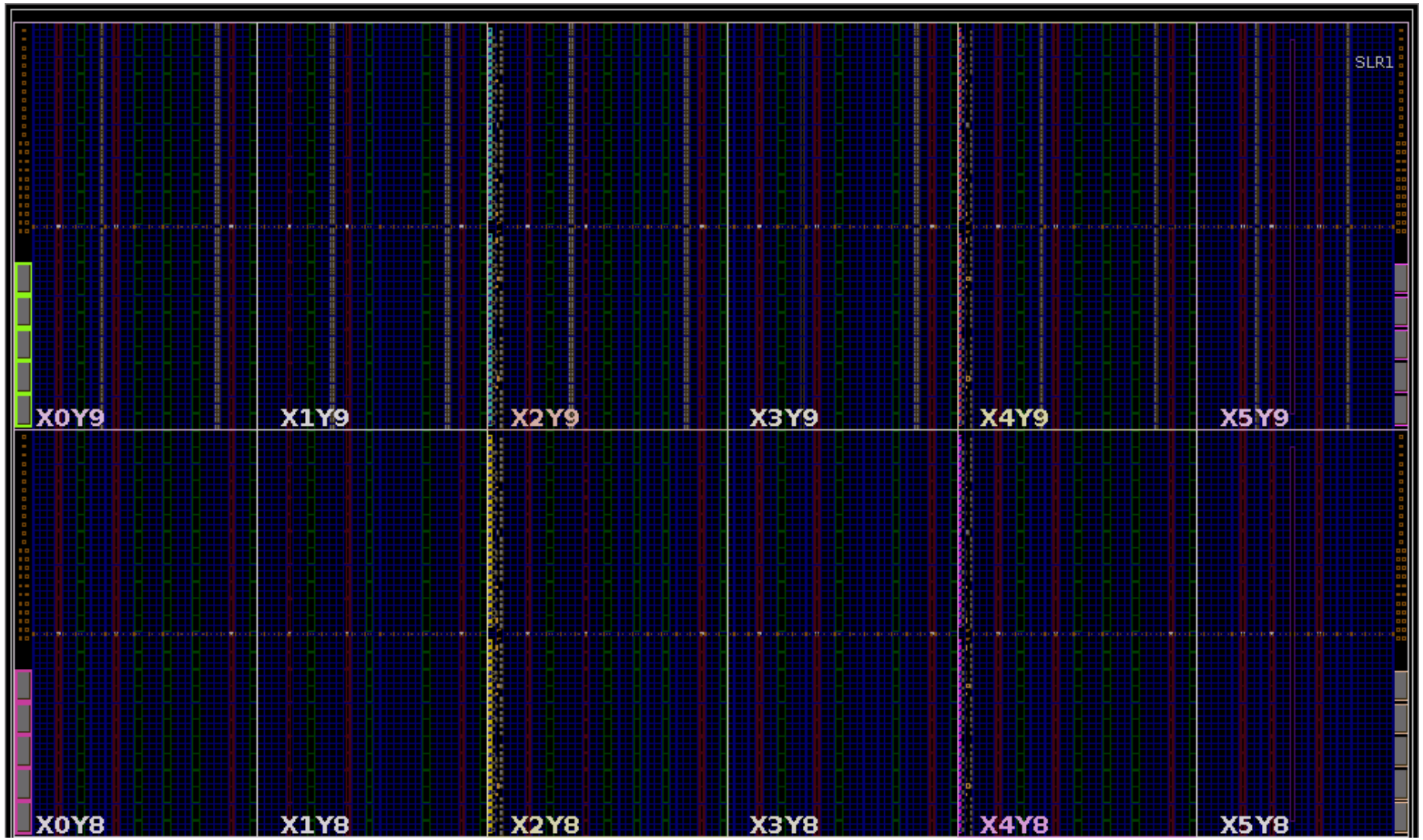
J. de Fine Licht and T. Hoefler

Productive parallel programming on FPGA using High-level Synthesis

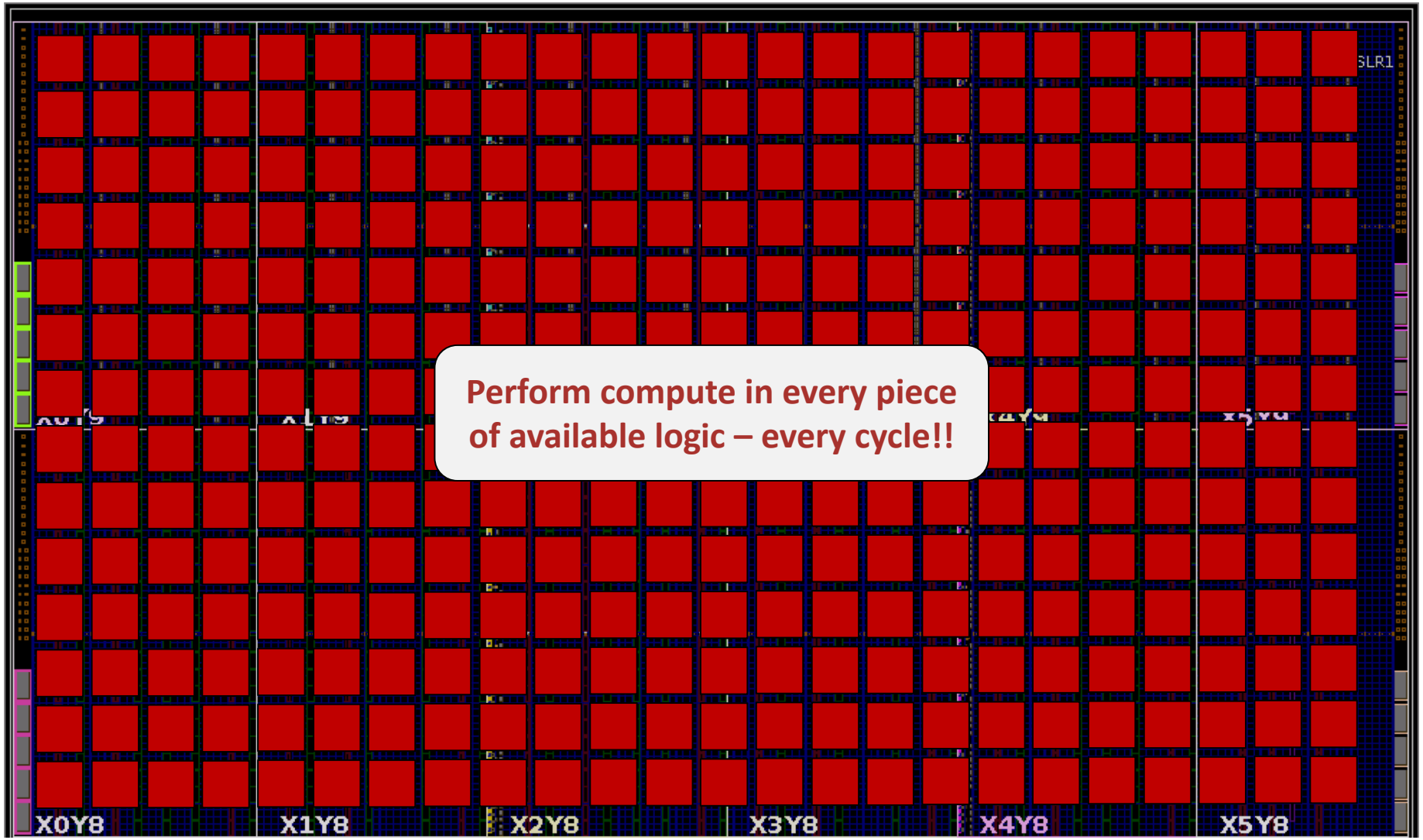
Code examples found at:

https://github.com/spcl/hls_tutorial_examples

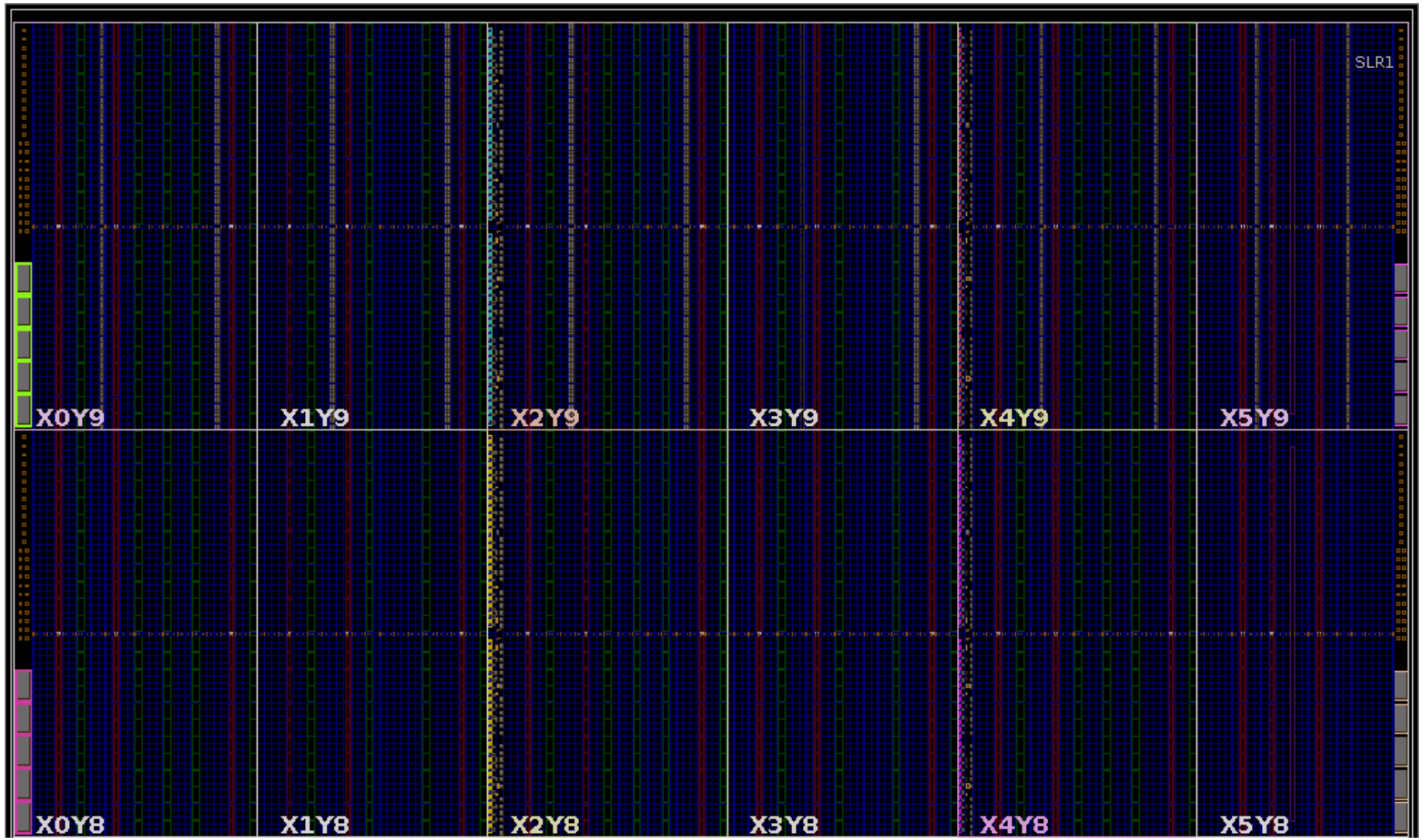
Our goal


SLR1
X0Y9
X1Y9
X2Y9
X3Y9
X4Y9
X5Y9
X0Y8
X1Y8
X2Y8
X3Y8
X4Y8
X5Y8

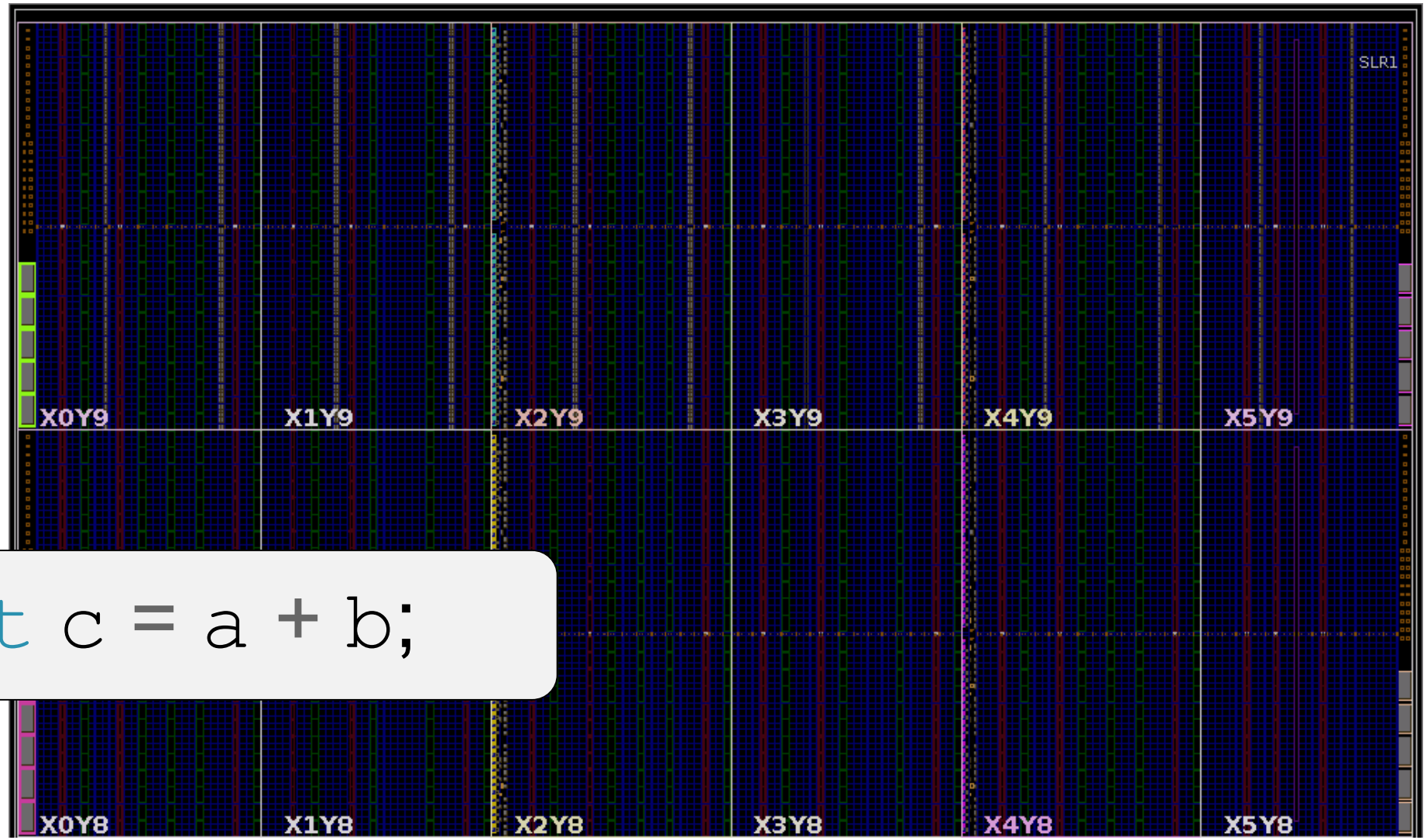
Our goal


SLR1
Perform compute in every piece of available logic – every cycle!!
X0Y8
X1Y8
X2Y8
X3Y8
X4Y8
X5Y8

A single operation

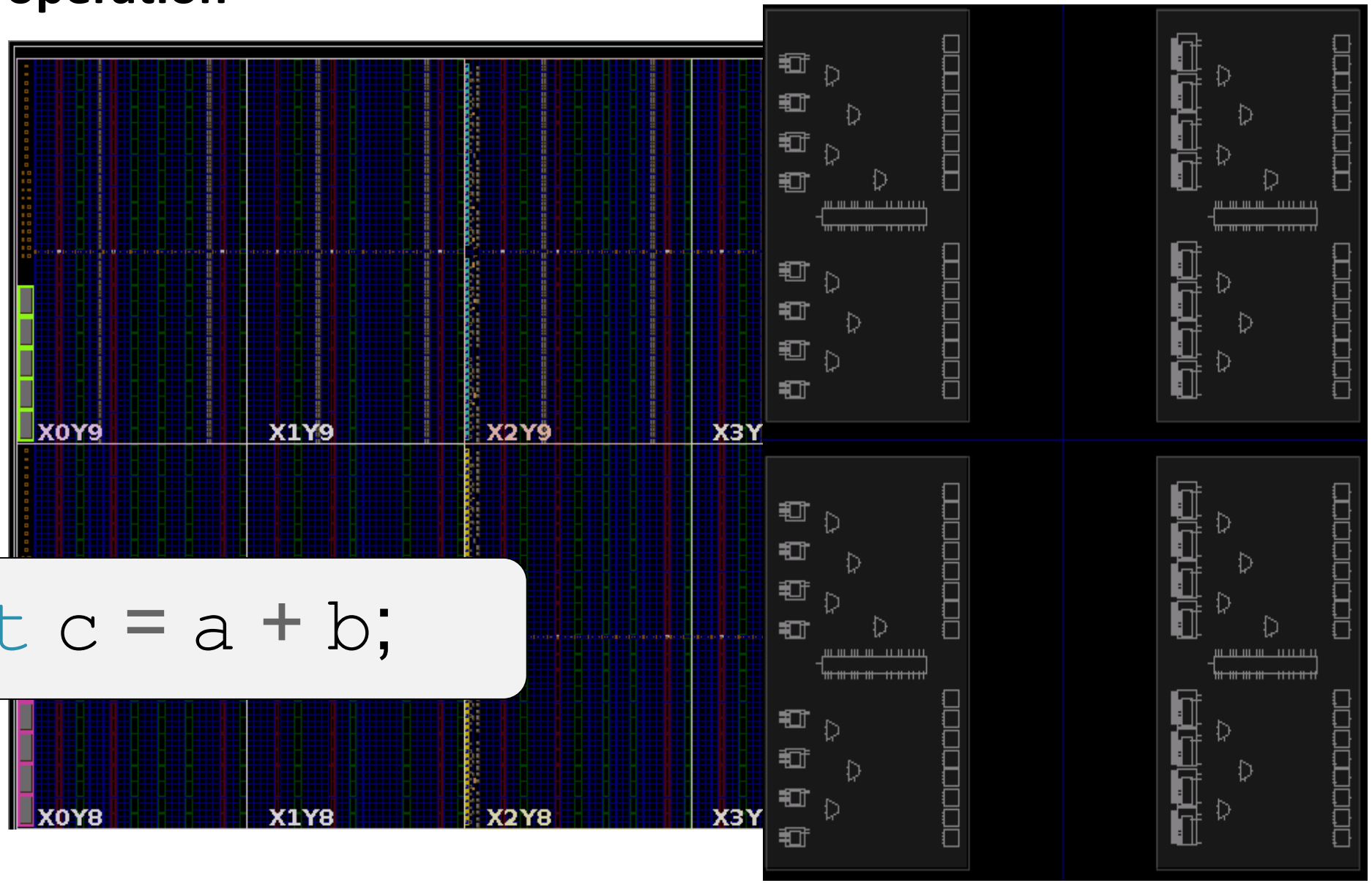


A single operation



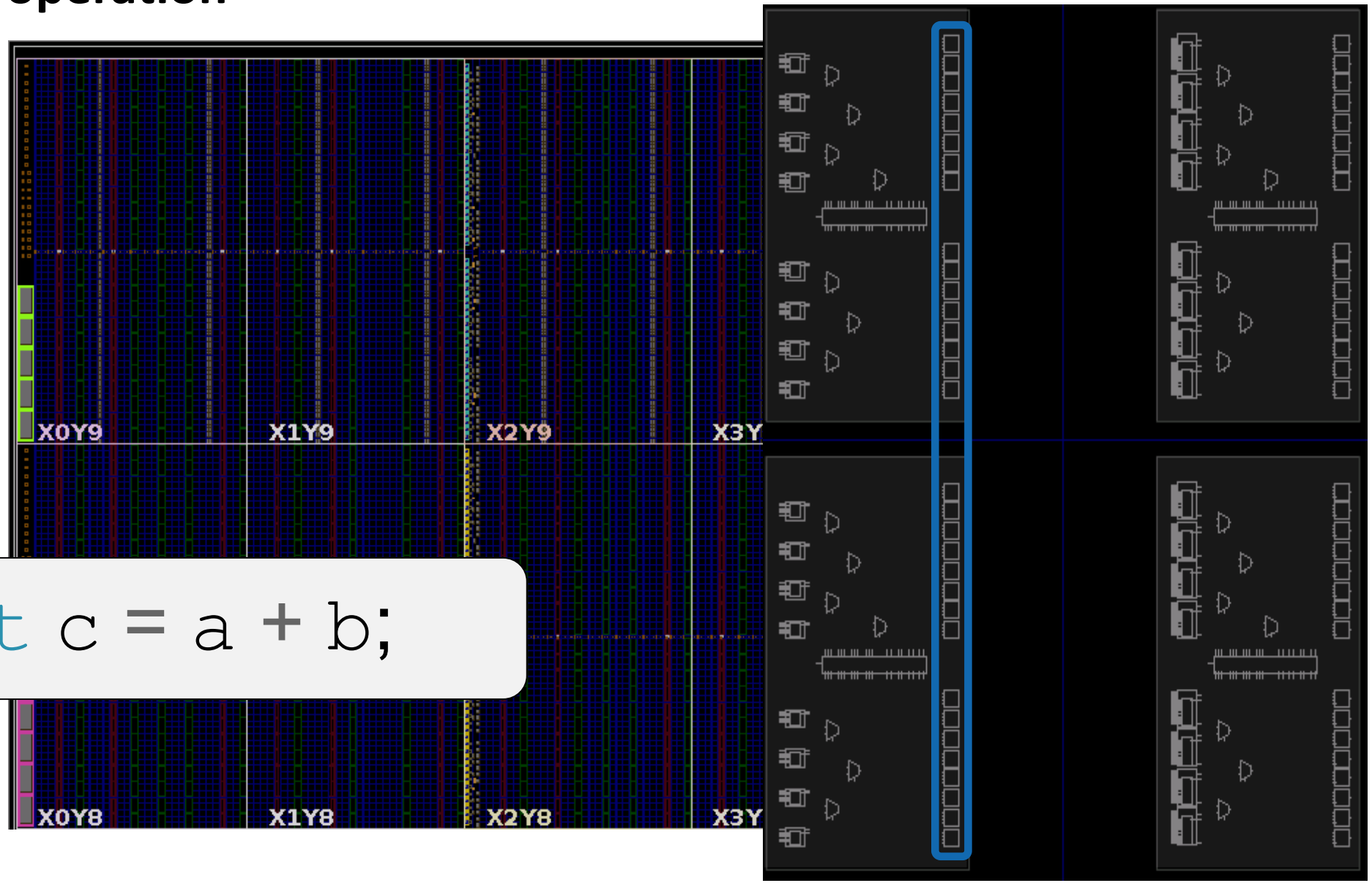

```
int c = a + b;
```

A single operation



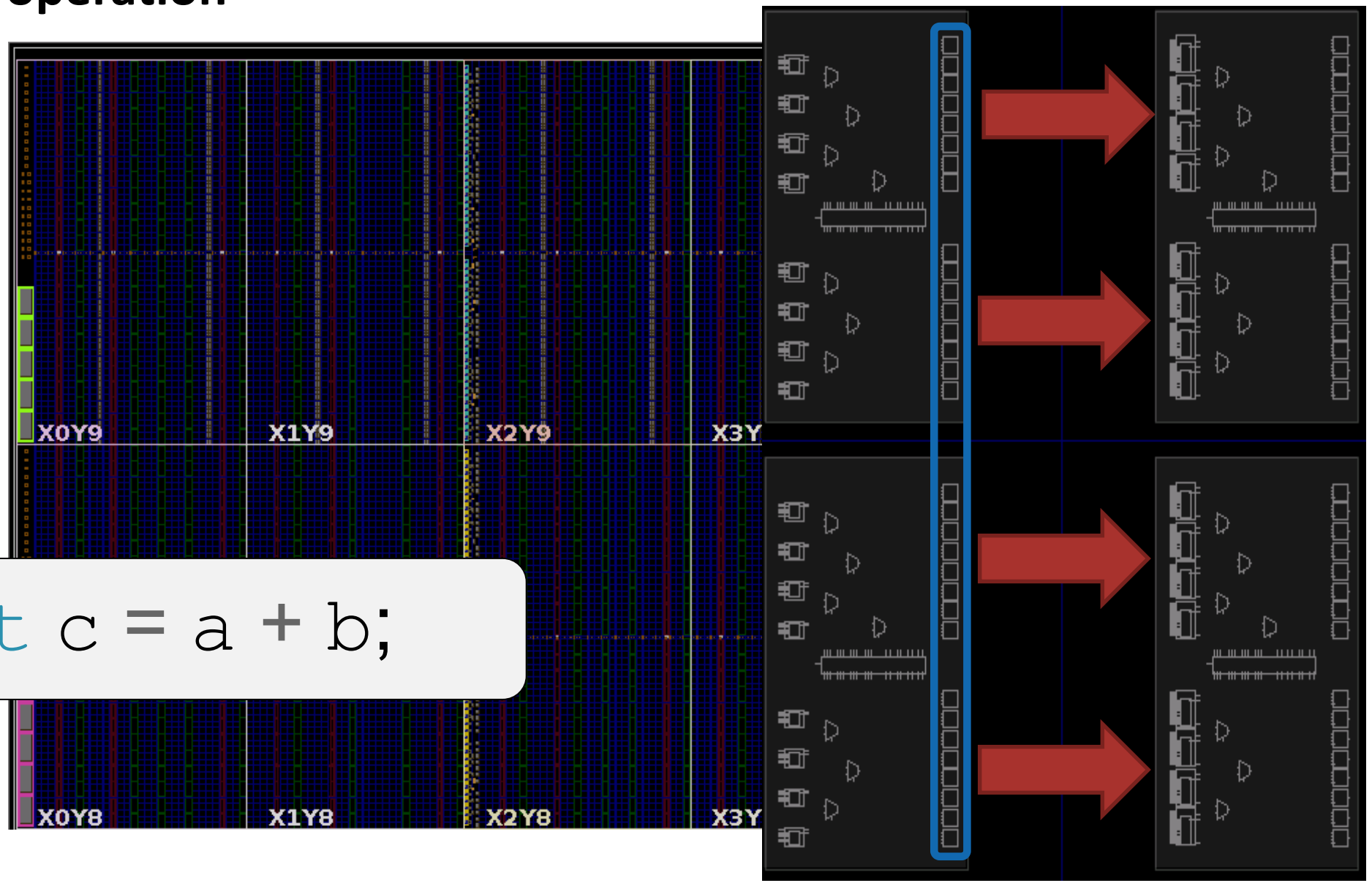

```
int c = a + b;
```

A single operation



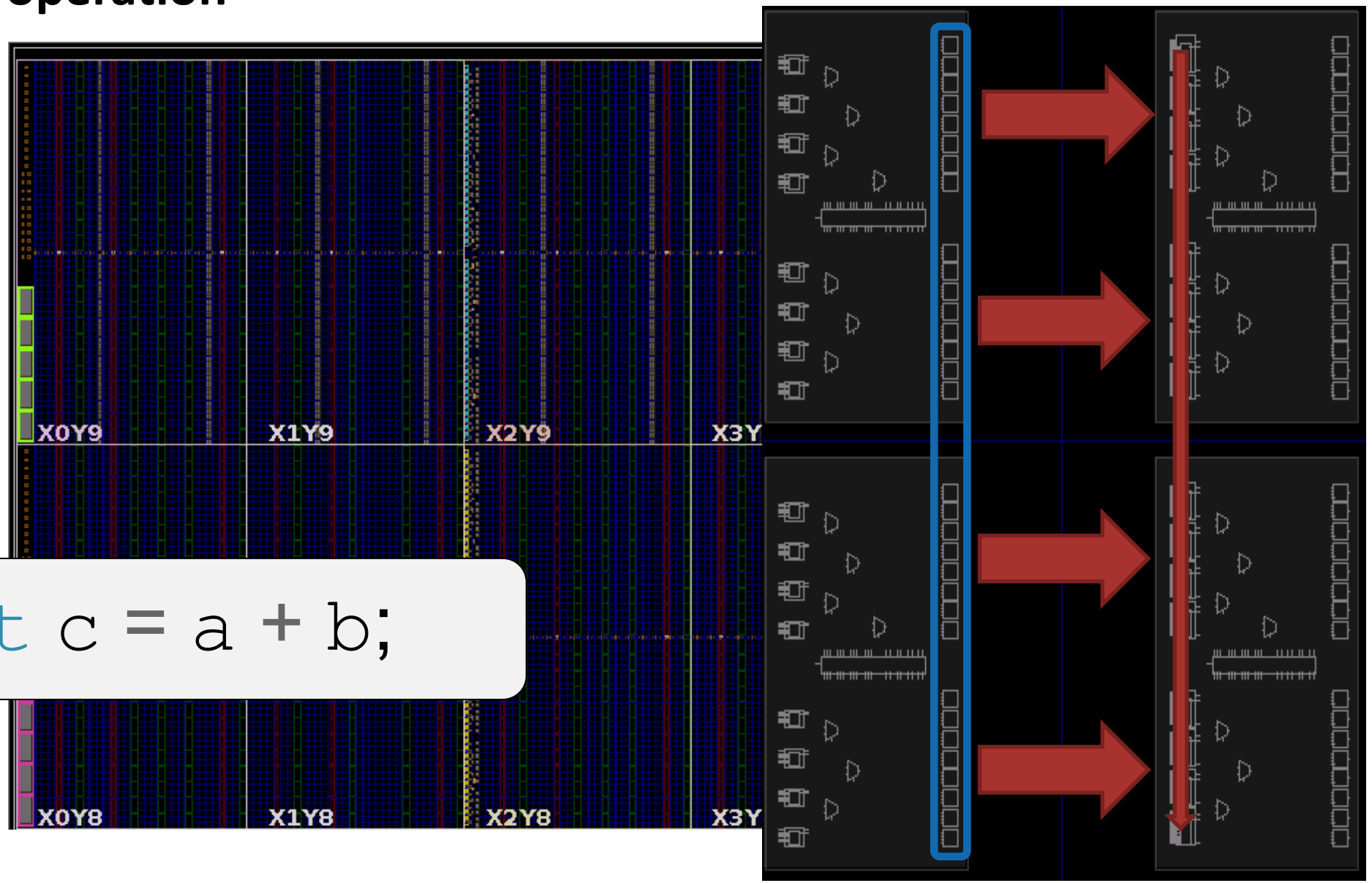

```
int c = a + b;
```

A single operation



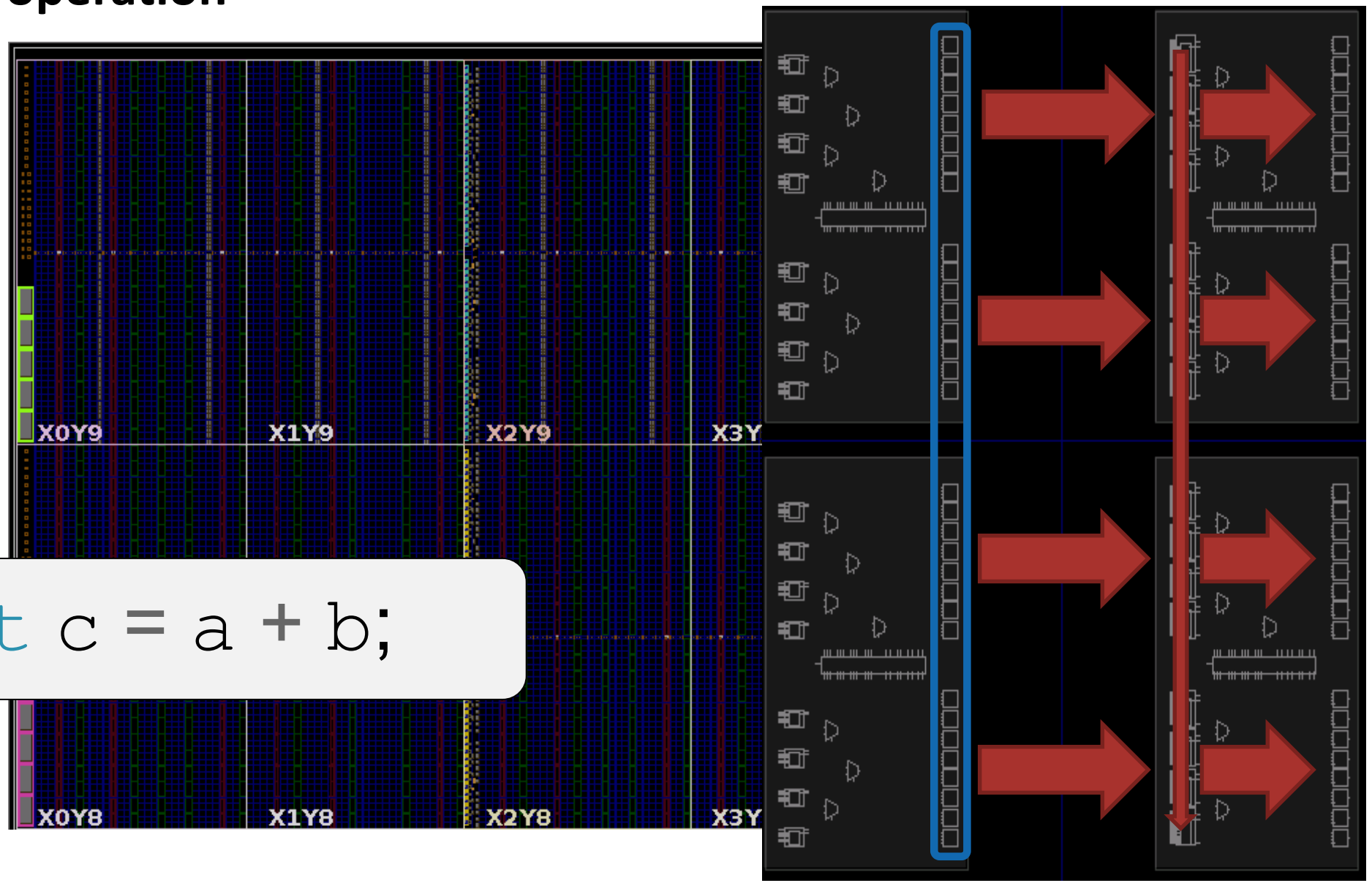

```
int c = a + b;
```

A single operation



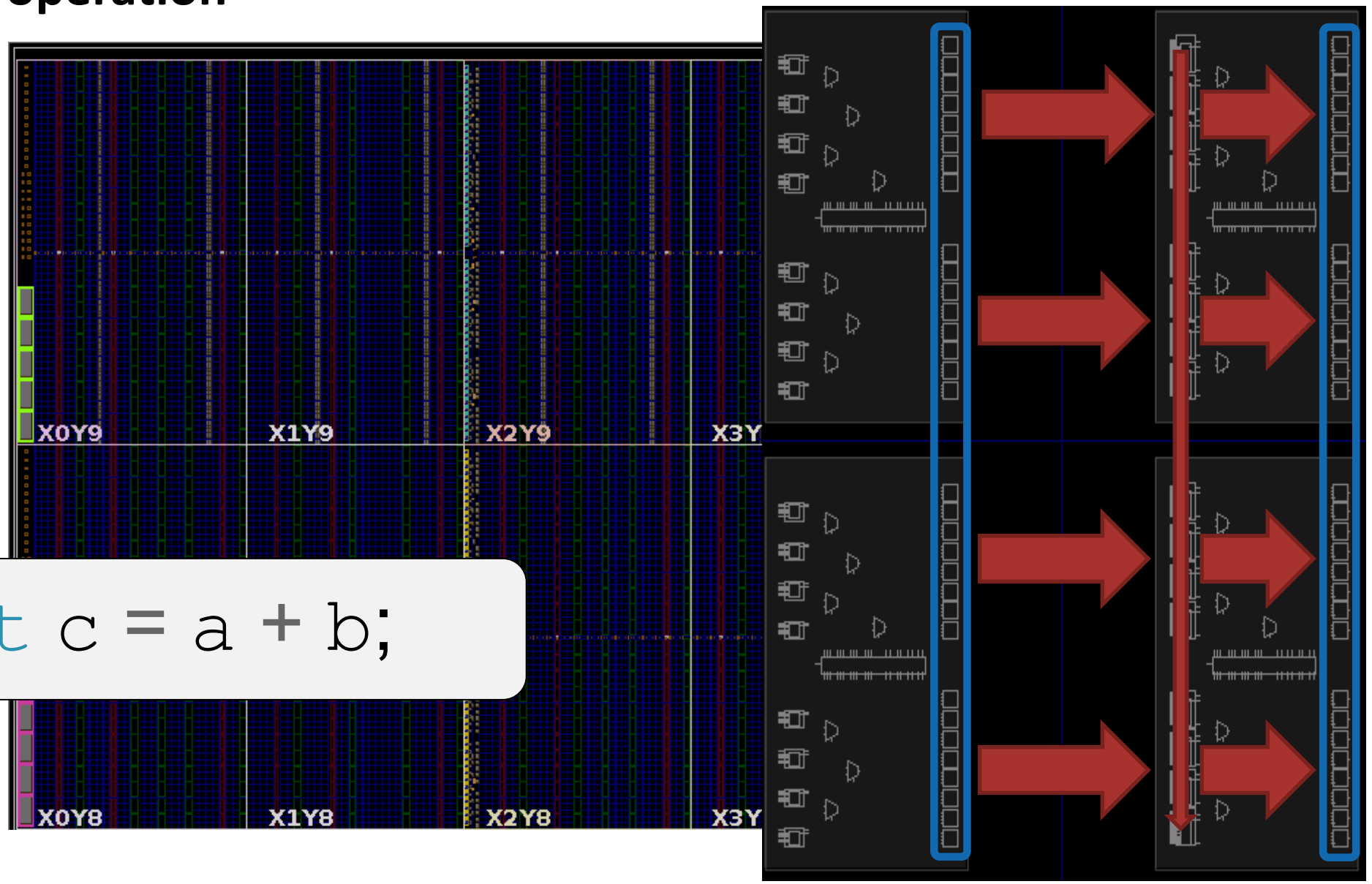

```
int c = a + b;
```

A single operation



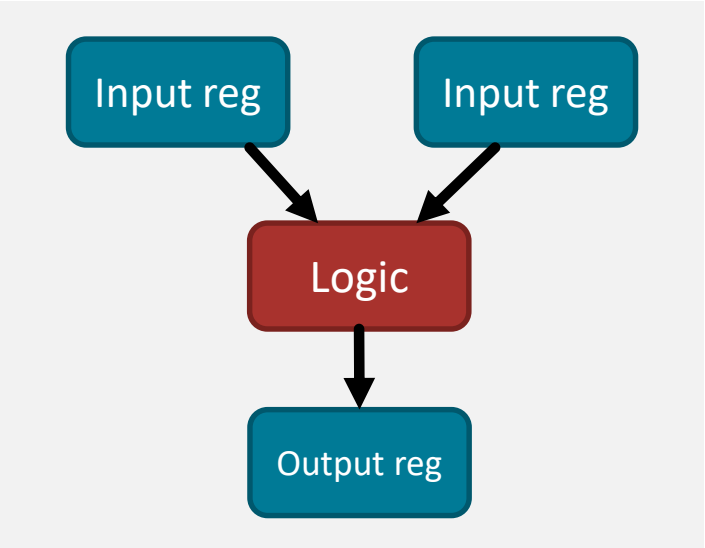

```
int c = a + b;
```

A single operation




```
int c = a + b;
```

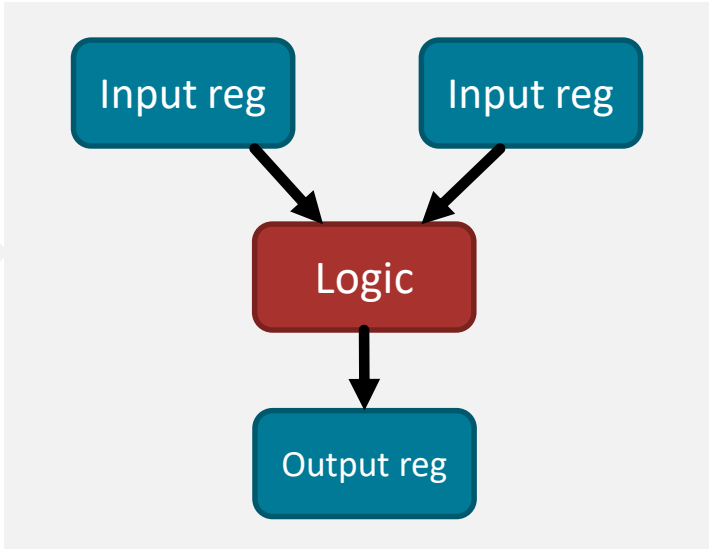
Register transfer level




```
int c = a + b;
```

Register transfer level

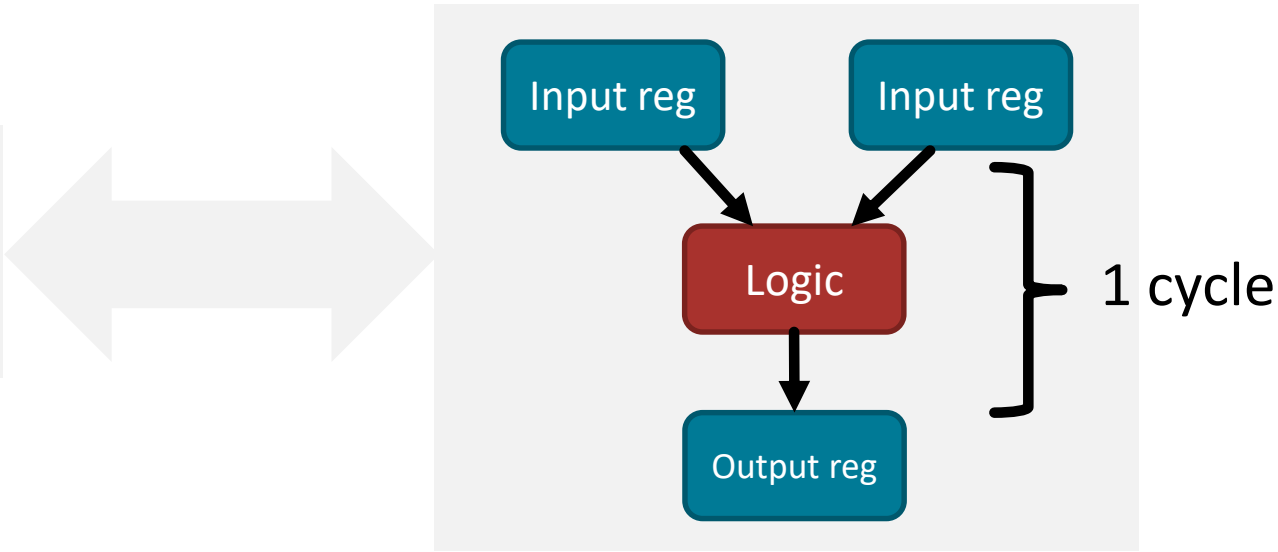
```
always @(posedge clk)
  if (start) begin
      out <= in + 1;
  end
```


```
int c = a + b;
```

Register transfer level

```
always @(posedge clk)
  if (start) begin
      out <= in + 1;
  end
```

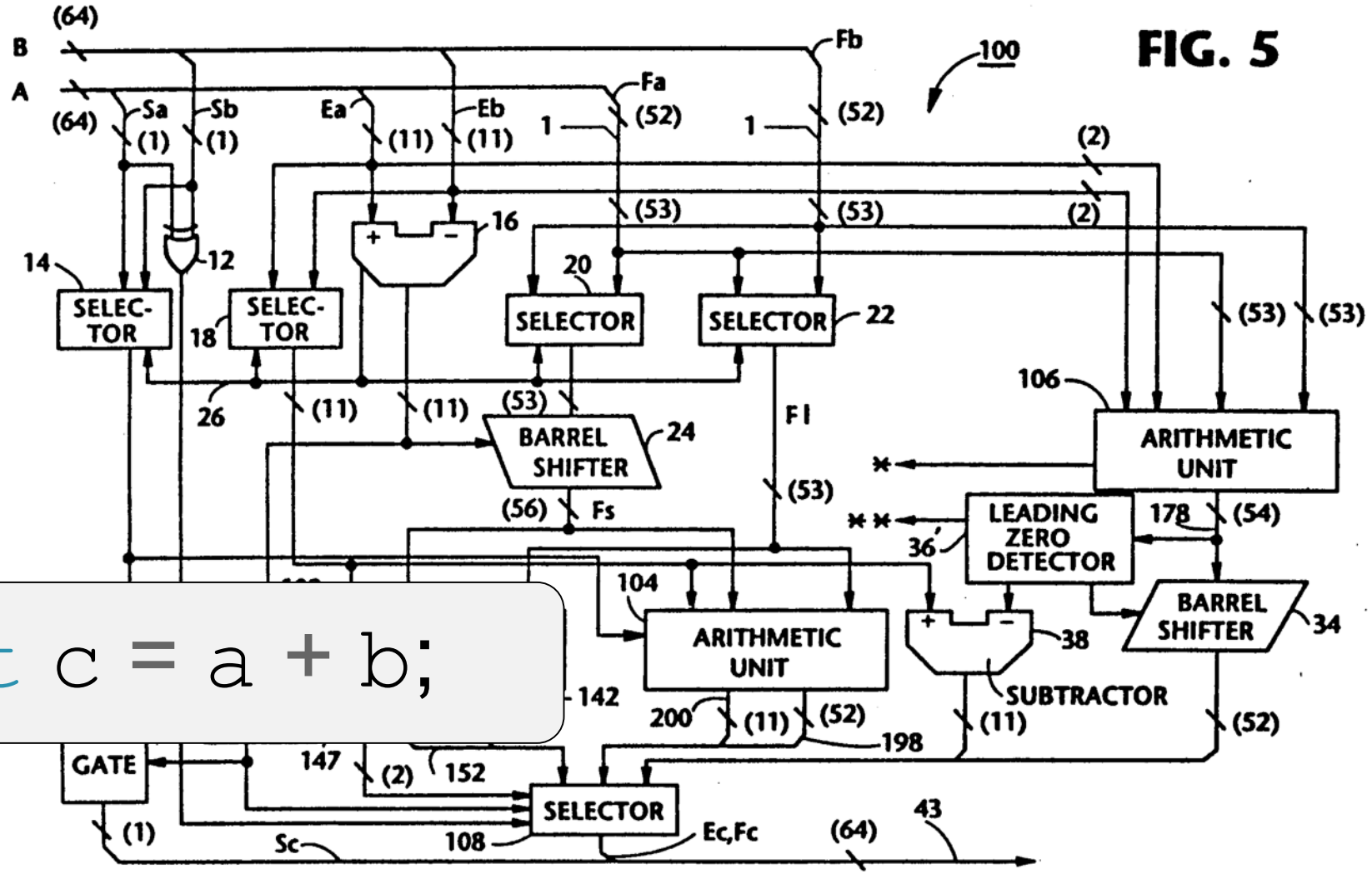



```
int c = a + b;
```

Single floating point operation

```
float c = a + b;
```

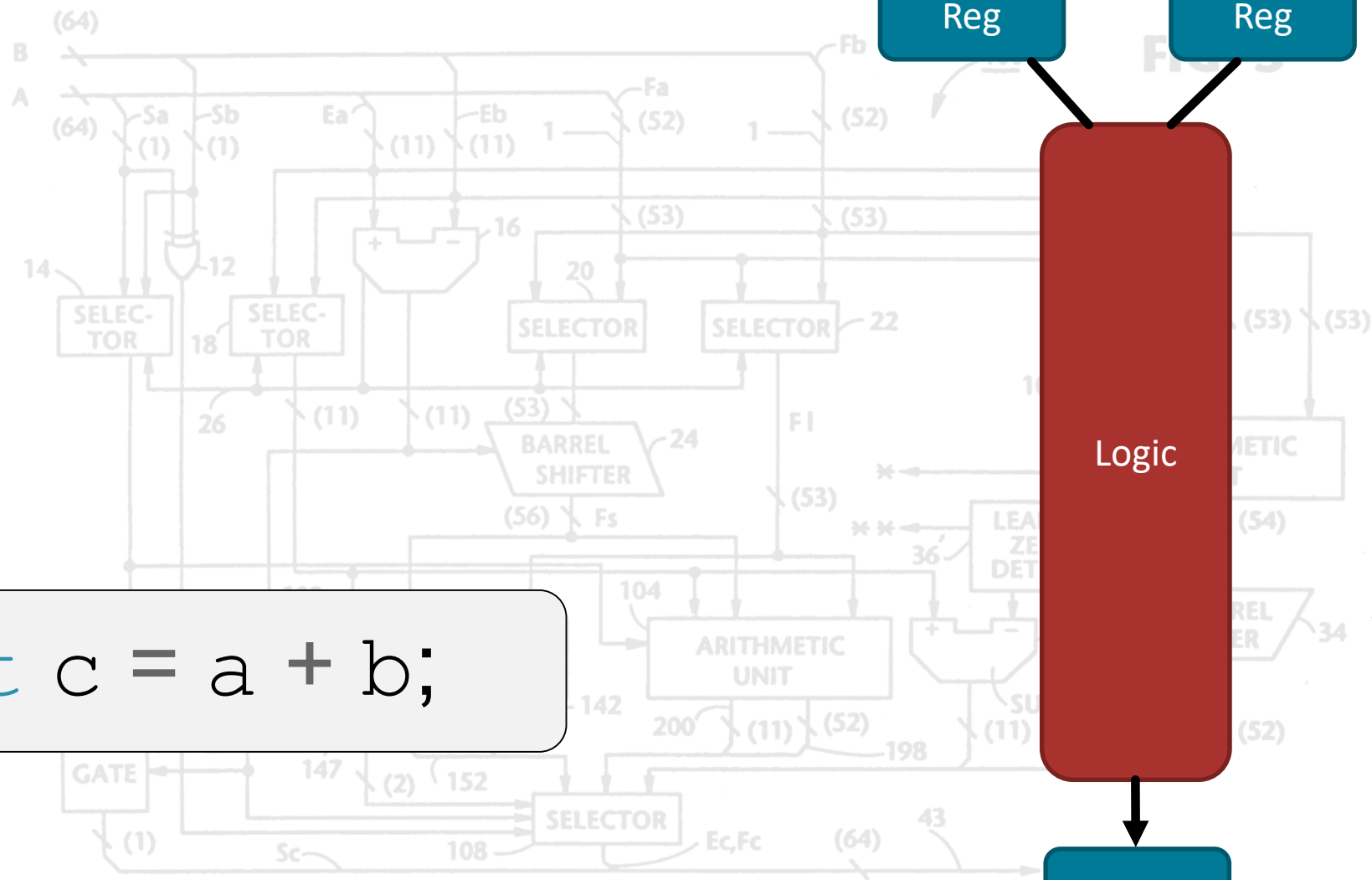
Single floating point operation




```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

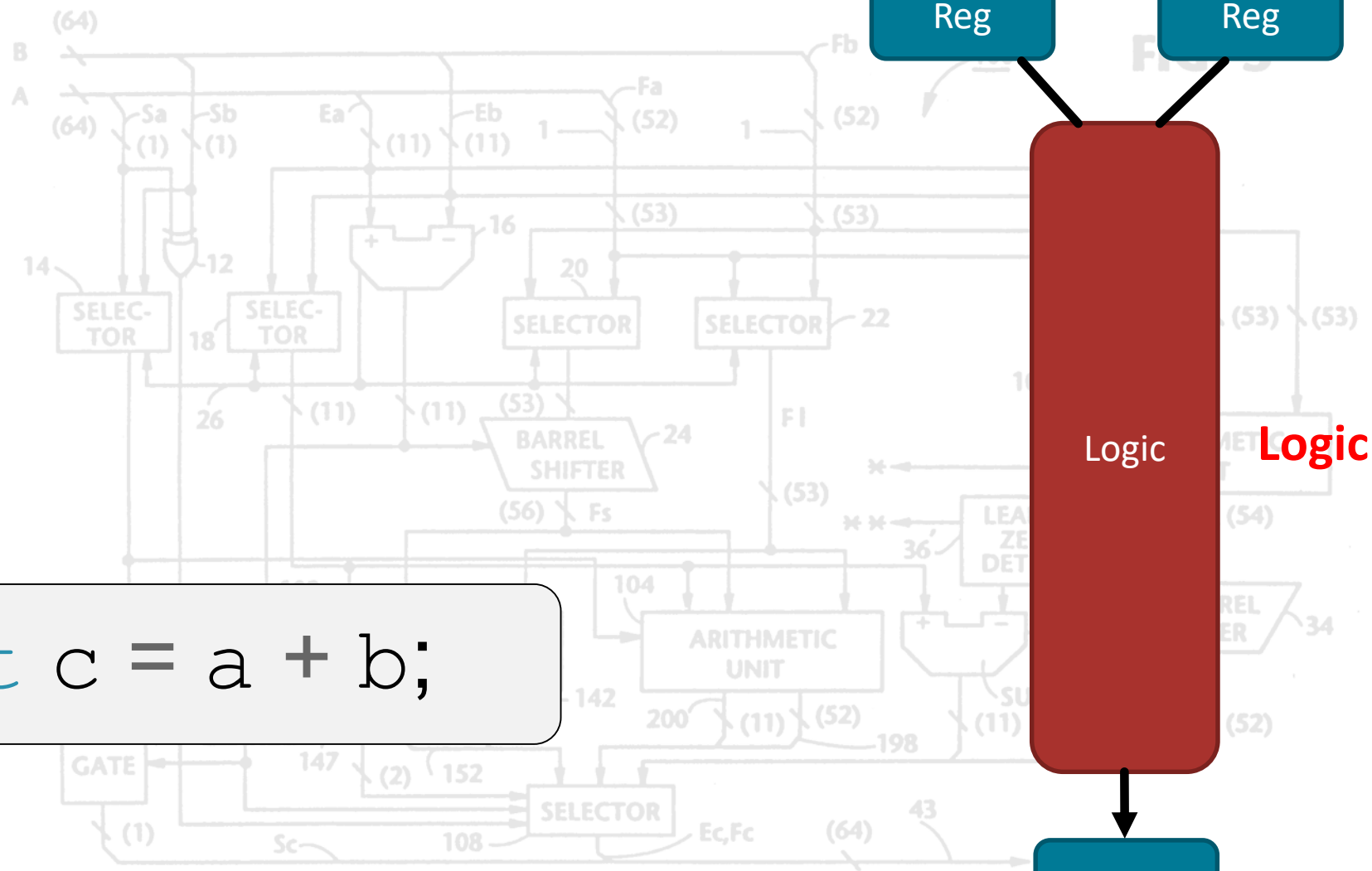
Single floating point operation




