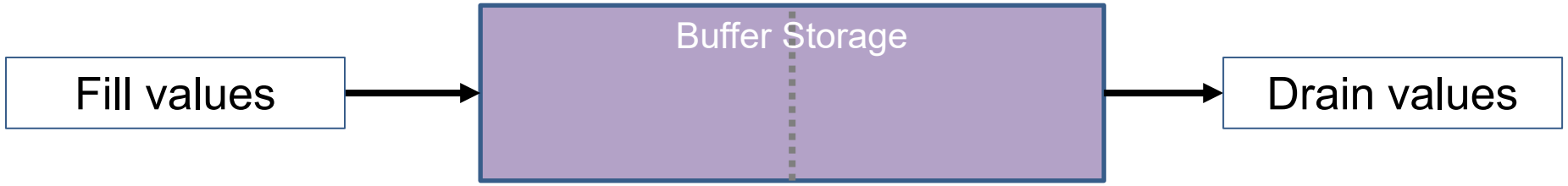


Rolling use

EDDO Strategies

Fill and use:

Double buffer

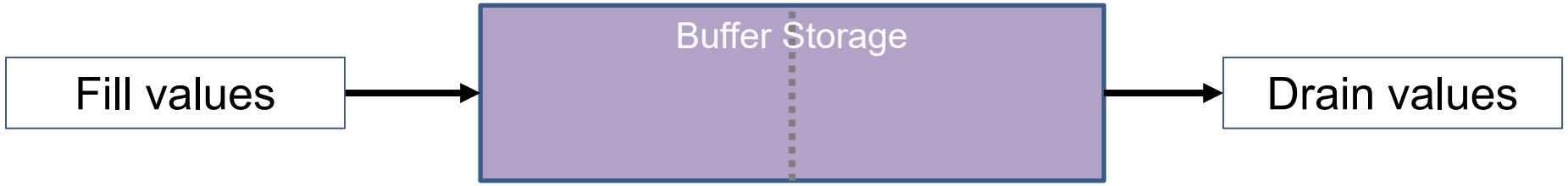


Rolling use

EDDO Strategies

Fill and use:

Double buffer

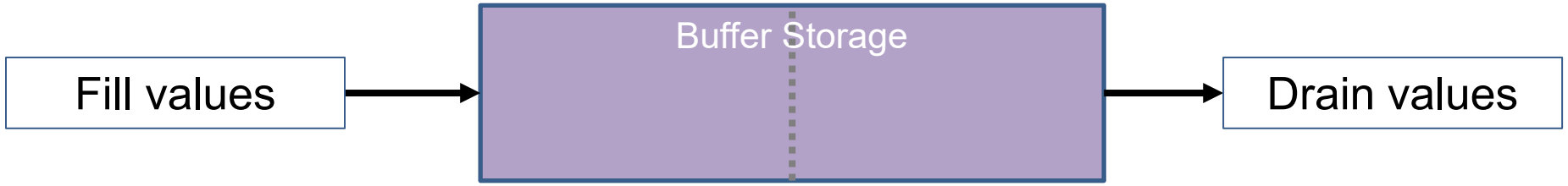


Rolling use

EDDO Strategies

Fill and use:

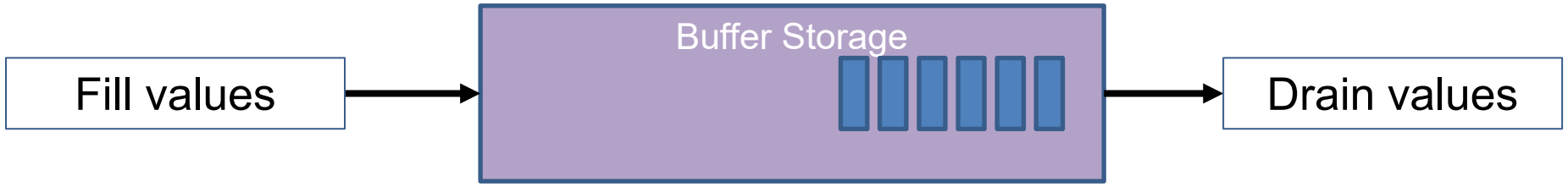
Double buffer



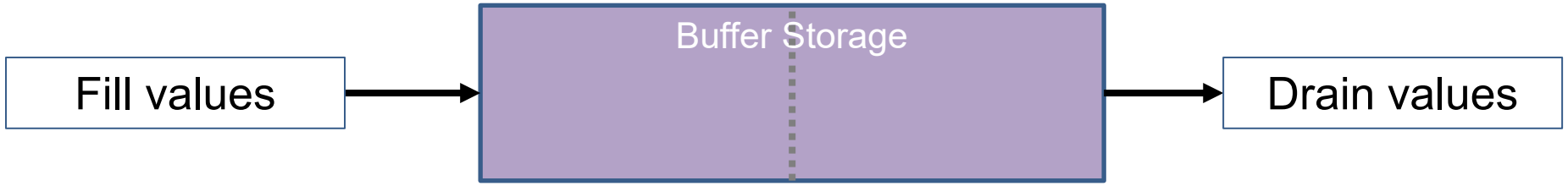
Rolling use

EDDO Strategies

Fill and use:



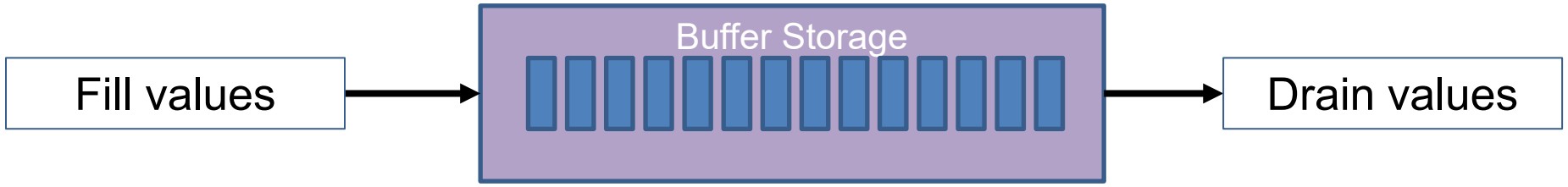
Double buffer



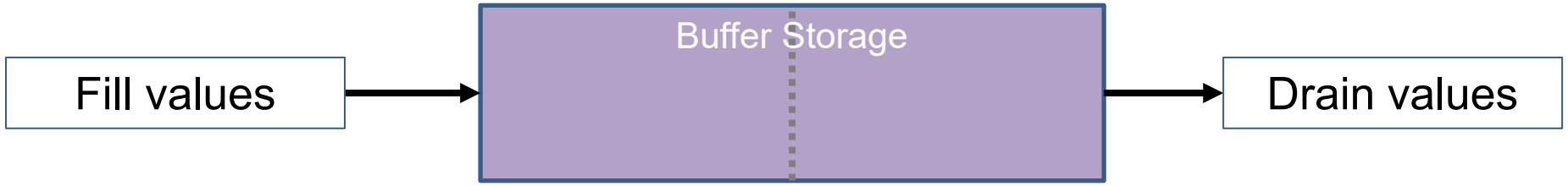
Rolling use

EDDO Strategies

Fill and use:



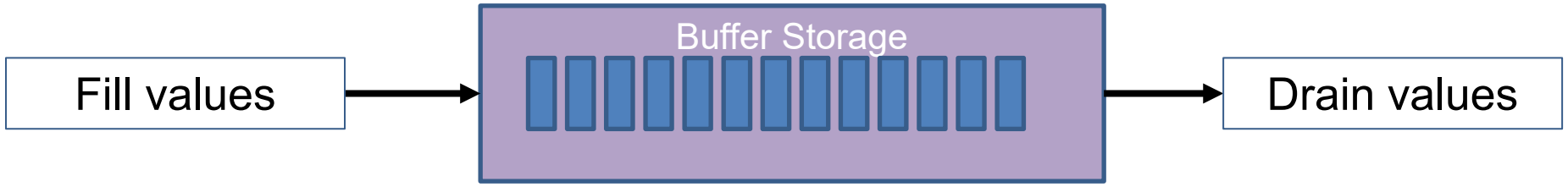
Double buffer



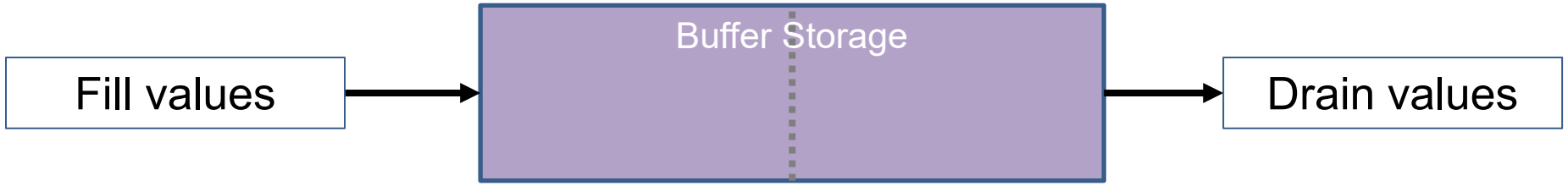
Rolling use

EDDO Strategies

Fill and use:



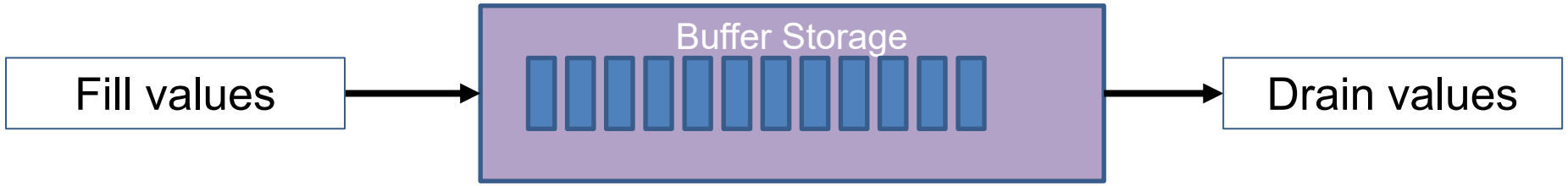
Double buffer



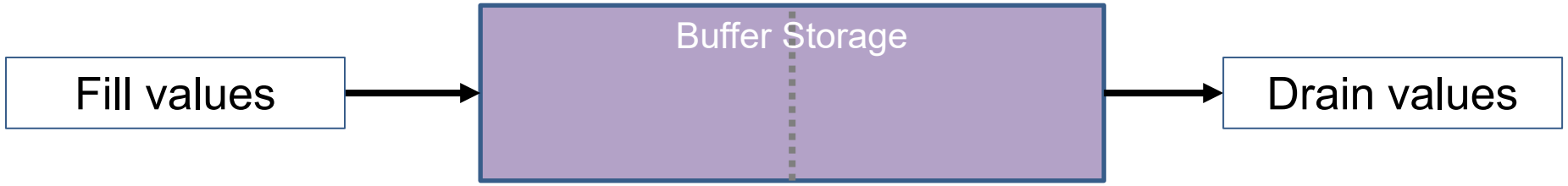
Rolling use

EDDO Strategies

Fill and use:



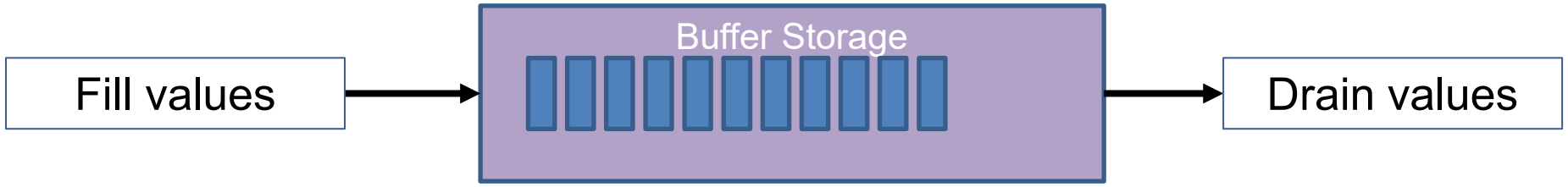
Double buffer



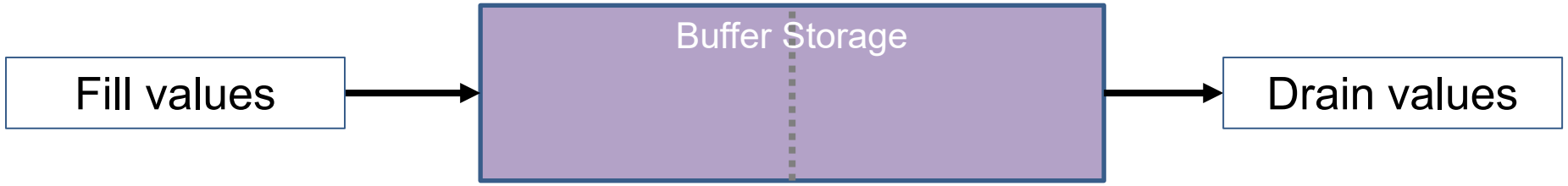
Rolling use

EDDO Strategies

Fill and use:



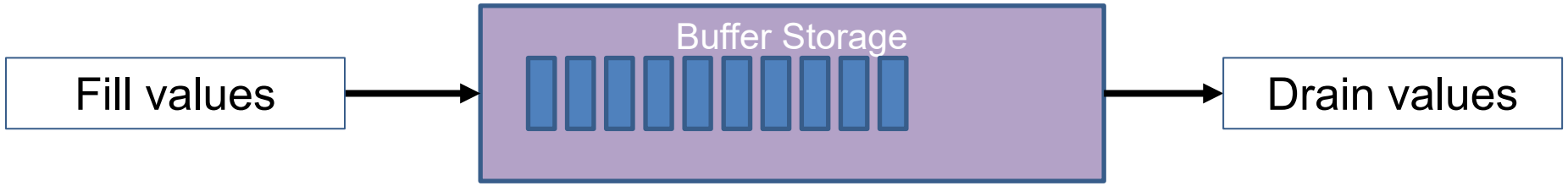
Double buffer



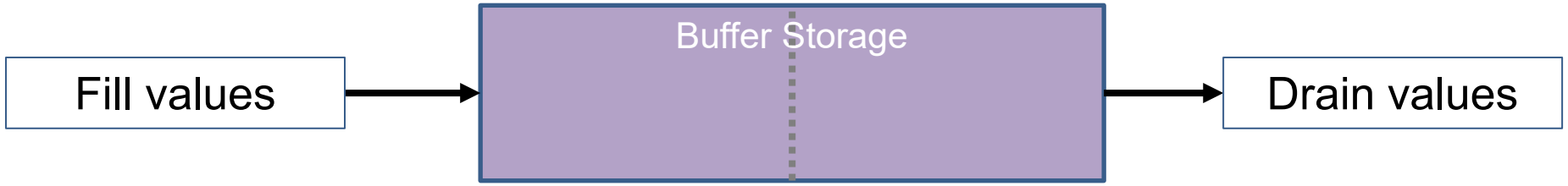
Rolling use

EDDO Strategies

Fill and use:



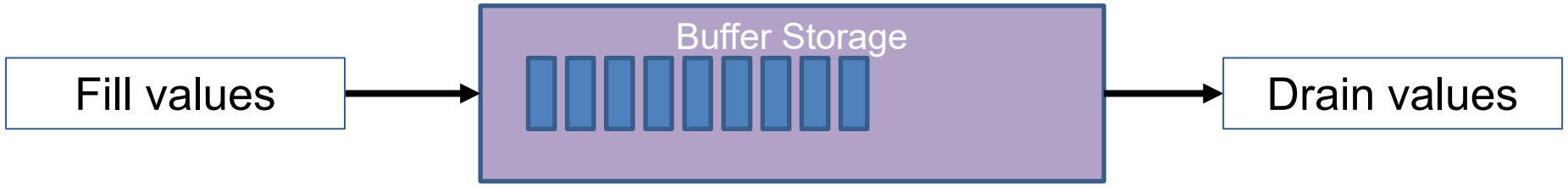
Double buffer



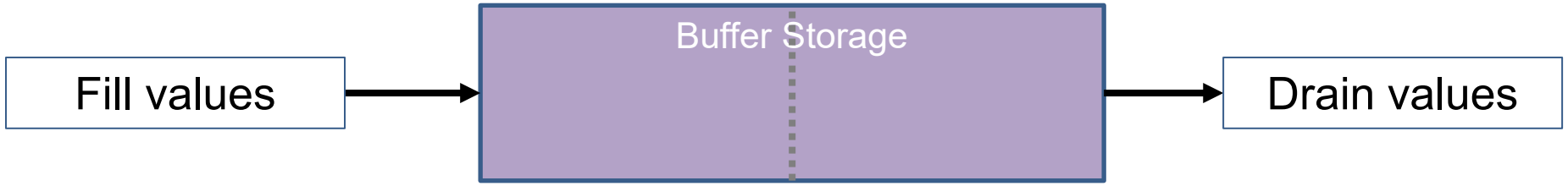
Rolling use

EDDO Strategies

Fill and use:



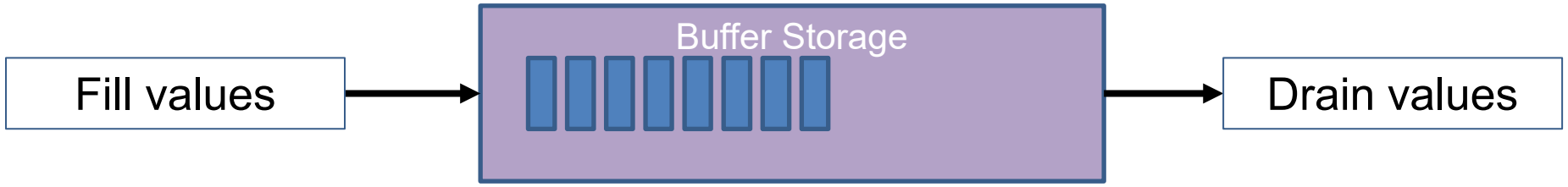
Double buffer



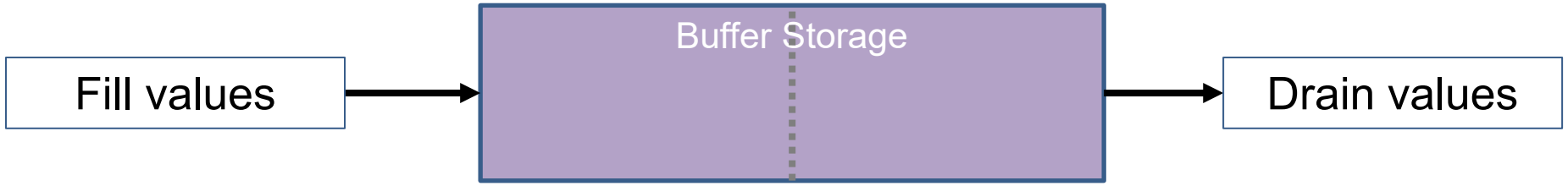
Rolling use

EDDO Strategies

Fill and use:



Double buffer

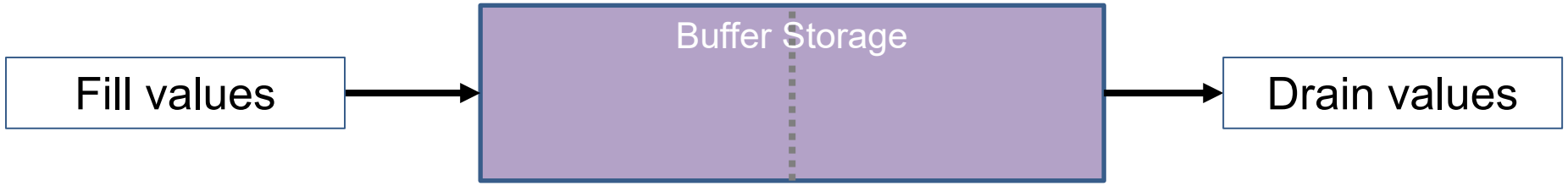


Rolling use

EDDO Strategies

Fill and use:

Double buffer

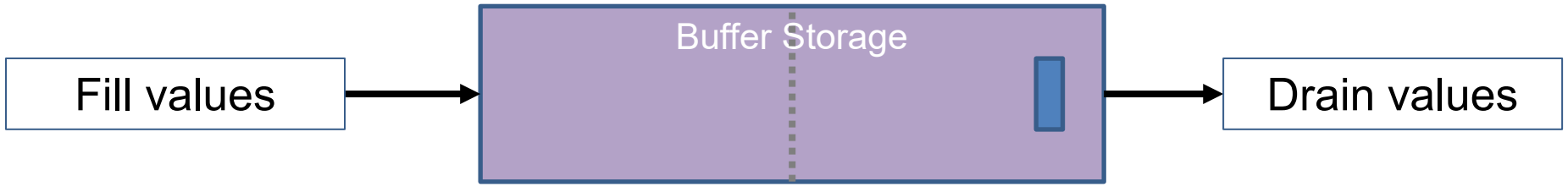


Rolling use

EDDO Strategies

Fill and use:

Double buffer

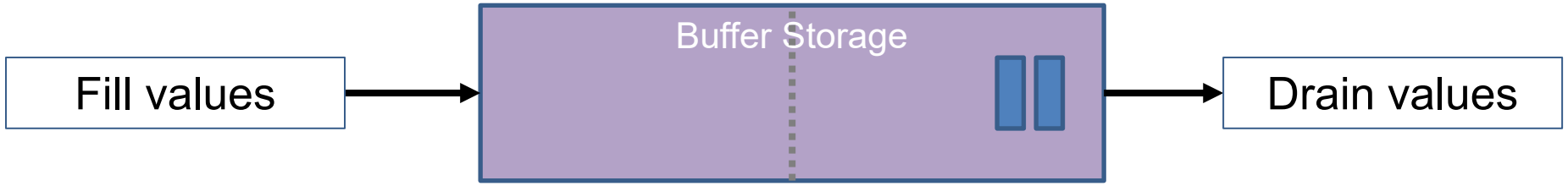


Rolling use

EDDO Strategies

Fill and use:

Double buffer

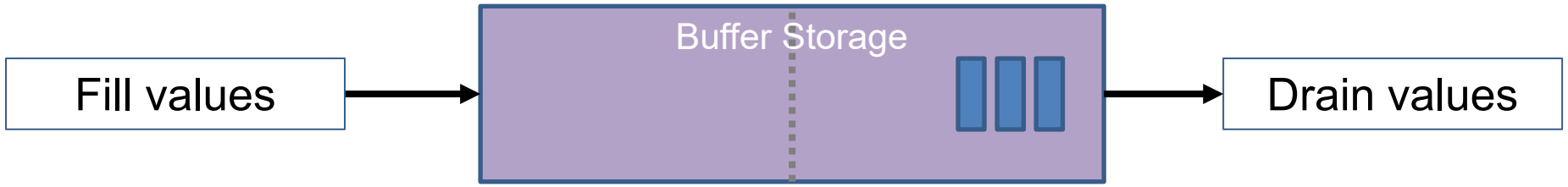


Rolling use

EDDO Strategies

Fill and use:

Double buffer

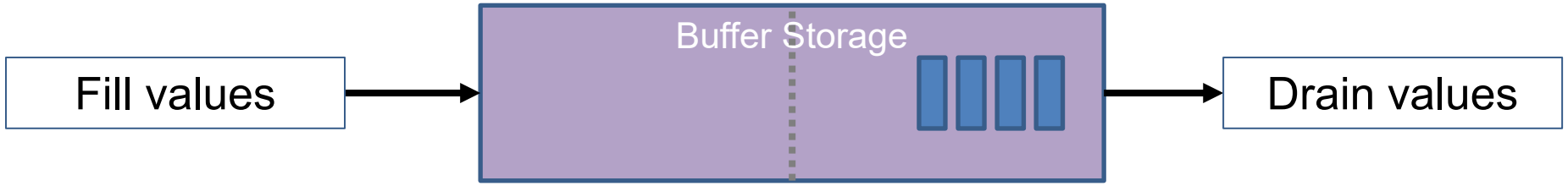


Rolling use

EDDO Strategies

Fill and use:

Double buffer

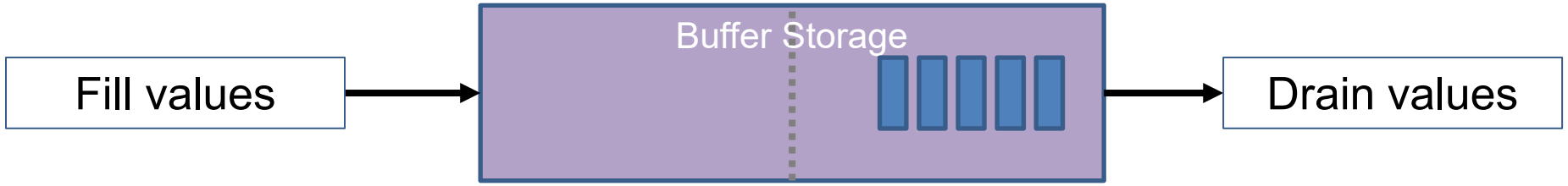


Rolling use

EDDO Strategies

Fill and use:

Double buffer

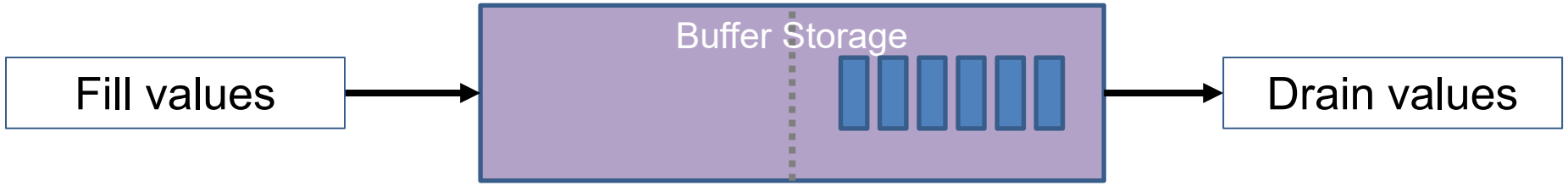


Rolling use

EDDO Strategies

Fill and use:

Double buffer

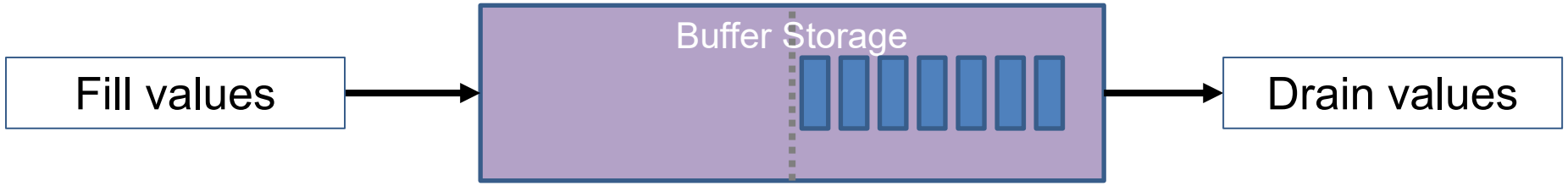


Rolling use

EDDO Strategies

Fill and use:

Double buffer

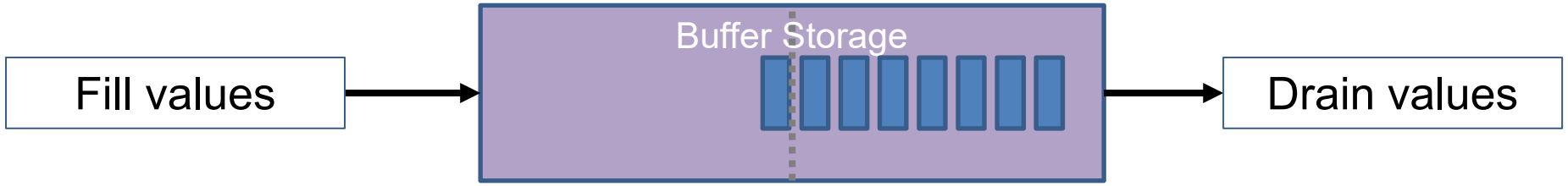


Rolling use

EDDO Strategies

Fill and use:

Double buffer

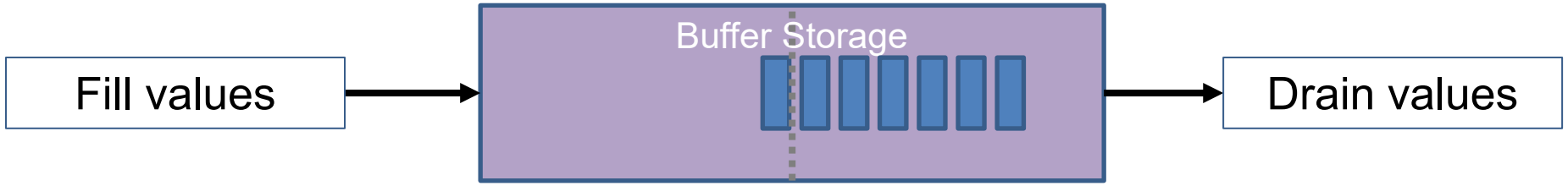


Rolling use

EDDO Strategies

Fill and use:

Double buffer

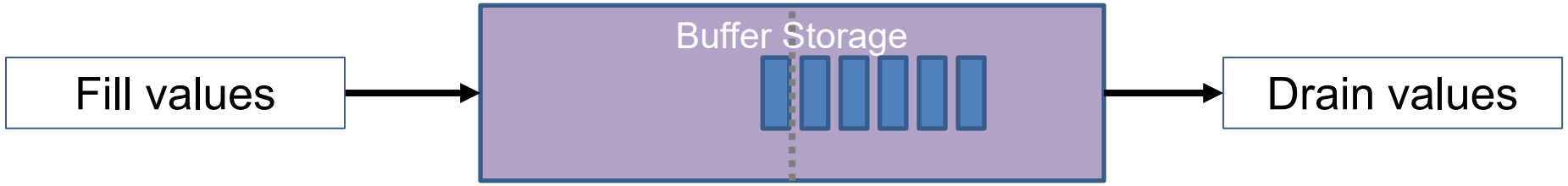


Rolling use

EDDO Strategies

Fill and use:

Double buffer

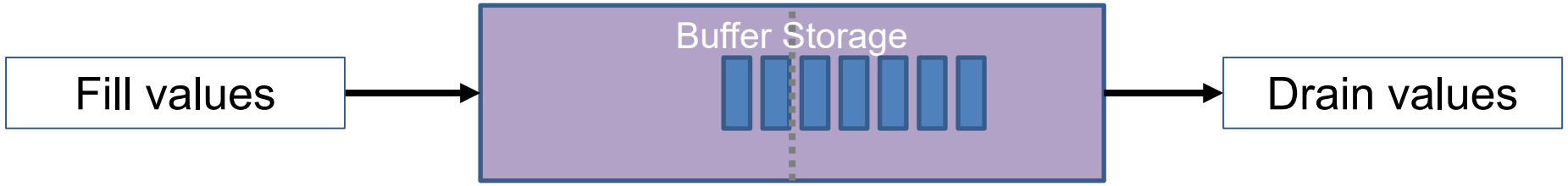


Rolling use

EDDO Strategies

Fill and use:

Double buffer

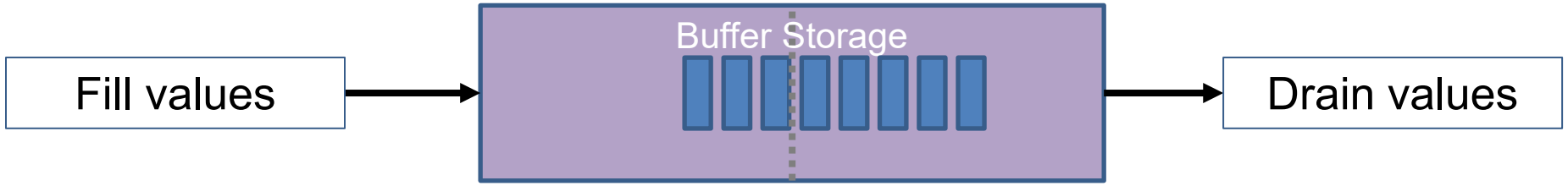


Rolling use

EDDO Strategies

Fill and use:

Double buffer

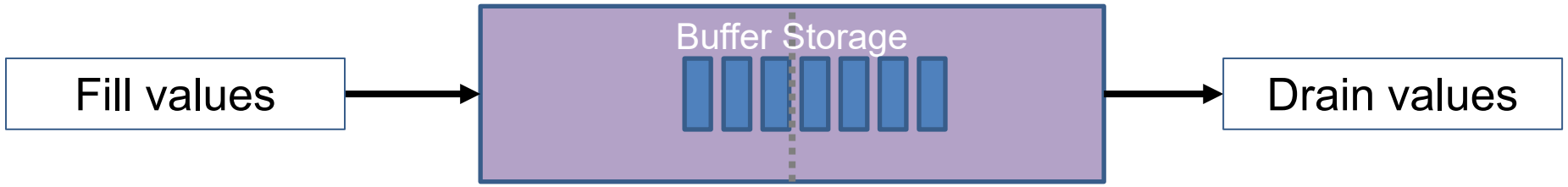


Rolling use

EDDO Strategies

Fill and use:

Double buffer

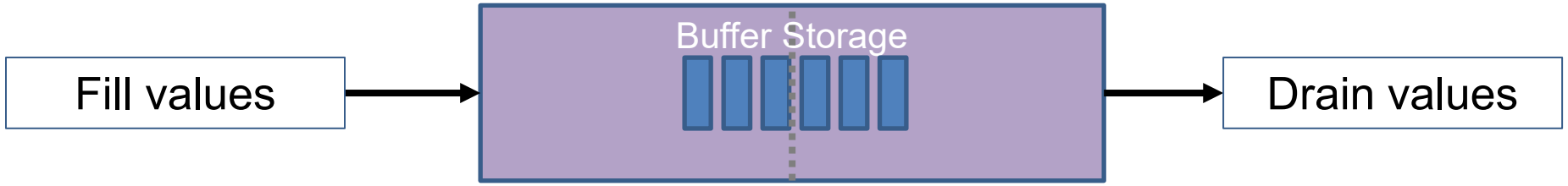


Rolling use

EDDO Strategies

Fill and use:

Double buffer

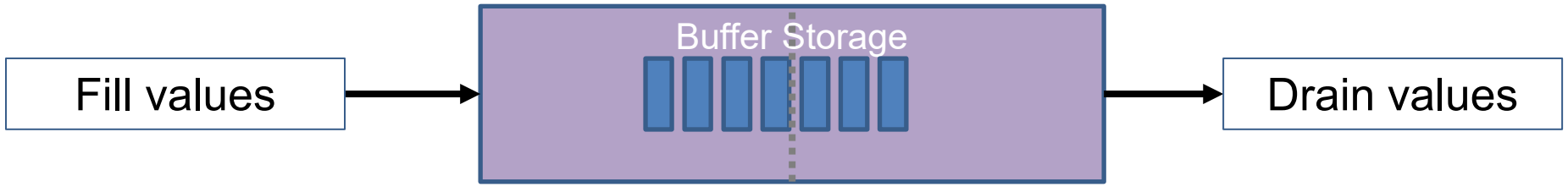


Rolling use

EDDO Strategies

Fill and use:

Double buffer

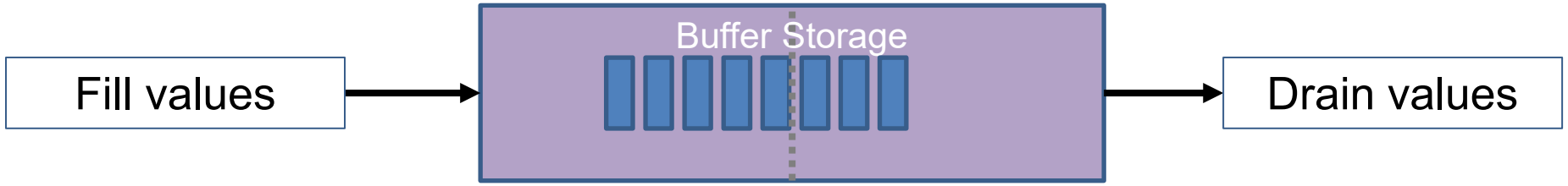


Rolling use

EDDO Strategies

Fill and use:

Double buffer

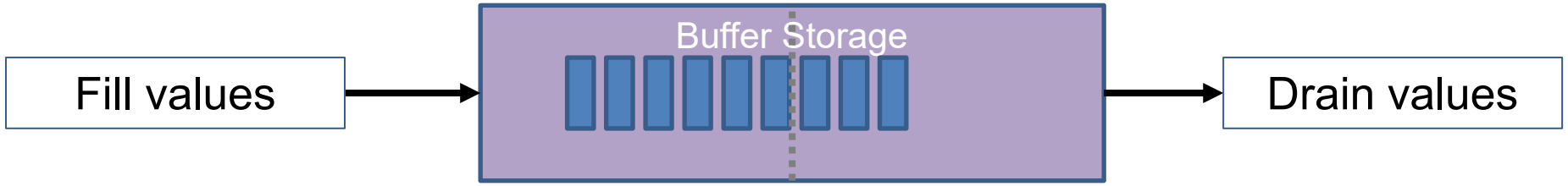


Rolling use

EDDO Strategies

Fill and use:

Double buffer

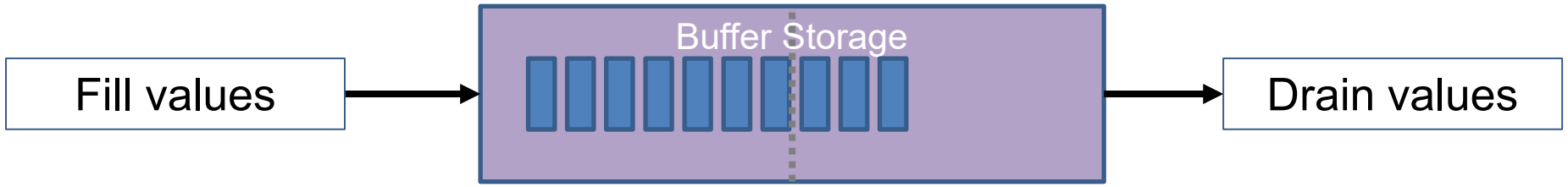


Rolling use

EDDO Strategies

Fill and use:

Double buffer

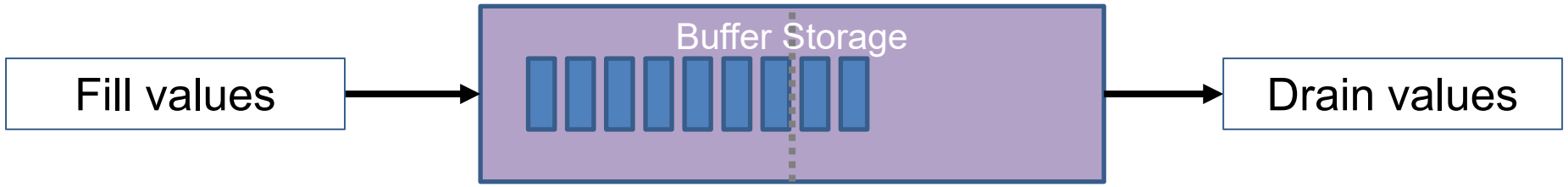


Rolling use

EDDO Strategies

Fill and use:

Double buffer

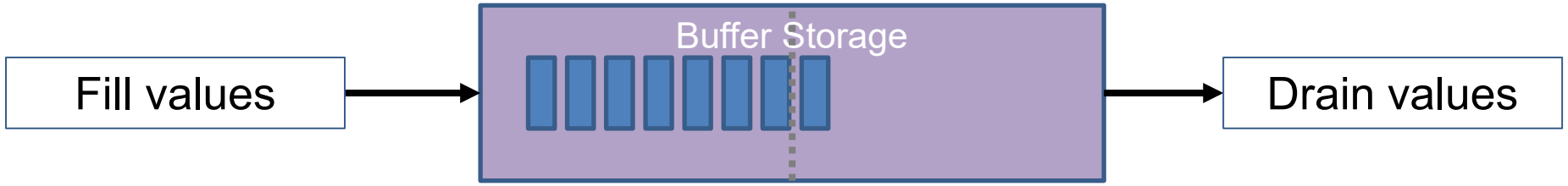


Rolling use

EDDO Strategies

Fill and use:

Double buffer

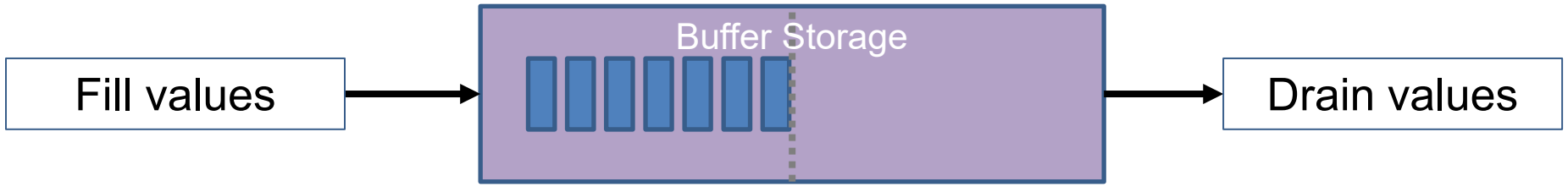


Rolling use

EDDO Strategies

Fill and use:

Double buffer

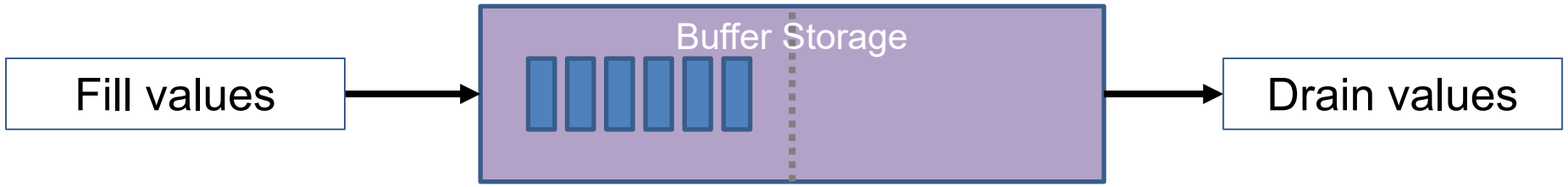


Rolling use

EDDO Strategies

Fill and use:

Double buffer

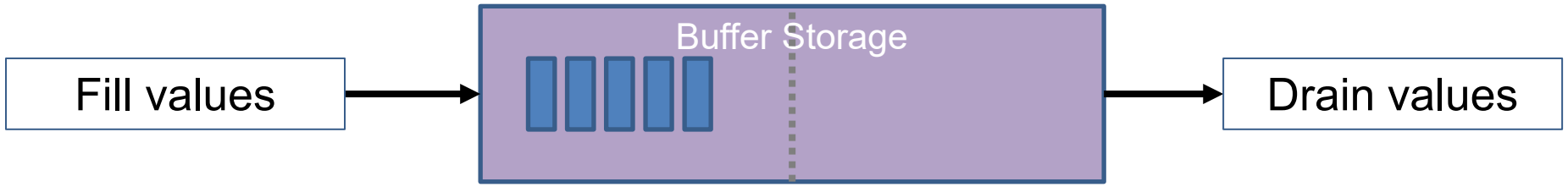


Rolling use

EDDO Strategies

Fill and use:

Double buffer

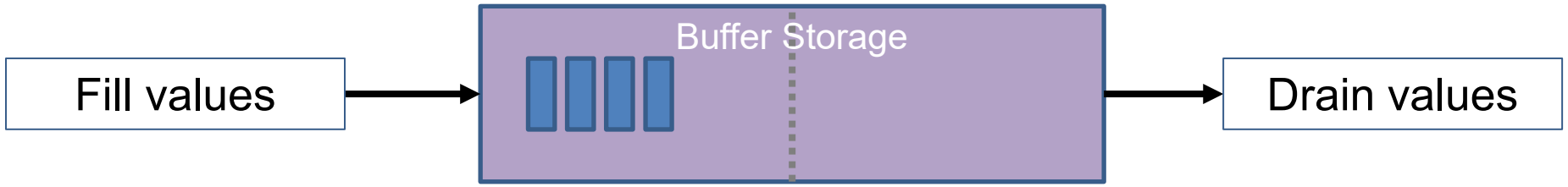


Rolling use

EDDO Strategies

Fill and use:

Double buffer

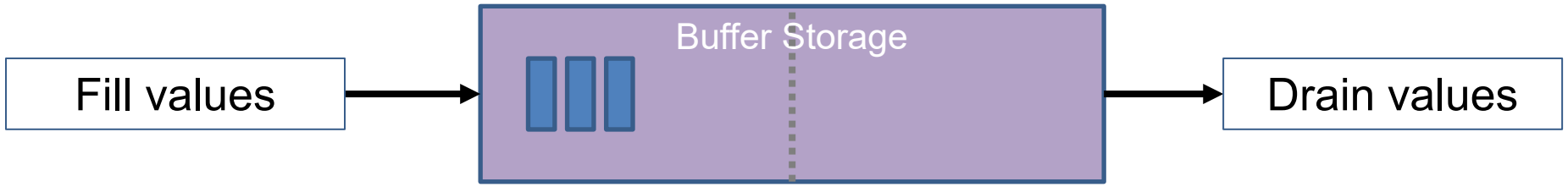


Rolling use

EDDO Strategies

Fill and use:

Double buffer

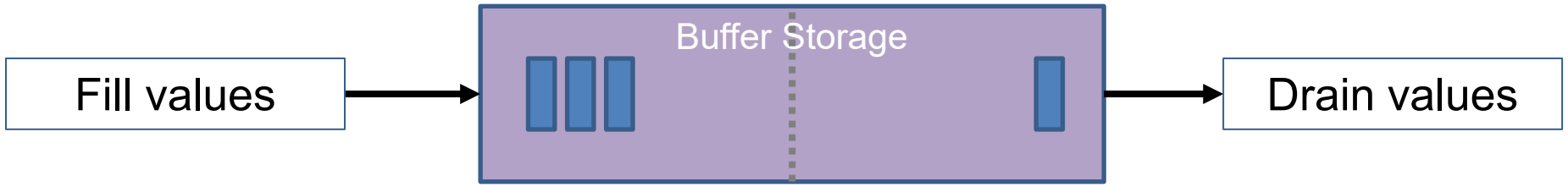


Rolling use

EDDO Strategies

Fill and use:

Double buffer

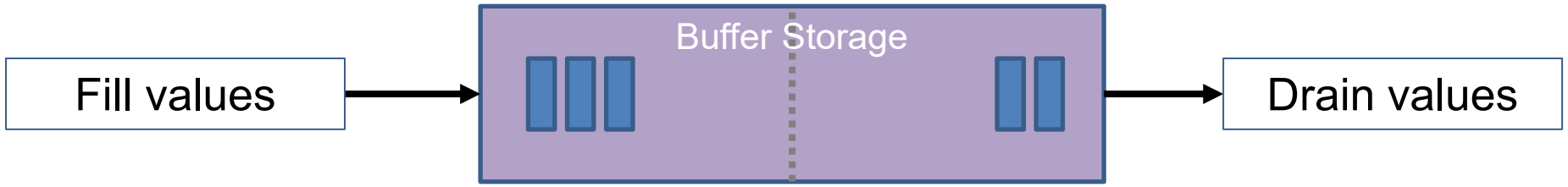


Rolling use

EDDO Strategies

Fill and use:

Double buffer

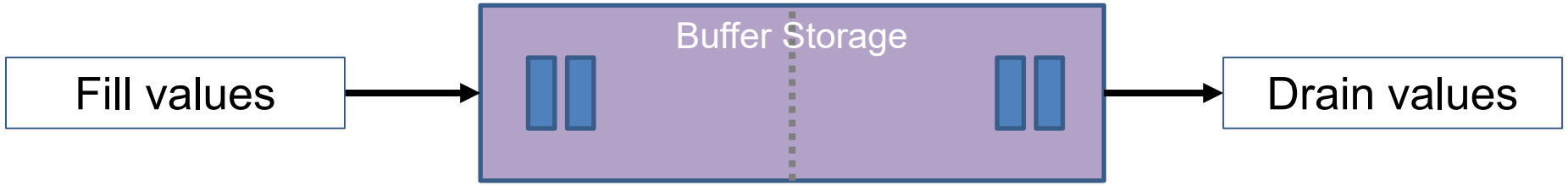


Rolling use

EDDO Strategies

Fill and use:

Double buffer

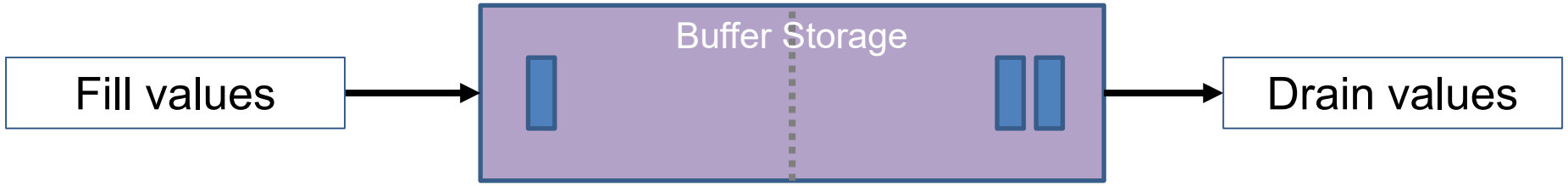


Rolling use

EDDO Strategies

Fill and use:

Double buffer

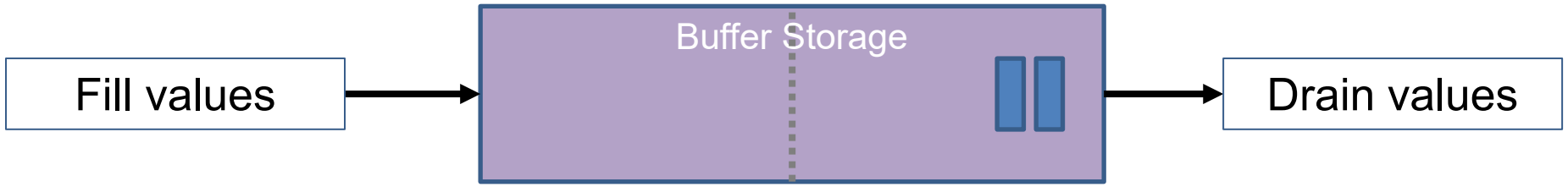


Rolling use

EDDO Strategies

Fill and use:

Double buffer

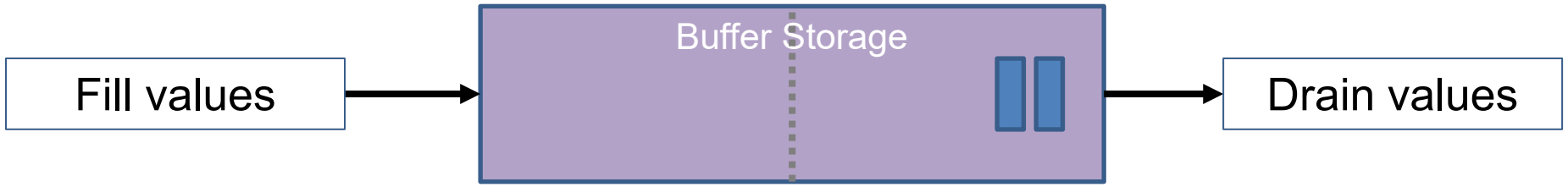


Rolling use

EDDO Strategies

Fill and use:

Double buffer

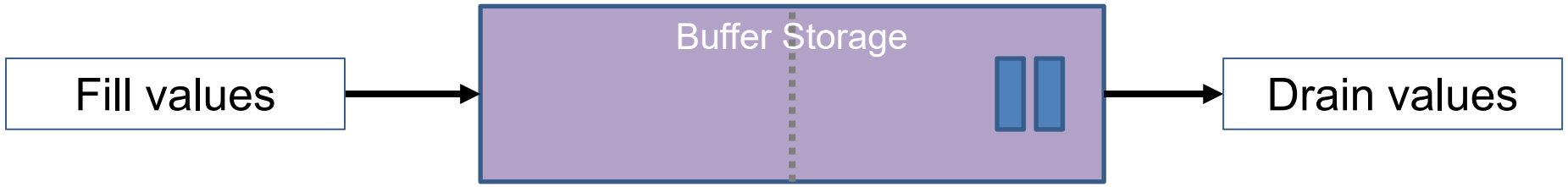


Rolling use

EDDO Strategies

Fill and use:

Double buffer

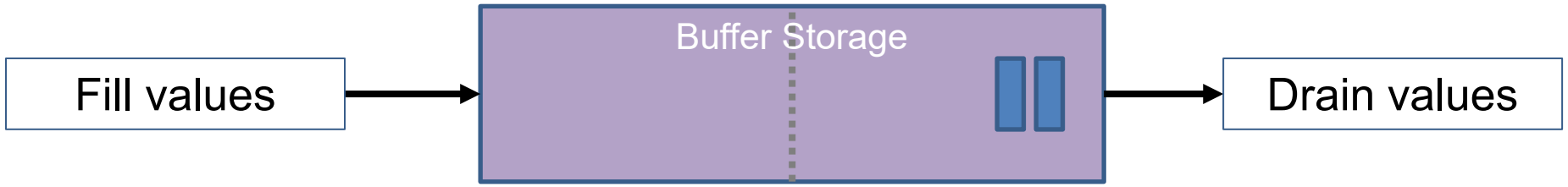


Rolling use

EDDO Strategies

Fill and use:

Double buffer

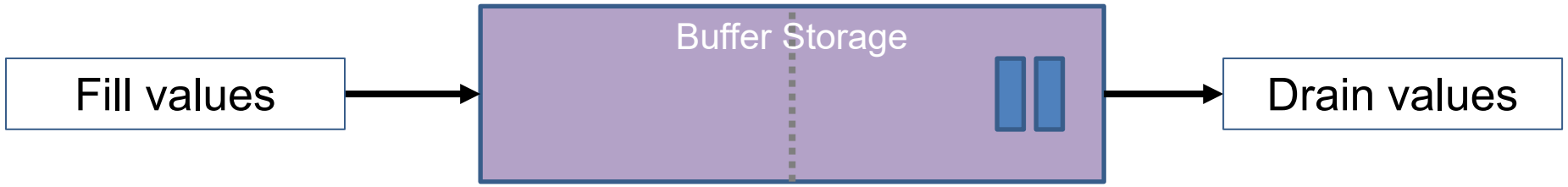


Rolling use

EDDO Strategies

Fill and use:

Double buffer

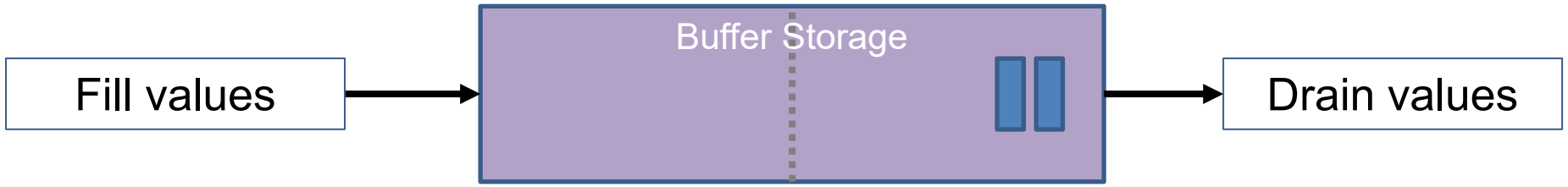


Rolling use

EDDO Strategies

Fill and use:

Double buffer

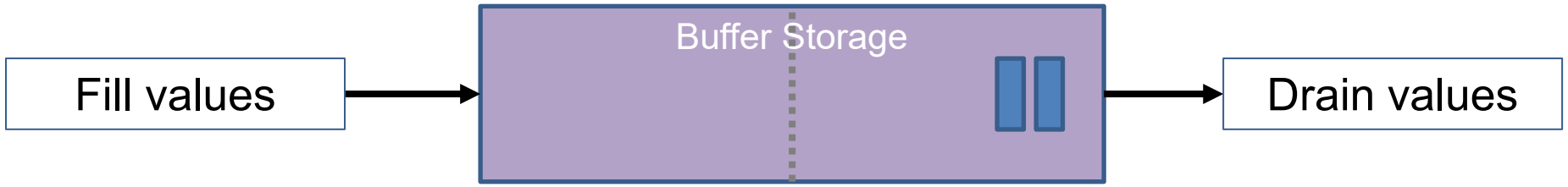


Rolling use

EDDO Strategies

Fill and use:

Double buffer

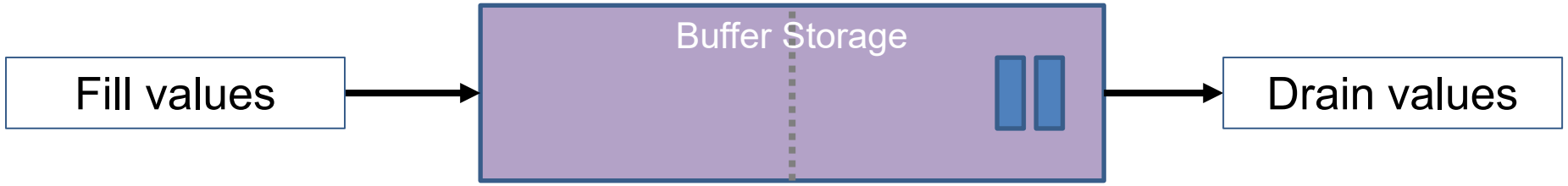


Rolling use

EDDO Strategies

Fill and use:

Double buffer

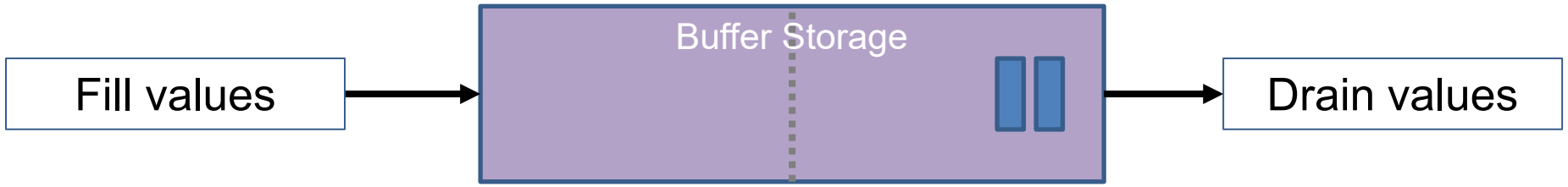


Rolling use

EDDO Strategies

Fill and use:

Double buffer

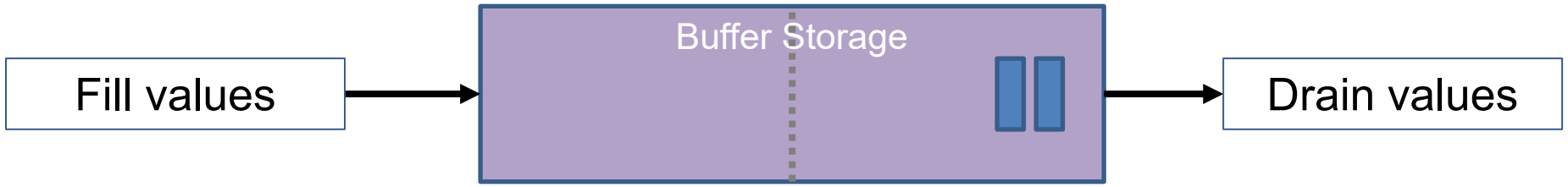


Rolling use

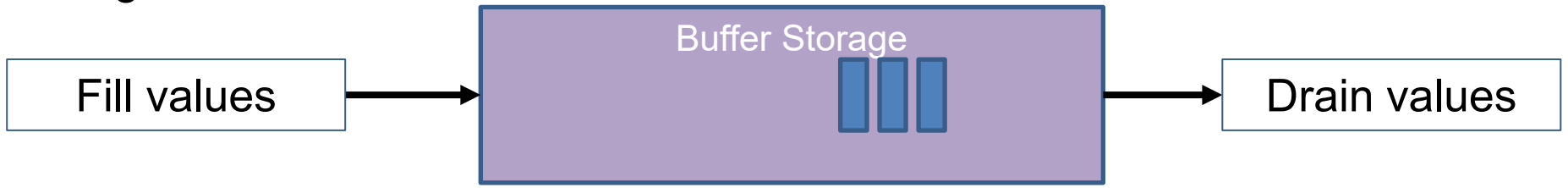
EDDO Strategies

Fill and use:

Double buffer



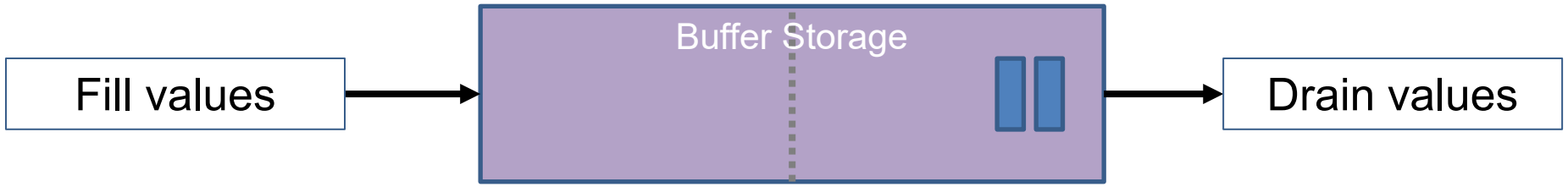
Rolling use



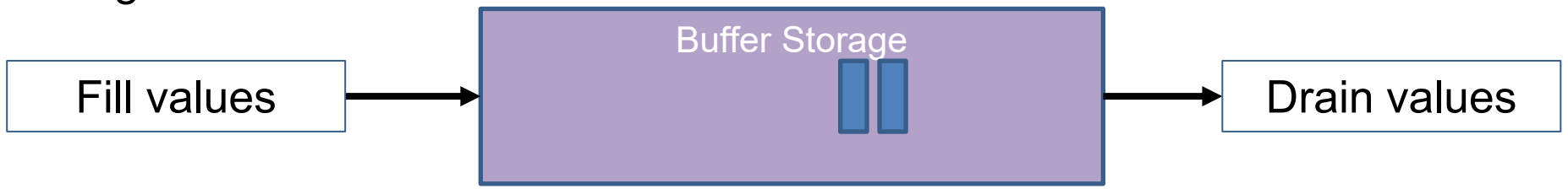
EDDO Strategies

Fill and use:

Double buffer



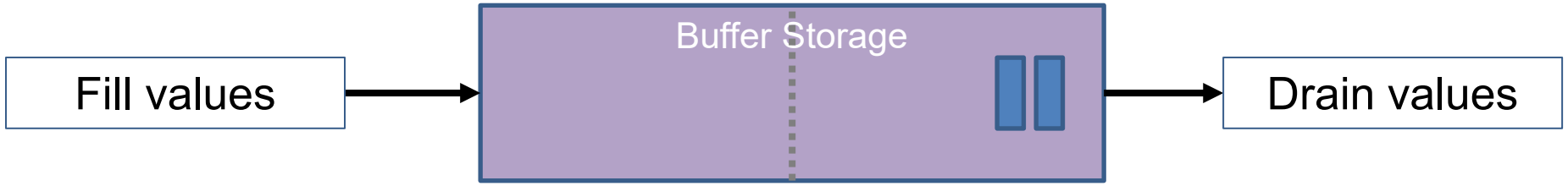
Rolling use



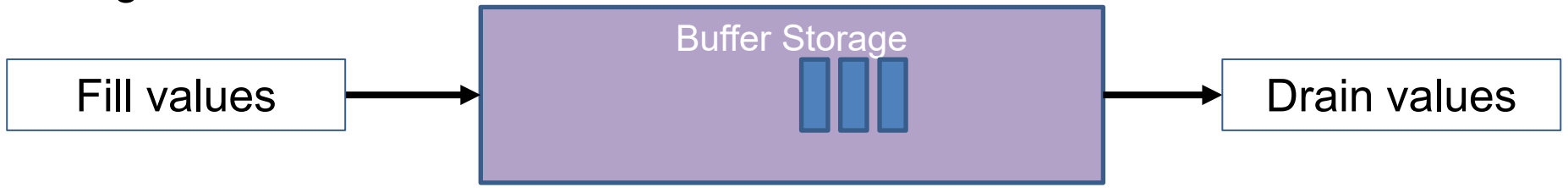
EDDO Strategies

Fill and use:

Double buffer



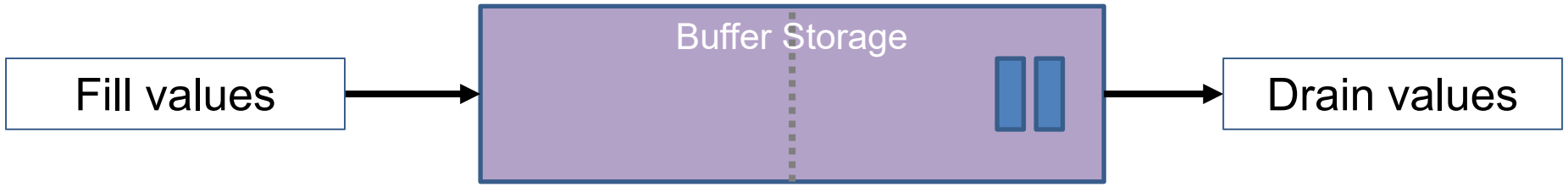
Rolling use



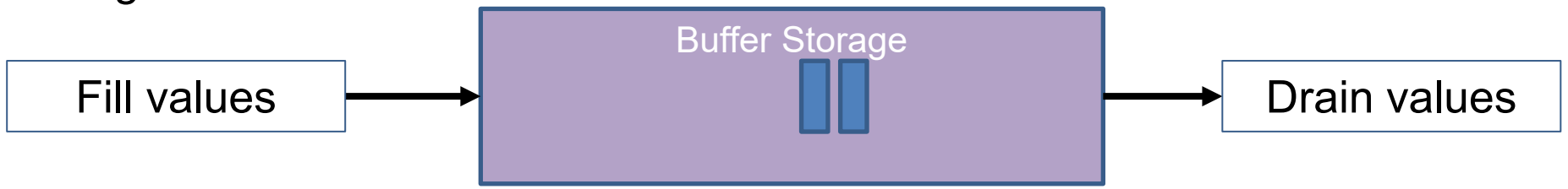
EDDO Strategies

Fill and use:

Double buffer



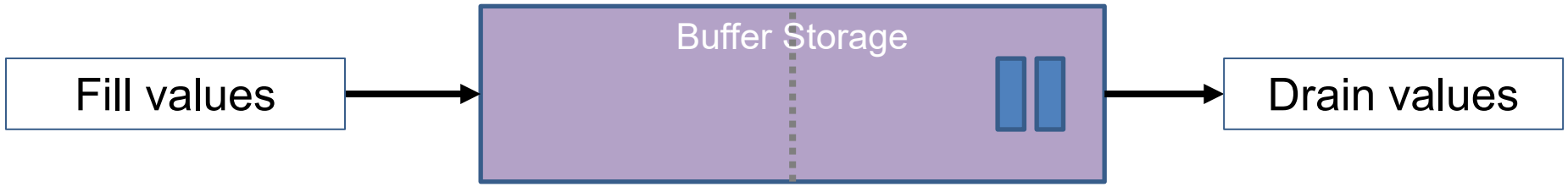
Rolling use



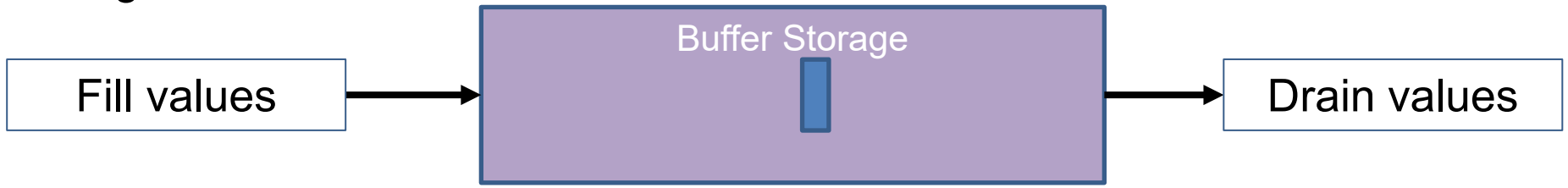
EDDO Strategies

Fill and use:

Double buffer



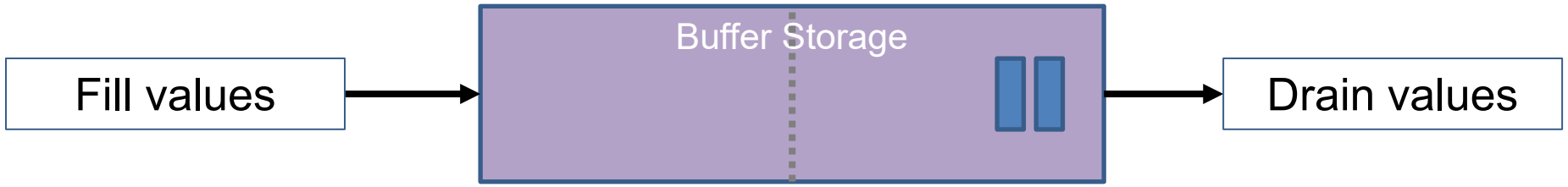
Rolling use



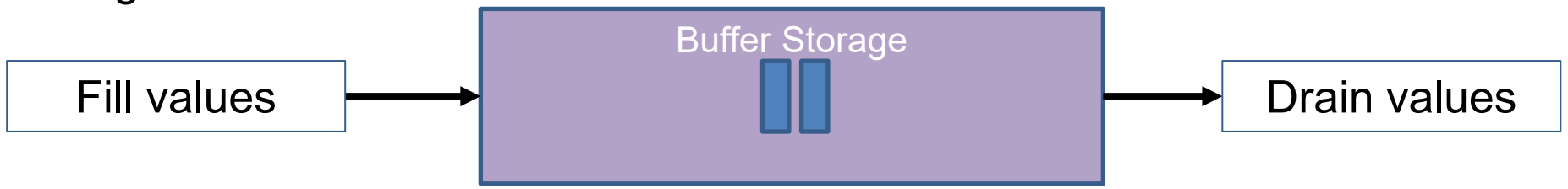
EDDO Strategies

Fill and use:

Double buffer



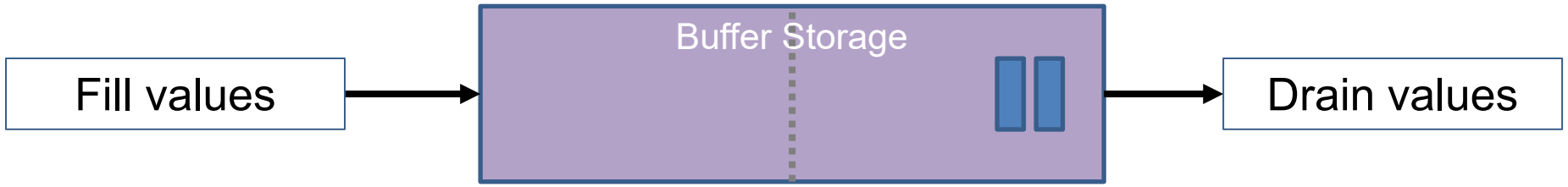
Rolling use



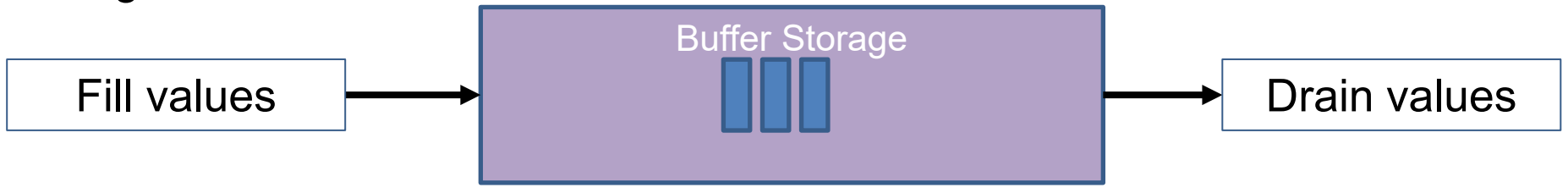
EDDO Strategies

Fill and use:

Double buffer



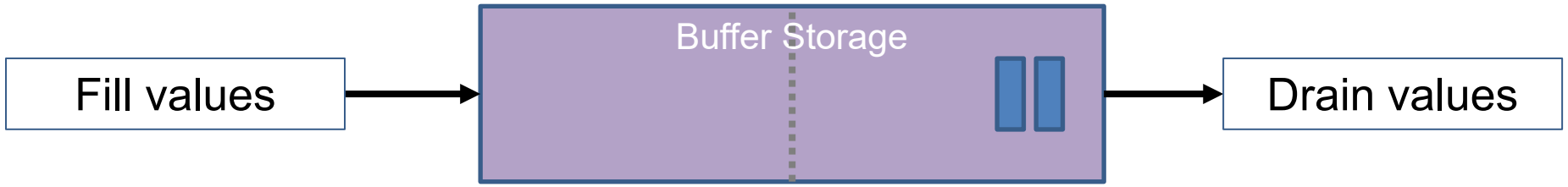
Rolling use



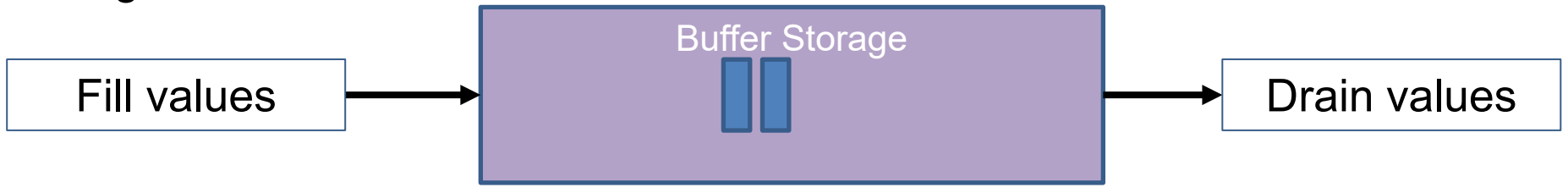
EDDO Strategies

Fill and use:

Double buffer



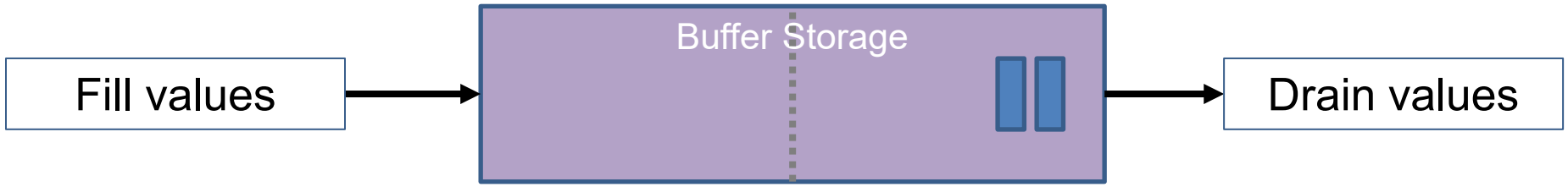
Rolling use



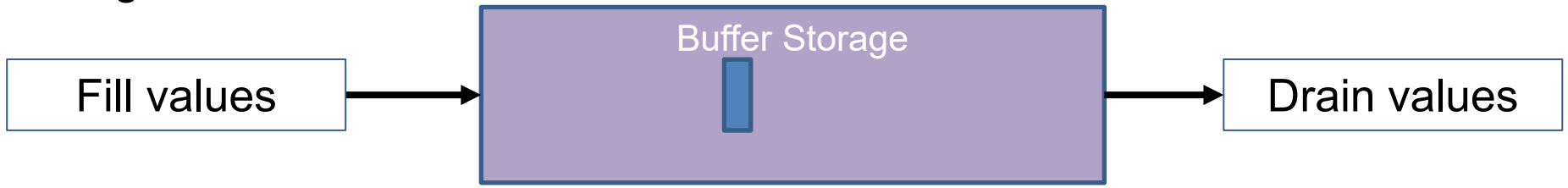
EDDO Strategies

Fill and use:

Double buffer



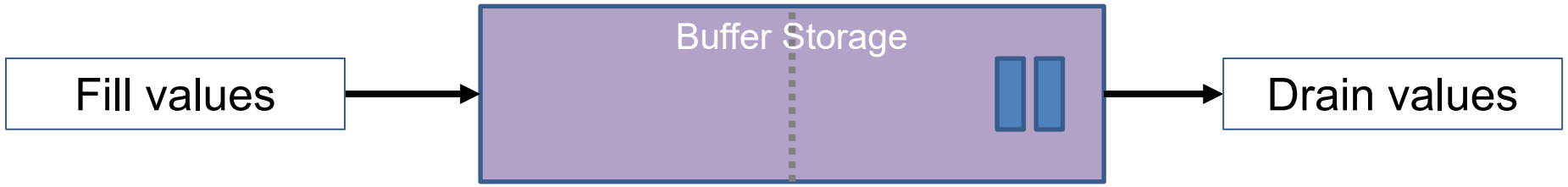
Rolling use



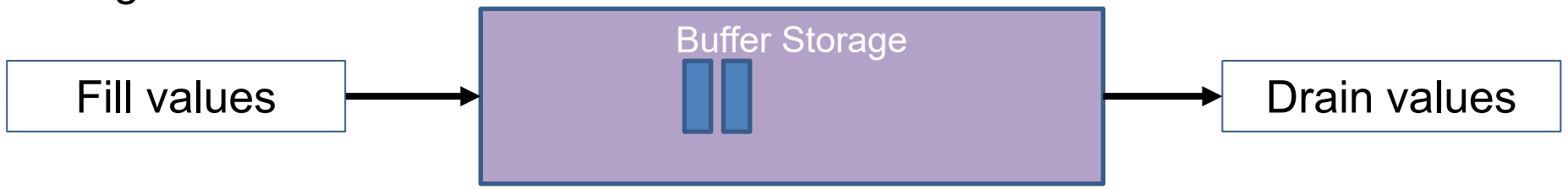
EDDO Strategies

Fill and use:

Double buffer



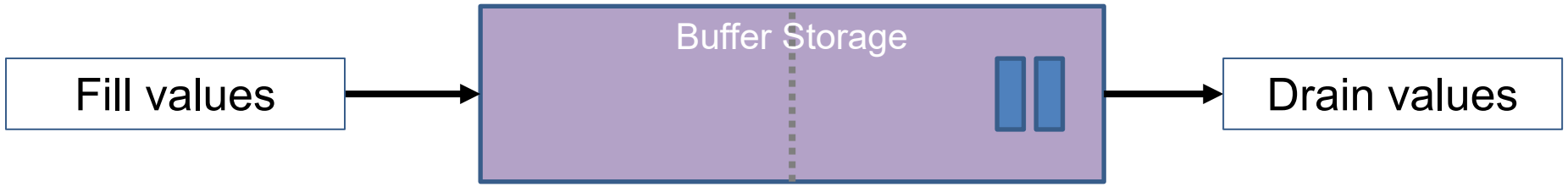
Rolling use



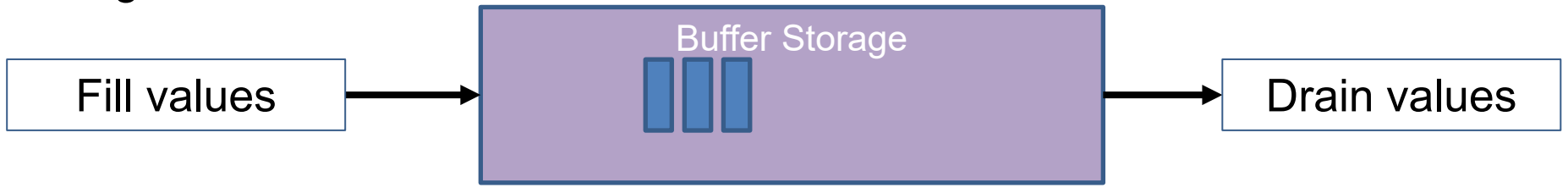
EDDO Strategies

Fill and use:

Double buffer



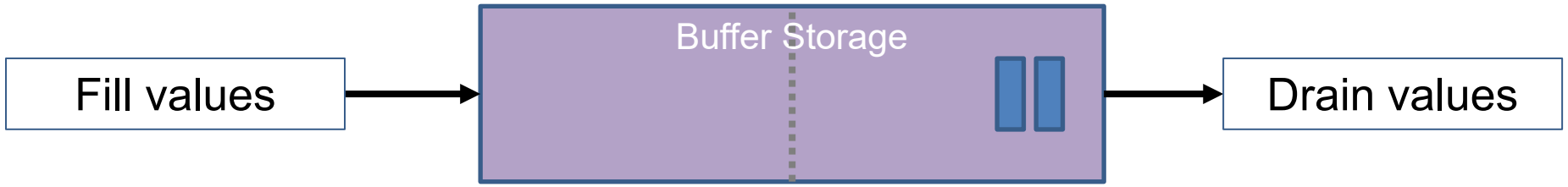
Rolling use



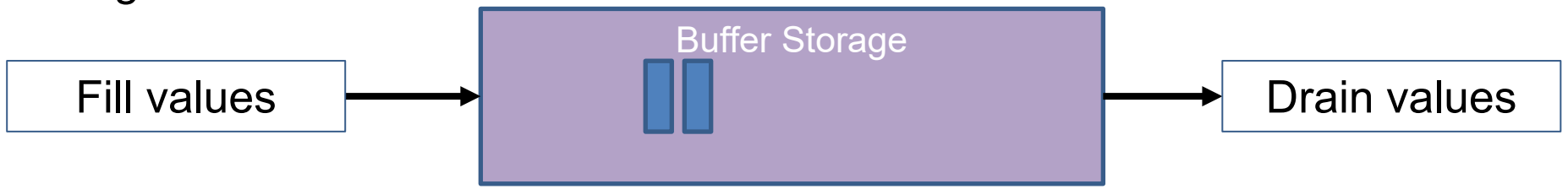
EDDO Strategies

Fill and use:

Double buffer



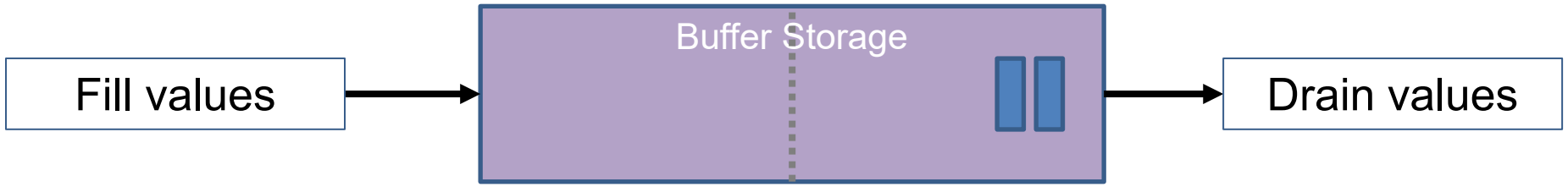
Rolling use



EDDO Strategies

Fill and use:

Double buffer

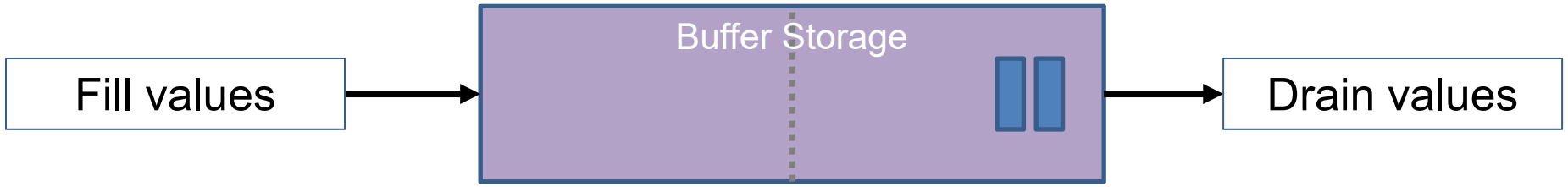


Rolling use

EDDO Strategies

Fill and use:

Double buffer

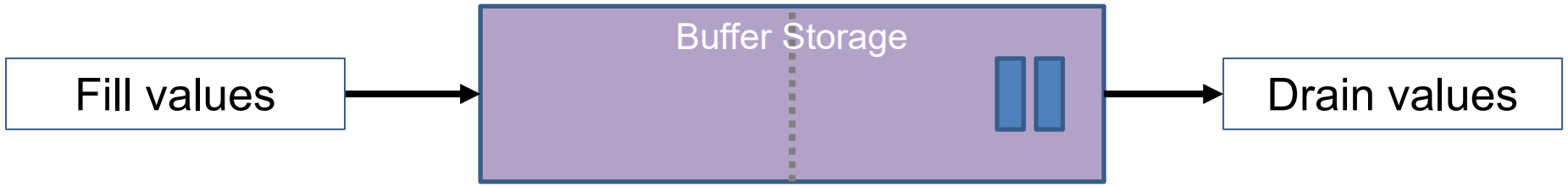


Rolling use

EDDO Strategies

Fill and use:

Double buffer

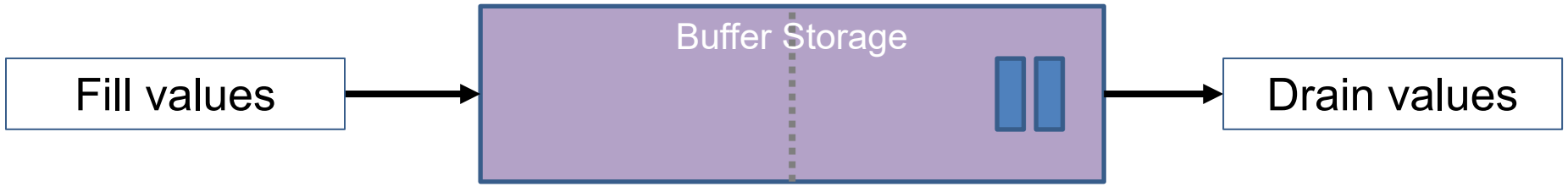


Rolling use

EDDO Strategies

Fill and use:

Double buffer

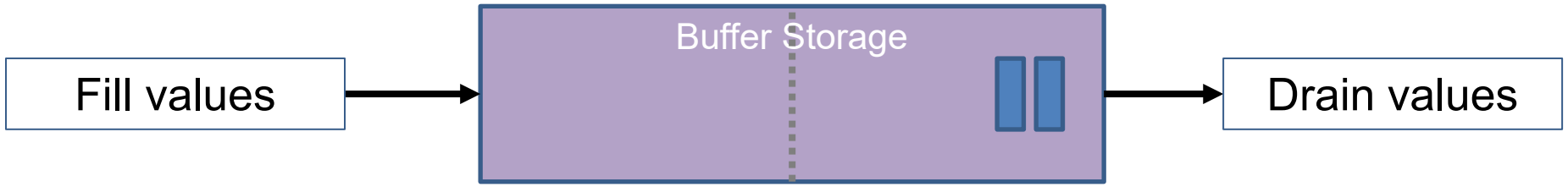


Rolling use

EDDO Strategies

Fill and use:

Double buffer

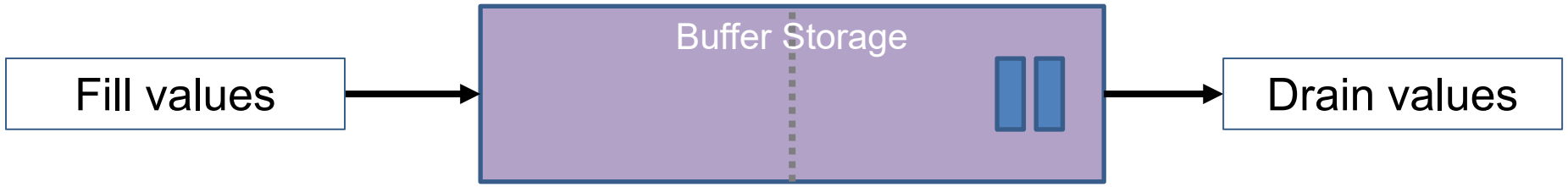


Rolling use

EDDO Strategies

Fill and use:

Double buffer

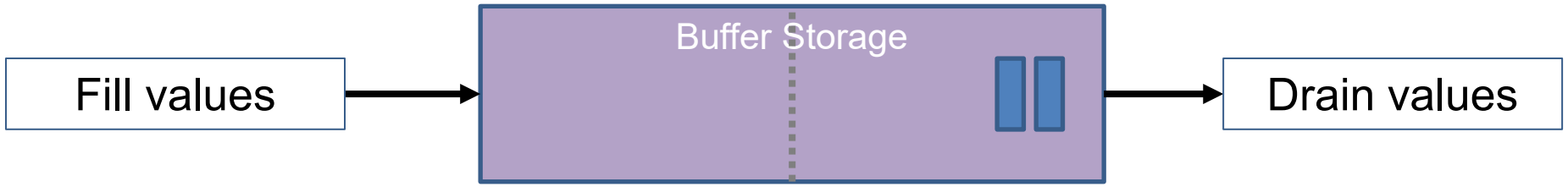


Rolling use

EDDO Strategies

Fill and use:

Double buffer

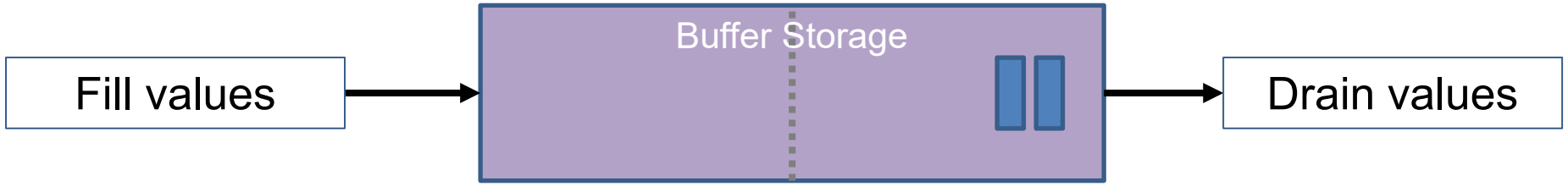


Rolling use

EDDO Strategies

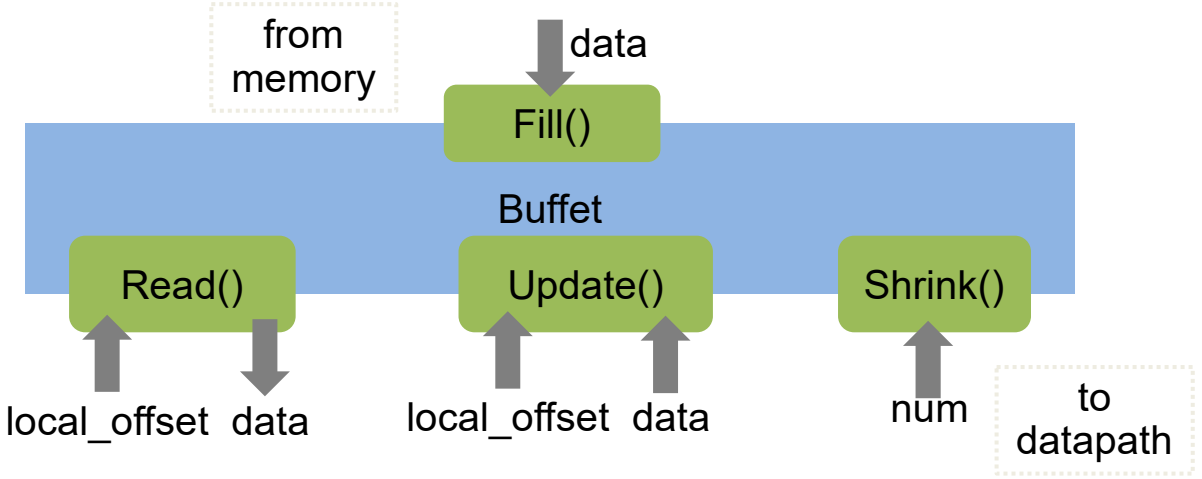
Fill and use:

Double buffer



Rolling use

Buffets



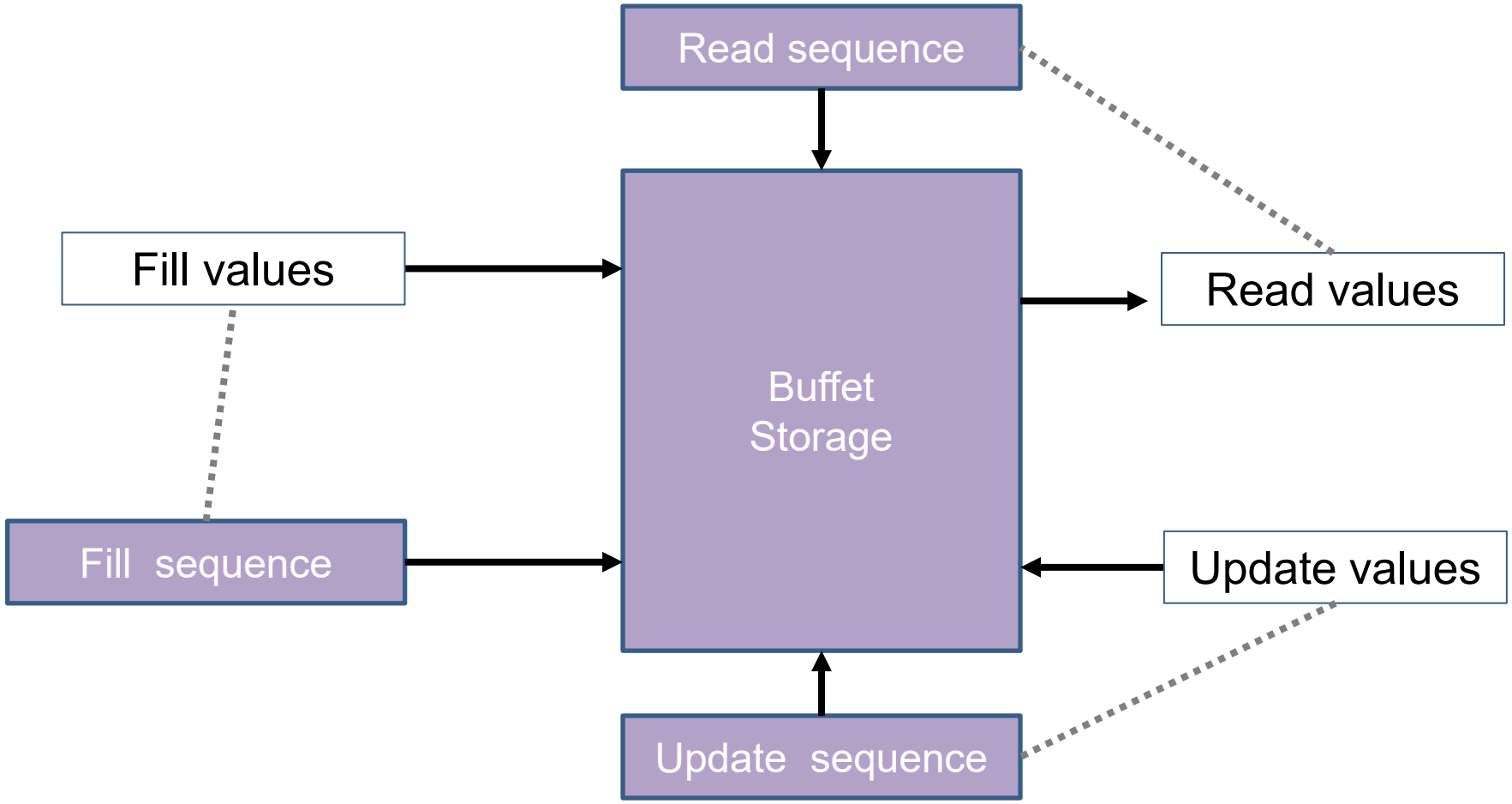
	Buffet
Hierarchically Composable	Yes
Fill/Access Sync.	Fine-grained
Operations	`Fill(data)` `Read(local_offset)` `Update(local_offset, data)` `Shrink(num)`
In-Place Updates	Yes
Hardware Complexity	RAM, head/tail sync logic, RAW hazard scoreboarding

- Compared to FIFO
  - Allows random access into live window
  - Allows updates of values in live window
- Compared to scratchpad:
  - Adds scoreboarding for synchronization
  - Allows arbitrary degrees of buffering
- Compared to cache
  - Addresses local (therefore fewer bits) and no tag store
  - Push model versus pull model for smaller landing zone
  - Raises level of abstraction beyond single value transfers

$$Fill \rightarrow (Read \rightarrow Update?)^* \rightarrow Read \rightarrow Shrink$$

Can be specialized, for example: “read-only” buffets that have fills but not updates

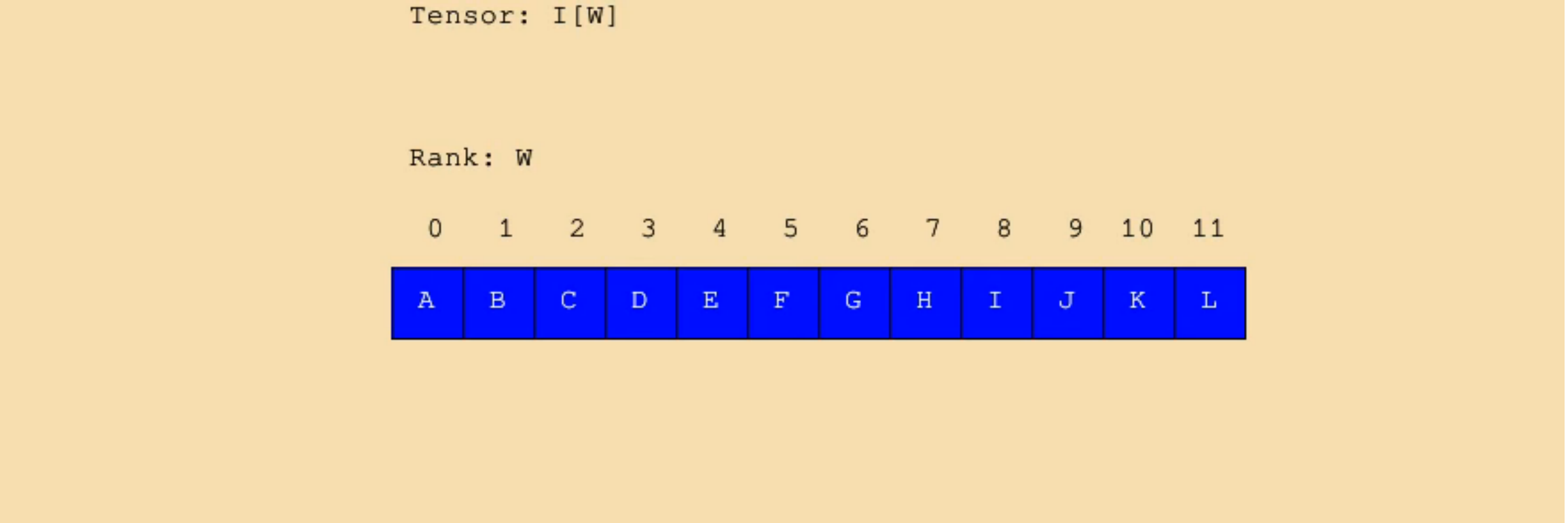
Buffet Usage Model



Buffet Behavioral Attributes

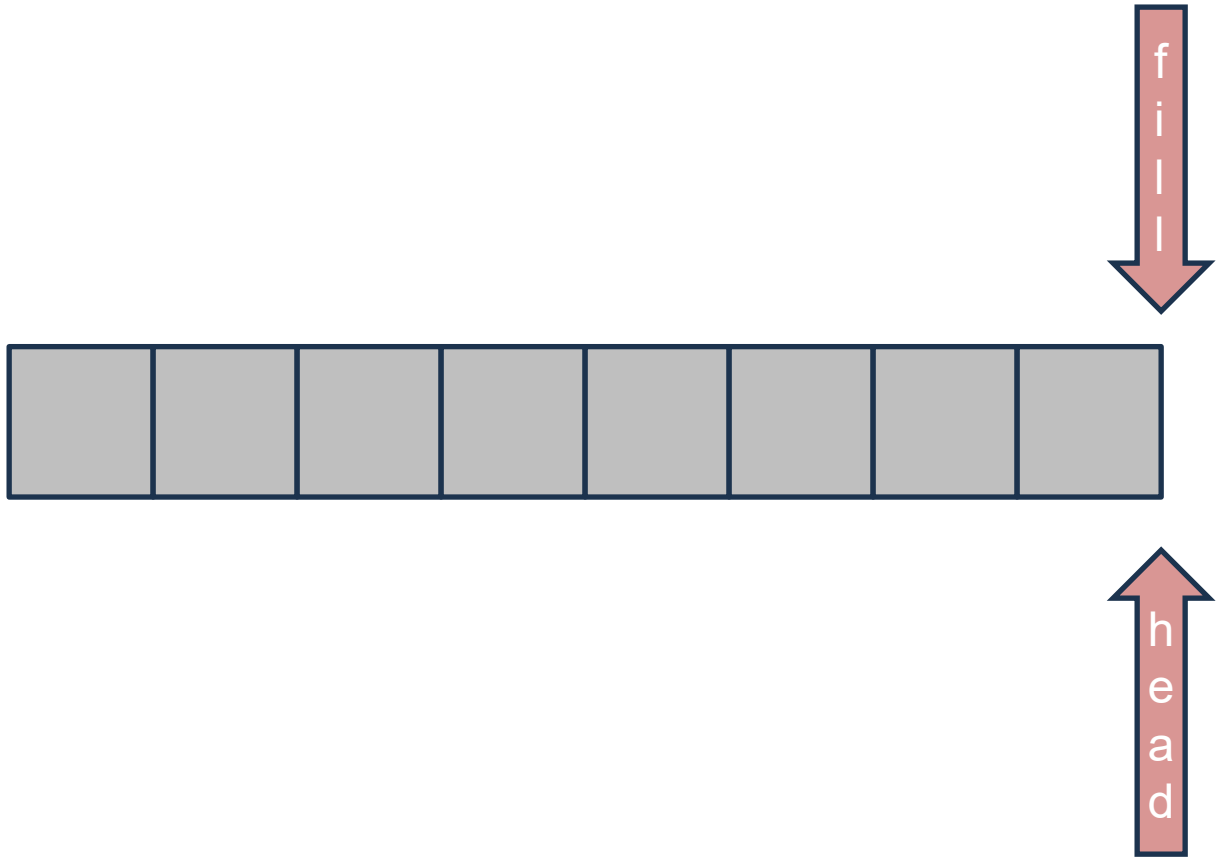
- Based on ‘fill’ address sequence, the buffet will pop values from ‘fill’ channel until buffer is full.
- Based on ‘read’ address sequence the buffet will try to **push** values down ‘read’ LI channel, but only if the value has been ‘filled’.
- ‘Read’ address sequence can also inform buffet that a value can be **dropped**, i.e., space freed. This is routed to the **shrink** control port.
- Based on ‘update’ address sequence the buffet will try to pop values from the ‘update’ channel
- Implementation may include multiple logical buffers inside a single physical buffer.