```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

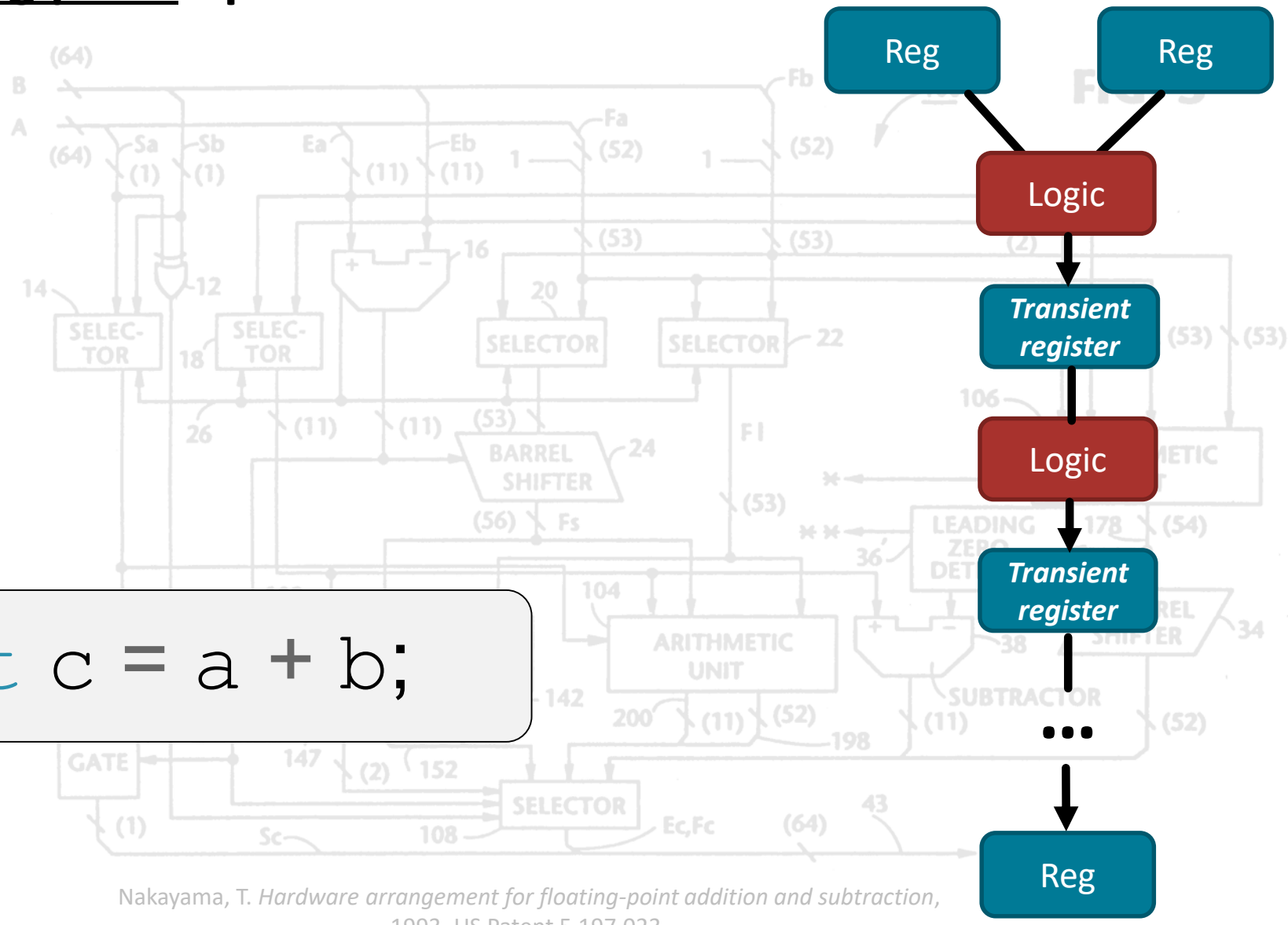
Single floating point operation




```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

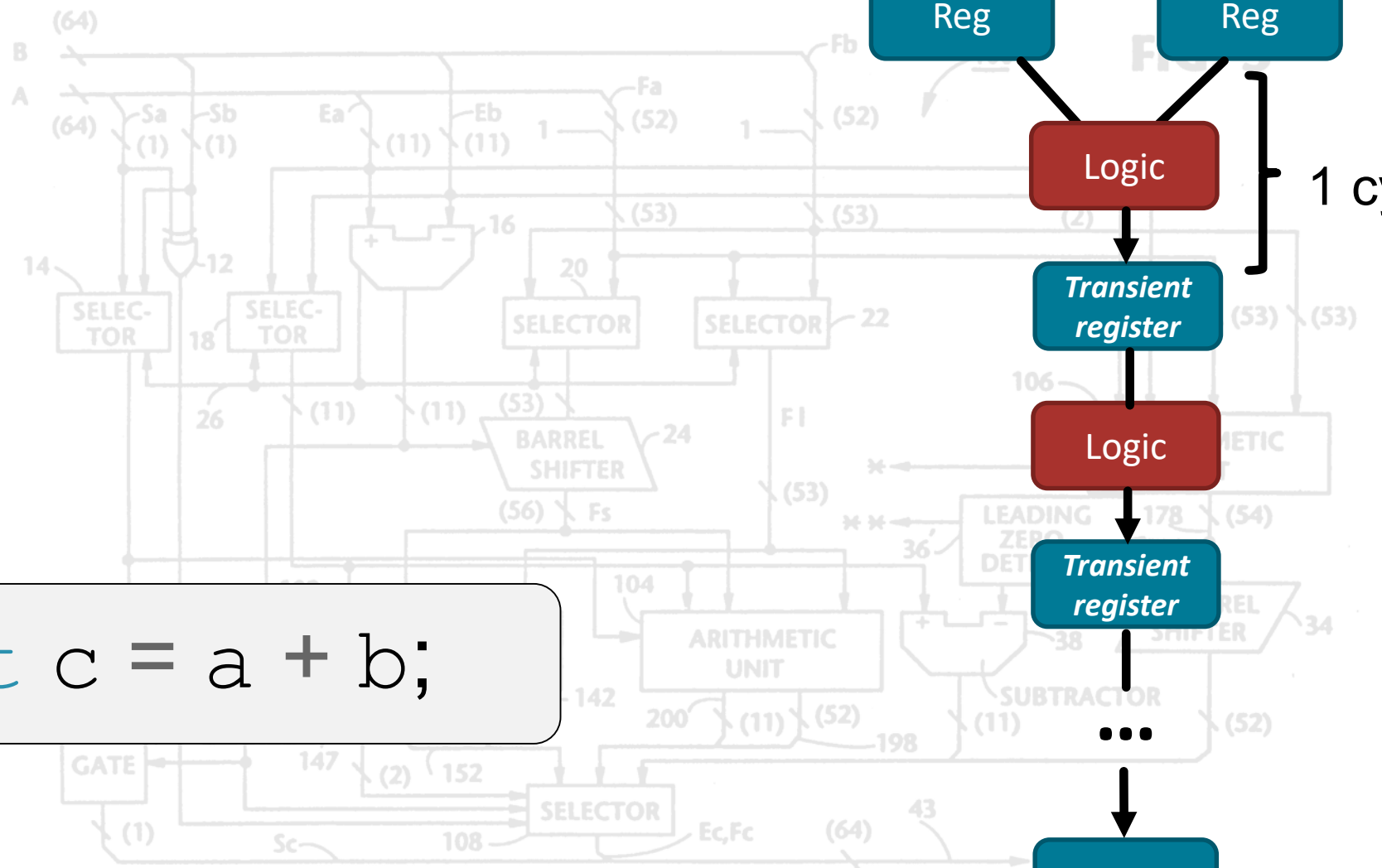
Single floating point operation




```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

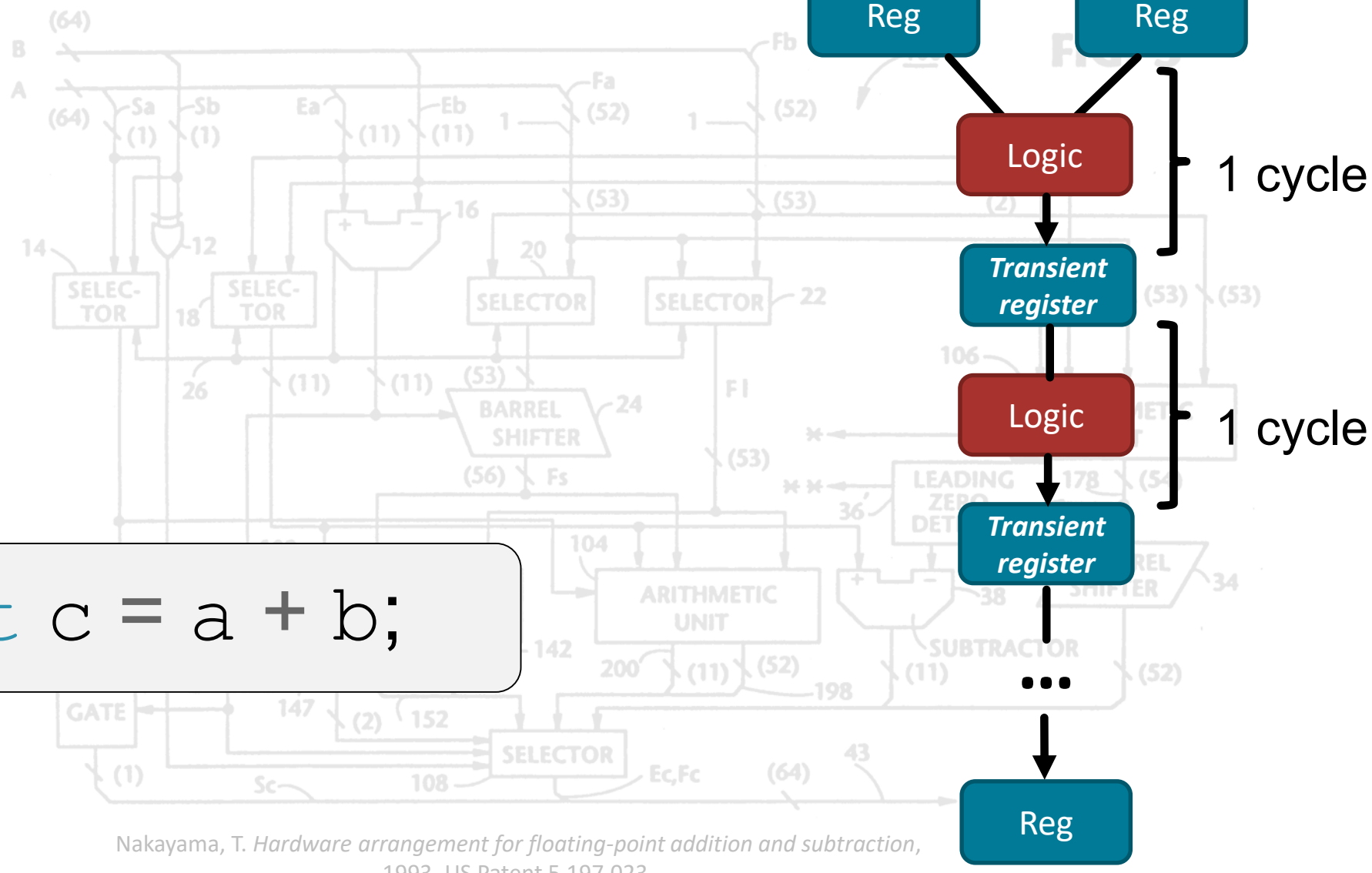
Single floating point operation




```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

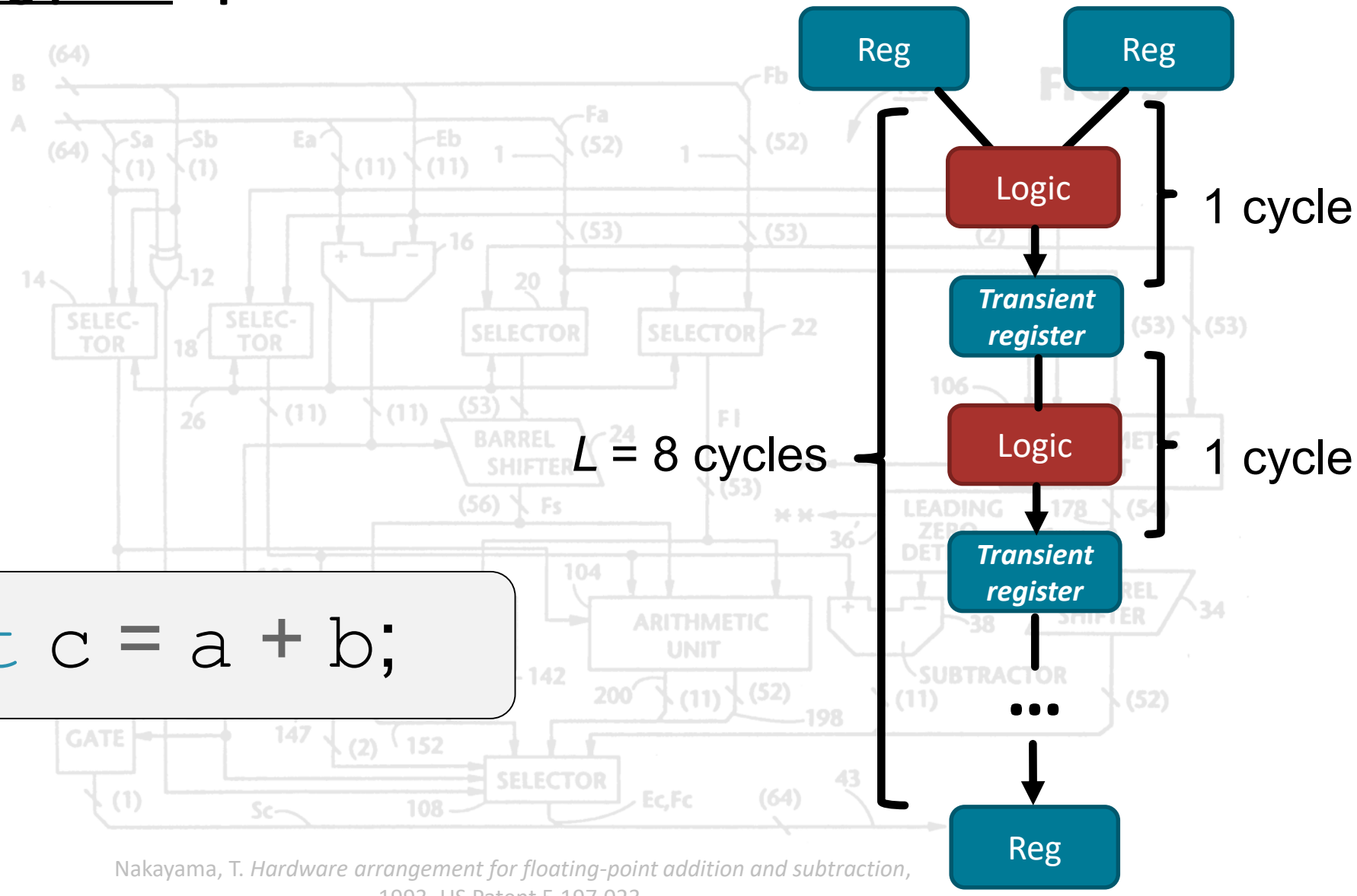
Single floating point operation




```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

Single floating point operation

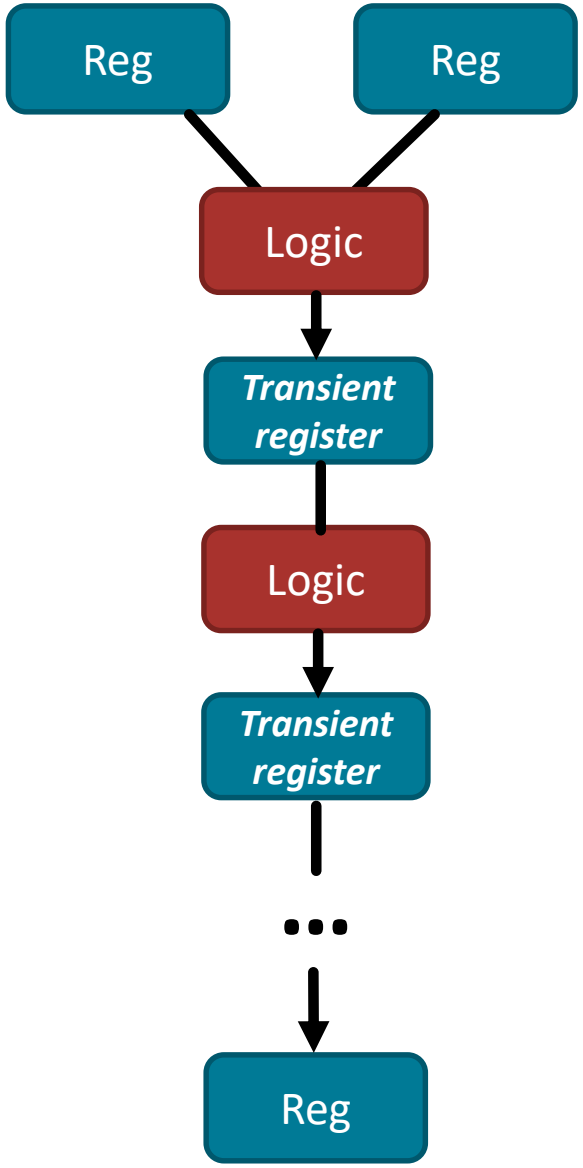



```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

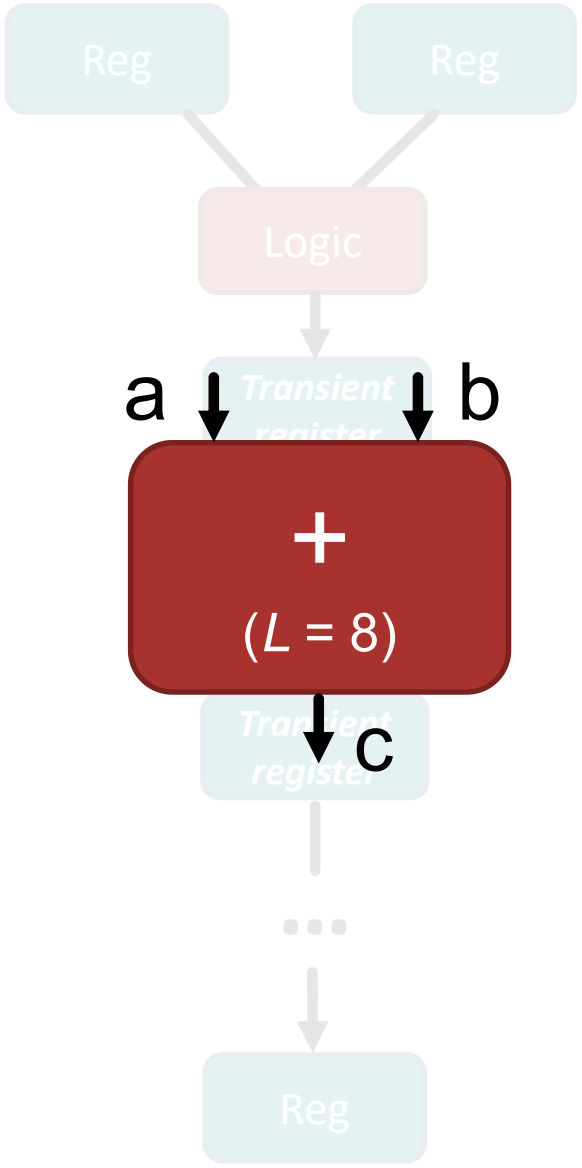
Our point of view

```
float c = a + b;
```



Our point of view

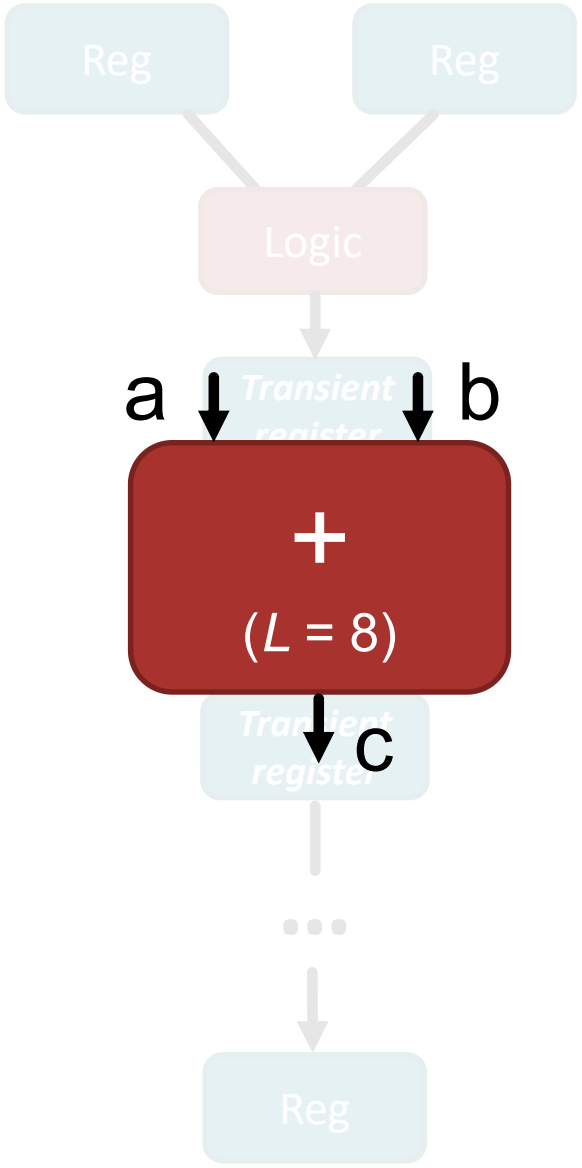
```
float c = a + b;
```



Our point of view

In HLS, we treat **pipelines**.

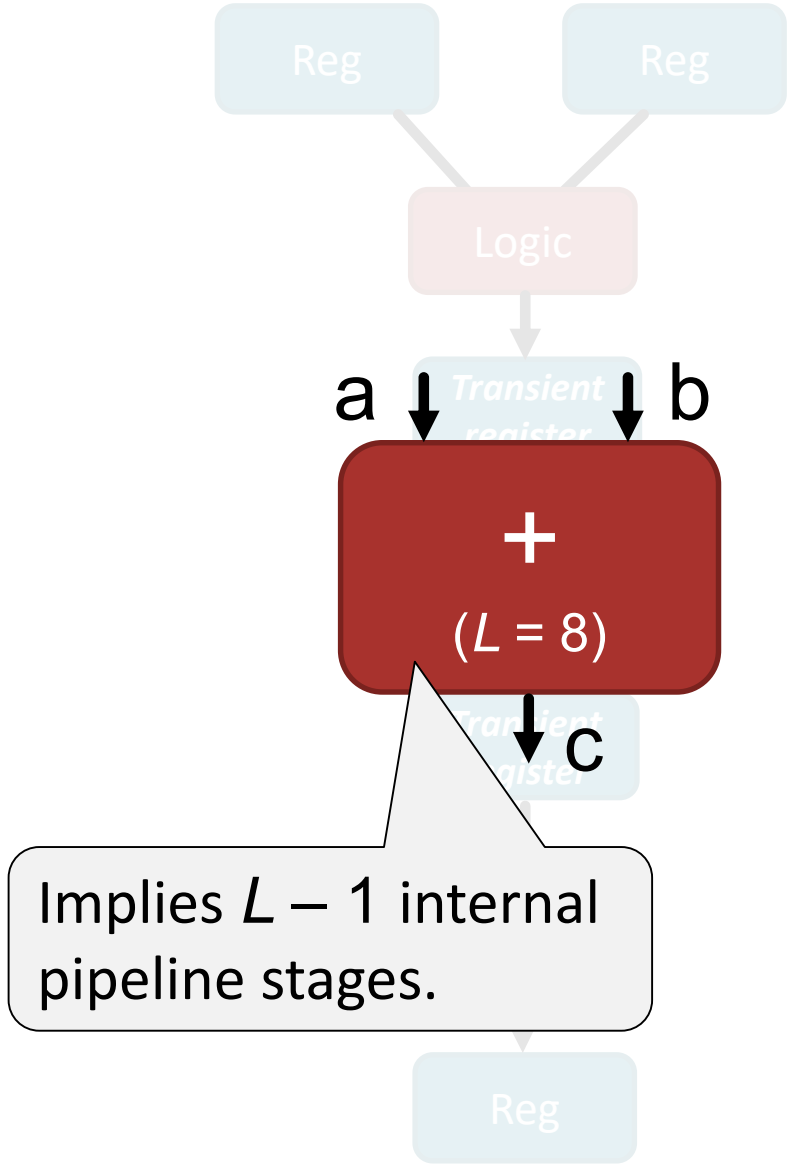
```
float c = a + b;
```



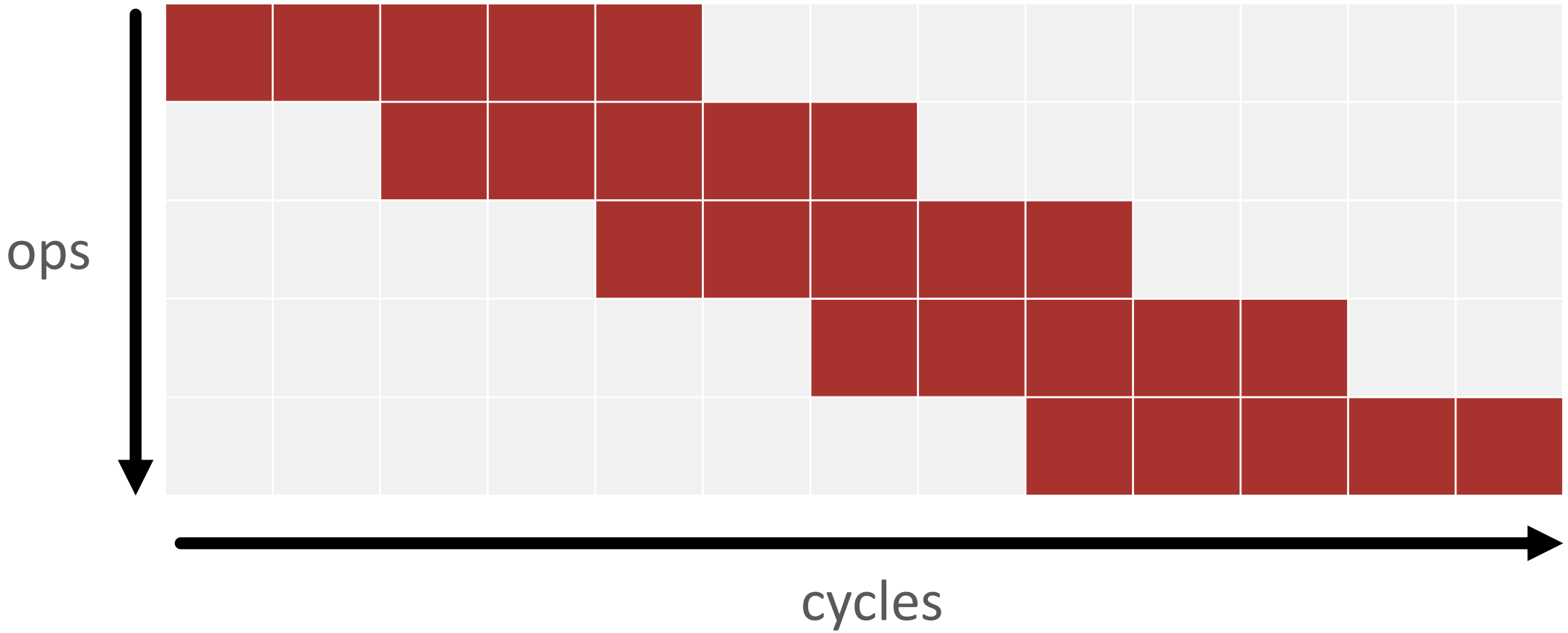
Our point of view

In HLS, we treat **pipelines**.

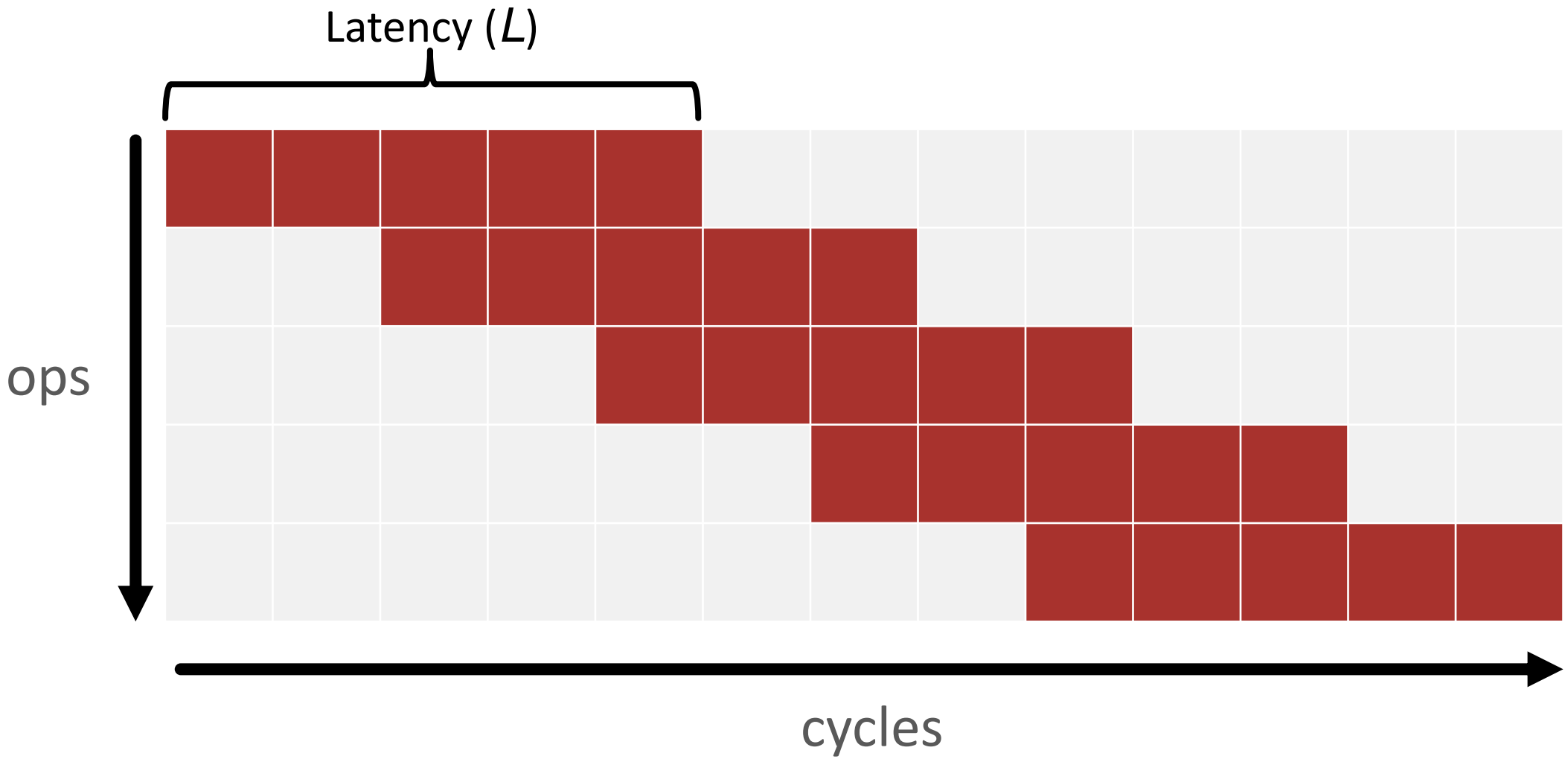
```
float c = a + b;
```



Pipelines



Pipelines



Multiple floating point operations

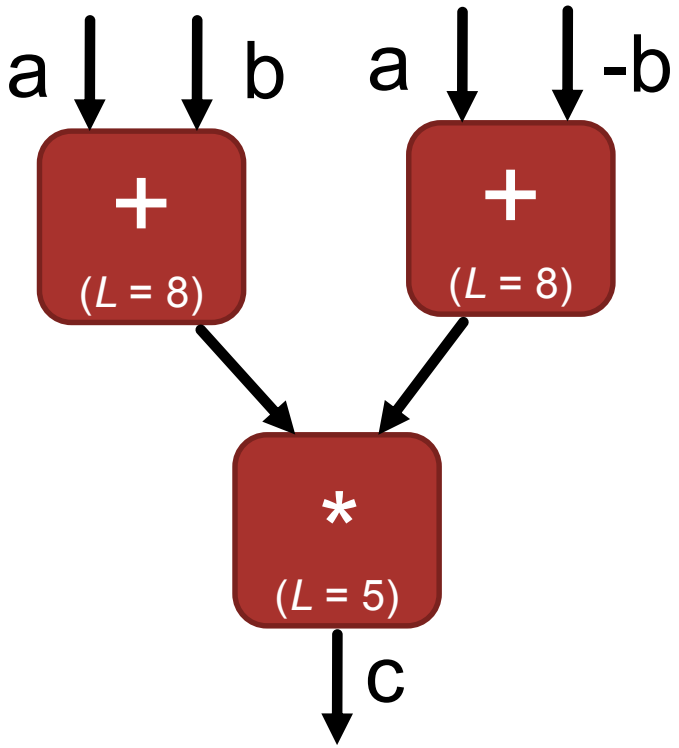
```
float c = (a + b) * (a - b);
```

Multiple floating point operations

```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

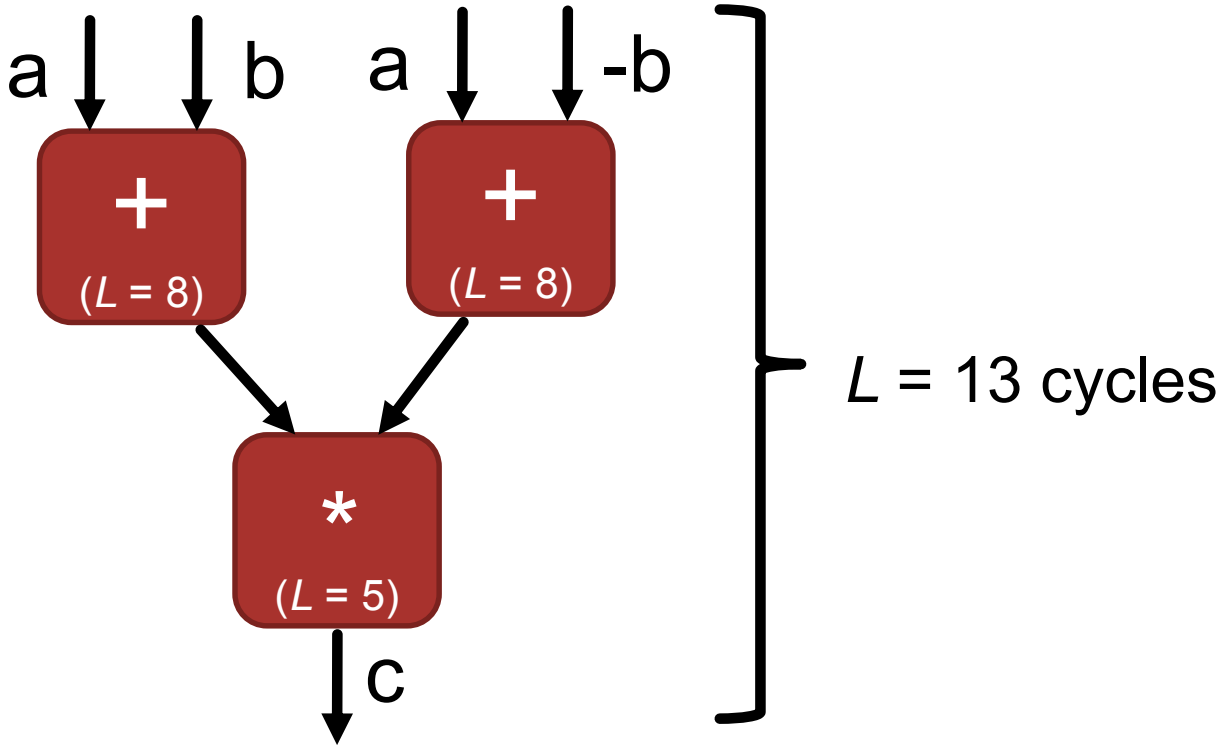
Multiple floating point operations




```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

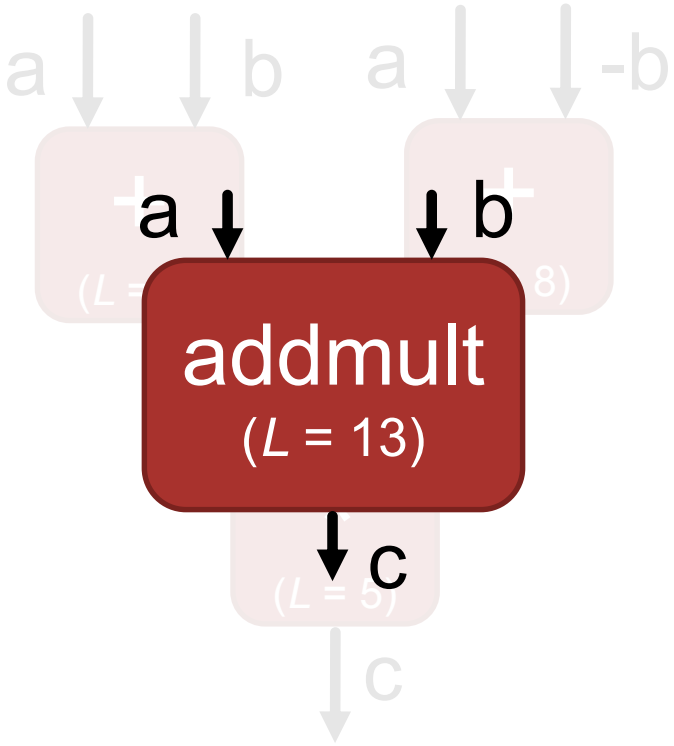
Multiple floating point operations




```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

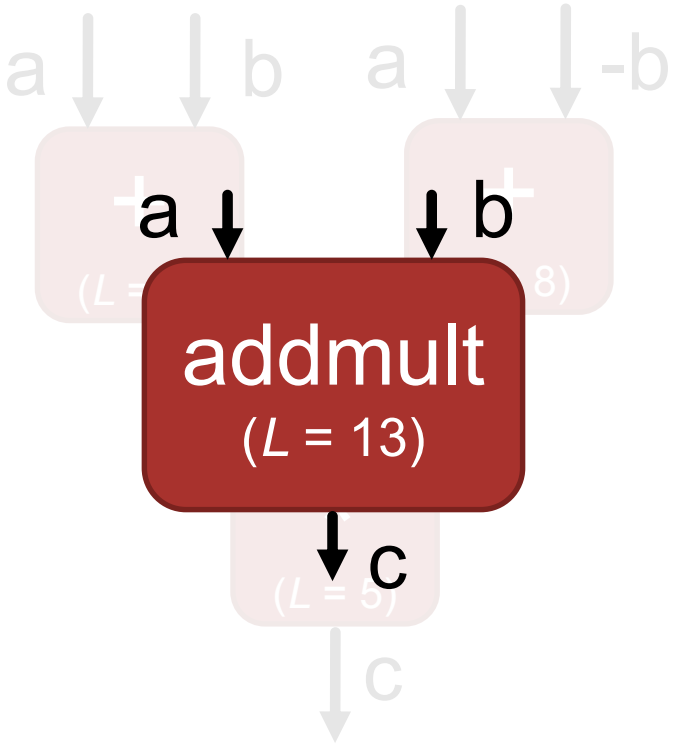
Multiple floating point operations

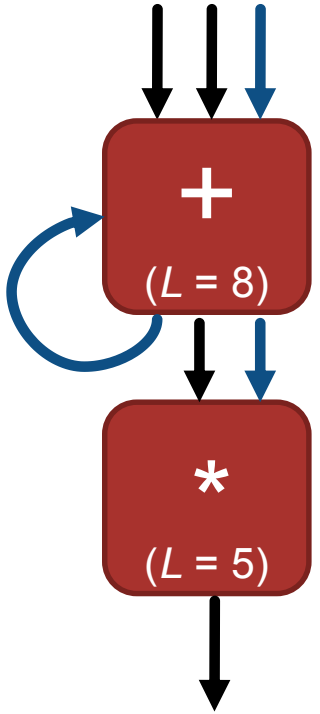



```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

Multiple floating point operations

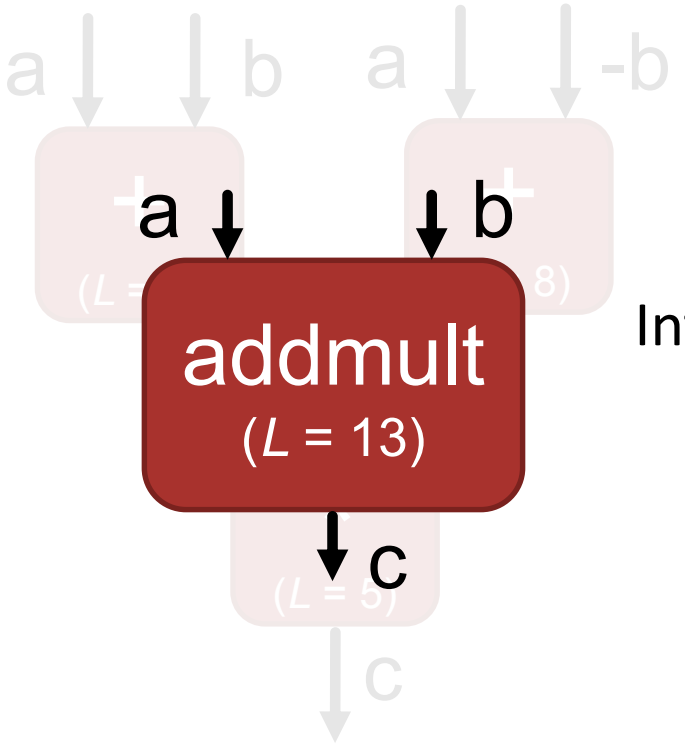




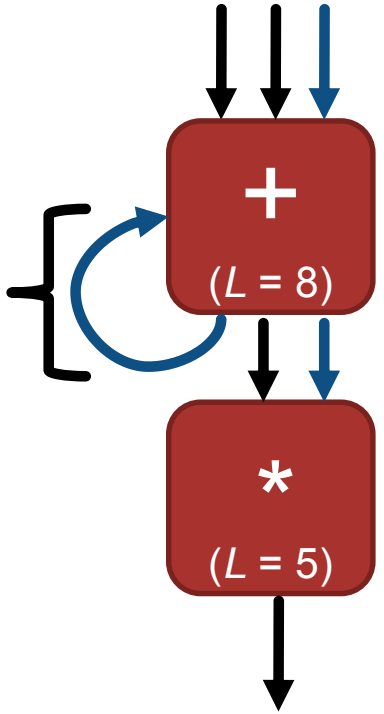

```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

Multiple floating point operations



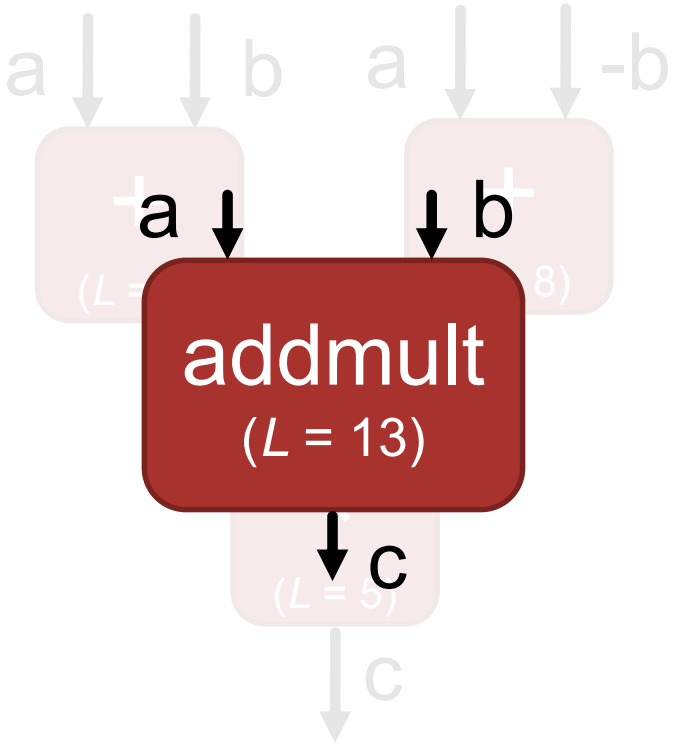
Interleave: +1 cycle

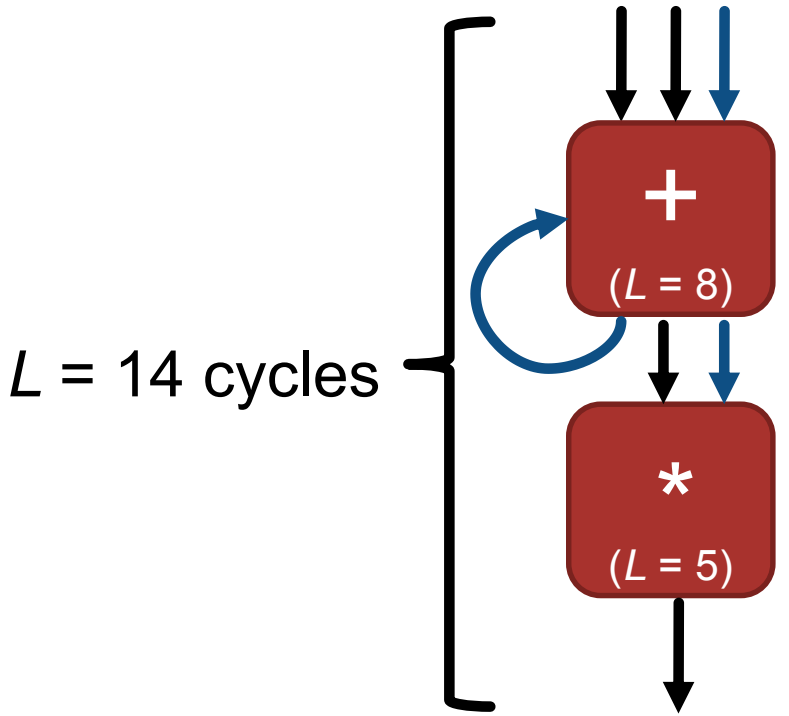



```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

Multiple floating point operations

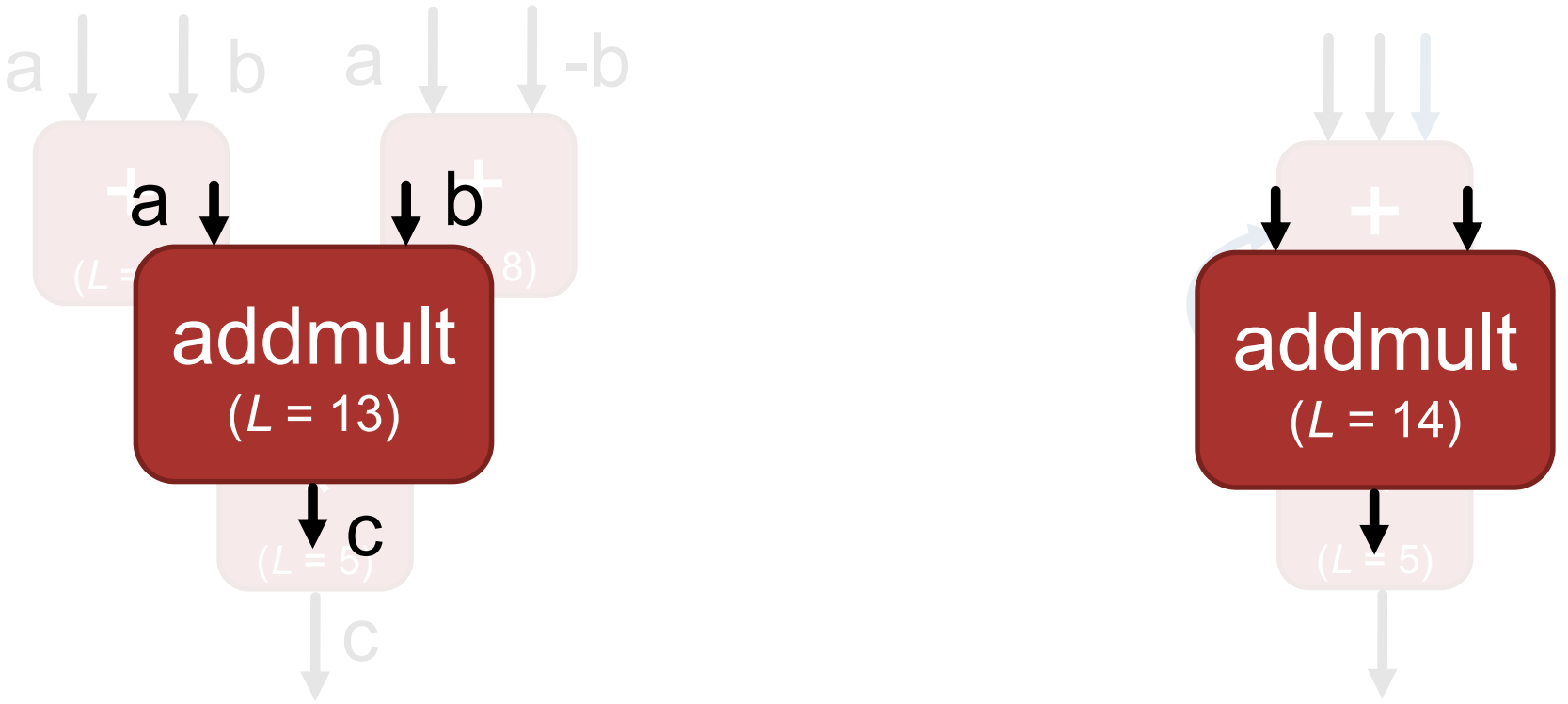





```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

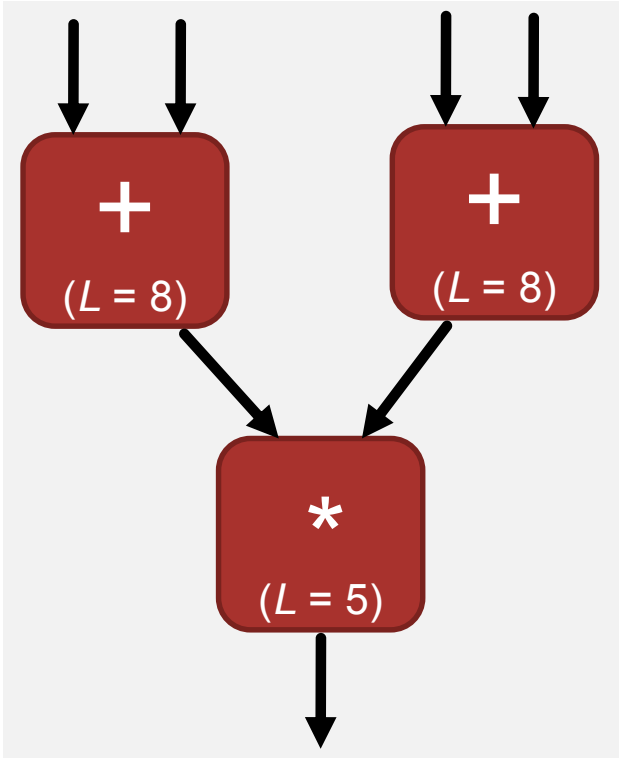
Multiple floating point operations

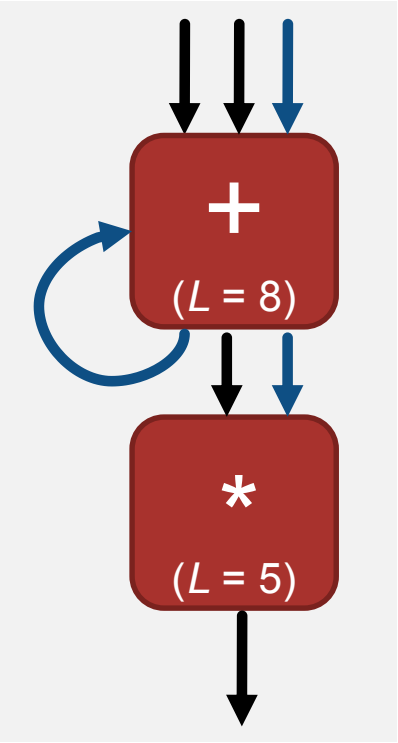



```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

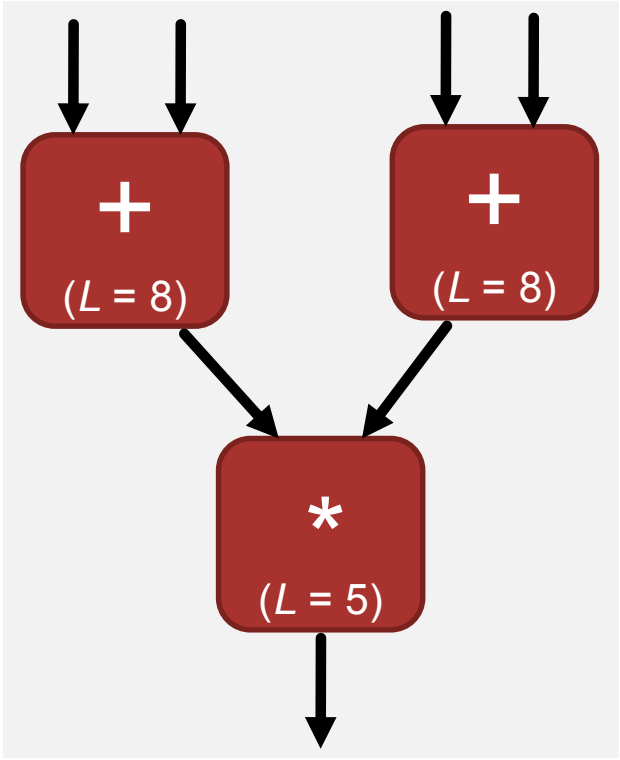
Initiation interval

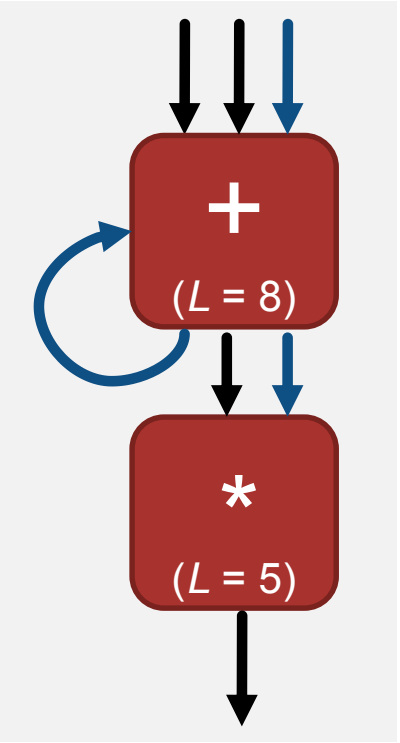




Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

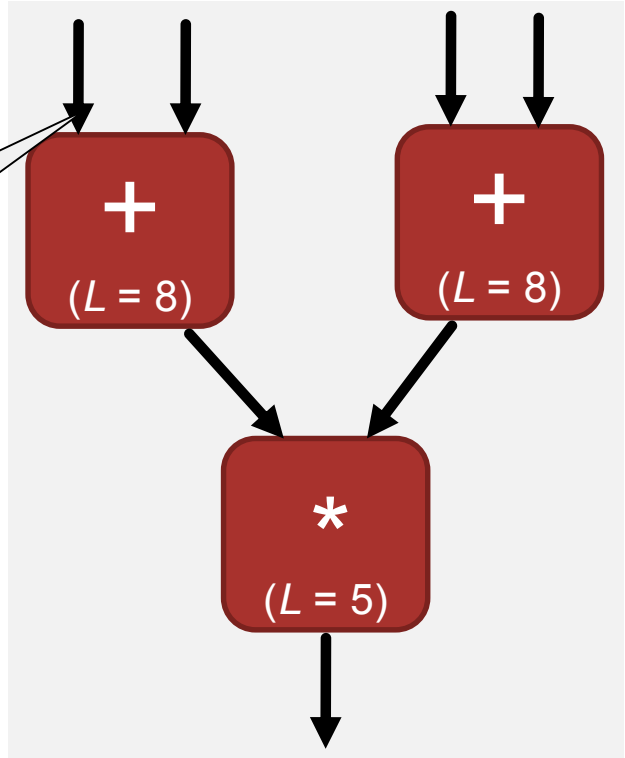


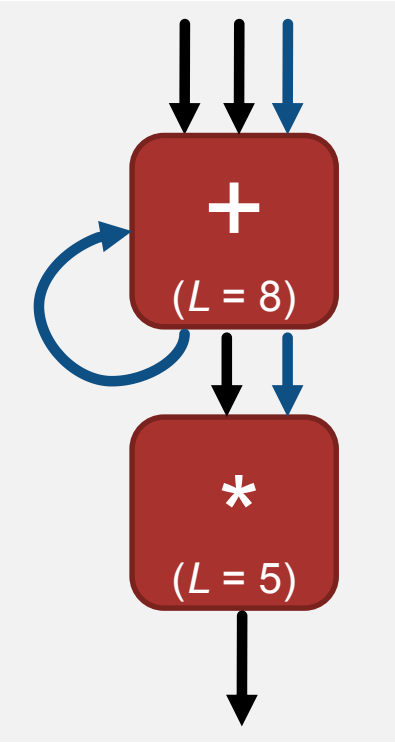


Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

Can accept all four inputs in parallel

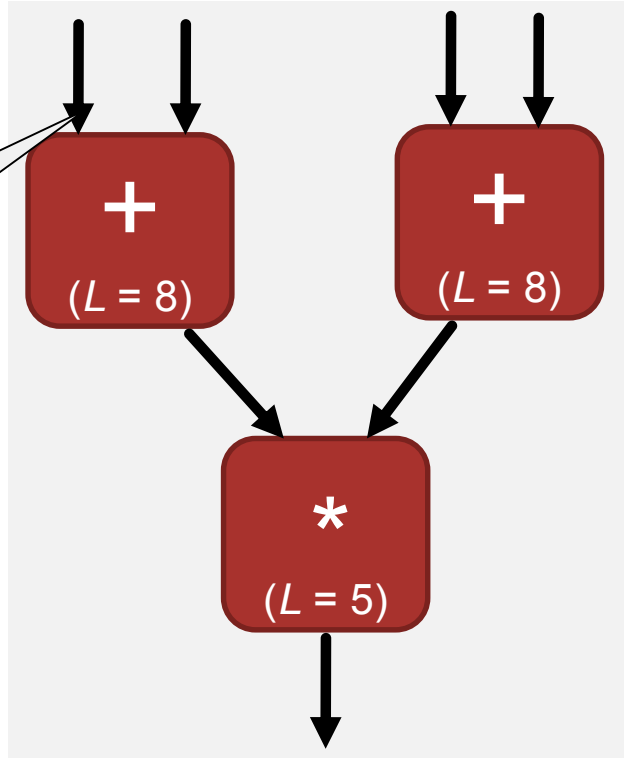


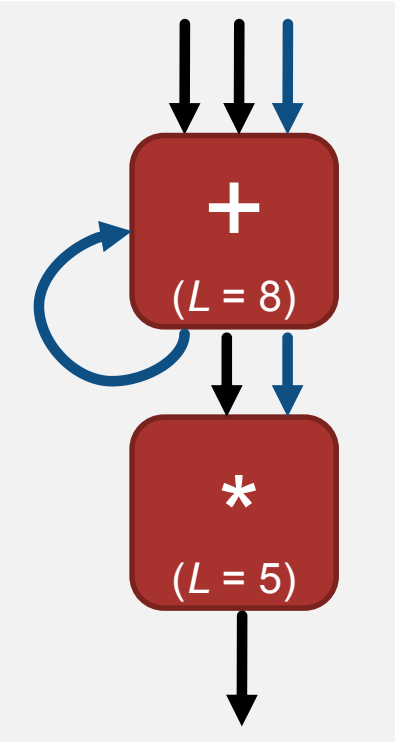


Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

Can accept all four inputs in parallel



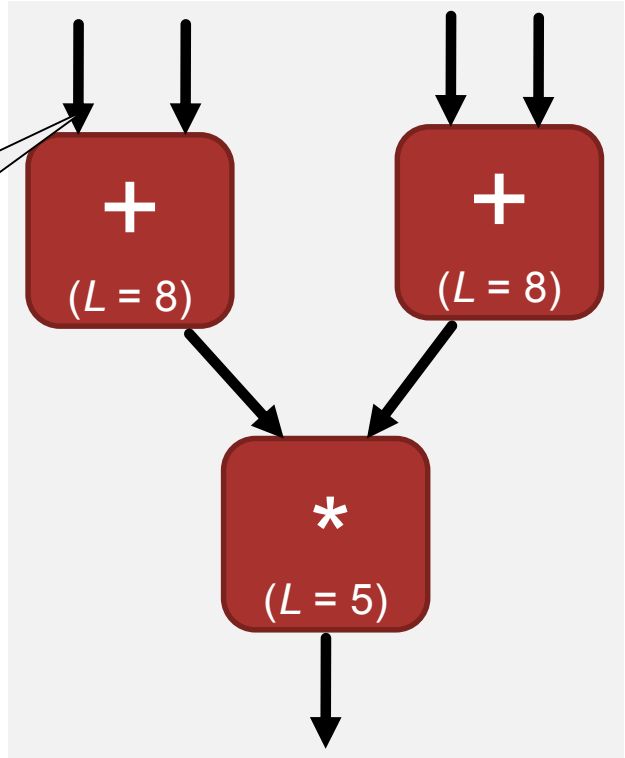


L = 13 cycles
I = 1 cycle
2 adds, 1 mult

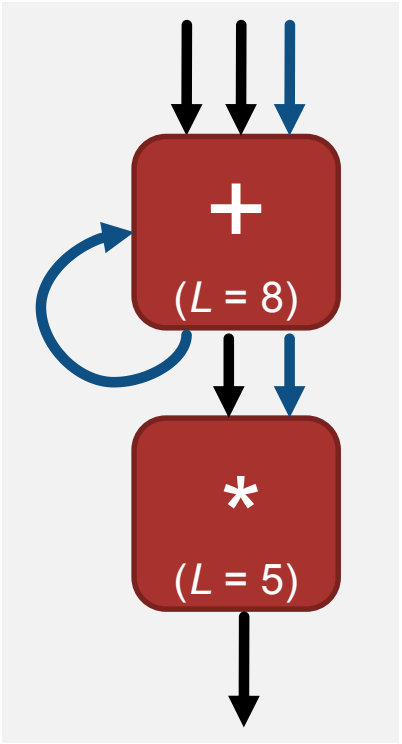
Initiation interval

In addition to **latency** (L), we introduce the property **initiation interval** ("II", here I).

Can accept all four inputs in parallel



L = 13 cycles
I = 1 cycle
2 adds, 1 mult

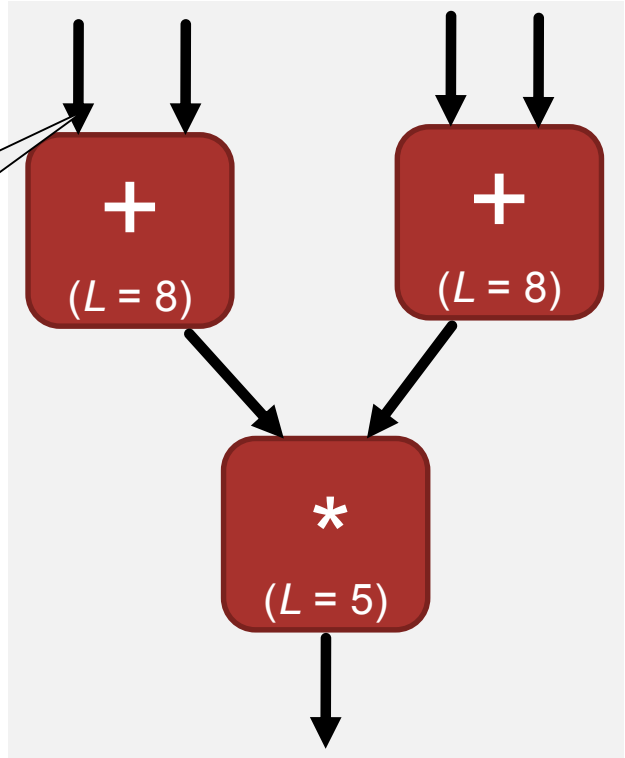


L = 14 cycles
I = 2 cycles
1 add, 1 mult

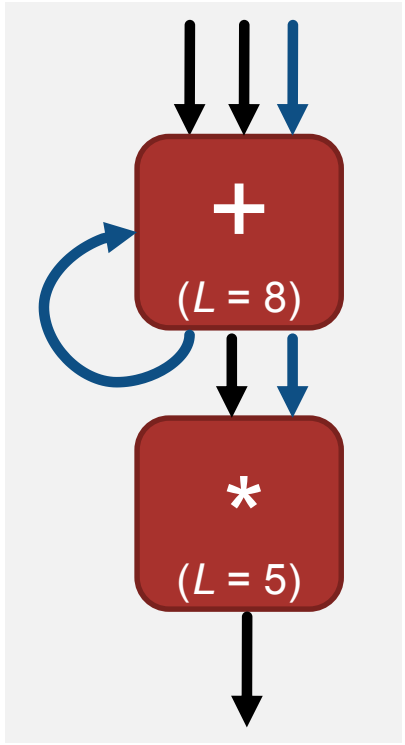
Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

Can accept all four inputs in parallel



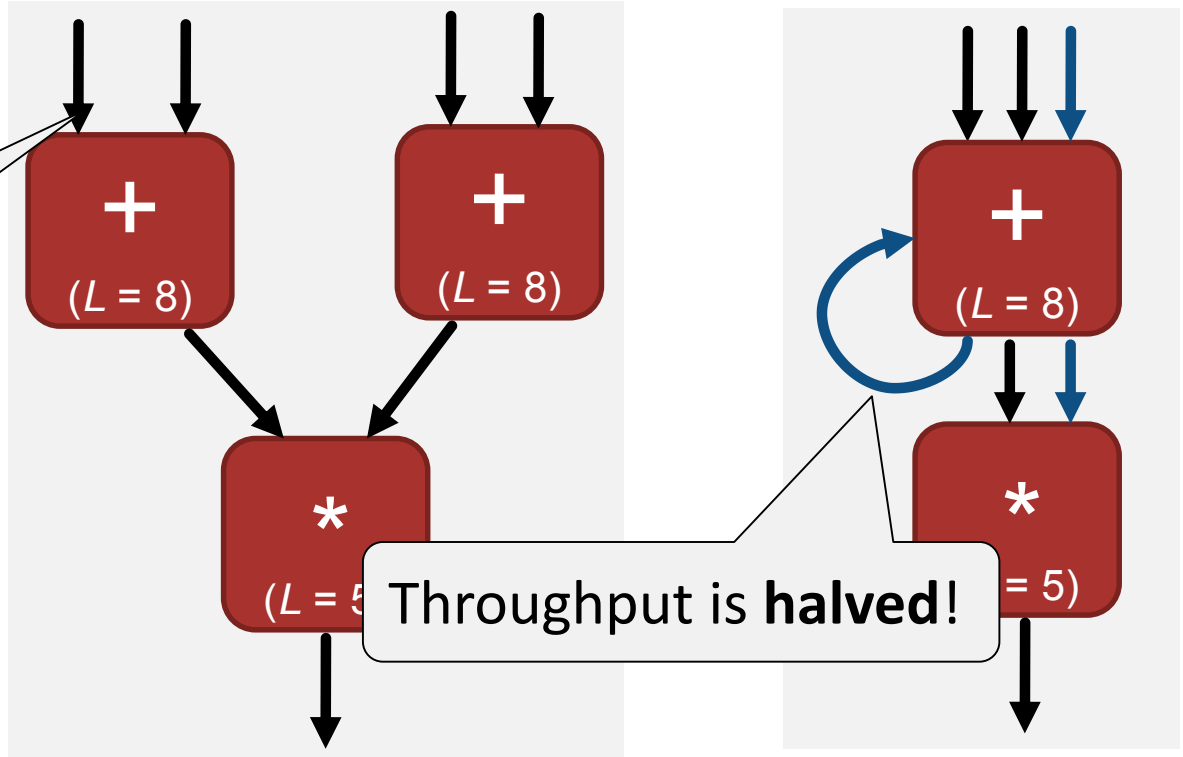
L = 13 cycles
I = 1 cycle
2 adds, 1 mult
3 op/1 cycle



L = 14 cycles
I = 2 cycles
1 add, 1 mult
3 op/2 cycles

Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).



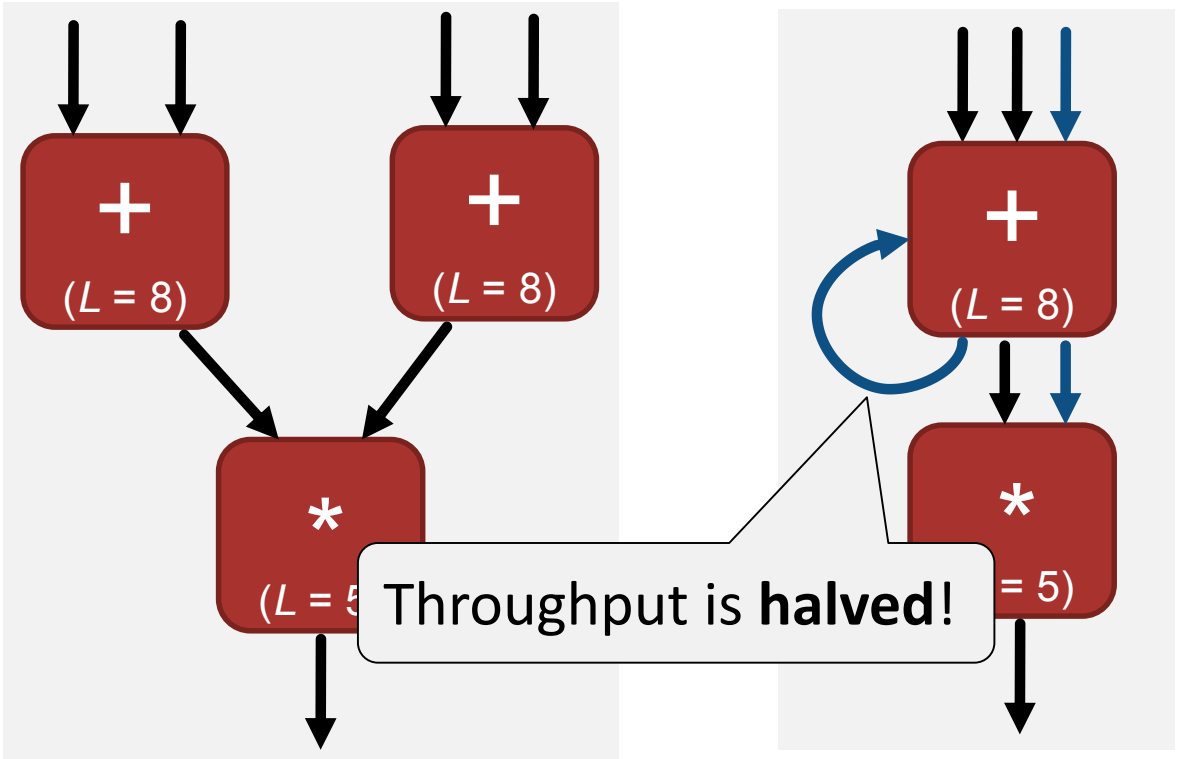
L = 13 cycles
I = 1 cycle
2 adds, 1 mult
3 op/1 cycle

L = 14 cycles
I = 2 cycles
1 add, 1 mult
3 op/2 cycles

Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

Interpretations:



L = 13 cycles
I = 1 cycle
2 adds, 1 mult
3 op/1 cycle

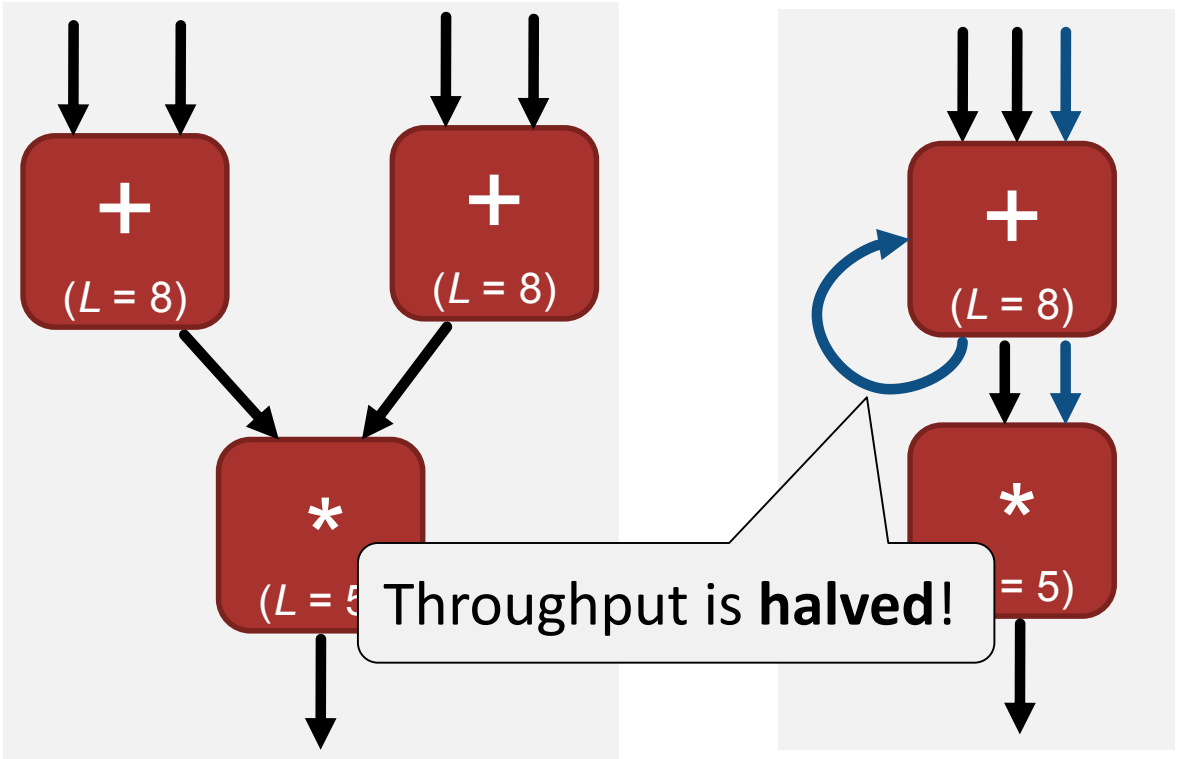
L = 14 cycles
I = 2 cycles
1 add, 1 mult
3 op/2 cycles

Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

Interpretations:

1. *No. of cycles before we can accept new inputs (sometimes called "gap")*



L = 13 cycles
I = 1 cycle
2 adds, 1 mult
3 op/1 cycle

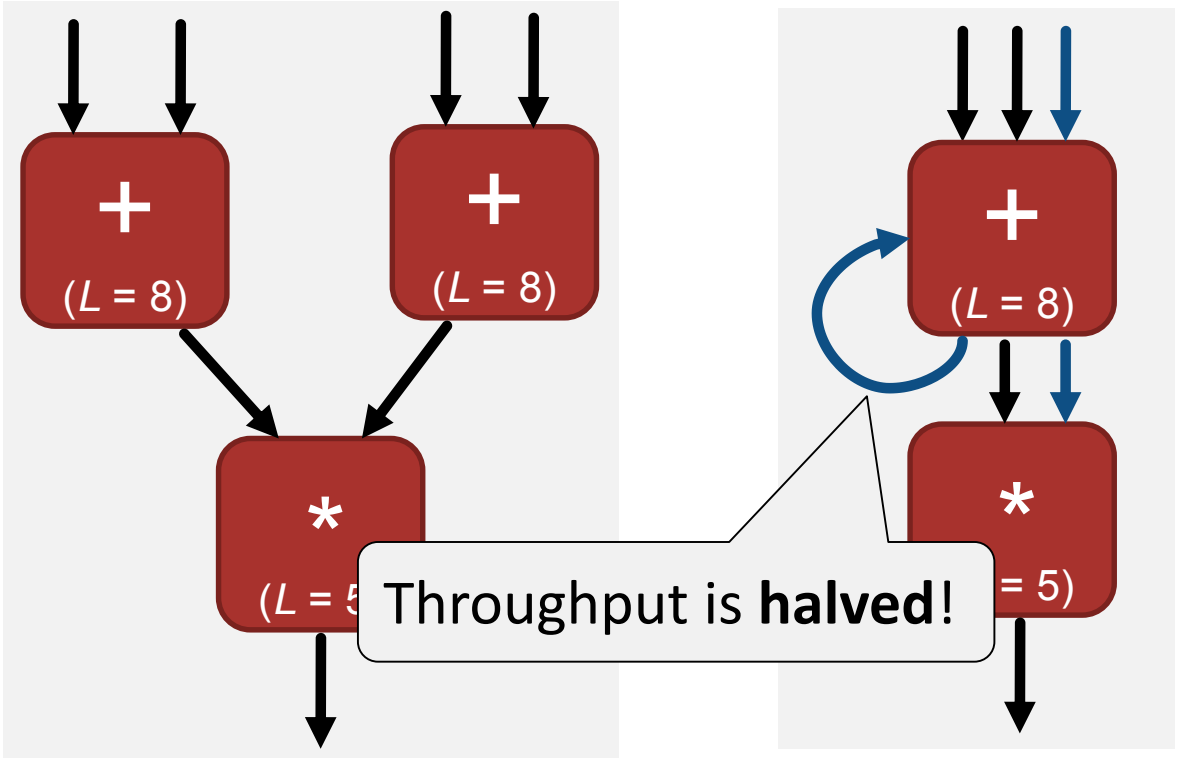
L = 14 cycles
I = 2 cycles
1 add, 1 mult
3 op/2 cycles

Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

Interpretations:

1. *No. of cycles before we can accept new inputs (sometimes called "gap")*
2. *Inverse throughput of the pipeline*



L = 13 cycles
I = 1 cycle
2 adds, 1 mult
3 op/1 cycle

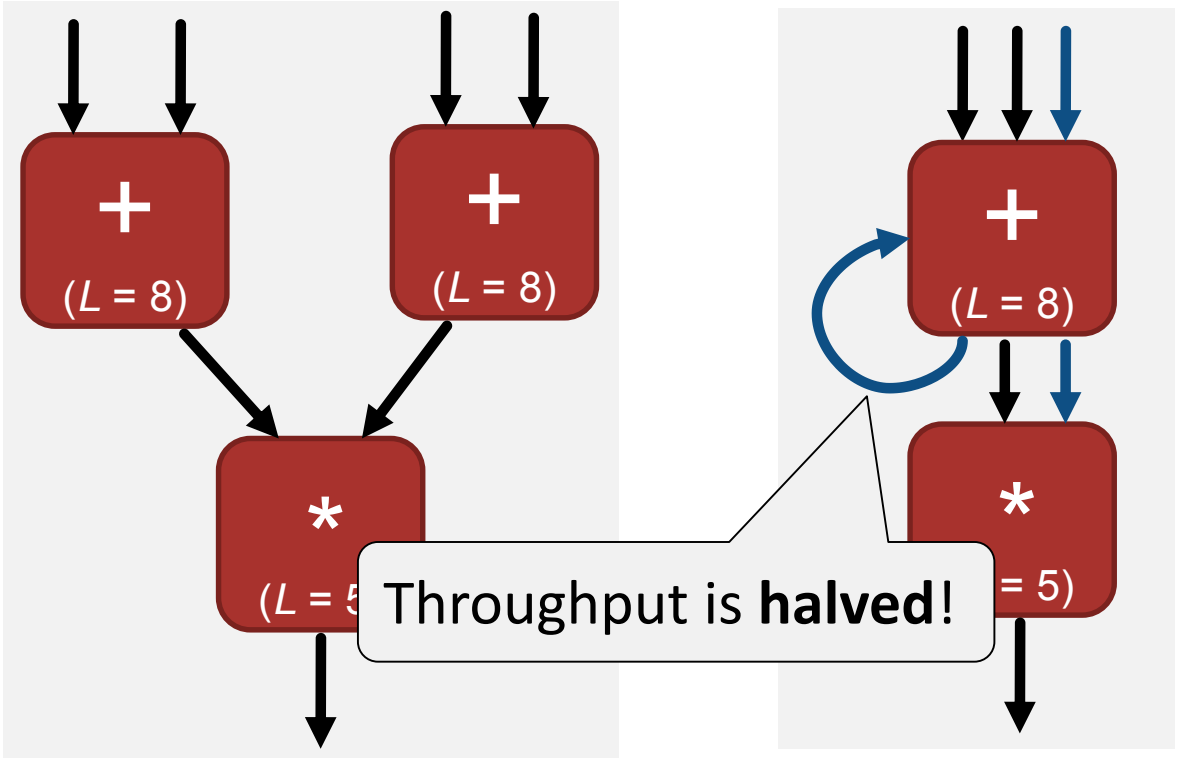
L = 14 cycles
I = 2 cycles
1 add, 1 mult
3 op/2 cycles

Initiation interval

In addition to **latency** (*L*), we introduce the property **initiation interval** ("II", here *I*).

Interpretations:

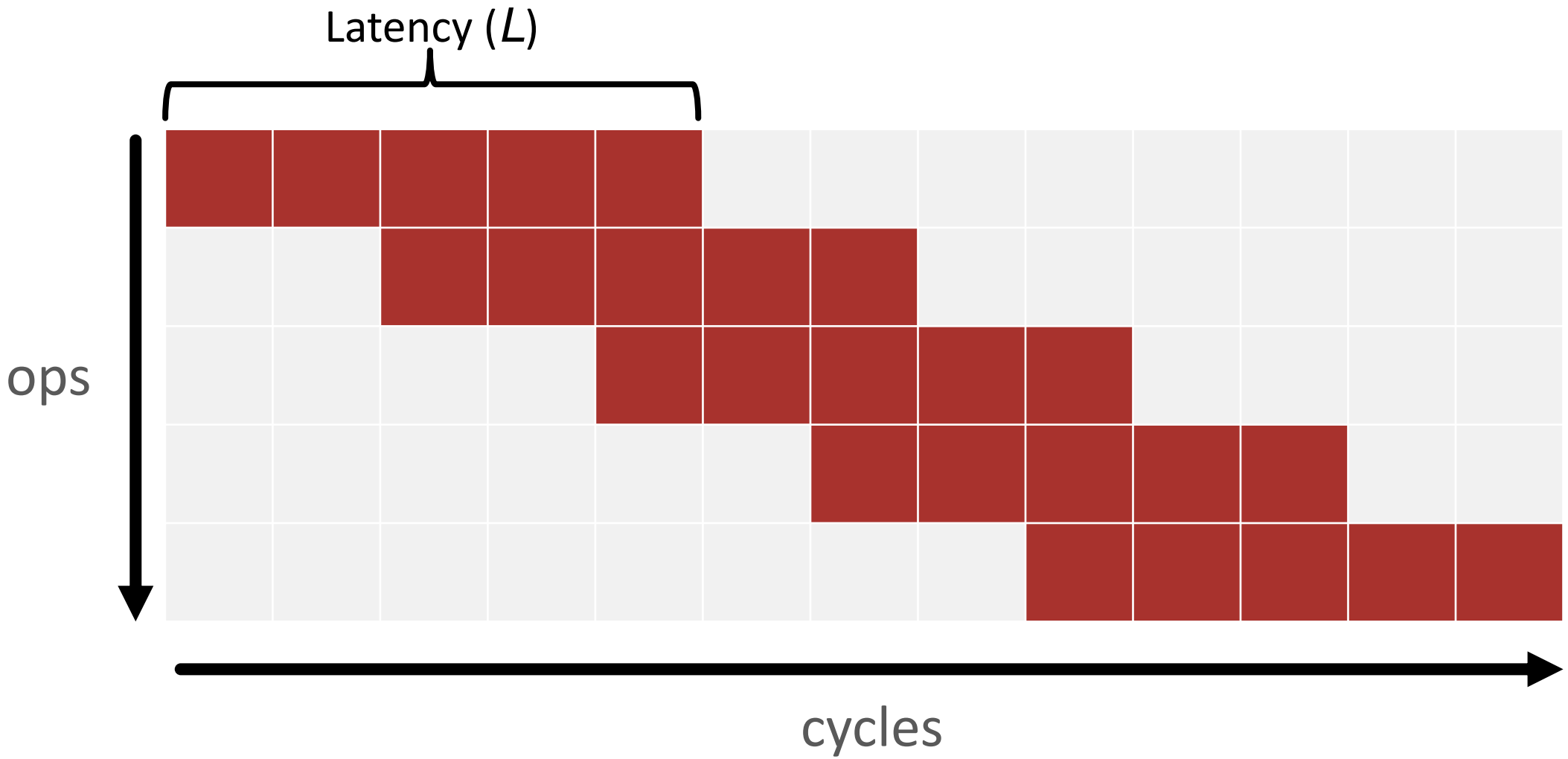
1. *No. of cycles before we can accept new inputs (sometimes called "gap")*
2. *Inverse throughput of the pipeline*
3. *Factor slowdown of your application ☺*



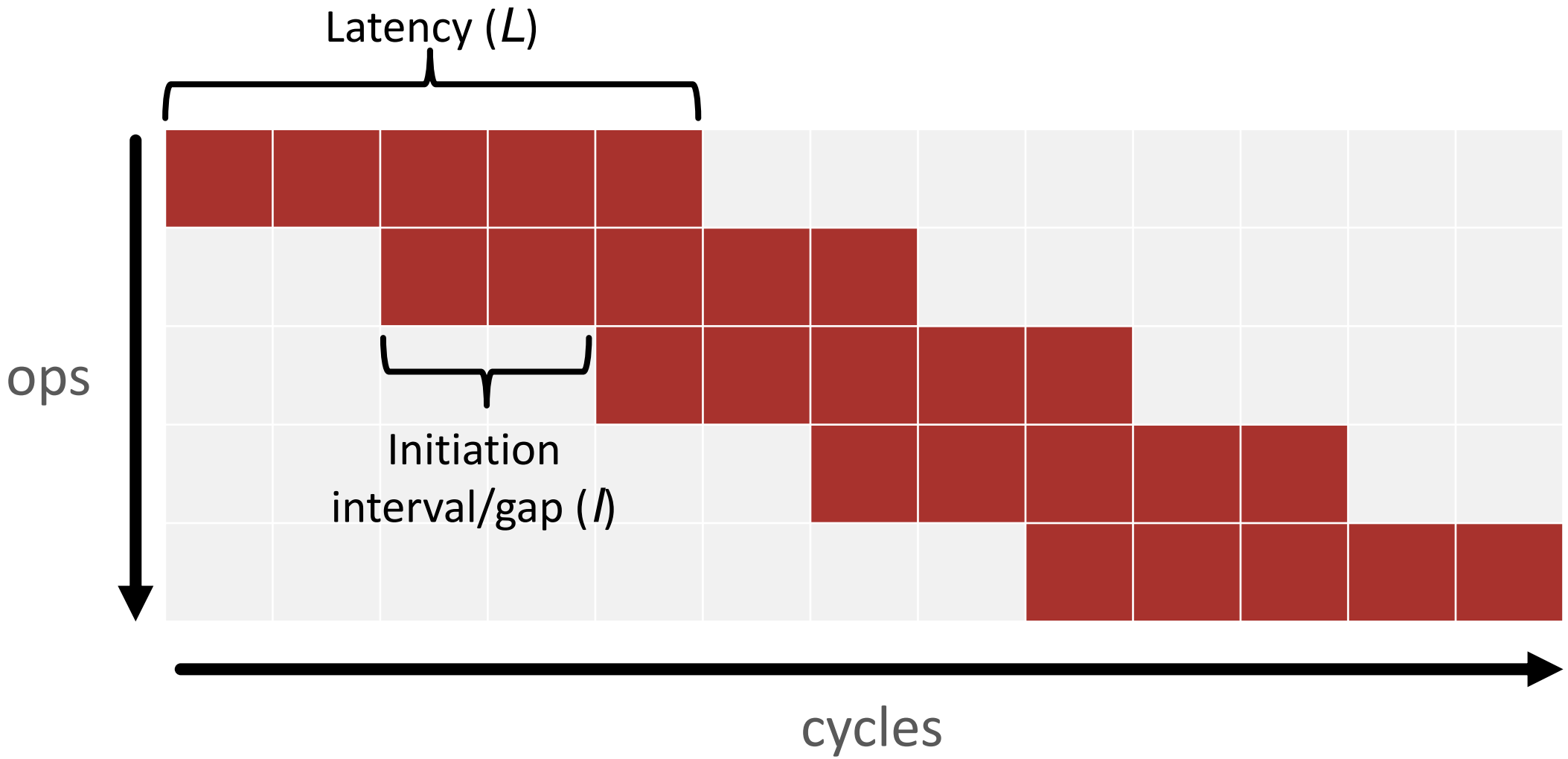
L = 13 cycles
I = 1 cycle
2 adds, 1 mult
3 op/1 cycle

L = 14 cycles
I = 2 cycles
1 add, 1 mult
3 op/2 cycles

Pipelines vol. II

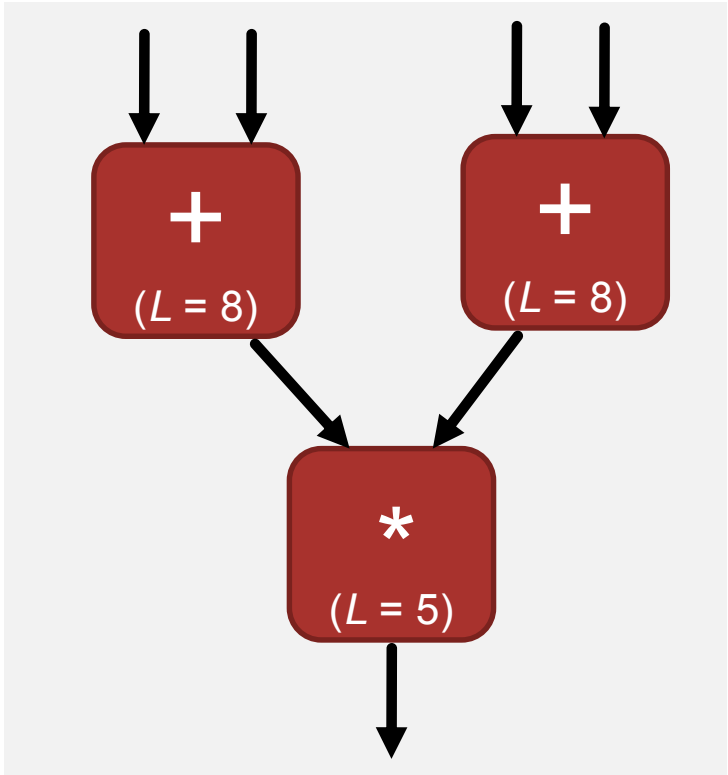


Pipelines vol. II



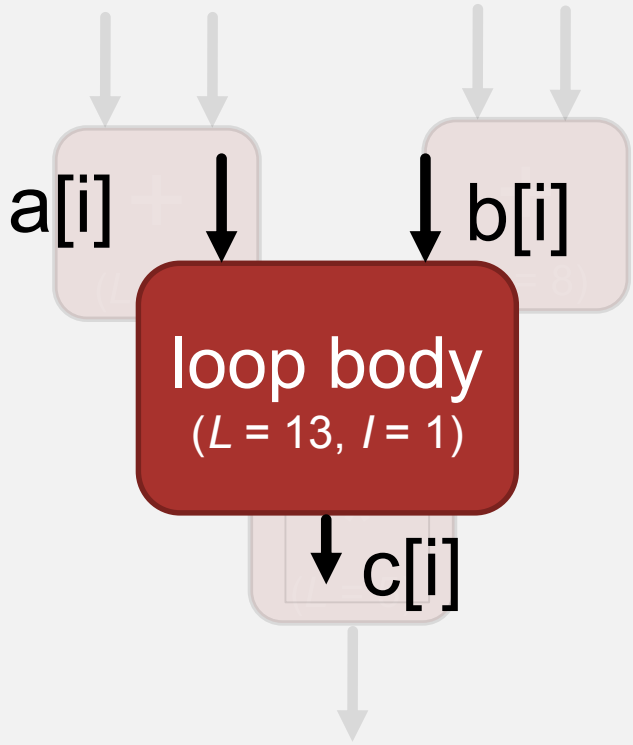
Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=1
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```



Adding loops

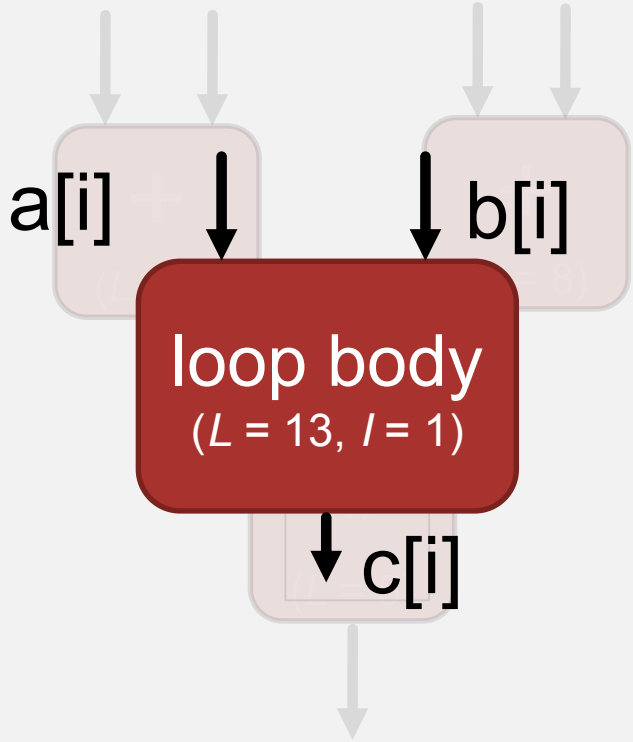
```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=1
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```



Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=1
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```

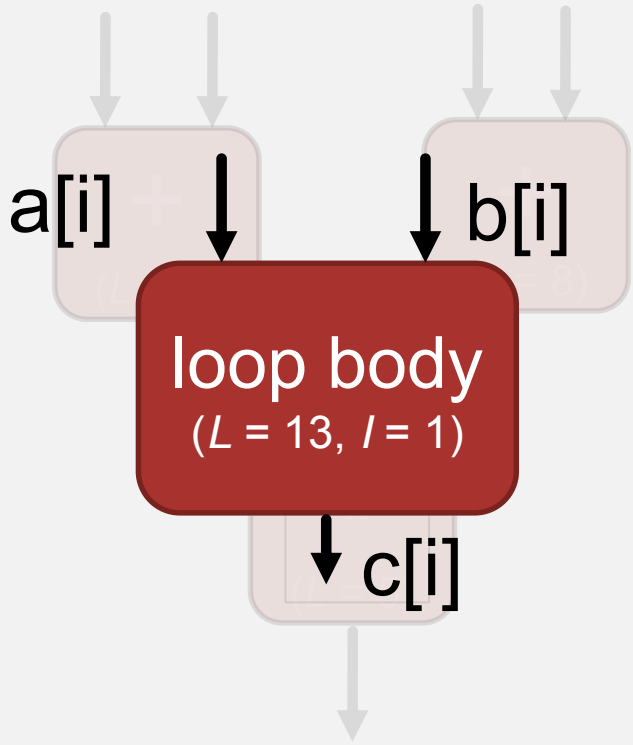
1 iteration	13 + 1 = 14 cycles
10 iterations	13 + 10 = 23 cycles
N iterations	13 + *N* cycles



Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=1
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```

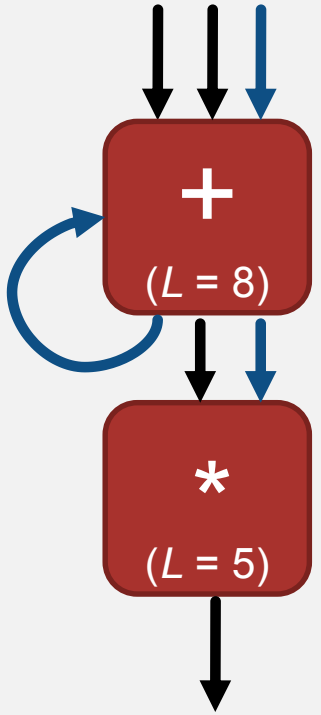
1 iteration	13 + 1 = 14 cycles
10 iterations	13 + 10 = 23 cycles
N iterations	13 + *N* cycles



Loop iterations affect the runtime **additively**, regardless of body content

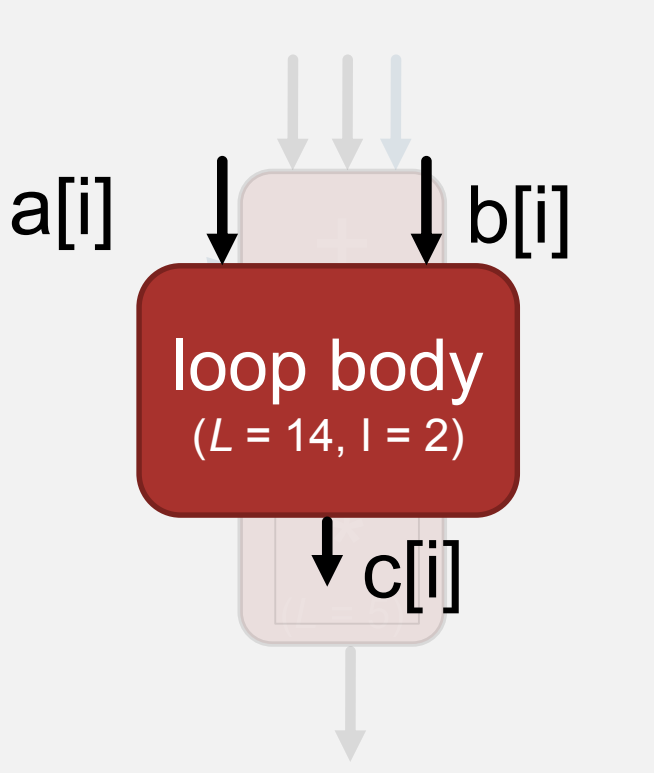
Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=2
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```