Sliding Window – 4



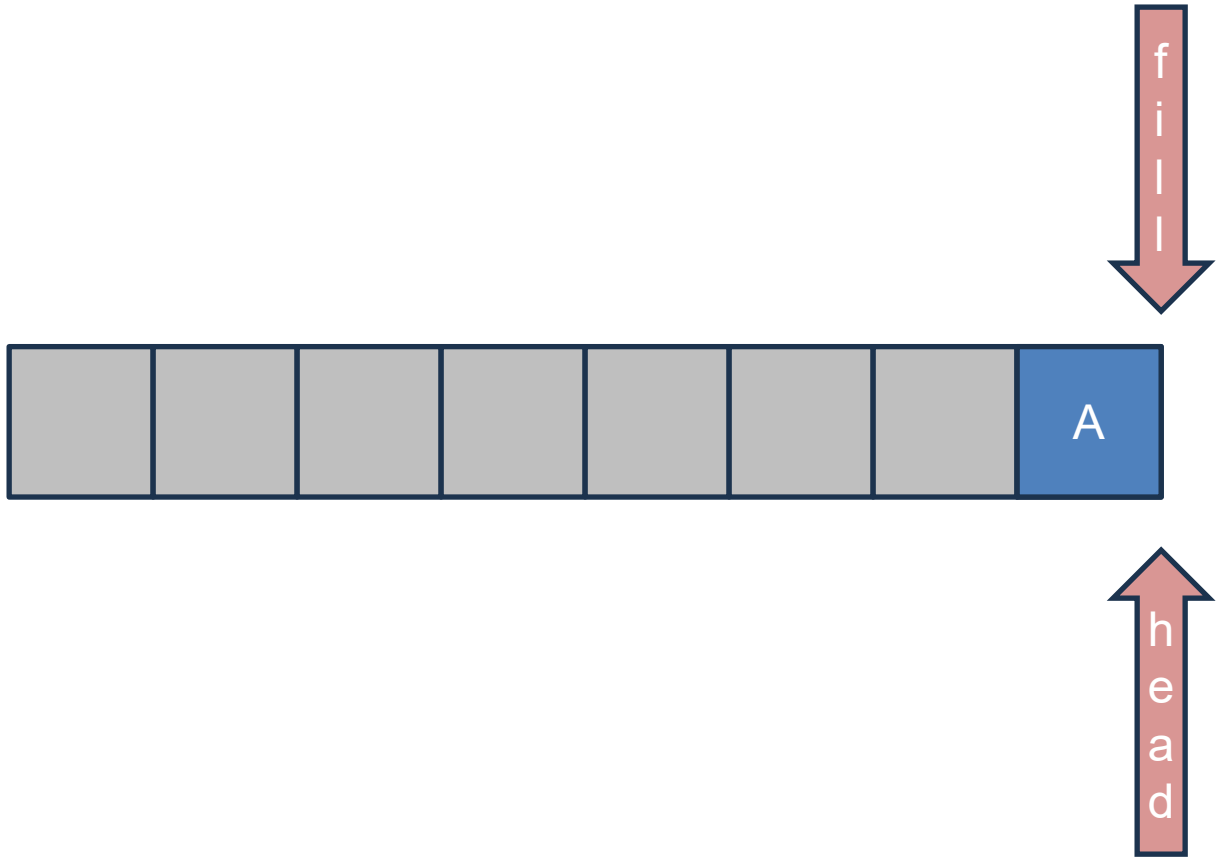
Buffet Control Example

Four-wide sliding window



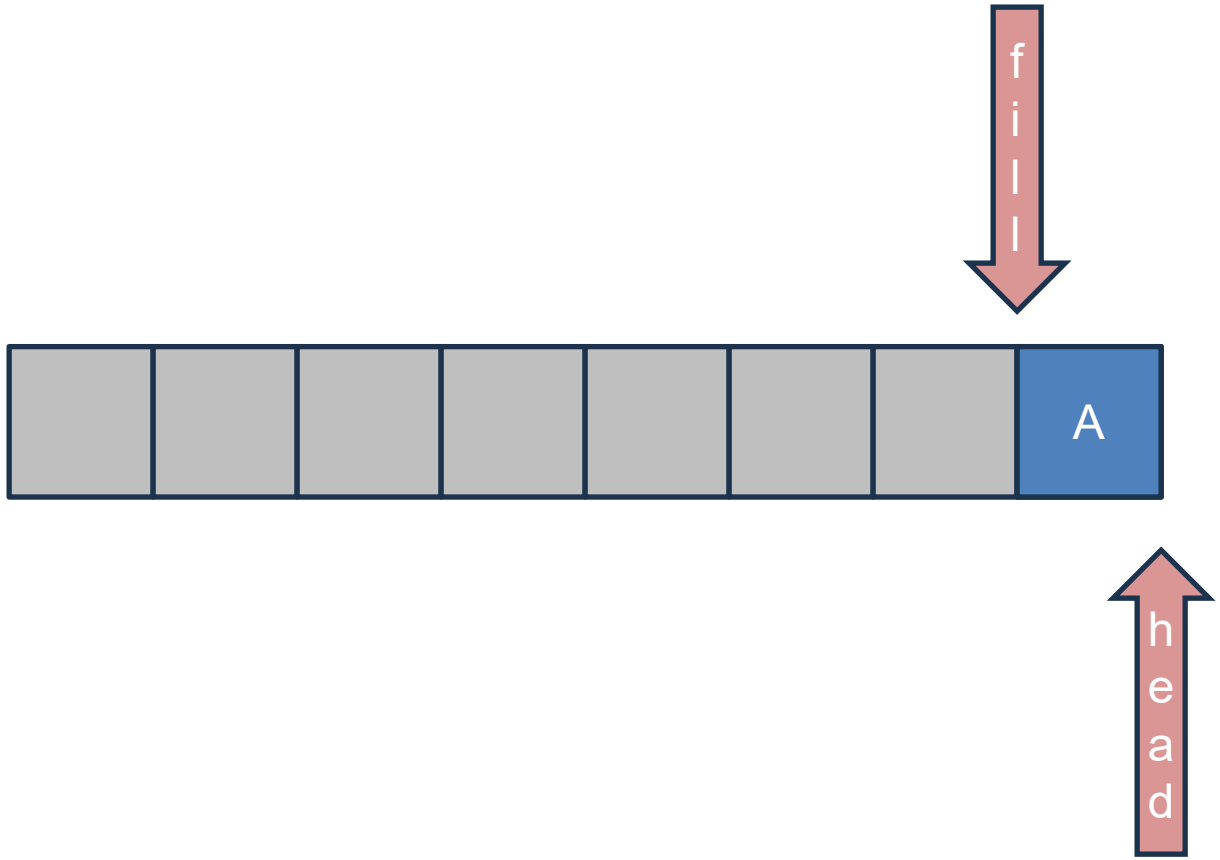
Buffet Control Example

Four-wide sliding window



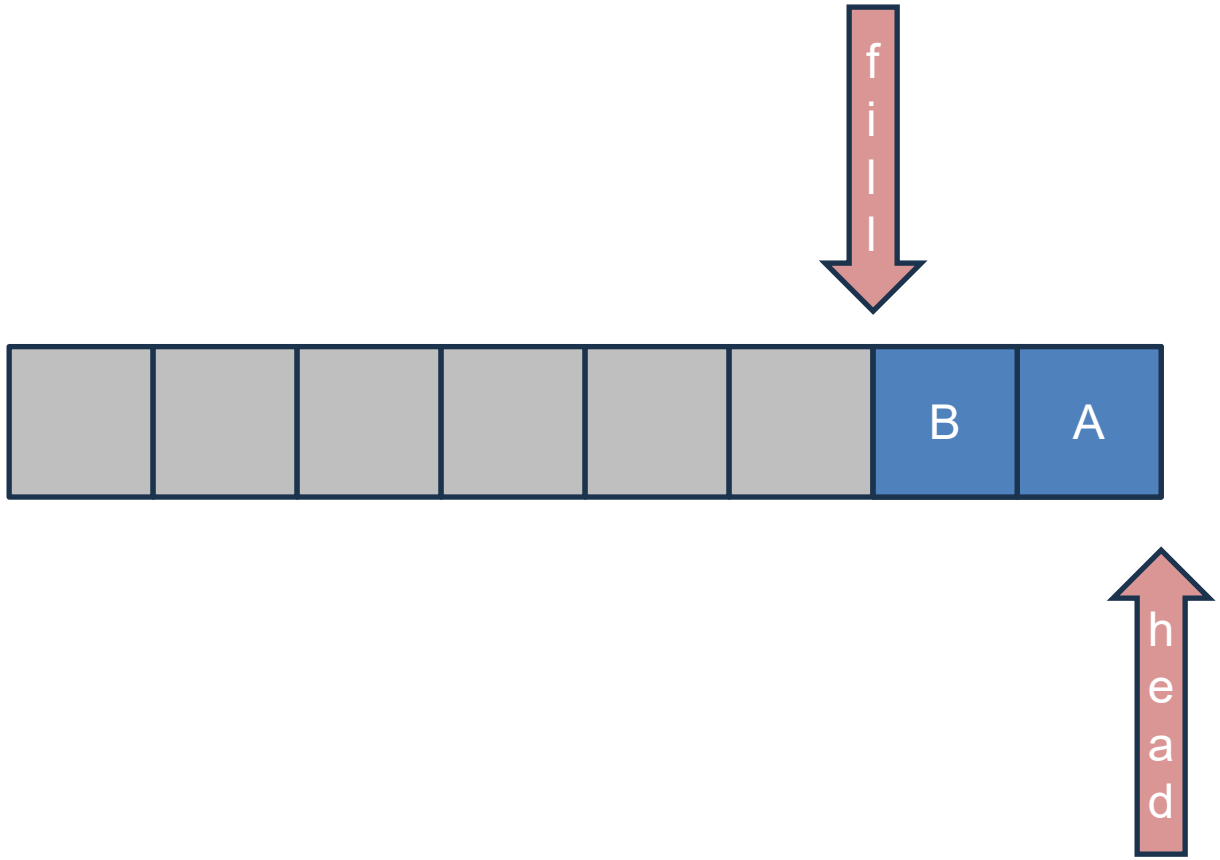
Buffet Control Example

Four-wide sliding window



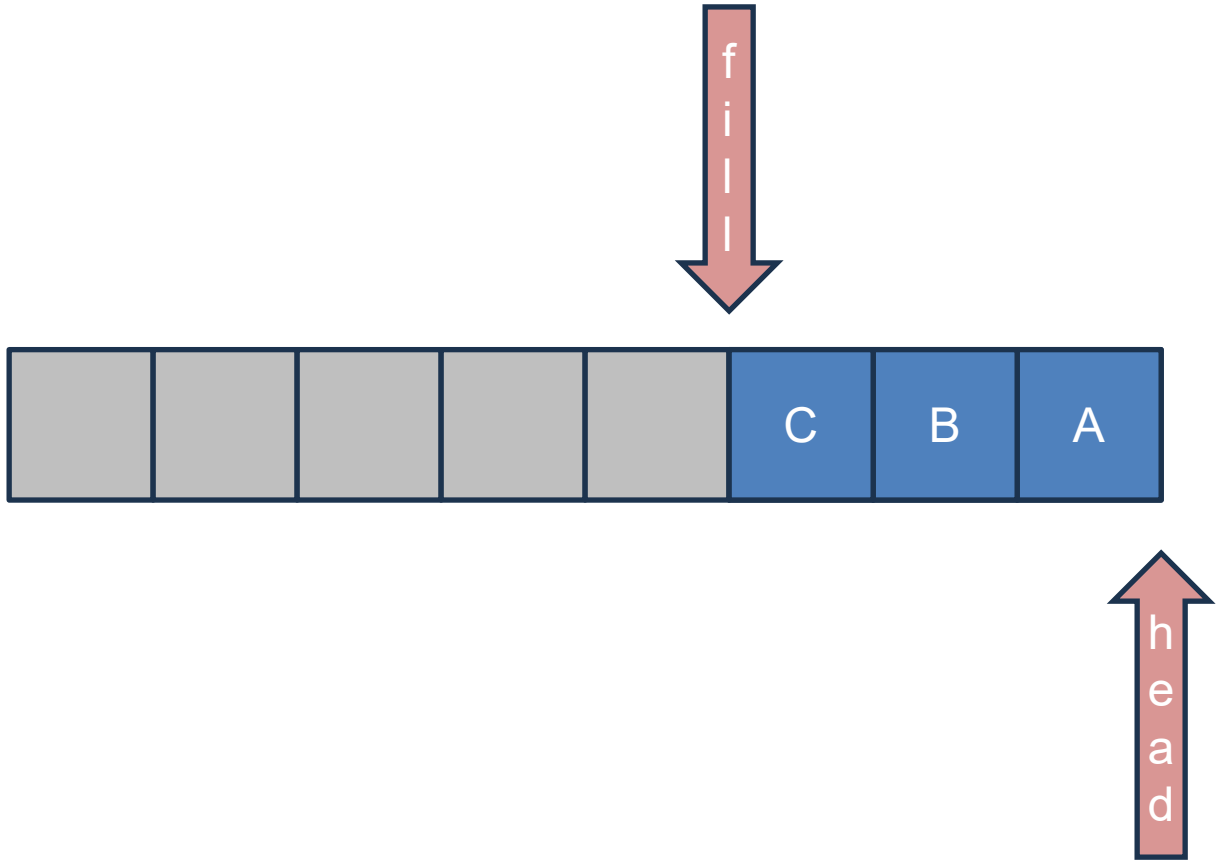
Buffet Control Example

Four-wide sliding window



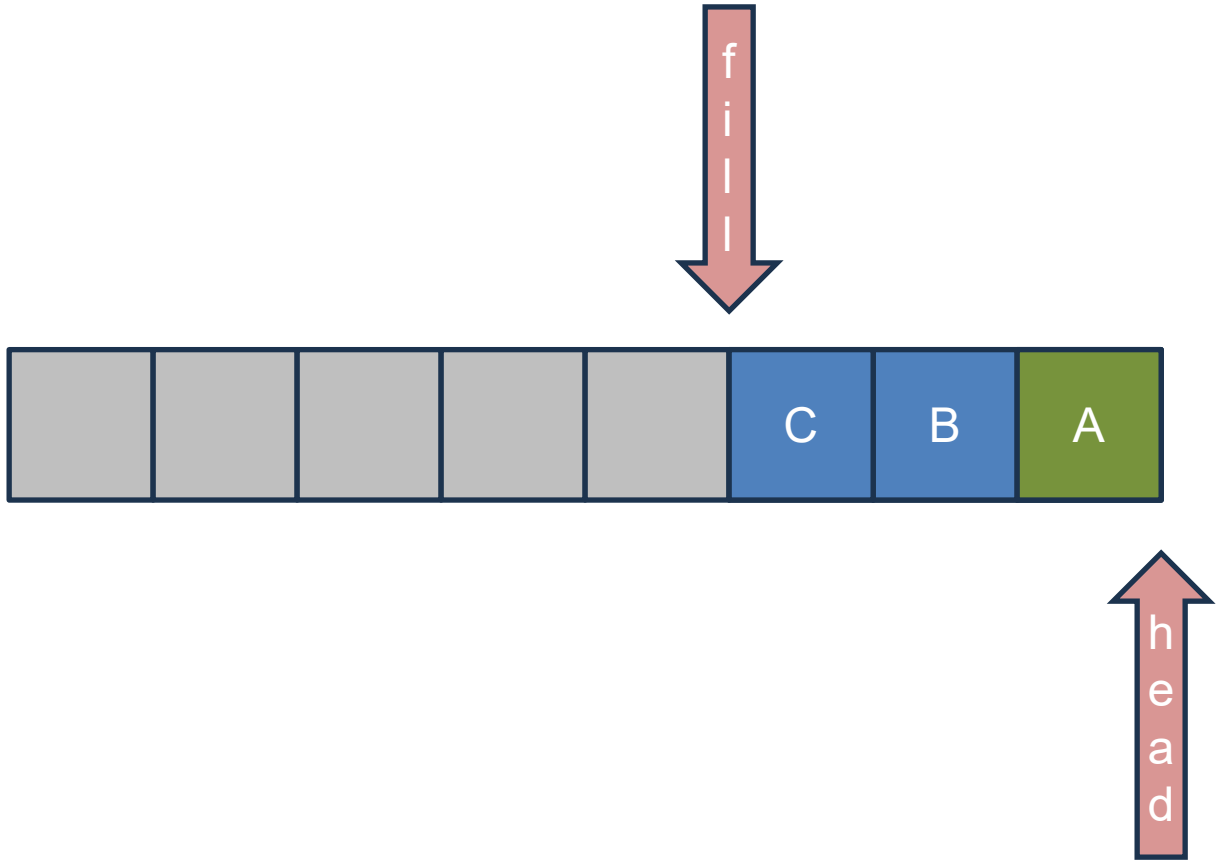
Buffet Control Example

Four-wide sliding window



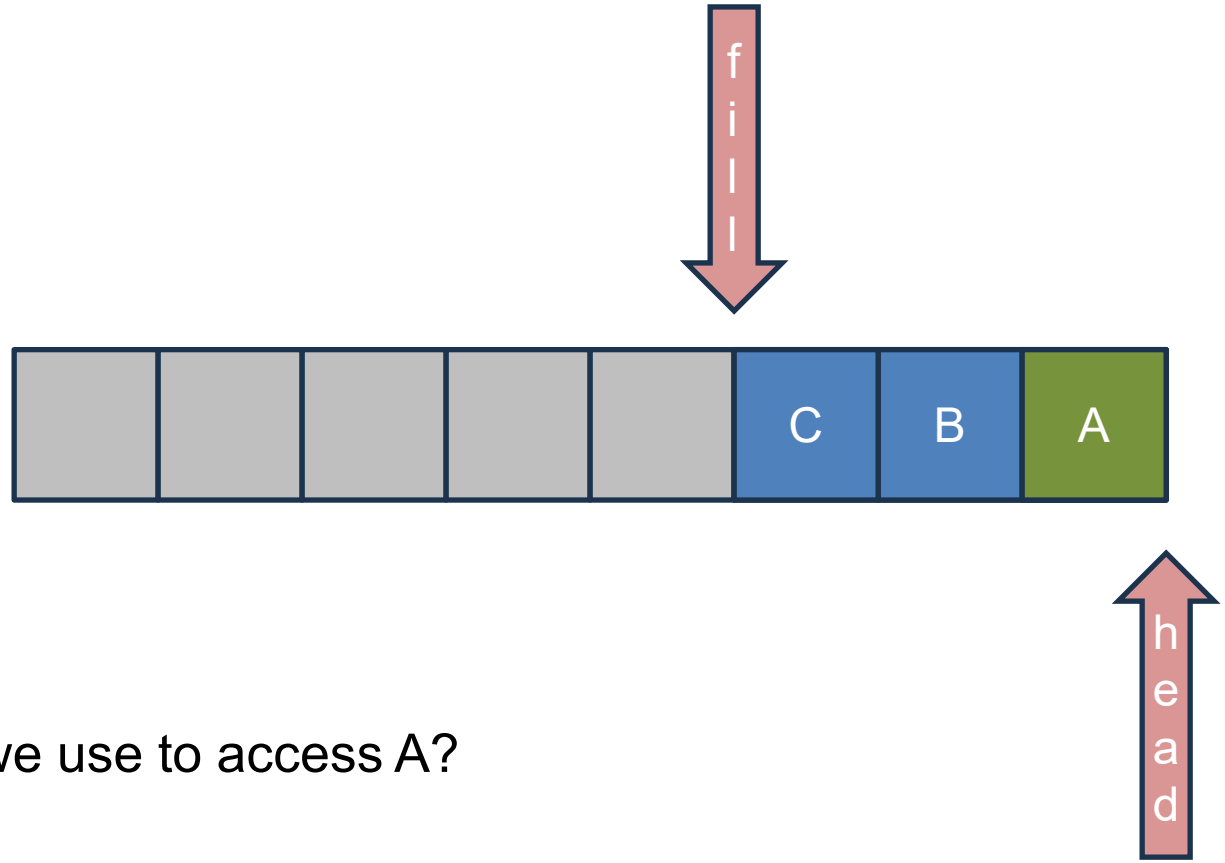
Buffet Control Example

Four-wide sliding window



Buffet Control Example

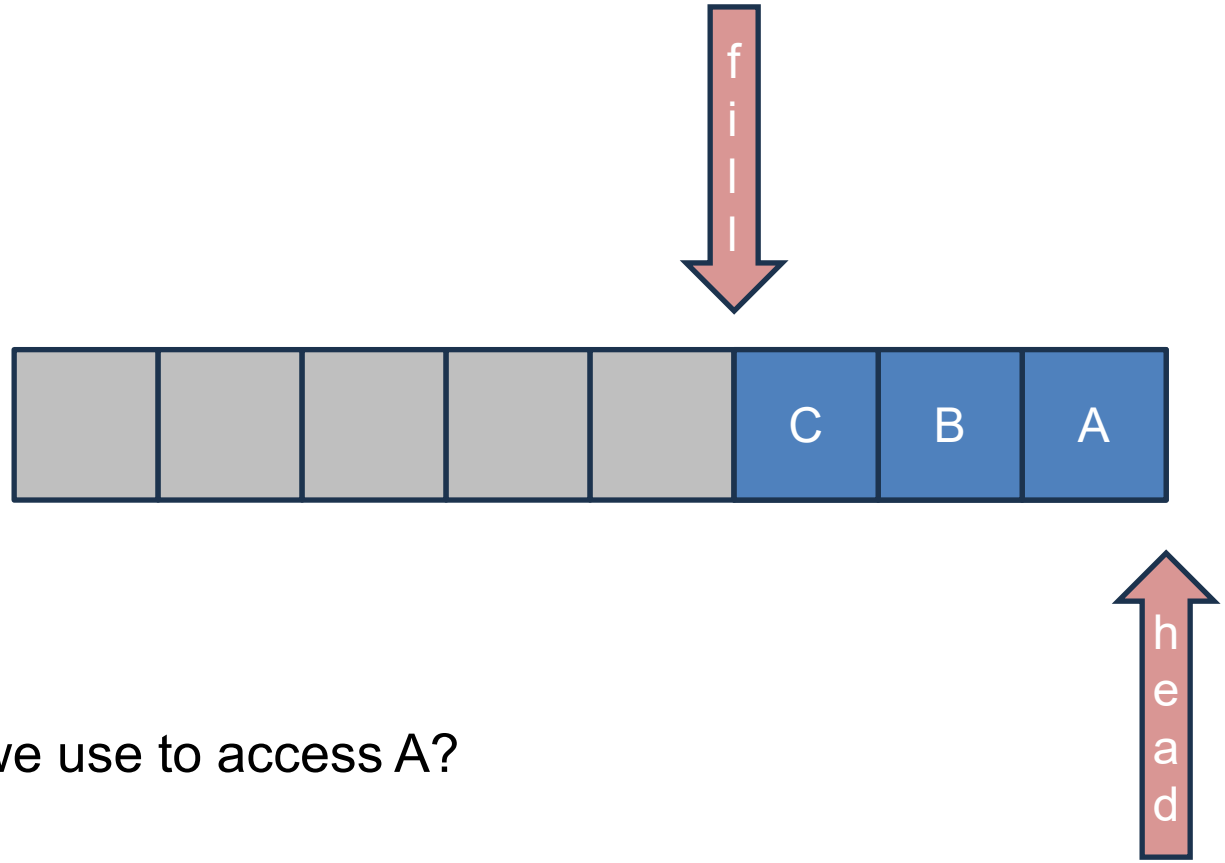
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

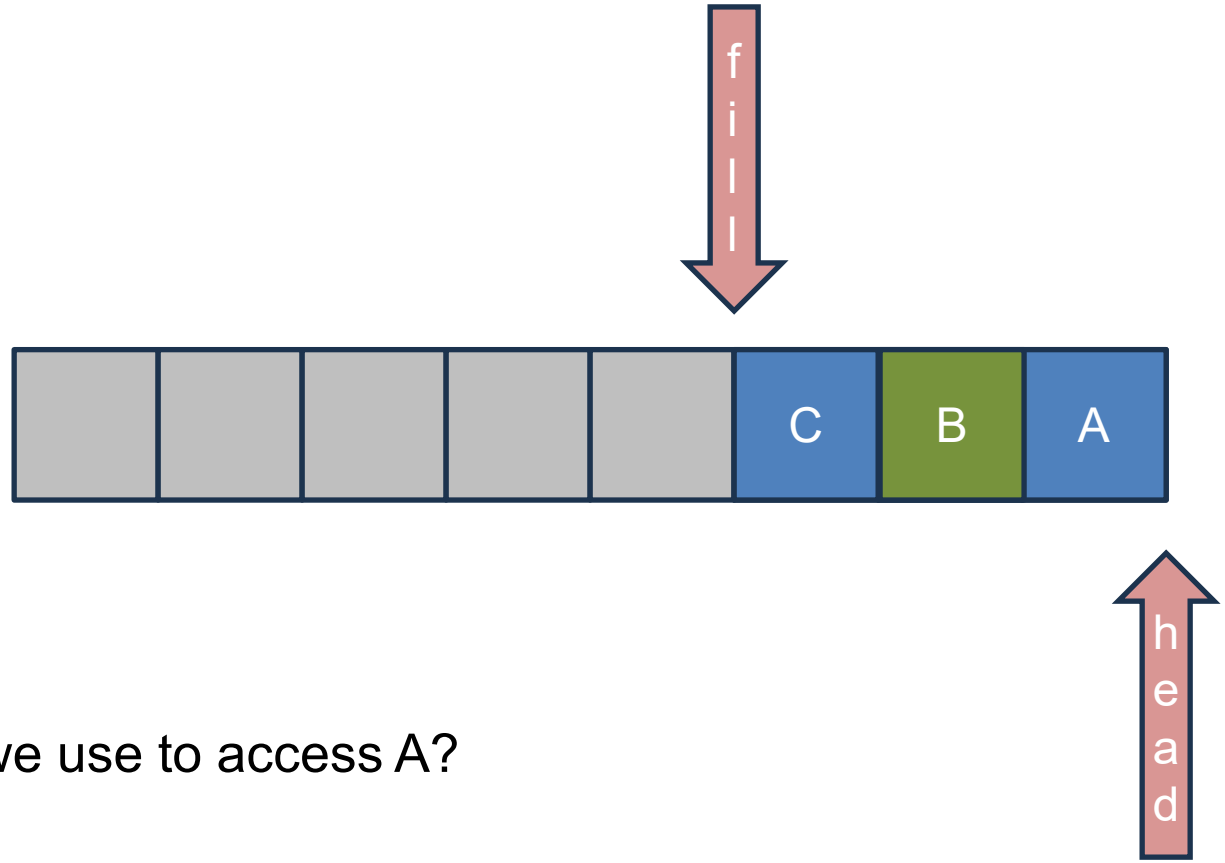
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

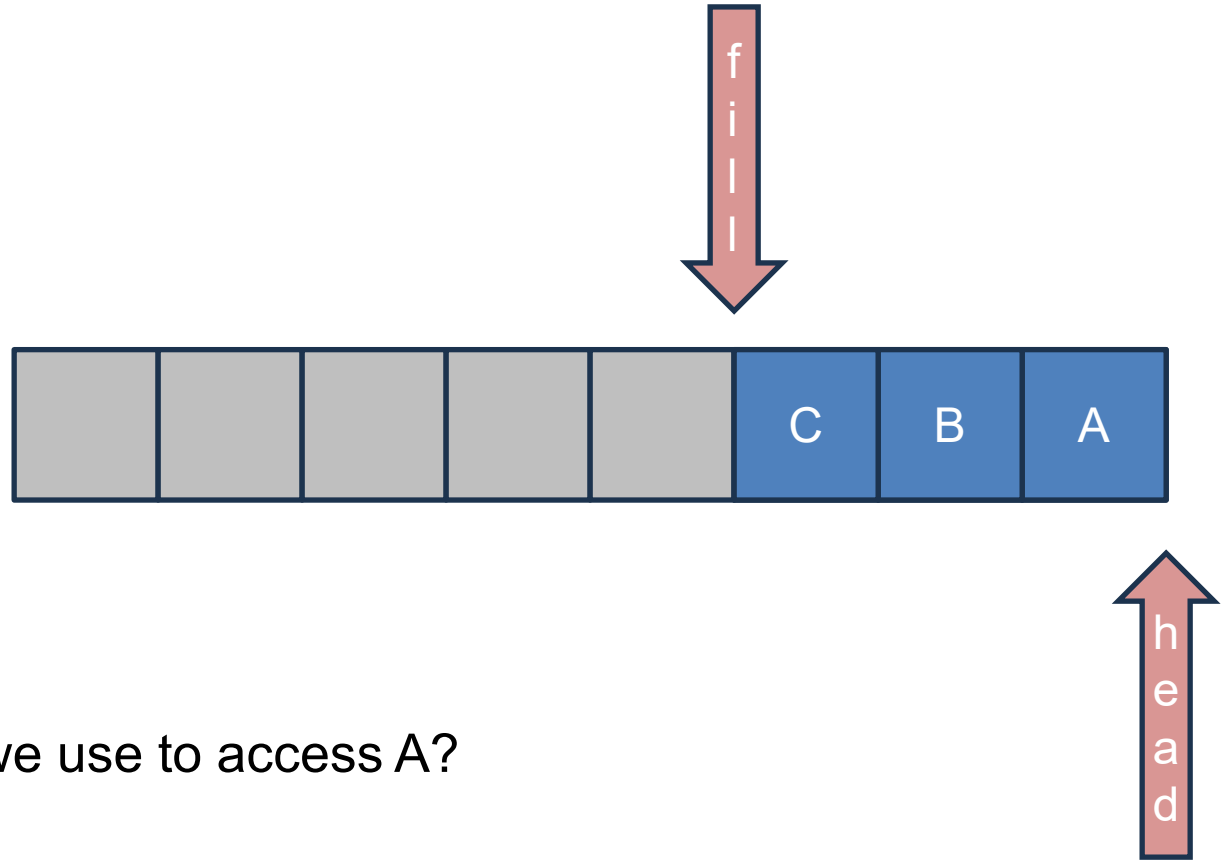
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

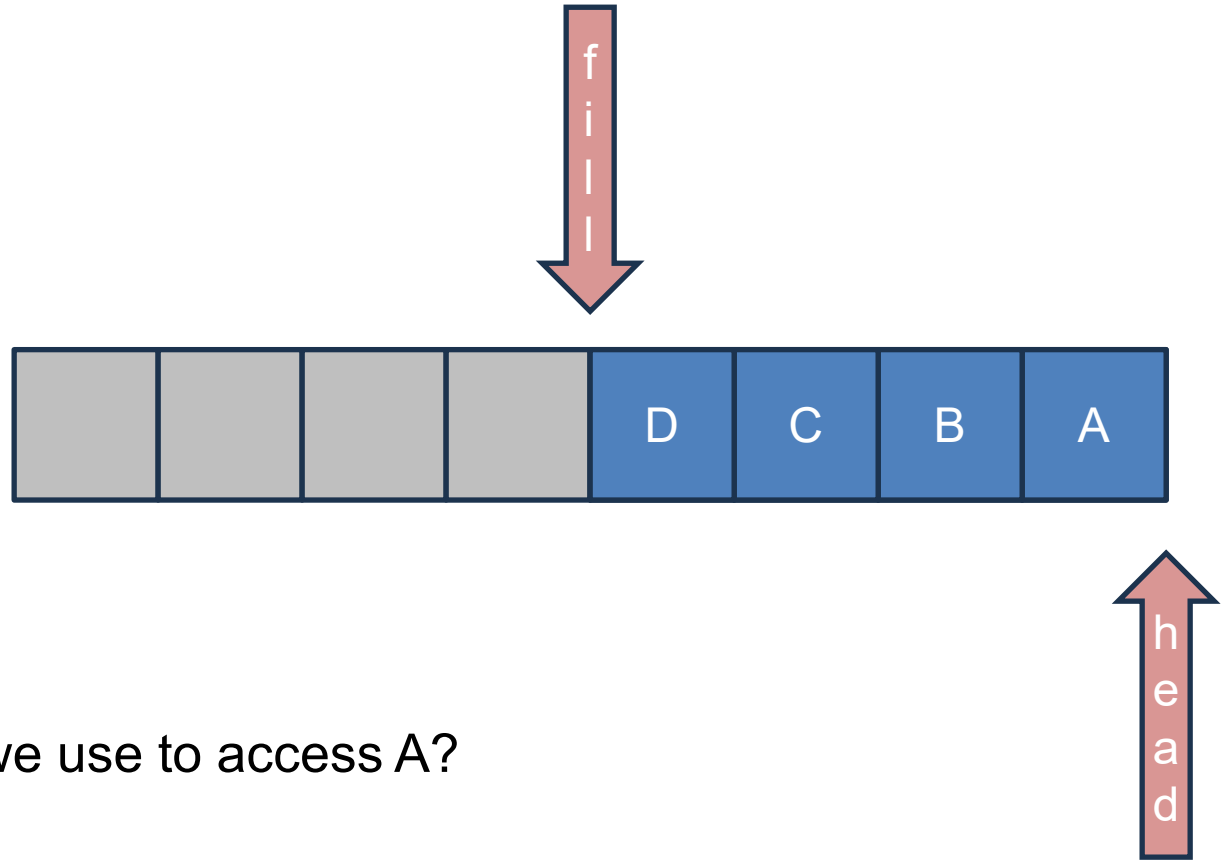
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

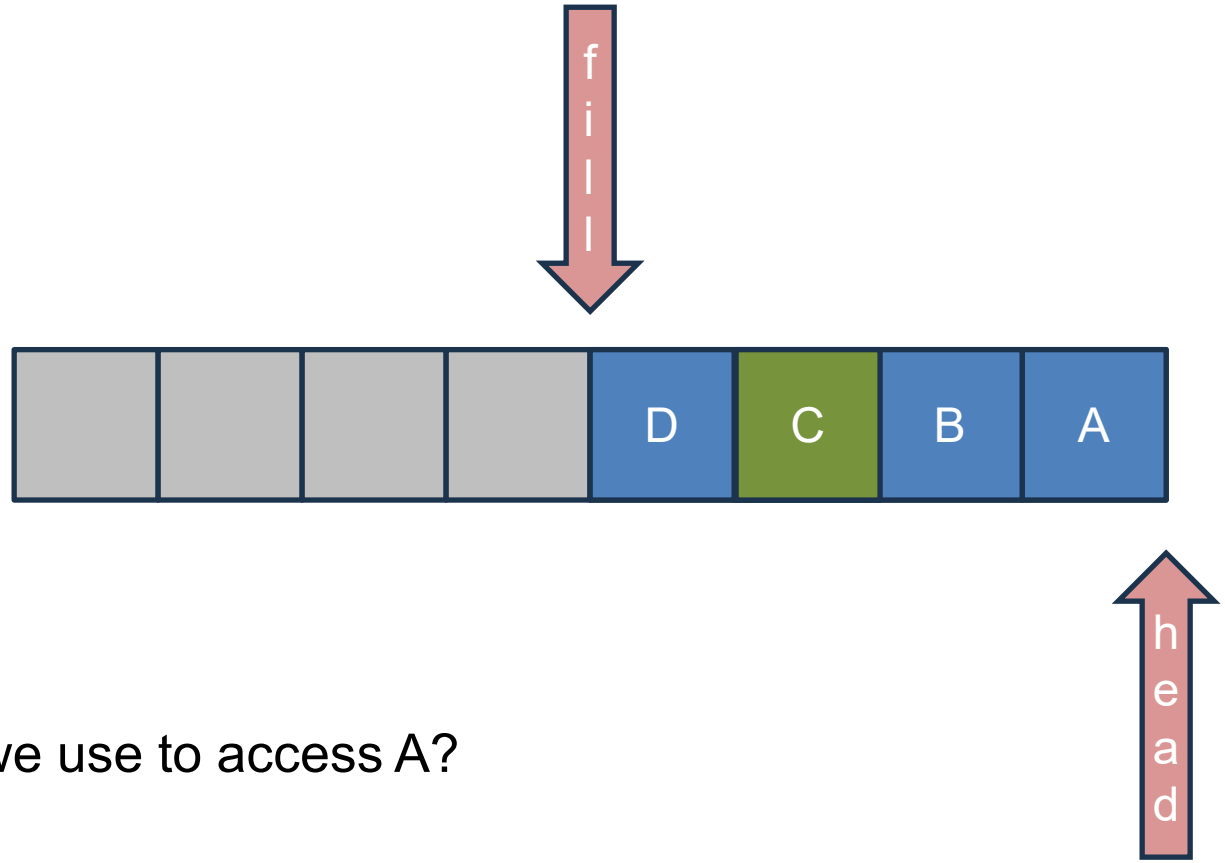
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

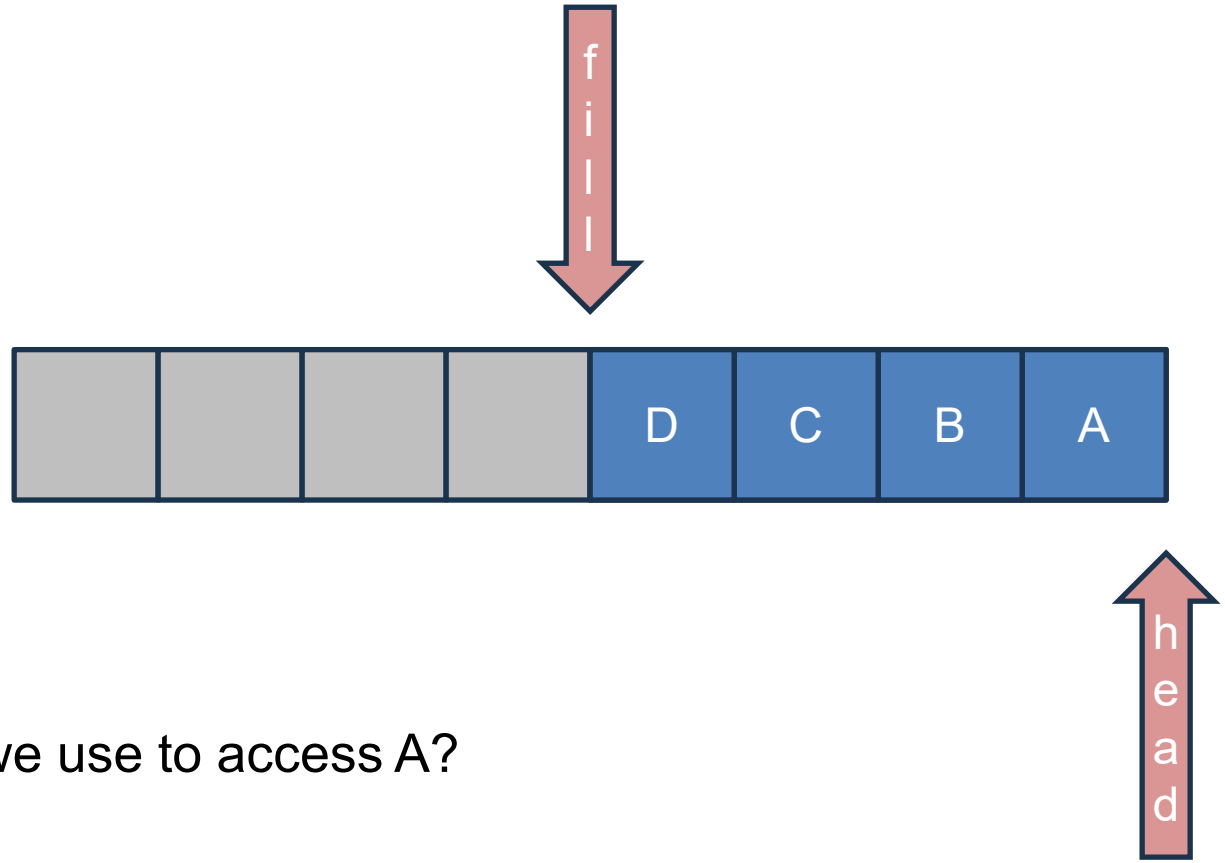
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

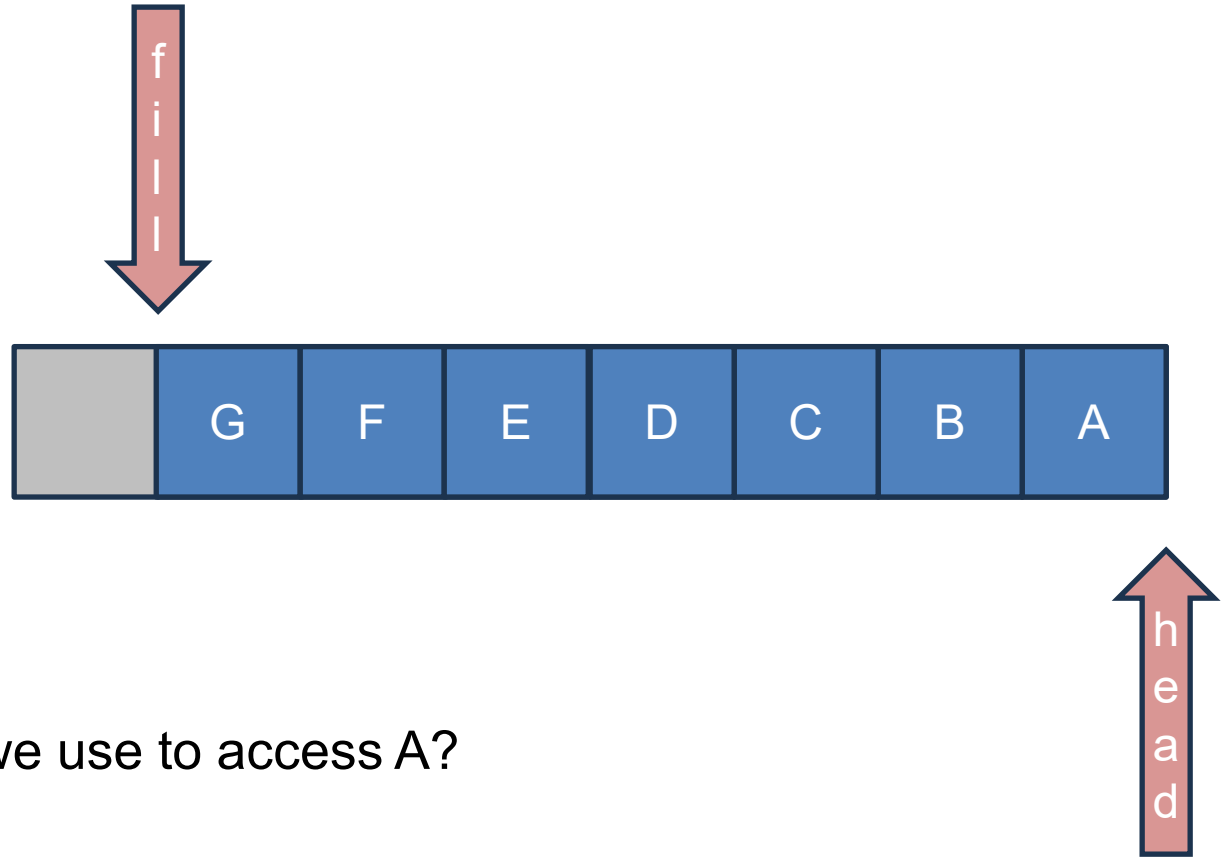
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

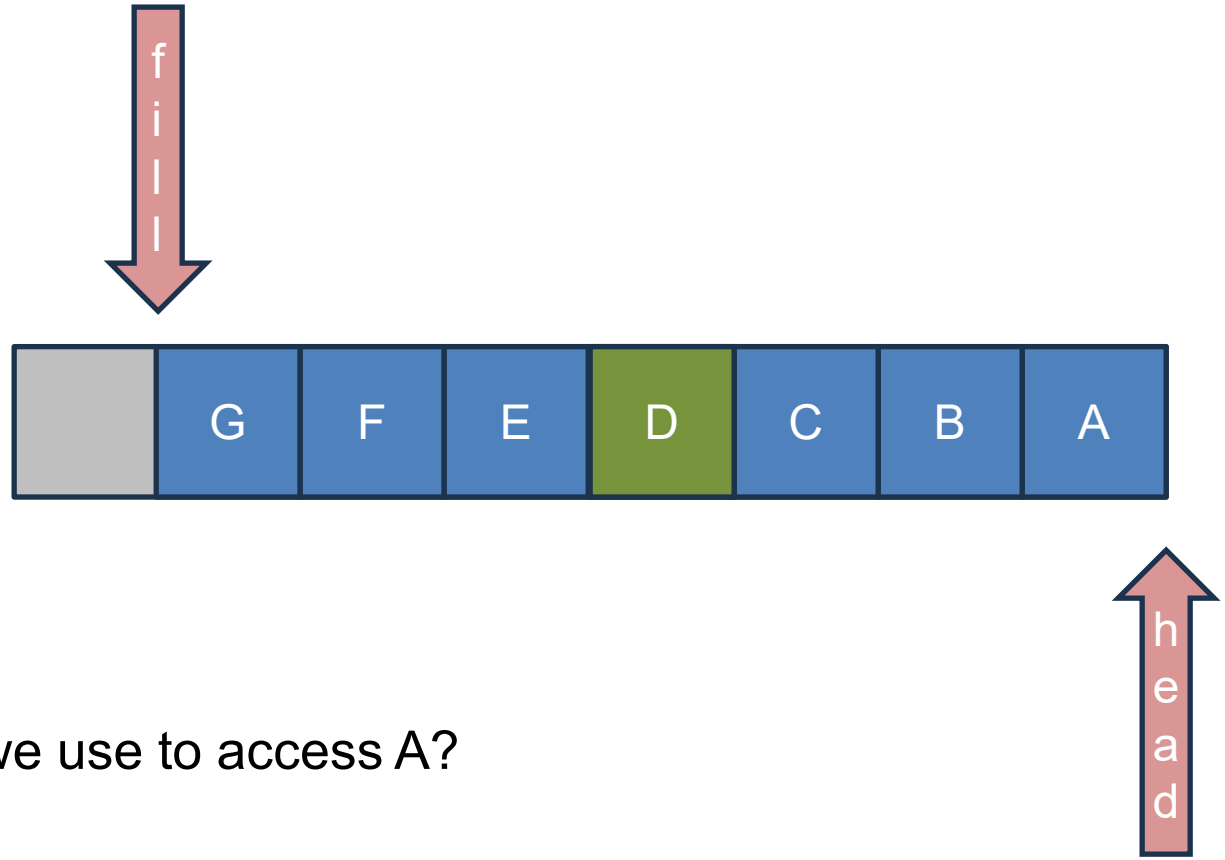
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

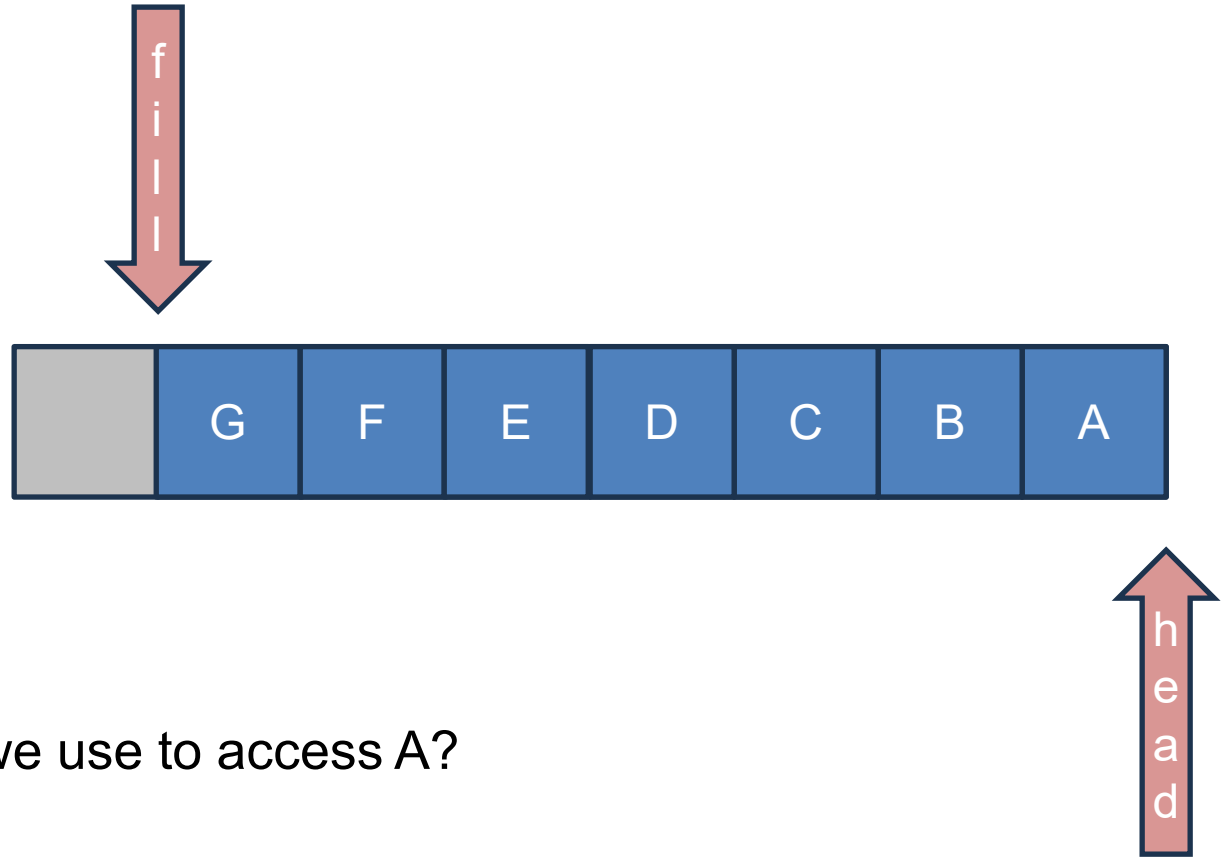
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

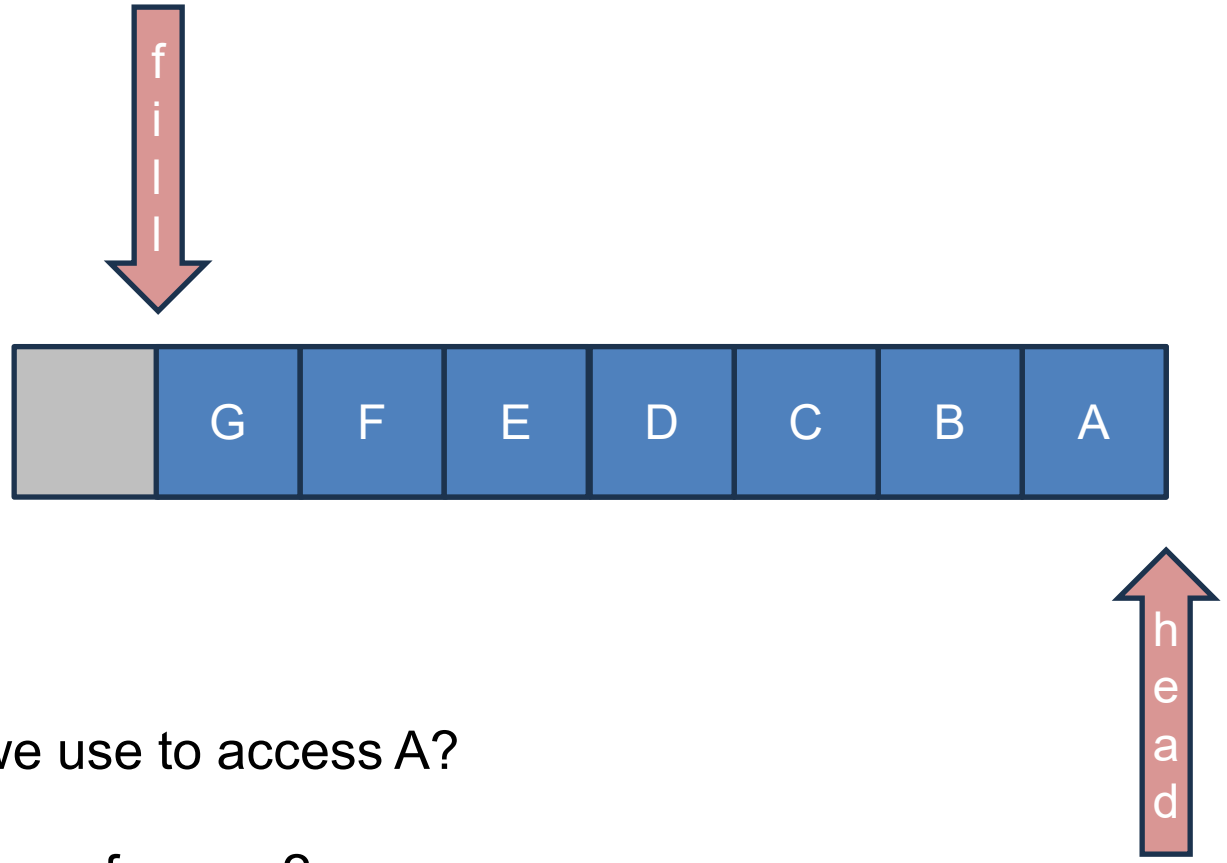
Four-wide sliding window



What offset do we use to access A?

Buffet Control Example

Four-wide sliding window

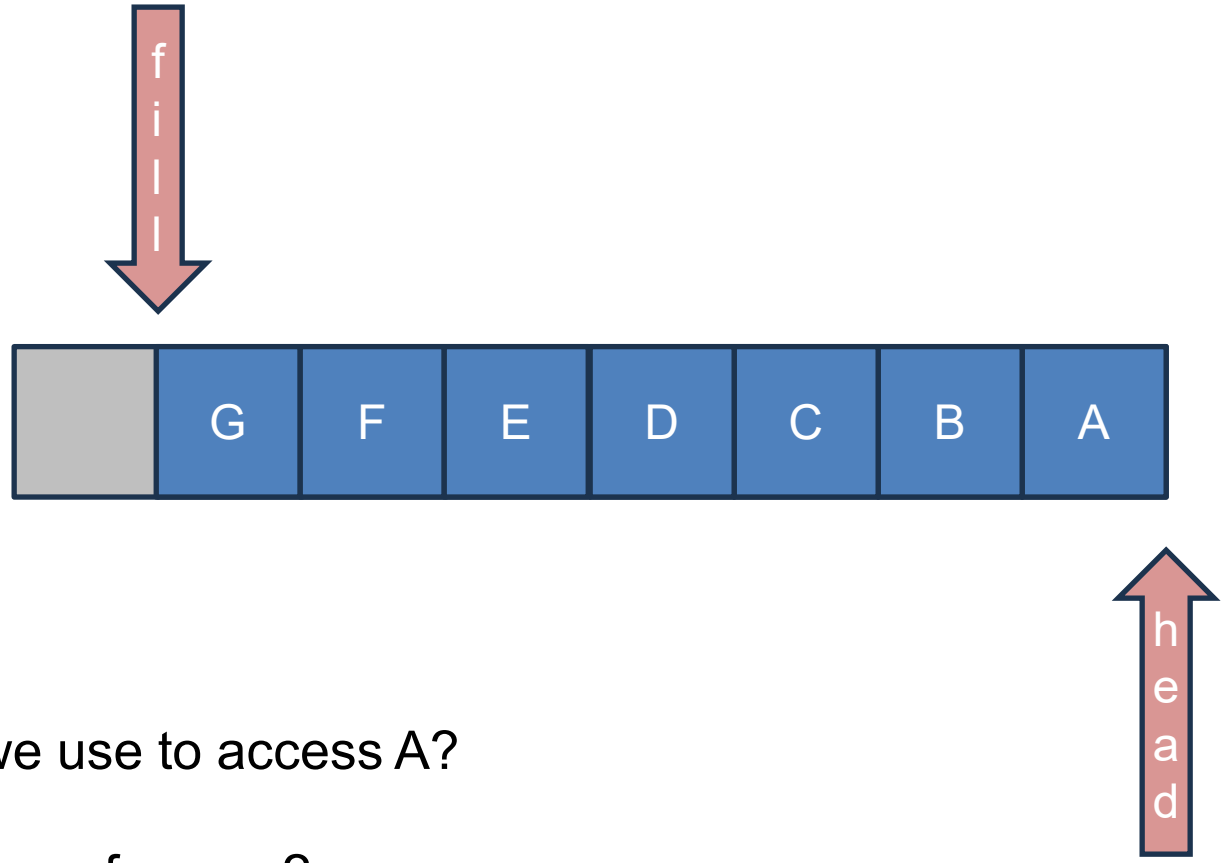


What offset do we use to access A?

Now where do we reference?

Buffet Control Example

Four-wide sliding window



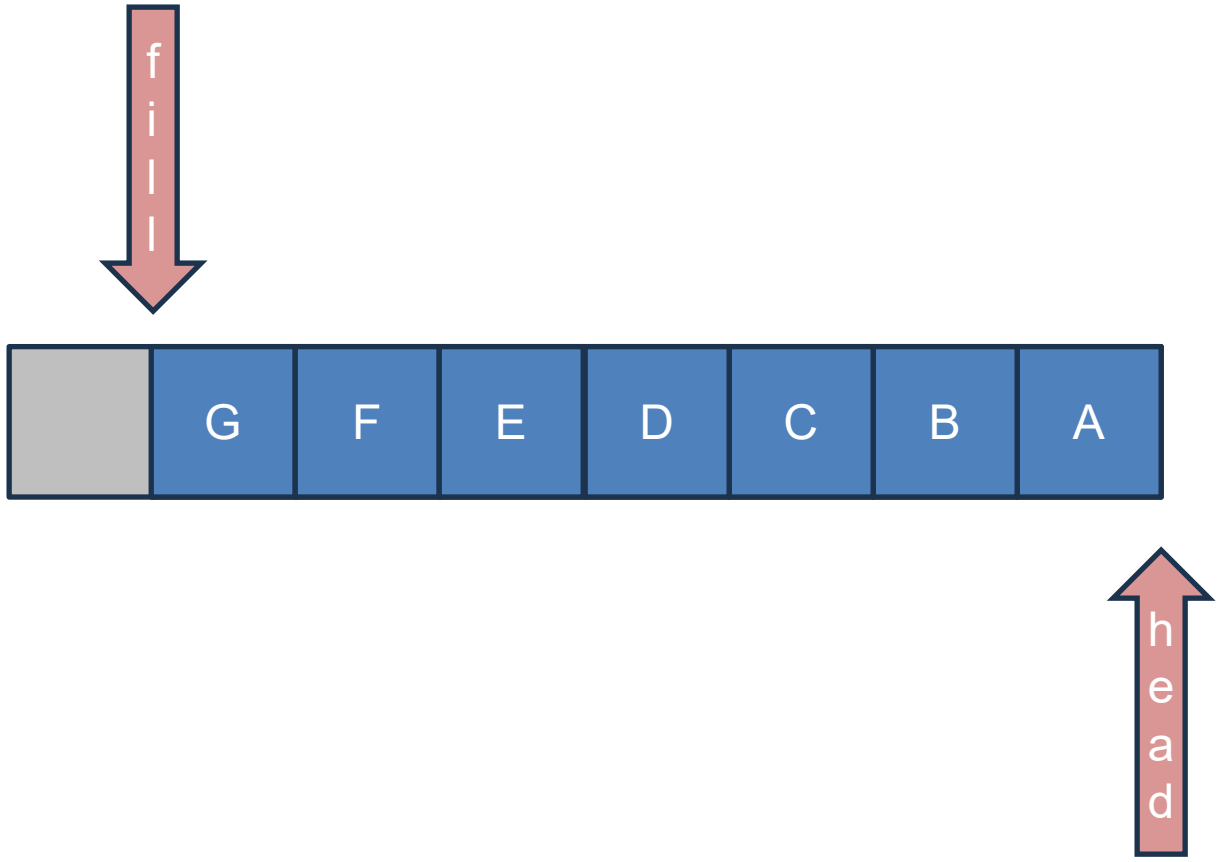
What offset do we use to access A?

Now where do we reference?

But do we need A anymore?

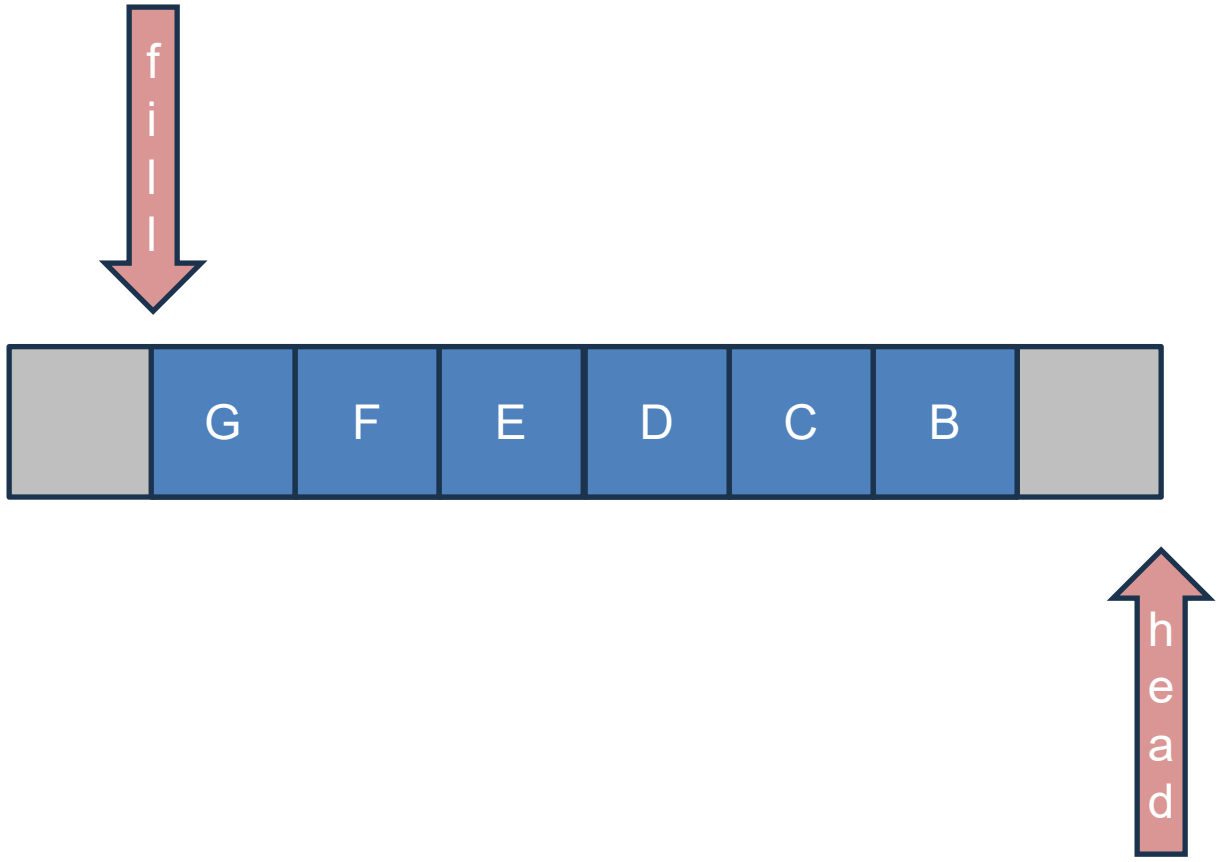
Buffet Control Example

Four-wide sliding window



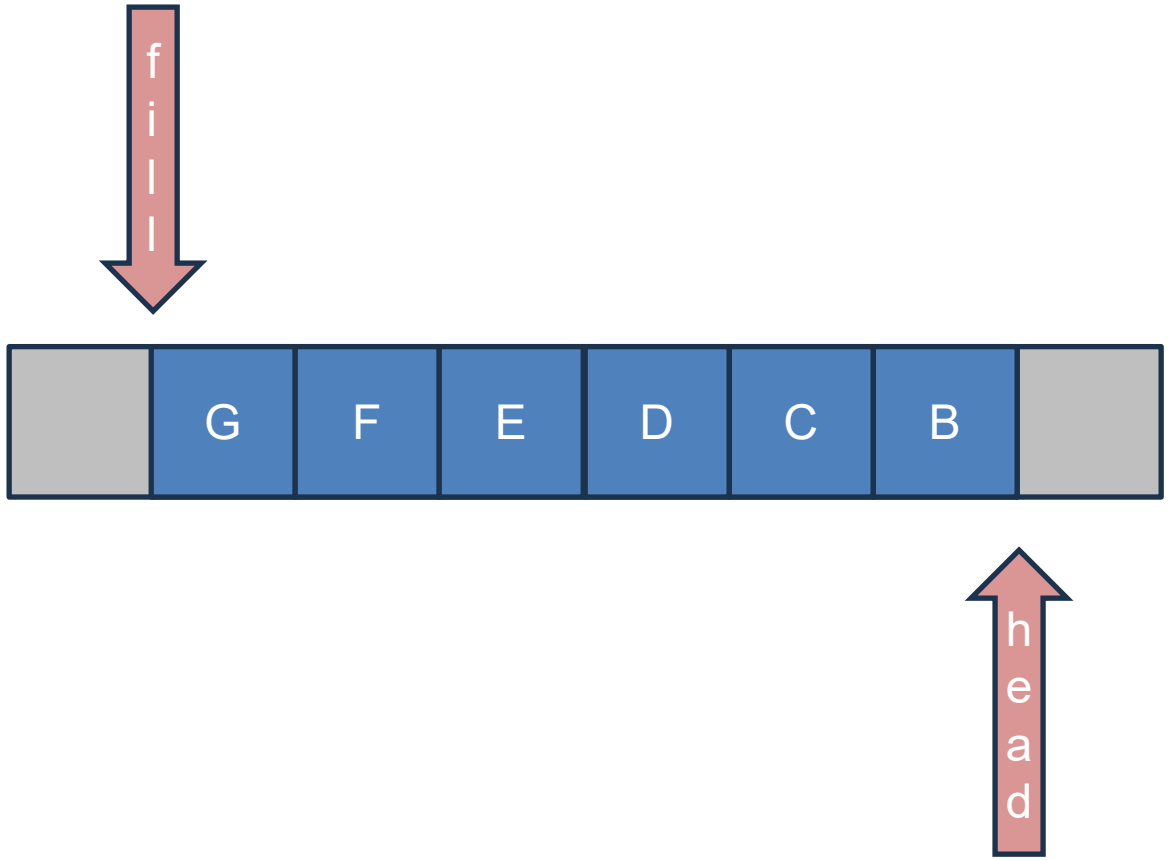
Buffet Control Example

Four-wide sliding window



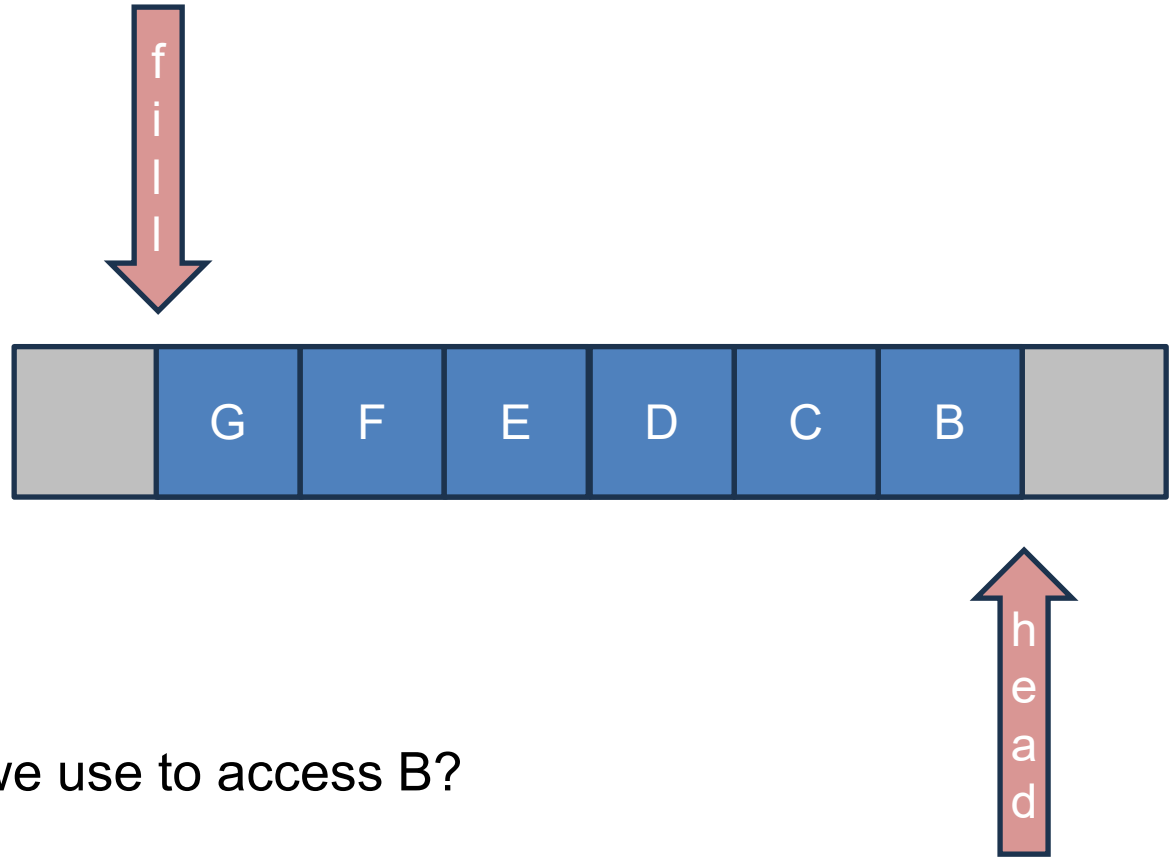
Buffet Control Example

Four-wide sliding window



Buffet Control Example

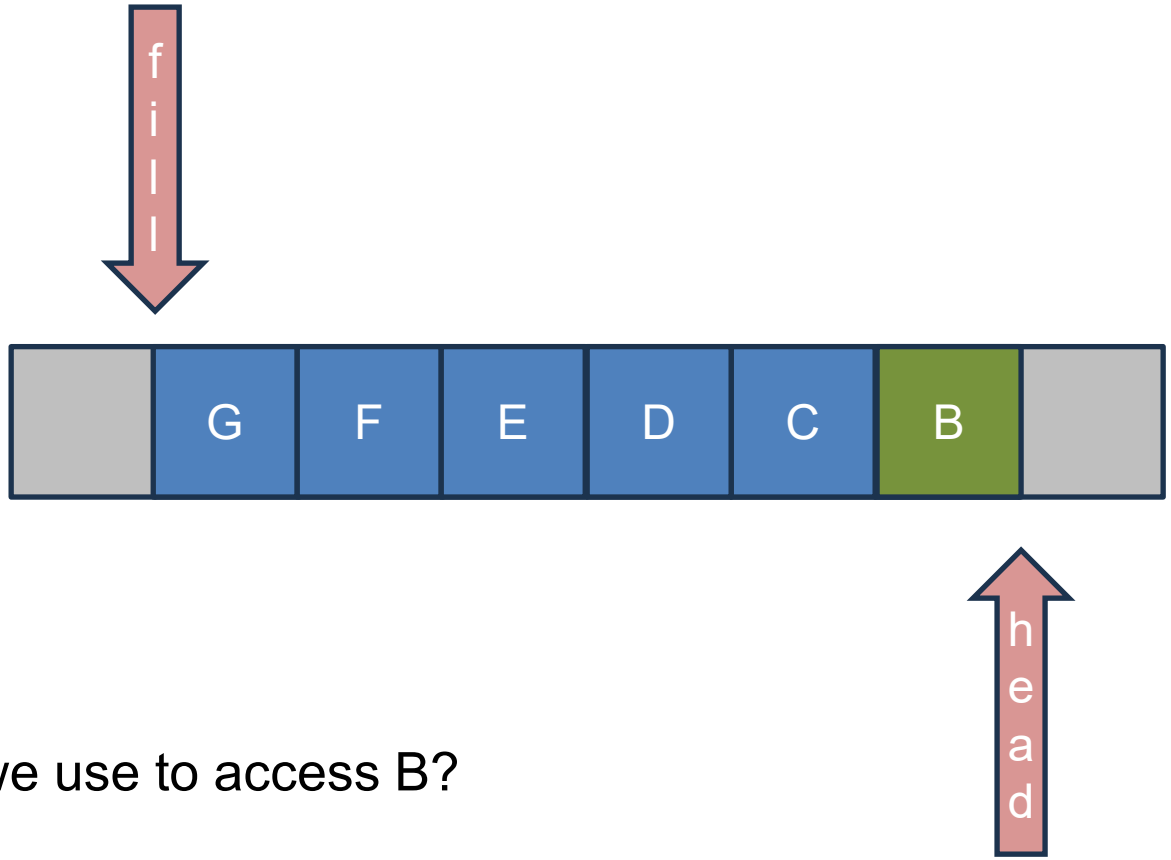
Four-wide sliding window



What offset do we use to access B?

Buffet Control Example

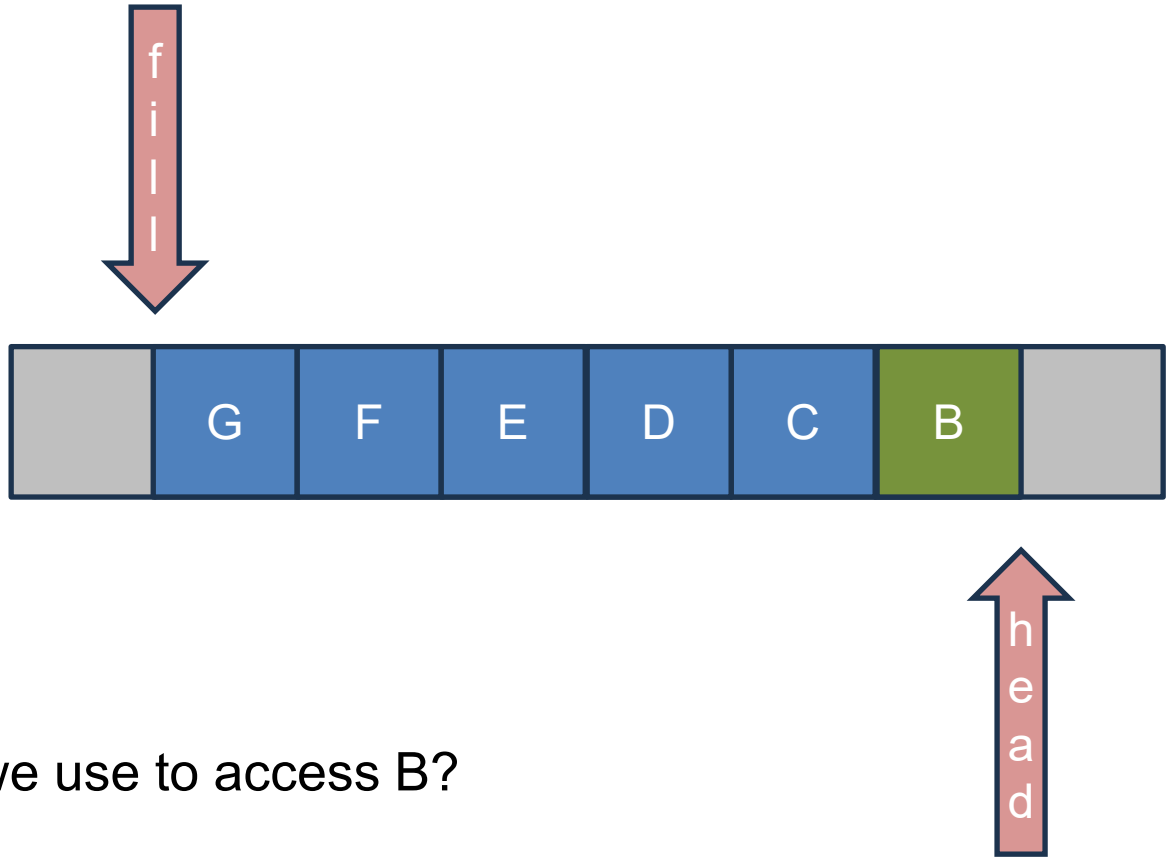
Four-wide sliding window



What offset do we use to access B?

Buffet Control Example

Four-wide sliding window

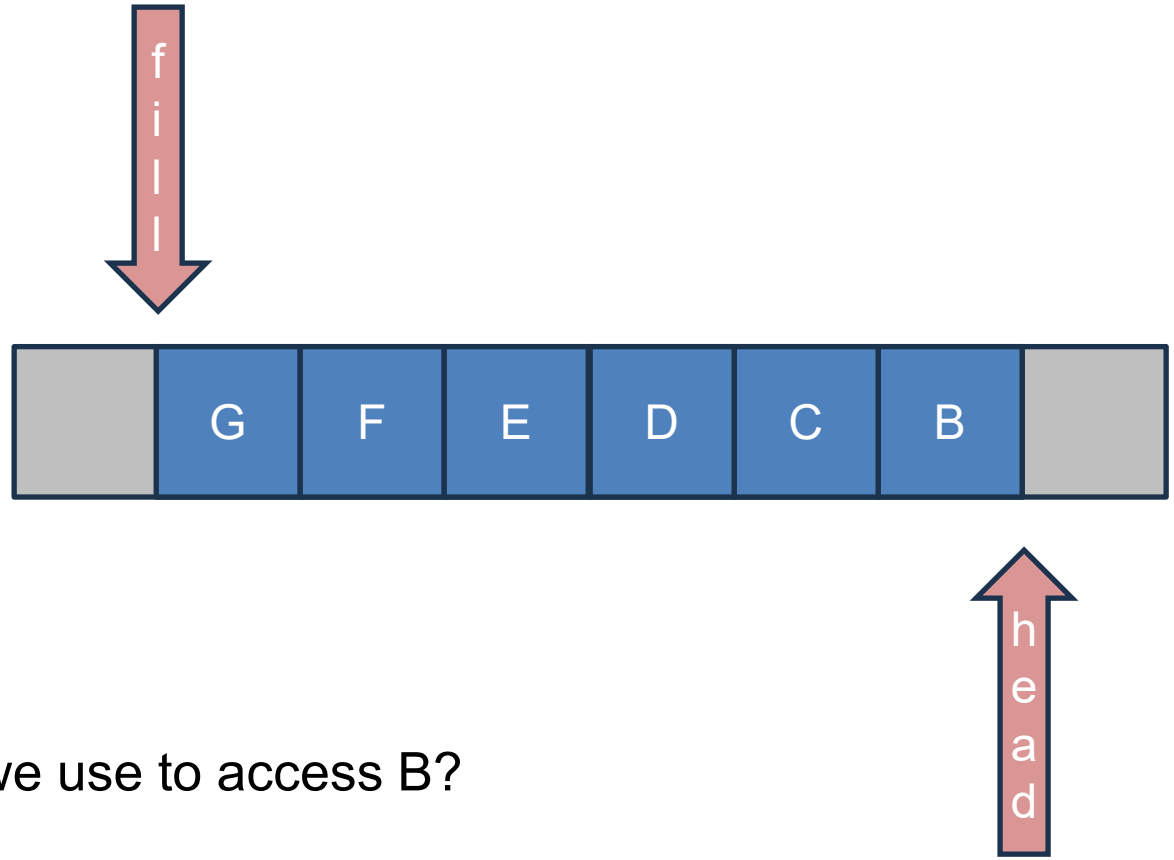


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

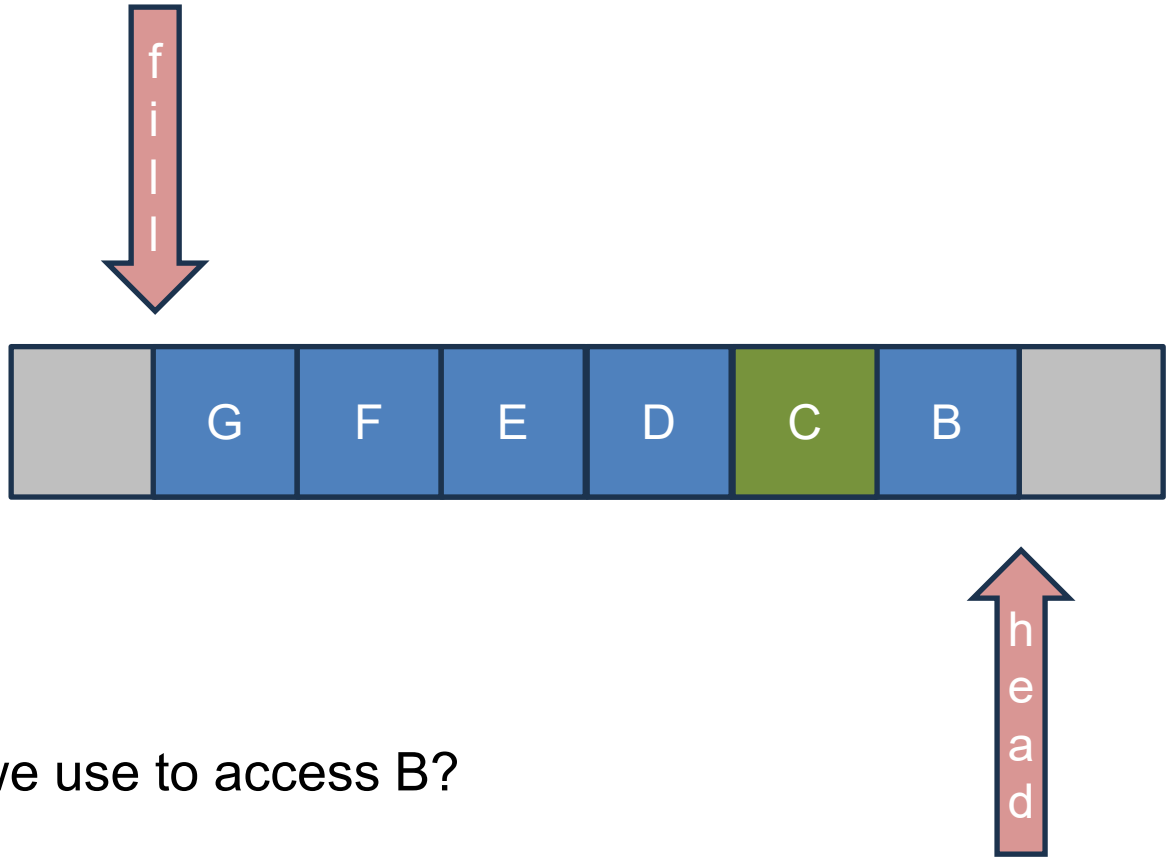


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

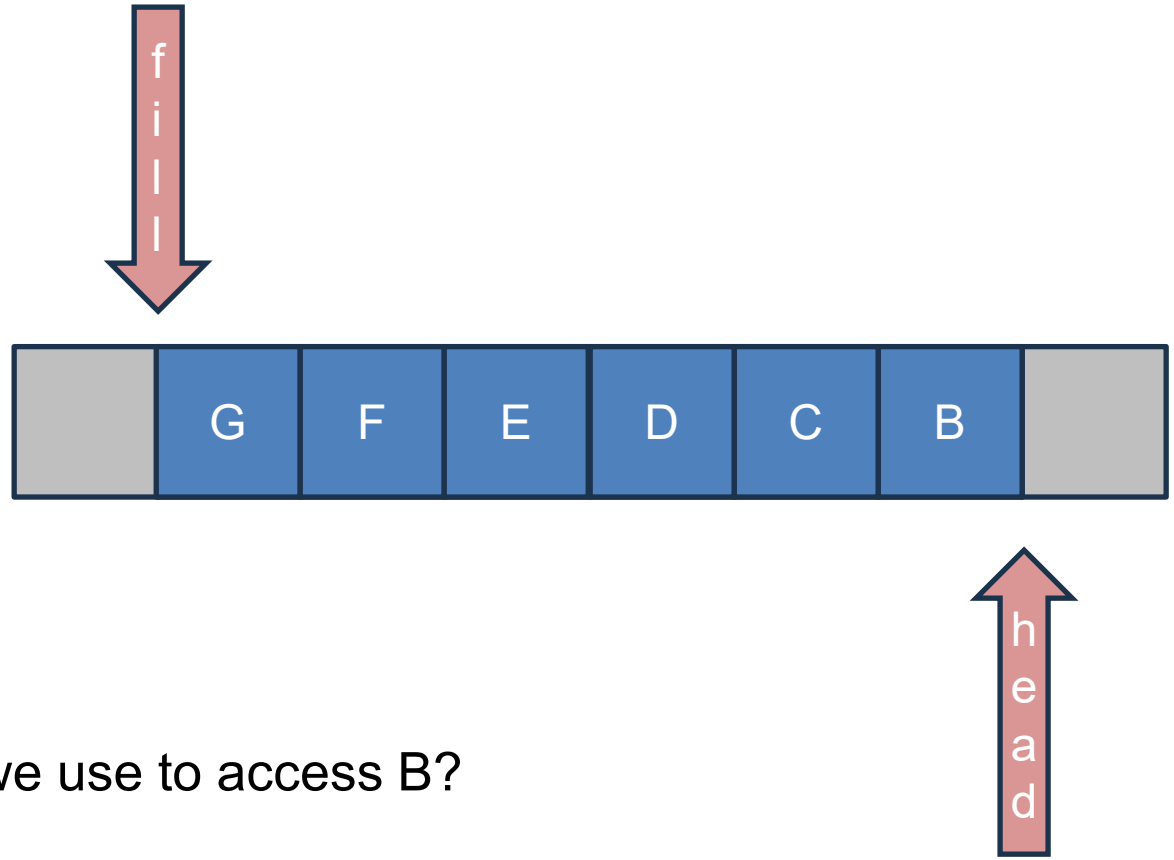


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

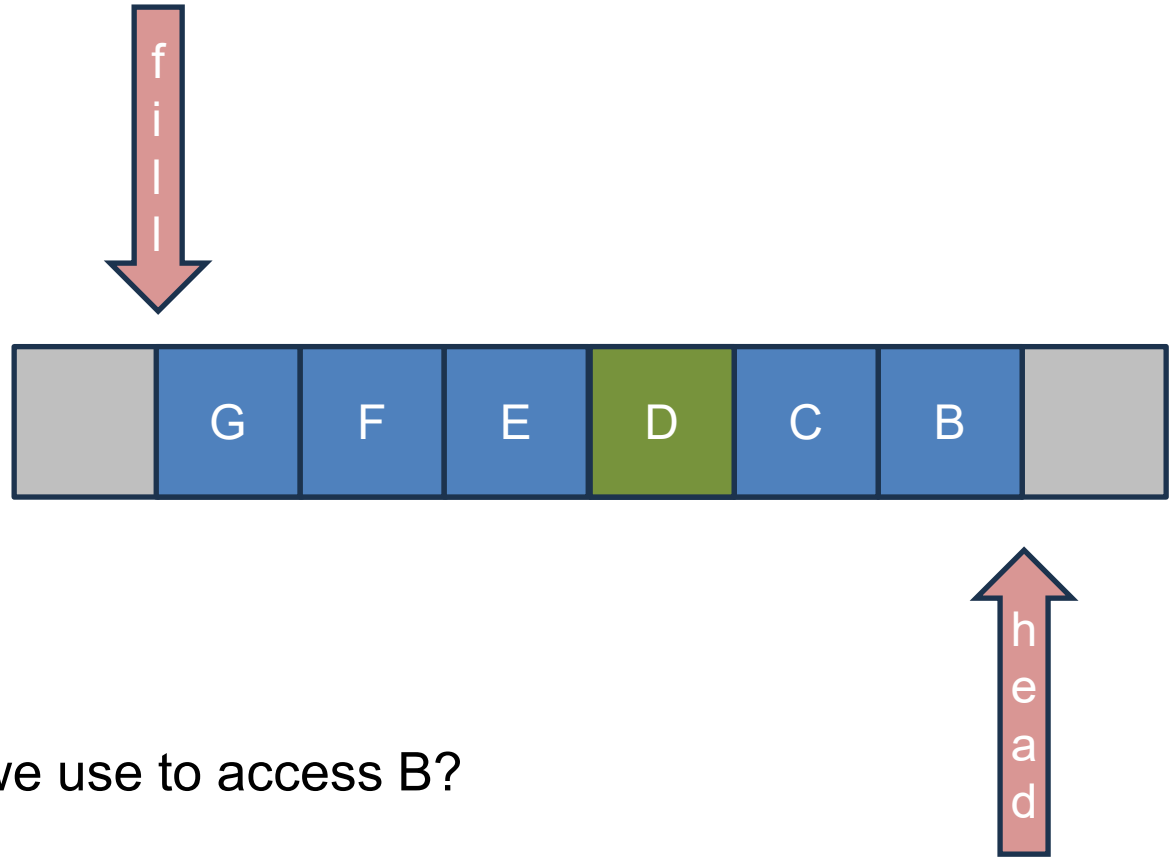


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

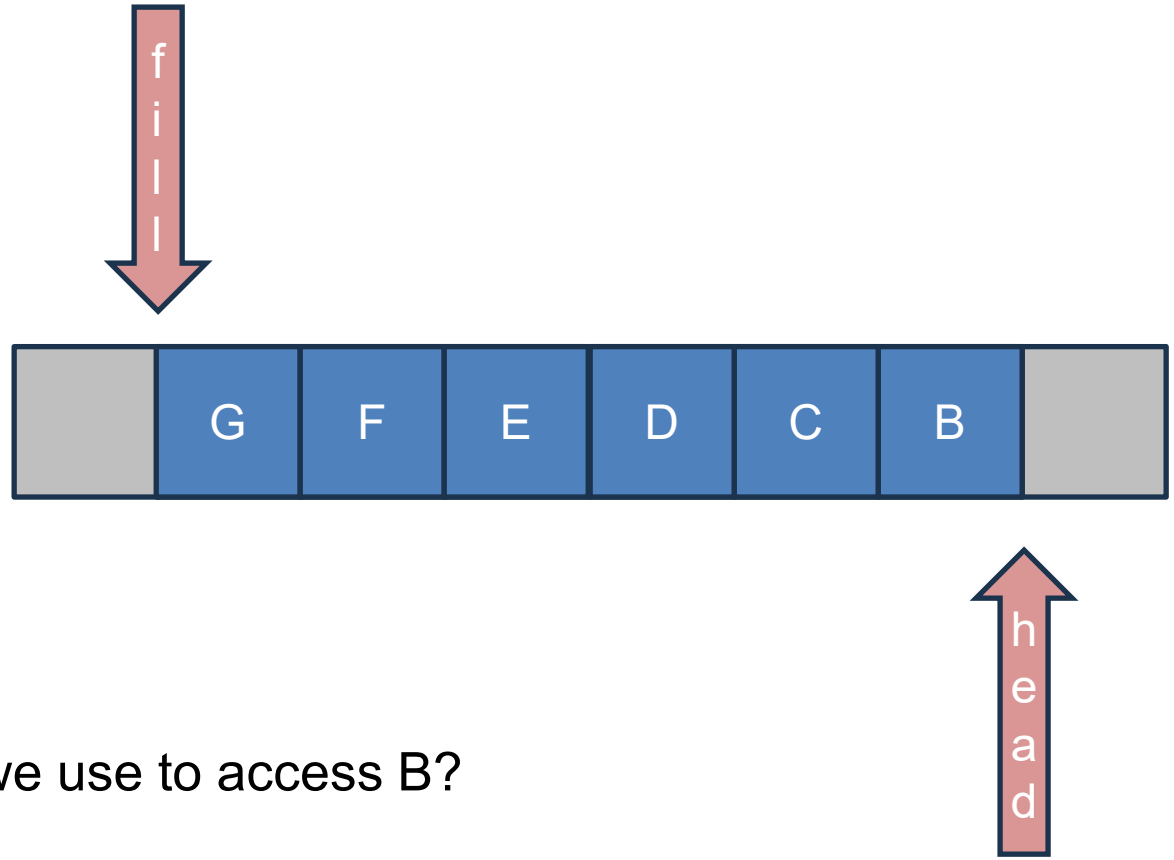


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

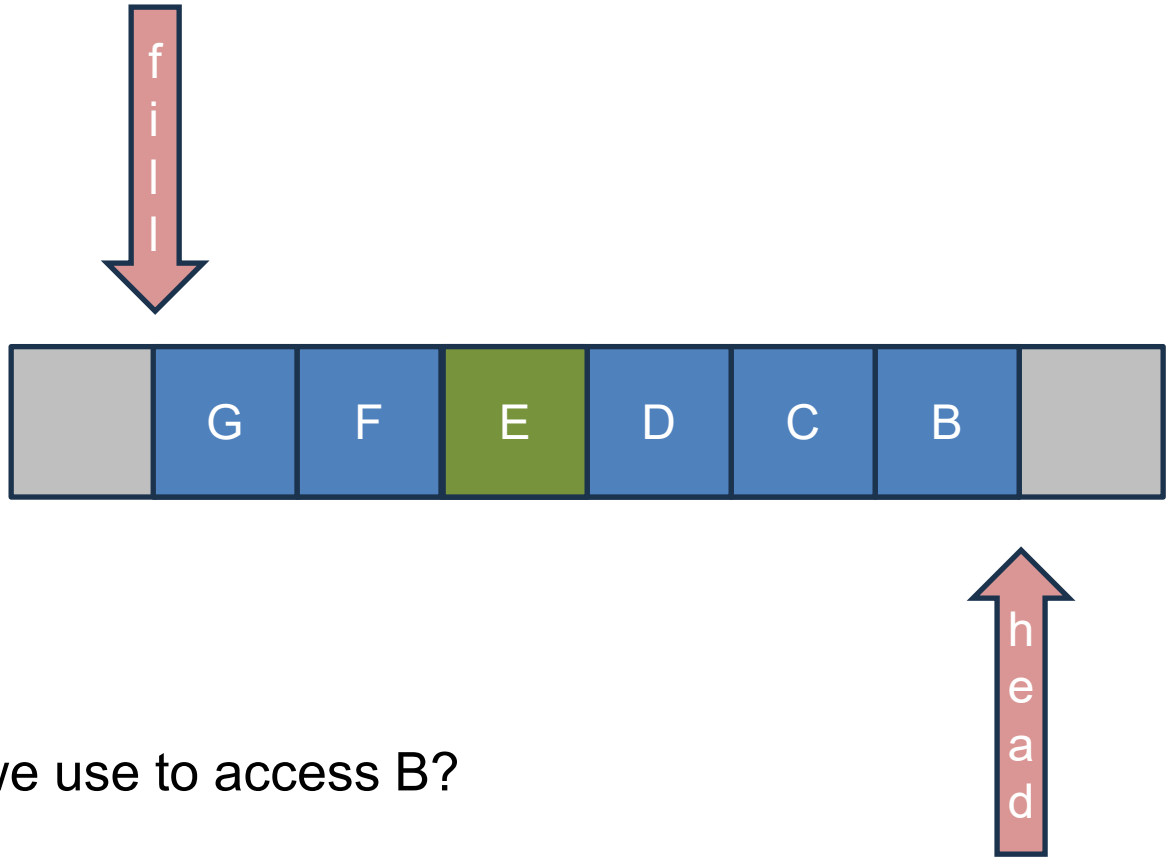


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

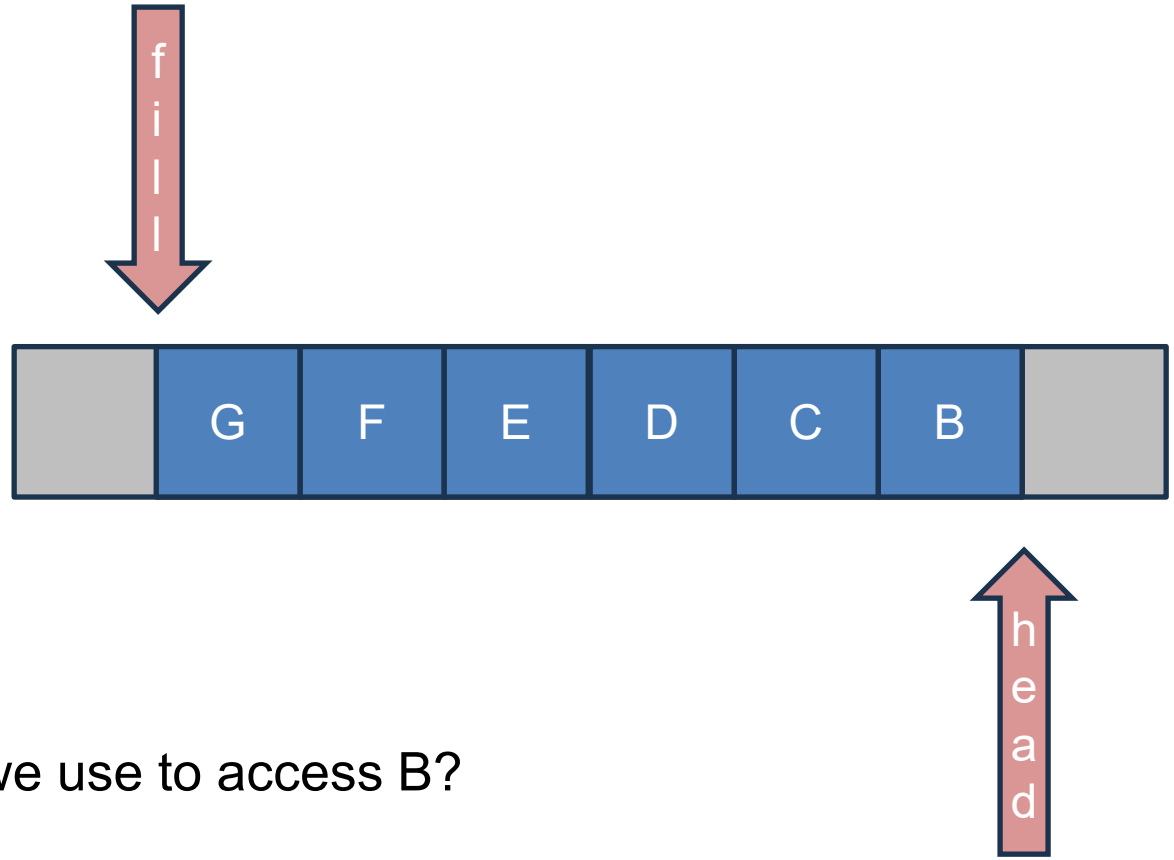


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

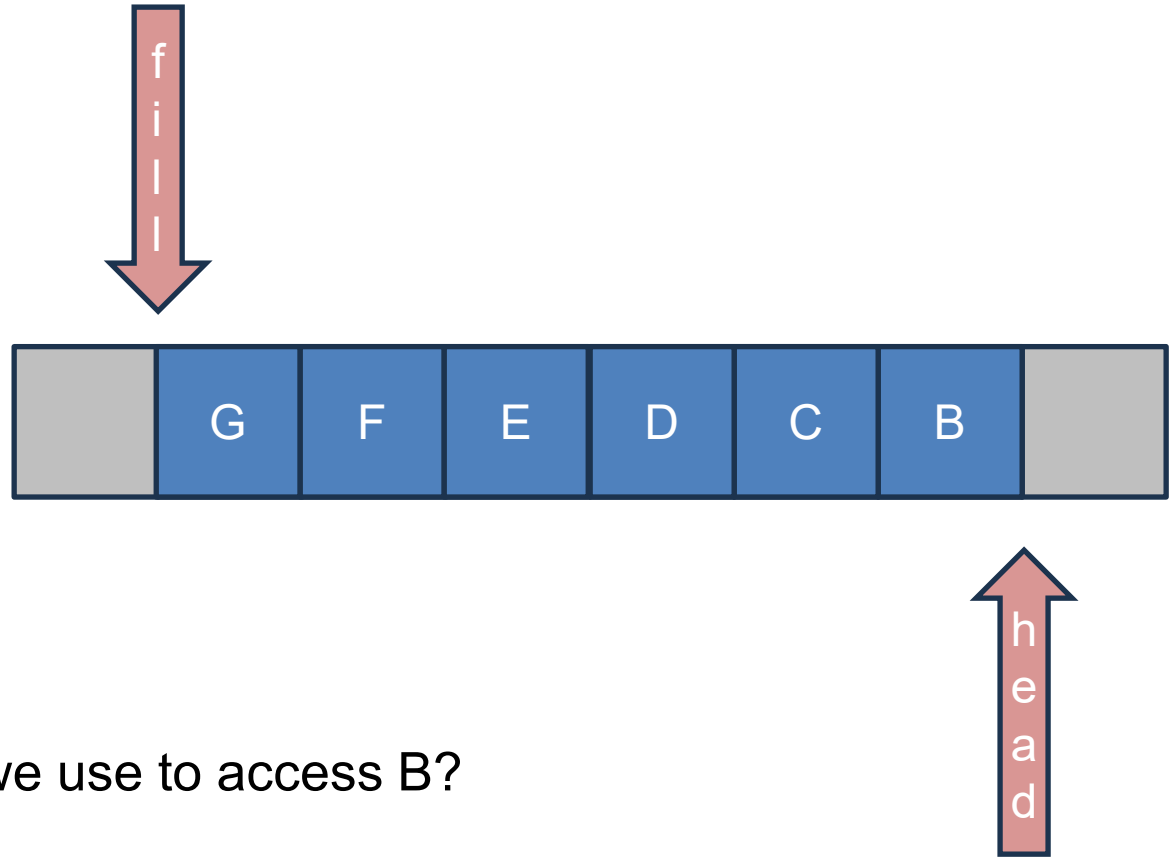


What offset do we use to access B?

What element is the end of this window?

Buffet Control Example

Four-wide sliding window

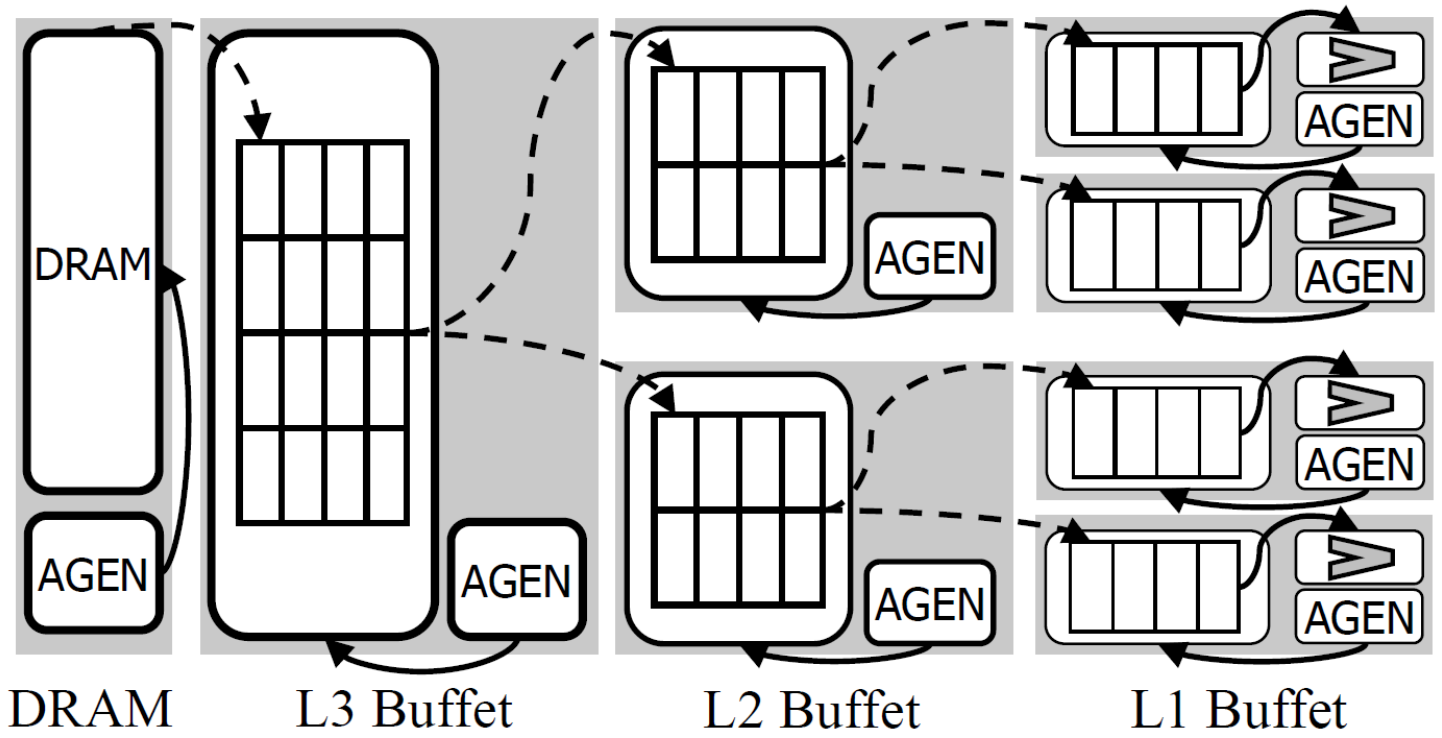


What offset do we use to access B?

What element is the end of this window?

What do we do now?

Buffets: Composable Idiom for E.D.D.O.



Transfers between levels only depend on a credit flow from the adjacent level.

This Lecture

This Lecture

- Continue understanding representation of a convolution using loop nests, including mapping

This Lecture

- Continue understanding representation of a convolution using loop nests, including mapping

- See how costs of a mapping can be determined from the loop nest representation.

This Lecture

- Continue understanding representation of a convolution using loop nests, including mapping
- See how costs of a mapping can be determined from the loop nest representation.
- See how loop nest can guide configuration of an accelerator.

This Lecture

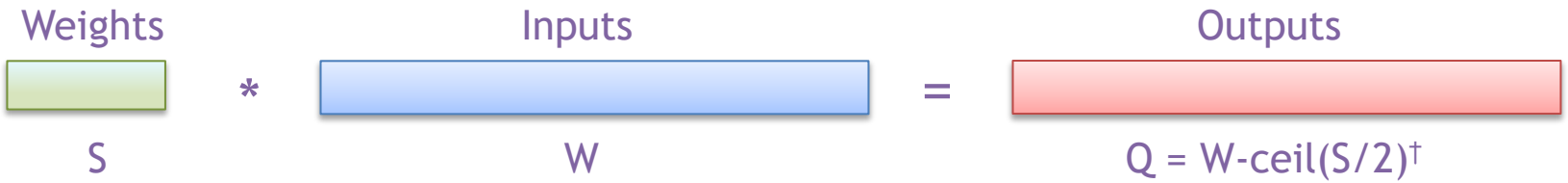
- Continue understanding representation of a convolution using loop nests, including mapping
- See how costs of a mapping can be determined from the loop nest representation.
- See how loop nest can guide configuration of an accelerator.
- Consider a loop nest representation for a full CNN layer and how to search for an optimal mapping

This Lecture

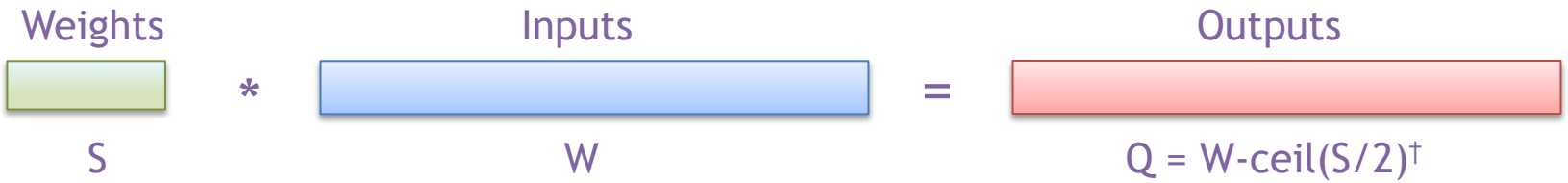
- Continue understanding representation of a convolution using loop nests, including mapping
- See how costs of a mapping can be determined from the loop nest representation.
- See how loop nest can guide configuration of an accelerator.
- Consider a loop nest representation for a full CNN layer and how to search for an optimal mapping
- Reading: Efficient Processing of Deep Neural Networks – Chap 5/6

Mapping Output Stationary to Hardware

1-D Convolution – Output Stationary

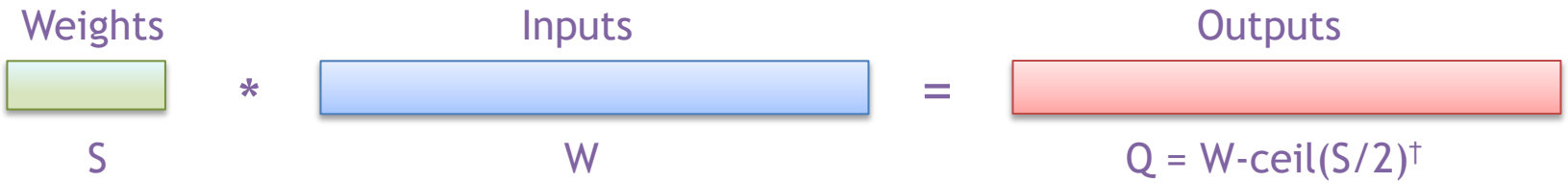


1-D Convolution – Output Stationary



$$O_q = I_{q+s} \times F_s$$

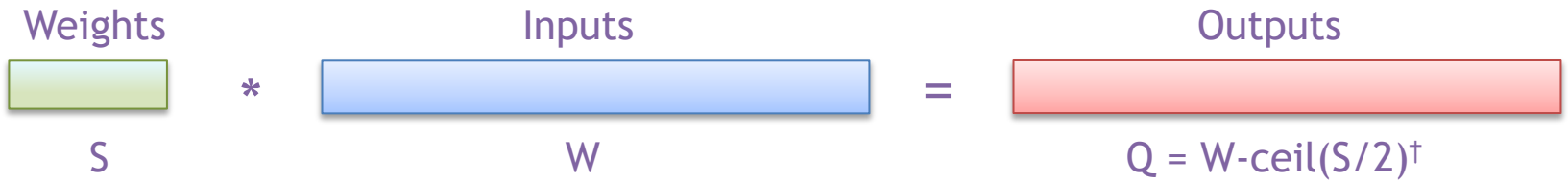
1-D Convolution – Output Stationary



$$O_q = I_{q+s} \times F_s$$

Traversal order (fastest to slowest): S, Q

1-D Convolution – Output Stationary

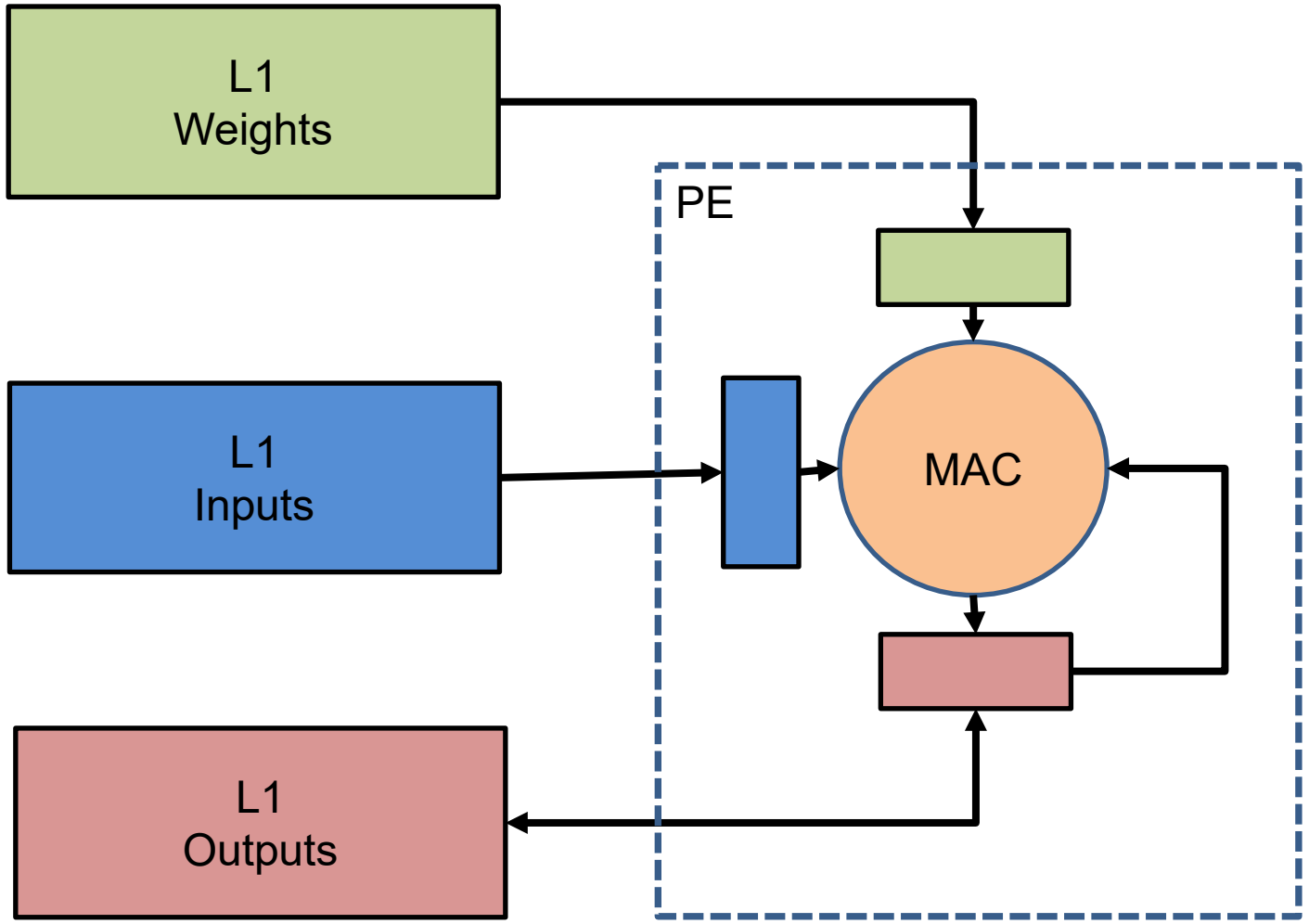


$$O_q = I_{q+s} \times F_s$$

Traversal order (fastest to slowest): S, Q

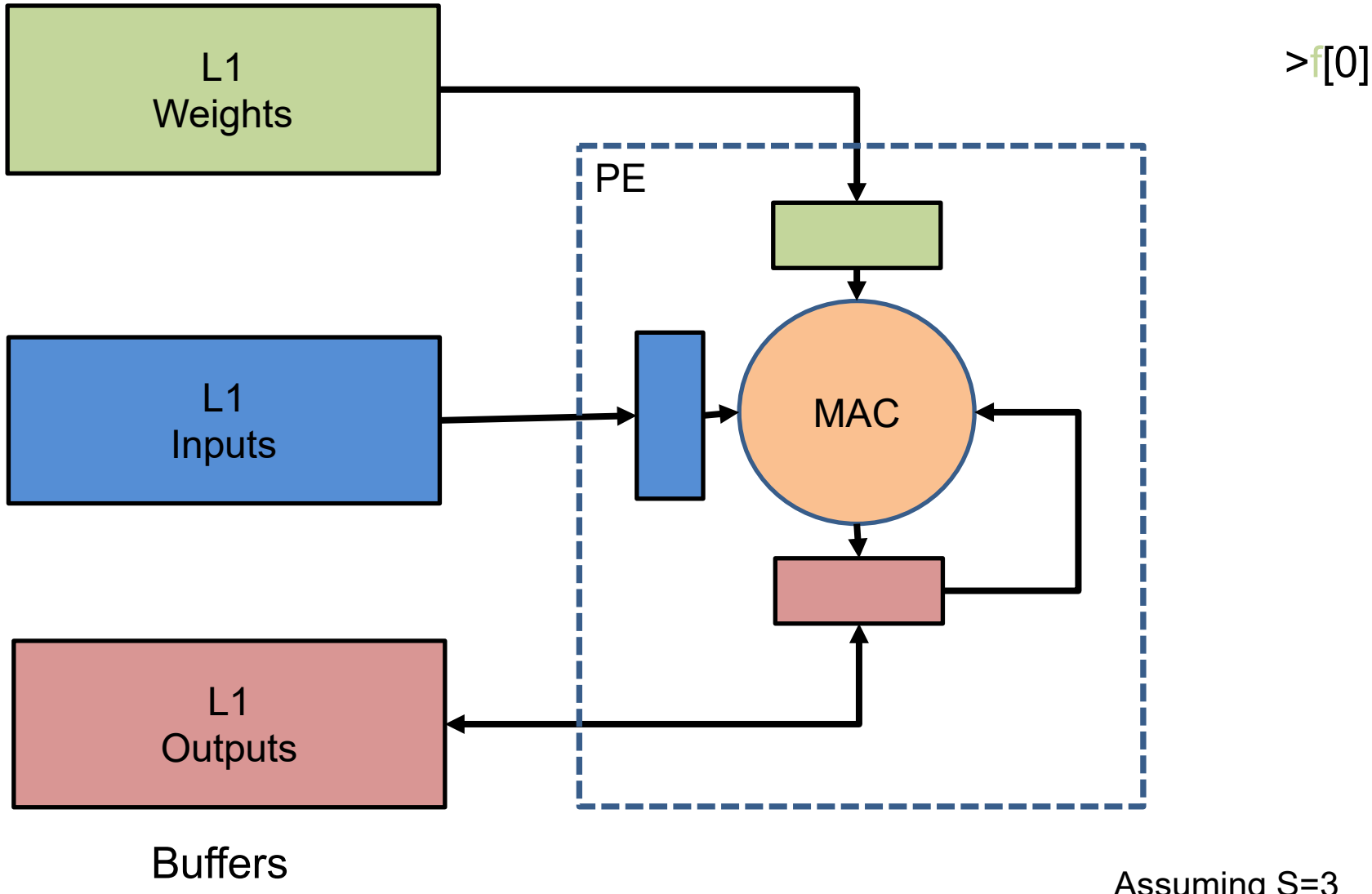
```
int i[W];      # Input activations
int f[S];      # Filter weights
int o[Q];      # Output activations

for q in [0, Q):
        for s in (0, S):
            o[q] += i[q+s]*f[s]
```

Single PE Output Stationary Flow



Assuming S=3

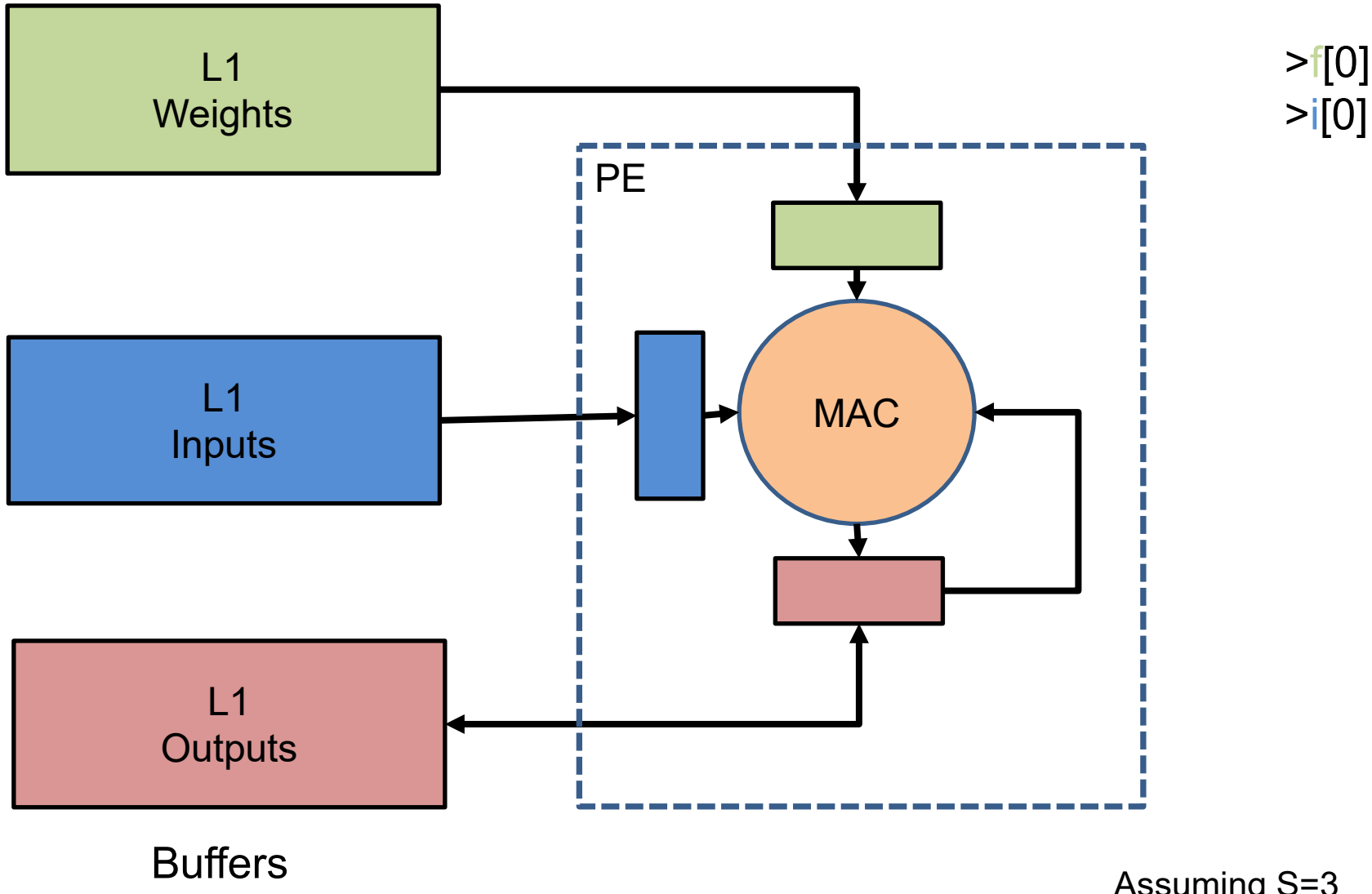
Single PE Output Stationary Flow



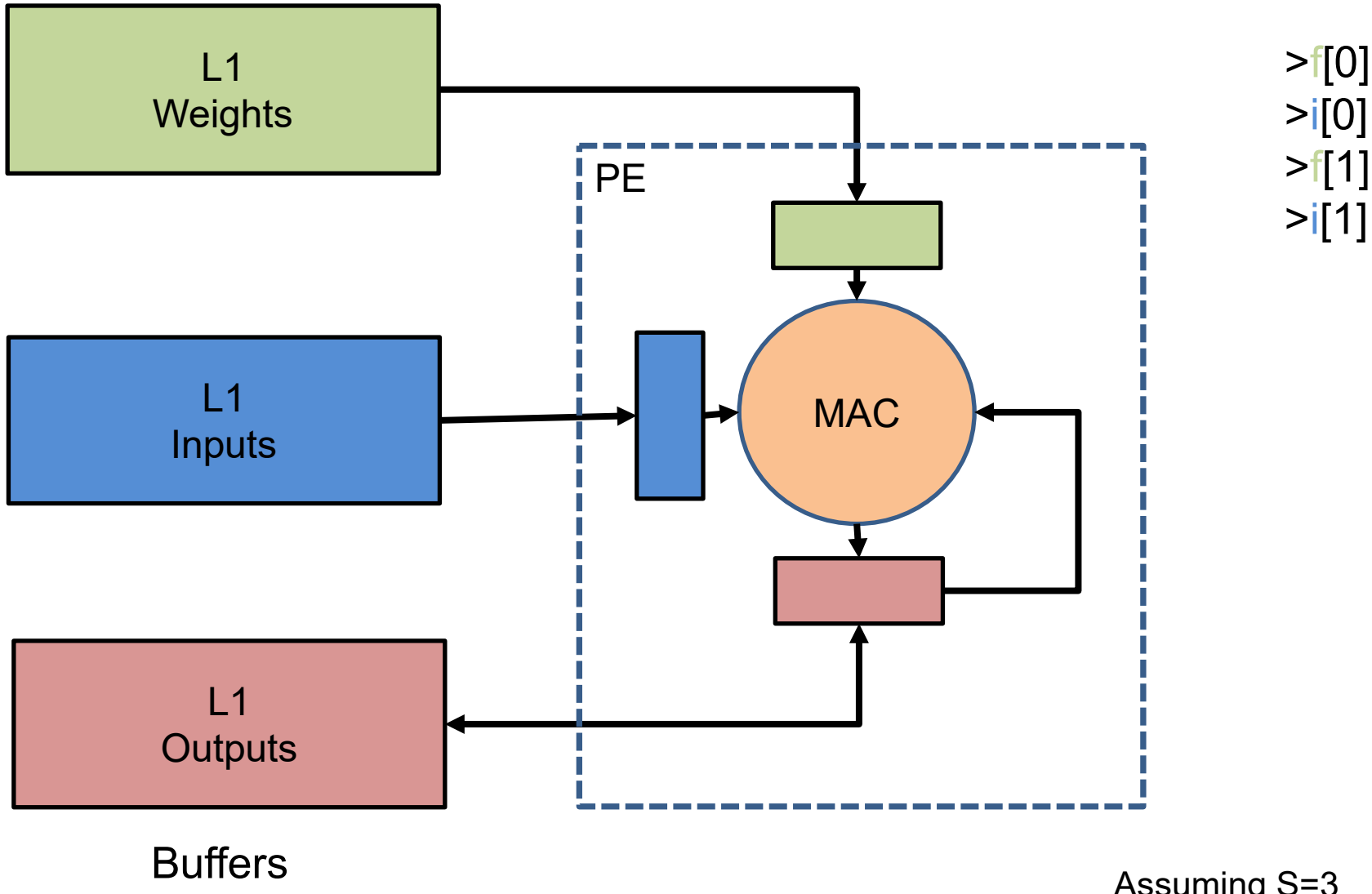
>f[0]