Adding loops

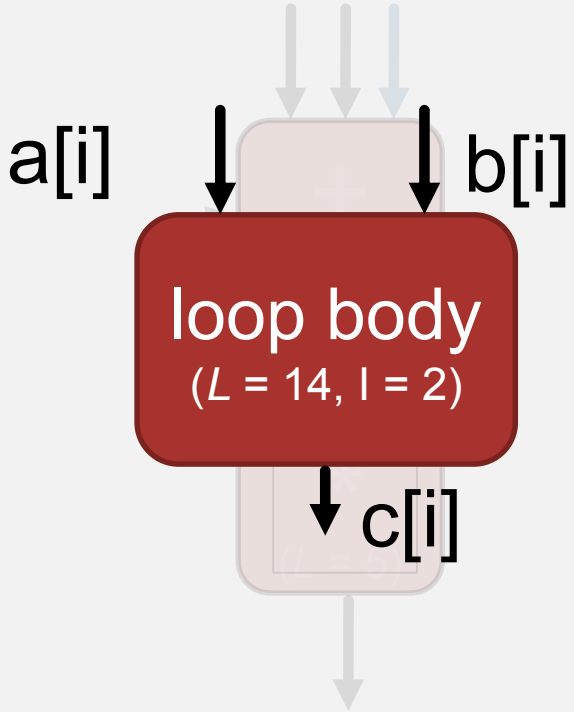
```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=2
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```



Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=2
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```

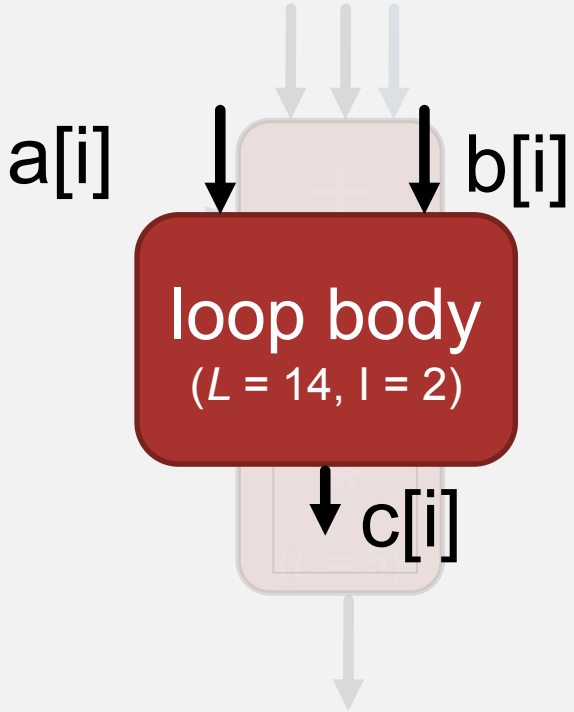
1 iteration	14 + 2 = 16 cycles
10 iterations	14 + 20 = 34 cycles
N iterations	14 + 2*N* cycles



Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=2
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```

1 iteration	14 + 2 = 16 cycles
10 iterations	14 + 20 = 34 cycles
N iterations	14 + 2*N* cycles



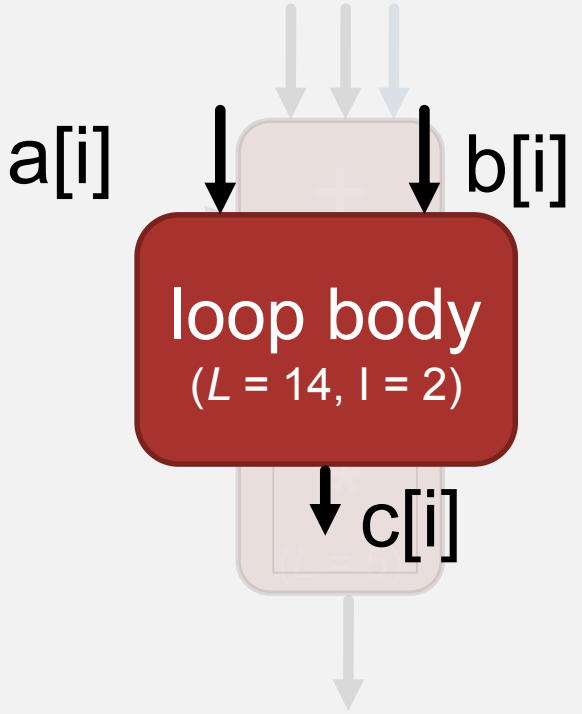
Initiation interval paid at *every iteration*

Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=2
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```

Generally:

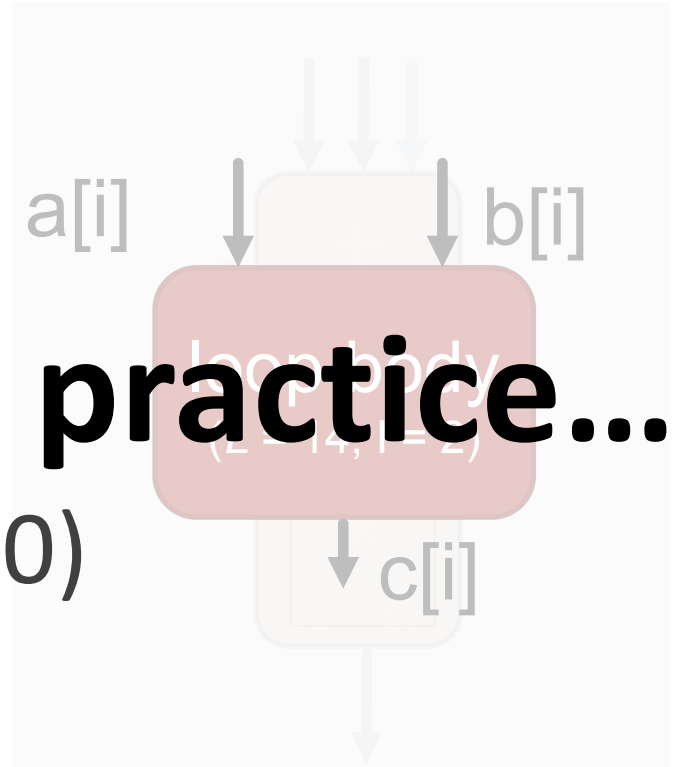
$$L_{\text{tot}} = L + I \cdot N$$



Initiation interval paid at *every iteration*

Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=2
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```



Generally:

$$L_{\text{tot}} = L + I \cdot N$$

Initiation interval paid at *every iteration*

Let's see this in practice…

(example 0)

Pipeline stalls in practice

Why do we worry so much? Just set $l = 1$…

Pipeline stalls in practice

Why do we worry so much? Just set $I = 1$…

Increased II is commonly found in the wild:

Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE II=1
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

1. Intra-iteration (now):
   - *Multiple accesses to the same interface*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE II=1
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

1. Intra-iteration (now):
   - *Multiple accesses to the same interface*

2. Inter-iteration (later)
   - *Data dependencies*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE II=1
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

1. Intra-iteration (now):
   - *Multiple accesses to the same interface*
2. Inter-iteration (later)
   - *Data dependencies*
3. Low throughput requirements (rare)
   - *e.g. input only received every 16 cycles*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE II=1
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

1. Intra-iteration (now):
   - *Multiple accesses to the same interface*
2. Inter-iteration (later)
   - *Data dependencies*
3. Low throughput requirements (rare)
   - *e.g. input only received every 16 cycles*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE II=1
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

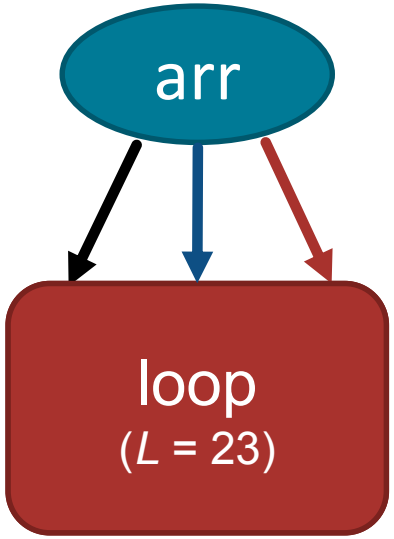
Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

1. Intra-iteration (now):
   - *Multiple accesses to the same interface*
2. Inter-iteration (later)
   - *Data dependencies*
3. Low throughput requirements (rare)
   - *e.g. input only received every 16 cycles*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE II=1
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



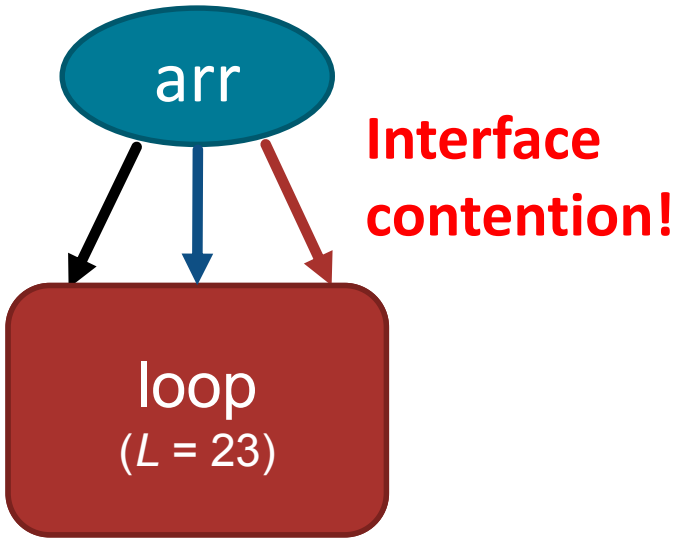
Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

1. Intra-iteration (now):
   - *Multiple accesses to the same interface*
2. Inter-iteration (later)
   - *Data dependencies*
3. Low throughput requirements (rare)
   - *e.g. input only received every 16 cycles*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE II=1
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



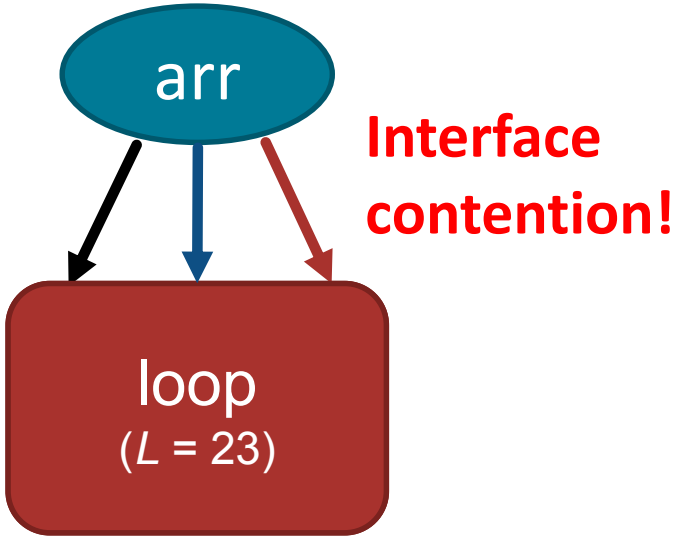
Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

Increased II is commonly found in the wild:

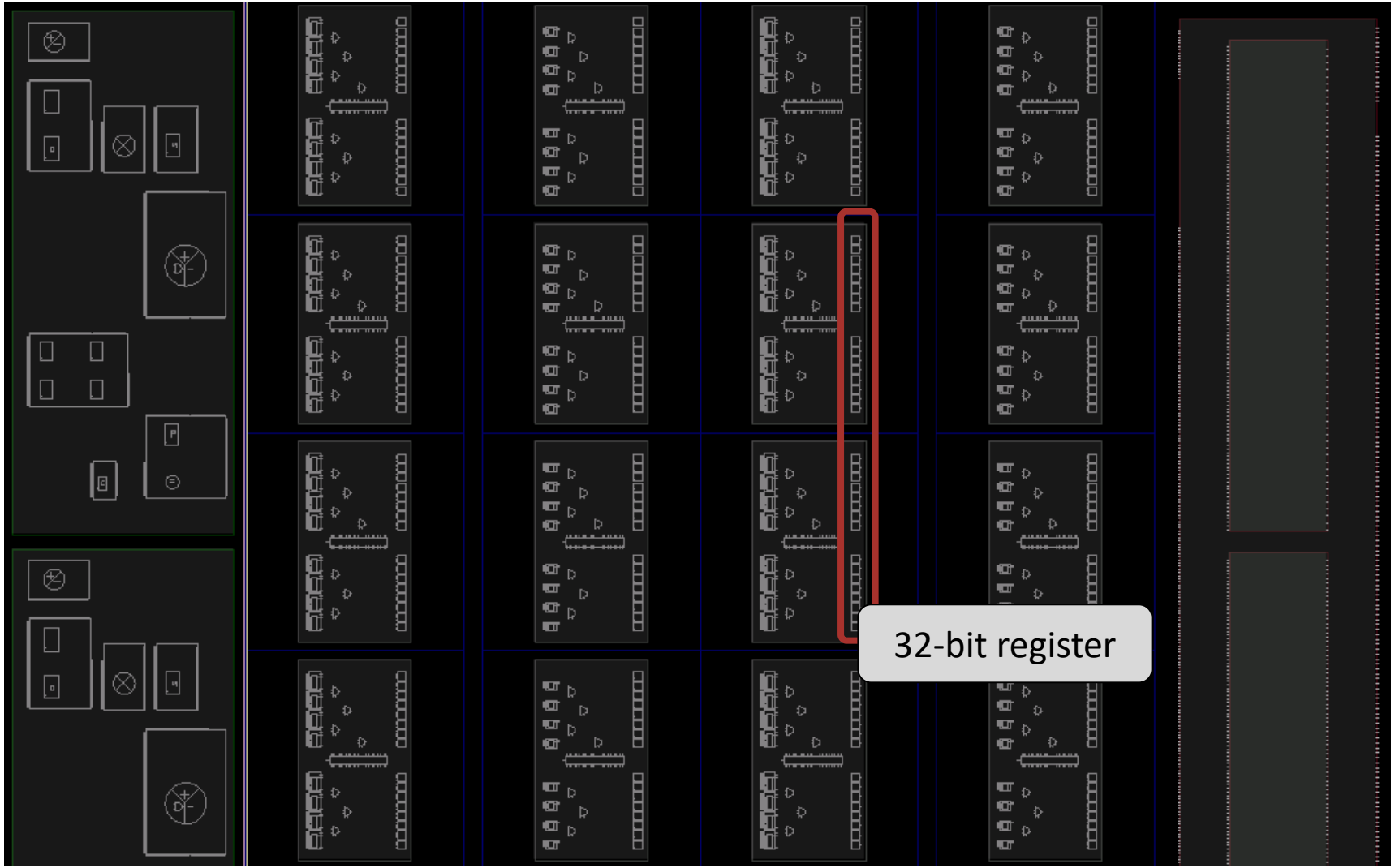
1. Intra-iteration (now):
   - *Multiple accesses to the same interface*
2. Inter-iteration (later)
   - *Data dependencies*
3. Low throughput requirements (rare)
   - *e.g. input only received every 16 cycles*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE ~~II=1~~ II=3
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

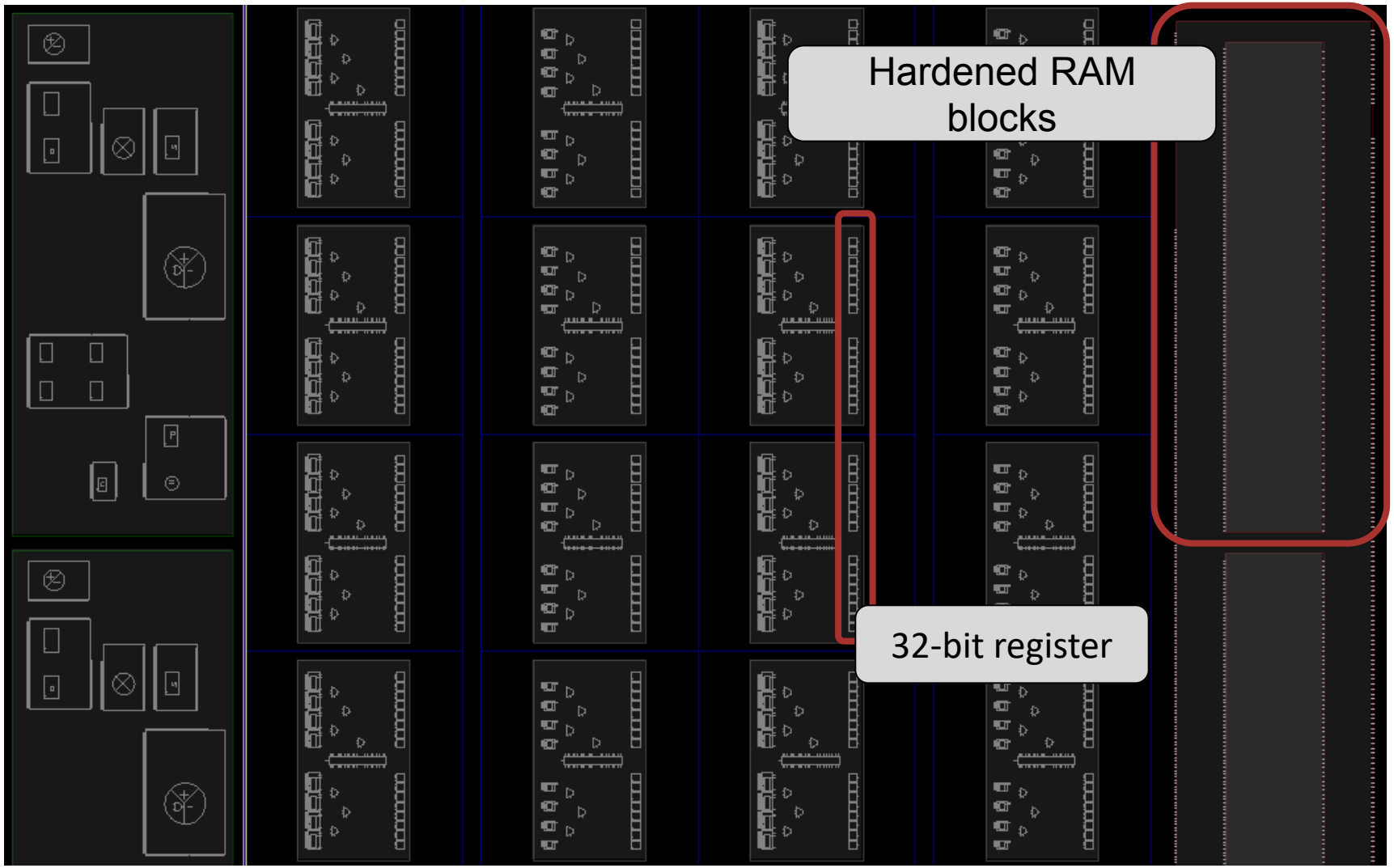


Fast memory everywhere

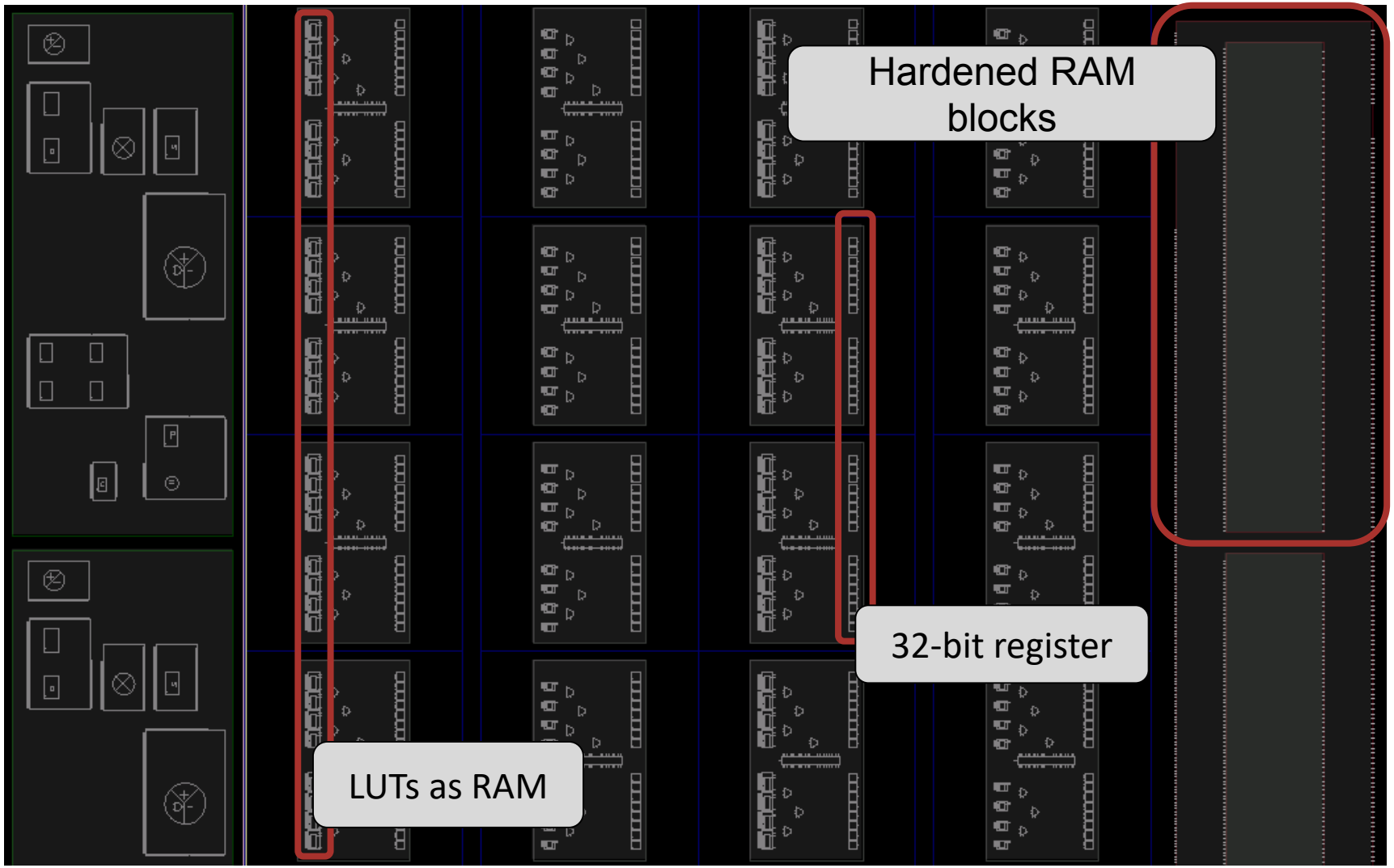
Fast memory everywhere



Fast memory everywhere

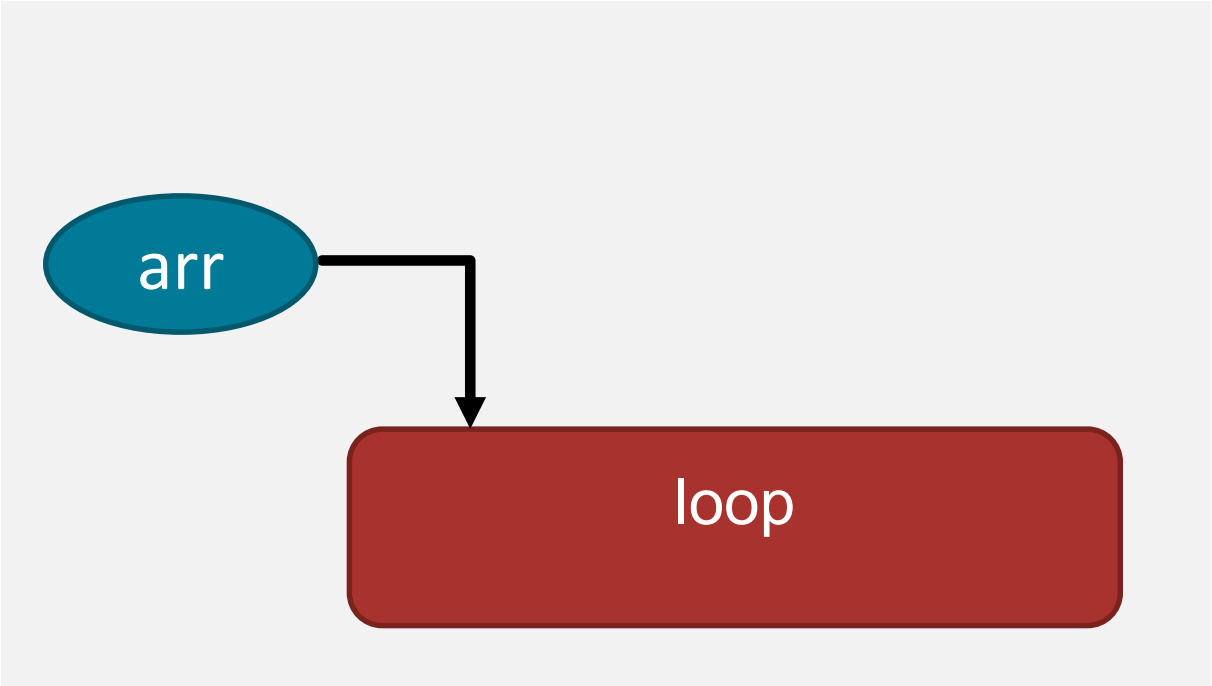


Fast memory everywhere



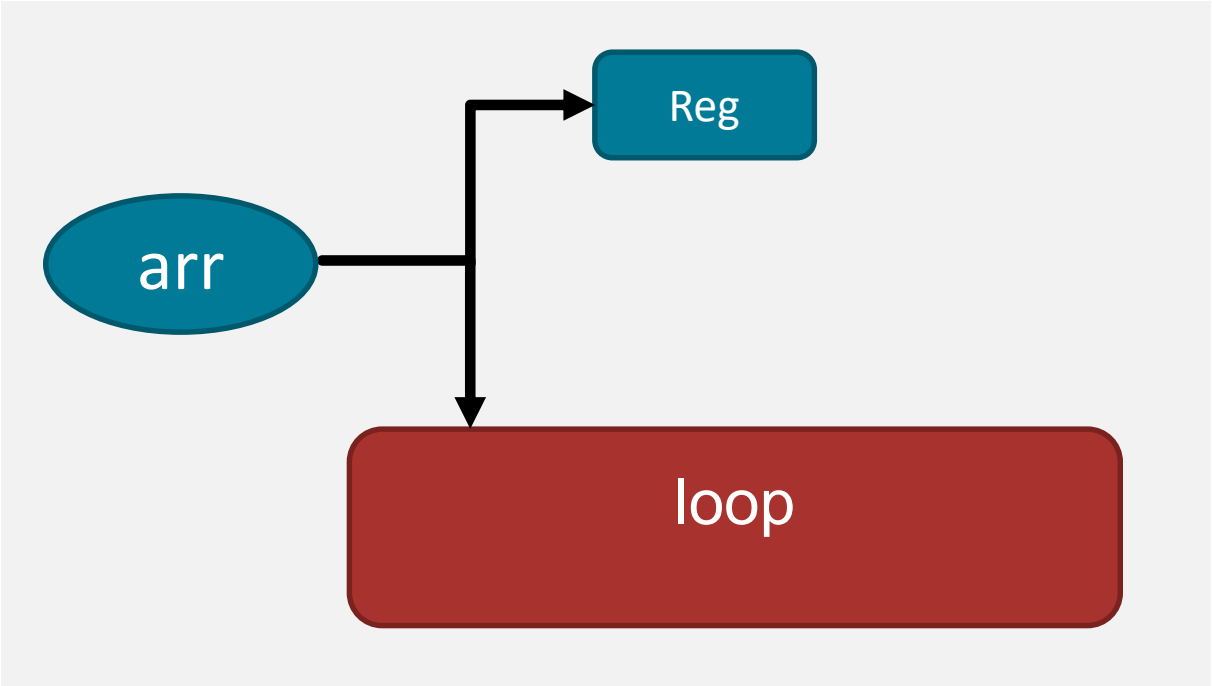
Inserting registers

```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



Inserting registers

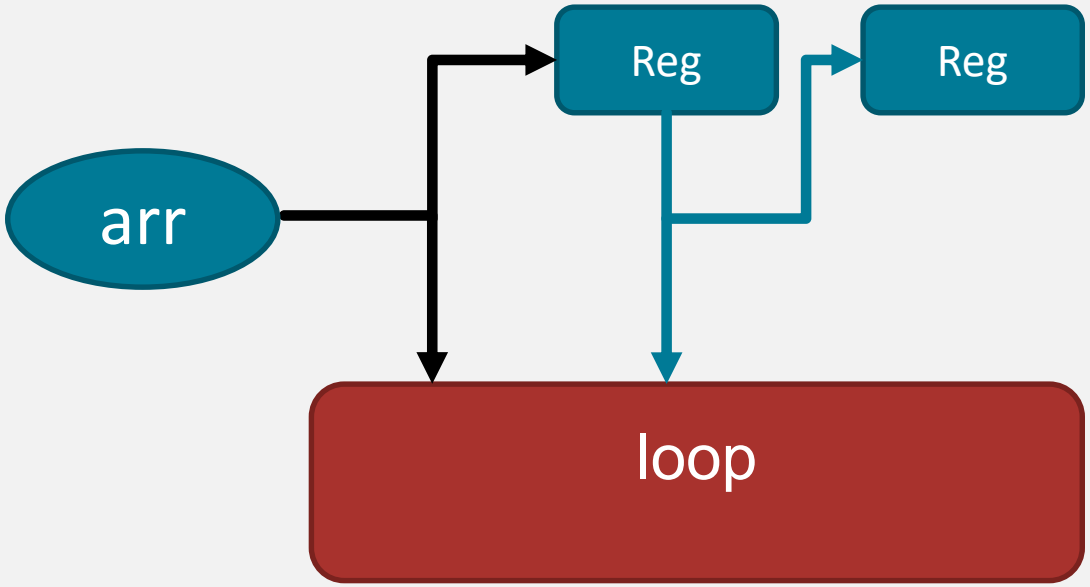
```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



Stage 1

Inserting registers

```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

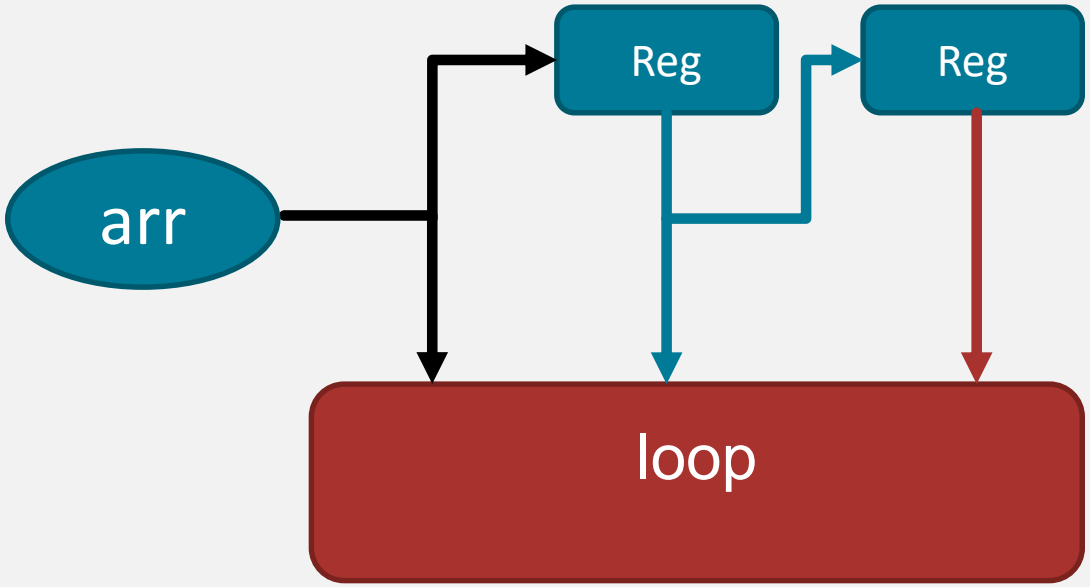


Stage 1

Stage 2

Inserting registers

```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



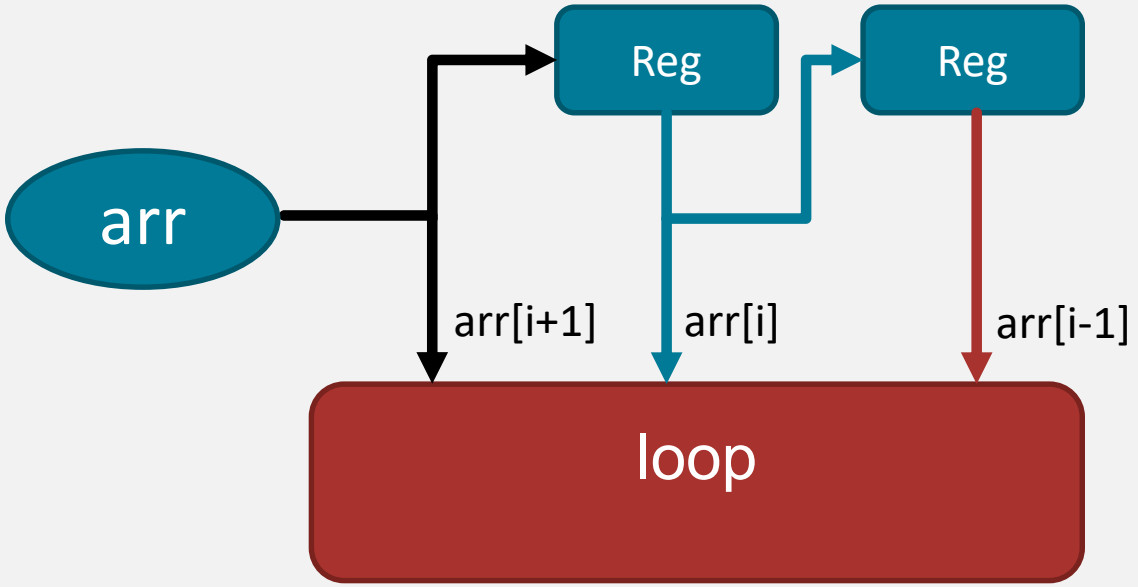
Stage 1

Stage 2

Stage 3

Inserting registers

```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



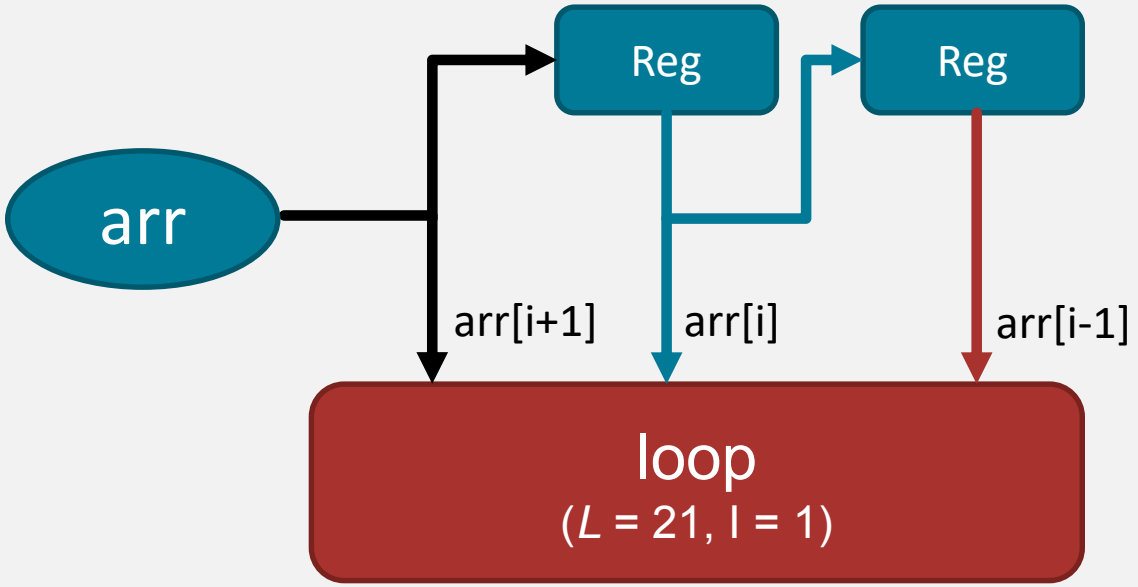
Stage 1

Stage 2

Stage 3

Inserting registers

```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



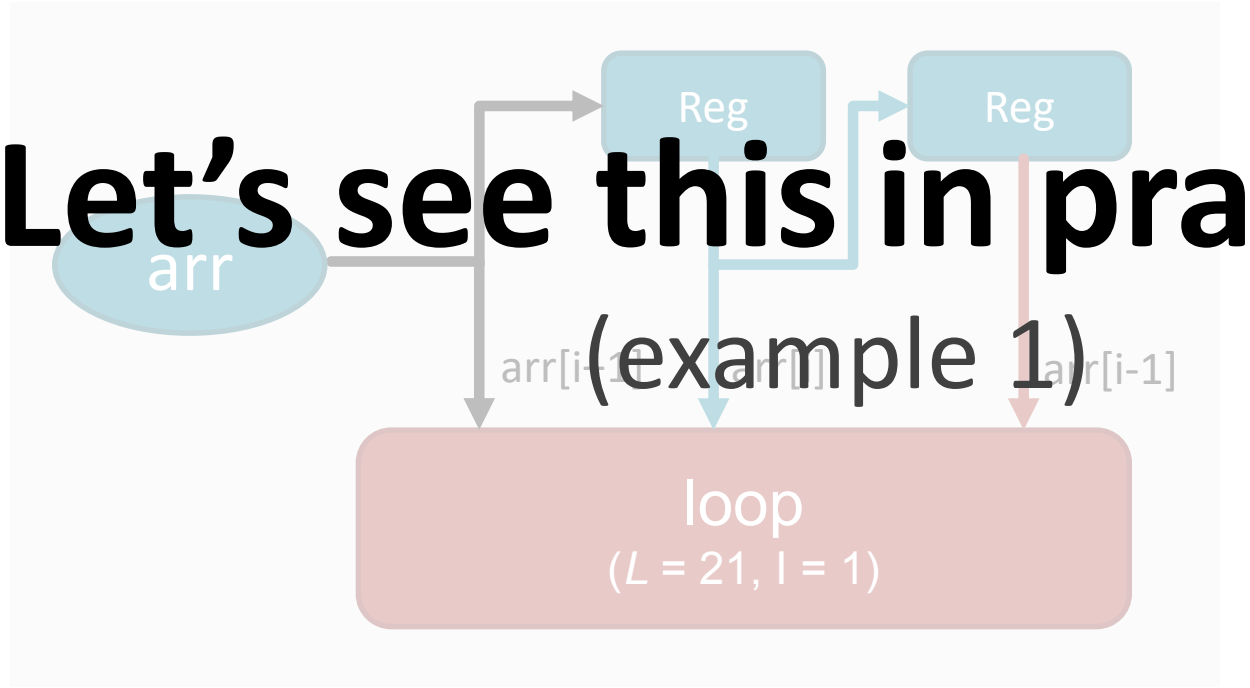
Stage 1

Stage 2

Stage 3

Inserting registers

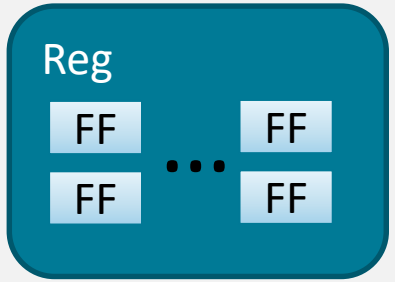
```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



Let's see this in practice…

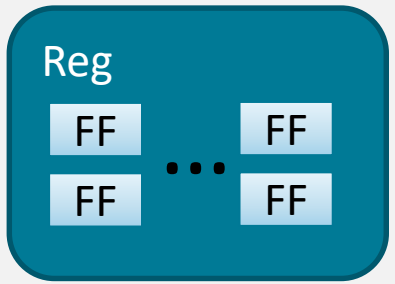
(example 1)

≥ Two classes of storage

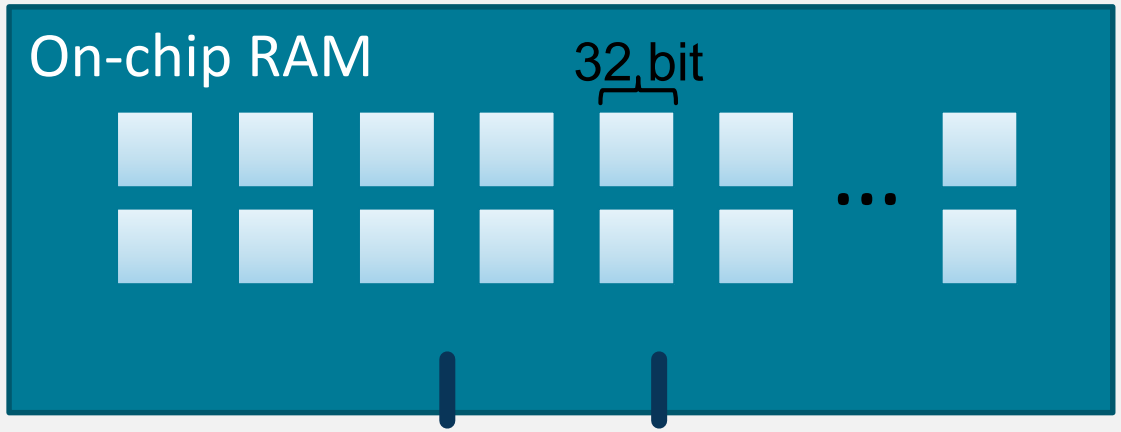


$4\ \text{Byte} = 32\ \text{bit} \cdot 1$

≥ Two classes of storage



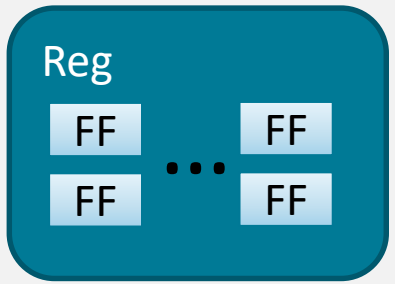
4 Byte = 32 bit · 1



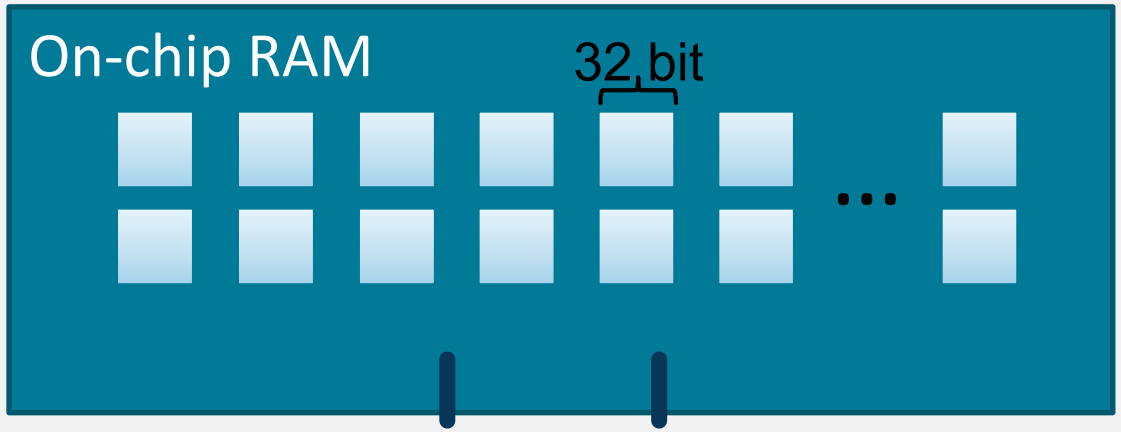
4096 Byte = 32 bit · 1024

(example configuration of Xilinx BRAM)

≥ Two classes of storage



4 Byte = 32 bit · 1

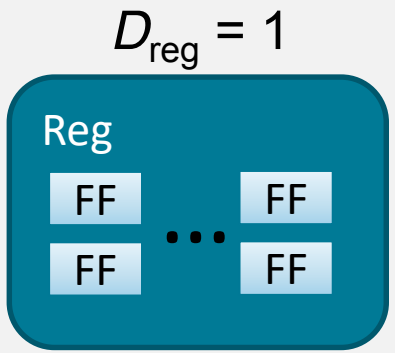


4096 Byte = 32 bit · 1024

(example configuration of Xilinx BRAM)

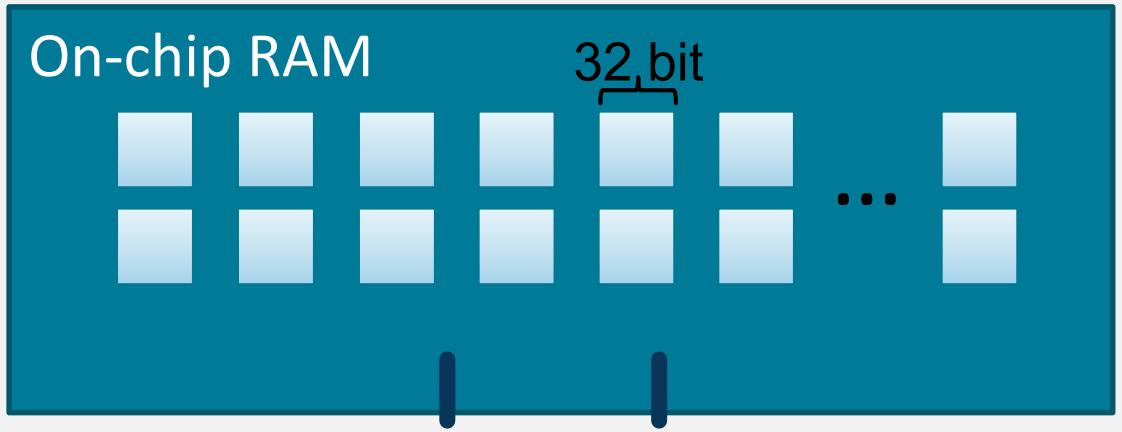
Useful to think of memory in terms of **depth** (*D*) and **width** (*W*)

≥ Two classes of storage



W_{reg} = 32 bit

4 Byte = 32 bit · 1

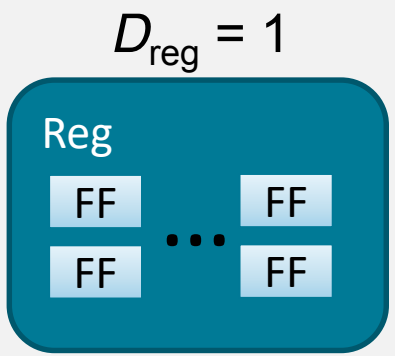


4096 Byte = 32 bit · 1024

(example configuration of Xilinx BRAM)

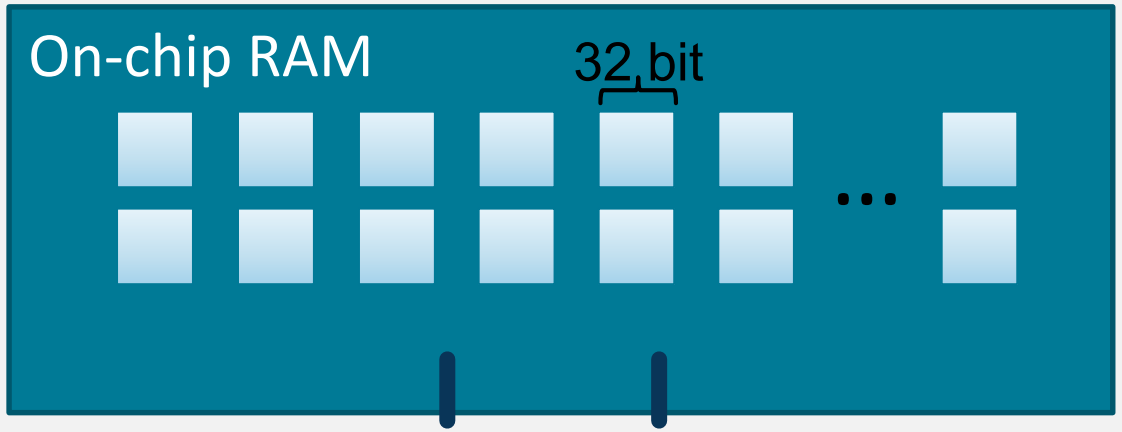
Useful to think of memory in terms of **depth** (*D*) and **width** (*W*)

≥ Two classes of storage



W_{reg} = 32 bit

4 Byte = 32 bit · 1



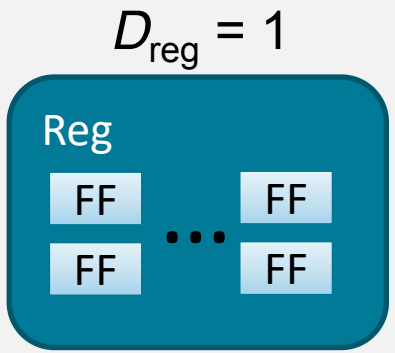
W_{RAM} = 32 bit

4096 Byte = 32 bit · 1024

(example configuration of Xilinx BRAM)

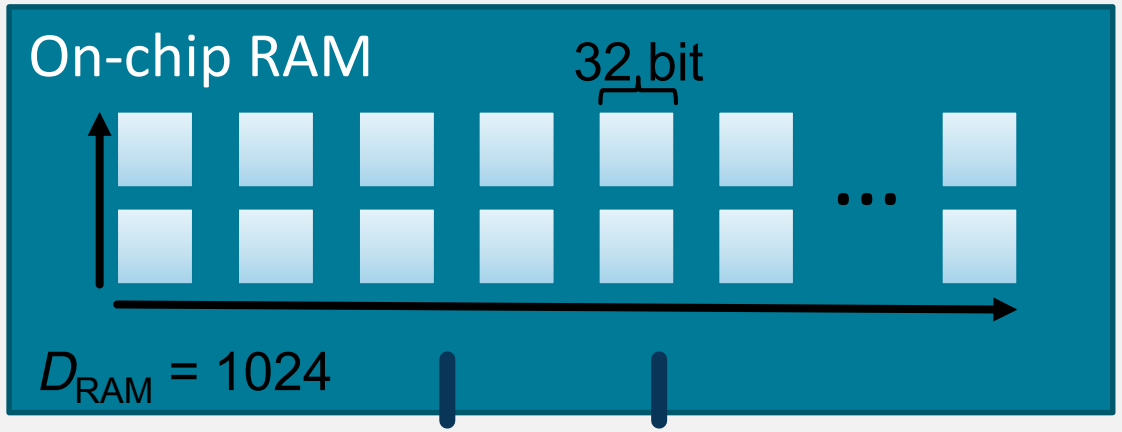
Useful to think of memory in terms of **depth** (*D*) and **width** (*W*)

≥ Two classes of storage



W_{reg} = 32 bit

4 Byte = 32 bit · 1



W_{RAM} = 32 bit

4096 Byte = 32 bit · 1024

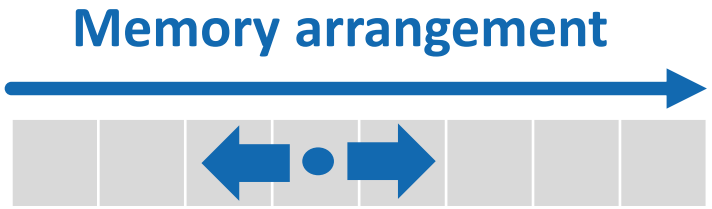
(example configuration of Xilinx BRAM)

Useful to think of memory in terms of **depth** (*D*) and **width** (*W*)

Buffer depth

1D stencil program:

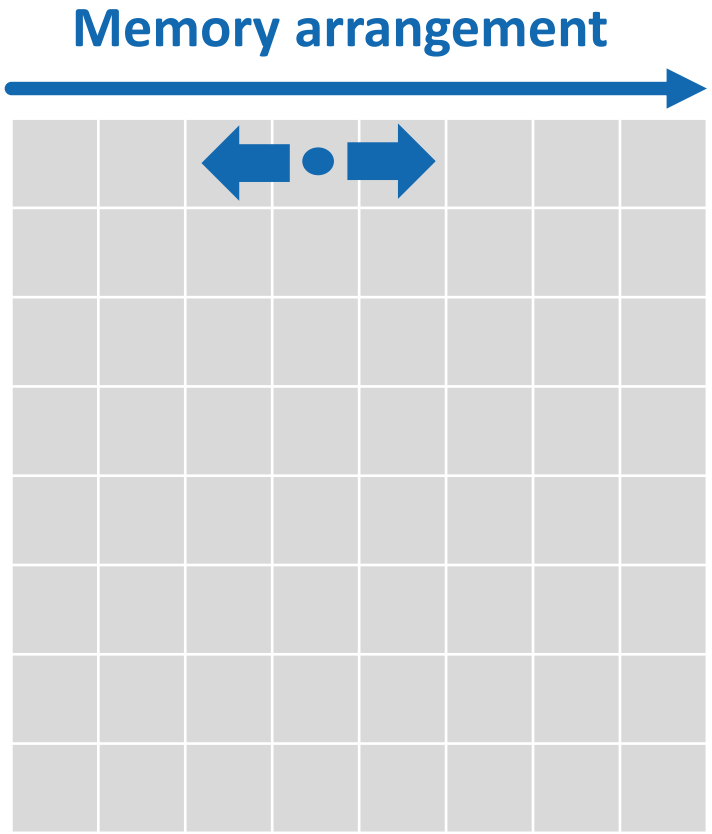
$W = 2,\ D = 1$



Buffer depth

1D stencil program:

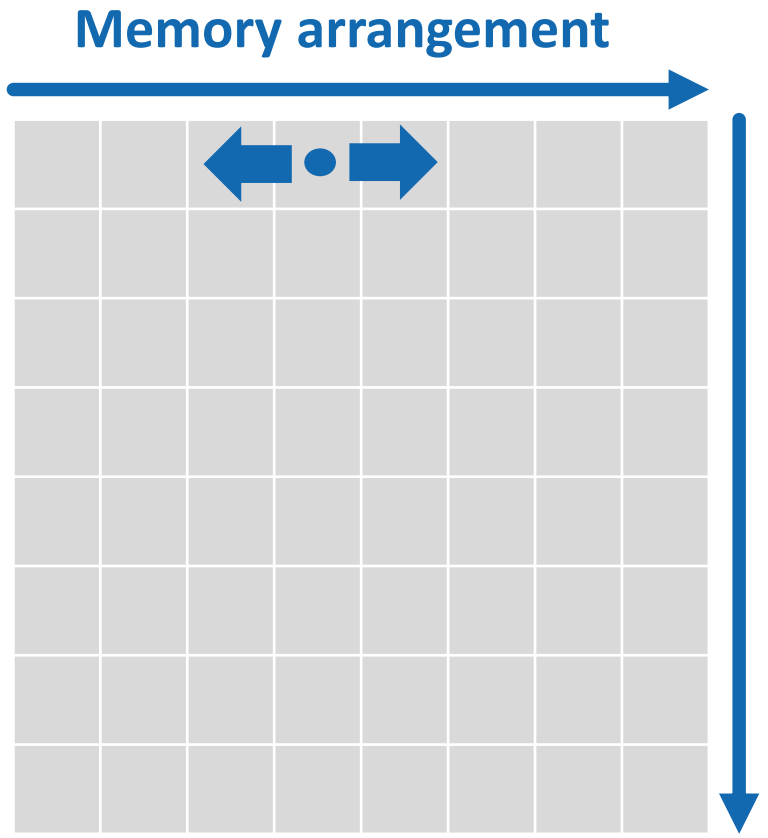
$W = 2,\ D = 1$



Buffer depth

1D stencil program:

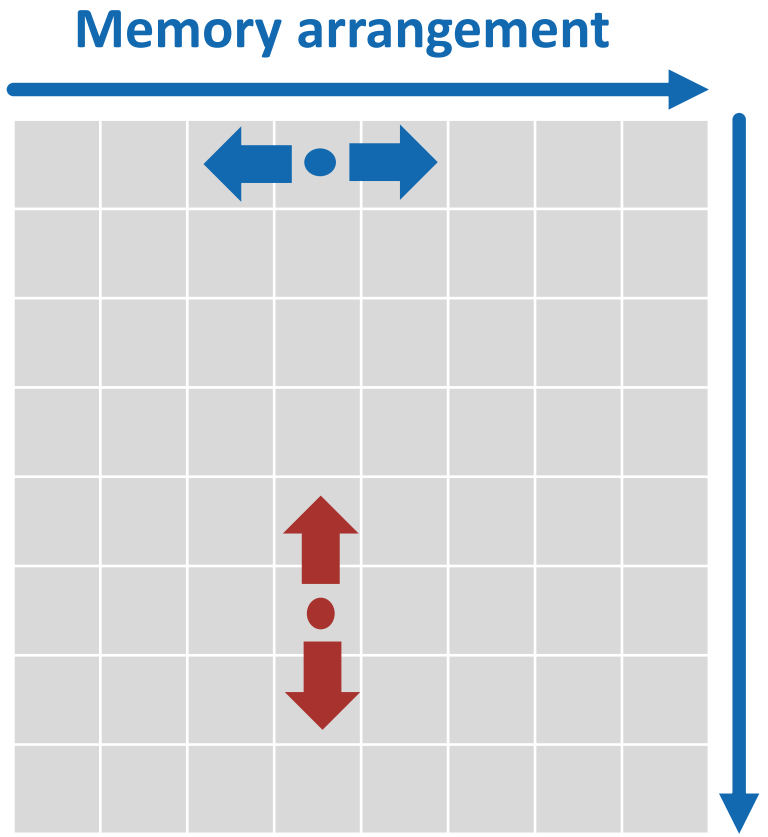
$W = 2,\ D = 1$



Buffer depth

1D stencil program:

$W = 2,\ D = 1$



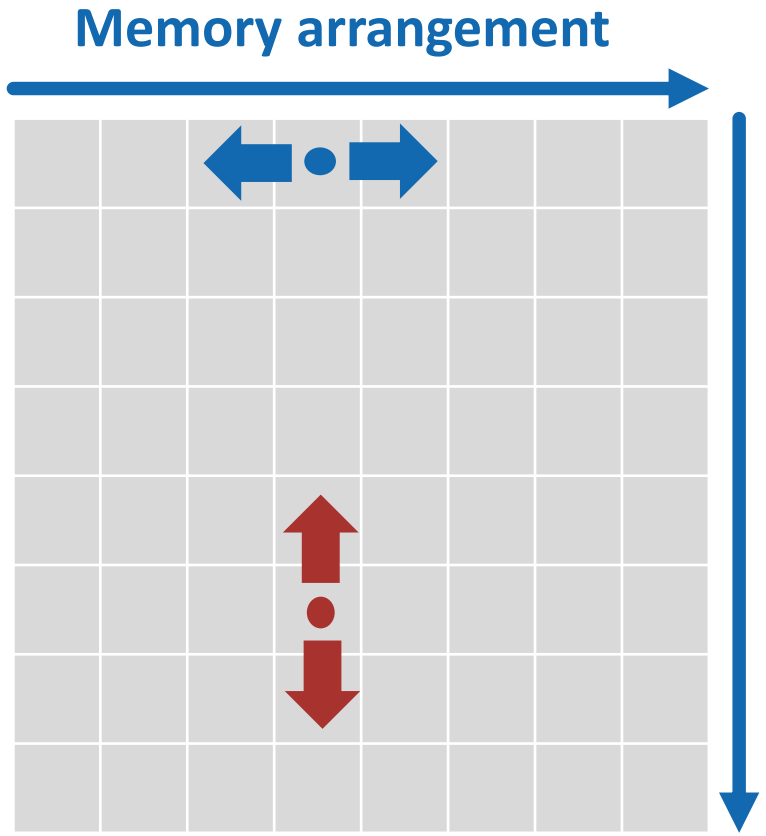
Buffer depth

1D stencil program:

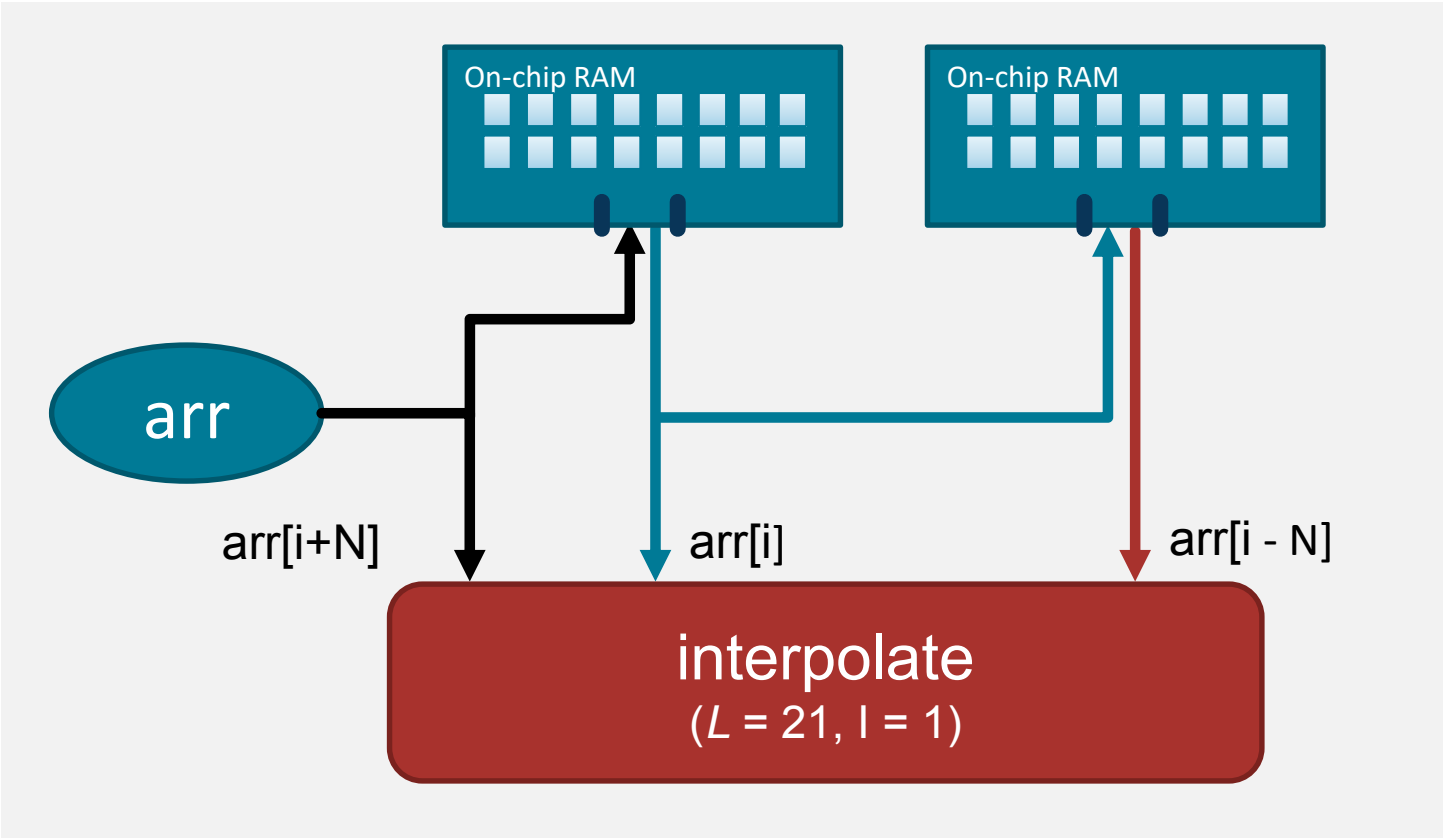
$W = 2,\ D = 1$

2D row-major:

$W = 2,\ D = N$



Modified example

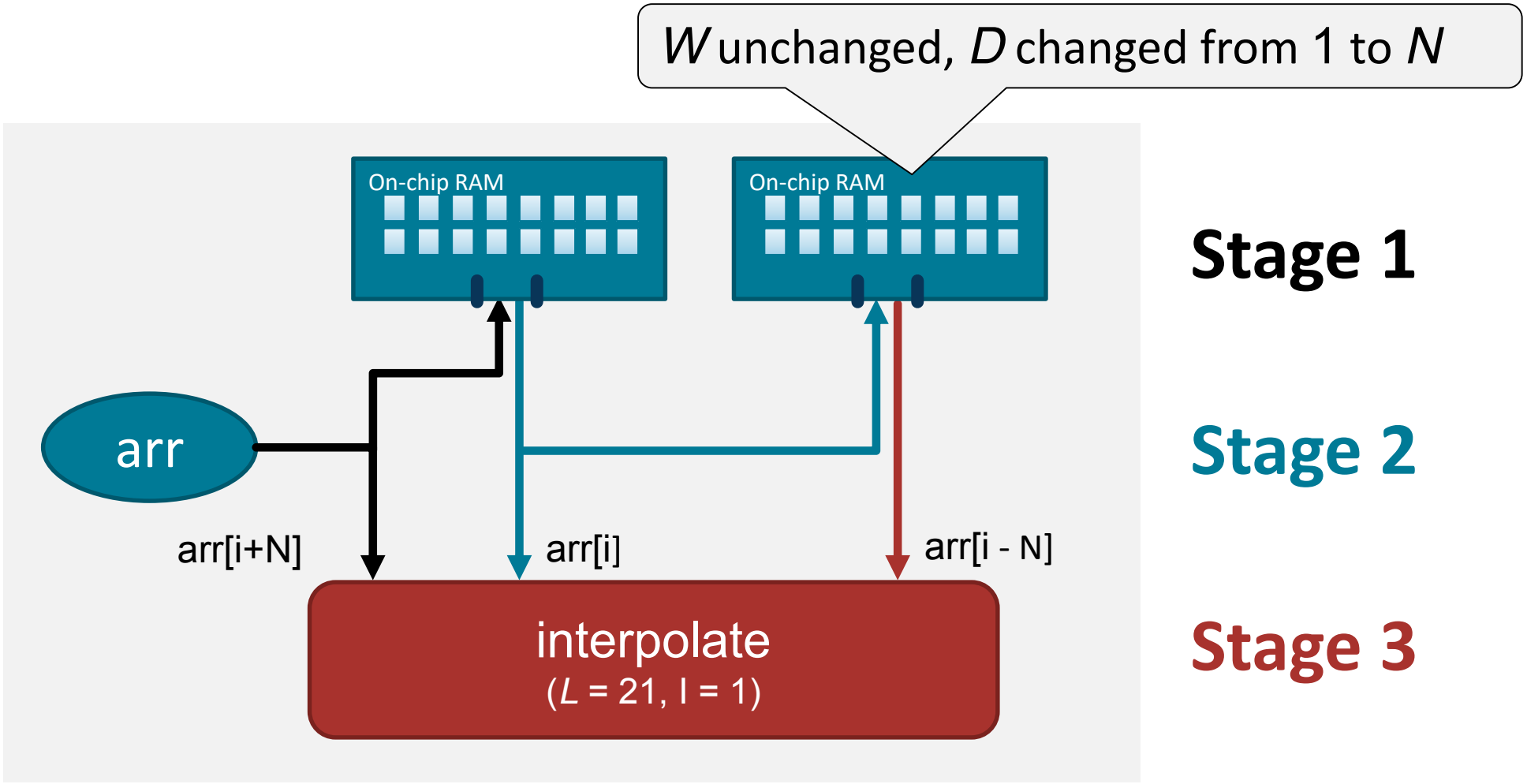


Stage 1

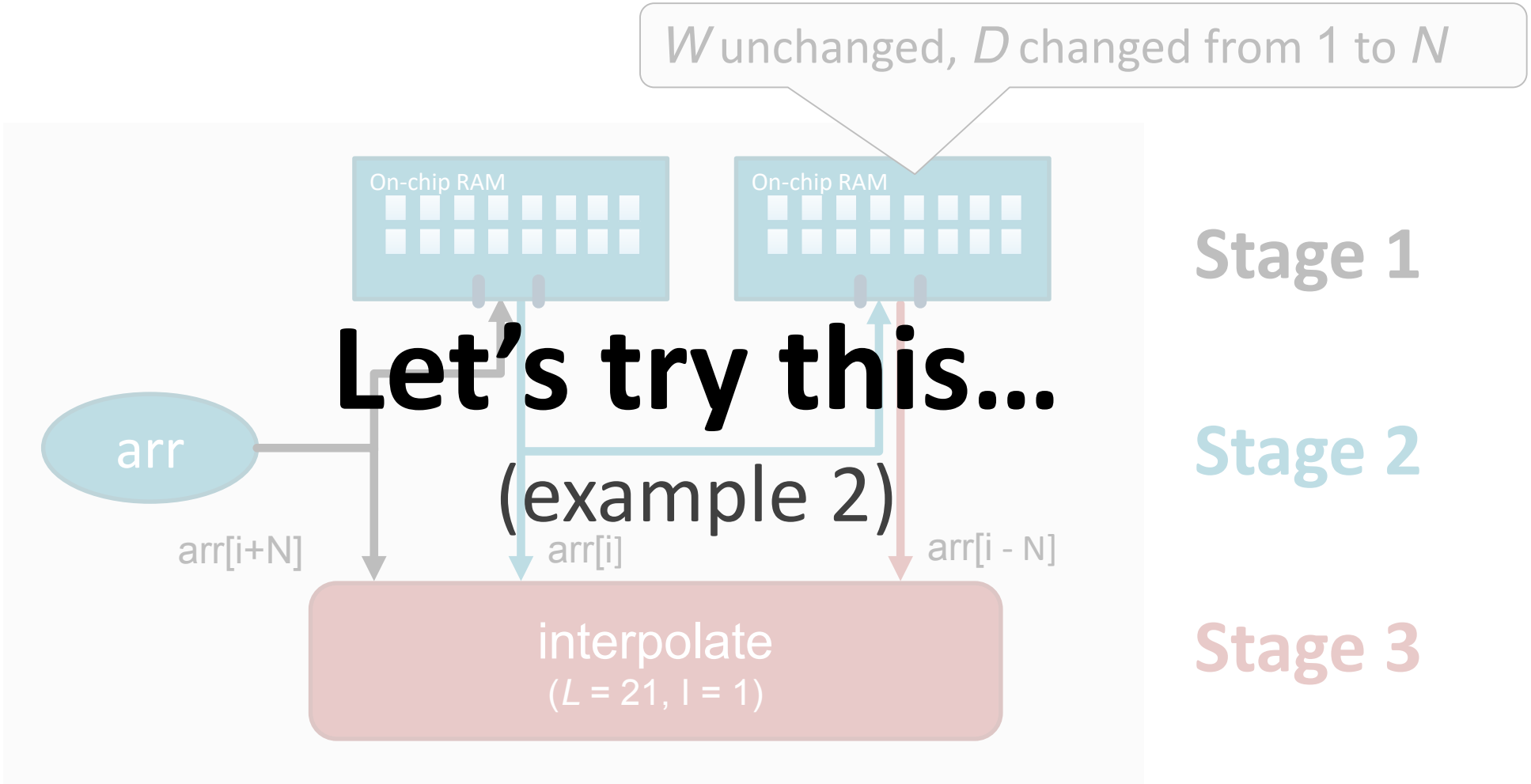
Stage 2

Stage 3

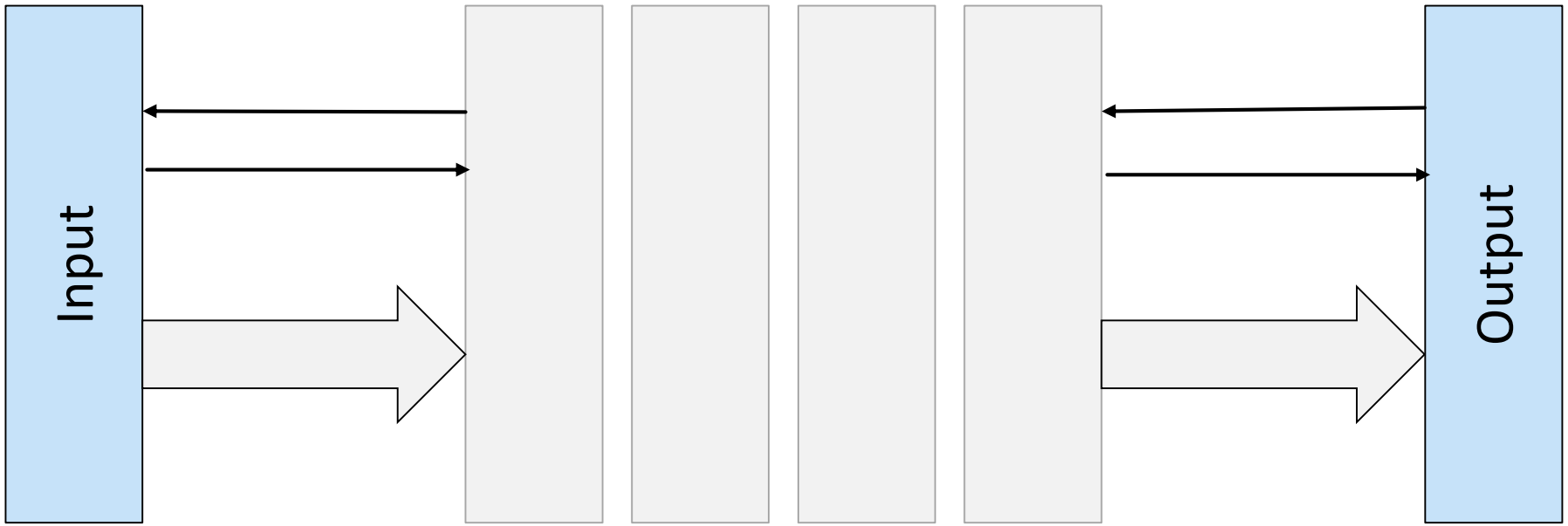
Modified example



Modified example

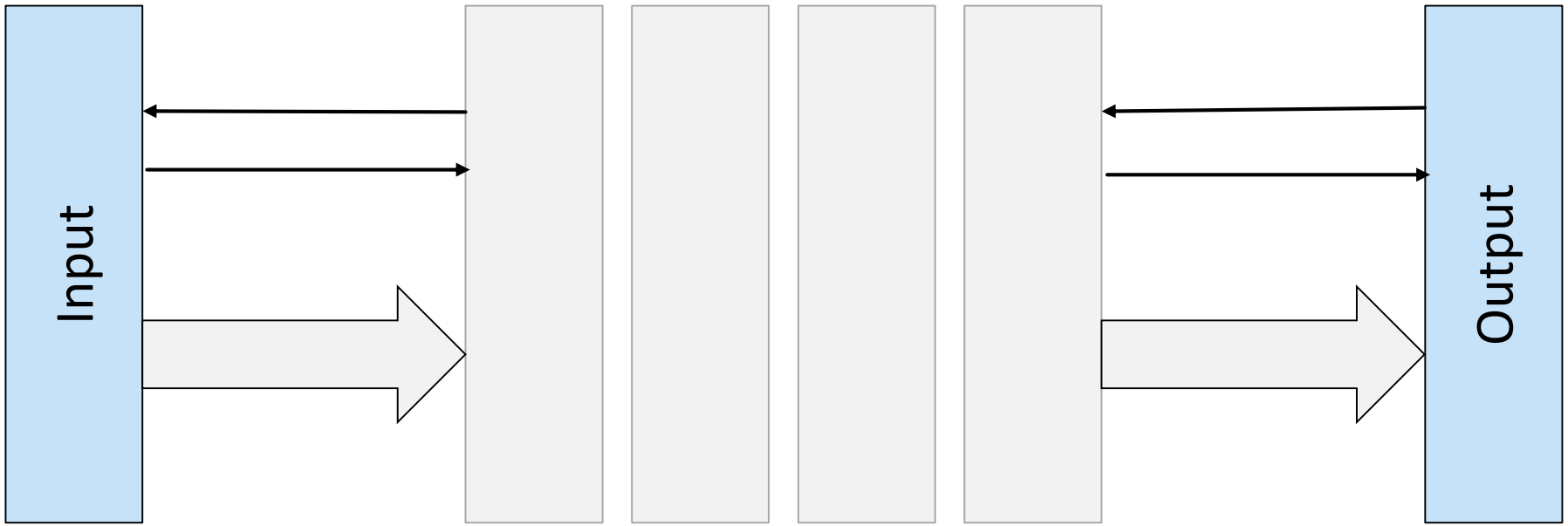


Streams



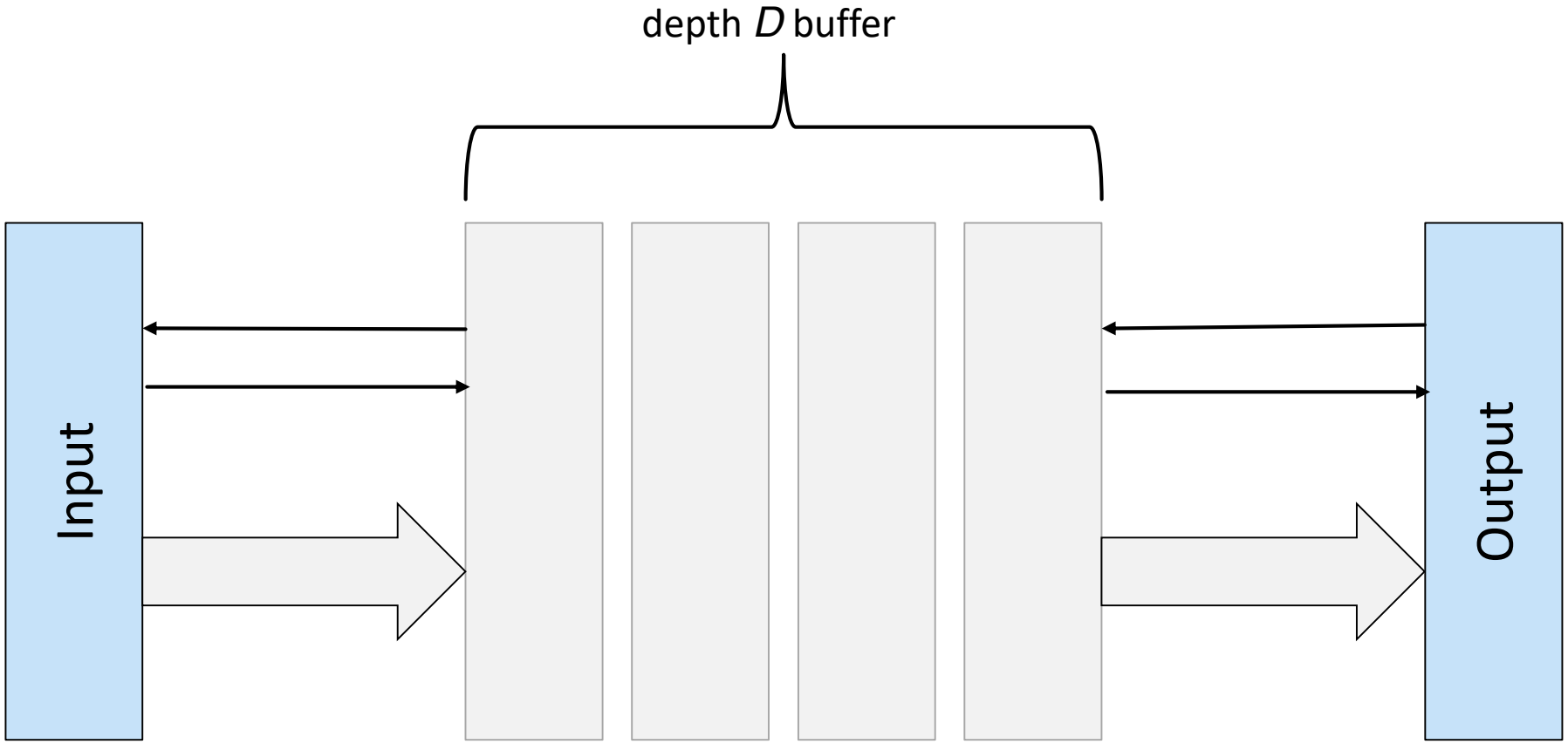
Streams

(FIFOs, queues...)



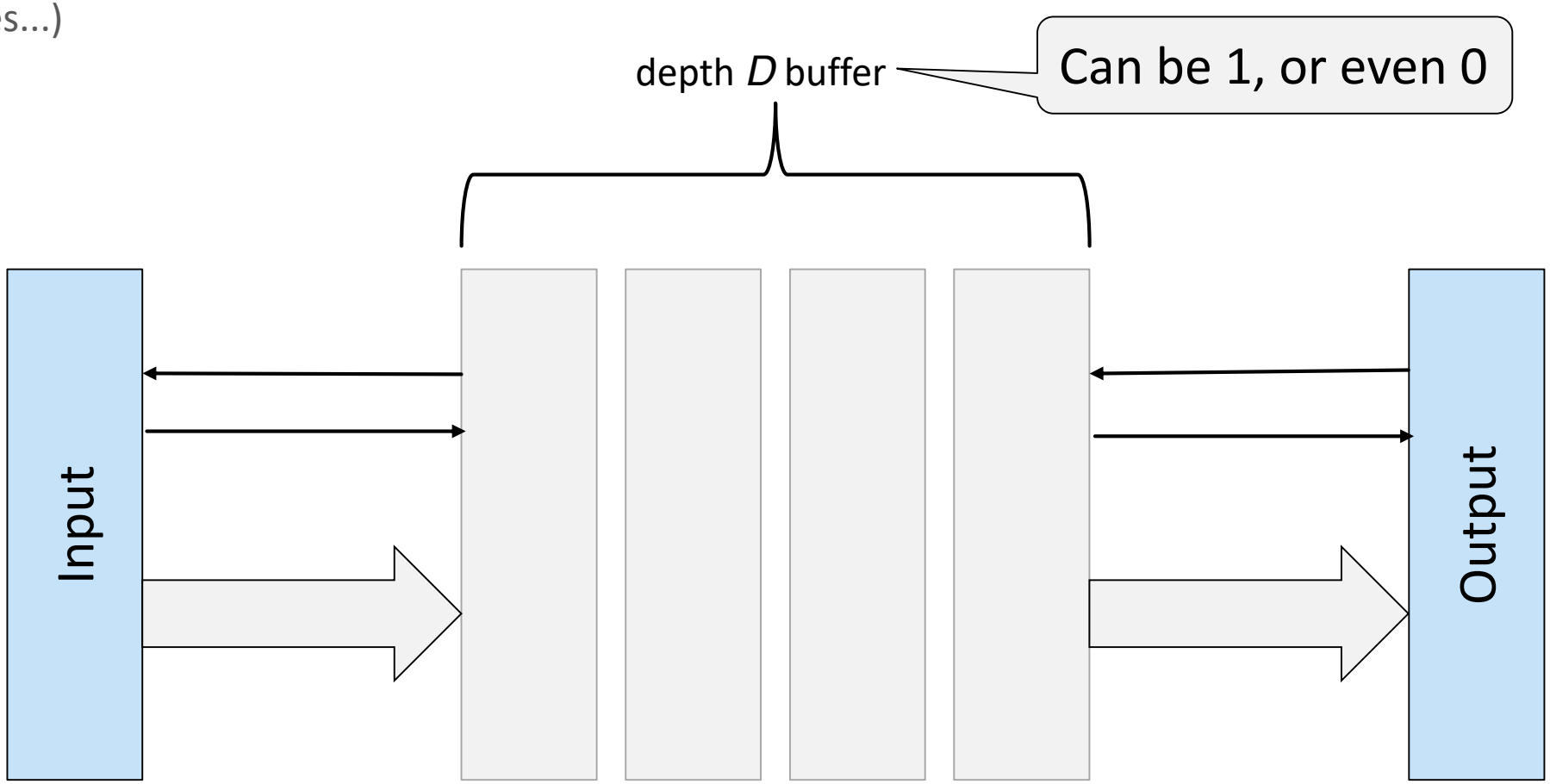
Streams

(FIFOs, queues...)



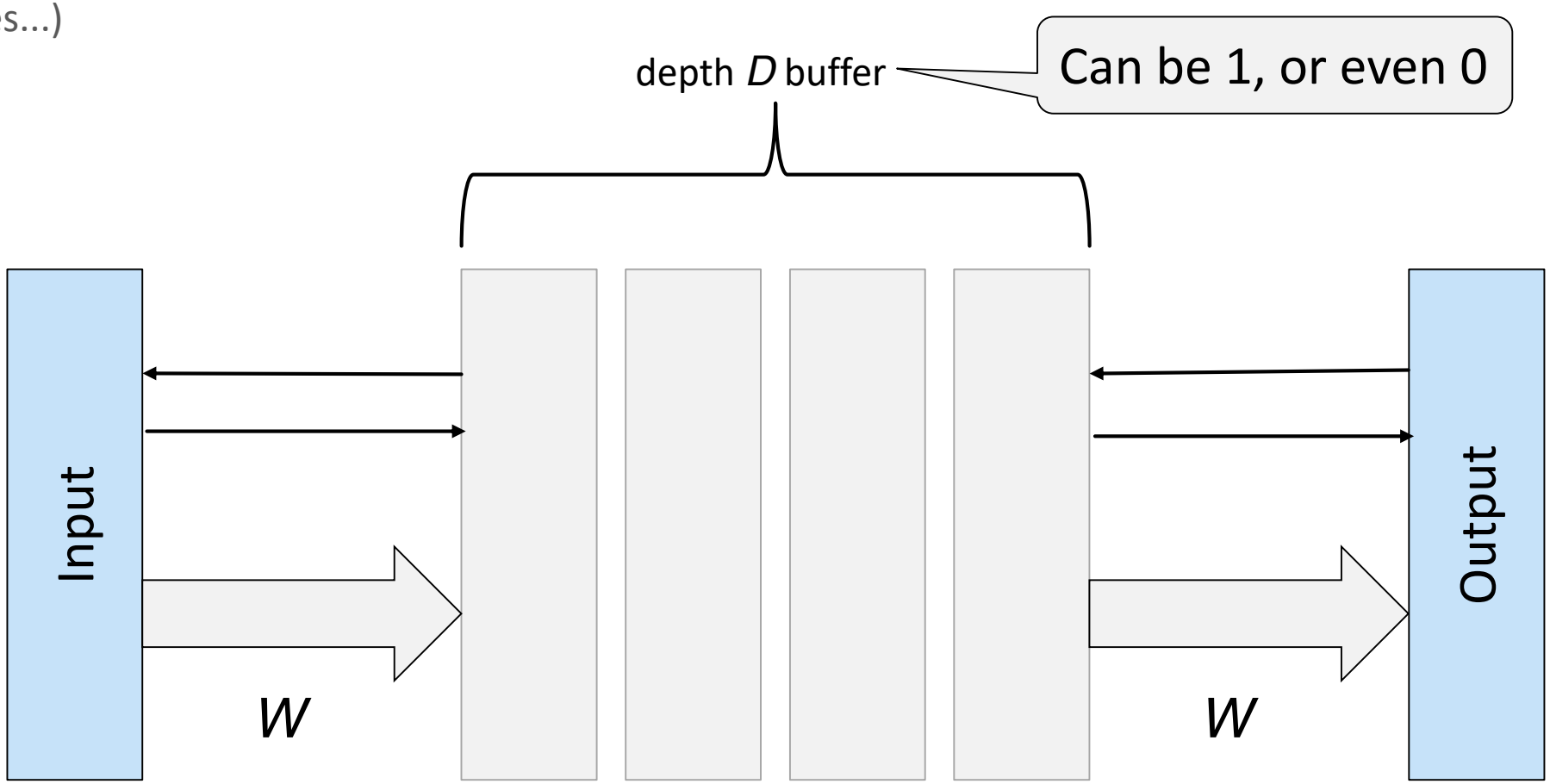
Streams

(FIFOs, queues...)



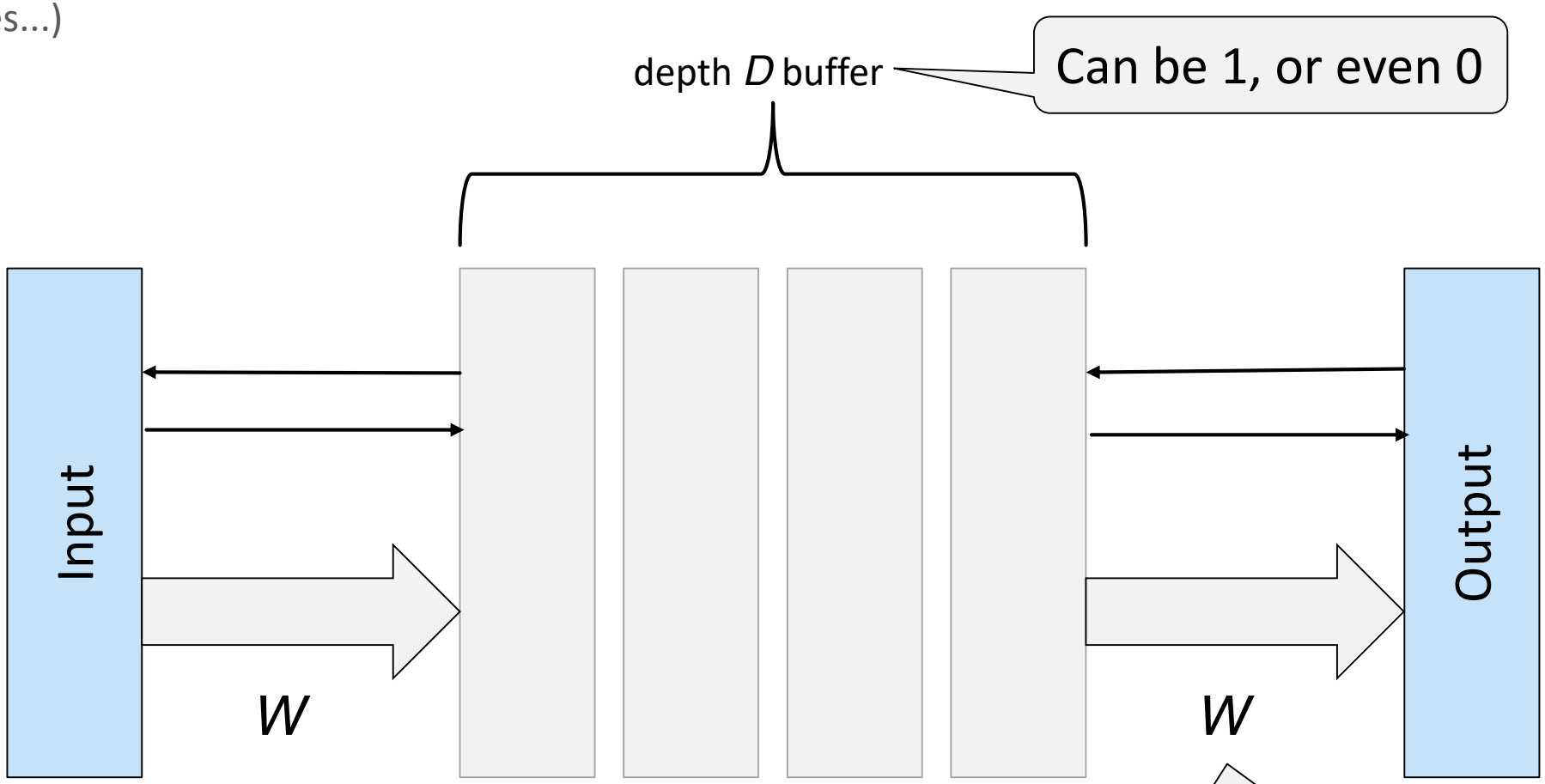
Streams

(FIFOs, queues...)



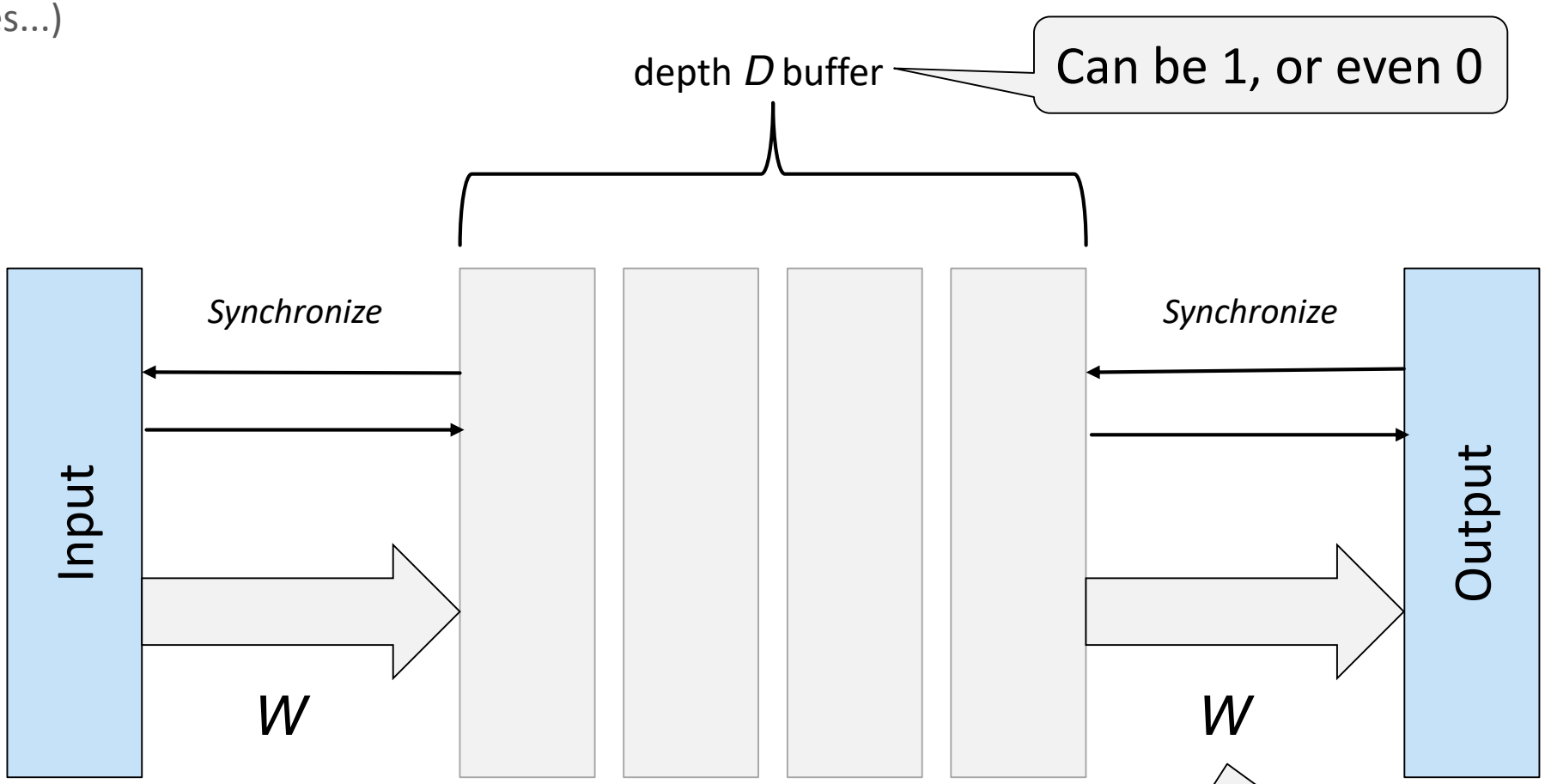
Streams

(FIFOs, queues...)

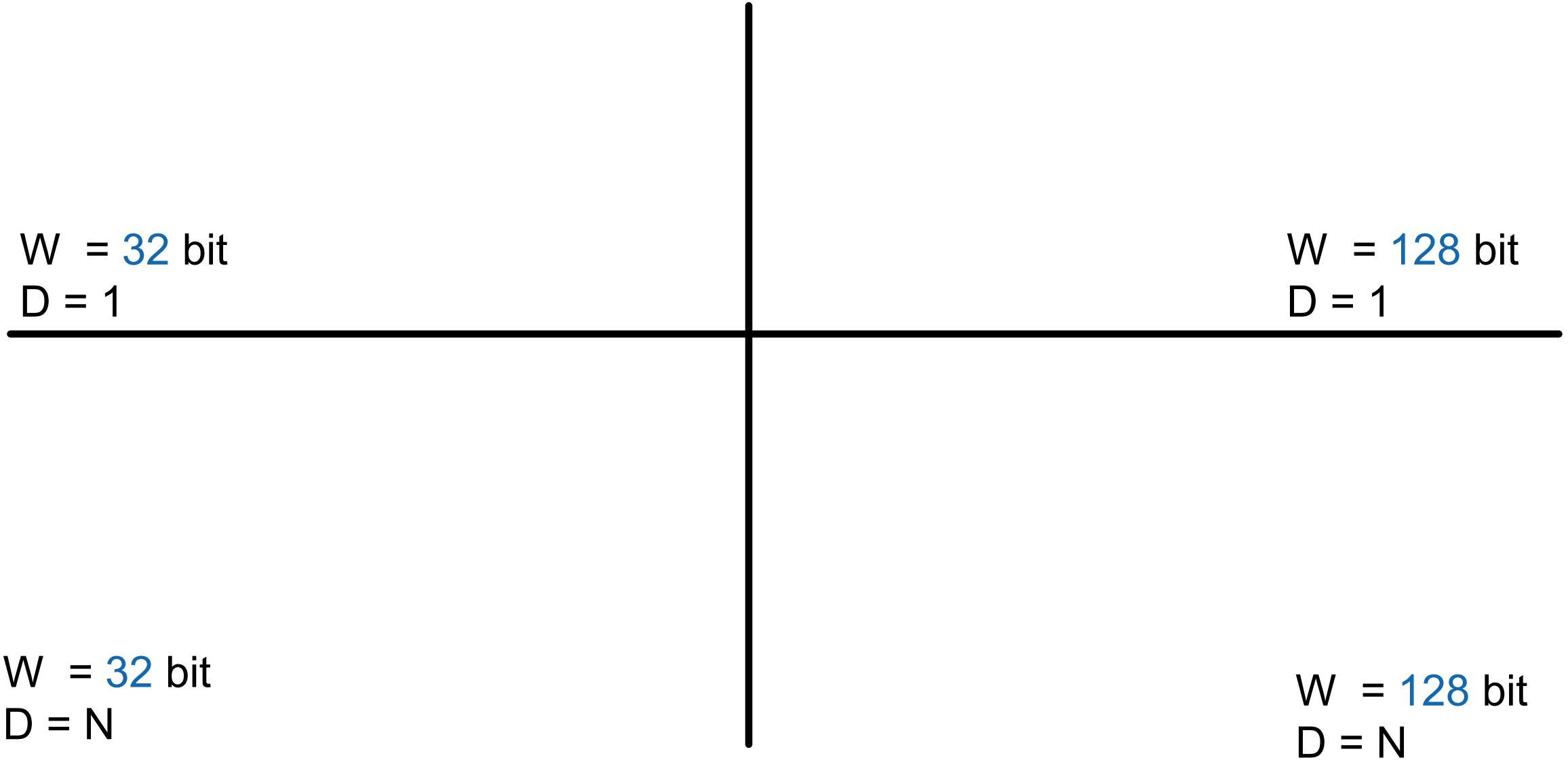


Streams

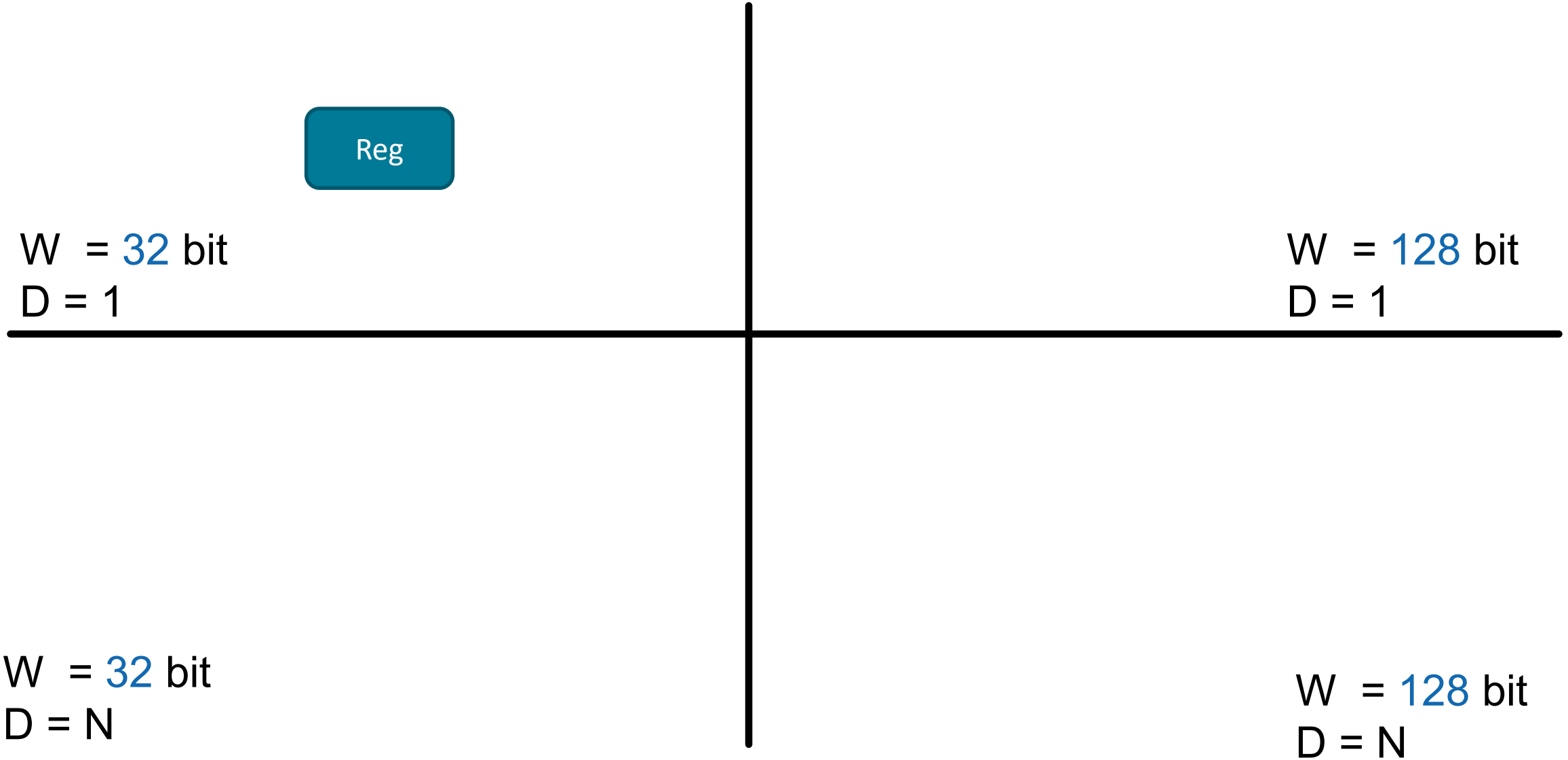
(FIFOs, queues...)



Thinking in width and depth



Thinking in width and depth



Thinking in width and depth

```
float reg;
```

W = 32 bit
D = 1

W = 128 bit
D = 1

W = 32 bit
D = N

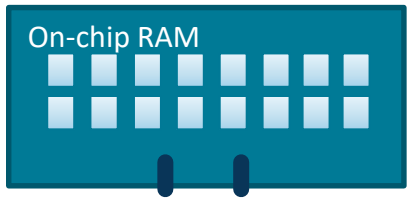
W = 128 bit
D = N

Thinking in width and depth

```
float reg;
```

W = 32 bit
D = 1

W = 128 bit
D = 1



W = 32 bit
D = N

W = 128 bit
D = N

Thinking in width and depth

```
float reg;
```

W = 32 bit
D = 1

W = 128 bit
D = 1

On-chip RAM

```
float buffer[N];
Stream<float> fifo(N);
```

W = 32 bit
D = N

W = 128 bit
D = N

Thinking in width and depth

```
float reg;
```

W = 32 bit
D = 1

W = 128 bit
D = 1

```
float buffer[N];
Stream<float> fifo(N);
```

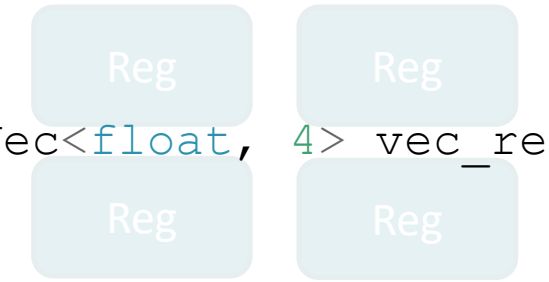
W = 32 bit
D = N

W = 128 bit
D = N

Thinking in width and depth

```
float reg;
```

W = 32 bit
D = 1




```
Vec<float, 4> vec_reg;
```

W = 128 bit
D = 1

```
float buffer[N];
Stream<float> fifo(N);
```

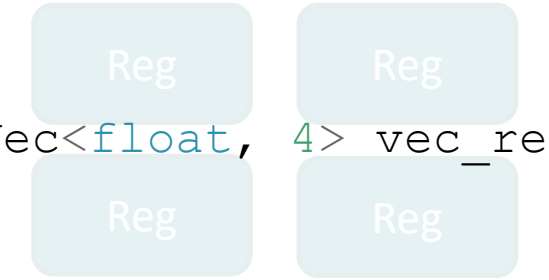
W = 32 bit
D = N

W = 128 bit
D = N

Thinking in width and depth

```
float reg;
```

W = 32 bit
D = 1

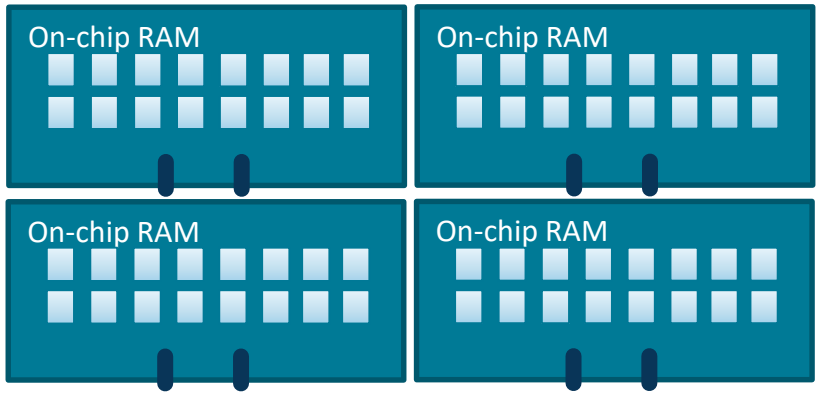



```
Vec<float, 4> vec_reg;
```

W = 128 bit
D = 1

```
float buffer[N];
Stream<float> fifo(N);
```

W = 32 bit
D = N

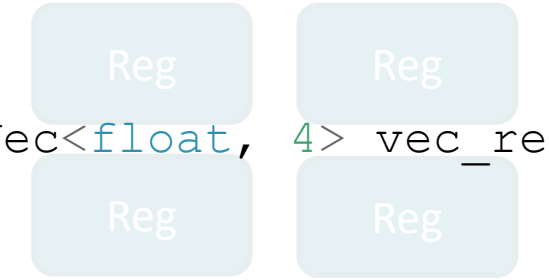


W = 128 bit
D = N

Thinking in width and depth

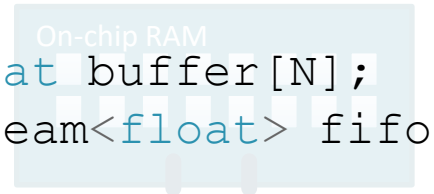
```
float reg;
```

W = 32 bit
D = 1



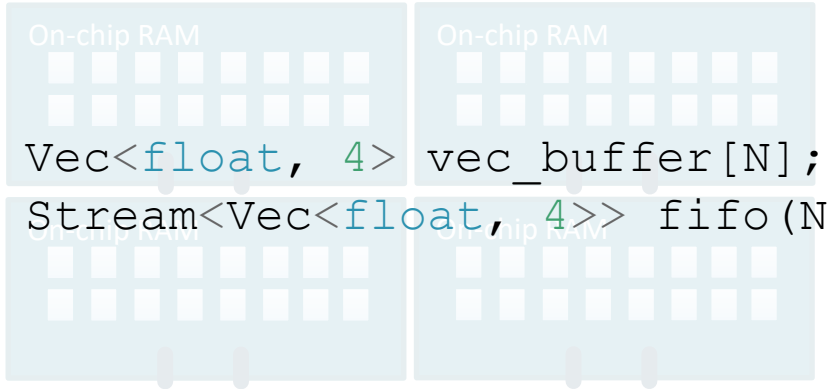

```
Vec<float, 4> vec_reg;
```

W = 128 bit
D = 1




```
float buffer[N];
Stream<float> fifo(N);
```

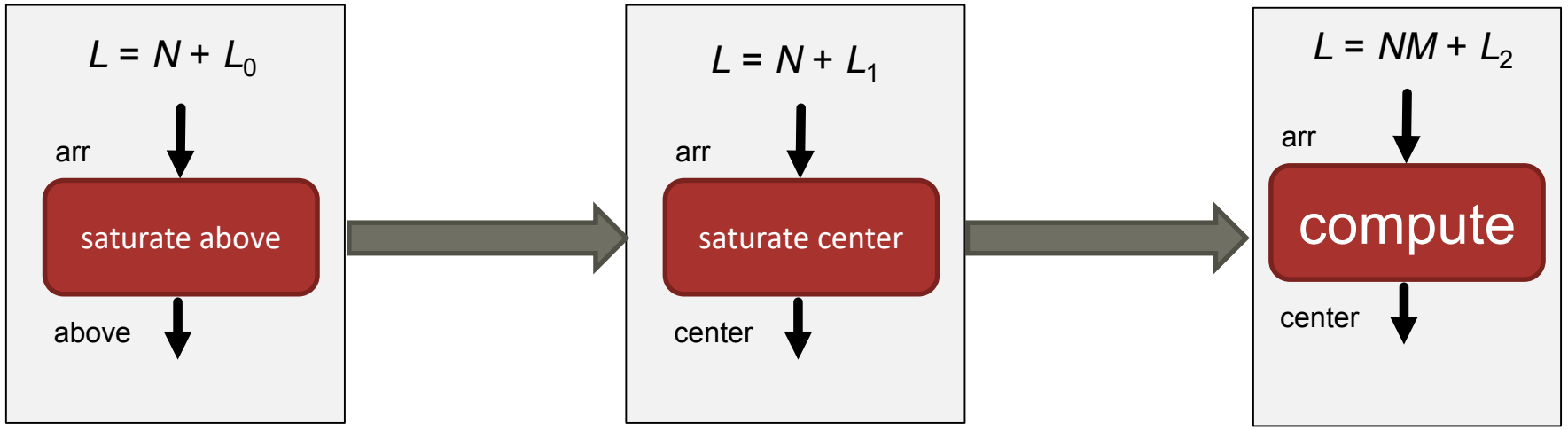
W = 32 bit
D = N




```
Vec<float, 4> vec_buffer[N];
Stream<Vec<float, 4>> fifo(N);
```