Assuming S=3

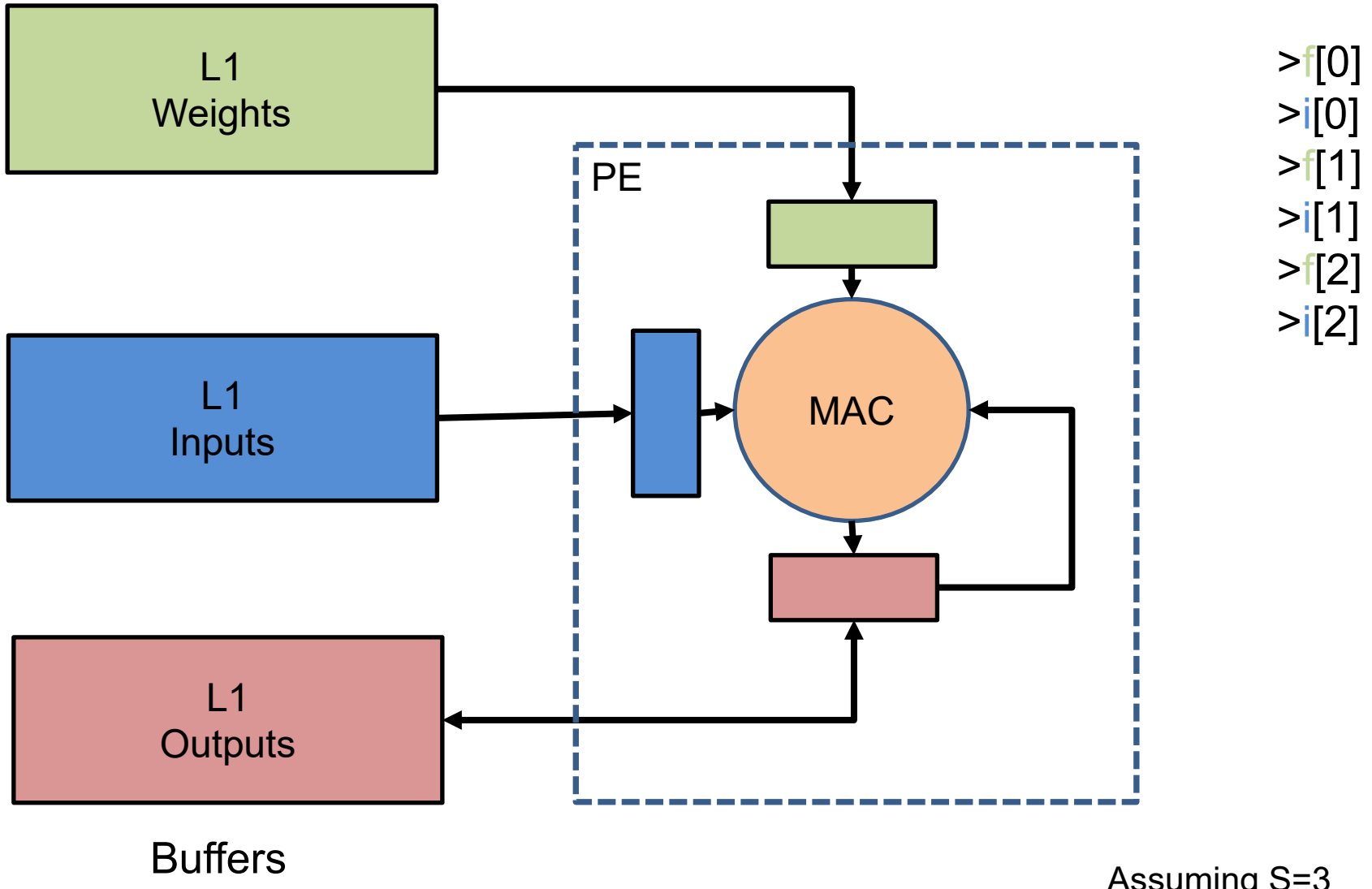
Single PE Output Stationary Flow



Single PE Output Stationary Flow

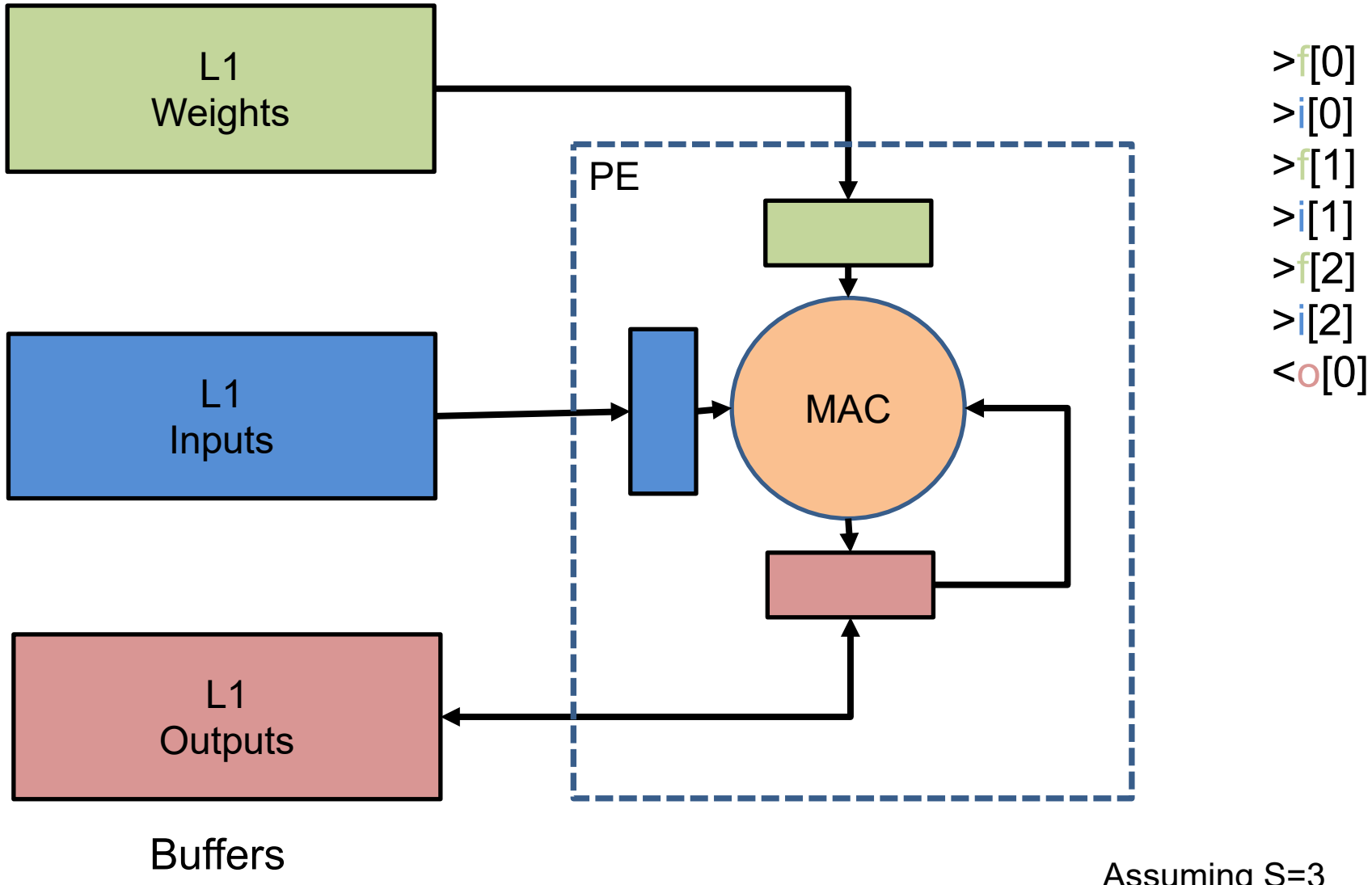


Single PE Output Stationary Flow

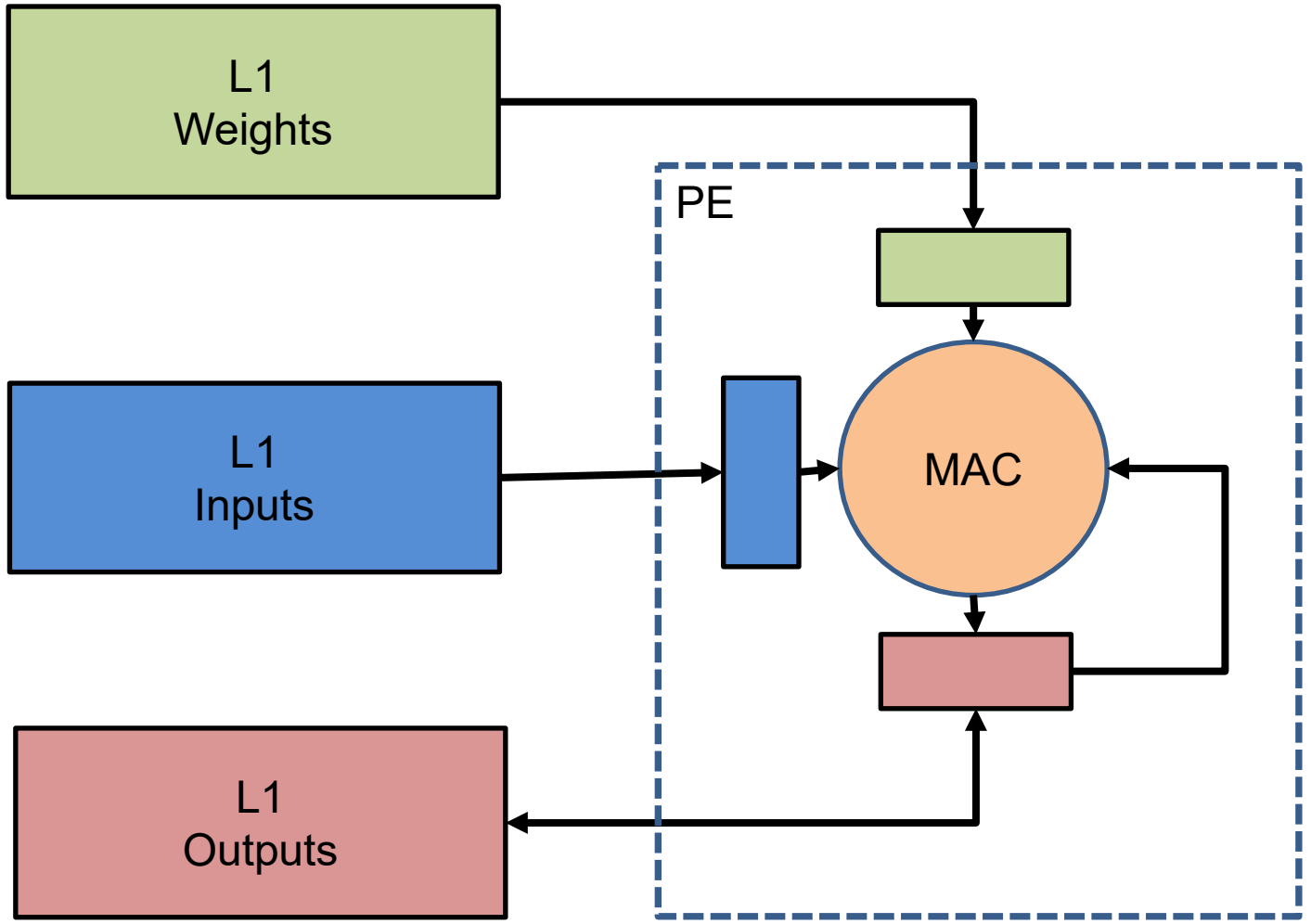


Assuming S=3

Single PE Output Stationary Flow



Single PE Output Stationary Flow

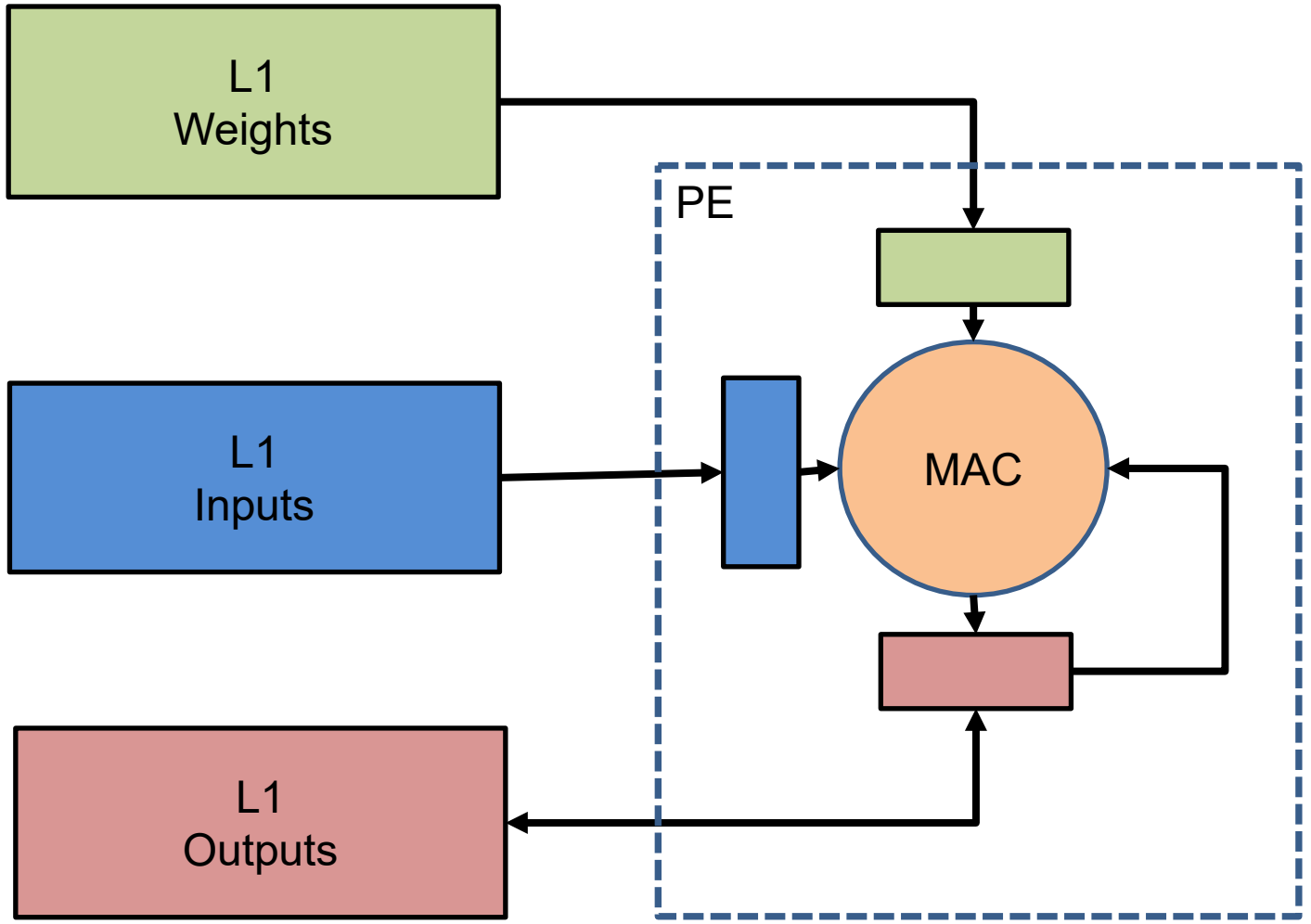


>f[0]
>i[0]
>f[1]
>i[1]
>f[2]
>i[2]
<o[0]

>f[0]
>i[1]

Assuming S=3

Single PE Output Stationary Flow

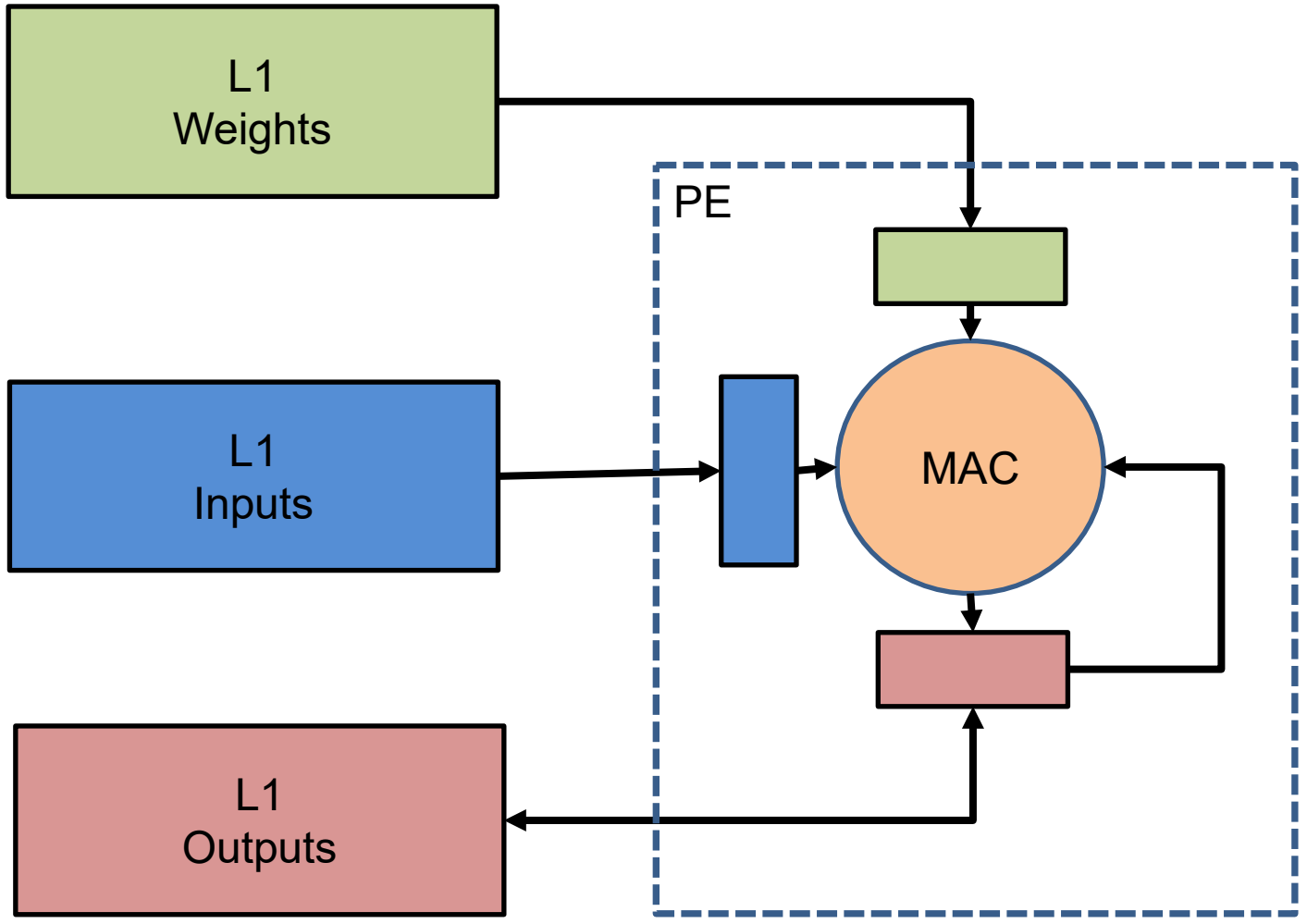


>f[0]
>i[0]
>f[1]
>i[1]
>f[2]
>i[2]
<o[0]

>f[0]
>i[1]
>f[1]
>i[2]

Assuming S=3

Single PE Output Stationary Flow

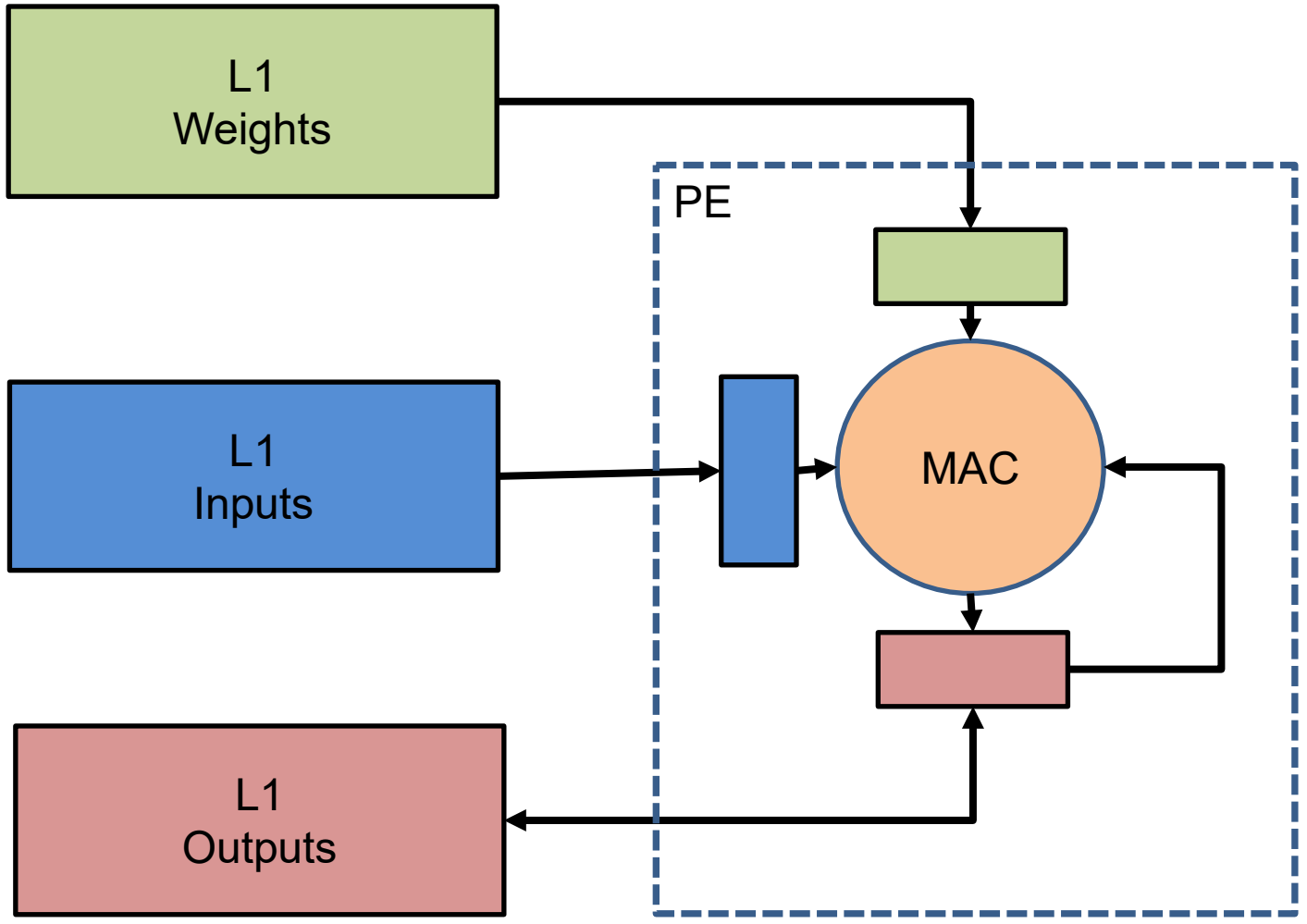


>f[0]
>i[0]
>f[1]
>i[1]
>f[2]
>i[2]
<o[0]

>f[0]
>i[1]
>f[1]
>i[2]
>f[2]
>i[3]
<o[1]

Assuming S=3

Single PE Output Stationary Flow

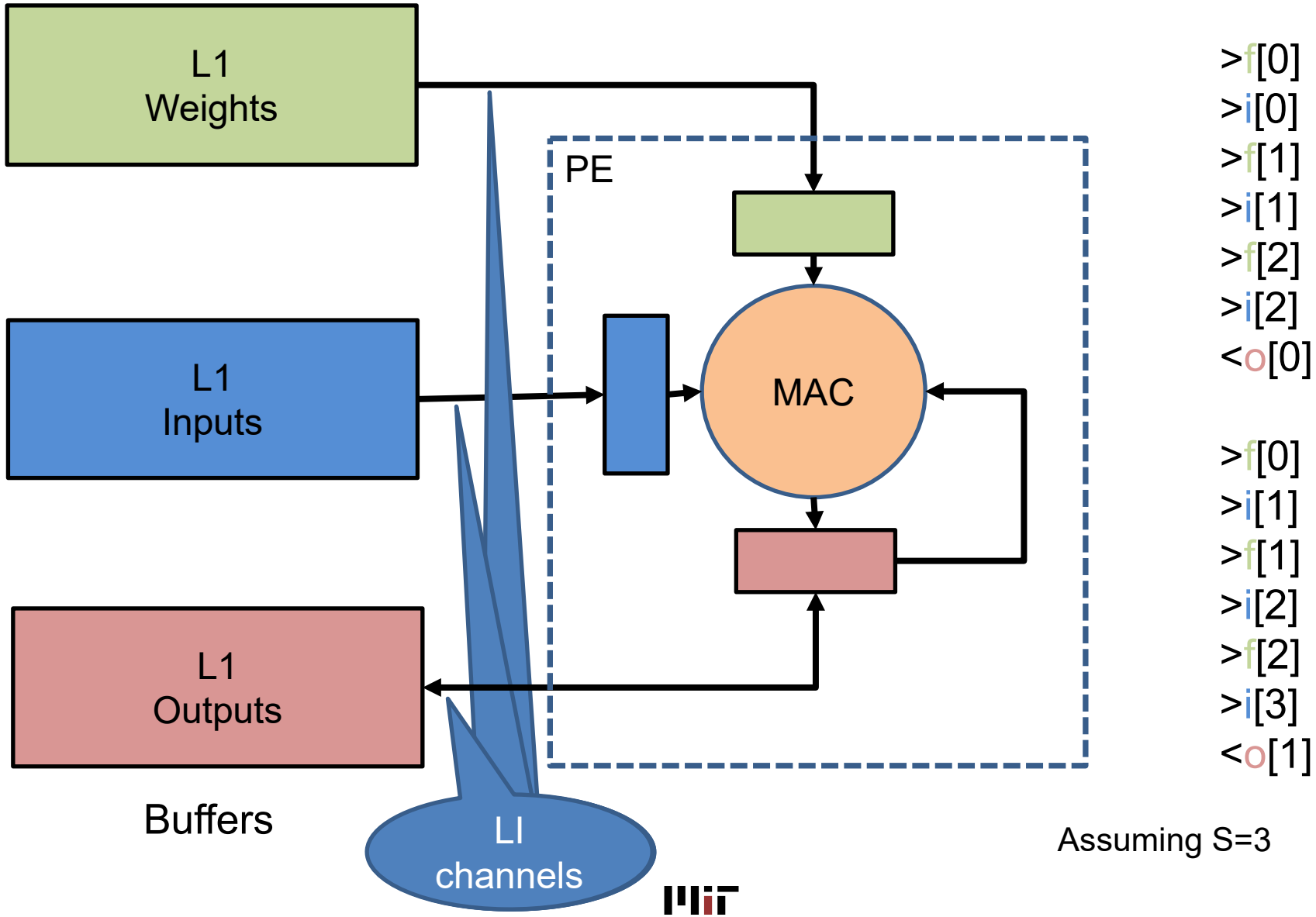


>f[0]
>i[0]
>f[1]
>i[1]
>f[2]
>i[2]
<o[0]

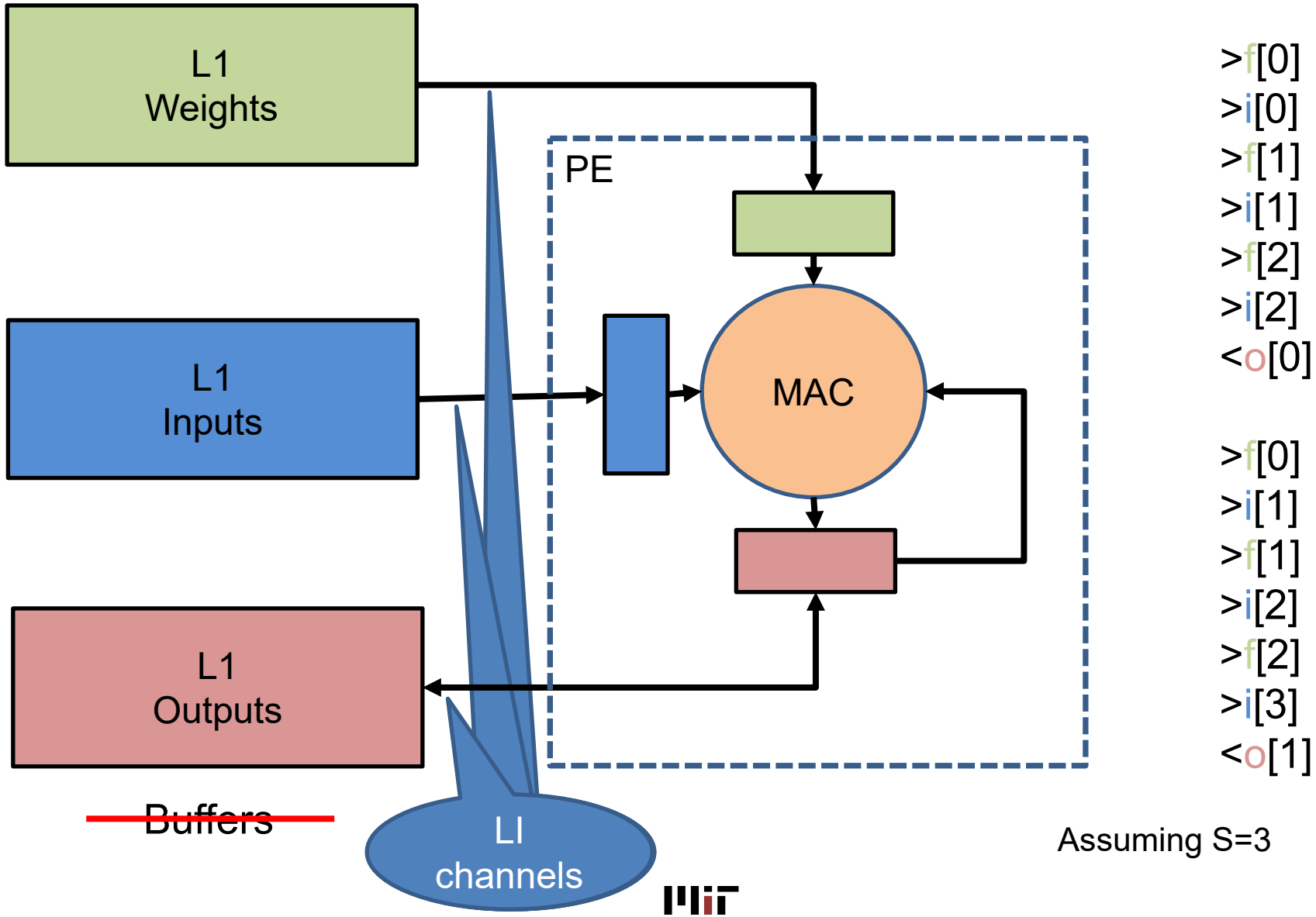
>f[0]
>i[1]
>f[1]
>i[2]
>f[2]
>i[3]
<o[1]

Assuming S=3

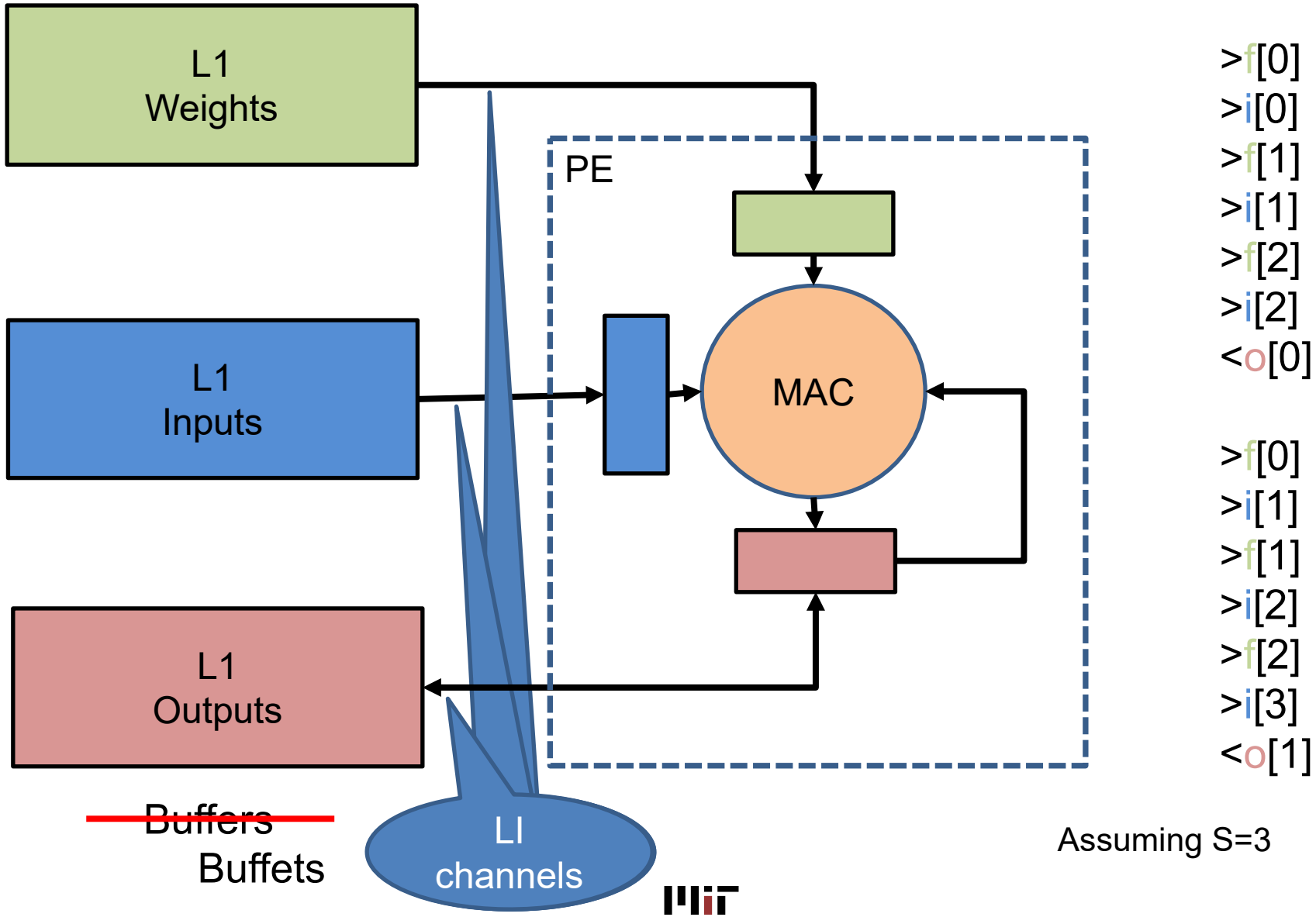
Single PE Output Stationary Flow



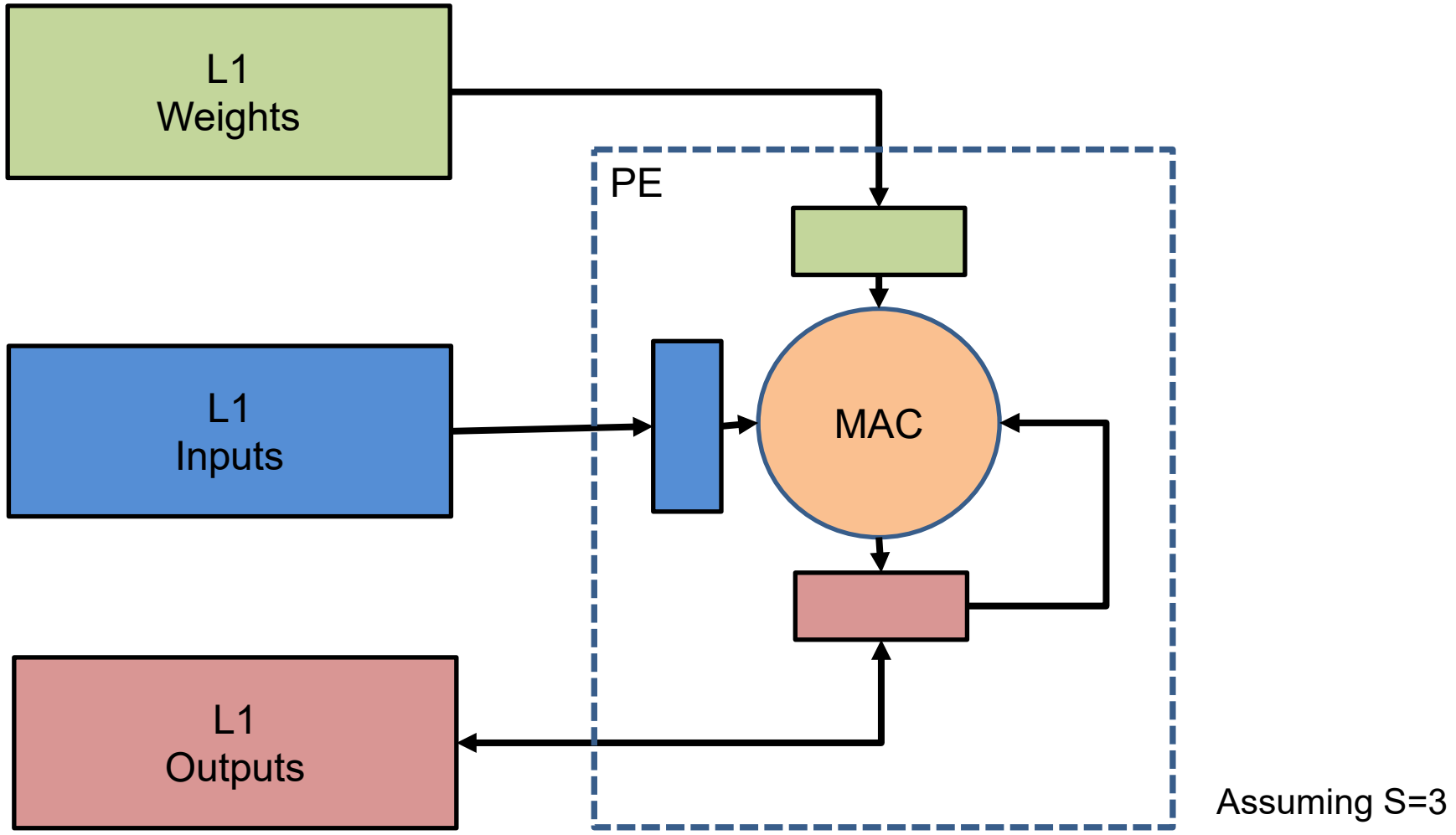
Single PE Output Stationary Flow



Single PE Output Stationary Flow



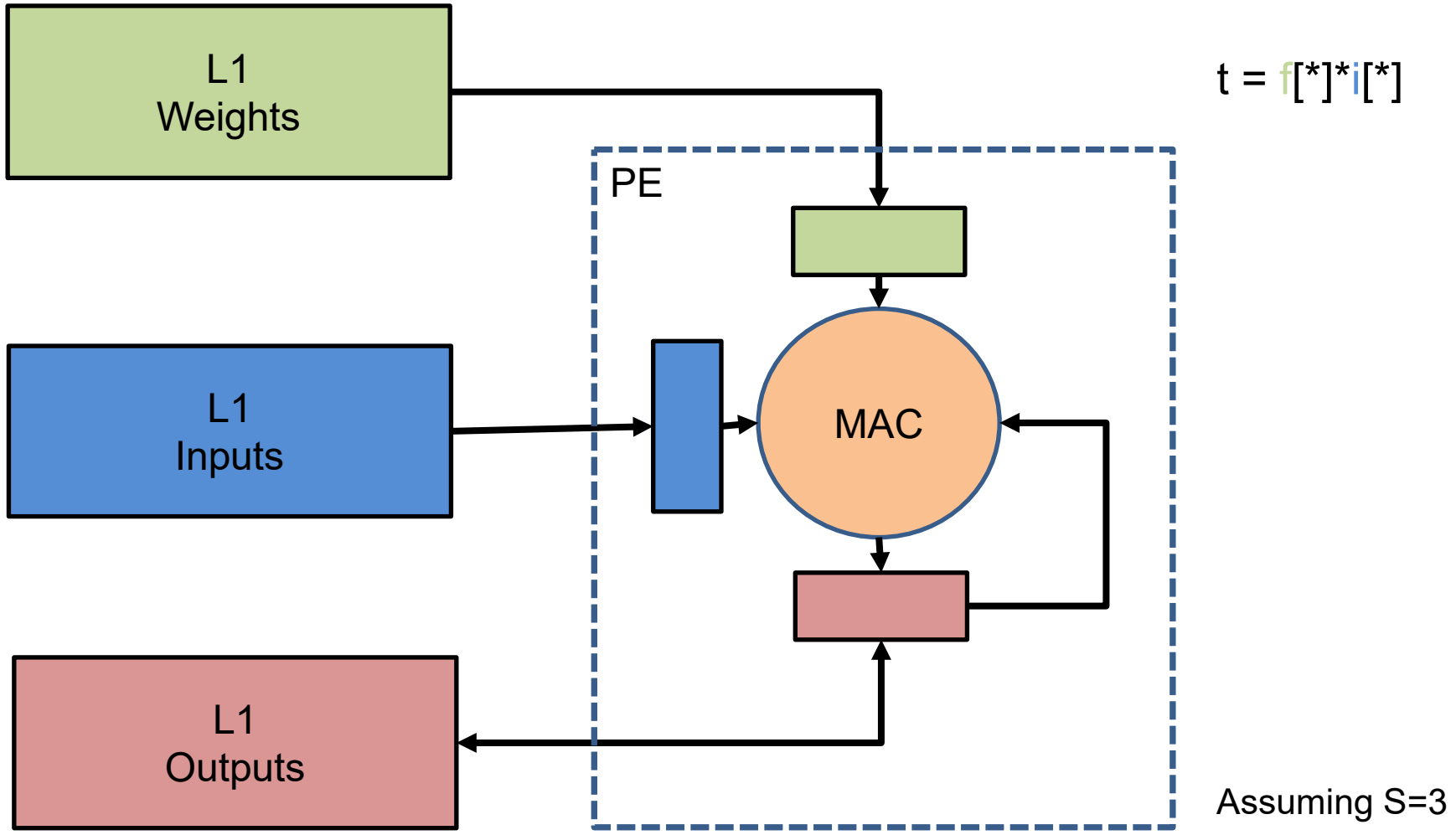
Single PE Output Stationary Flow



Buffets

x[*] is input or output on LI channel stream

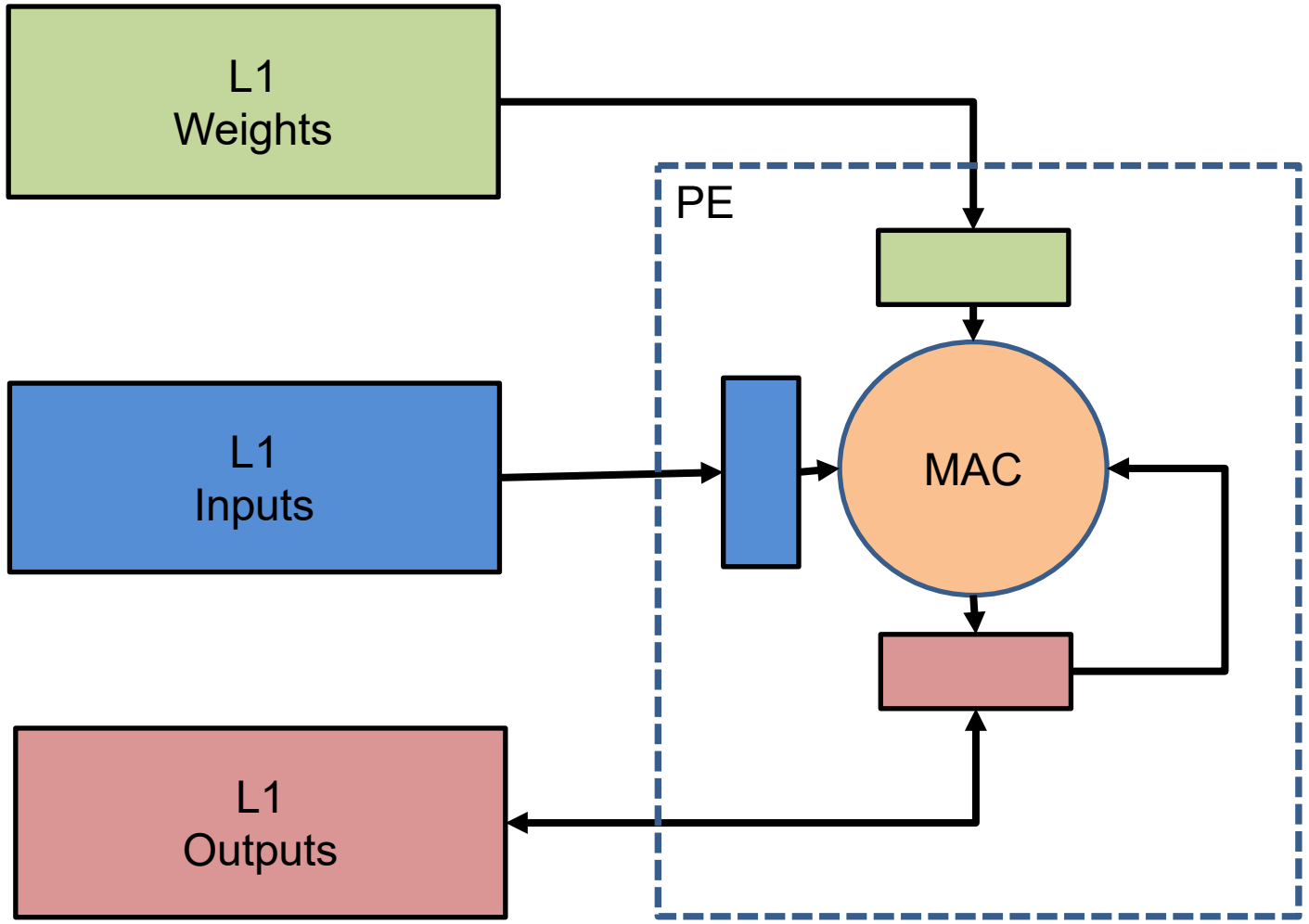
Single PE Output Stationary Flow



Buffets

x[*] is input or output on LI channel stream

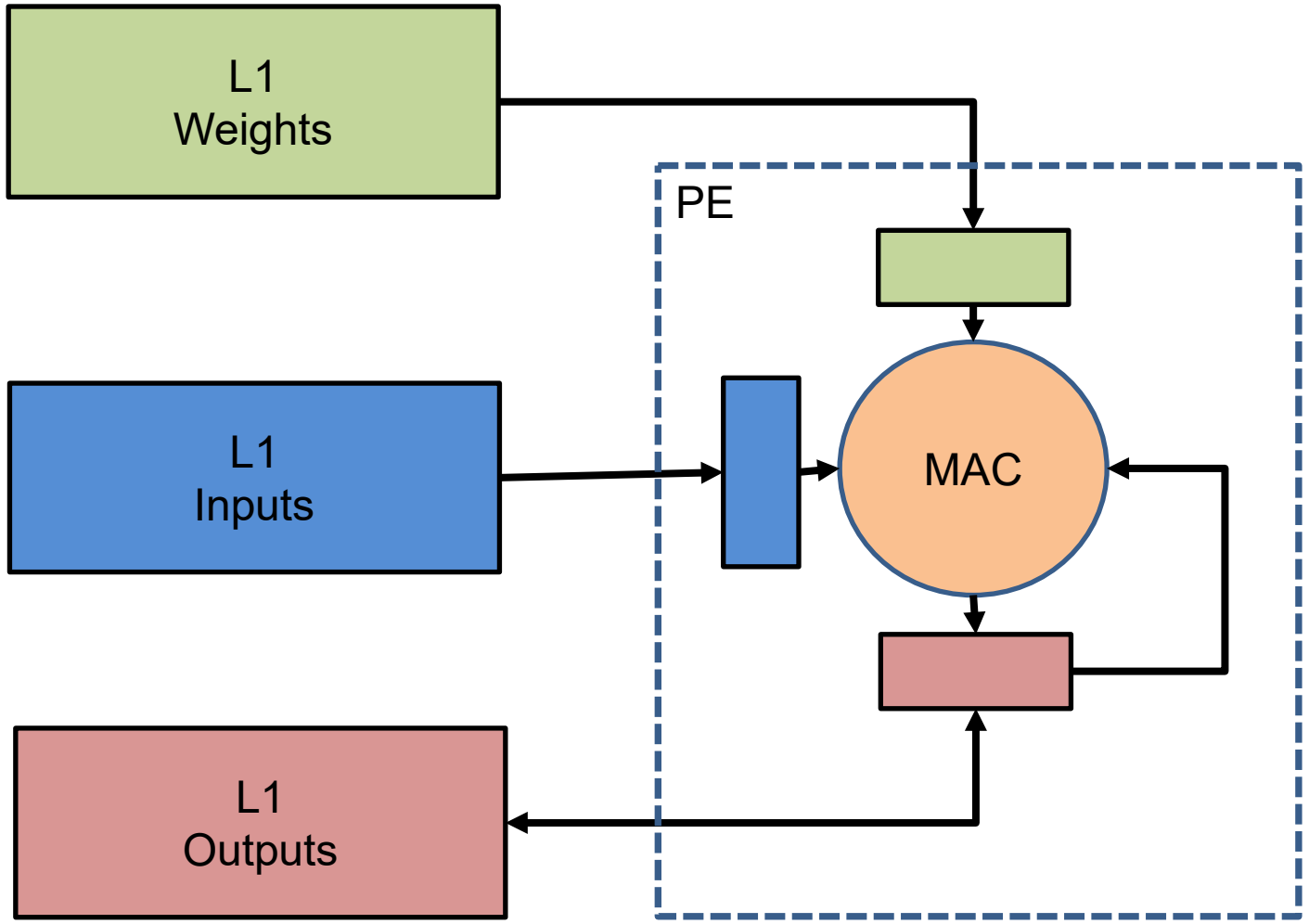
Single PE Output Stationary Flow



Buffets

x[*] is input or output on LI channel stream

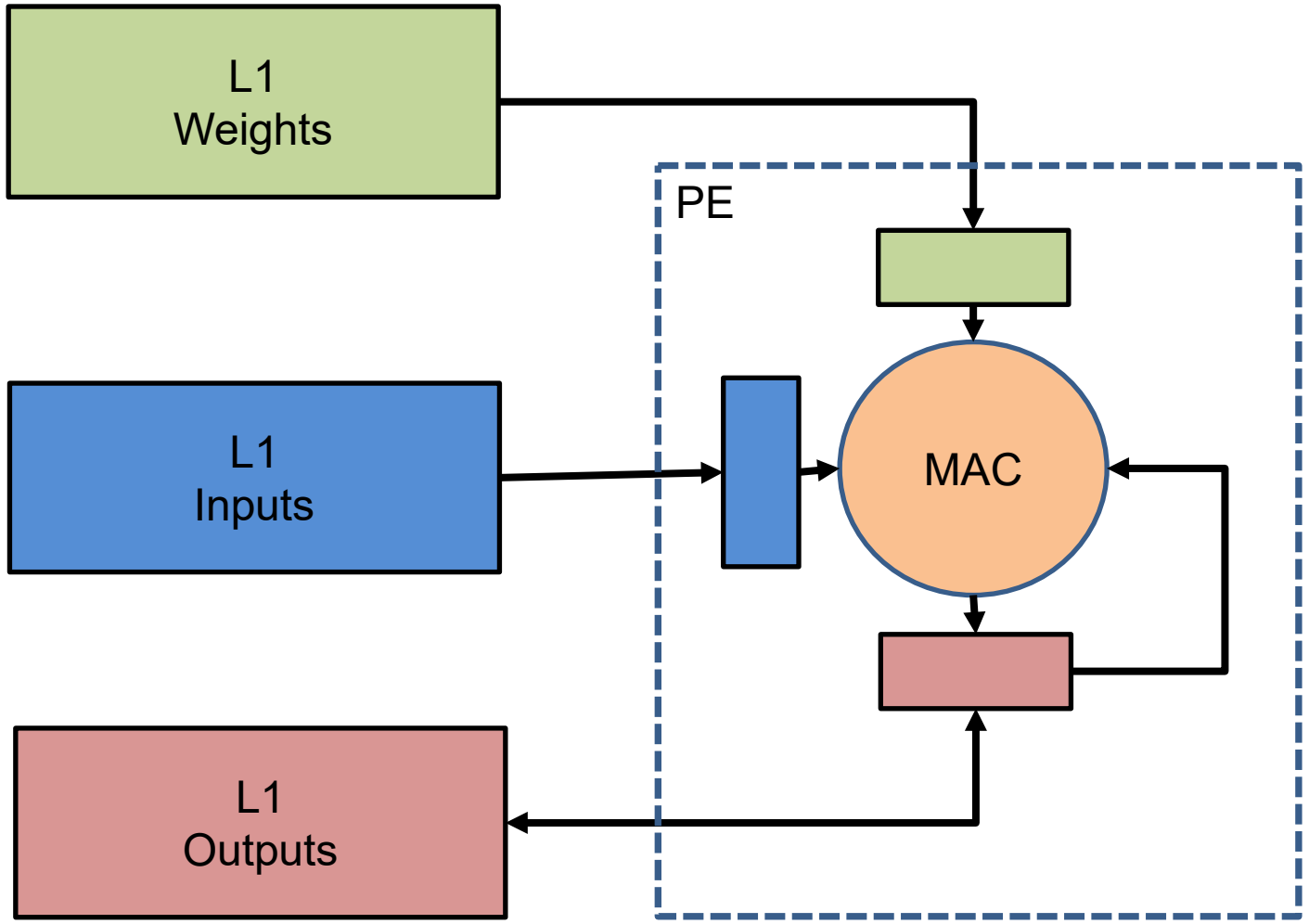
Single PE Output Stationary Flow



Buffets

x[*] is input or output on LI channel stream

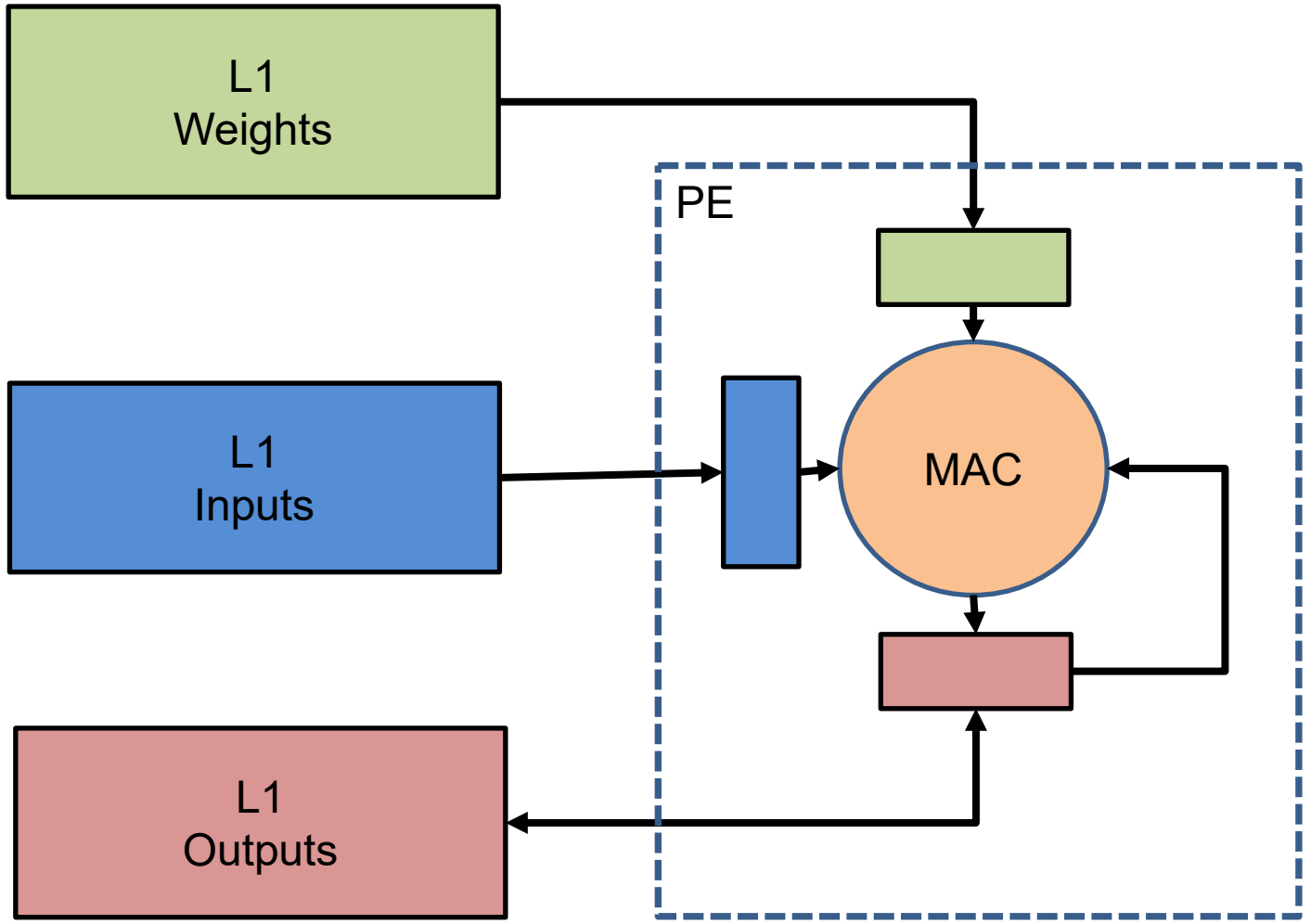
Single PE Output Stationary Flow



Buffets

x[*] is input or output on LI channel stream

Single PE Output Stationary Flow



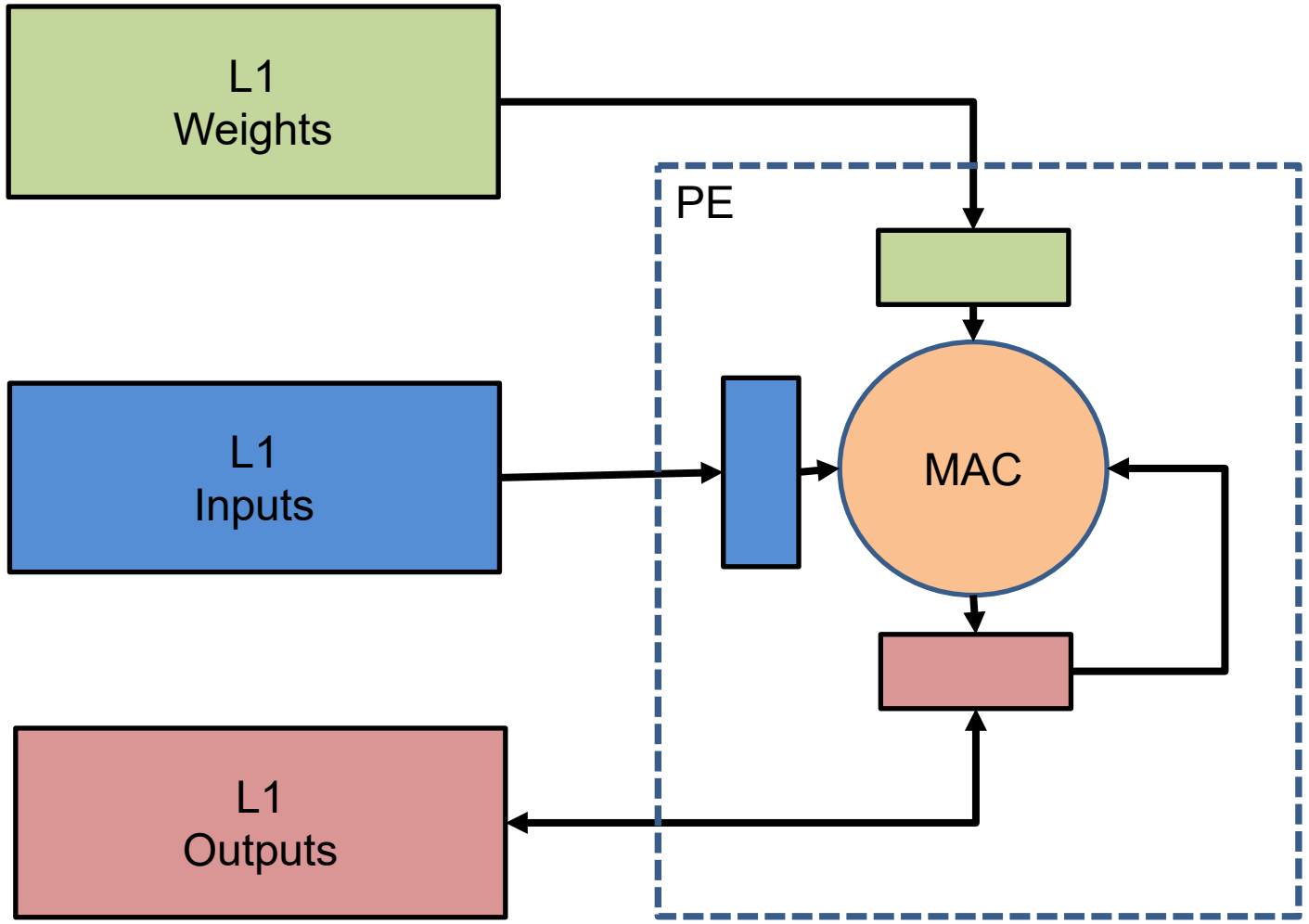

```
t = f[*]*i[*]
t += f[*]*i[*]
t += f[*]*i[*]
o[*] = t

t = f[*]*i[*]
```

Assuming S=3

Buffets

x[*] is input or output on LI channel stream

Single PE Output Stationary Flow

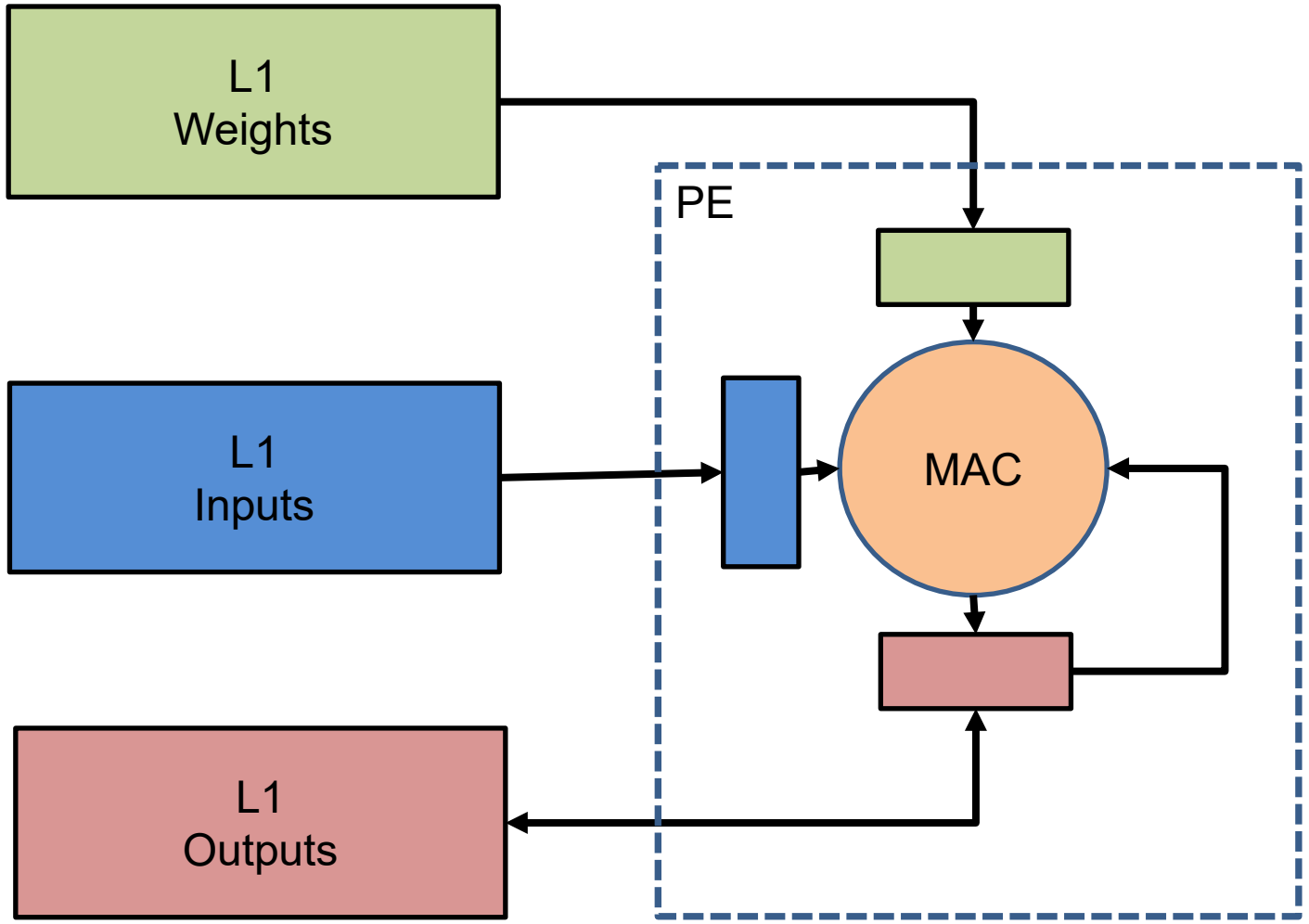



```
t = f[*]*i[*]
t += f[*]*i[*]
t += f[*]*i[*]
o[*] = t

t = f[*]*i[*]
t += f[*]*i[*]
```

Assuming S=3

x[*] is input or output on LI channel stream

Single PE Output Stationary Flow

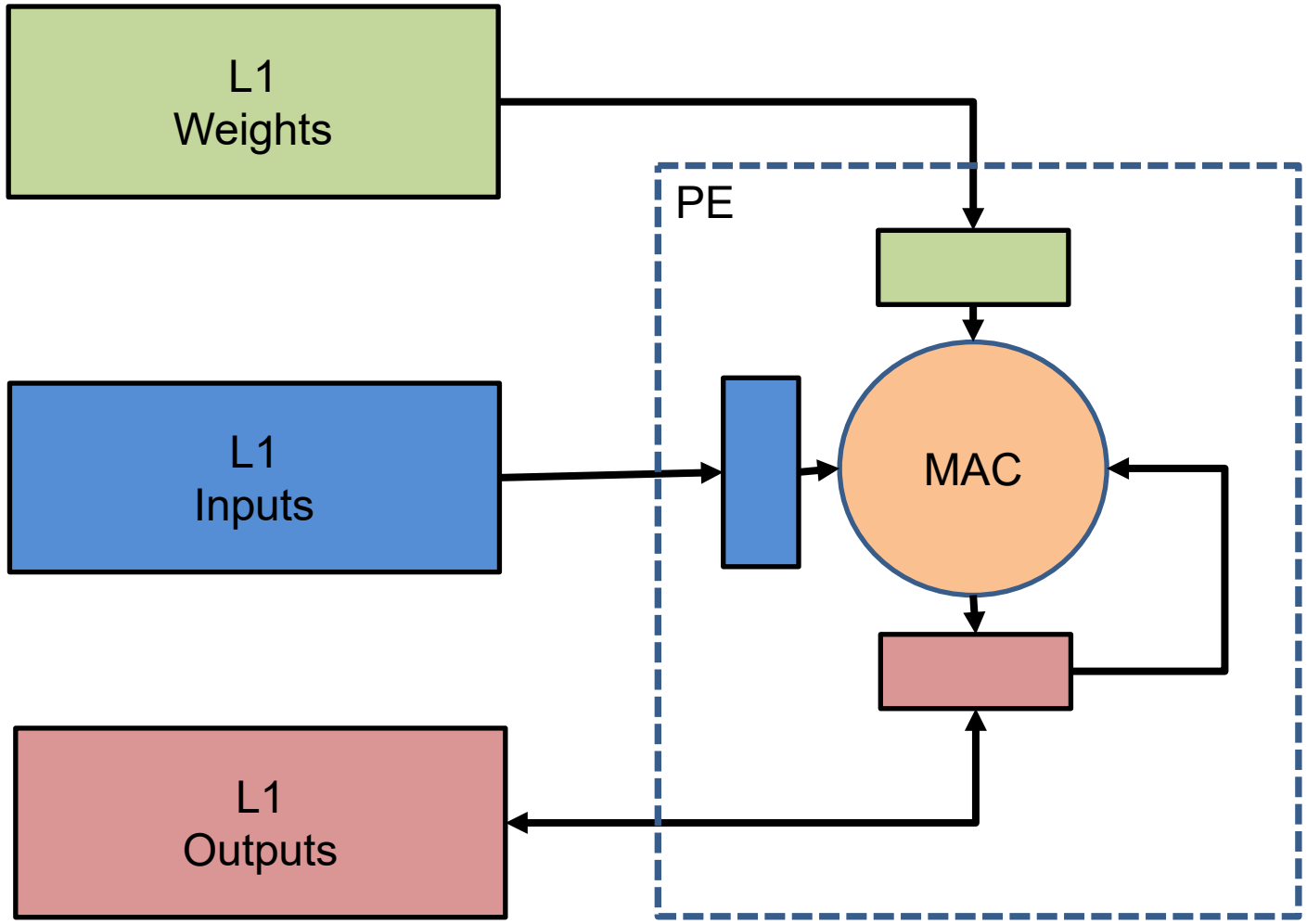



```
t = f[*]*i[*]
t += f[*]*i[*]
t += f[*]*i[*]
o[*] = t

t = f[*]*i[*]
t += f[*]*i[*]
t += f[*]*i[*]
```

Assuming S=3

x[*] is input or output on LI channel stream

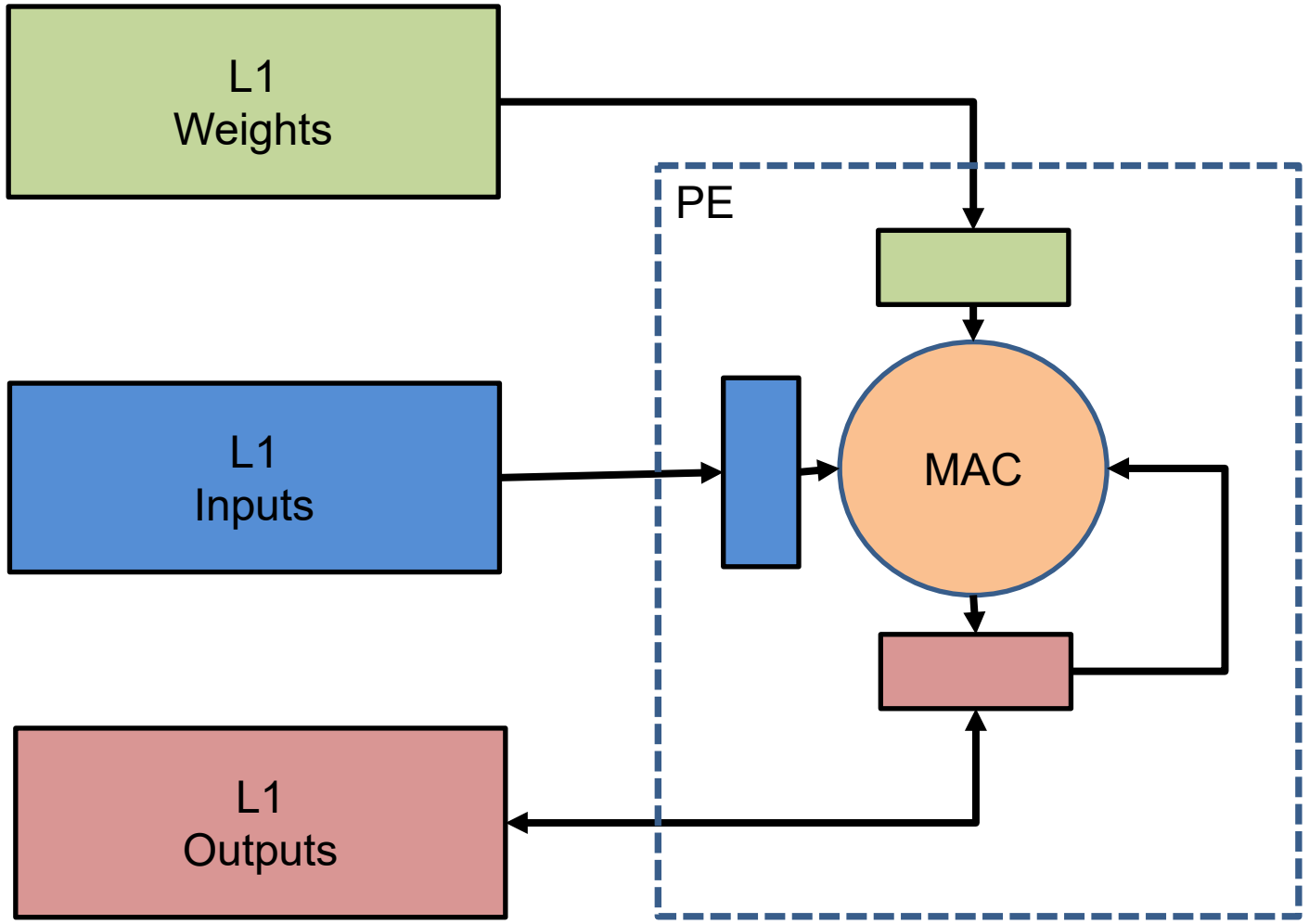
Single PE Output Stationary Flow



Buffets

x[*] is input or output on LI channel stream

Single PE Output Stationary Flow




```
t = f[*]*i[*]
t += f[*]*i[*]
t += f[*]*i[*]
o[*] = t

t = f[*]*i[*]
t += f[*]*i[*]
t += f[*]*i[*]
o[*] = t

•
•
•
```

Assuming S=3

x[*] is input or output on LI channel stream

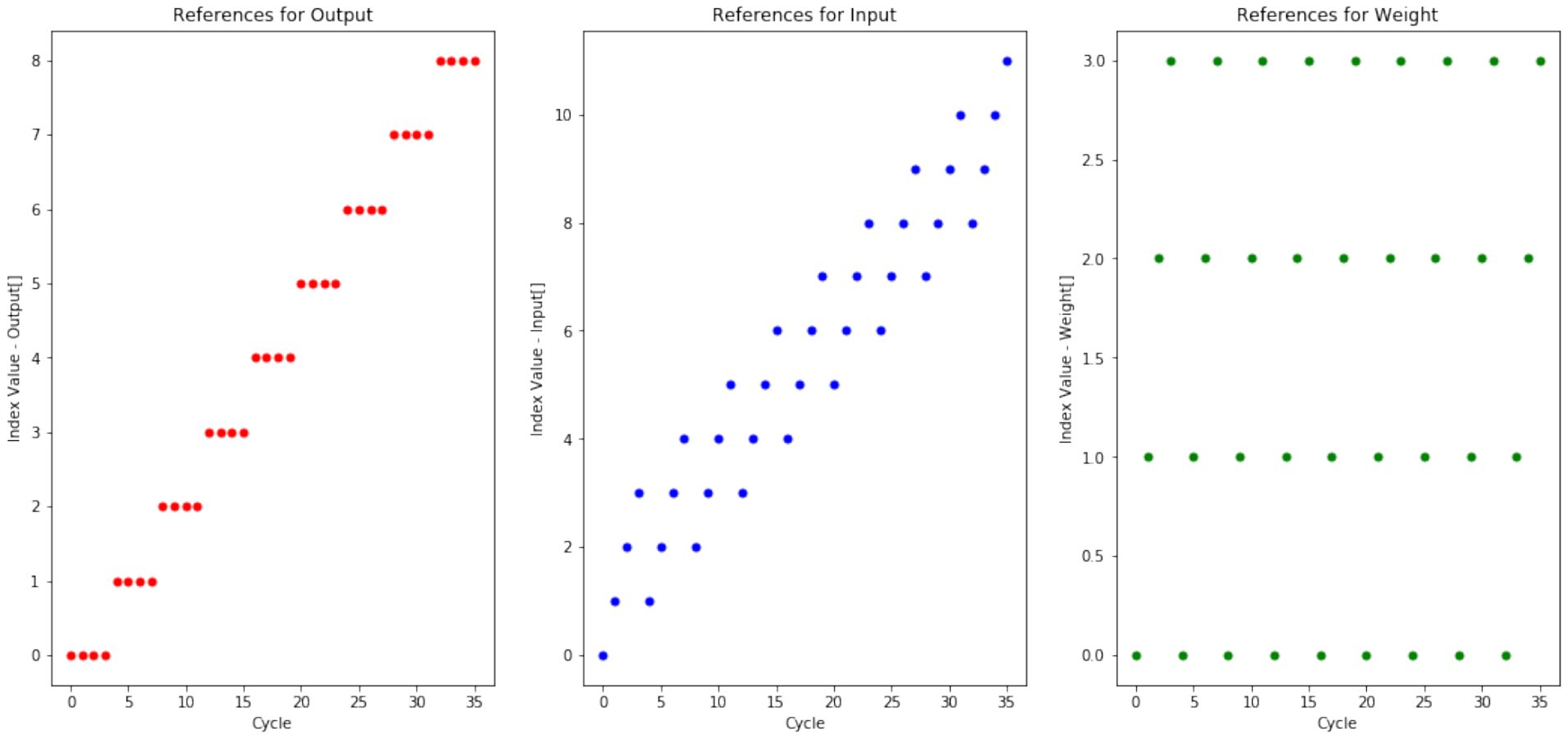
LI Channel-based Buffer vs Scratchpads

- PE does not generate addresses, i.e., no load or store address calculations
- Address generation is not serialized with arithmetic operations, e.g., as loads or stores
- PE does not need register target for each scratchpad request in flight
- If the channel operations are guaranteed never to block then the channel logic can be optimized away and the reads/writes can happen systolically.

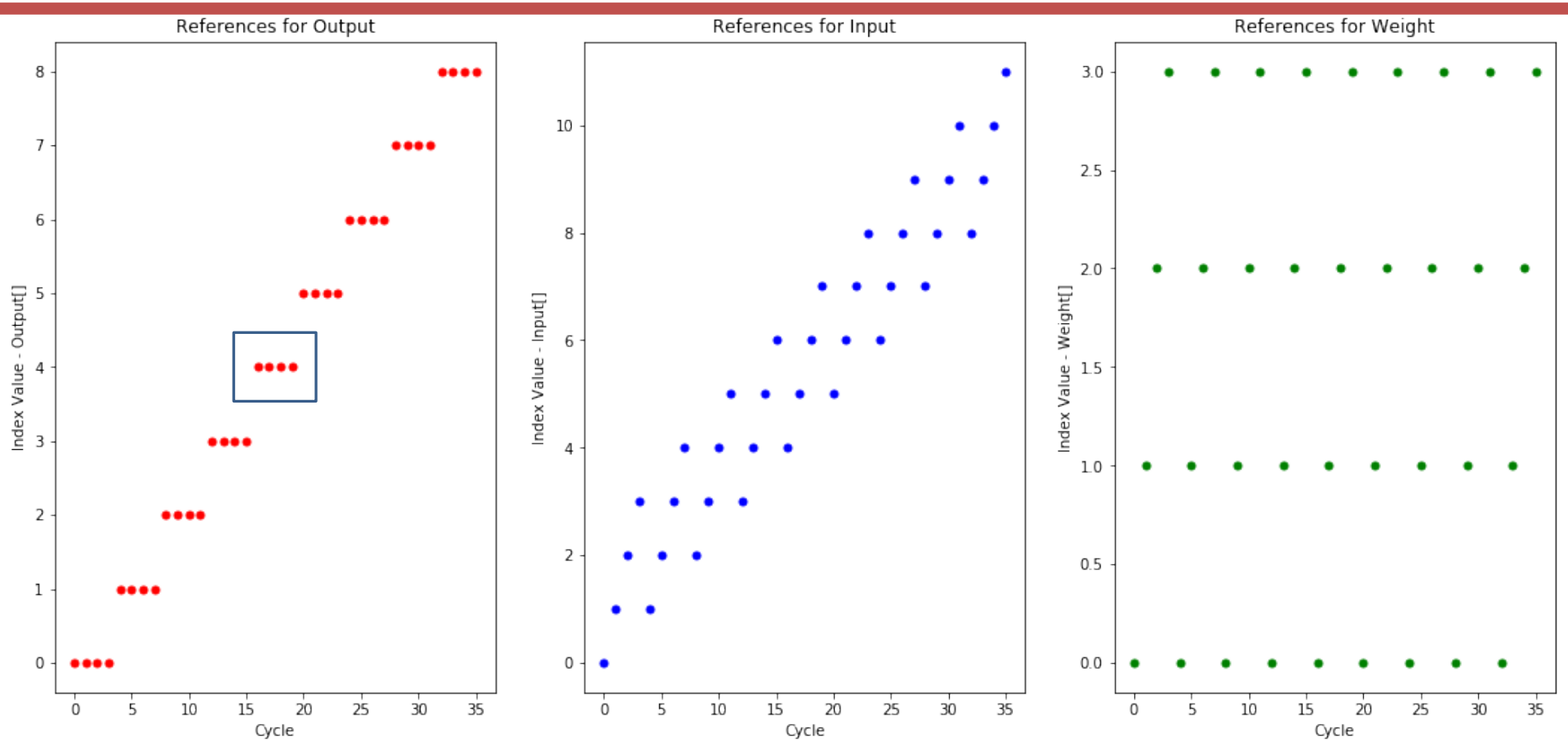
Output Stationary – Reference Pattern

```
for q in [0, Q):
  for s in [0, S):
    o[q] += i[q+s]*f[s]
```

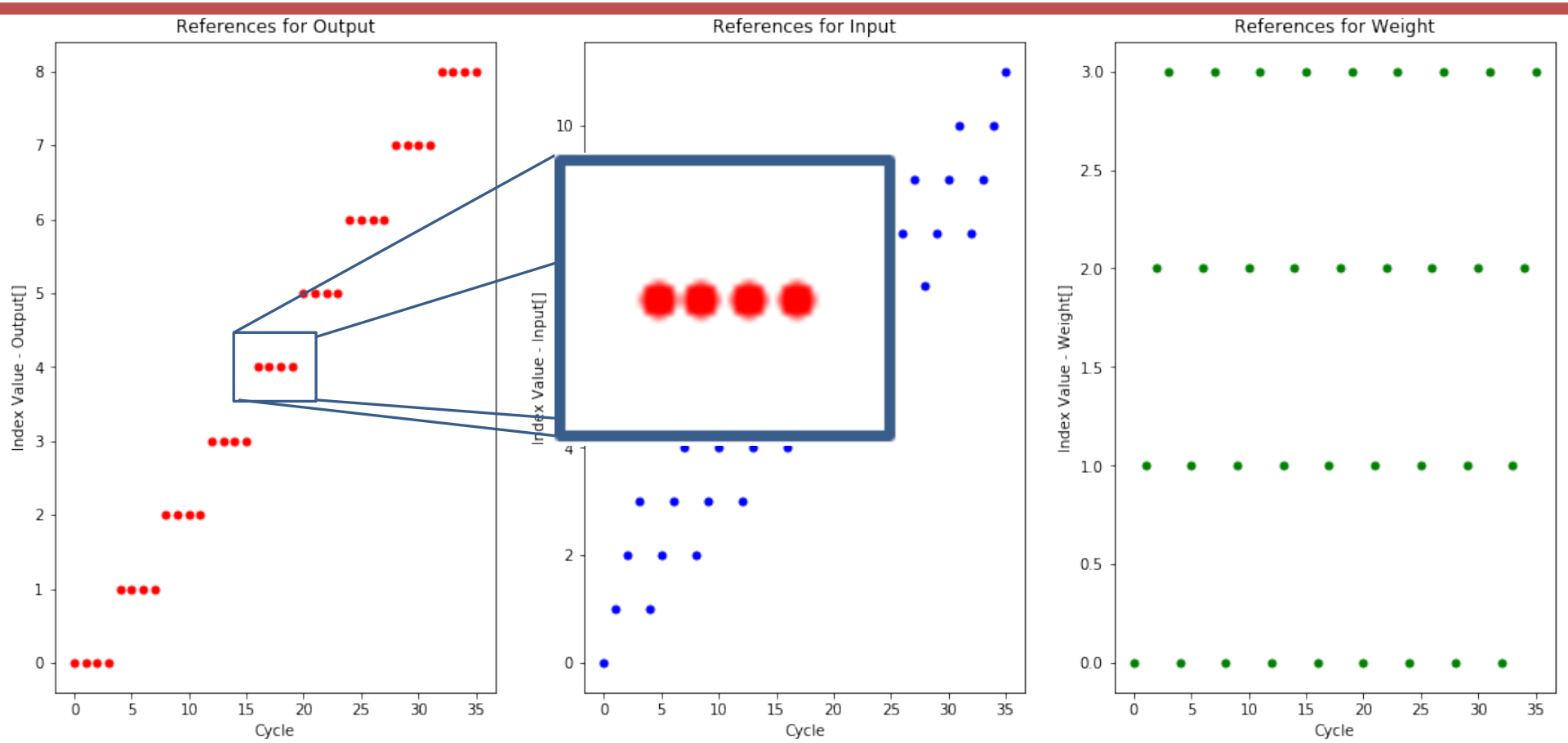
Layer Shape:
- S = 4
- Q = 9
- W = 12



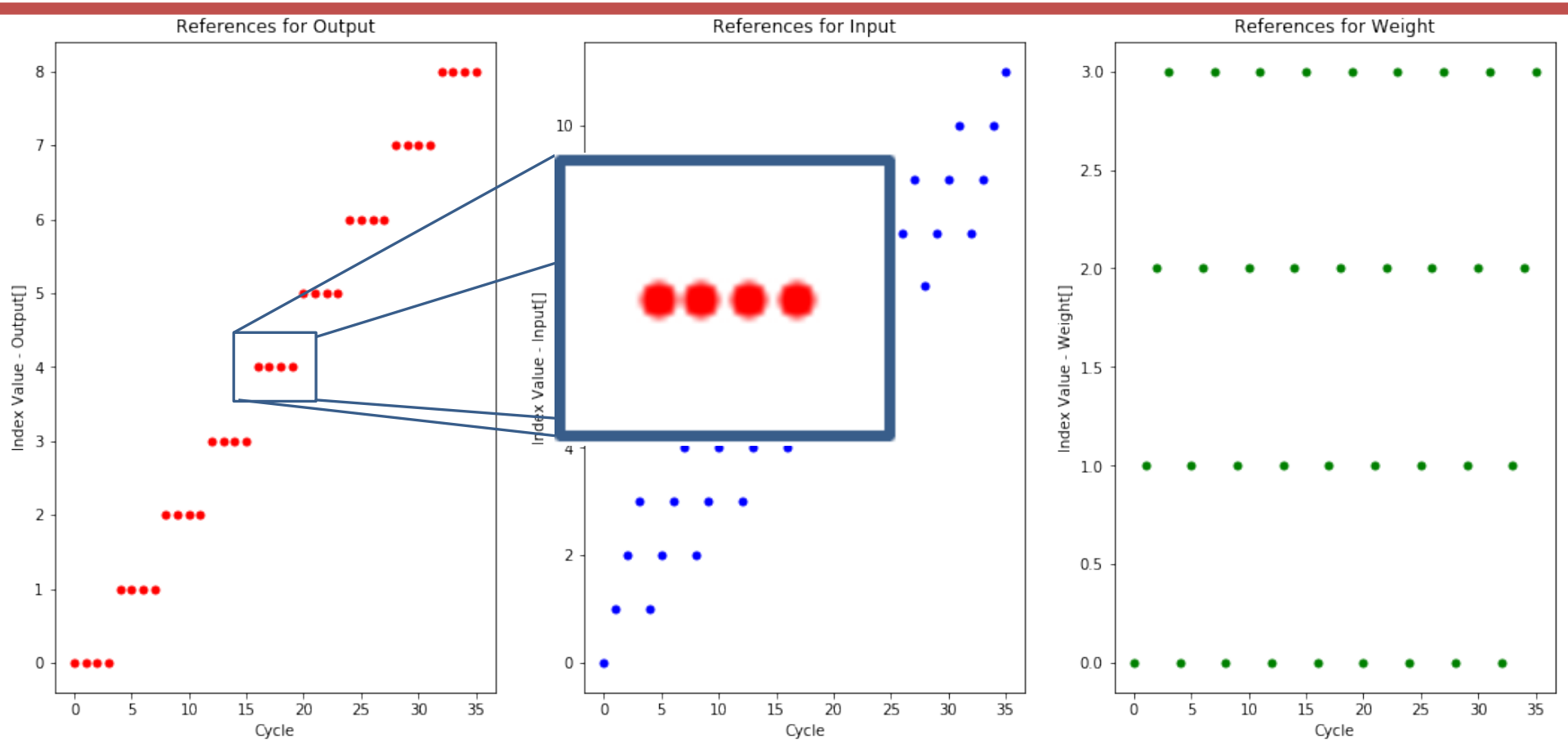
Output Stationary – Reference Pattern



Output Stationary – Reference Pattern

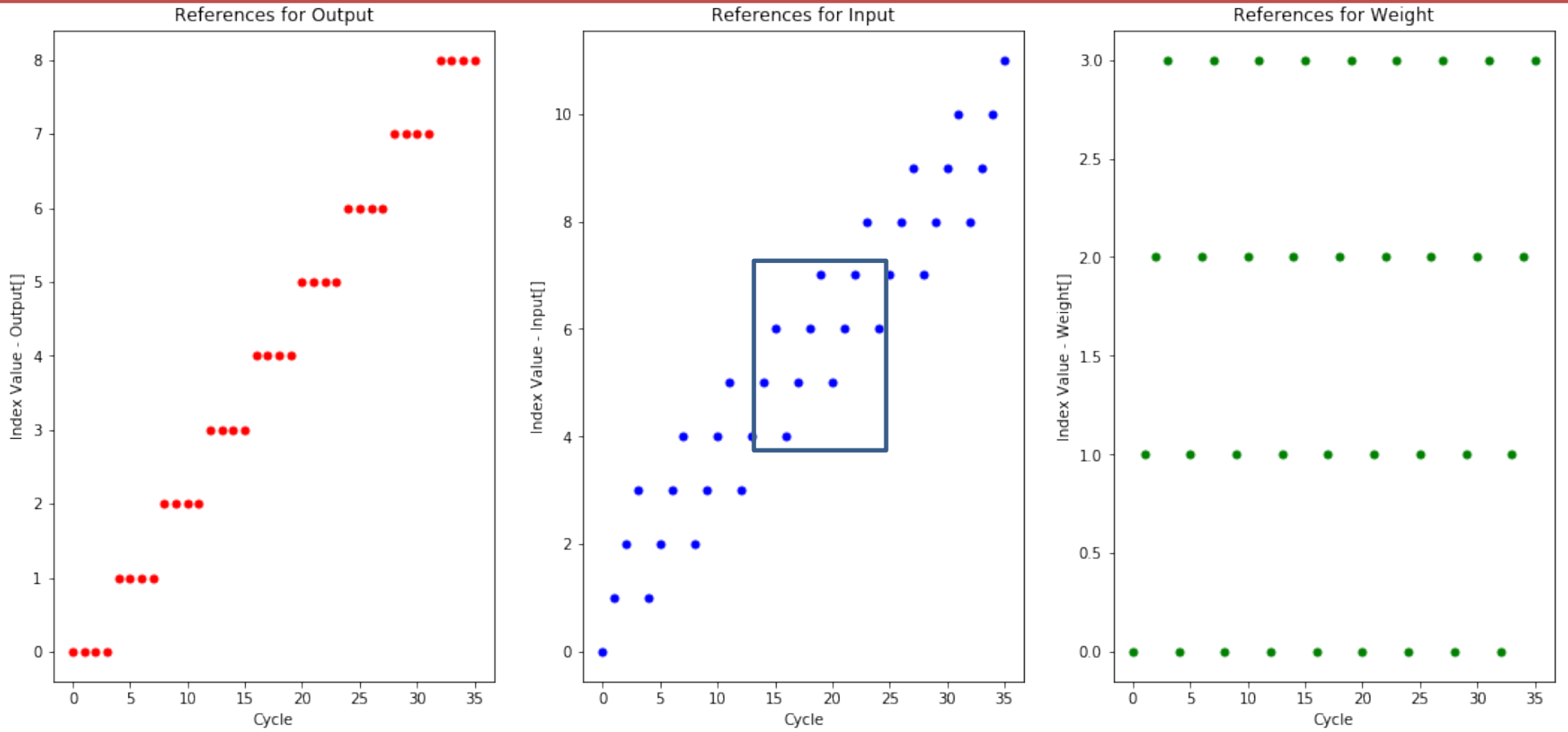


Output Stationary – Reference Pattern



Observations: - Single output is reused many times (S)

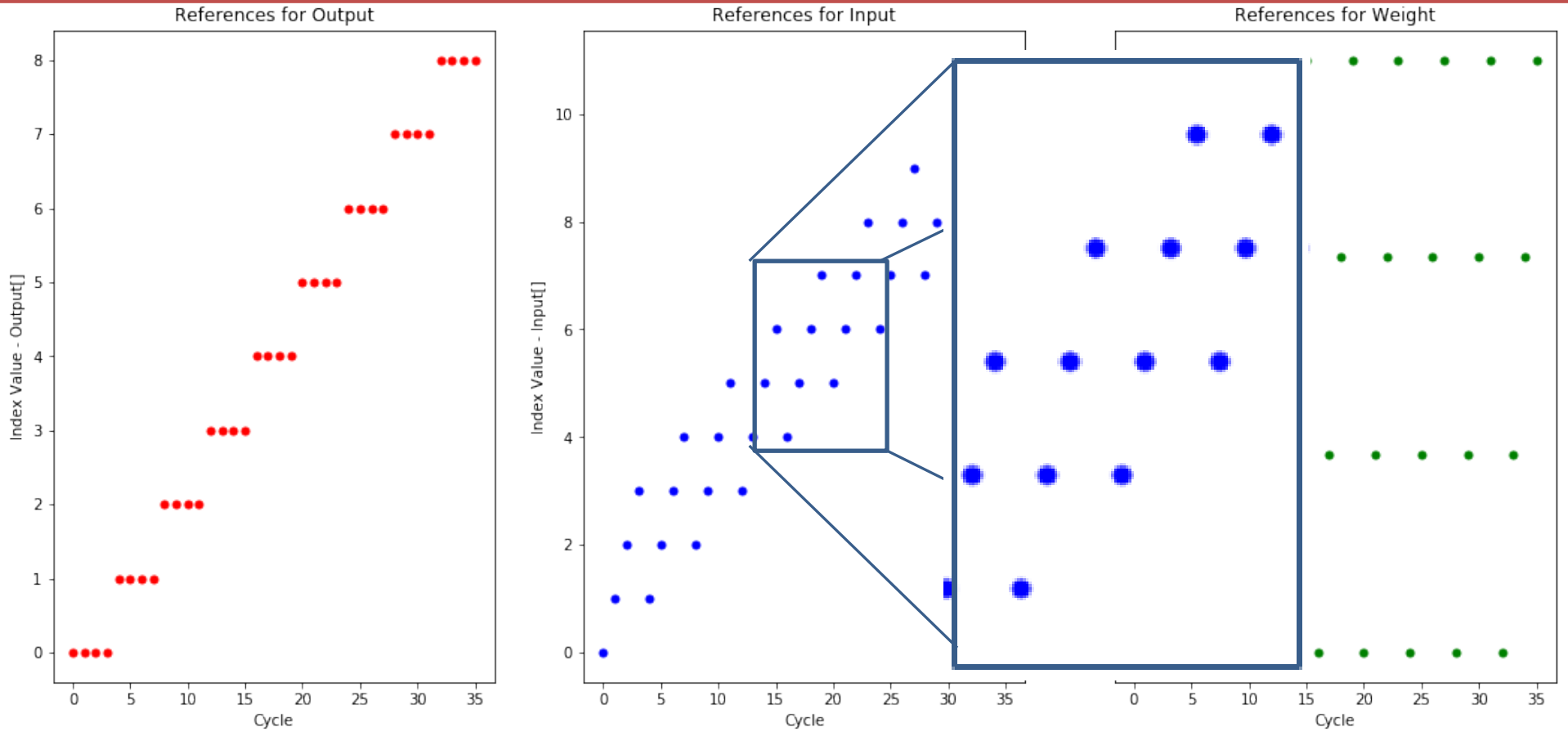
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

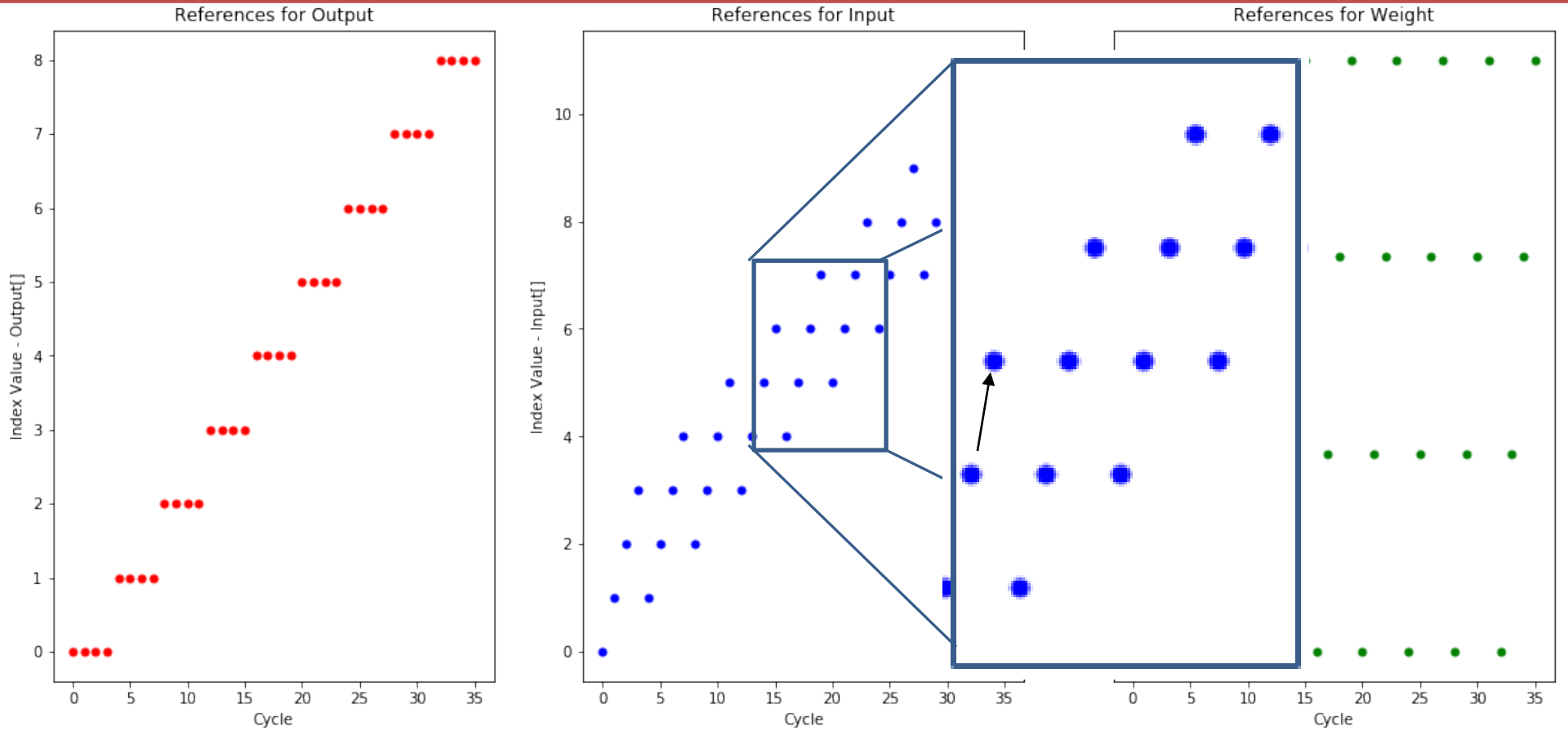
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

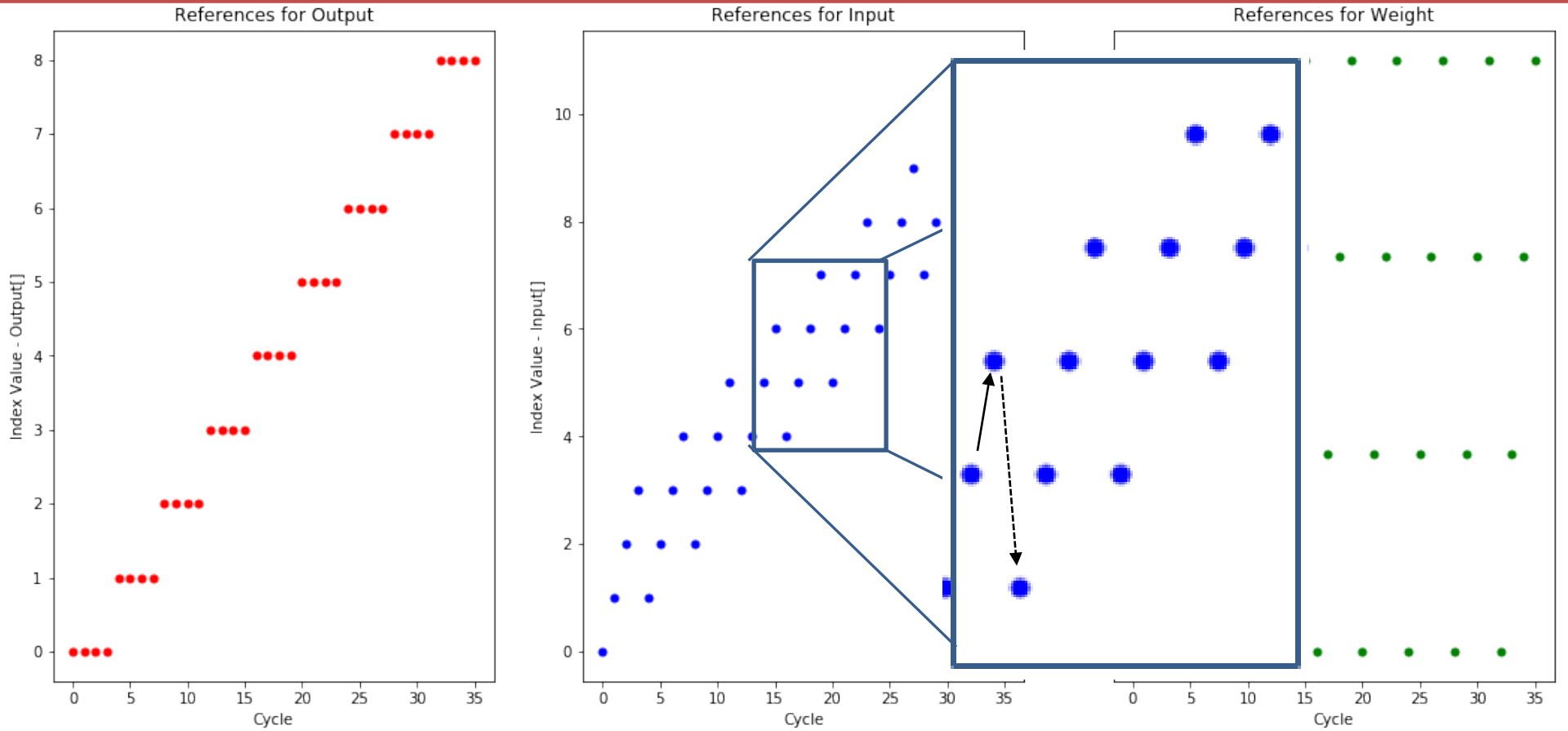
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

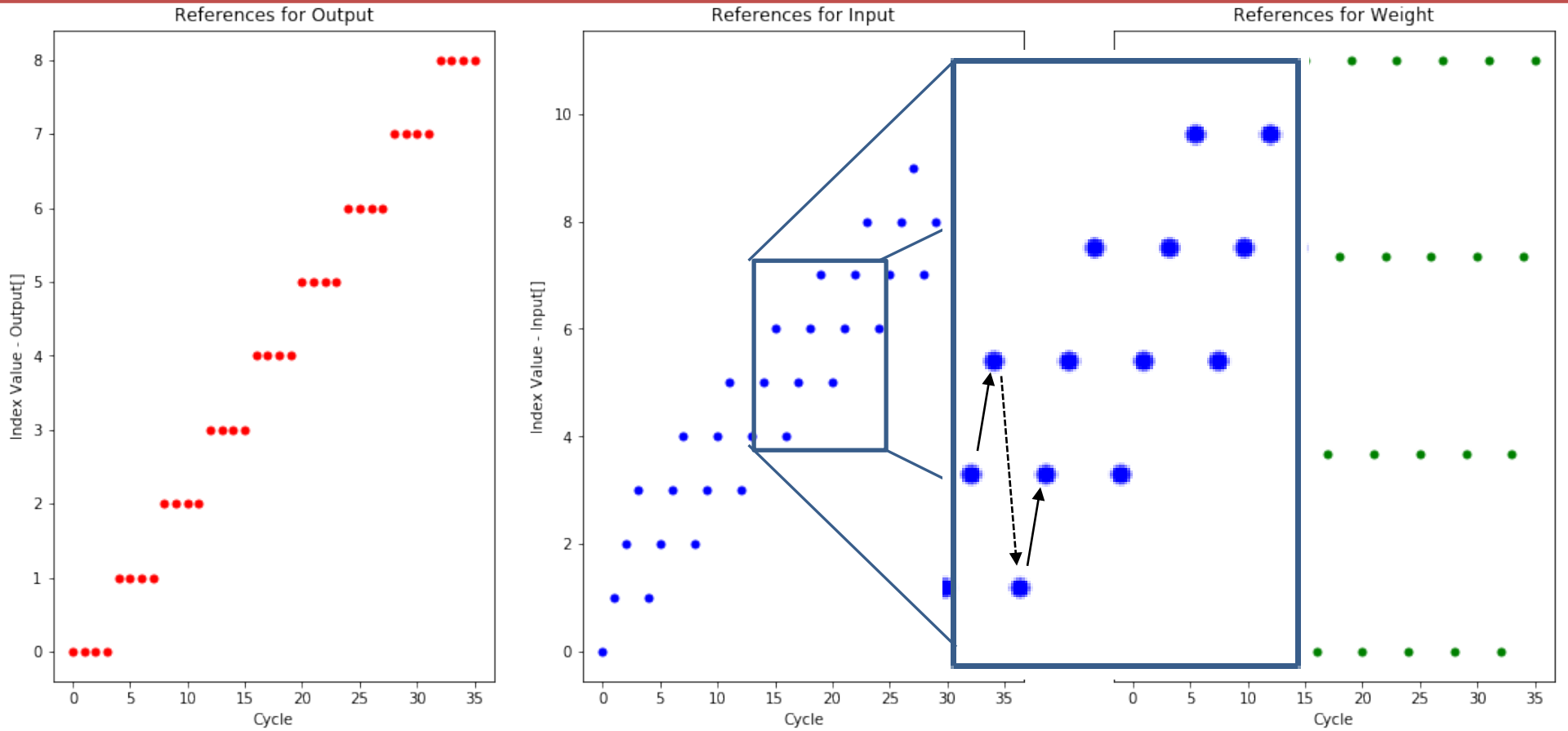
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

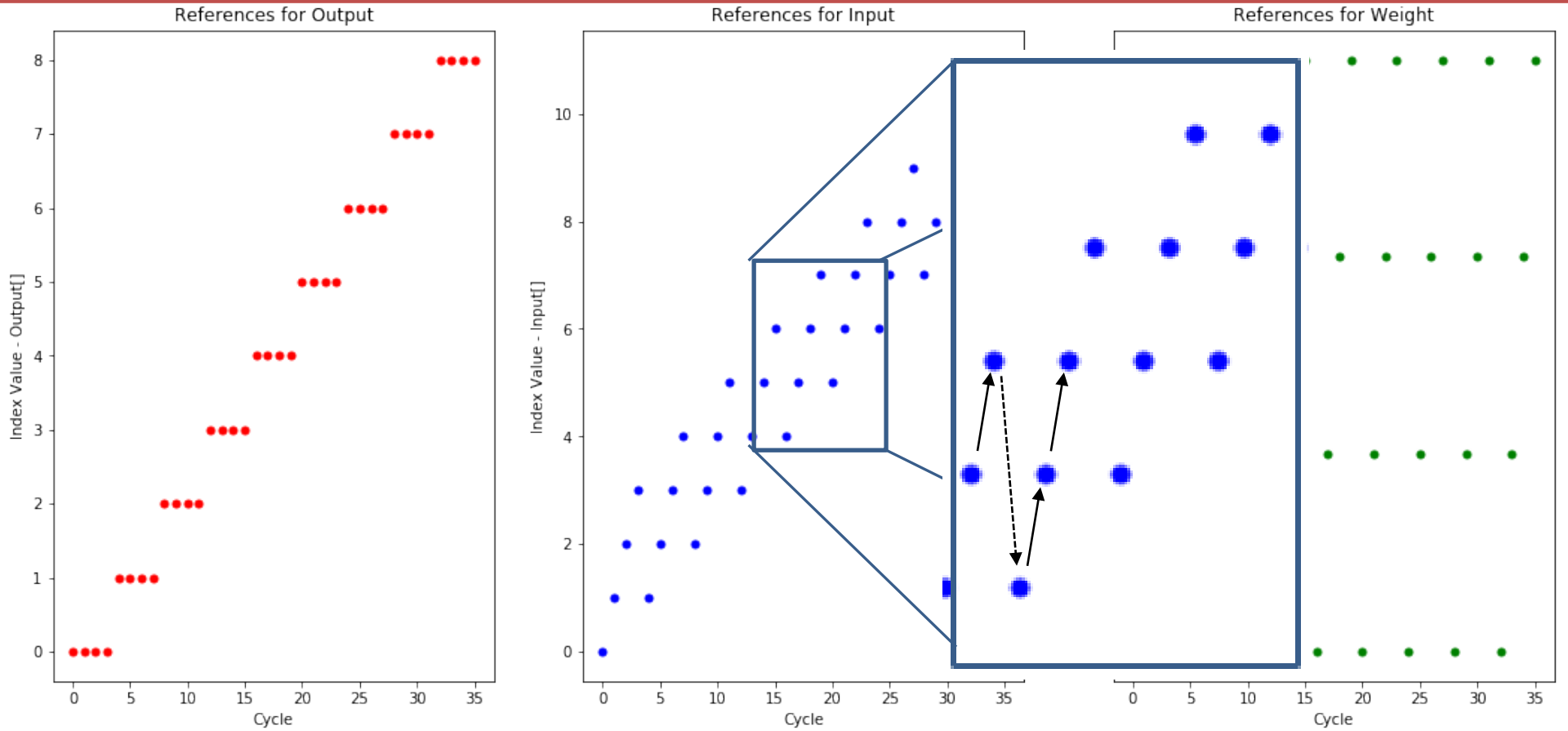
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

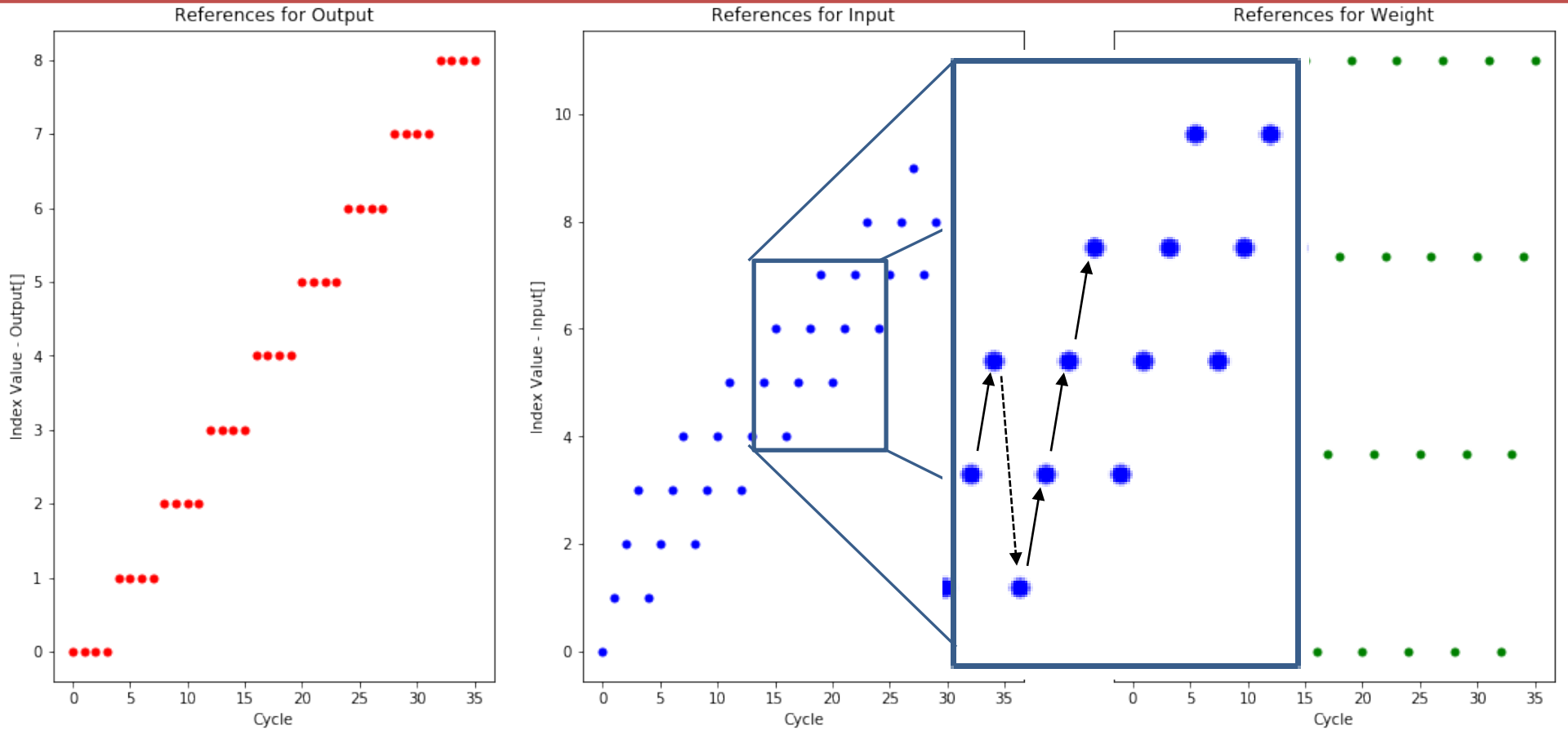
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

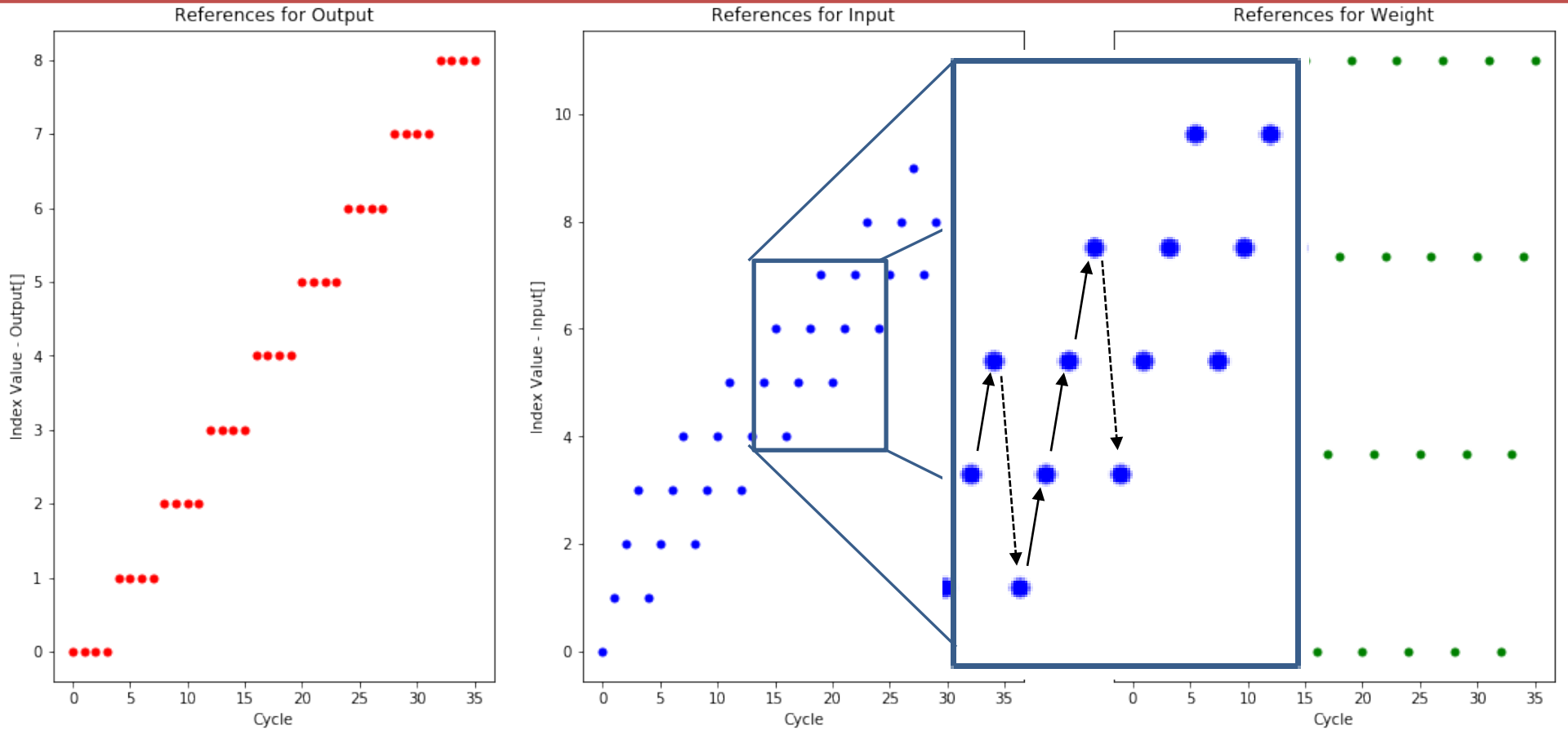
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

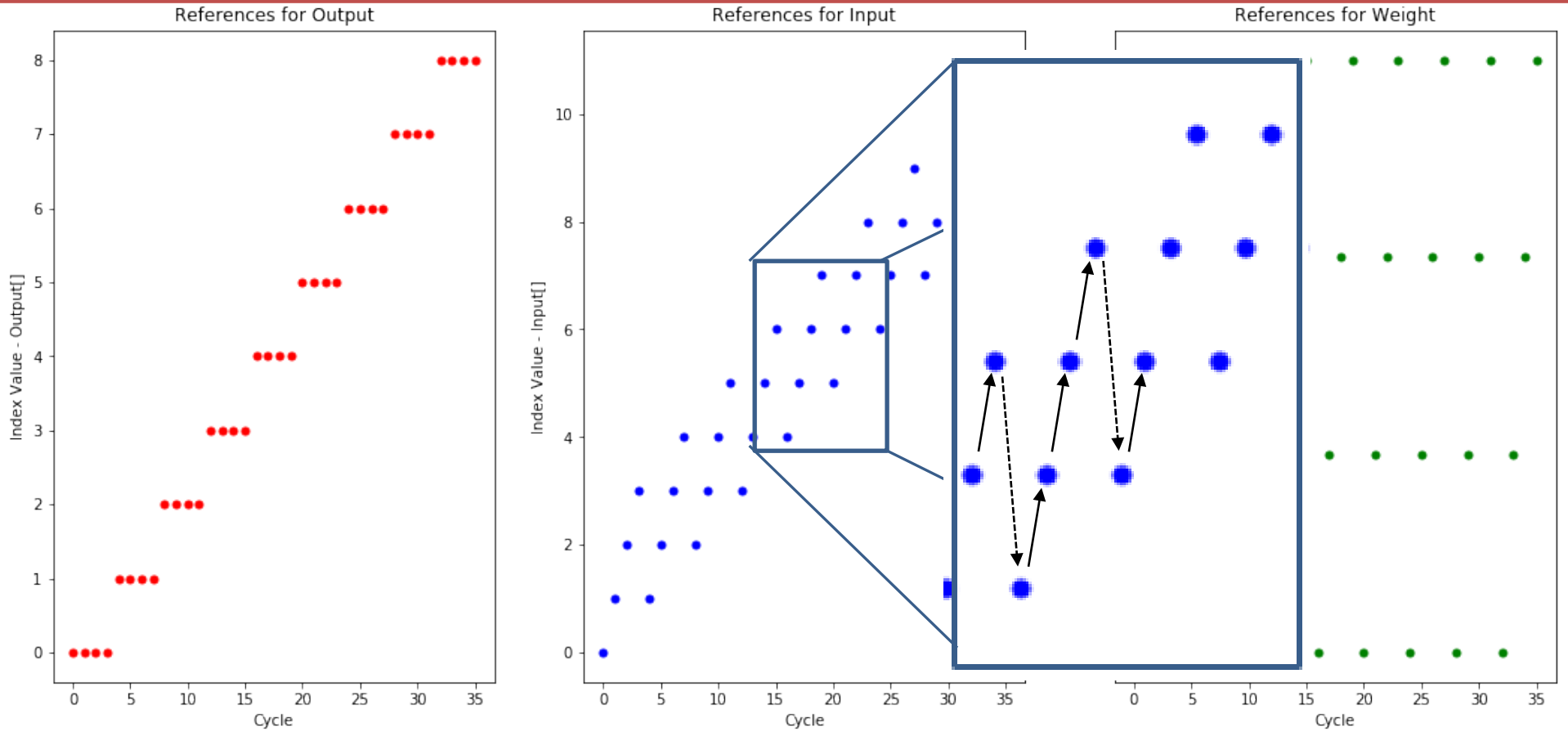
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