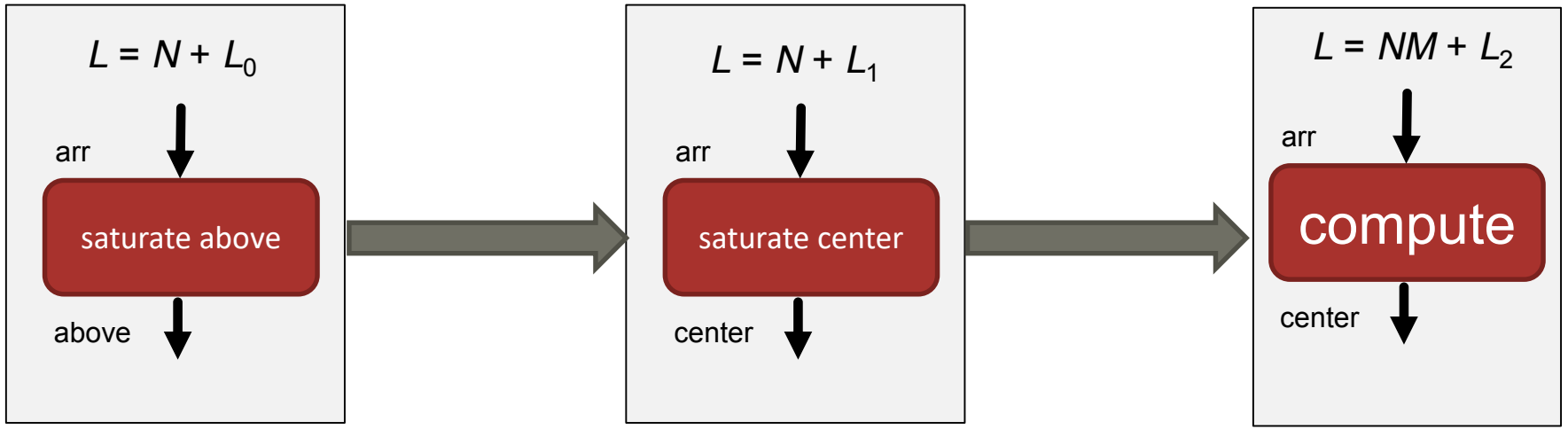
W = 128 bit
D = N

Finite state machine



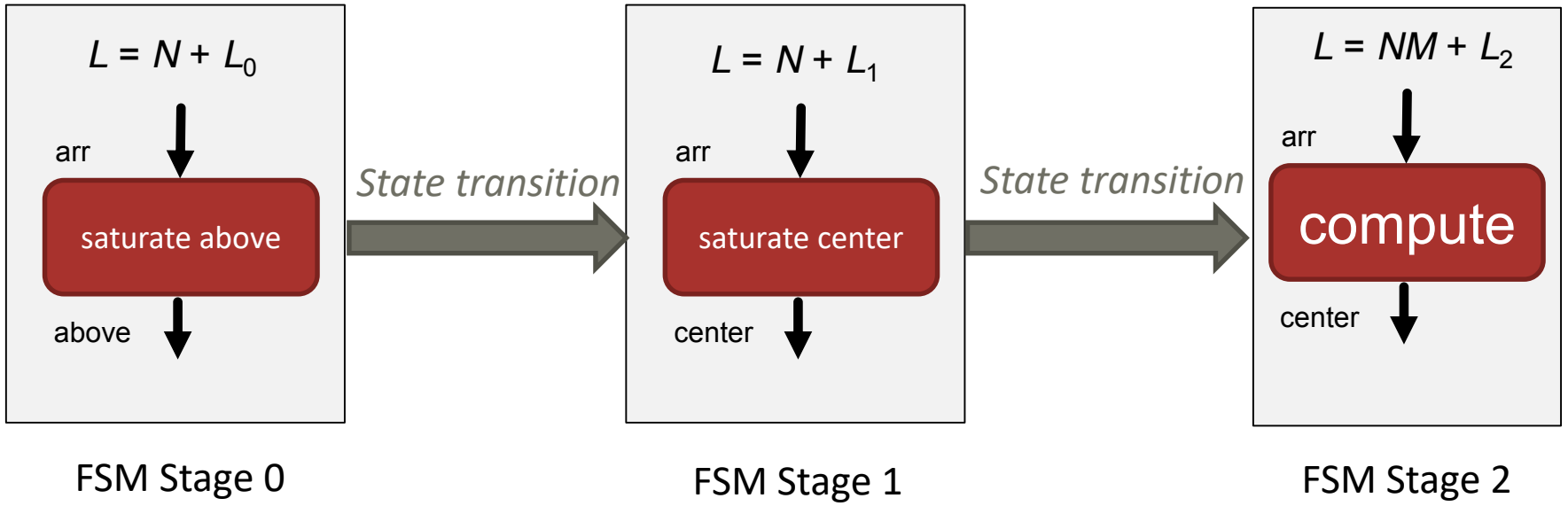
FSM Stage 0 FSM Stage 1 FSM Stage 2

Finite state machine

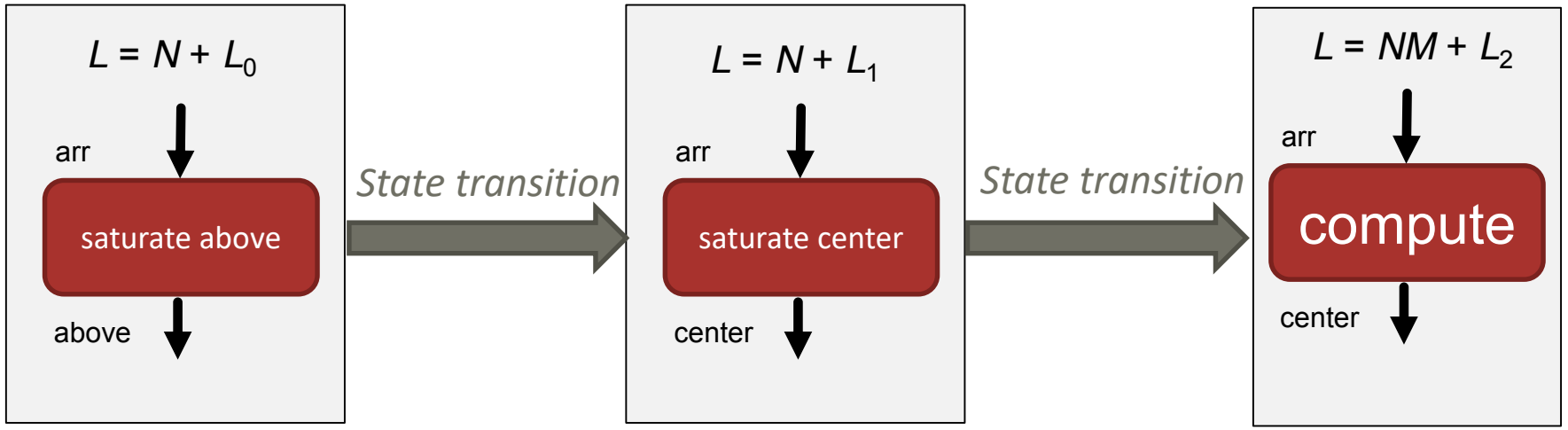


FSM Stage 0 FSM Stage 1 FSM Stage 2

Finite state machine

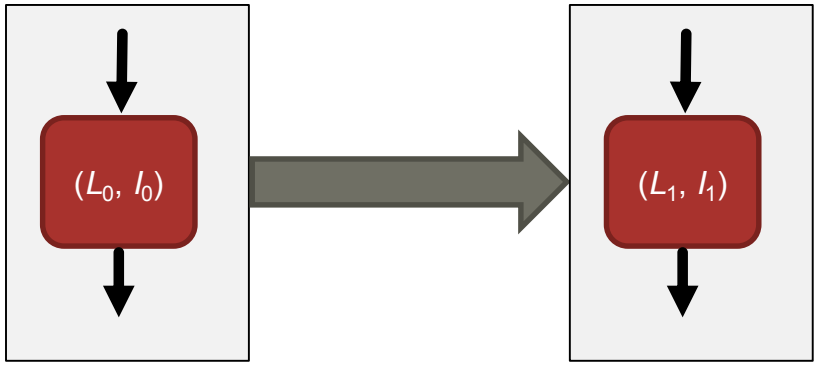


Finite state machine

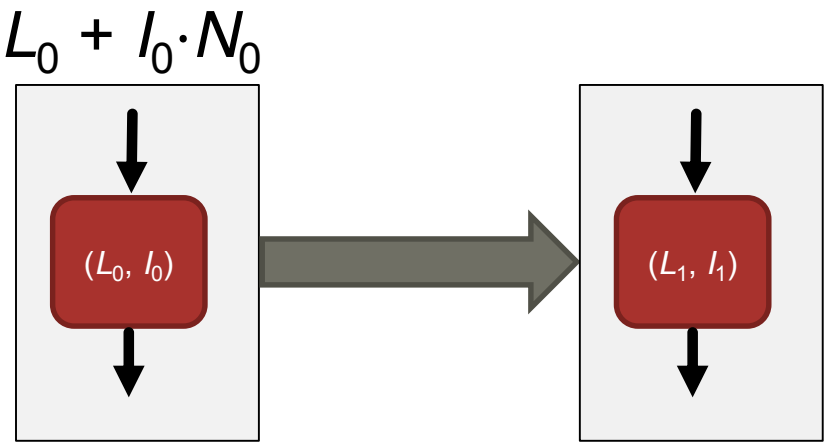


Sequential execution of
(pipeline-)**parallel** stages

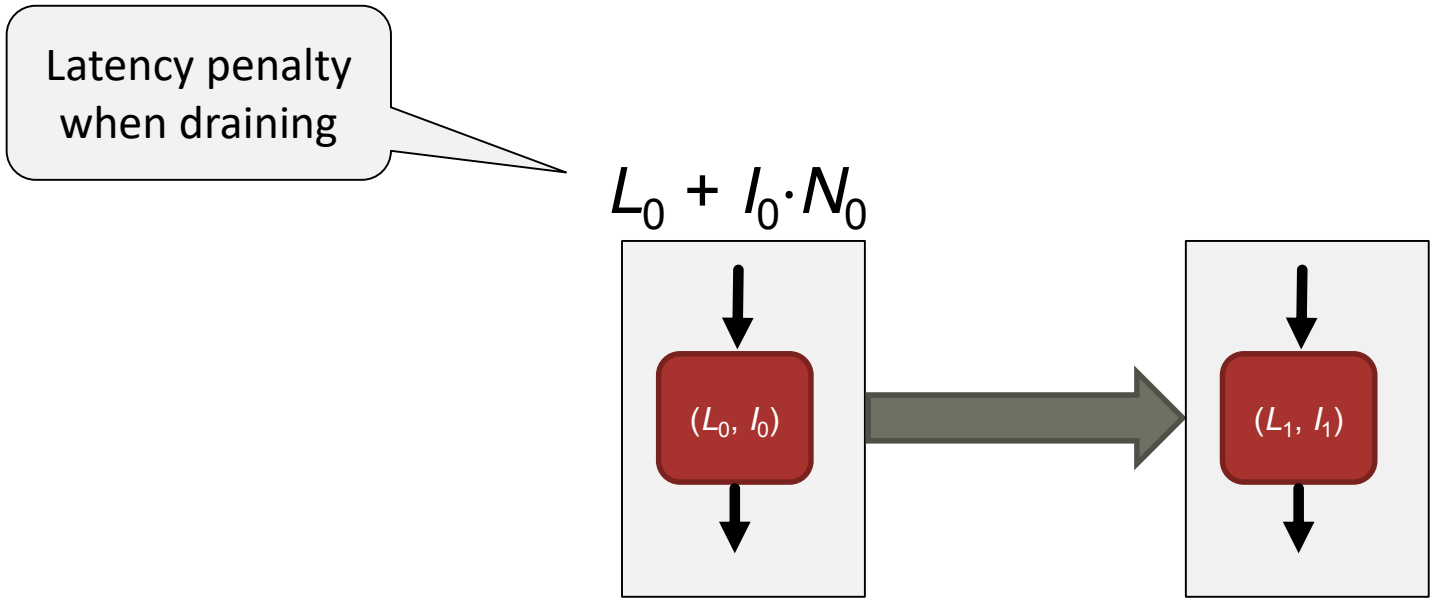
Draining and saturation phases



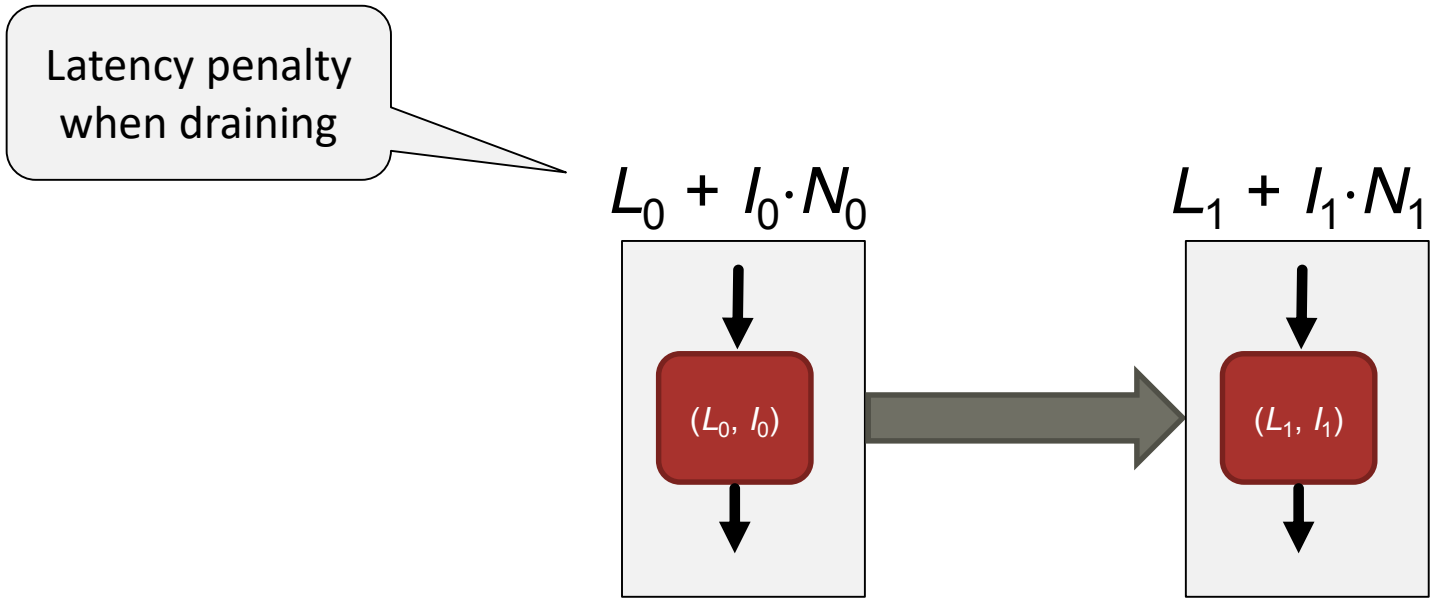
Draining and saturation phases



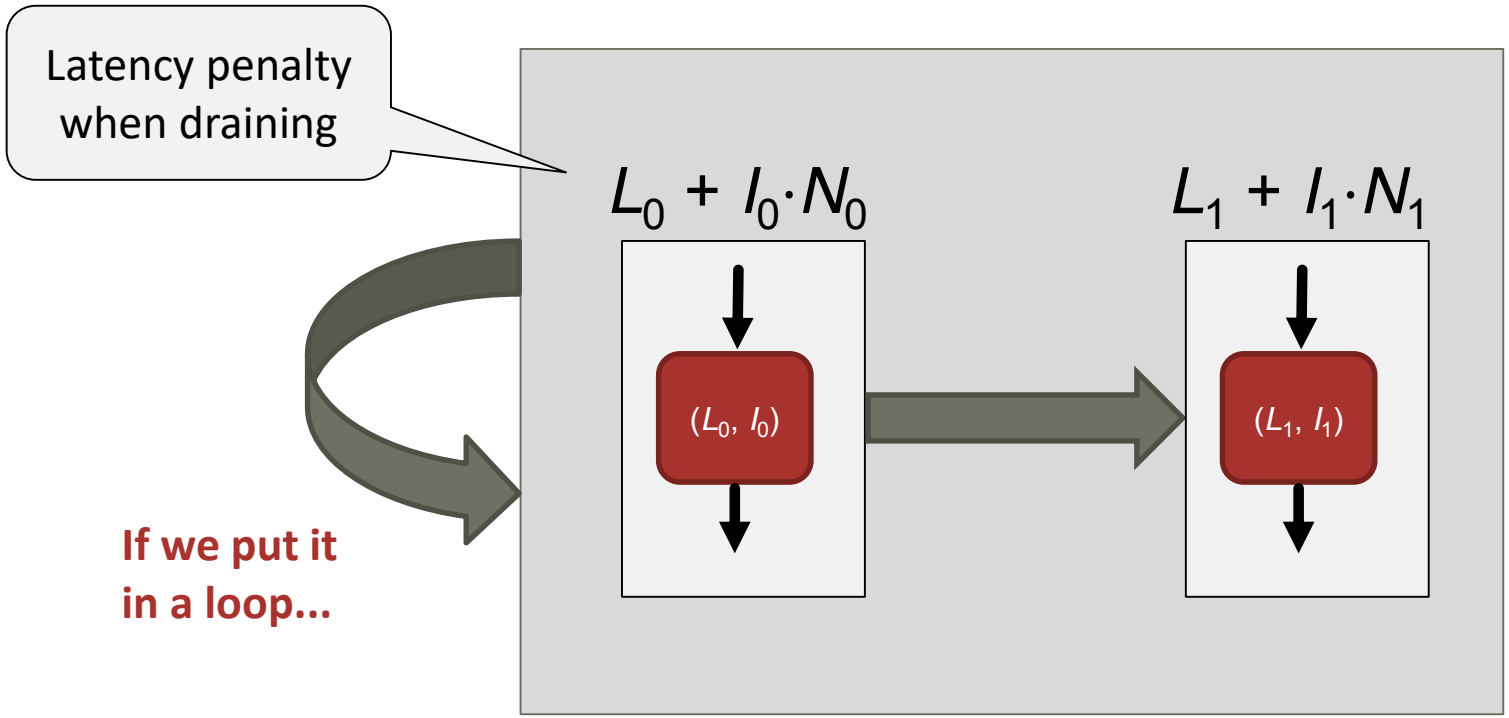
Draining and saturation phases



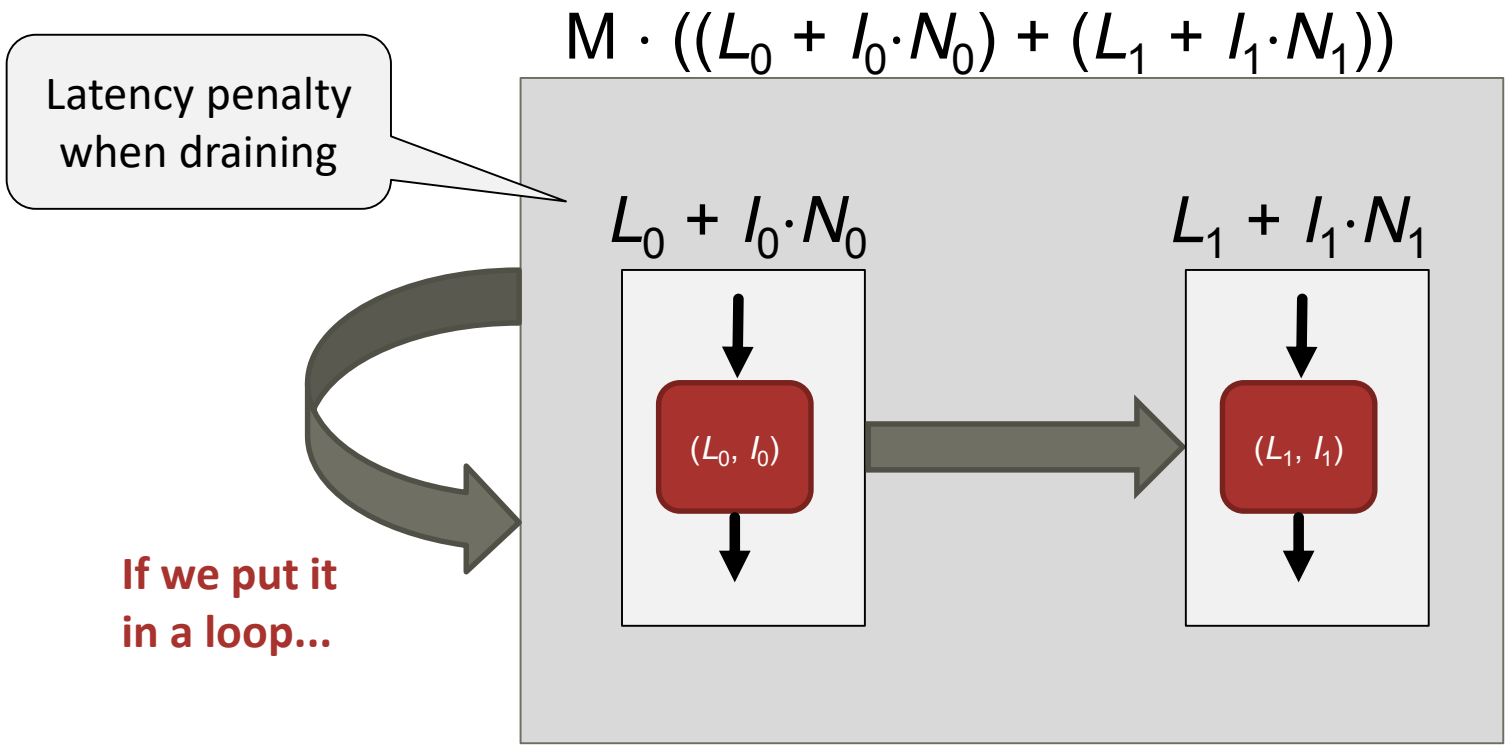
Draining and saturation phases



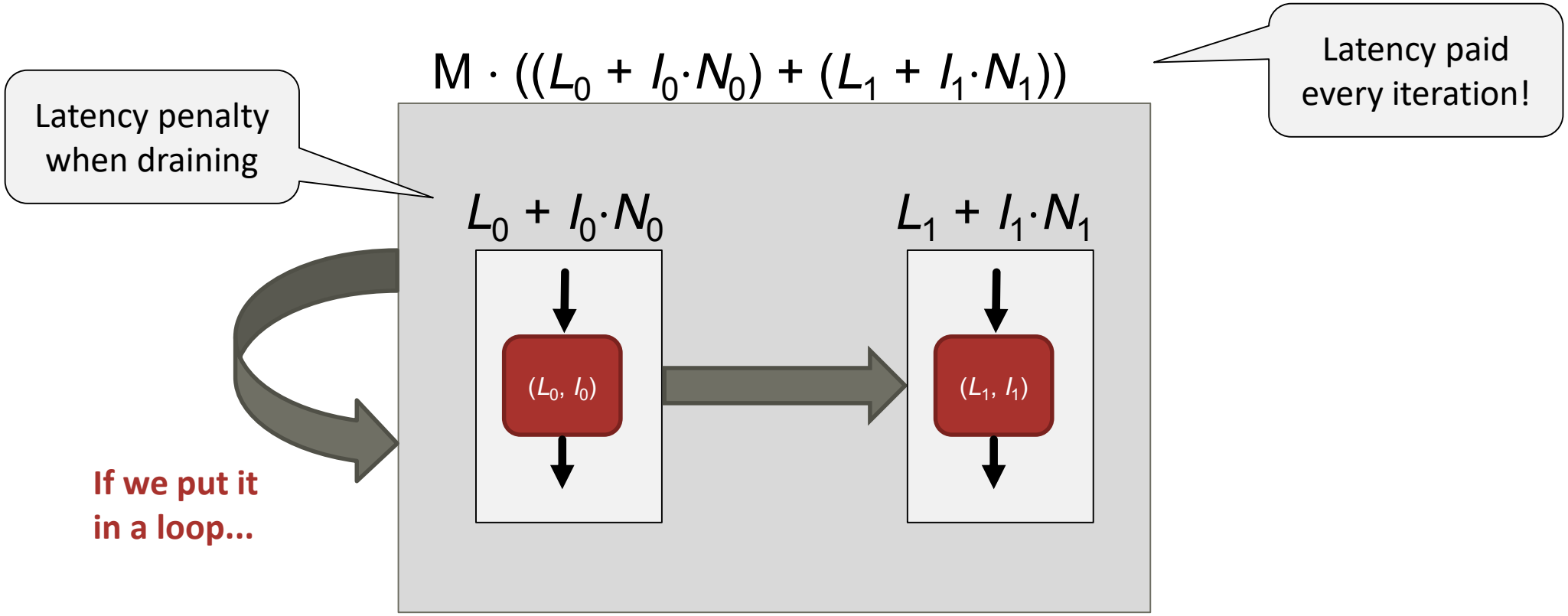
Draining and saturation phases



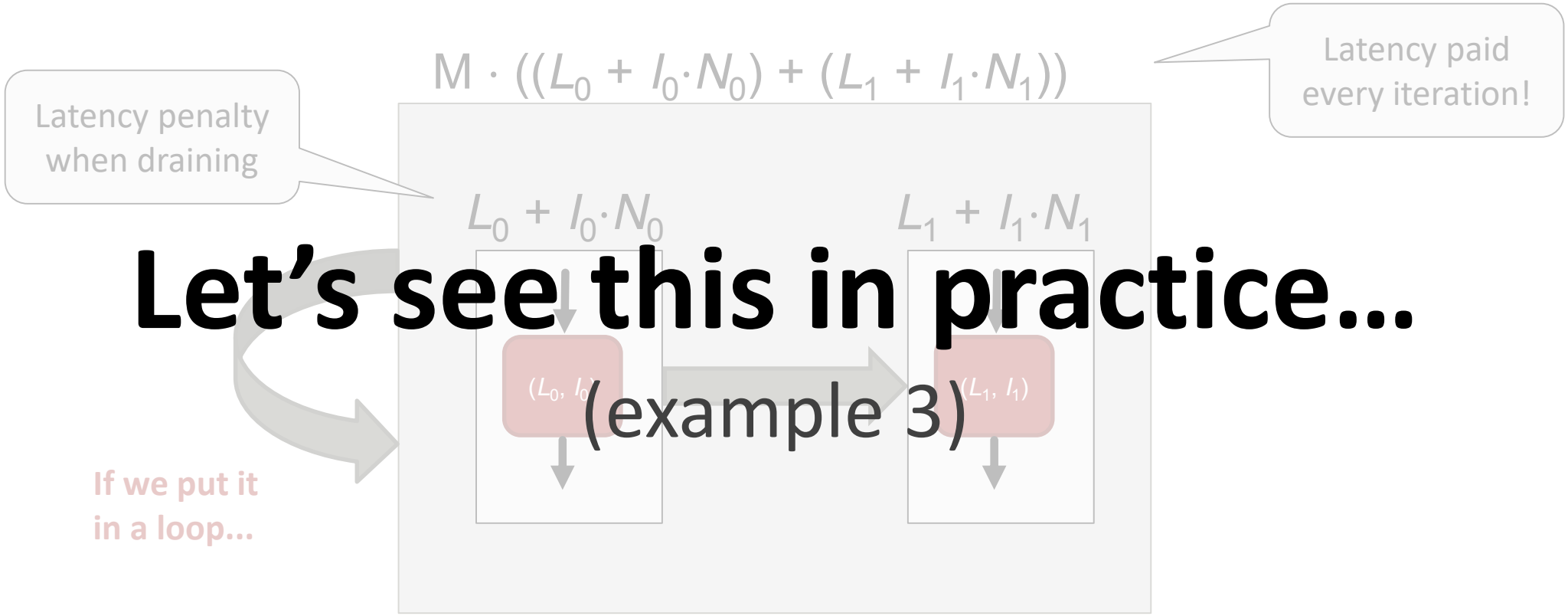
Draining and saturation phases



Draining and saturation phases



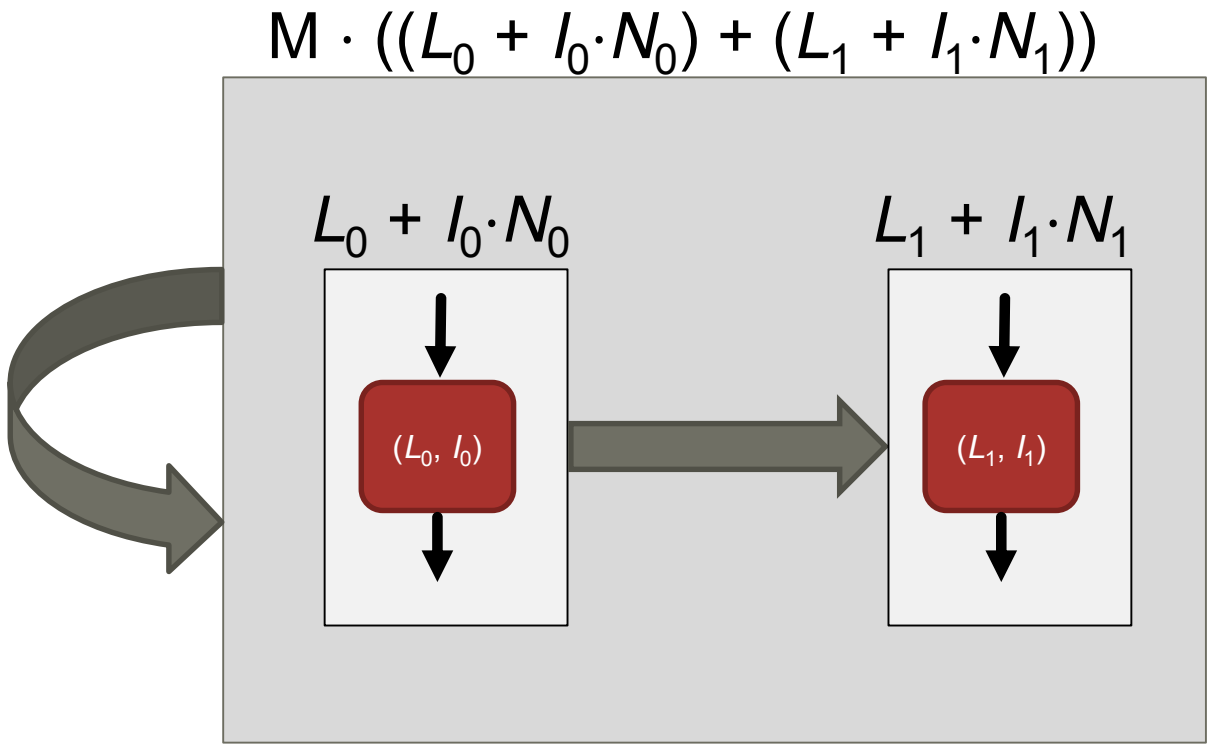
Draining and saturation phases



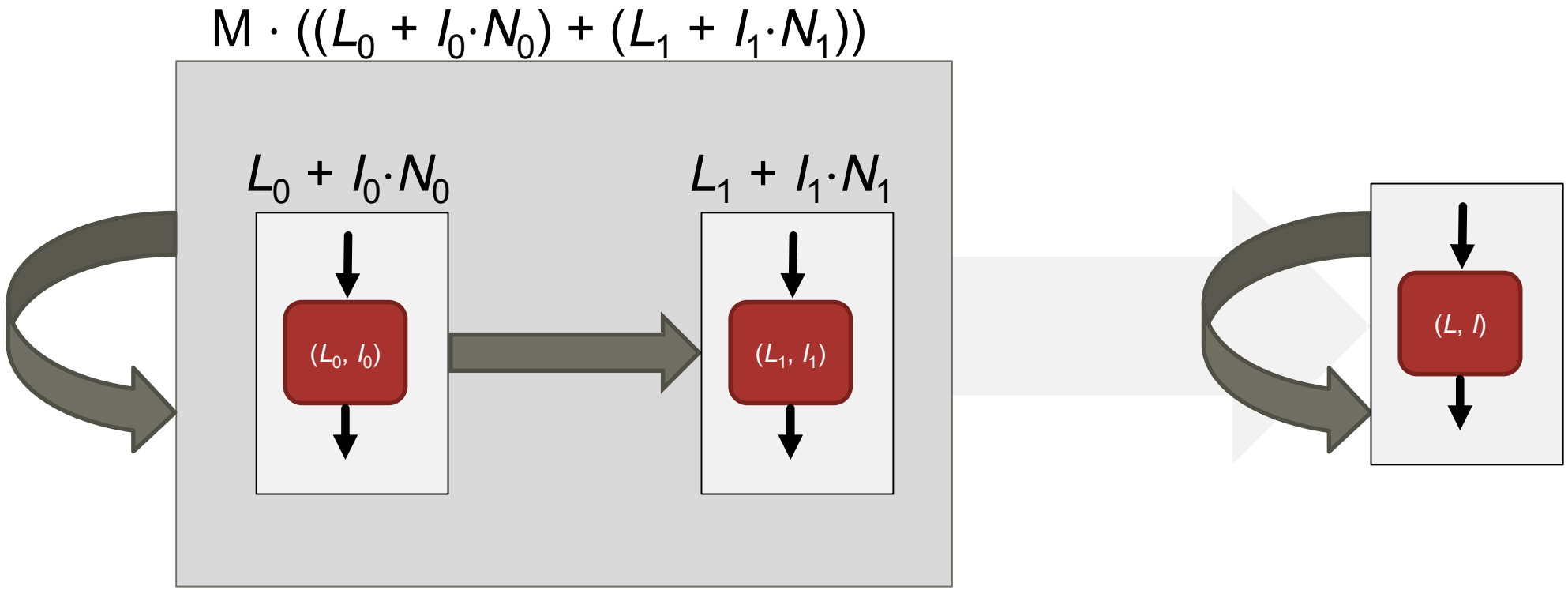
Let's see this in practice…

(example 3)

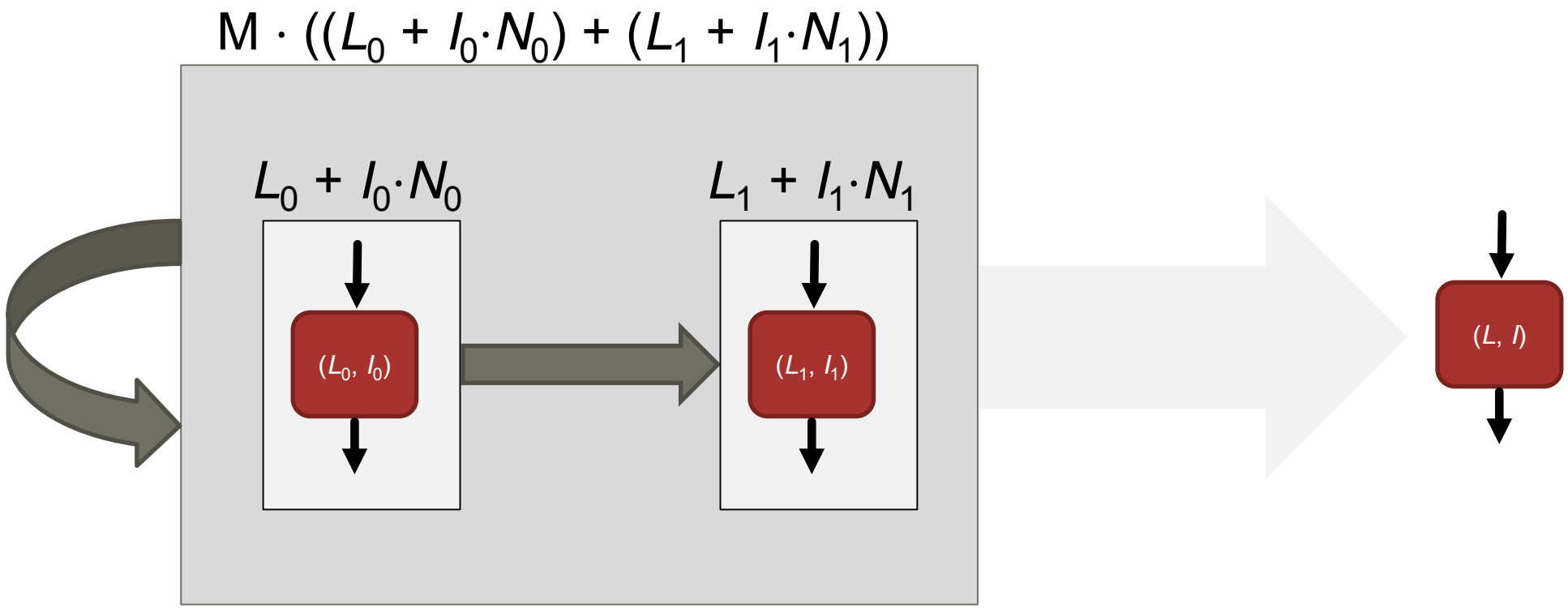
Flattening



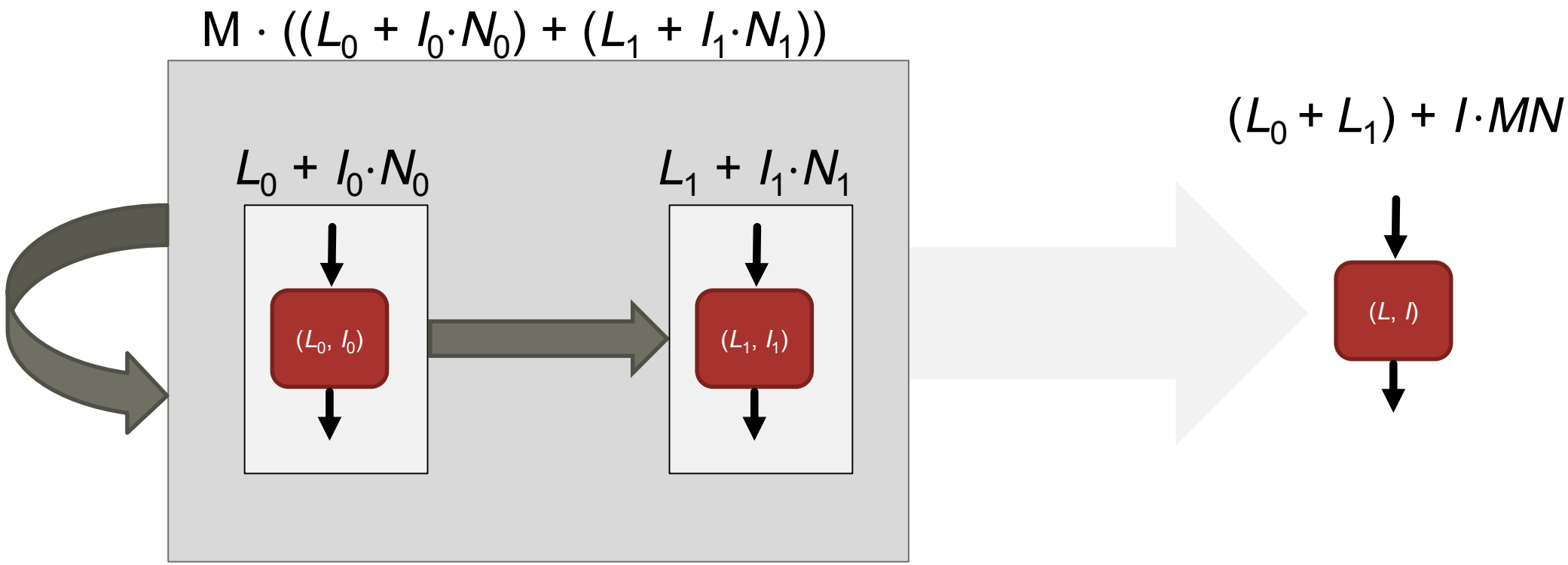
Flattening



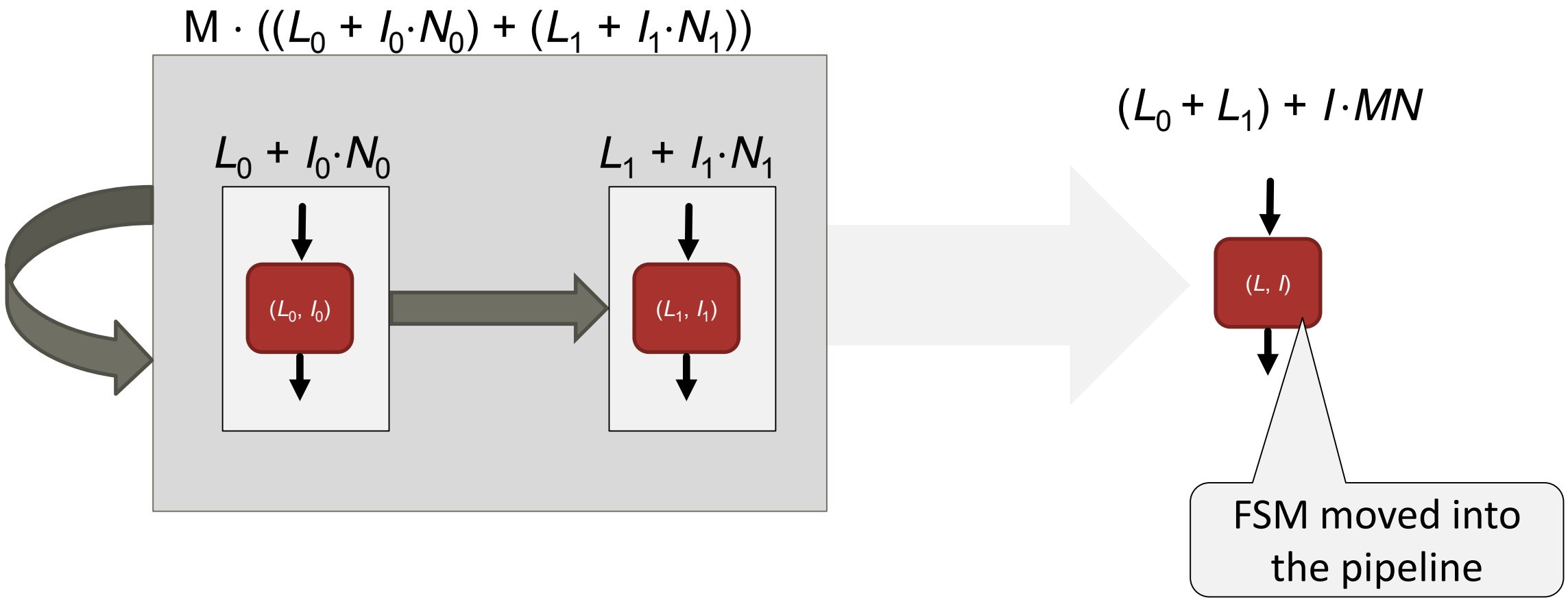
Flattening



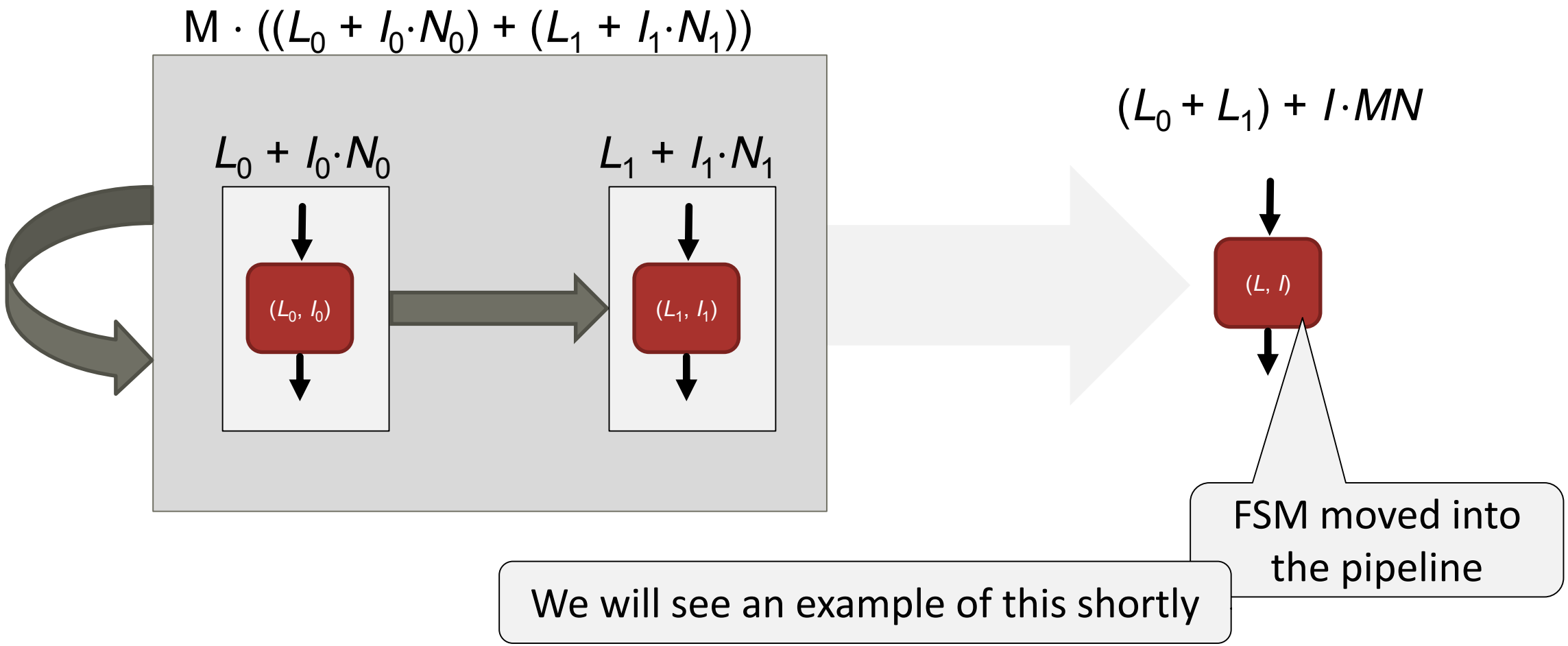
Flattening



Flattening

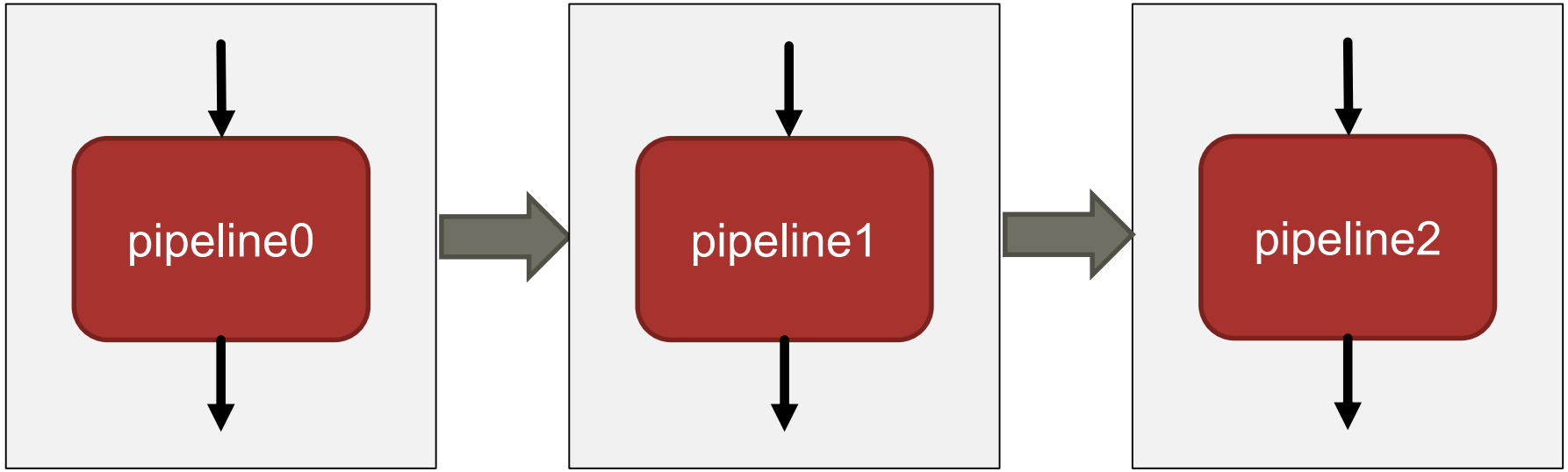

M · ((L0 + l0·N0) + (L1 + l1·N1))
L0 + l0·N0
L1 + l1·N1
(L0, l0)
(L1, l1)
(L0 + L1) + l·MN
(L, l)
FSM moved into the pipeline

Flattening


M · ((L0 + I0·N0) + (L1 + I1·N1))
L0 + I0·N0
L1 + I1·N1
(L0, I0)
(L1, I1)
(L0 + L1) + I·MN
(L, I)
FSM moved into the pipeline
We will see an example of this shortly