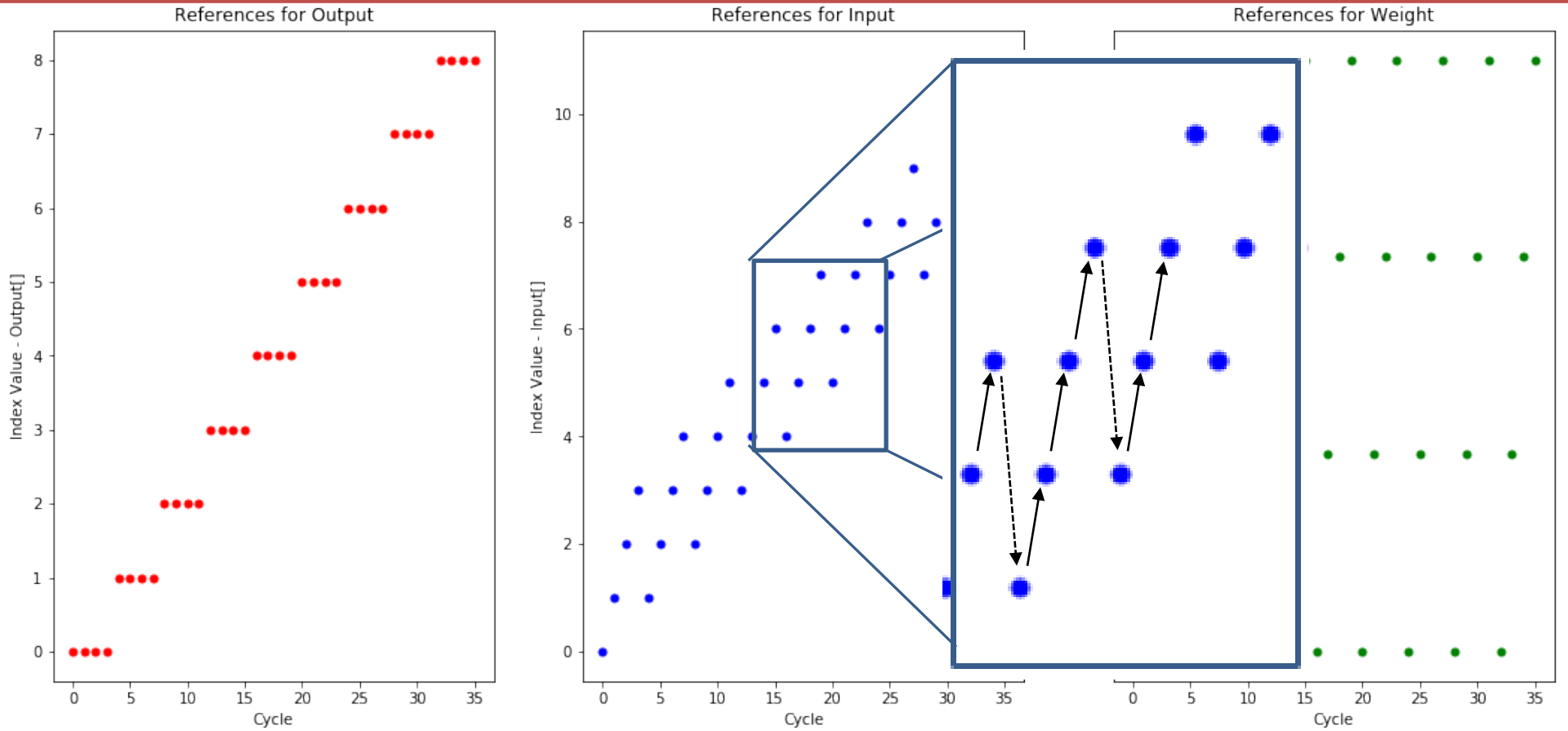
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

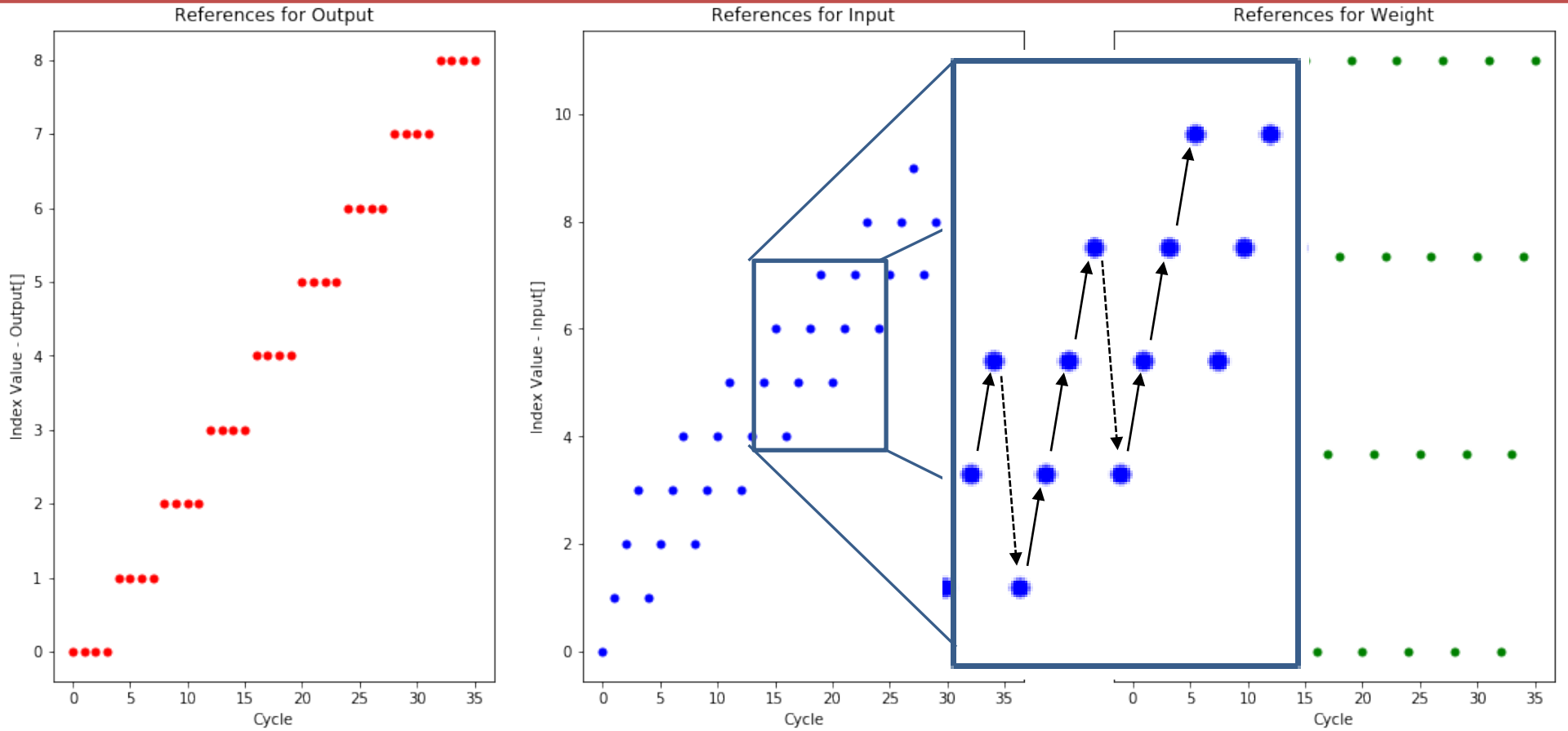
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

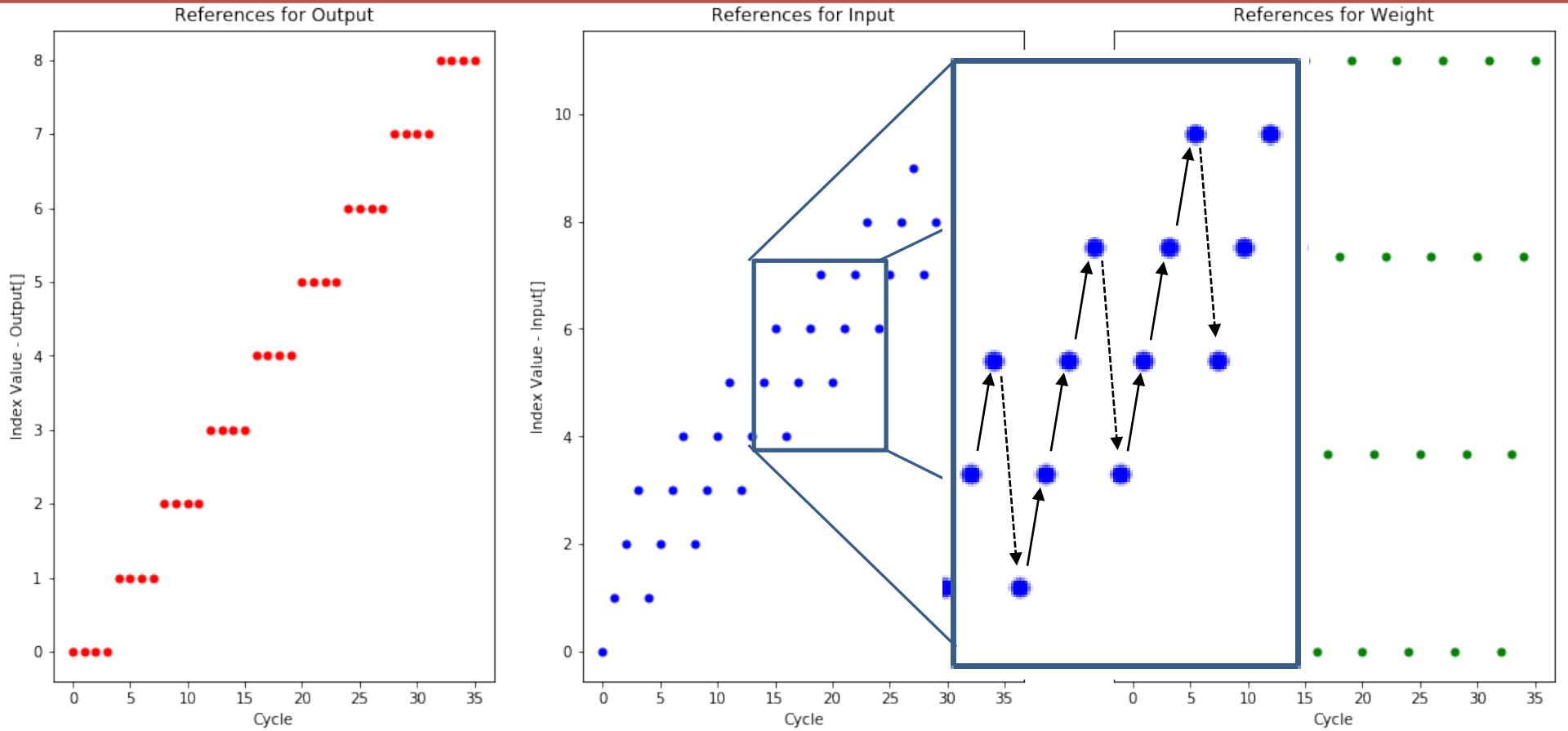
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

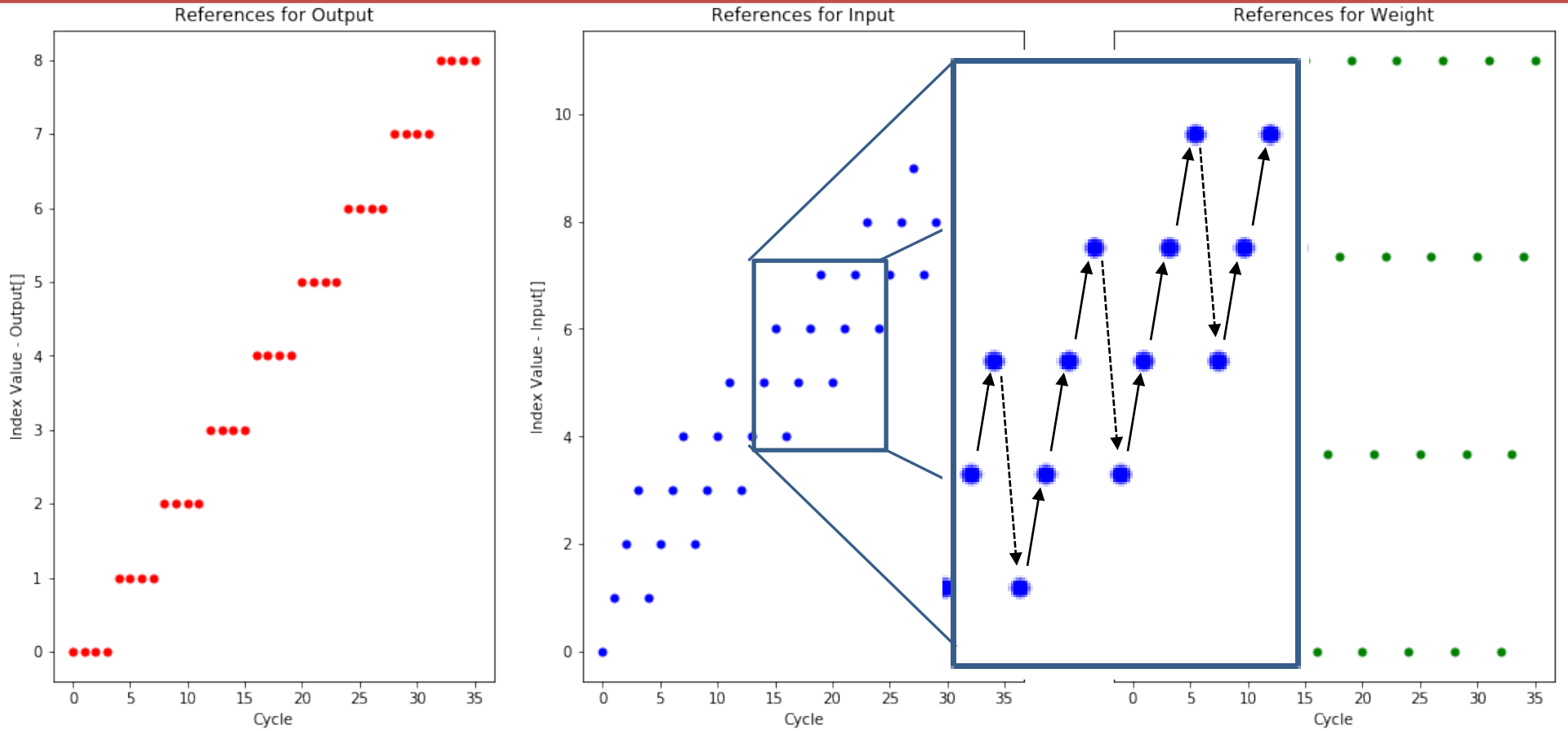
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

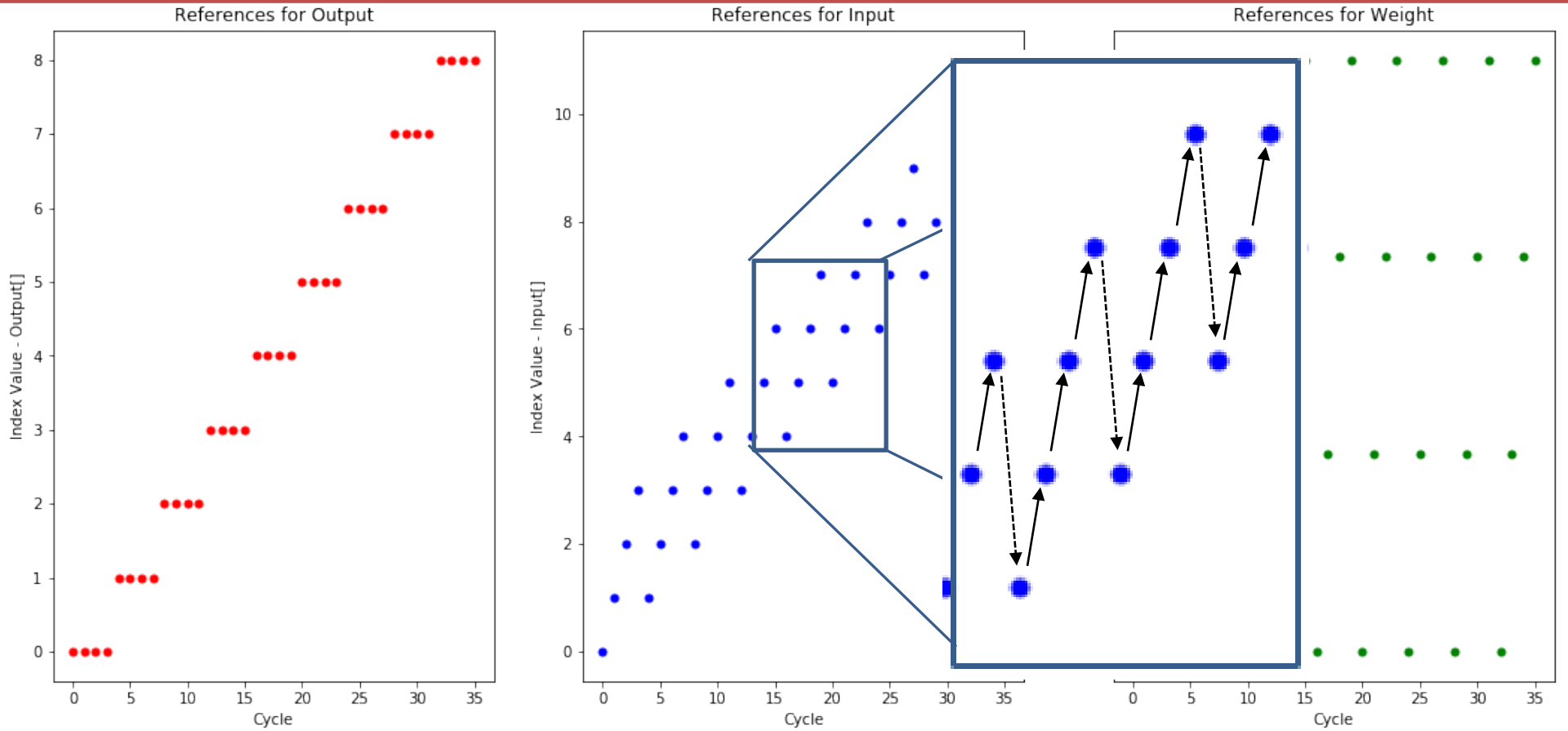
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly

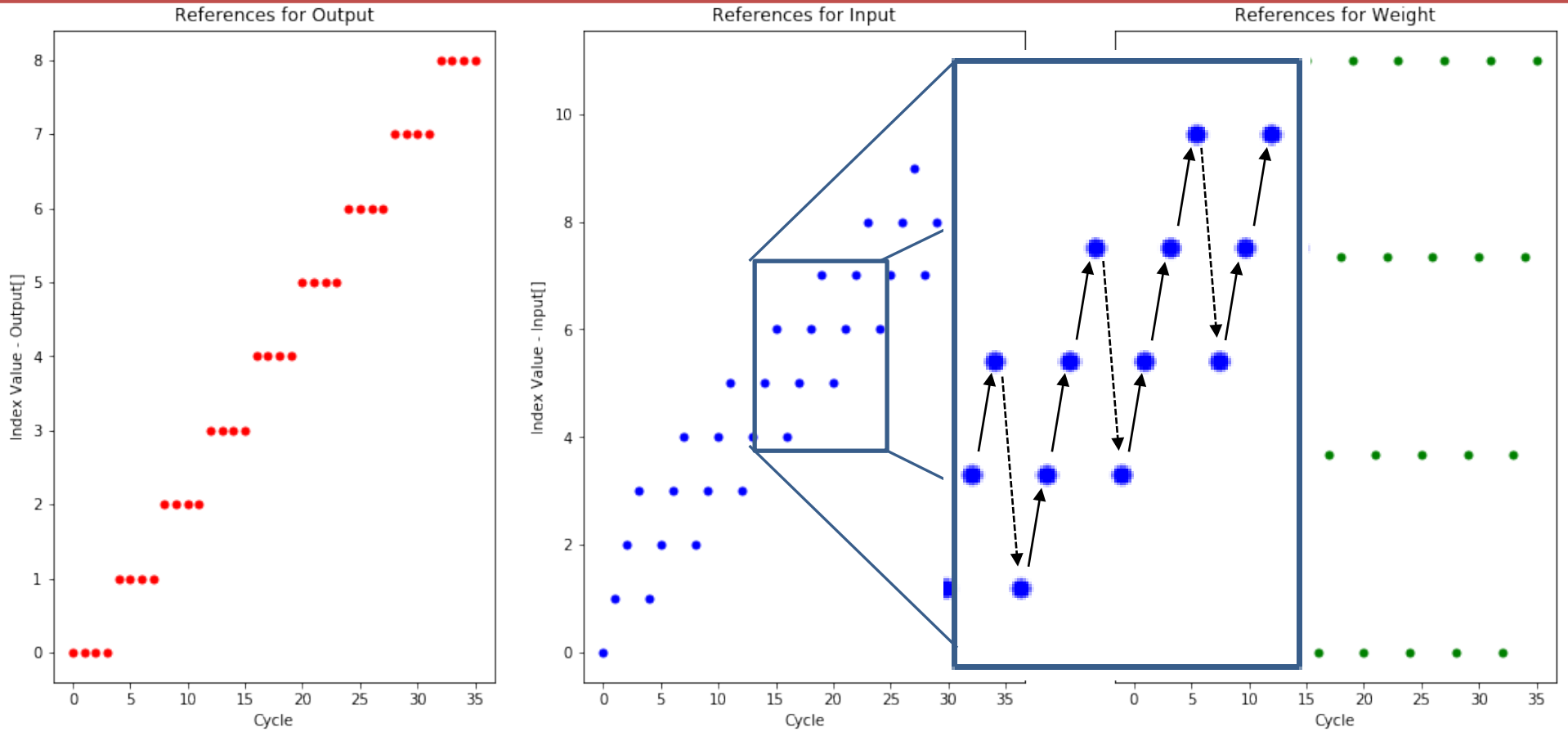
Output Stationary – Reference Pattern



Observations:

- Single output is reused many times (S)
- All weights reused repeatedly
- Sliding window of inputs (size = S)

Output Stationary – Reference Pattern



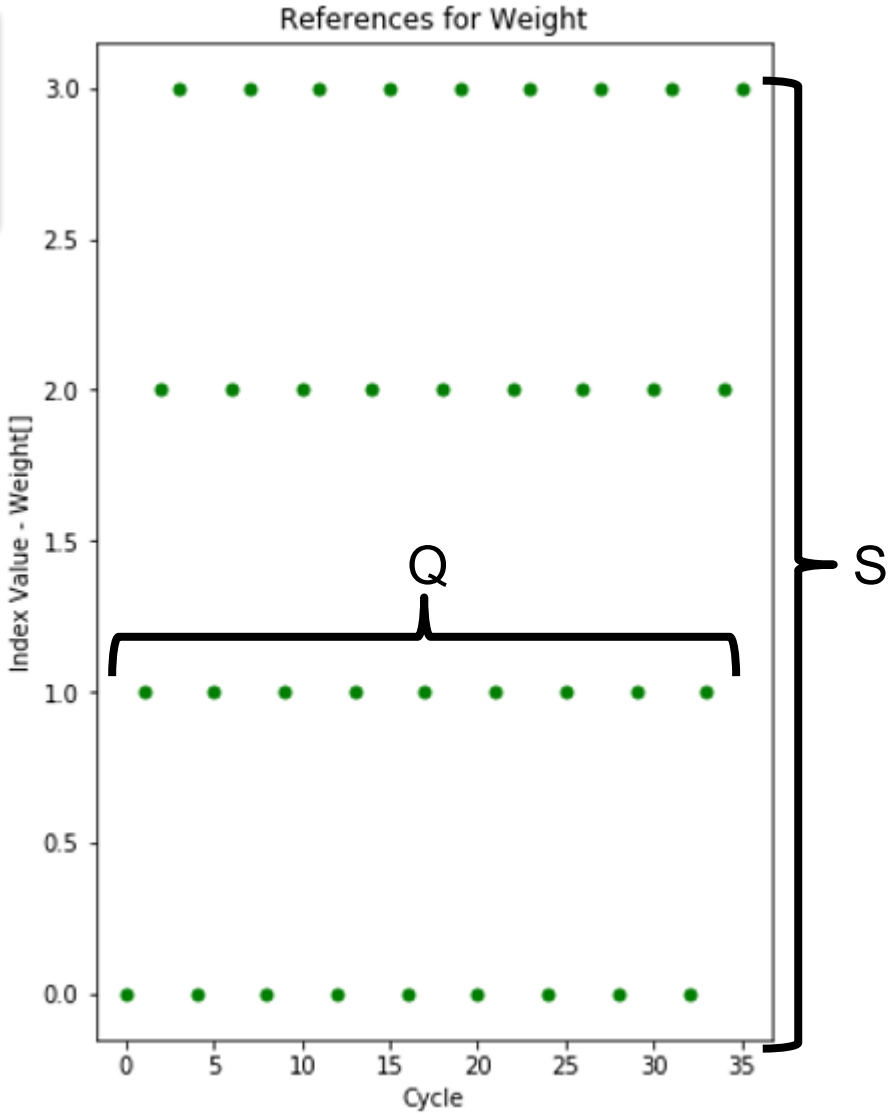
Observations:

- Single output is reused many times (S)
- All weights reused repeatedly
- Sliding window of inputs (size = S)

L1 Data Accesses - Weights

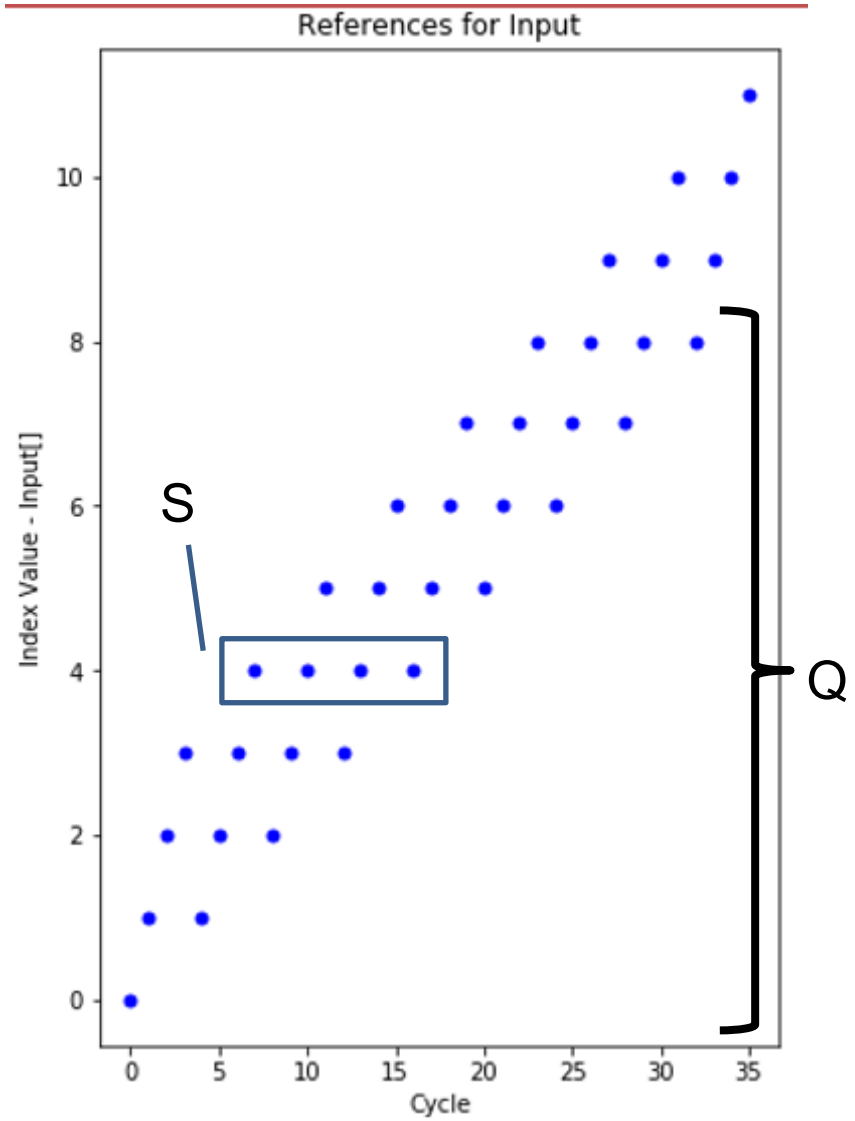
```
for q in [0, Q):
  for s in [0, S):
    o[q] += i[q+s]*f[s]
```

	OS
MACs	Q*S
Weight Reads	Q*S
Input Reads	
Output Reads	
Output Writes	



L1 Data Accesses - Inputs

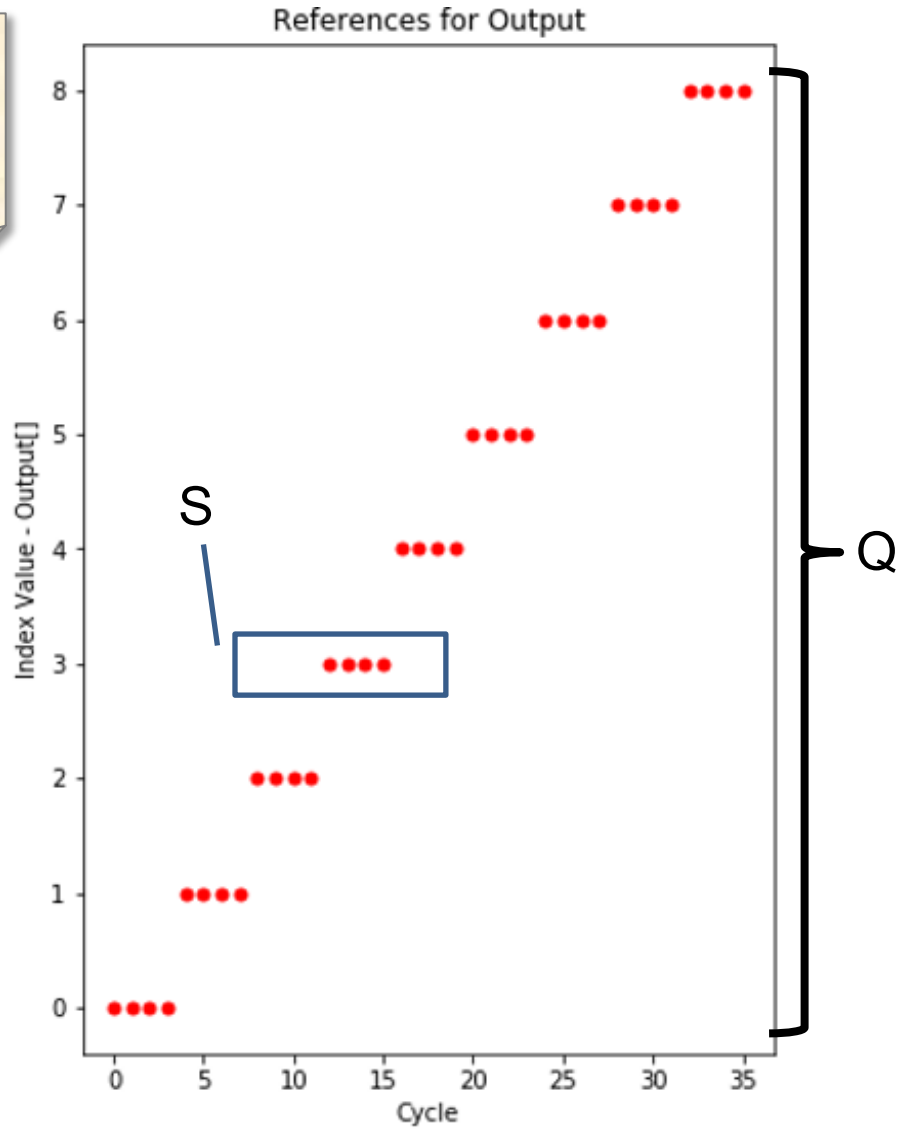
	OS
MACs	Q*S
Weight Reads	Q*S
Input Reads	Q*S
Output Reads	
Output Writes	



L1 Data Accesses - Outputs

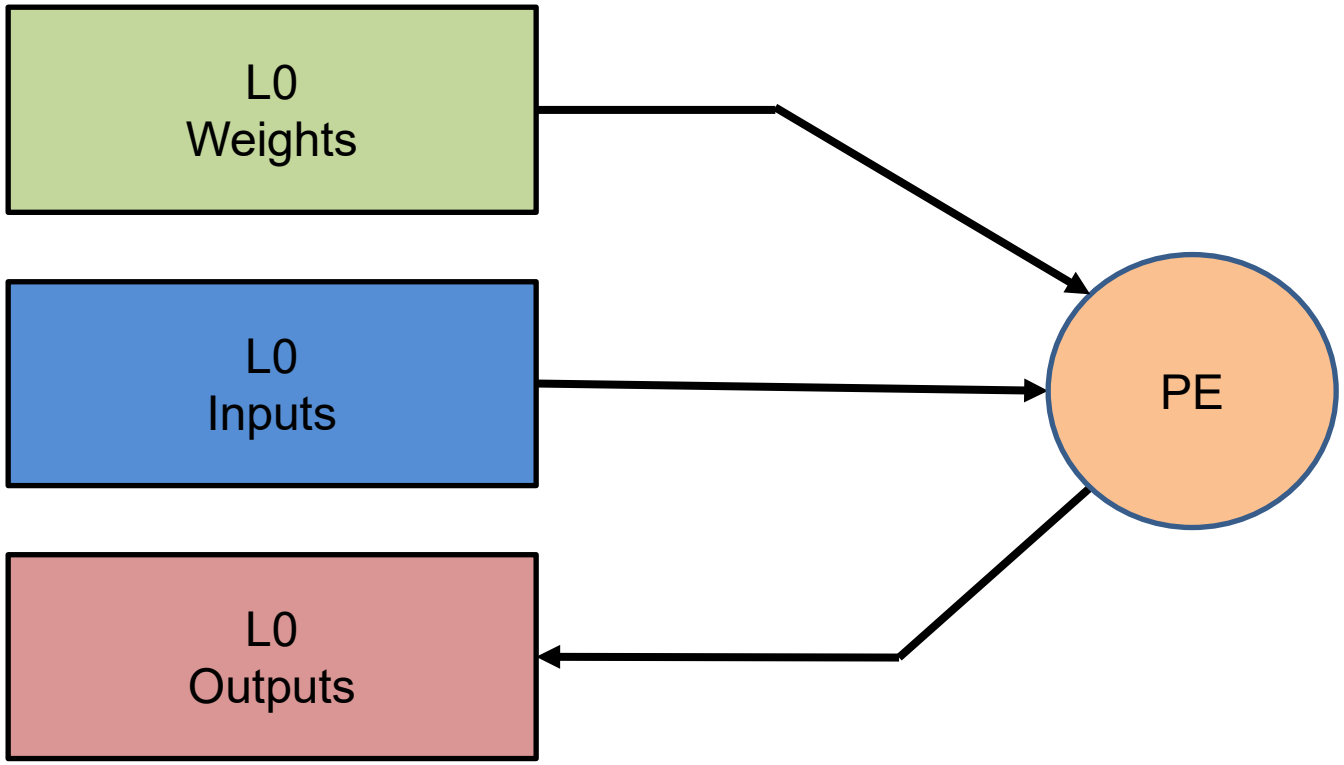
```
for q in [0, Q):
  for s in [0, S):
    o[q] += i[q+s]*f[s]
```

	OS
MACs	Q*S
Weight Reads	Q*S
Input Reads	Q*S
Output Reads	0
Output Writes	Q

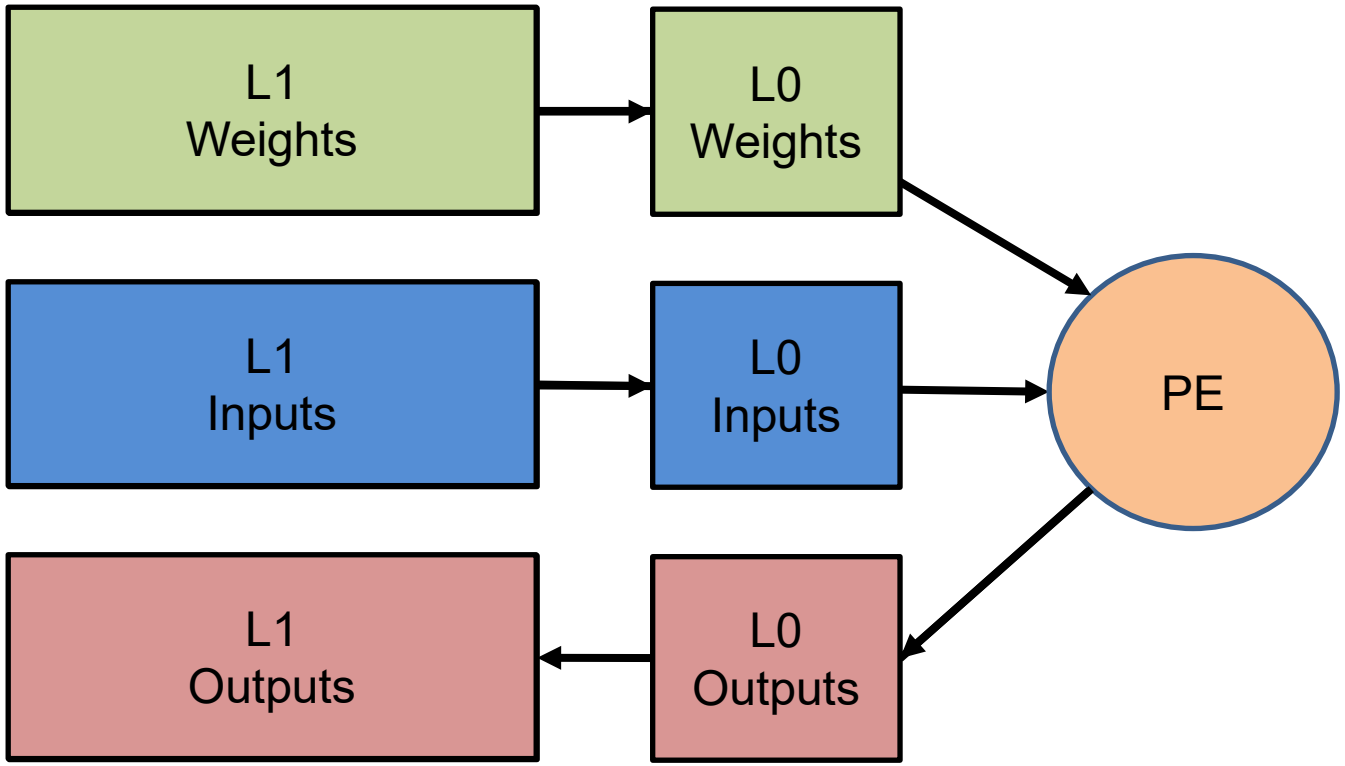


Intermediate Buffering

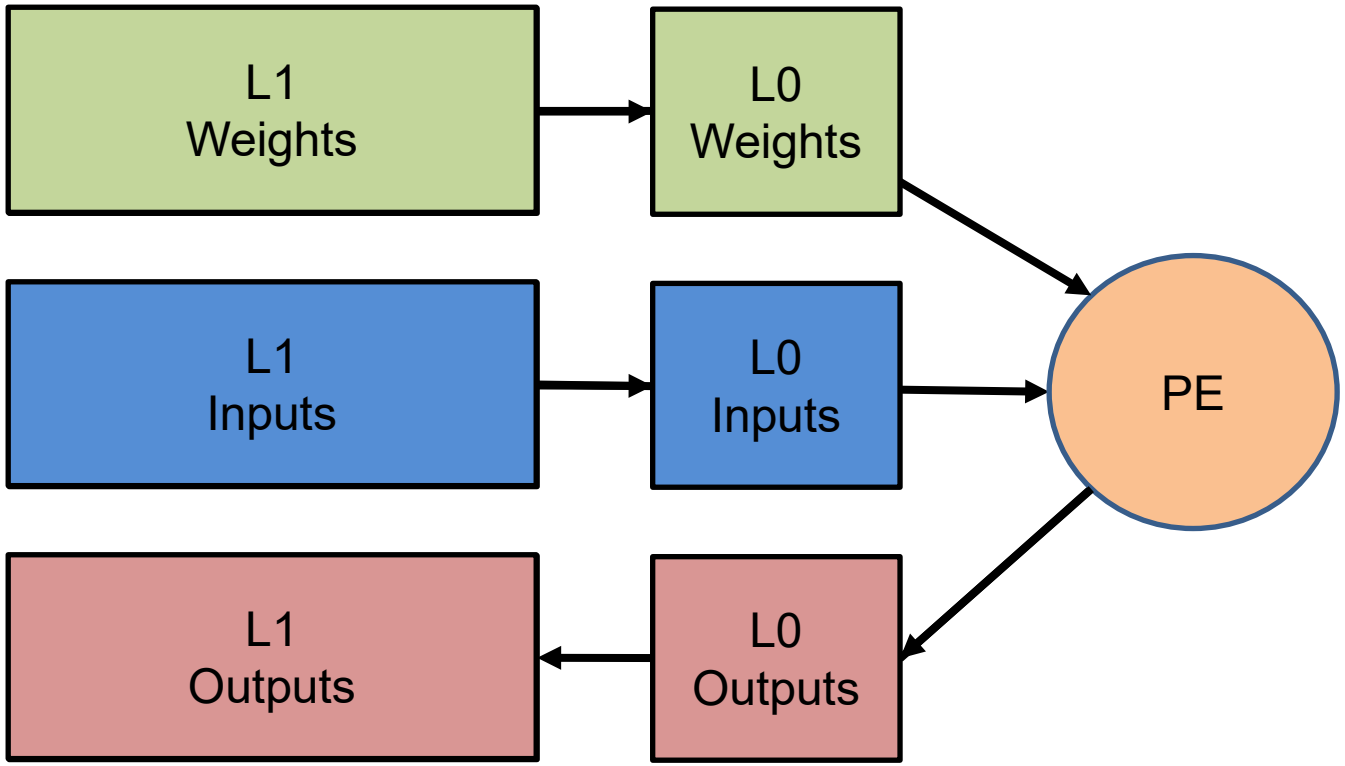
Intermediate Buffering



Intermediate Buffering



Intermediate Buffering



How will this be reflected in the loop nest?

1-D Convolution – Buffered


```
int i[W];      # Input activations
int f[S];      # Filter Weights
int o[Q];      # Output activations

// Level 1
for q1 in [0, Q1):
  for s1 in 0, S1):
    // Level 0
    for q0 in [0, Q0):
      for s0 in [0, S0):
          o[q1*Q0+q0] += i[q1*Q0+q0 + s1*S0+s0]
                       * f[s1*S0+s0]
```

Note Q and S are factored so:
Q0*Q1 = Q
S0*S1 = S

† Assuming: 'valid' style convolution

Buffer sizes

```
// Level 1
for q1 in 0, Q1):
  for s1 in [0 to S1):
    // Level 0
    for q0 in [0, Q0):
      for s0 in [0, S0):
          o[q1*Q0+q0] += i[q1*Q0+q0 + s1*S0+s0]* f[s1*S0+s0];
}
```

- Level 0 buffer size is volume needed in each Level 1 iteration.
- Level 1 buffer size is volume needed to be preserved and re-delivered in future (usually successive) Level 1 iterations.

- A legal assignment of loop limits will fit into the hardware's buffer sizes

Buffered – 1D Convolution Einsum

$$O_q = I_{q+s} \times F_s$$

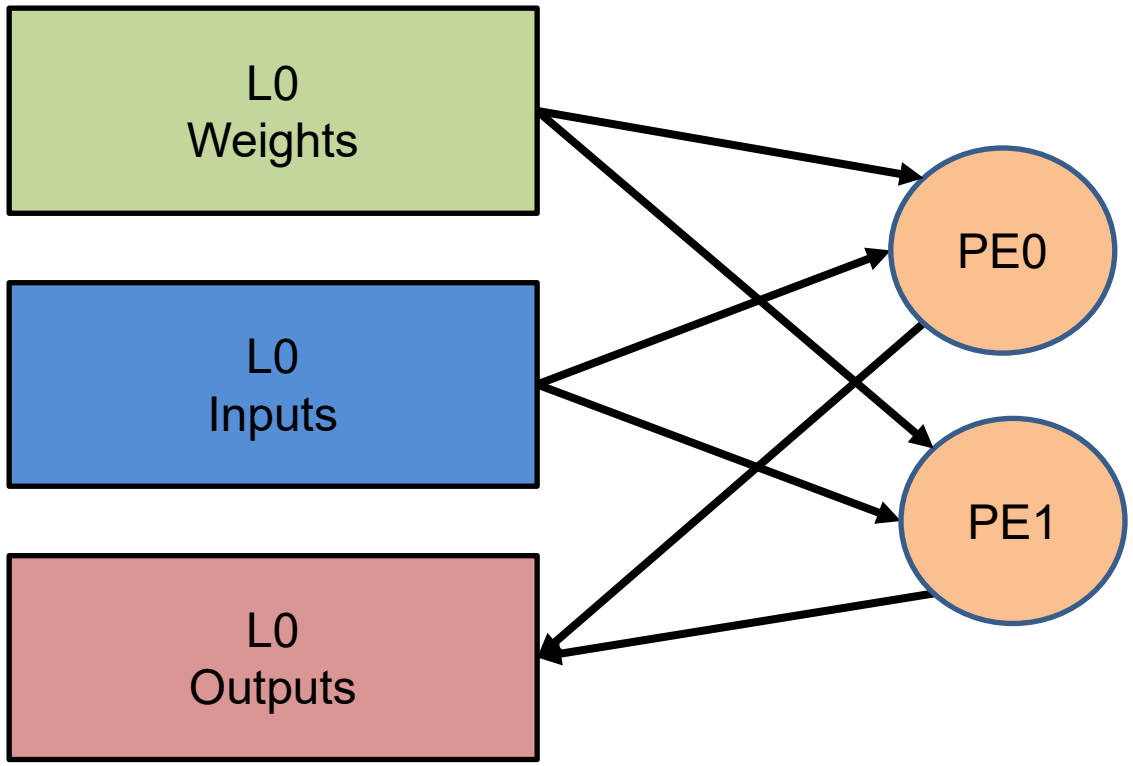
Split: S by S0 and Q by Q0

$$O_{q1*Q0+q0} = I_{q1*Q0+q0+s1*S0+s0} \times F_{s*S0+s0}$$

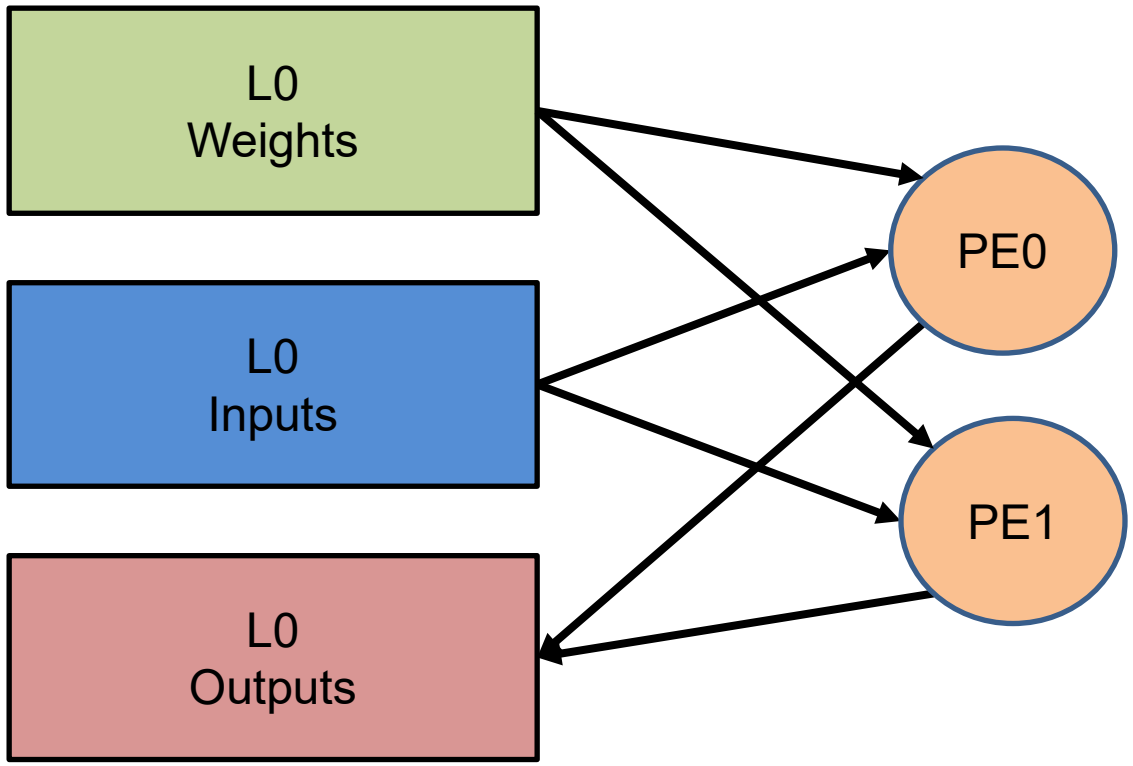
Traversal order (fastest to slowest): S0, Q0, S1, Q1

Spatial Mapping

Spatial PEs

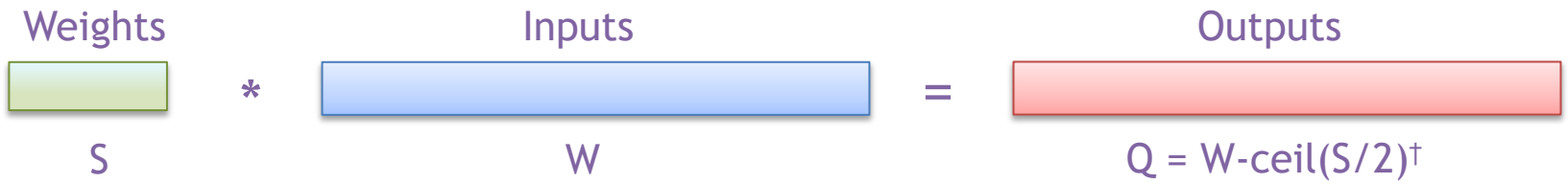


Spatial PEs



How will this be reflected in the loop nest?

1-D Convolution – Spatial



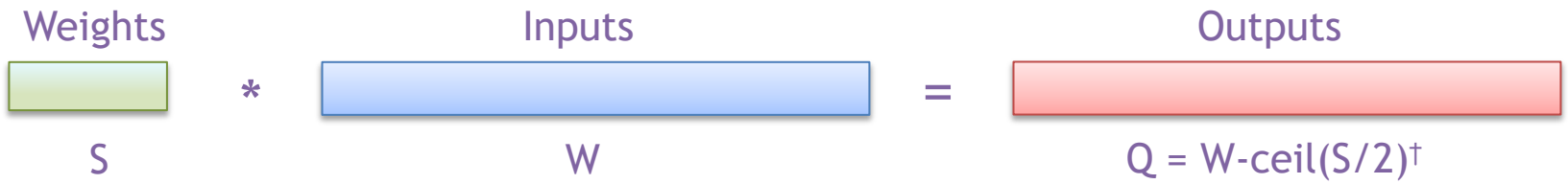

```
int i[W];      # Input activations
int f[S];      # Filter Weights
int o[Q];      # Output activations

// Level 1
parallel-for q1 in [0, Q1):
  parallel-for s1 in [0, S1):
    // Level 0
    for s0 in S0):
      for q0 in 0, Q0):
          o[q1*Q0+q0] += i[q1*Q0+q0 + s1*S0+s0]
                       * f[s1*S0+s0];
}
```

Note:
Q0*Q1 = Q
S0*S1 = S

Q1 = 1 => q1 = 0

† Assuming: 'valid' style convolution

1-D Convolution – Spatial



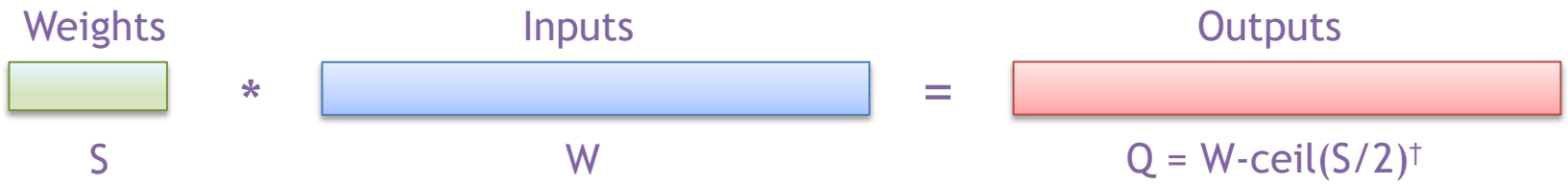

```
int i[W];      # Input activations
int f[S];      # Filter Weights
int o[Q];      # Output activations

// Level 1
parallel-for q1 in [0, Q1):
  parallel-for s1 in [0, S1):
    // Level 0
    for s0 in S0):
      for q0 in 0, Q0):
          o[q1*Q0+q0] += i[q1*Q0+q0 + s1*S0+s0]
                       * f[s1*S0+s0];
}
```

Note:
Q0*Q1 = Q
S0*S1 = S

Q1 = 1 => q1 = 0

† Assuming: 'valid' style convolution

1-D Convolution – Spatial




```
int i[W];      # Input activations
int f[S];      # Filter Weights
int o[Q];      # Output activations

// Level 1
parallel-for q1 in [0, Q1):
  parallel-for s1 in [0, S1):
    // Level 0
    for s0 in S0):
      for q0 in 0, Q0):
          o[q1*Q0+q0] += i[q1*Q0+q0 + s1*S0+s0]
                       * f[s1*S0+s0];
}
```

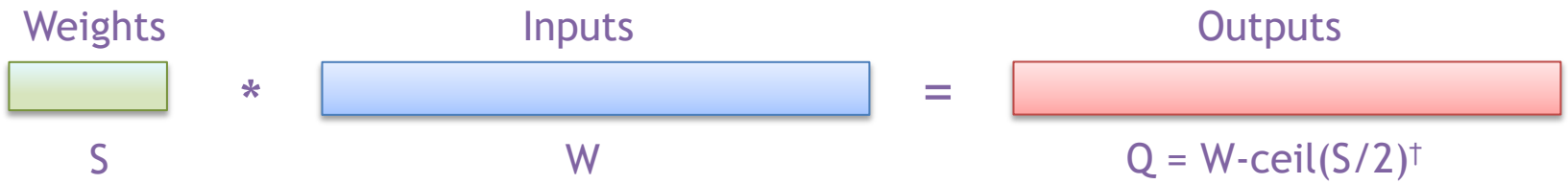
Note:
Q0*Q1 = Q
S0*S1 = S

Q1 = 1 => q1 = 0

S0 = 1, S1 = 2

† Assuming: 'valid' style convolution

1-D Convolution – Spatial




```
int i[W];      # Input activations
int f[S];      # Filter Weights
int o[Q];      # Output activations

// Level 1
parallel-for s1 in [0, S1):
  // Level 0
  for s0 in [0, S0):
    for q in [0, Q):
      o[q] += i[q+s1*S0+s0] * f[s1*S0+s0]
```

Note:
Q0*Q1 = Q
S0*S1 = S

† Assuming: 'valid' style convolution

Spatial – 1D Convolution Einsum

$$O_q = I_{q+s} \times F_s$$

Split: S by S0

$$O_q = I_{q+s1*S1+s0} \times F_{s*S0+s0}$$

Traversal order (fastest to slowest): S0, Q

Spatial – 1D Convolution Einsum

$$O_q = I_{q+s} \times F_s$$

Split: S by S0

$$O_q = I_{q+s1*S1+s0} \times F_{s*S0+s0}$$

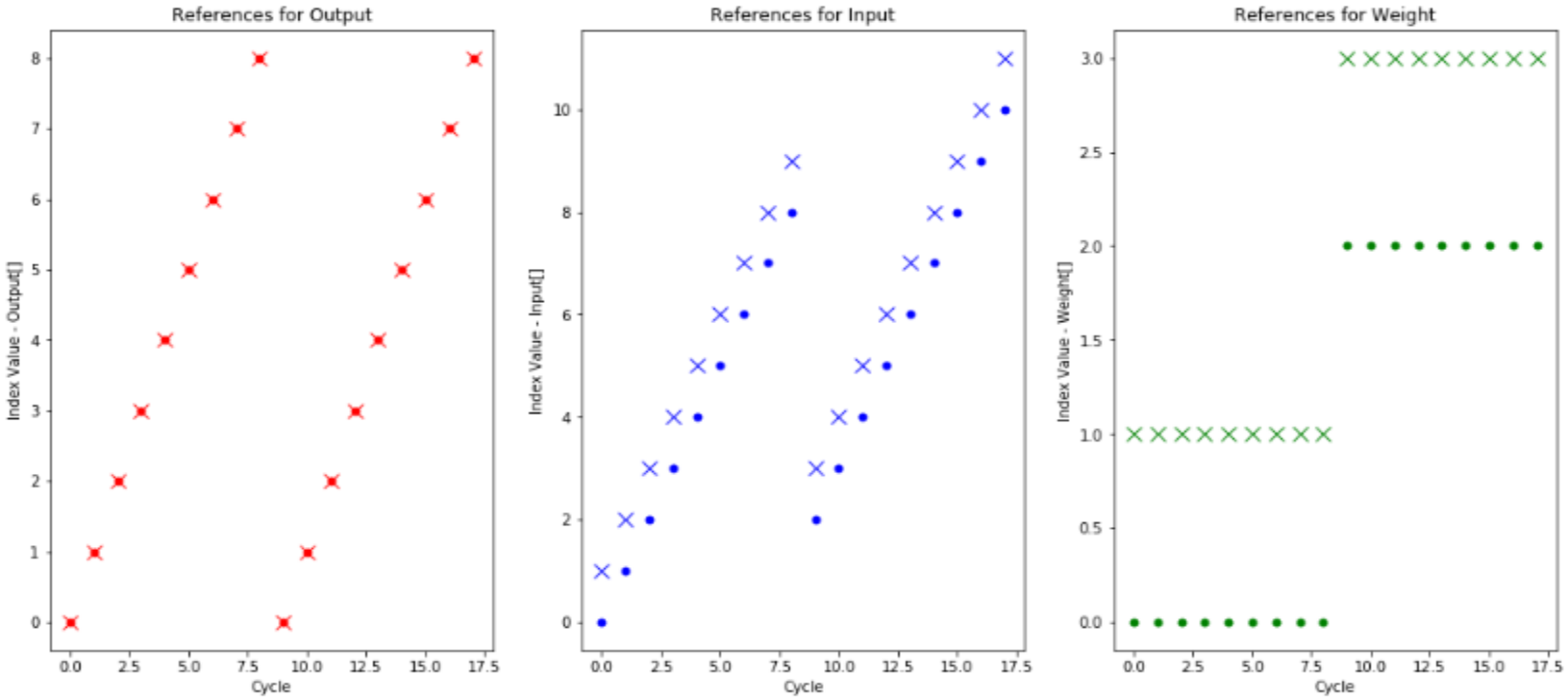
Traversal order (fastest to slowest): S0, Q