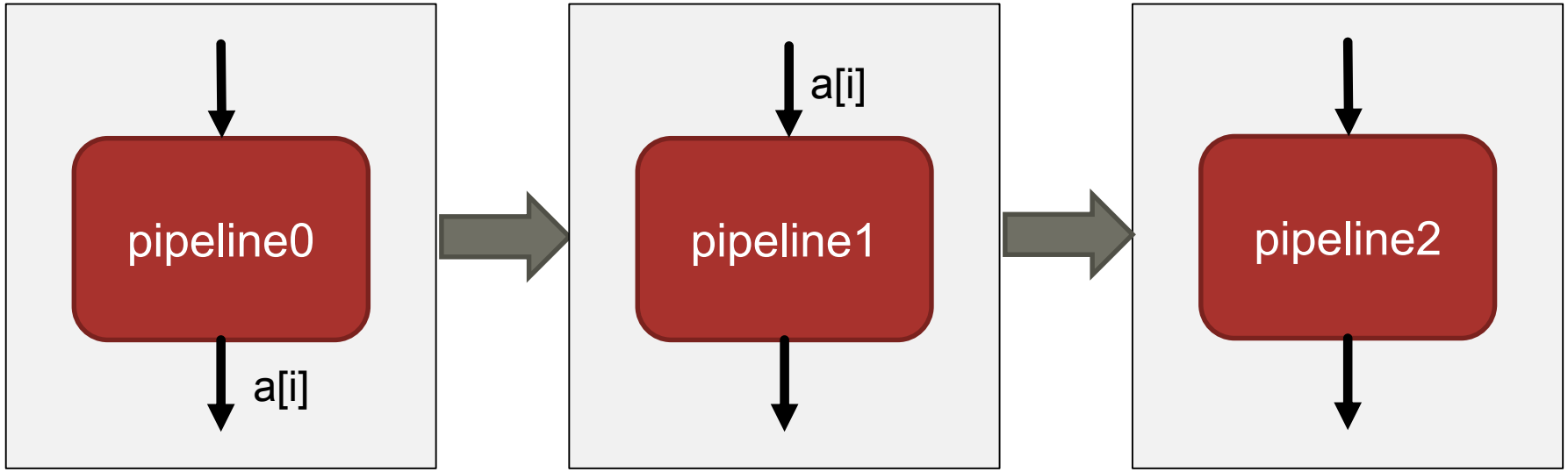
(Pipeline) parallel pipelines

$$(L_0 + l_0 \cdot N_0) + (L_1 + l_1 \cdot N_1) + (L_2 + l_2 \cdot N_2)$$



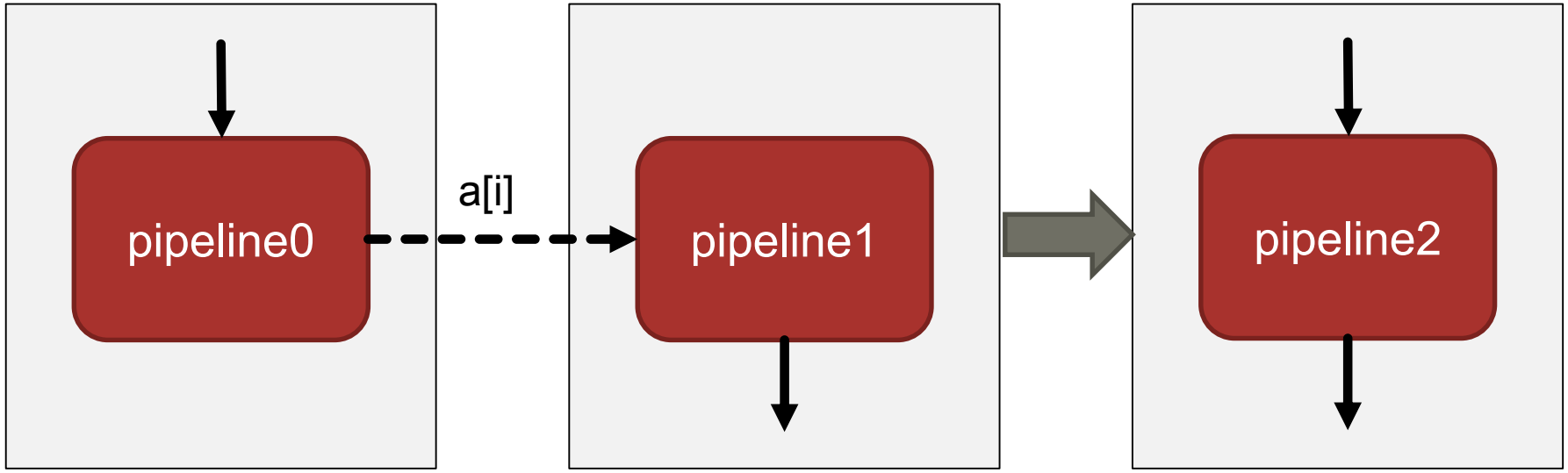
(Pipeline) parallel pipelines

$$(L_0 + I_0 \cdot N_0) + (L_1 + I_1 \cdot N_1) + (L_2 + I_2 \cdot N_2)$$



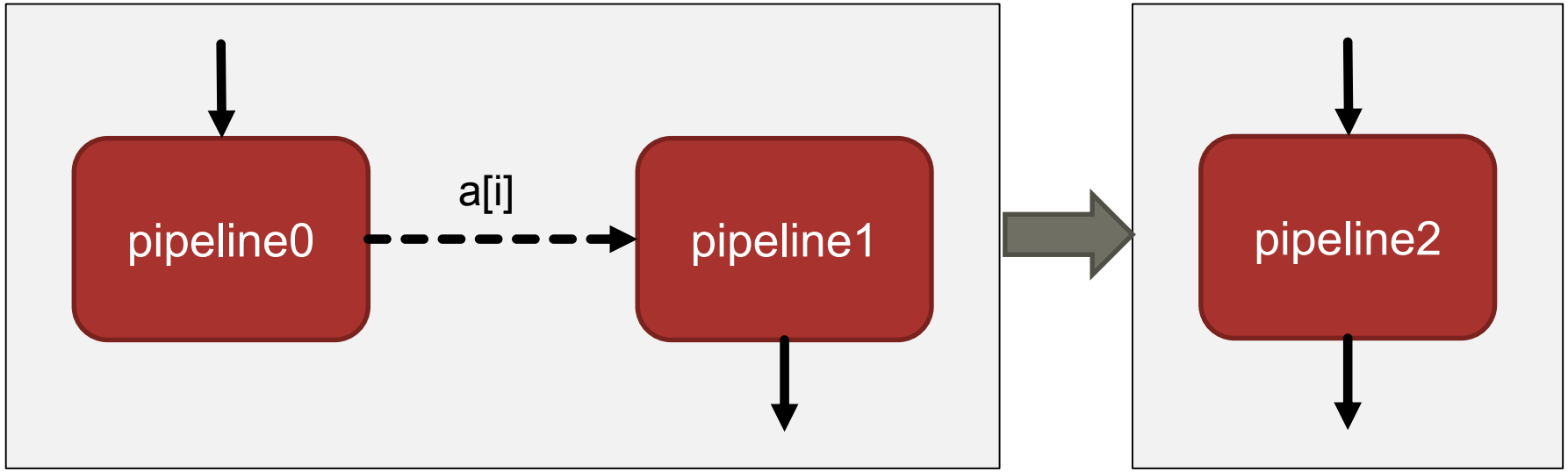
(Pipeline) parallel pipelines

$$(L_0 + I_0 \cdot N_0) + (L_1 + I_1 \cdot N_1) + (L_2 + I_2 \cdot N_2)$$



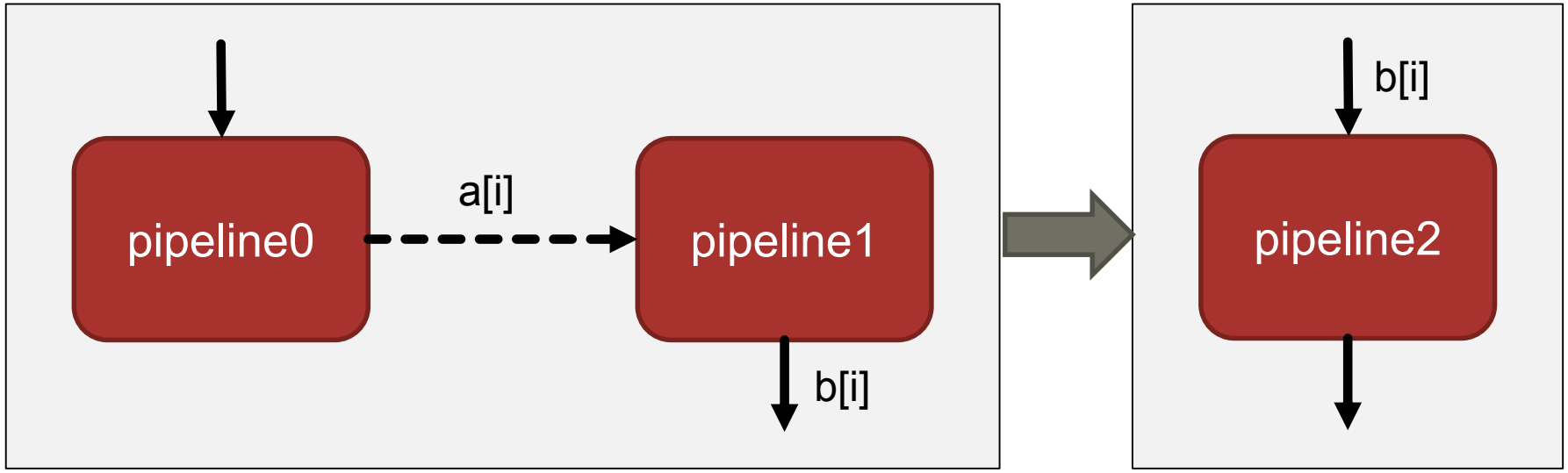
(Pipeline) parallel pipelines

$$((L_0 + L_1) + I \cdot N) + (L_2 + I_2 \cdot N_2)$$



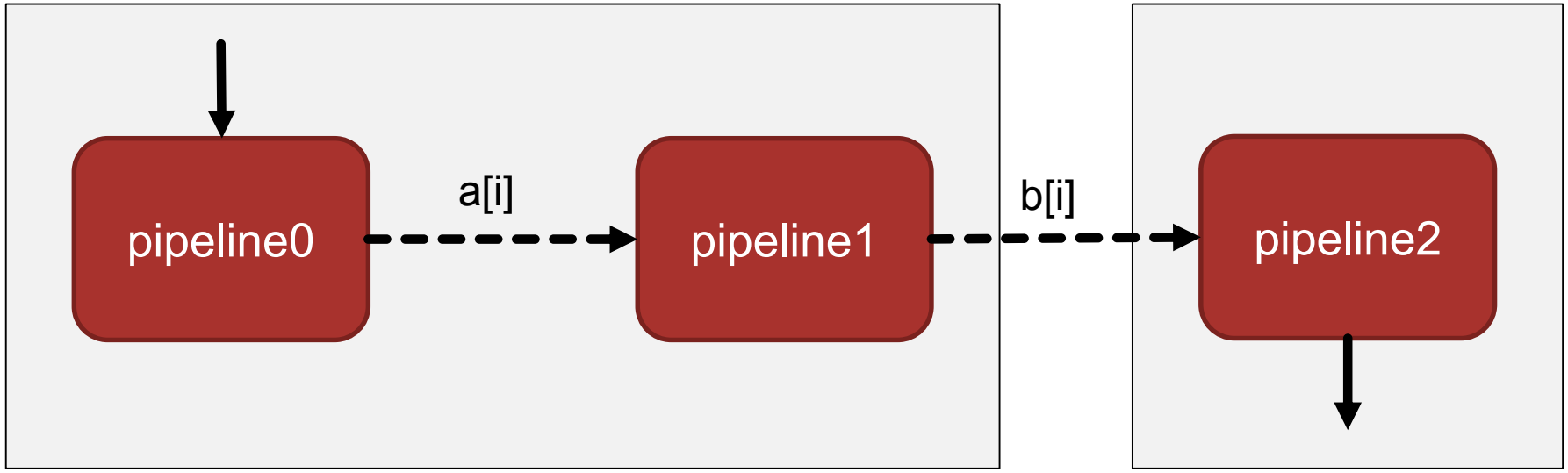
(Pipeline) parallel pipelines

$$((L_0 + L_1) + I \cdot N) + (L_2 + I_2 \cdot N_2)$$



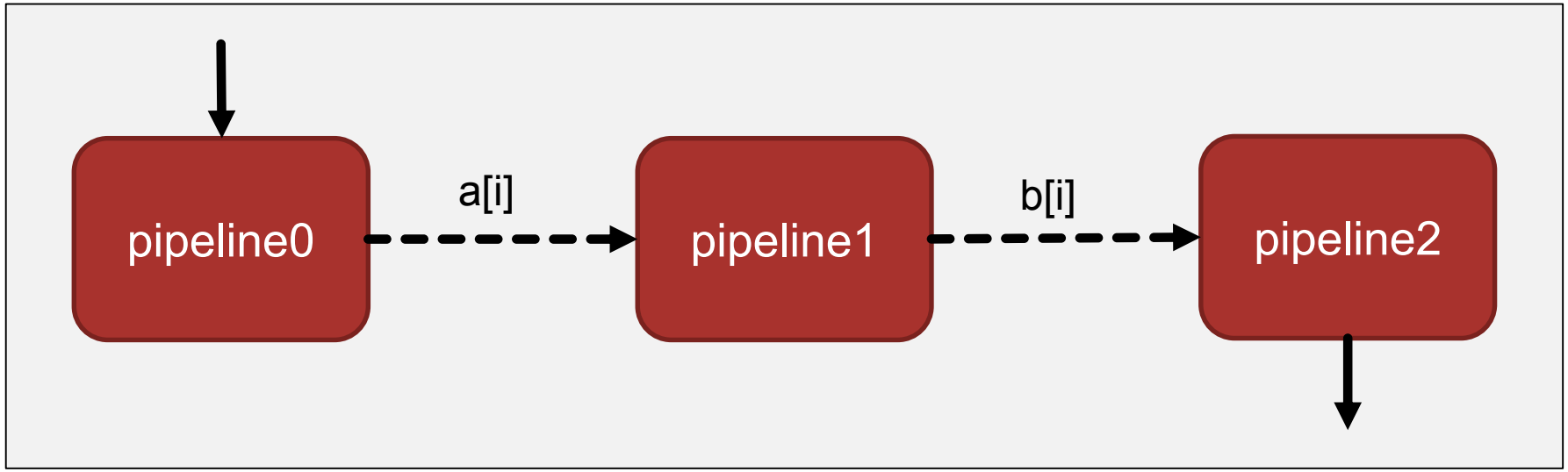
(Pipeline) parallel pipelines

$$((L_0 + L_1) + I \cdot N) + (L_2 + I_2 \cdot N_2)$$



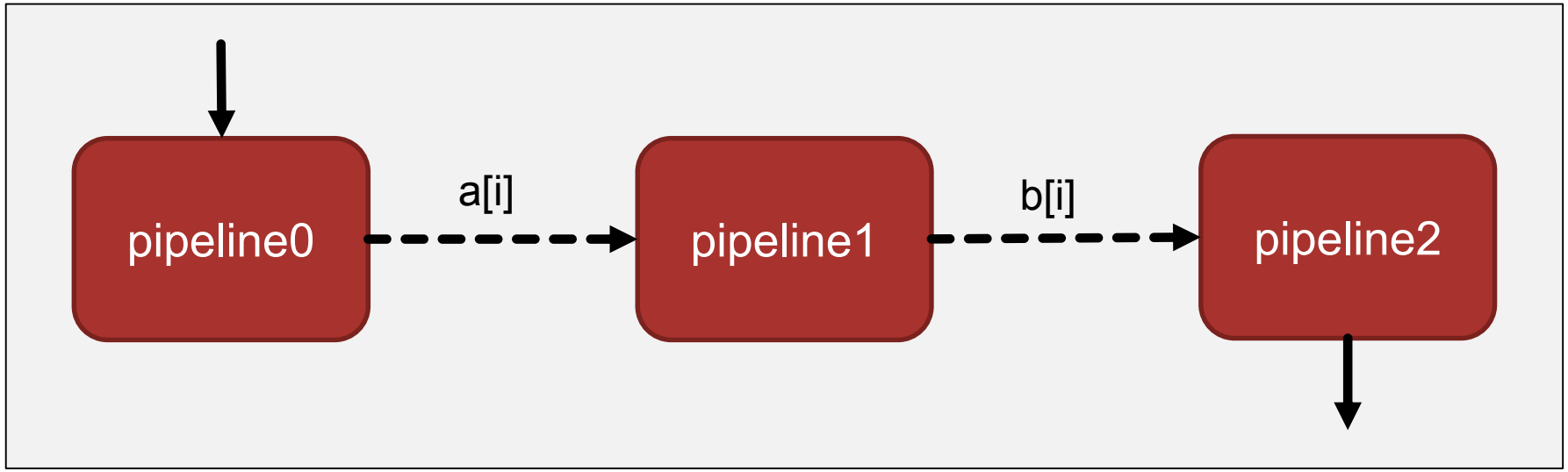
(Pipeline) parallel pipelines

$$(L_0 + L_1 + L_2) + I \cdot N$$

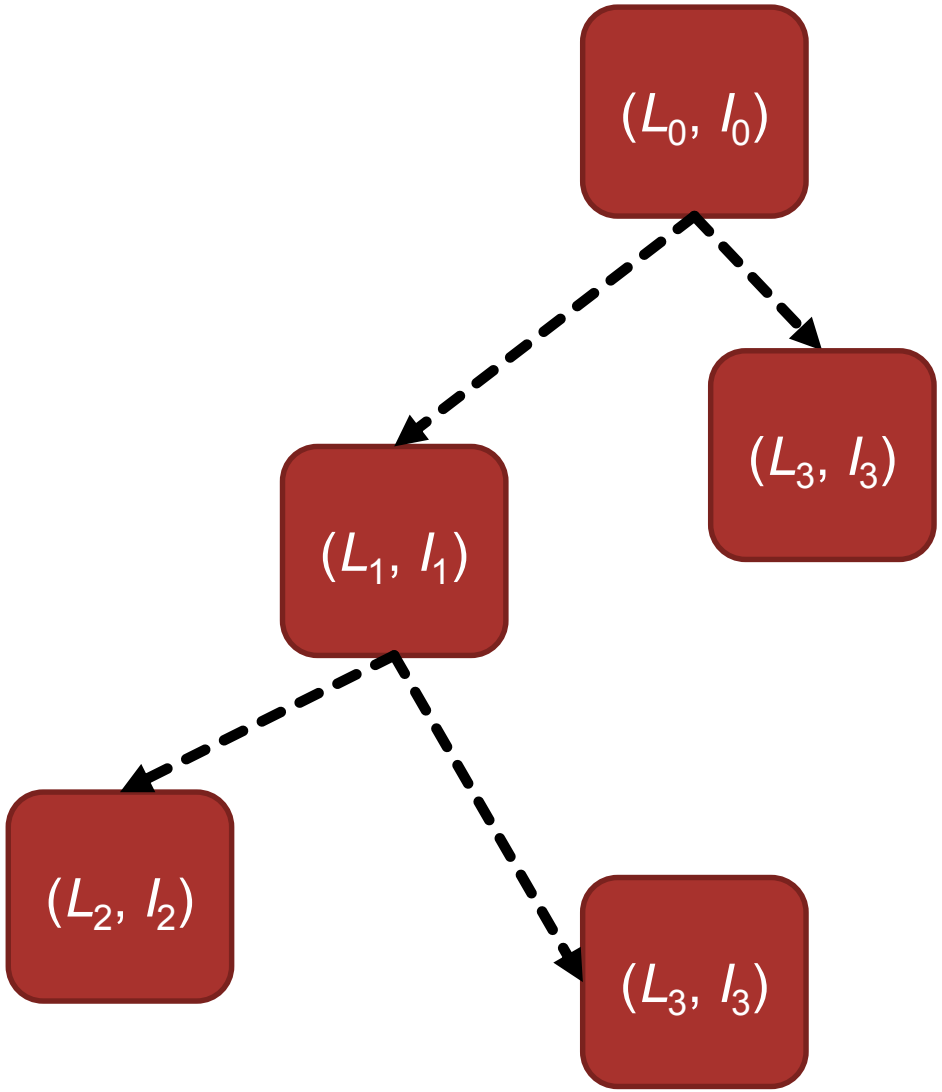


(Pipeline) parallel pipelines

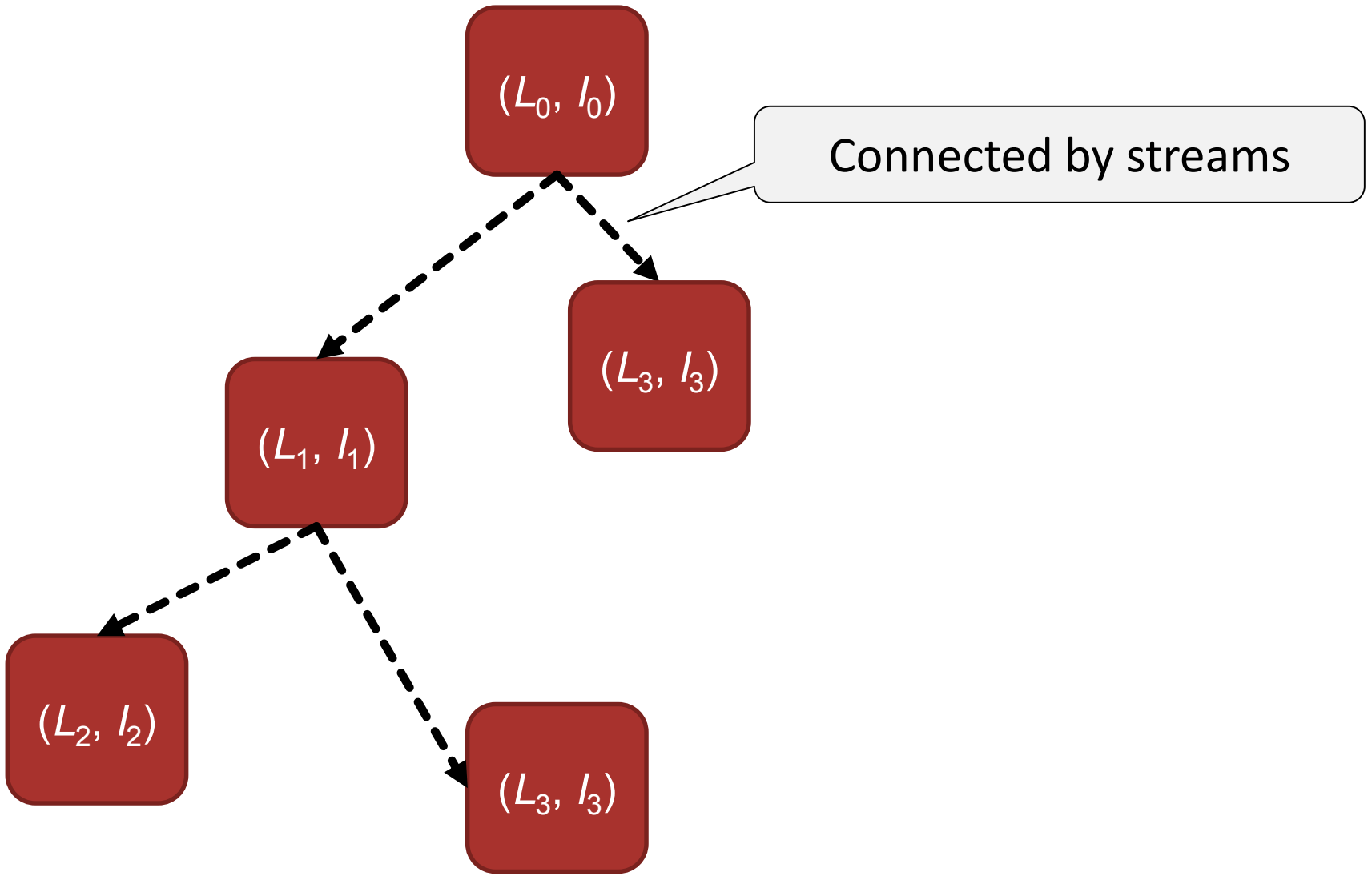
$$(L_0 + L_1 + L_2) + I \cdot N$$



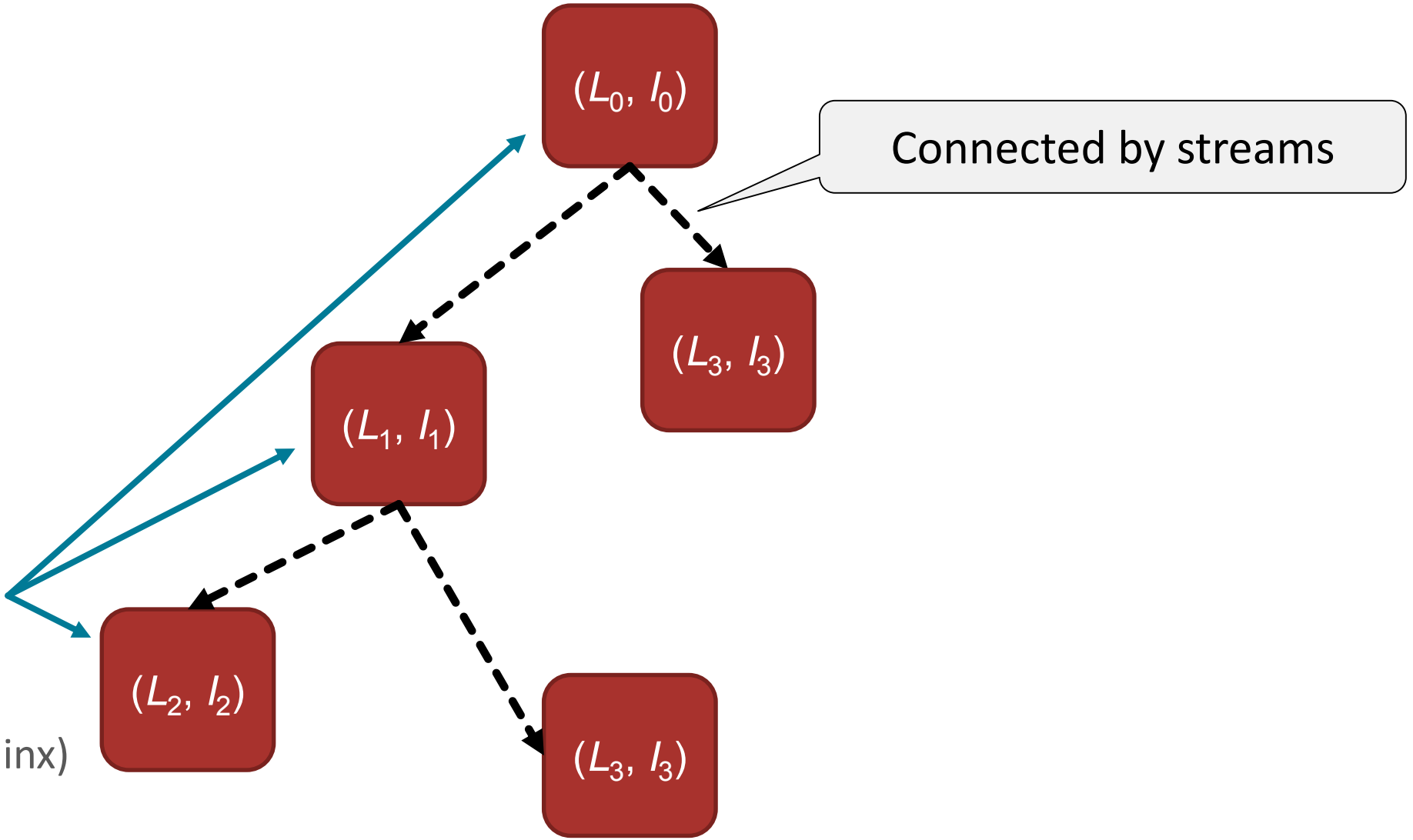
Processing elements



Processing elements



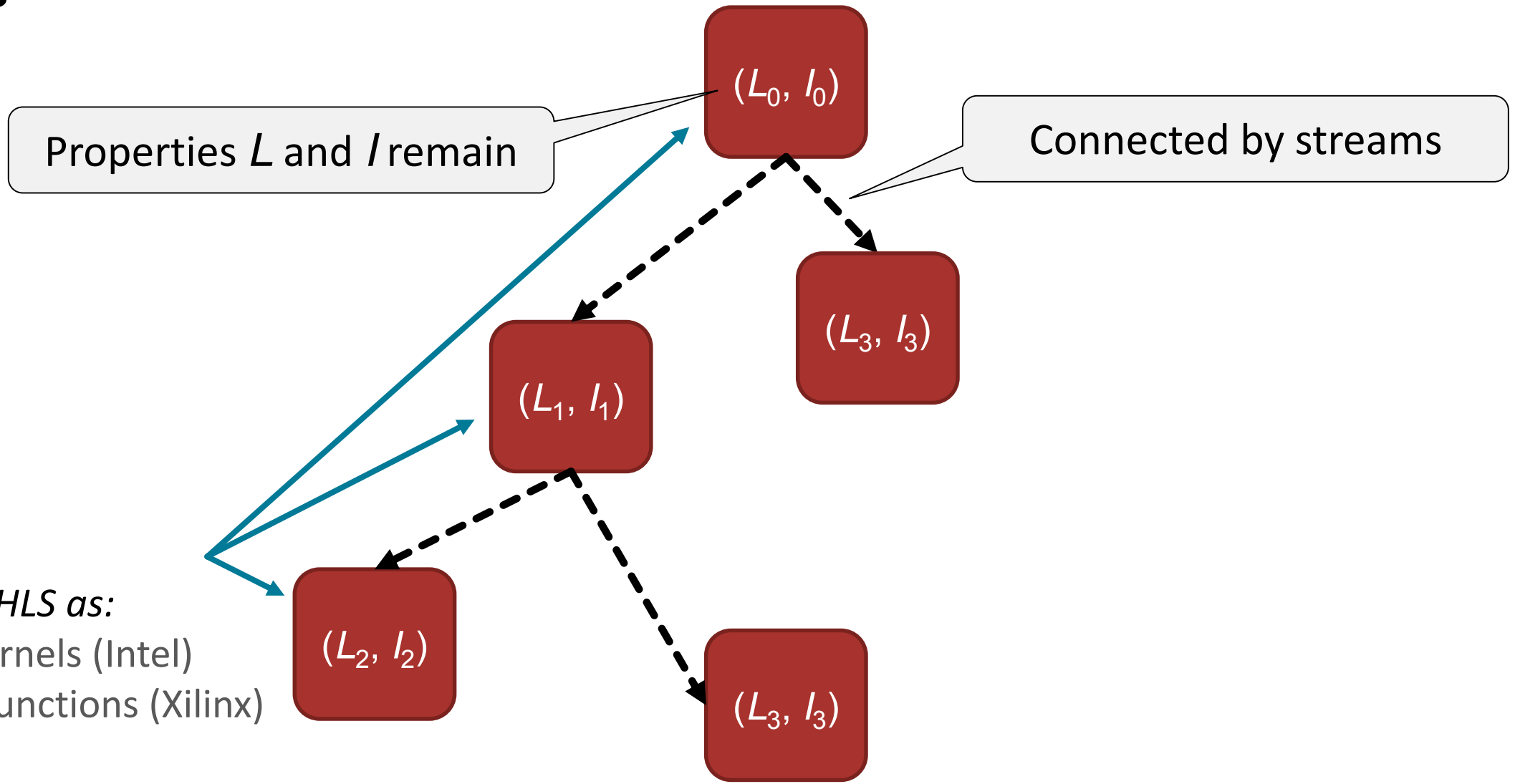
Processing elements



Expressed in HLS as:

- OpenCL kernels (Intel)
- Top-level functions (Xilinx)

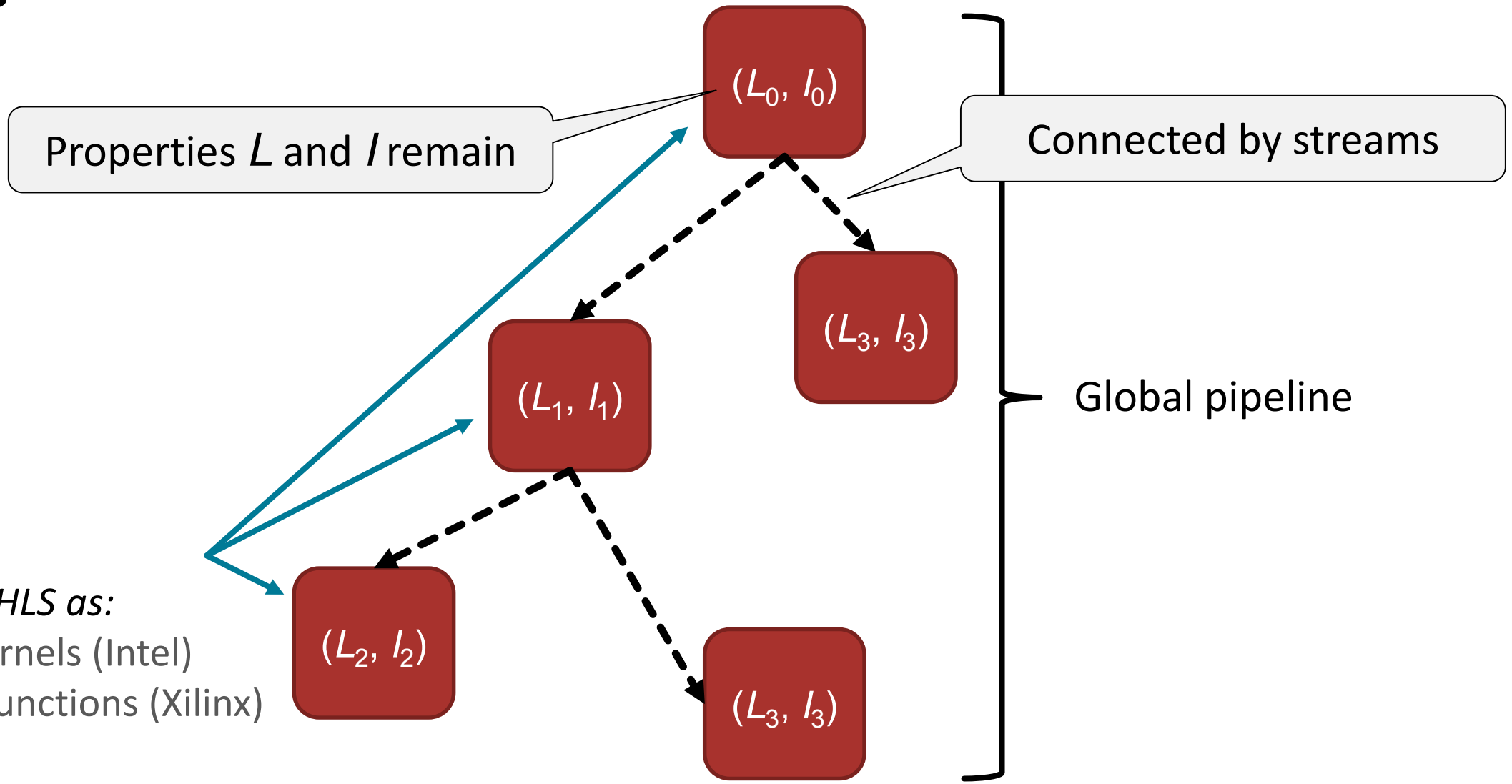
Processing elements



Expressed in HLS as:

- OpenCL kernels (Intel)
- Top-level functions (Xilinx)

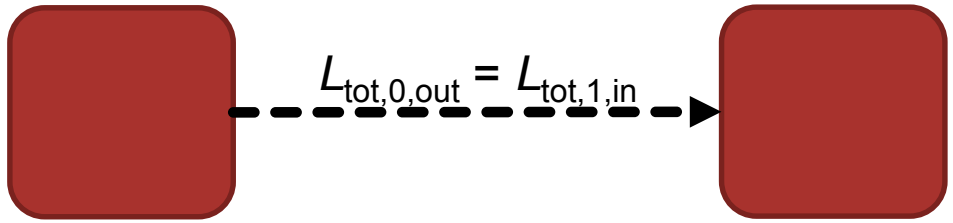
Processing elements



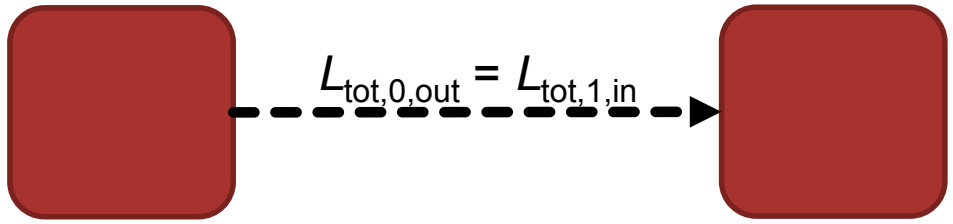
Expressed in HLS as:

- OpenCL kernels (Intel)
- Top-level functions (Xilinx)

Properties of the global pipeline



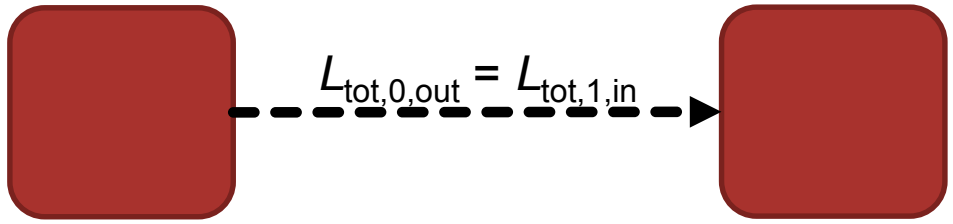
Properties of the global pipeline



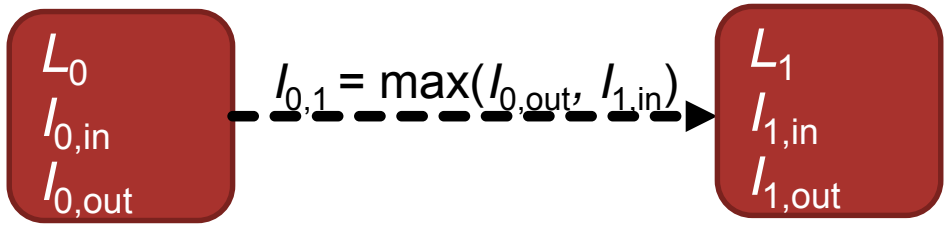
What goes in must come out:

Every stream write needs a corresponding read

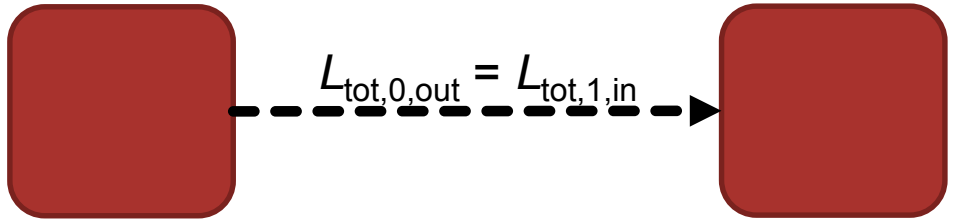
Properties of the global pipeline



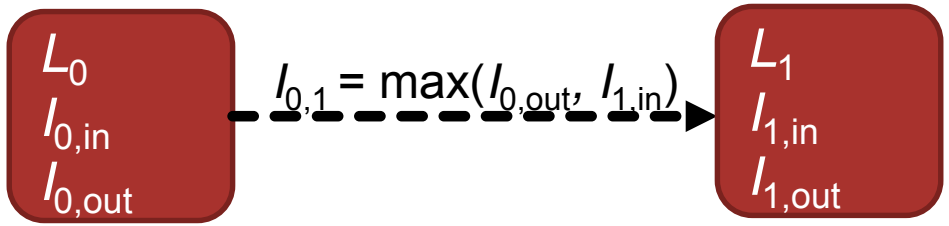
What goes in must come out:
Every stream write needs a corresponding read



Properties of the global pipeline

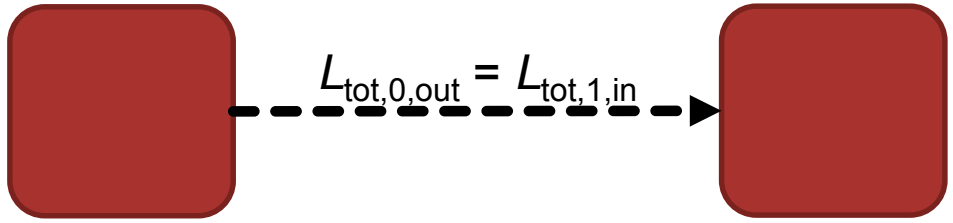


What goes in must come out:
Every stream write needs a corresponding read

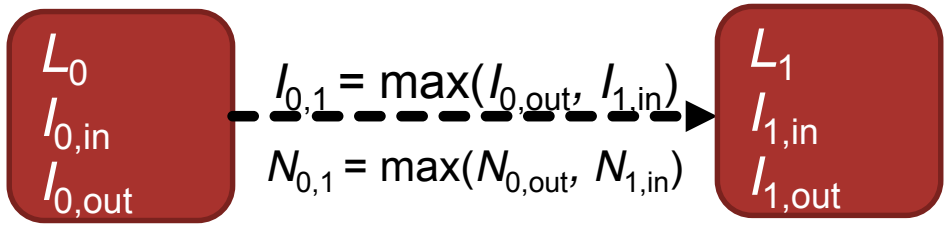


Slow dominates:
Can only stream with the highest initiation interval (producer/consumer) at each interface

Properties of the global pipeline

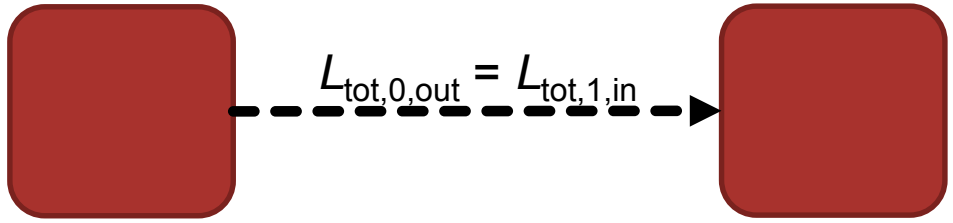


What goes in must come out:
Every stream write needs a corresponding read

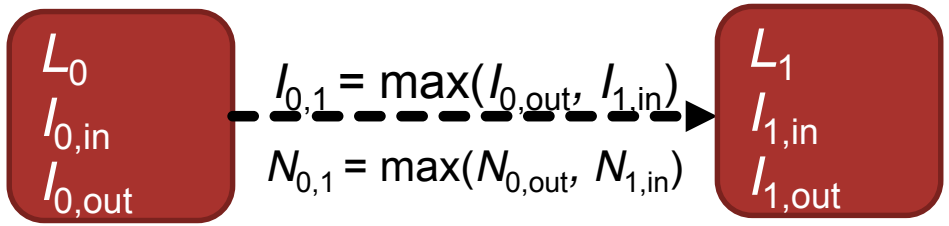


Slow dominates:
Can only stream with the highest initiation interval (producer/consumer) at each interface

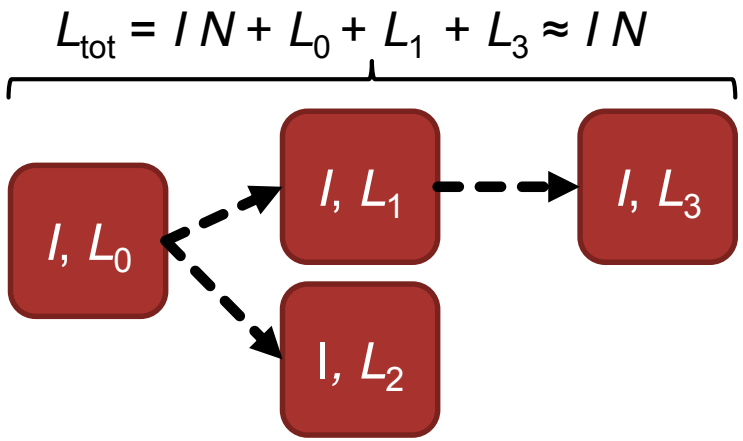
Properties of the global pipeline



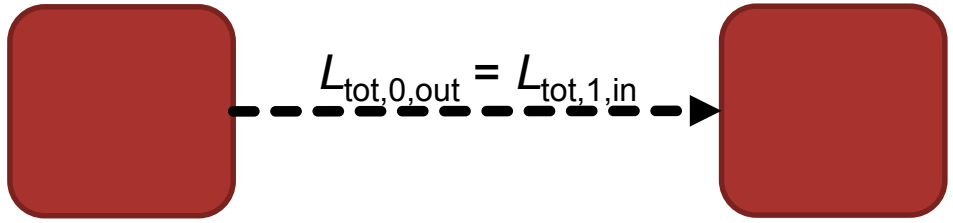
What goes in must come out:
Every stream write needs a corresponding read



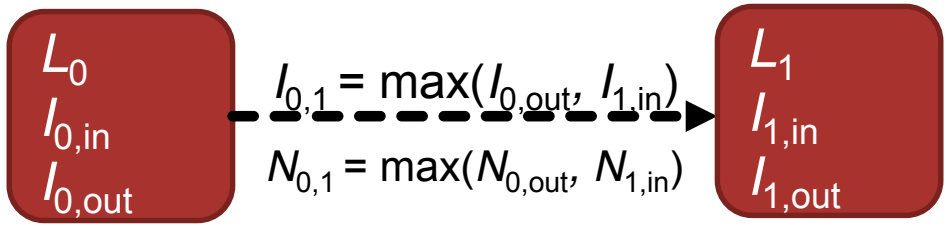
Slow dominates:
Can only stream with the highest initiation interval (producer/consumer) at each interface



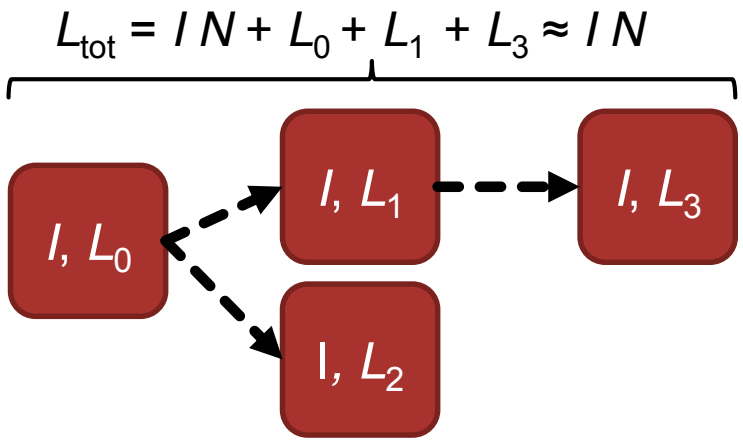
Properties of the global pipeline



What goes in must come out:
Every stream write needs a corresponding read



Slow dominates:
Can only stream with the highest initiation interval (producer/consumer) at each interface



Depth is "free":
In a perfect pipeline for large N, the influence of pipeline latency is negligible w.r.t. the total runtime

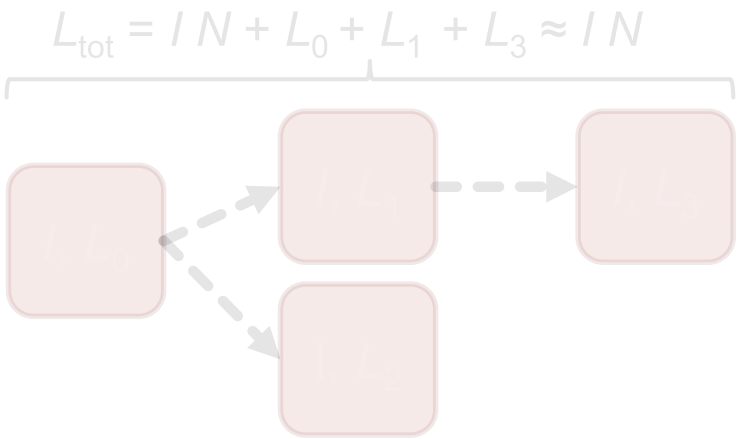
Properties of the global pipeline

What goes in must come out:
Every stream write needs a corresponding read

Sync dimensions:
Can only stream with the highest initiation interval (producer/consumer) at each interface

Let's see this in practice…

(example 4)



Depth is "free":
In a perfect pipeline for large N, the influence of pipeline latency is negligible w.r.t. the total runtime

Unrolling

Much stronger meaning on FPGA than for an instruction-based architecture.

Unrolling

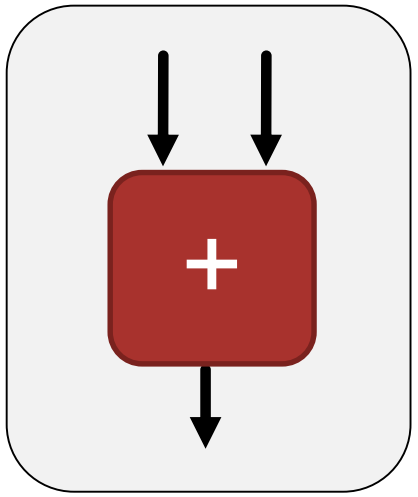
Much stronger meaning on FPGA than for an instruction-based architecture.

```
for (int w = 0; w < 4; ++w) {
  #pragma HLS PIPELINE II=1
  res[w] = a[w] + b[w];
}
```

Unrolling

Much stronger meaning on FPGA than for an instruction-based architecture.

```
for (int w = 0; w < 4; ++w) {
  #pragma HLS PIPELINE II=1
  res[w] = a[w] + b[w];
}
```

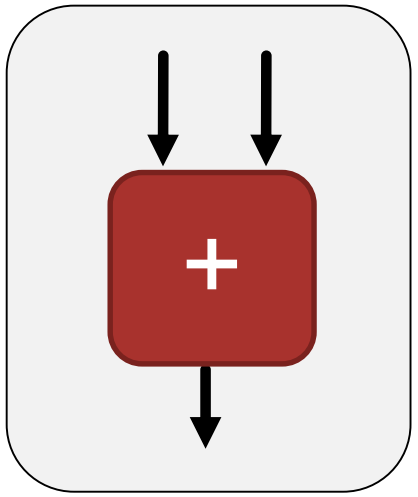


Unrolling

Much stronger meaning on FPGA than for an instruction-based architecture.

```
for (int w = 0; w < 4; ++w) {
  #pragma HLS PIPELINE II=1
  res[w] = a[w] + b[w];
}
```

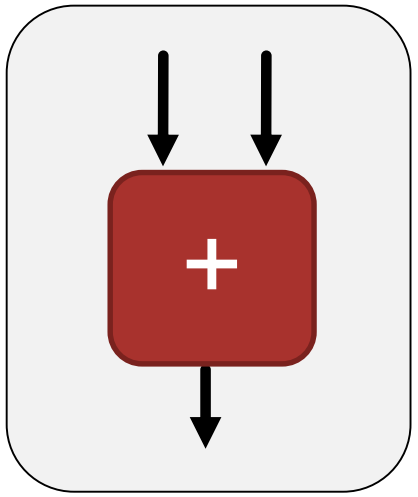
```
#pragma HLS PIPELINE II=1
for (int w = 0; w < 4; ++w) {
  #pragma HLS UNROLL
  res[w] = a[w] + b[w];
}
```



Unrolling

Much stronger meaning on FPGA than for an instruction-based architecture.

```
for (int w = 0; w < 4; ++w) {
  #pragma HLS PIPELINE II=1
  res[w] = a[w] + b[w];
}
```

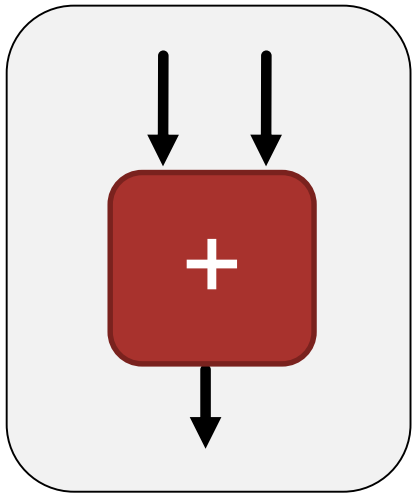



```
#pragma HLS PIPELINE II=1
res[0] = a[0] + b[0];
res[1] = a[1] + b[1];
res[2] = a[2] + b[2];
res[3] = a[3] + b[3];
```

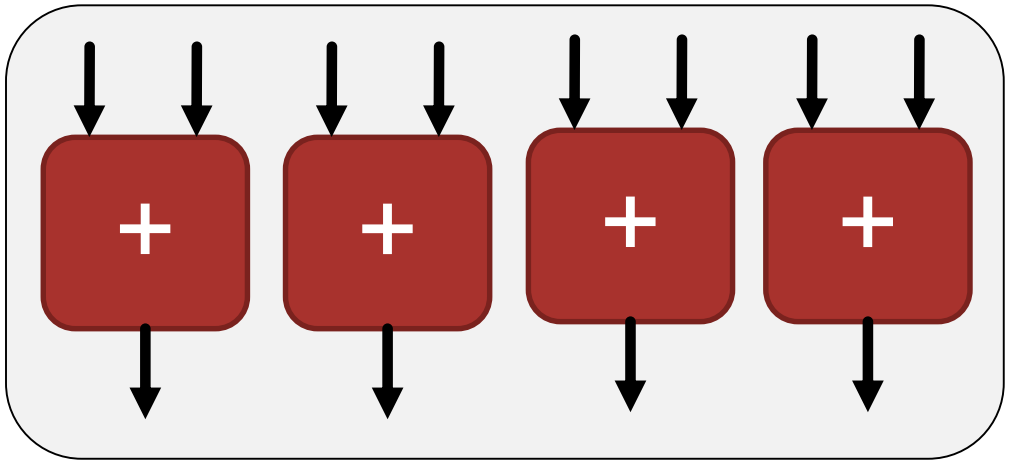
Unrolling

Much stronger meaning on FPGA than for an instruction-based architecture.

```
for (int w = 0; w < 4; ++w) {
  #pragma HLS PIPELINE II=1
  res[w] = a[w] + b[w];
}
```



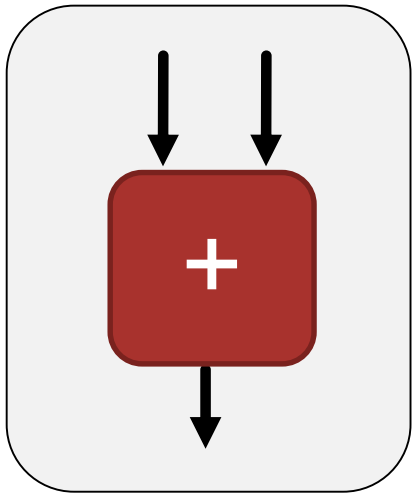

```
#pragma HLS PIPELINE II=1
res[0] = a[0] + b[0];
res[1] = a[1] + b[1];
res[2] = a[2] + b[2];
res[3] = a[3] + b[3];
```



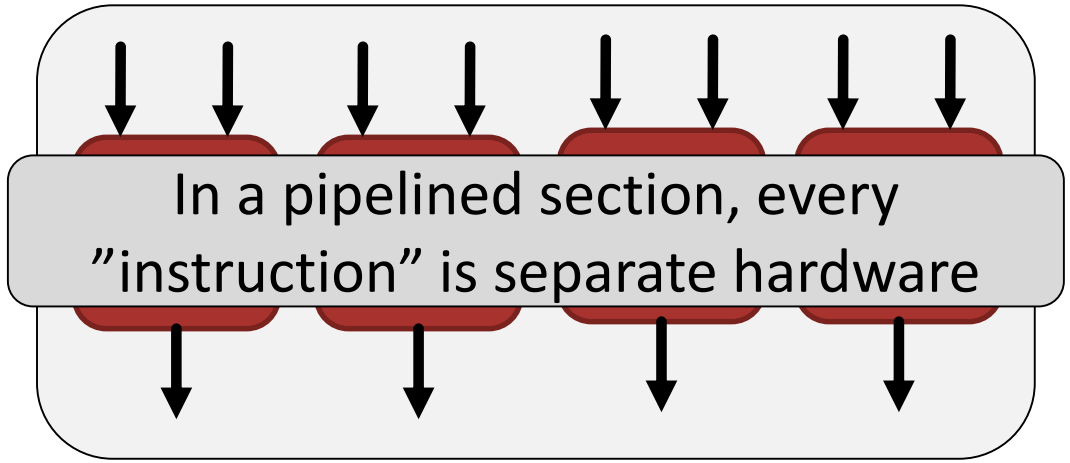
Unrolling

Much stronger meaning on FPGA than for an instruction-based architecture.

```
for (int w = 0; w < 4; ++w) {
  #pragma HLS PIPELINE II=1
  res[w] = a[w] + b[w];
}
```




```
#pragma HLS PIPELINE II=1
res[0] = a[0] + b[0];
res[1] = a[1] + b[1];
res[2] = a[2] + b[2];
res[3] = a[3] + b[3];
```