Parallel: S1

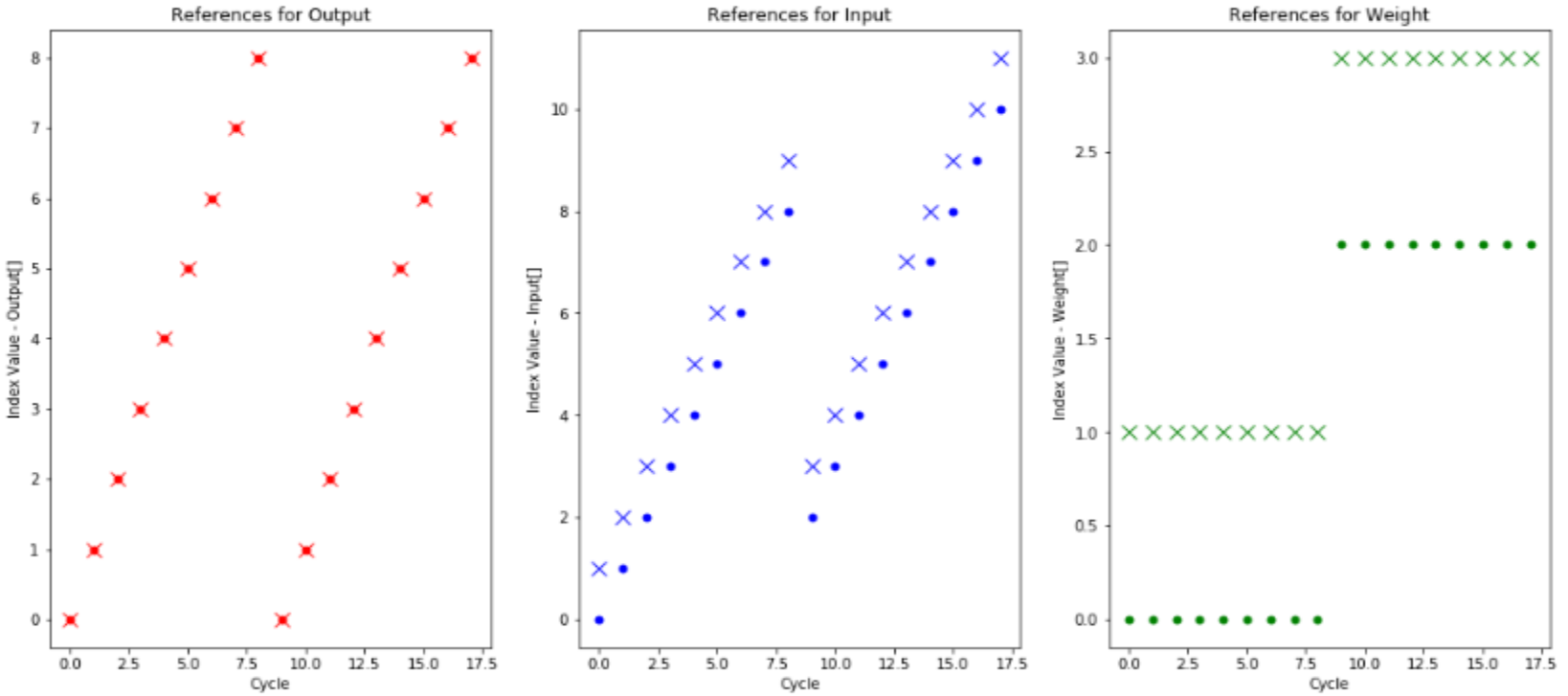
Spatial Weight Stationary References

Shape:
- S = 4
- Q = 9
- W = 12

Loop limits:
- S1 = 2
- S0 = 2
- Q0 = 9

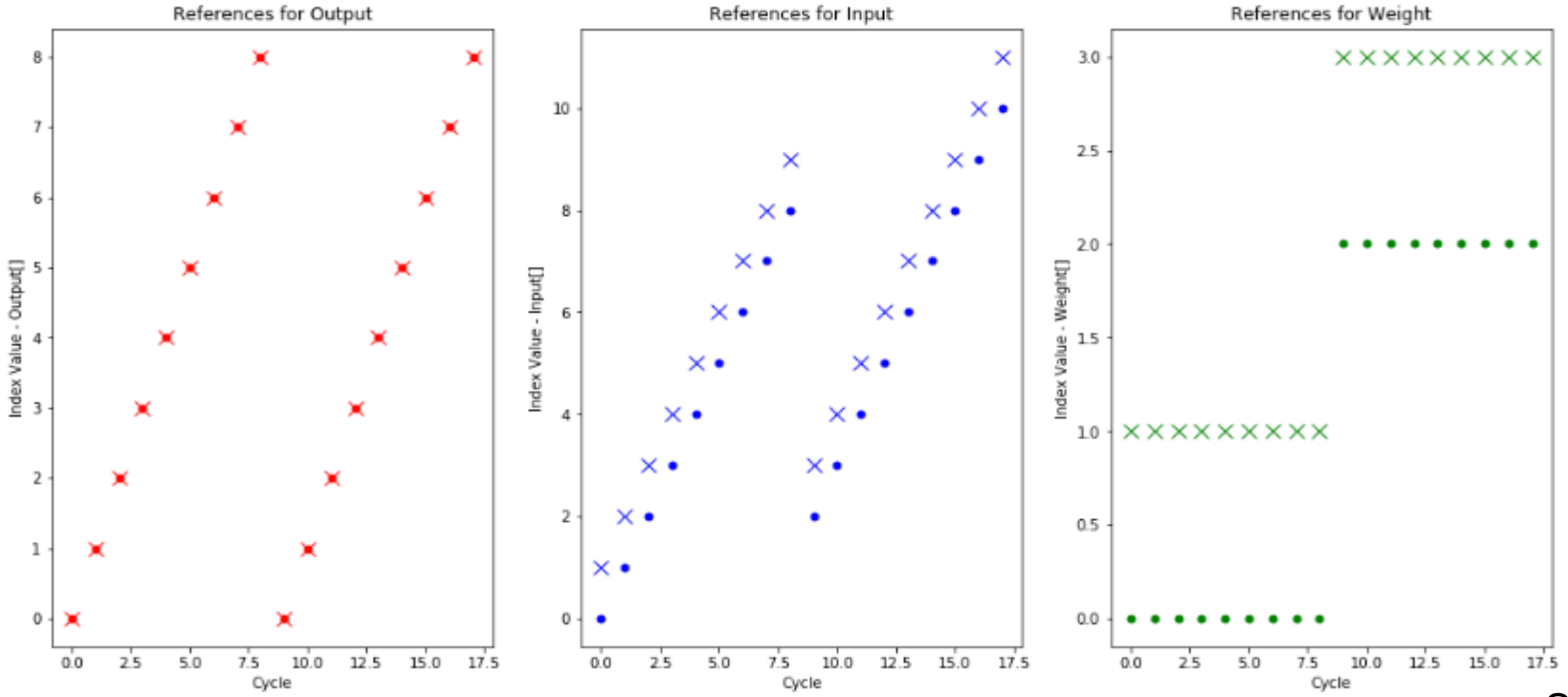


Spatial Weight Stationary References



S1 = 2
S0 = 2
Q0 = 9

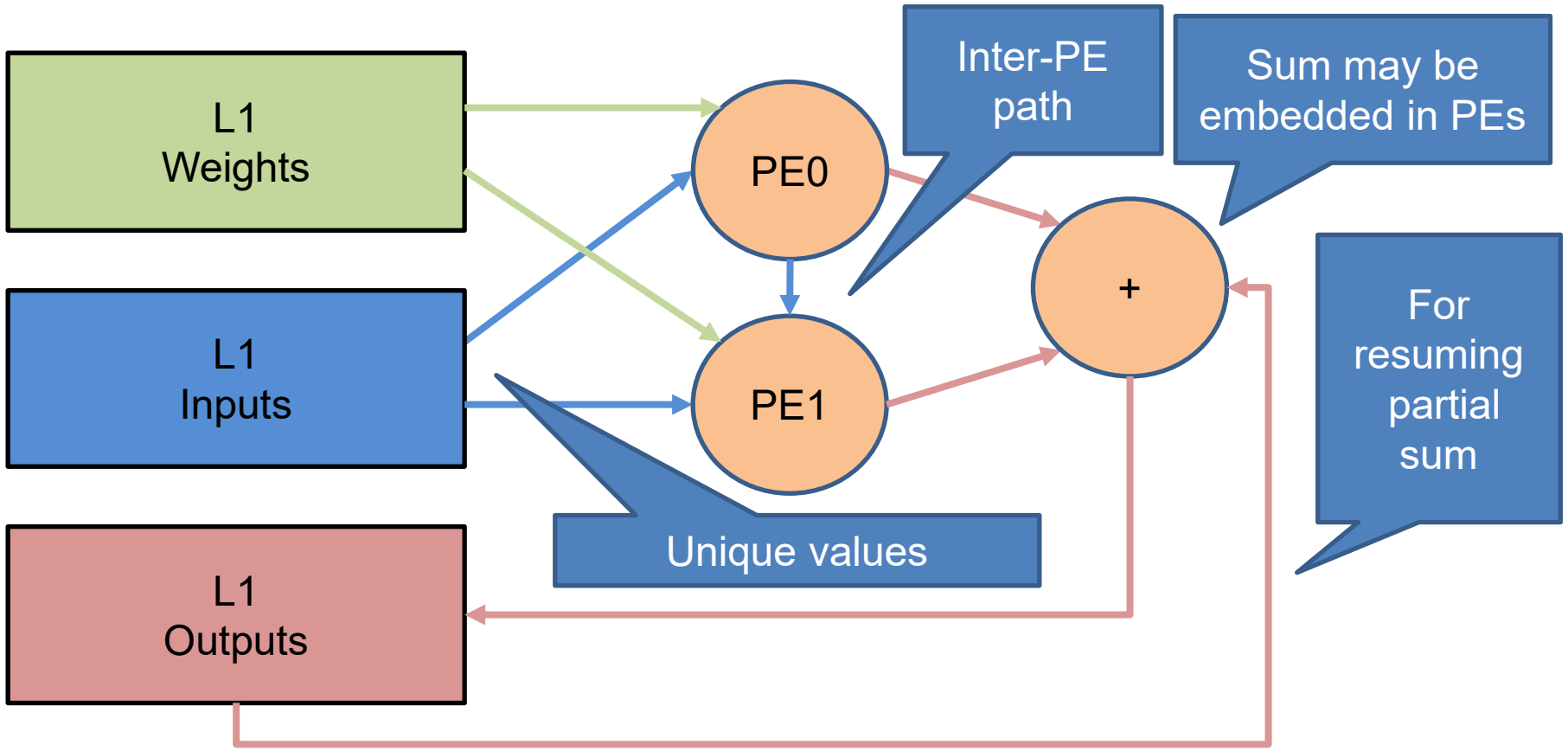
Spatial Weight Stationary References



S1 = 2
S0 = 2
Q0 = 9

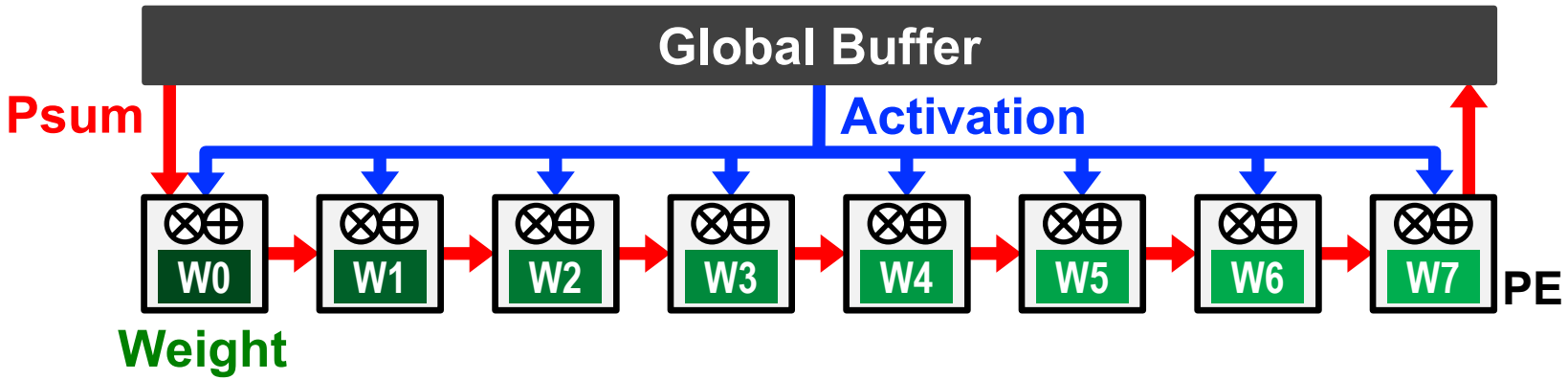
- Number of cycles half that of before
- Single weight per PE used for a long time (unicast to each PE)
- Inputs reused in next cycle (opportunity for inter-PE communication)
- Inputs are also reused after a long interval, implying a large window (Q)
- Partial sums are reused in same cycle (opportunity for spatial sum)
- Partial sums reused after a long interval, very large window (size = Q)

Spatial PEs



What if hardware cannot do a spatial sum?

Weight Stationary (WS)



- Note that activations are multi-cast.

- To achieve this behavior we need to "skew" the activity in the PEs so instead of needing activation in adjacent cycles there are needed in the same cycle!

Buffered – 1D Convolution Einsum

$$O_q = I_{q+s} \times F_s$$

Split: S by S0

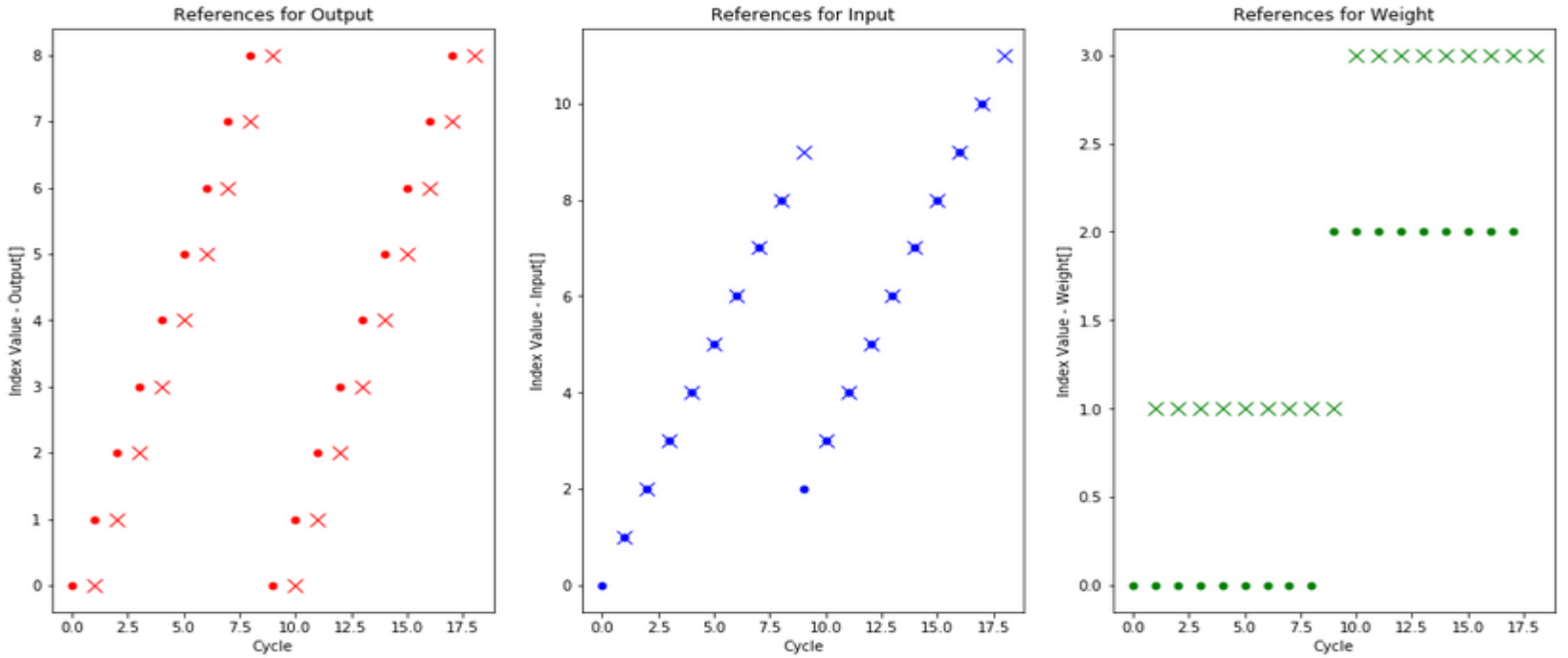
$$O_q = I_{q+s1*S1+s0} \times F_{s*S0+s0}$$

Traversal order (fastest to slowest): S0, Q

Parallel: S1

Time Skew: +s1

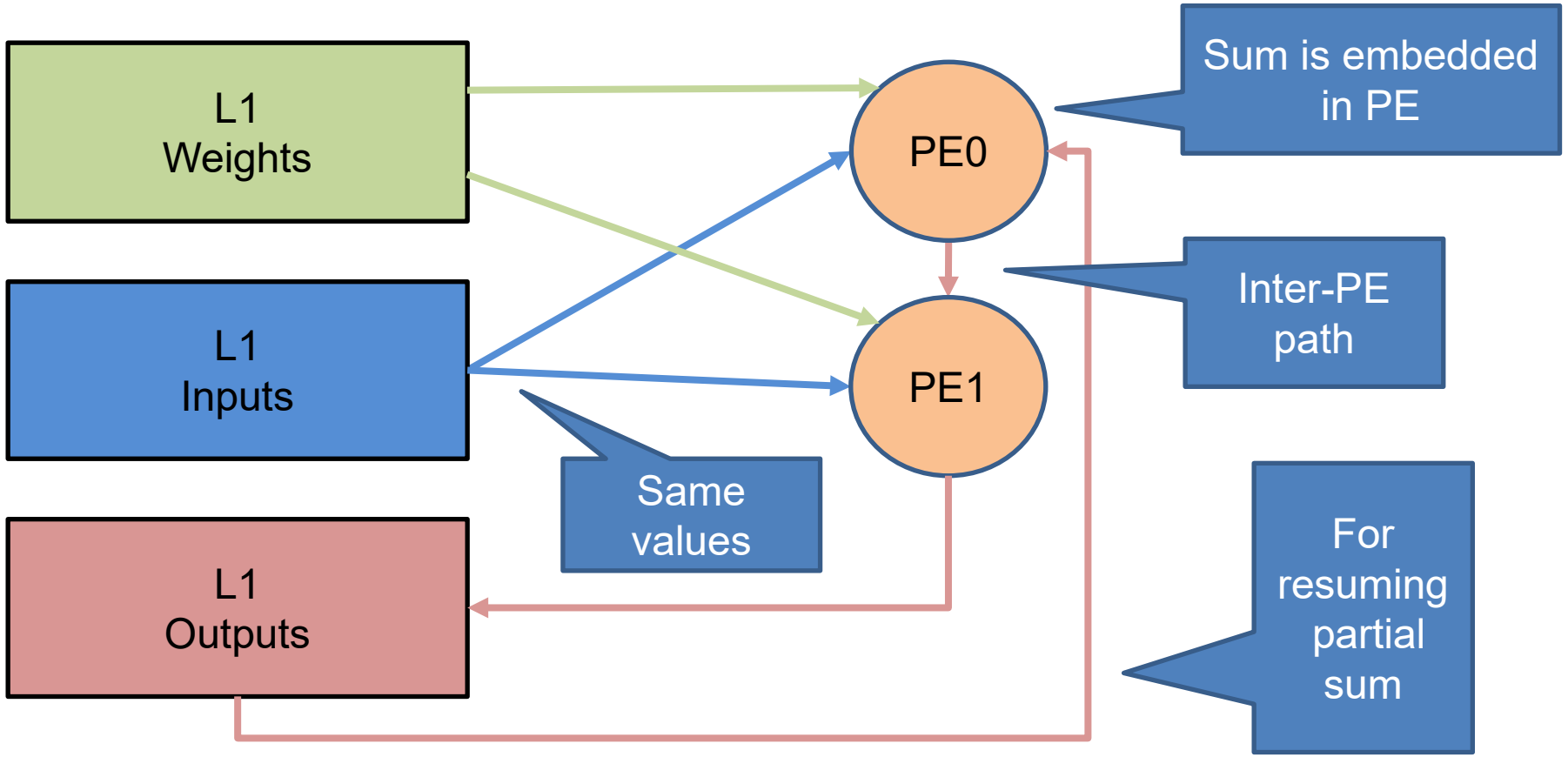
Spatial Weight Stationary (Skewed)



S1 = 2
S0 = 2
Q0 = 9

- Single weight per PE used for a long time (unicast to each PE)
- Inputs used **simultaneously** at both PEs (opportunity for multicast)
- Inputs are also reused after a long interval, implying a large window (Q)
- Partial sums are reused are **reused in adjacent cycles** in adjacent PEs
  opportunity for inter-PE communication and temporal sum
- Partial sums reused after a long interval, very large window (size = Q)

Spatial PEs



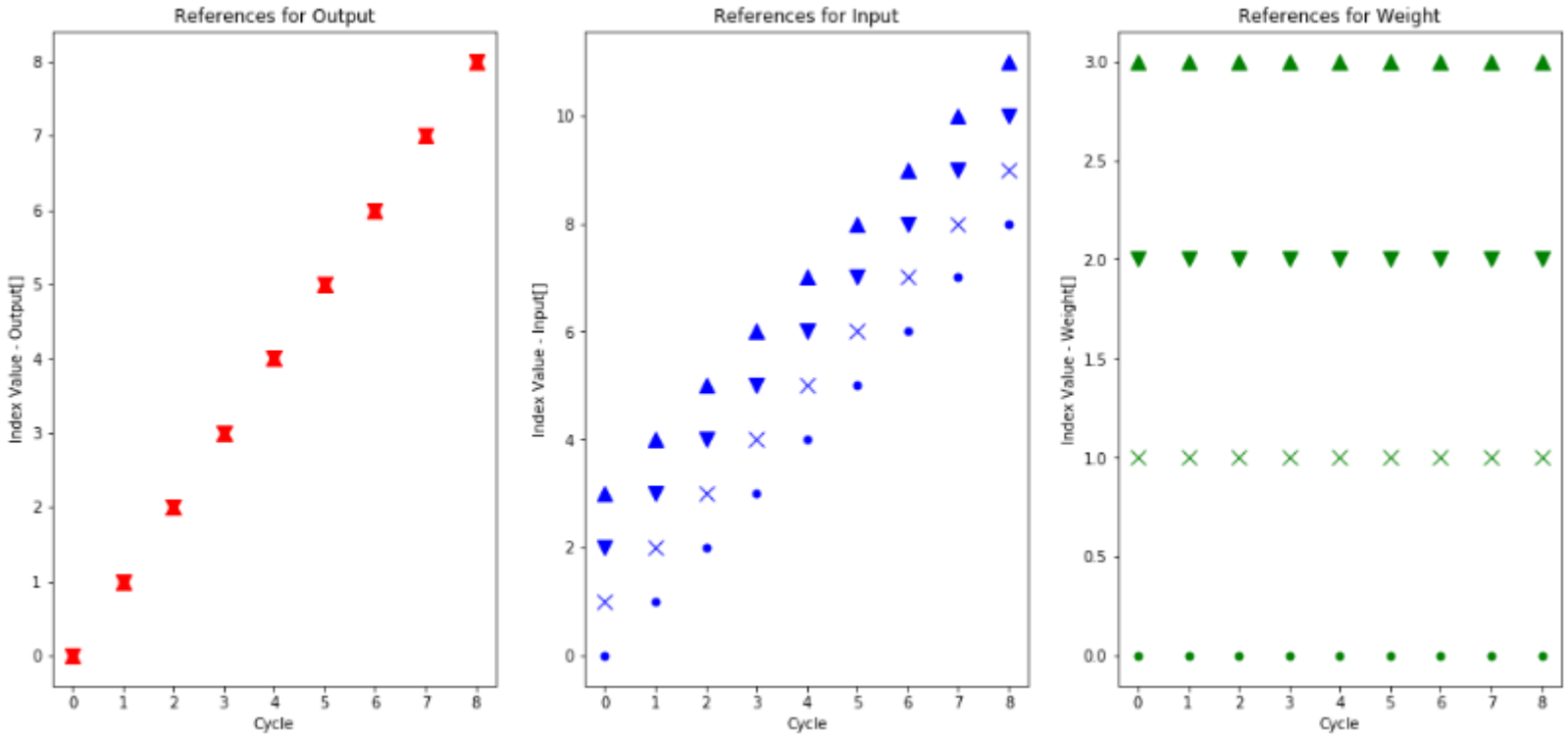
Weights are still unique, but note lower bandwidth and bursty demand.

Is there a way to avoid the large input and psum window?

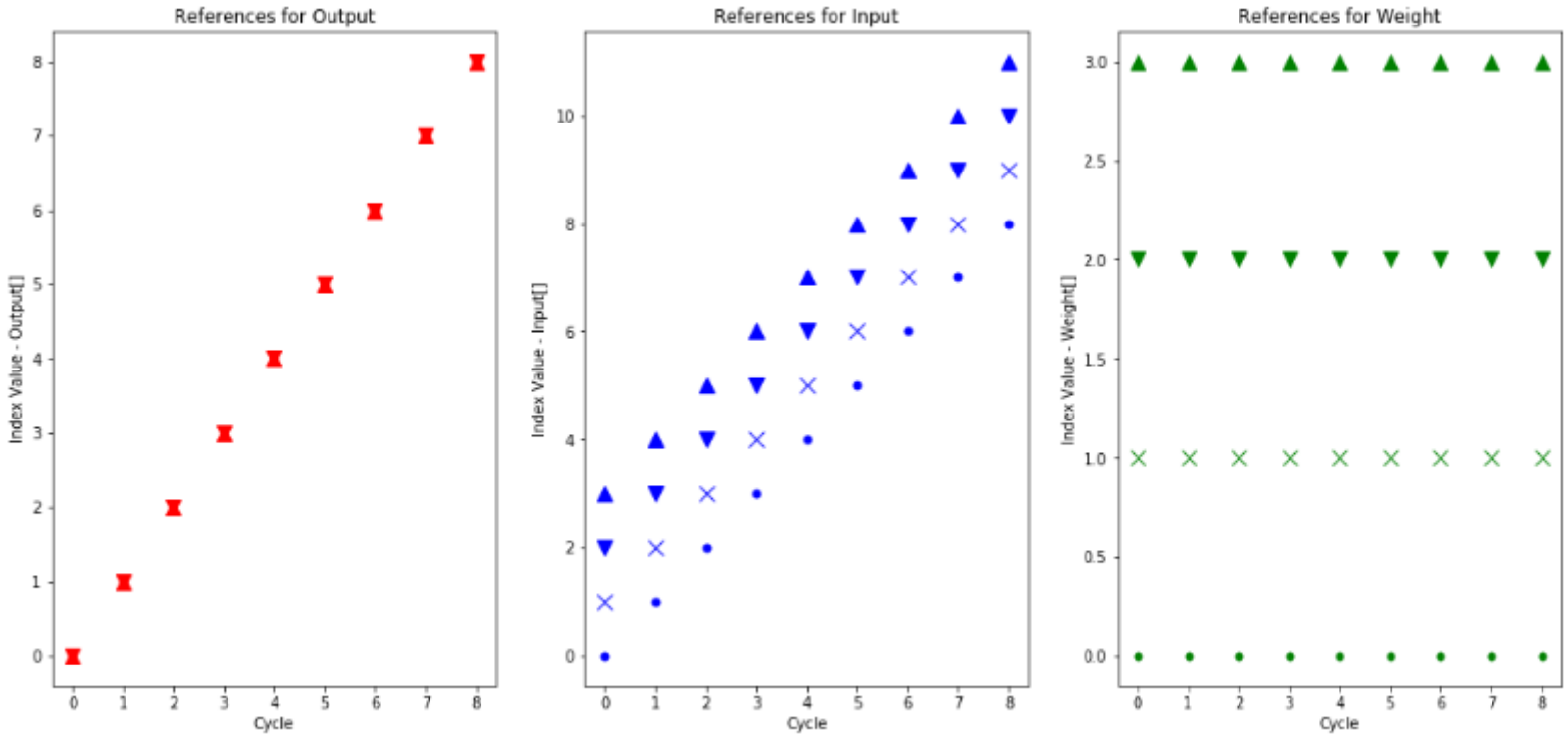
With S1 = S

Shape:
- S = 4
- Q = 9
- W = 12

Loop limits:
- S1 = 4
- S0 = 1
- Q0 = 9



With S1 = S


References for Output
Index Value - Output[]
Cycle
References for Input
Index Value - Input[]
Cycle
References for Weight
Index Value - Weight[]
Cycle

Mapping Process

Mapping

Definition: selecting the placement and scheduling in space and time of every operation (including delivering the appropriate operands) required for a DNN computation onto the hardware function units of the accelerator.

Mapping

Definition: selecting the placement and scheduling in space and time of every operation (including delivering the appropriate operands) required for a DNN computation onto the hardware function units of the accelerator.

Steps: Within the constraints of the hardware, select for each level of the storage hierarchy:

Mapping

Definition: selecting the placement and scheduling in space and time of every operation (including delivering the appropriate operands) required for a DNN computation onto the hardware function units of the accelerator.

Steps: Within the constraints of the hardware, select for each level of the storage hierarchy:

- A dataflow (**for** loop order)

Mapping

Definition: selecting the placement and scheduling in space and time of every operation (including delivering the appropriate operands) required for a DNN computation onto the hardware function units of the accelerator.

Steps: Within the constraints of the hardware, select for each level of the storage hierarchy:

- A dataflow (**for** loop order)
- A partitioning (**for** loop limits) both spatial and temporal

Mapping

Definition: selecting the placement and scheduling in space and time of every operation (including delivering the appropriate operands) required for a DNN computation onto the hardware function units of the accelerator.

Steps: Within the constraints of the hardware, select for each level of the storage hierarchy:

- A dataflow (**for** loop order)
- A partitioning (**for** loop limits) both spatial and temporal
- Other behavioral details…, e.g., bypassing

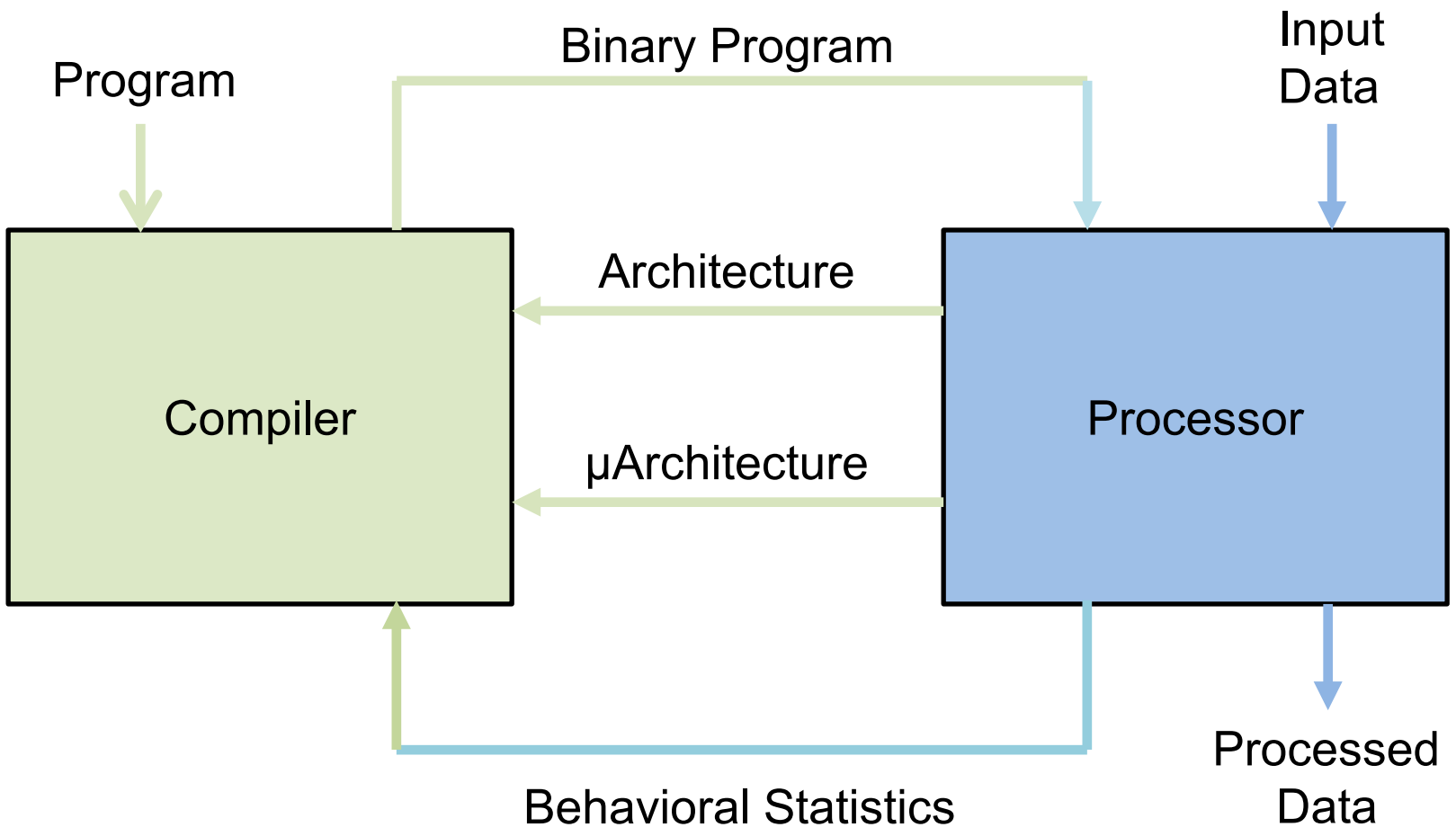
Mapping

Definition: selecting the placement and scheduling in space and time of every operation (including delivering the appropriate operands) required for a DNN computation onto the hardware function units of the accelerator.

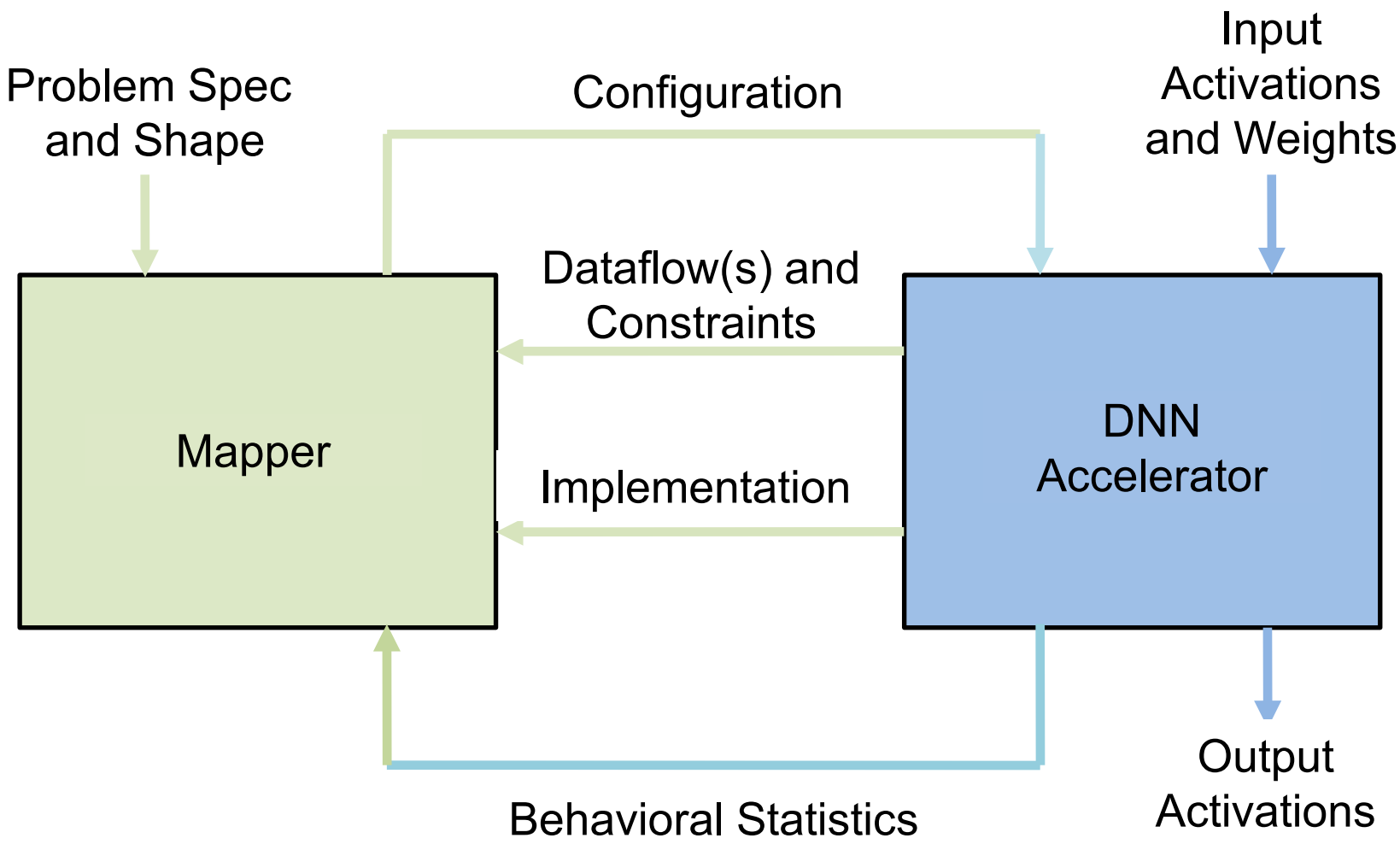
Steps: Within the constraints of the hardware, select for each level of the storage hierarchy:

- A dataflow (**for** loop order)
- A partitioning (**for** loop limits) both spatial and temporal
- Other behavioral details…, e.g., bypassing
- A binding computation to specific hardware units

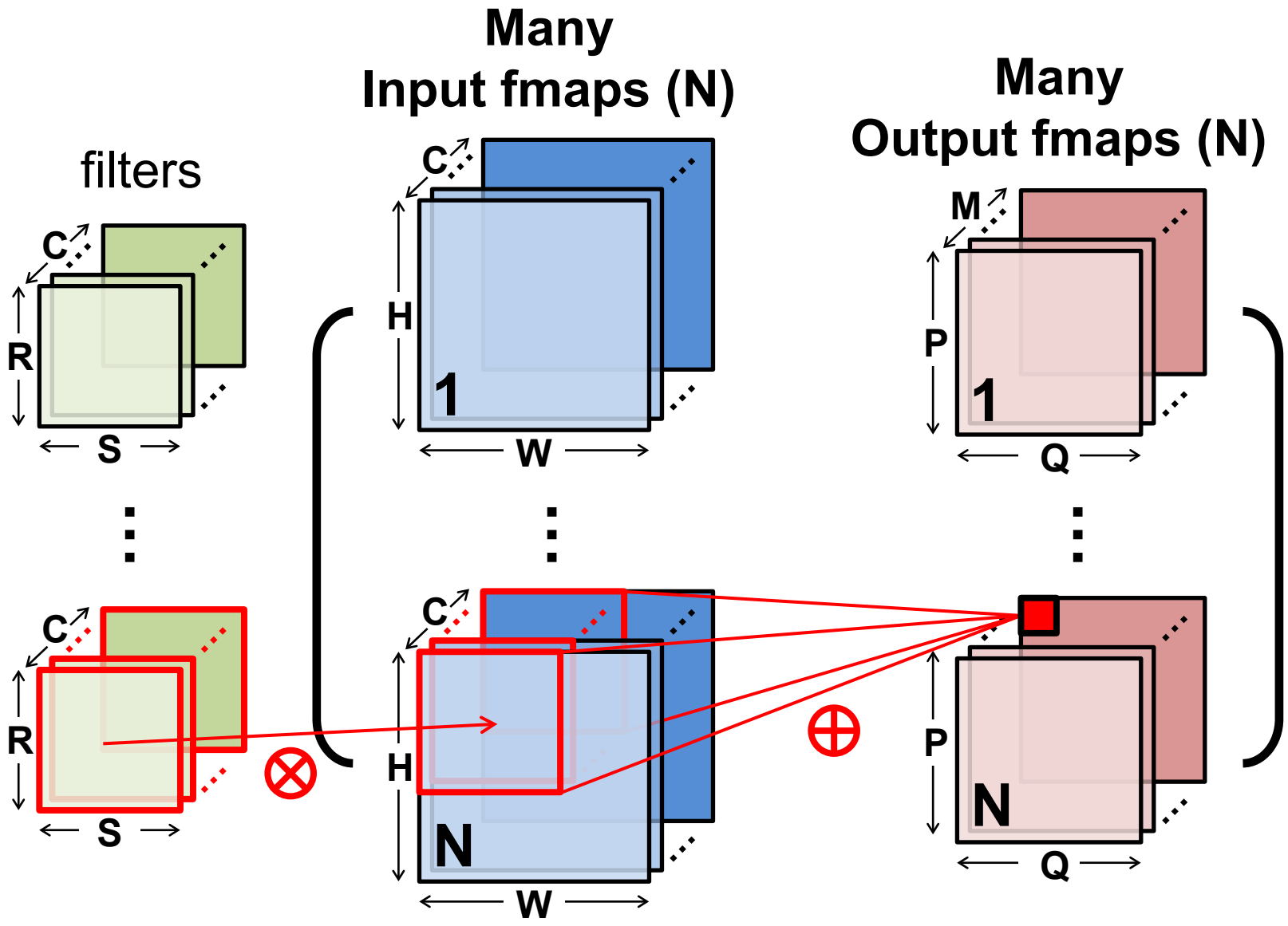
CPU Compute Model



DNN Compute Model

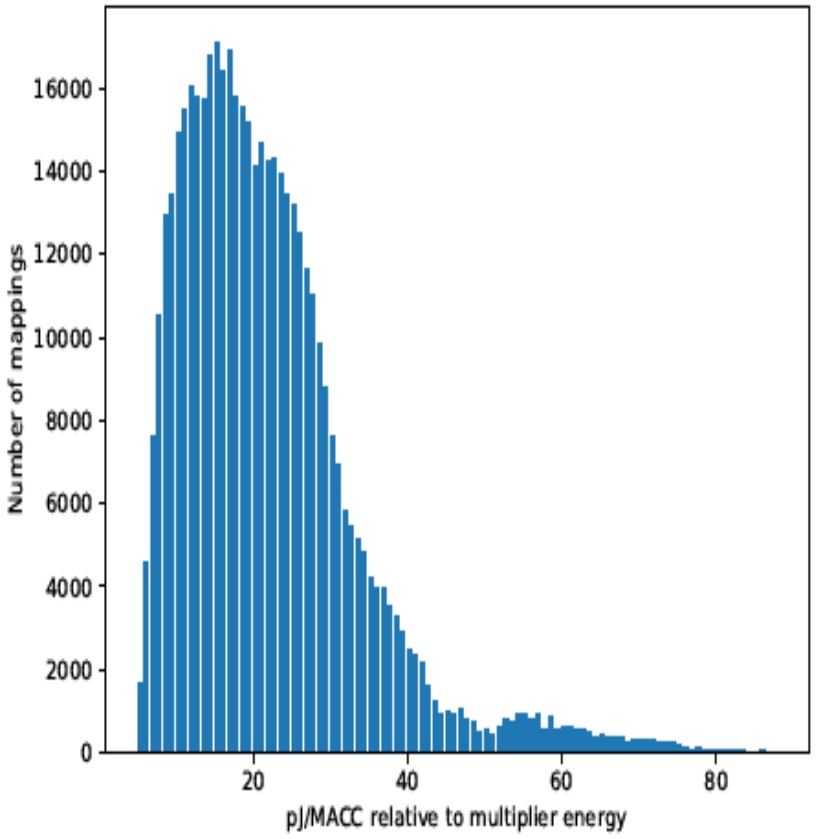


Convolution (CONV) Layer



Mapping Choices

Energy-efficiency of peak-perf mappings of a single problem



480,000 mappings shown

Spread: 19x in energy efficiency

Only 1 is optimal, 9 others within 1%

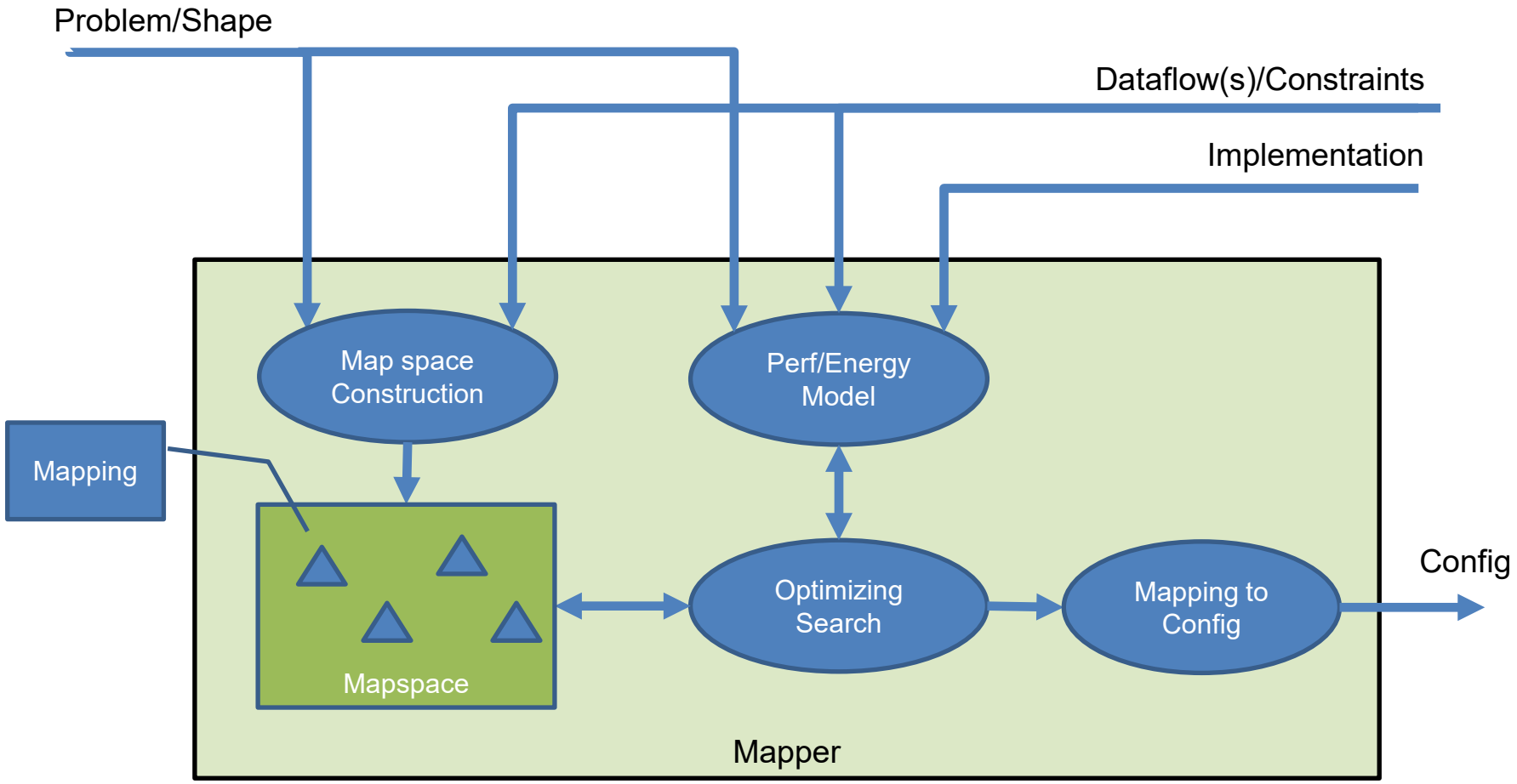
A model needs a mapper to evaluate a DNN workload on an architecture

6,582 mappings have min. DRAM accesses but vary 11x in energy efficiency

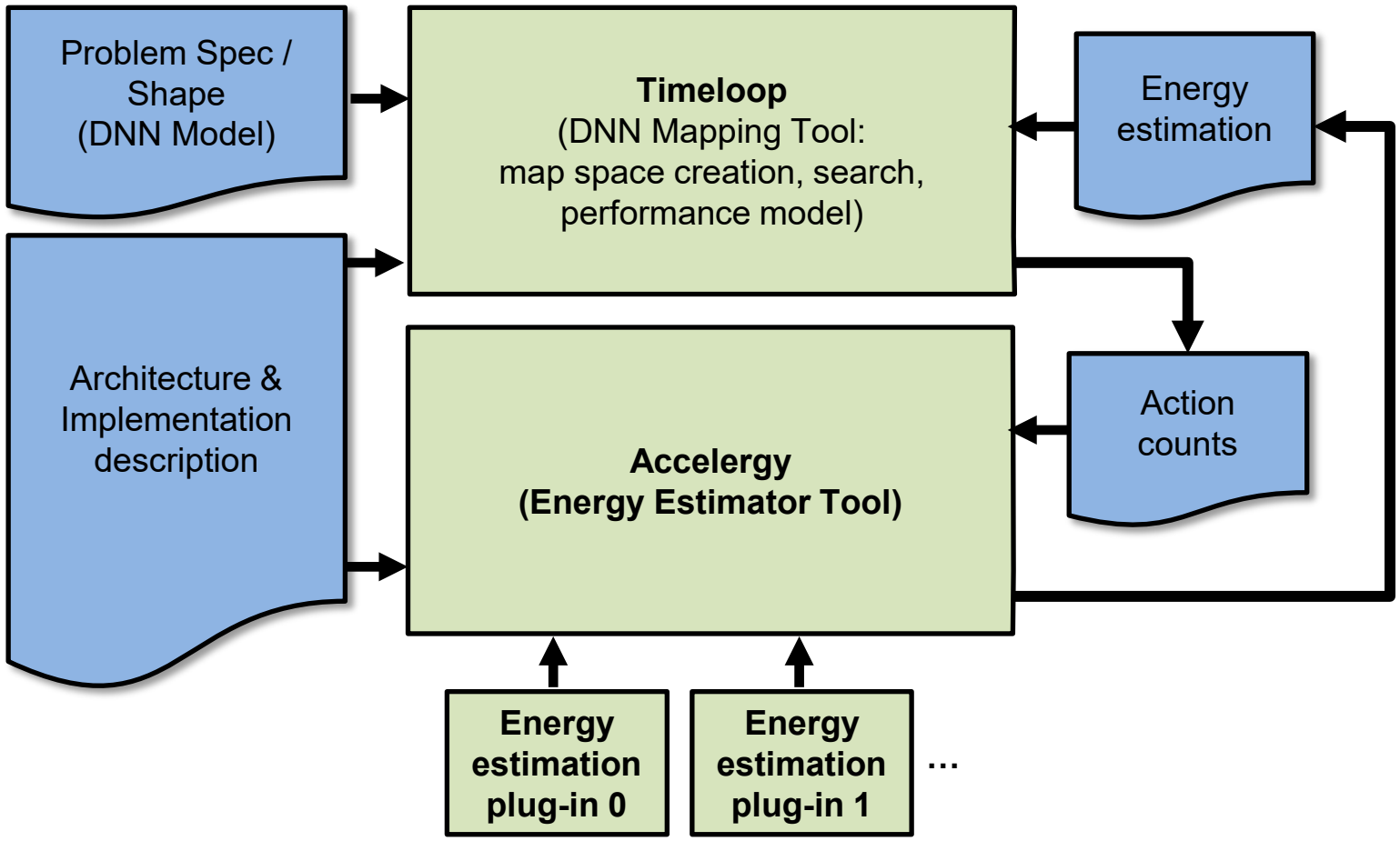
A mapper needs a good cost model to find an optimal mapping

Source: Parashar, Timeloop

Mapper Organization



Timeloop Accelergy

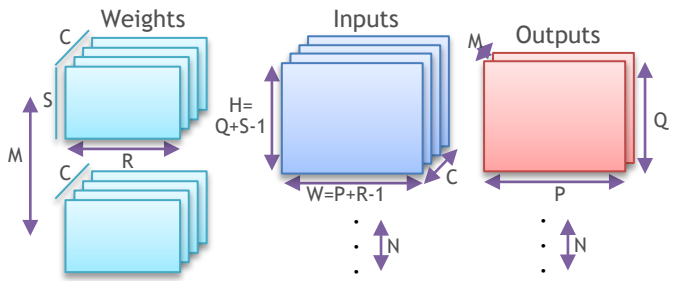


Workload Specification

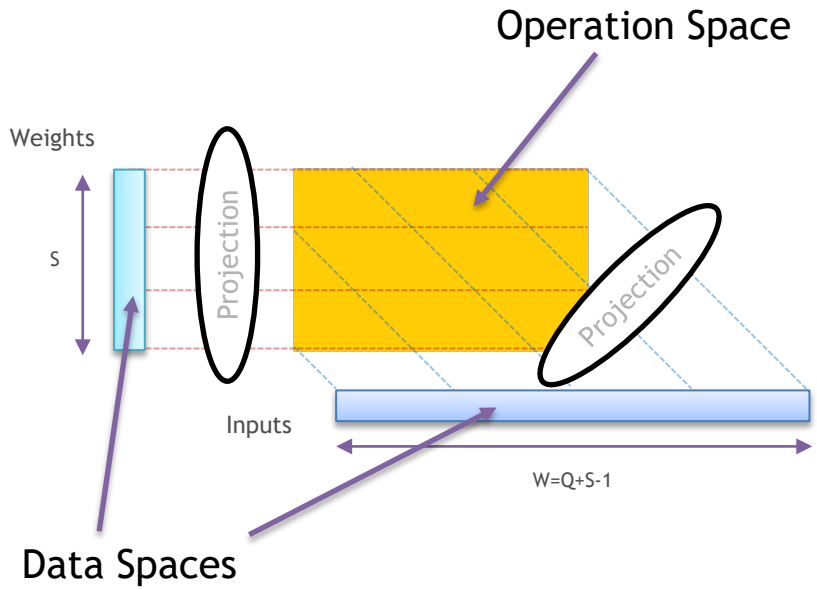
- Deep Loop Nest

Provided to Timeloop

$$O_{n,m,p,q} = I_{n,c,Up+r,Uq+s} \times F_{m,c,r,s}$$



Inferred by Timeloop



Source: Parashar, Timeloop

Architecture Specifications

- Temporal reuse features
  - Number of buffer levels and buffer sizes
  - Buffer bypassing capabilities
  - Network topology
  - …
- Parallelism and spatial reuse features
  - Topology of spatial fractures
  - Multicast capabilities
  - Inter-PE network, e.g., spatial sum and forwarding reuse
  - …
- Constraints
  - Index sequence restrictions, e.g., allowable strides
  - Fixed level 0 loop nest, e.g., fixed vector width
  - Fixed level 1 spatial mappings, e.g., input/output channel array
  - …

Determines the legal mappings:
loop permutations (dataflows) and associated loop limits

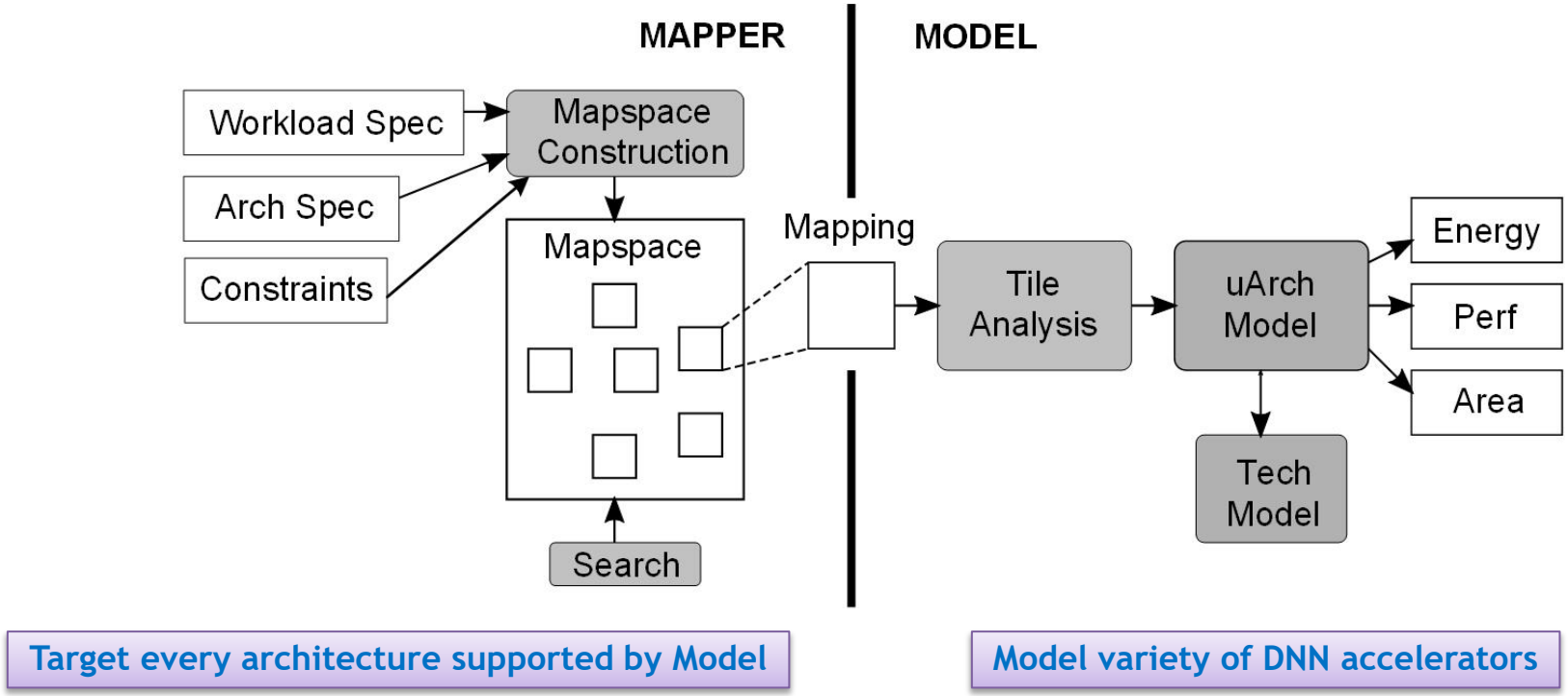
Implementation Specifications

- Buffer bandwidth and latency
- Buffer port and banking organization
- PE vectorization
- Network bandwidth and latency, e.g., router costs
- Shared or per-datatype network links
- …

Determines the latency and energy consumption of a mapping.

Timeloop

- Tool for Evaluation and Architectural Design-Space Exploration of DNN Accelerators



Source: Parashar, Timeloop

Next Lecture: Calculating Data Movement

Thank you!