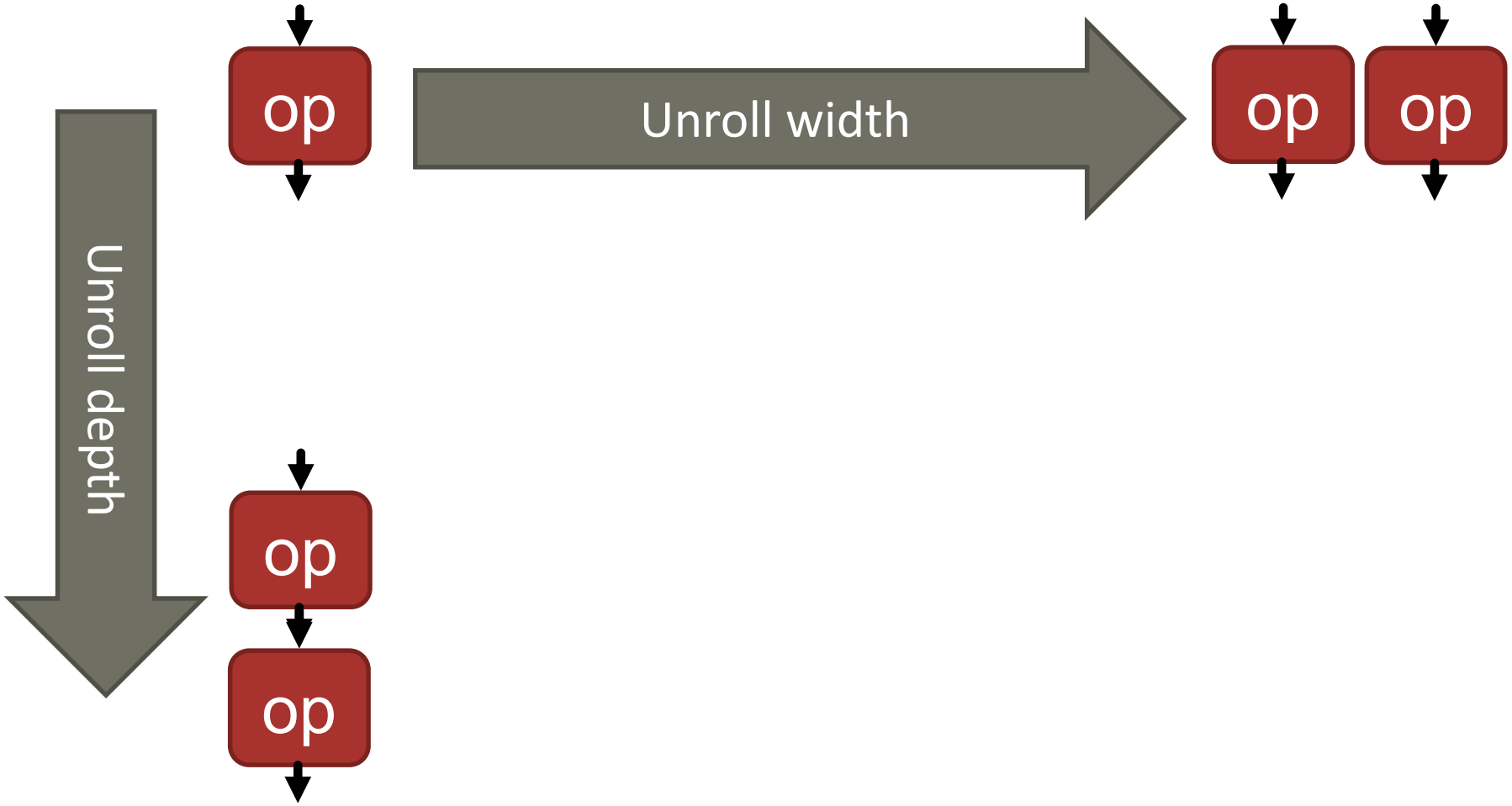


Unrolling is scaling

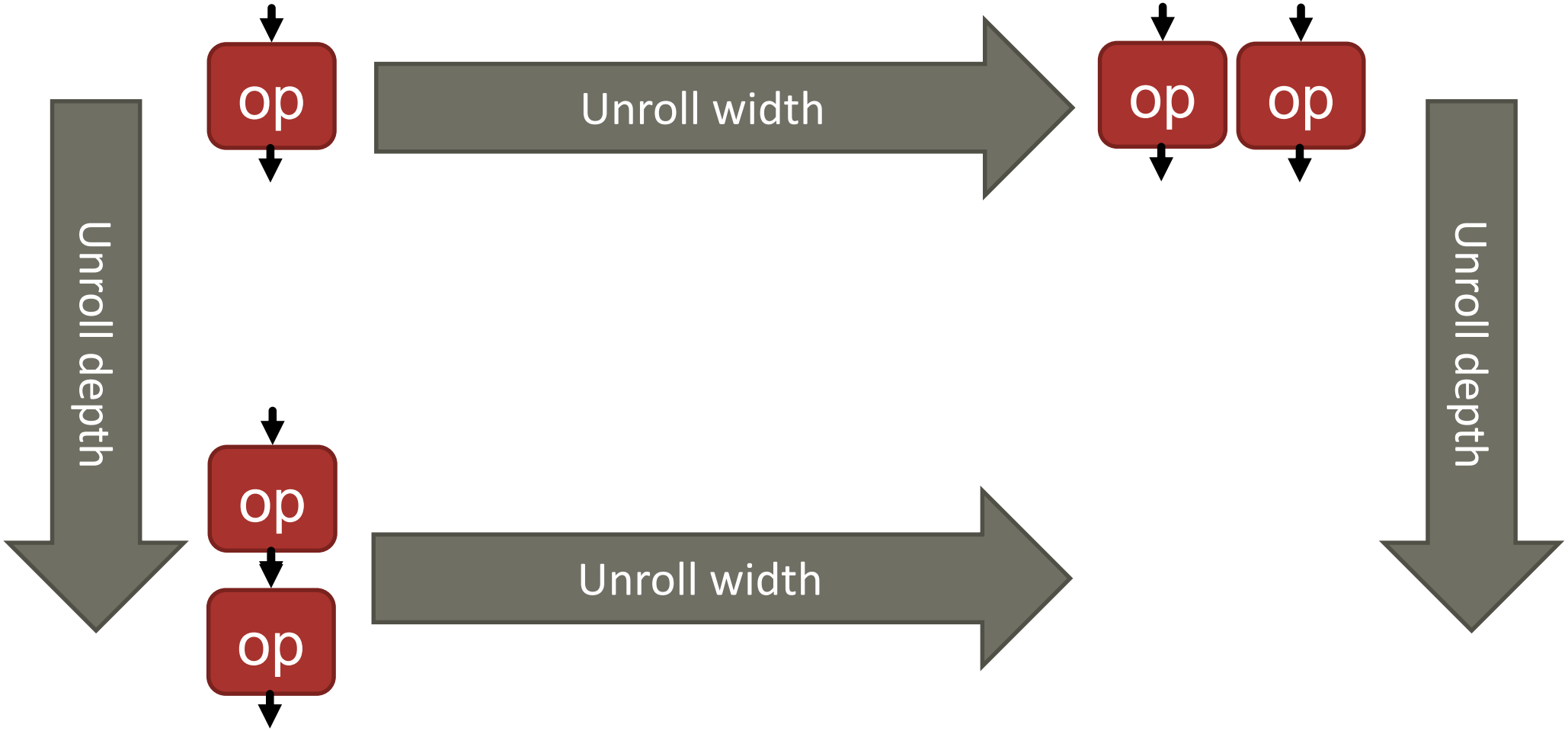
Unrolling is scaling

Unrolling is scaling

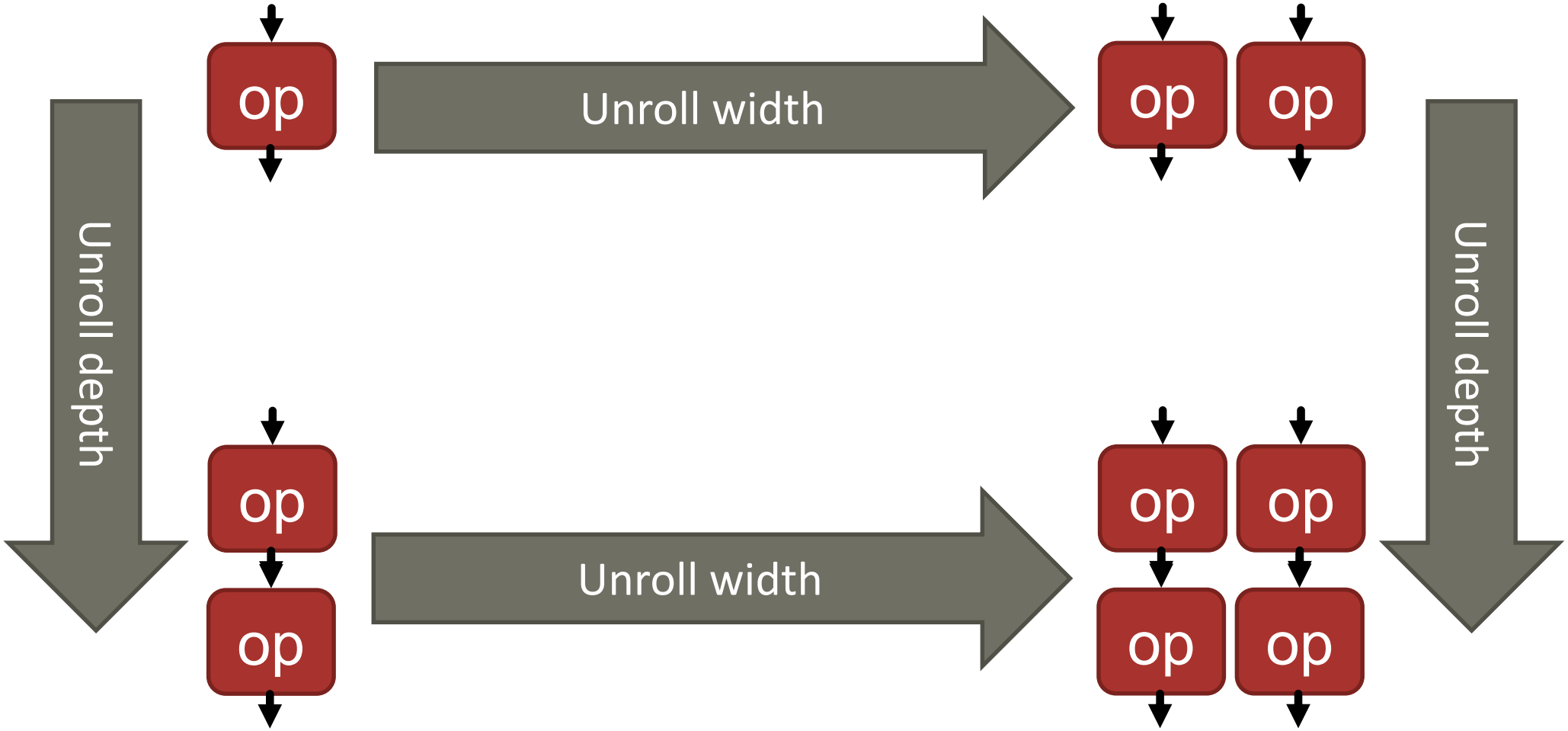
Unrolling is scaling



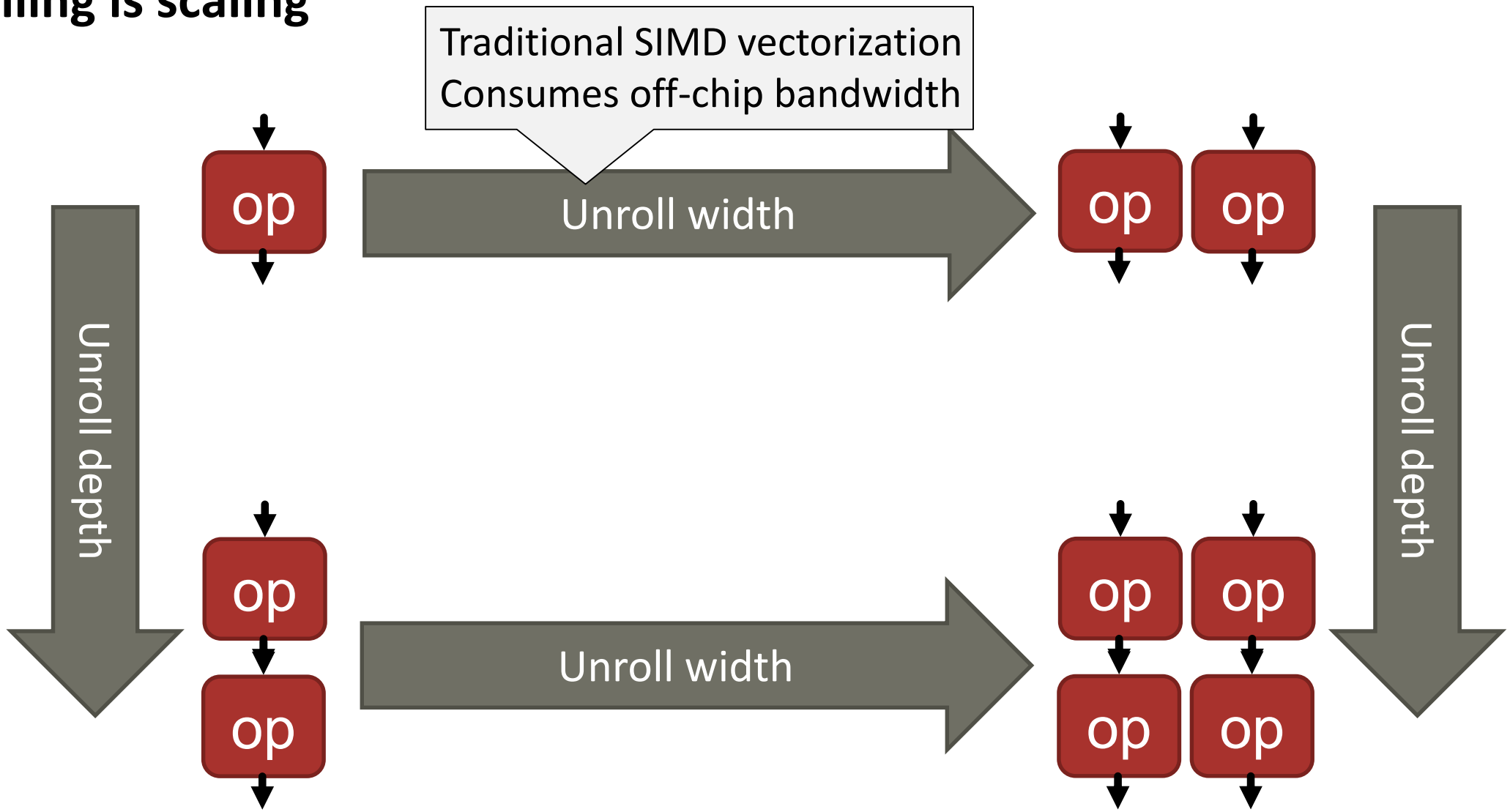
Unrolling is scaling



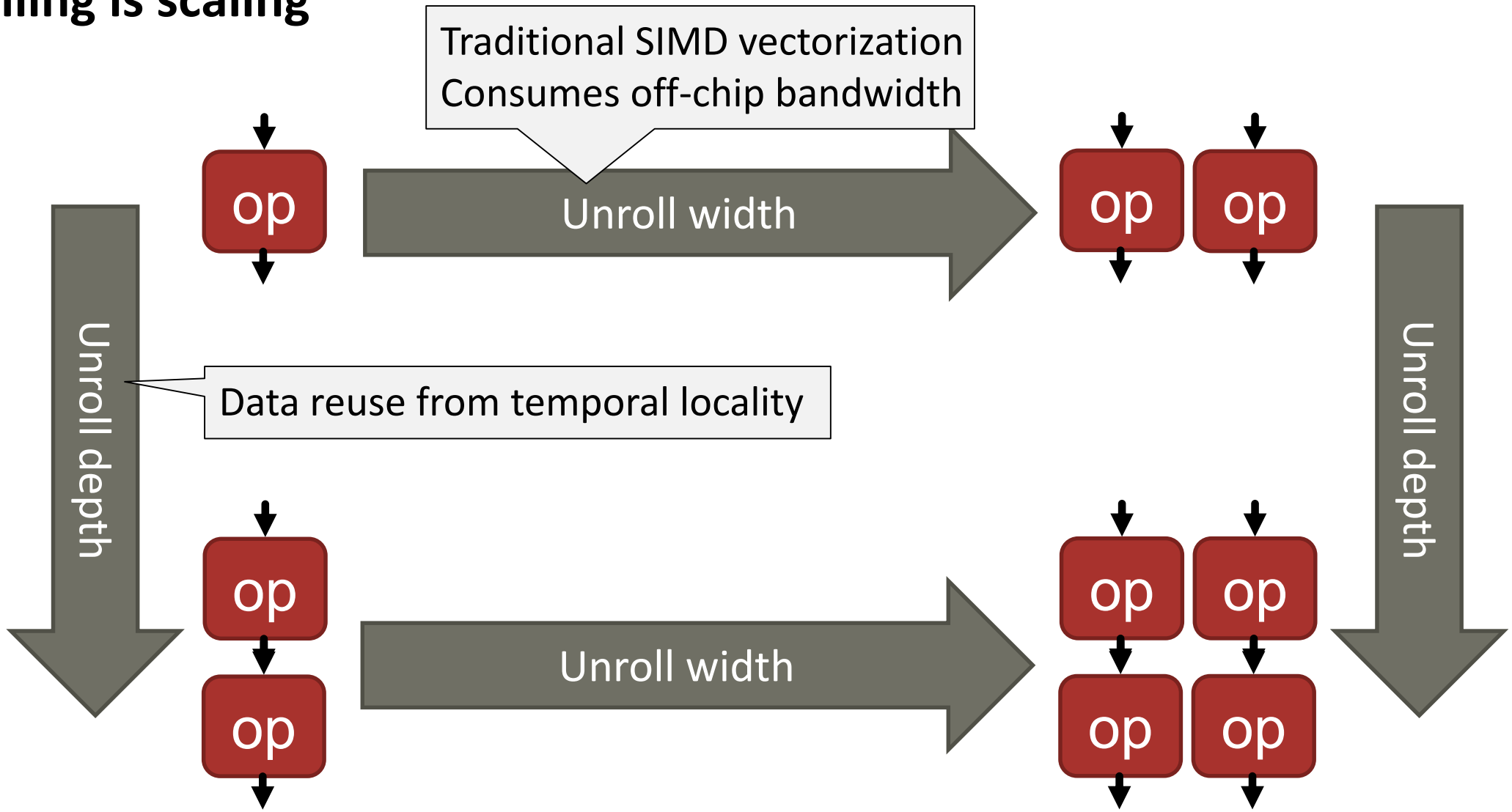
Unrolling is scaling



Unrolling is scaling

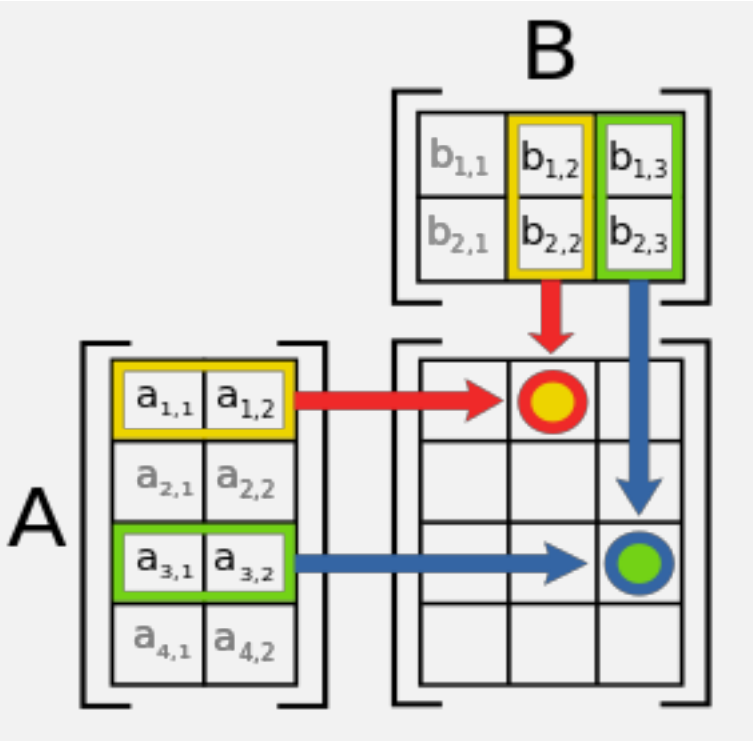


Unrolling is scaling



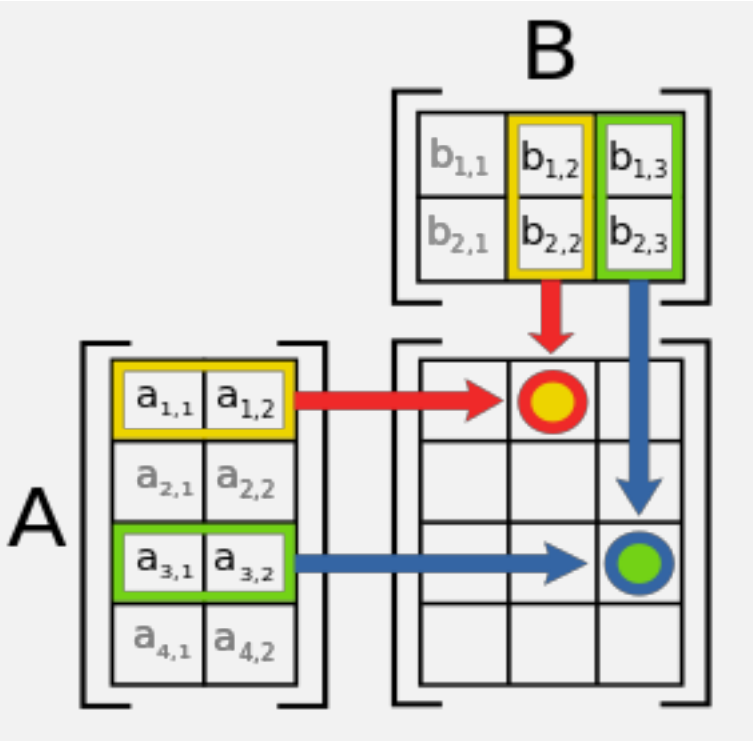
Matrix-matrix multiplication

```
for (int n = 0; n < N; ++n)
  for (int p = 0; p < P; ++p)
    for (int m = 0; m < M; ++m)
      C[n, p] += A[n, m] * B[m, p];
```



Matrix-matrix multiplication

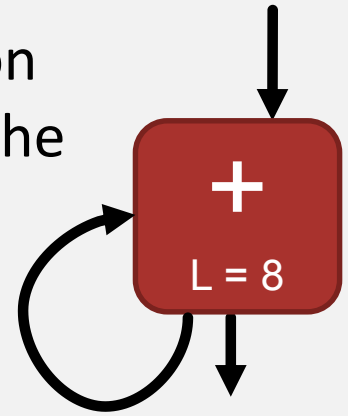
```
for (int n = 0; n < N; ++n)
  for (int p = 0; p < P; ++p)
    for (int m = 0; m < M; ++m)
      C[n, p] += A[n, m] * B[m, p];
```

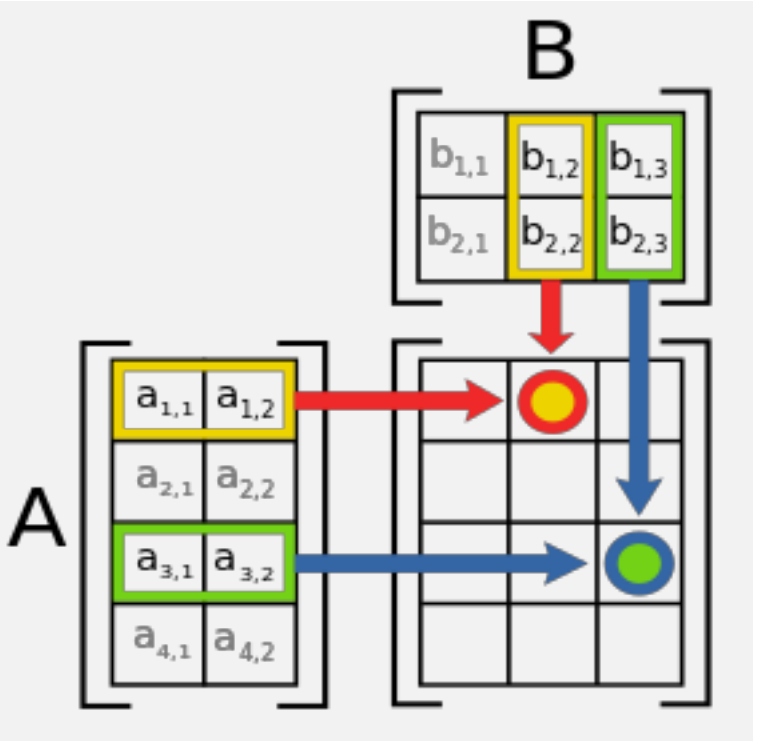


Matrix-matrix multiplication

```
for (int n = 0; n < N; ++n)
  for (int p = 0; p < P; ++p)
    for (int m = 0; m < M; ++m)
      C[n, p] += A[n, m] * B[m, p];
```

Every iteration depends on the previous:



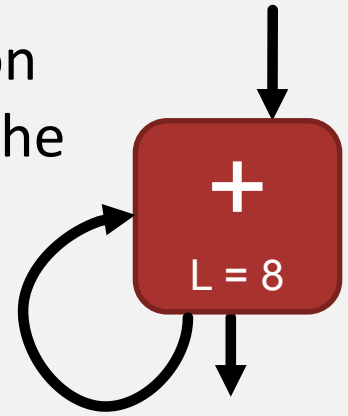


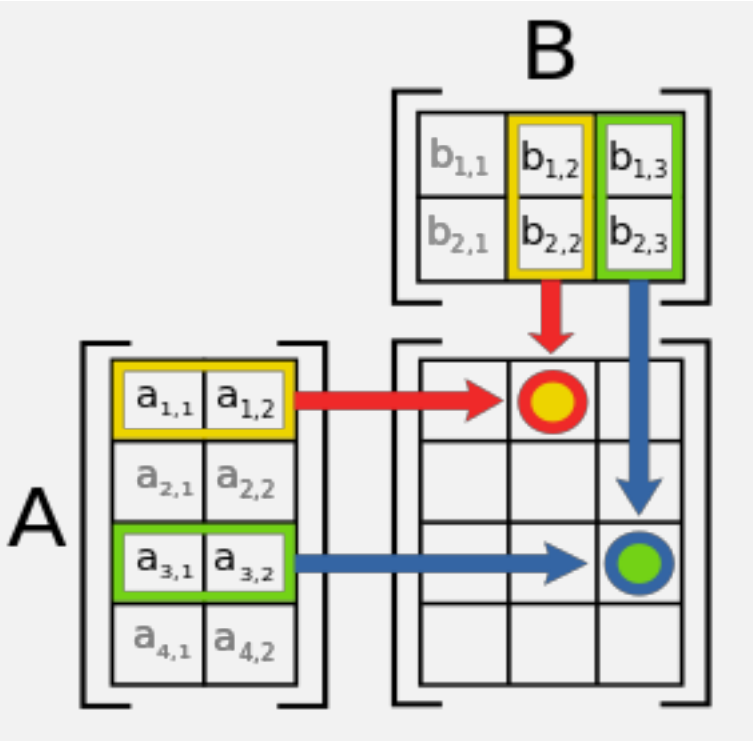
Matrix-matrix multiplication

```
for (int n = 0; n < N; ++n)
  for (int p = 0; p < P; ++p)
    for (int m = 0; m < M; ++m)
      C[n, p] += A[n, m] * B[m, p];
```

Every iteration depends on the previous:

$I = 8$



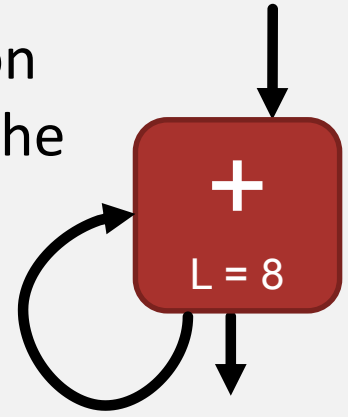


Matrix-matrix multiplication

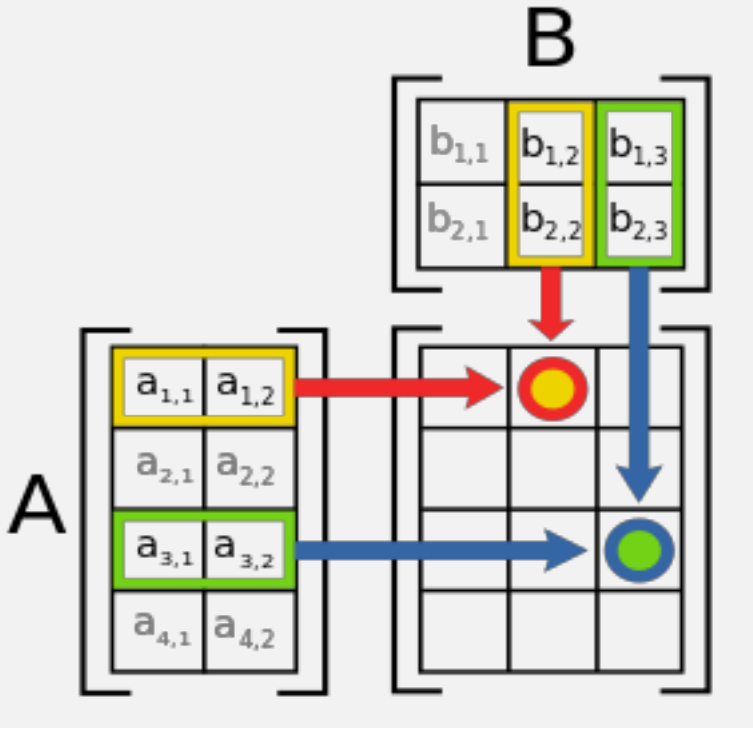
```
for (int n = 0; n < N; ++n)
  for (int p = 0; p < P; ++p)
    for (int m = 0; m < M; ++m)
      C[n, p] += A[n, m] * B[m, p];
```

Our intuition of temporal locality does not work here!

Every iteration depends on the previous:

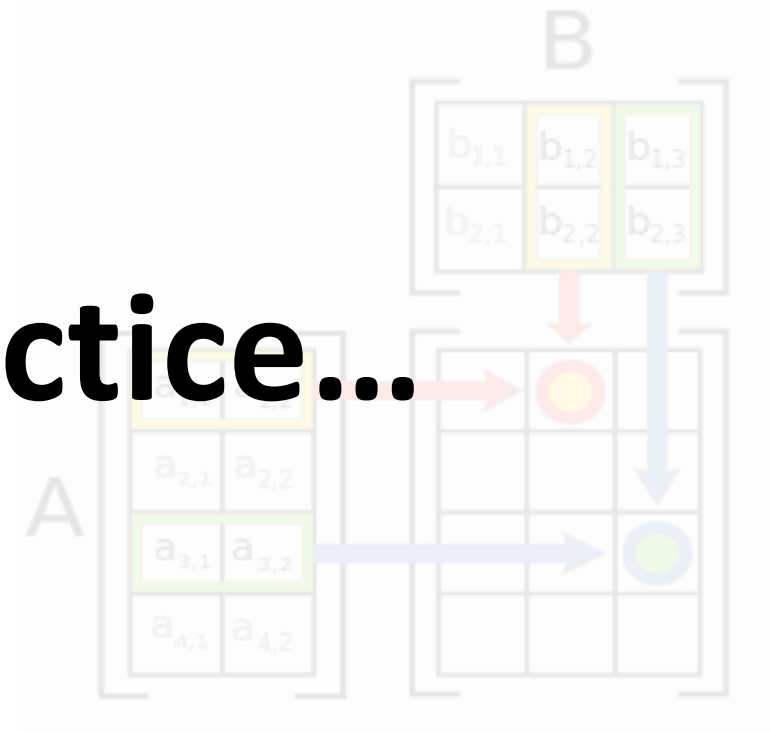


$I = 8$



Matrix-matrix multiplication

```
for (int n = 0; n < N; ++n)
  for (int p = 0; p < P; ++p)
    for (int m = 0; m < M; ++m)
      C[n, p] += A[n, m] * B[m, p];
```



Let's see this in practice…

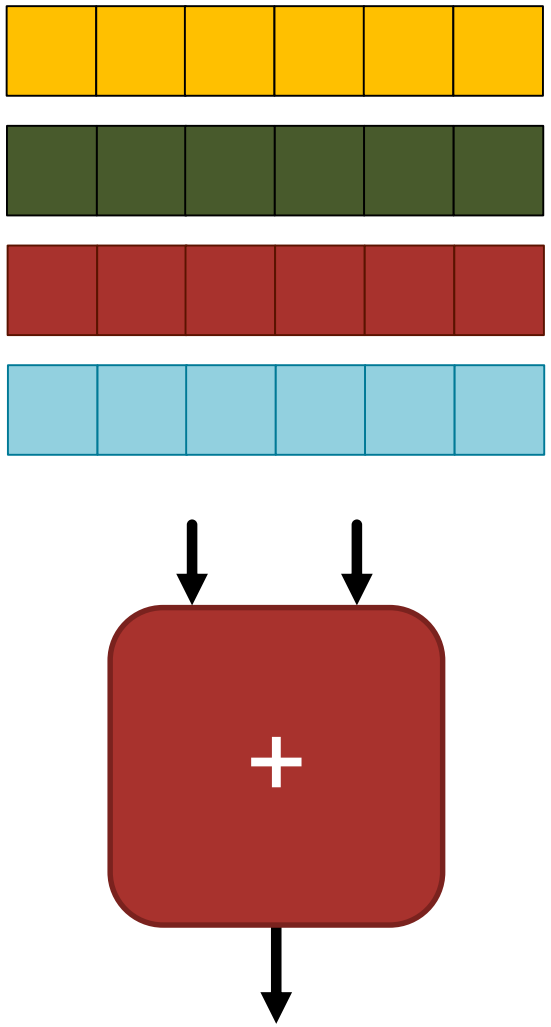
(example 5)

Our intuition of temporal locality does not work here!

Every iteration depends on the previous:

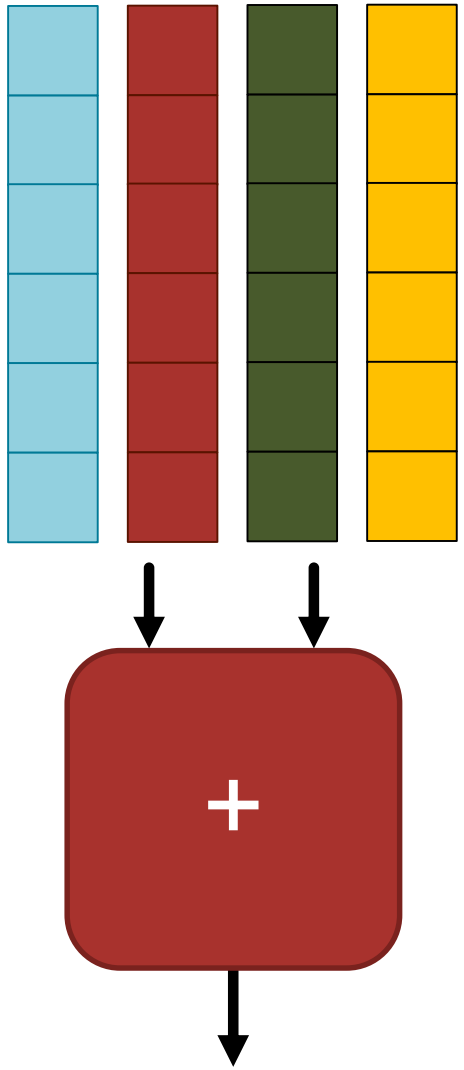
l = 8

Solving loop-carried data dependencies



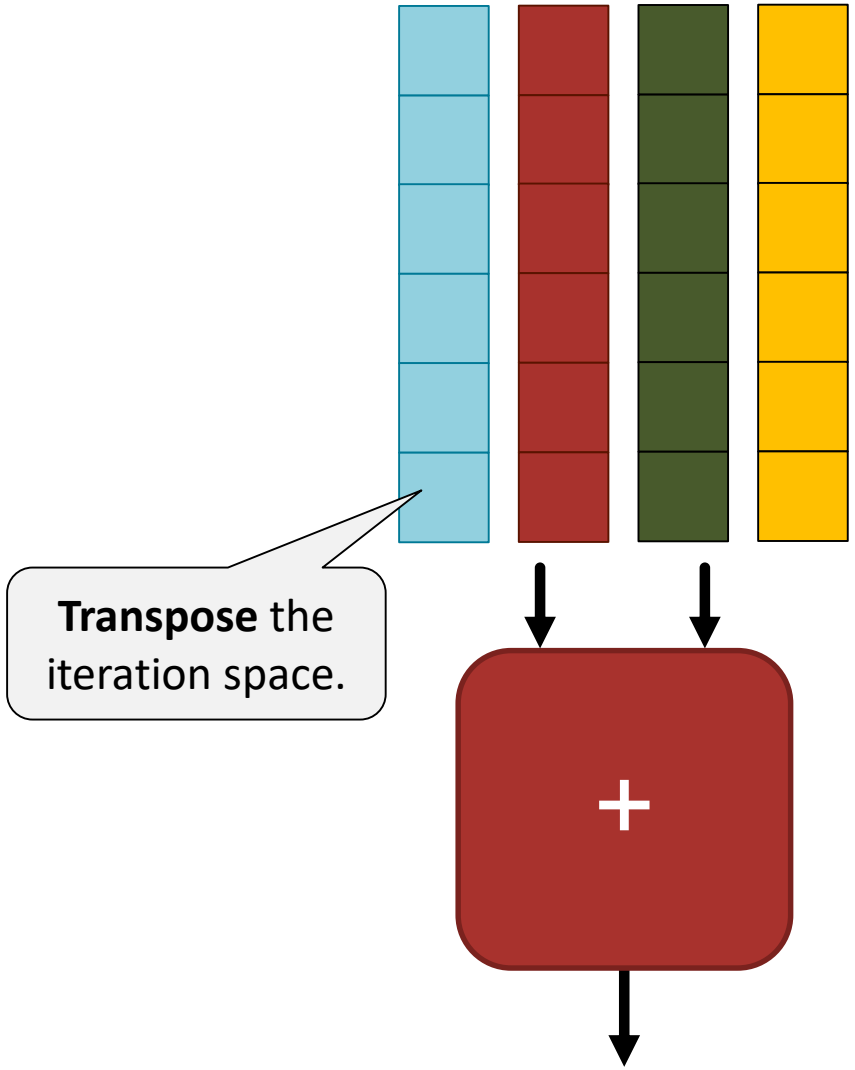

```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += in.Pop();
  }
  out.Push(acc);
}
```

Solving loop-carried data dependencies



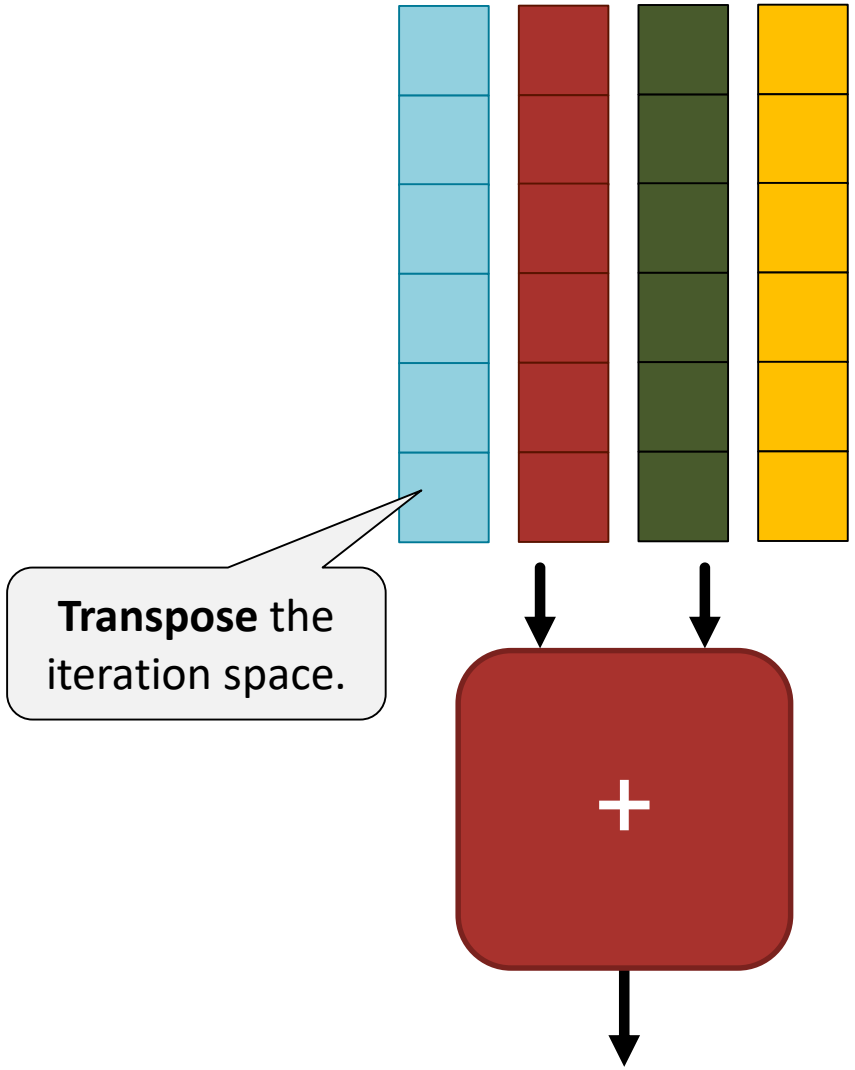

```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += in.Pop();
  }
  out.Push(acc);
}
```

Solving loop-carried data dependencies




```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += in.Pop();
  }
  out.Push(acc);
}
```

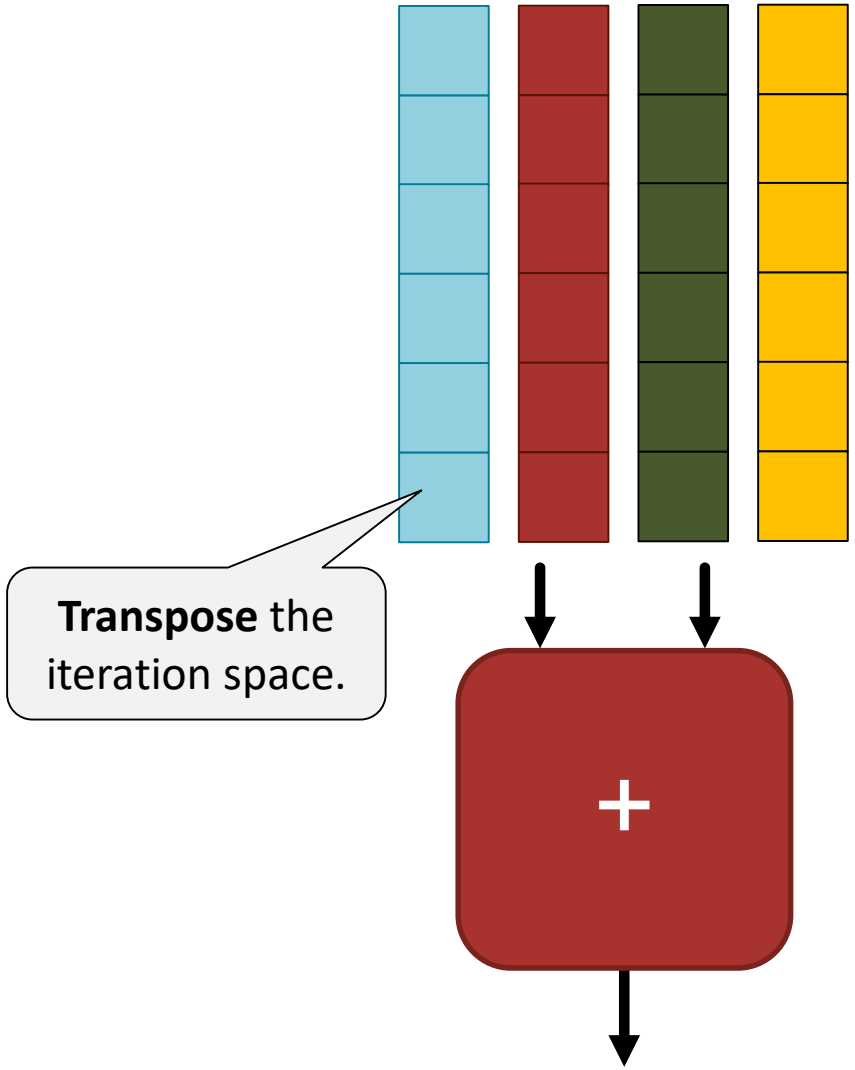
Solving loop-carried data dependencies




```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += in.Pop();
  }
  out.Push(acc);
}
```

```
float acc[N];
for (int j = 0; j < M; ++j) {
  for (int i = 0; i < N; ++i) {
    acc[i] += in.Pop();
  }
}
for (int i = 0; i < N; ++i) {
  out.Push(acc[i]);
}
```

Solving loop-carried data dependencies

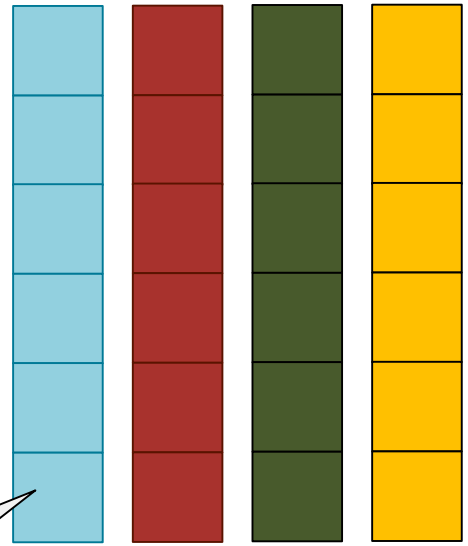



```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += in.Pop();
  }
  out.Push(acc);
```

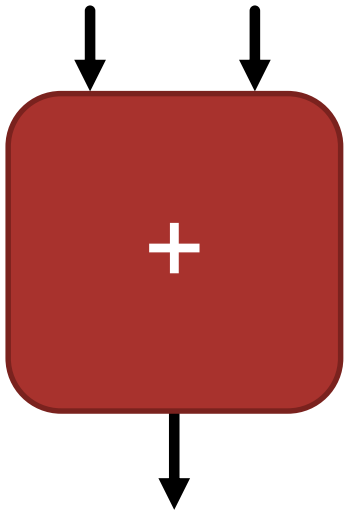
Comes at the cost of buffer space.

```
float acc[N];
for (int j = 0; j < M; ++j) {
  for (int i = 0; i < N; ++i) {
    acc[i] += in.Pop();
  }
}
for (int i = 0; i < N; ++i) {
  out.Push(acc[i]);
}
```

Solving loop-carried data dependencies



Transpose the iteration space.




```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += in.Pop();
  }
  out.Push(acc);
```

Comes at the cost of buffer space.

```
float acc[N];
for (int j = 0; j < M; ++j) {
  for (int i = 0; i < N; ++i) {
    acc[i] += in.Pop();
  }
}
for (int i = 0; i < N; ++i) {
  out.Push(acc[i]);
}
```

Only needs to be larger than the latency

Locality in the program

```
for (int n = 0; n < N; ++n) {
  float acc[P];

  for (int k = 0; k < K; ++k) {
    const auto a = A[n*K + k];

    for (int m = 0; m < M; ++m) {
      #pragma HLS PIPELINE II=1
      const float prev = (k == 0) ? 0 : acc[m];
      acc[m] = prev + a * B[k*M + m];
    }
  }
  // ...
}
```

Locality in the program

```
for (int n = 0; n < N; ++n) {
  float acc[P];

  for (int k = 0; k < K; ++k) {
    const auto a = A[n*K + k];

    for (int m = 0; m < M; ++m) {
      #pragma HLS PIPELINE II=1
      const float prev = (k == 0) ? 0 : acc[m];
      acc[m] = prev + a * B[k*M + m];
    }
  }
  // ...
}
```

Spatial locality: Vectorizable.

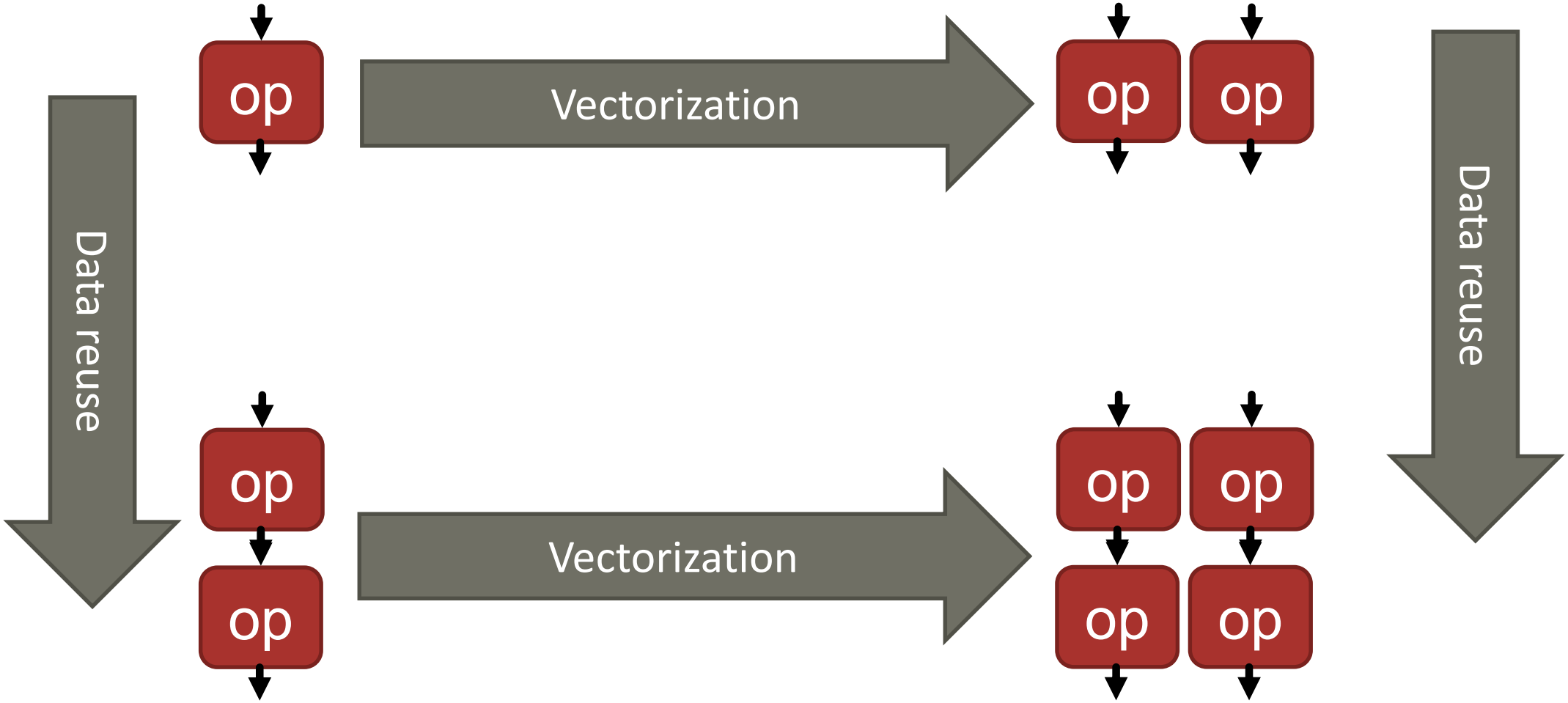
Locality in the program

```
for (int n = 0; n < N; ++n) {
  float acc[P];

  for (int k = 0; k < K; ++k) {
    const auto a = A[n*K + k];

    for (int m = 0; m < M; ++m) {
      #pragma HLS PIPELINE II=1
      const float prev = (k == 0) ? 0 : acc[m];
      acc[m] = prev + a * B[k*M + m];
    }
  }
  // ...
}
```

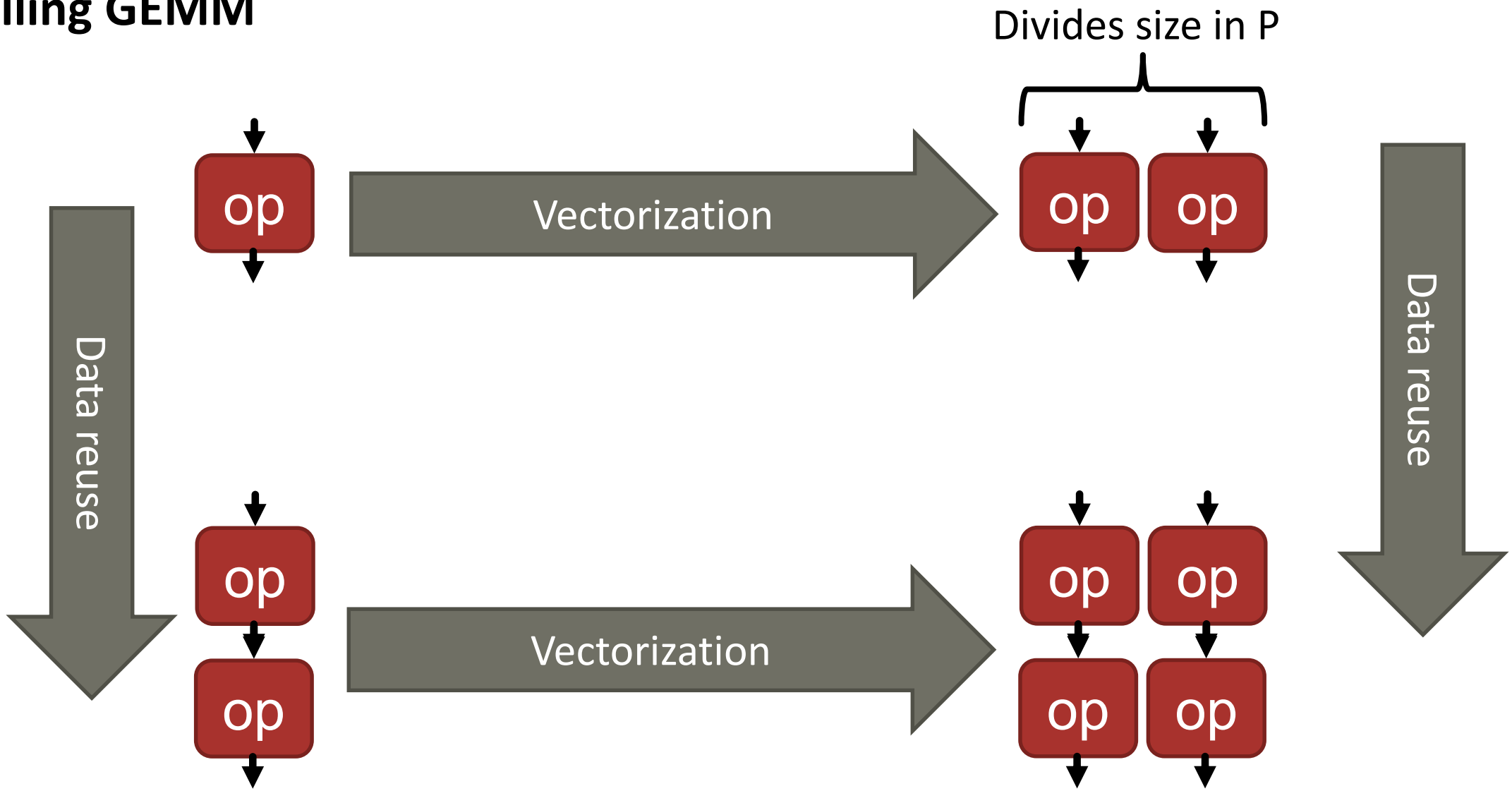
Temporal locality: Reused P times. Load more of these and tile N!

Spatial locality: Vectorizable.

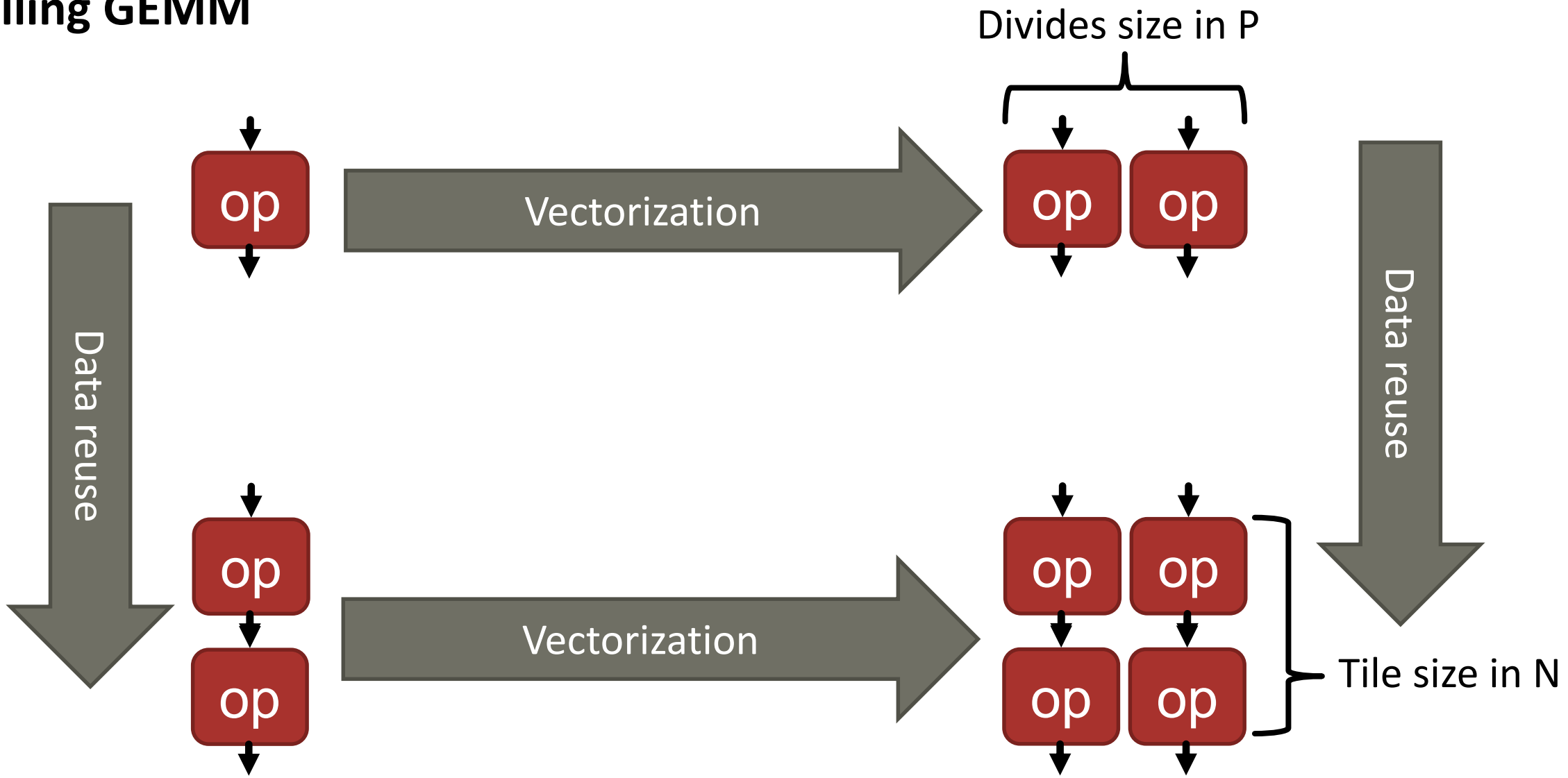
Unrolling GEMM



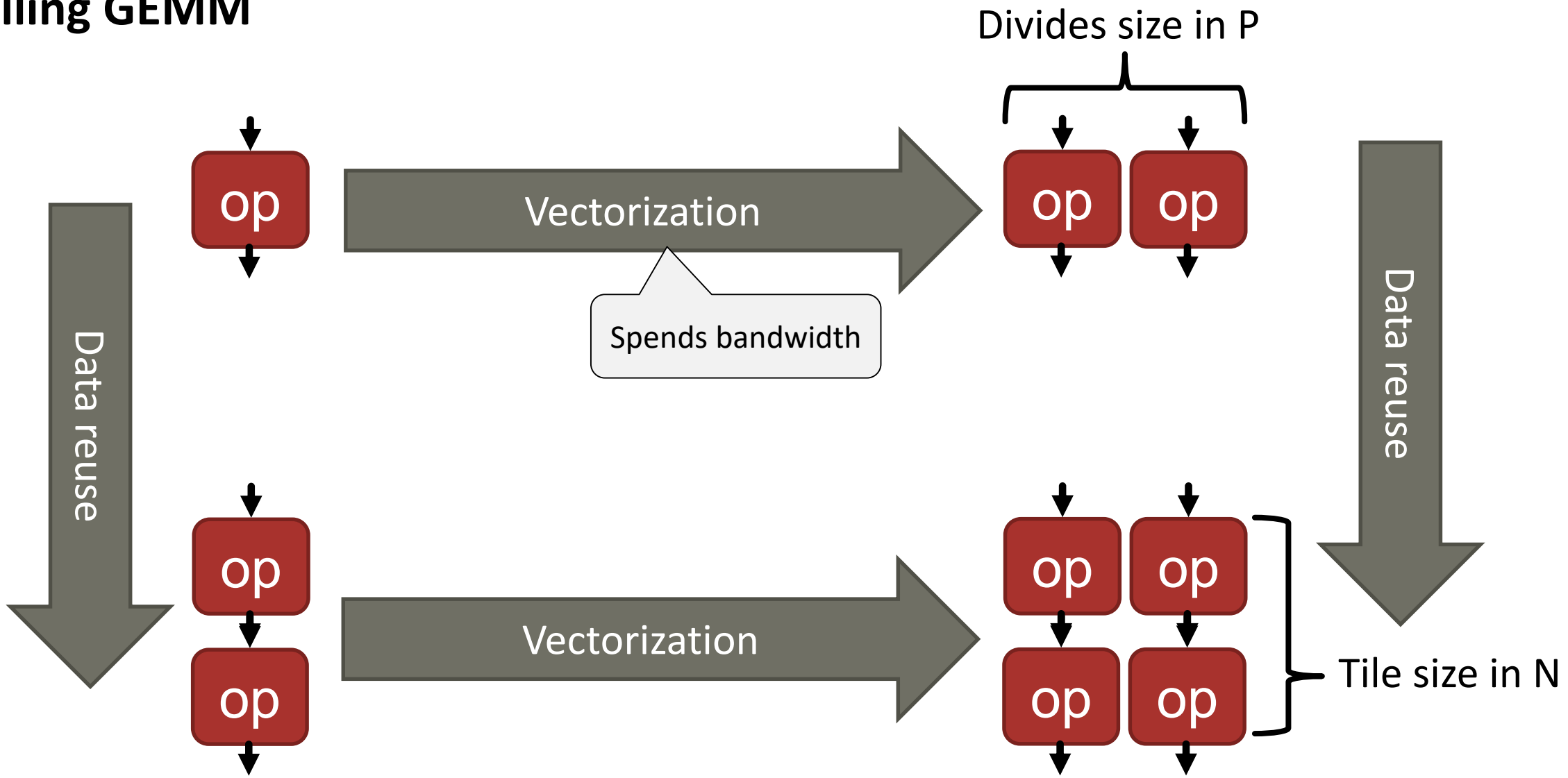
Unrolling GEMM



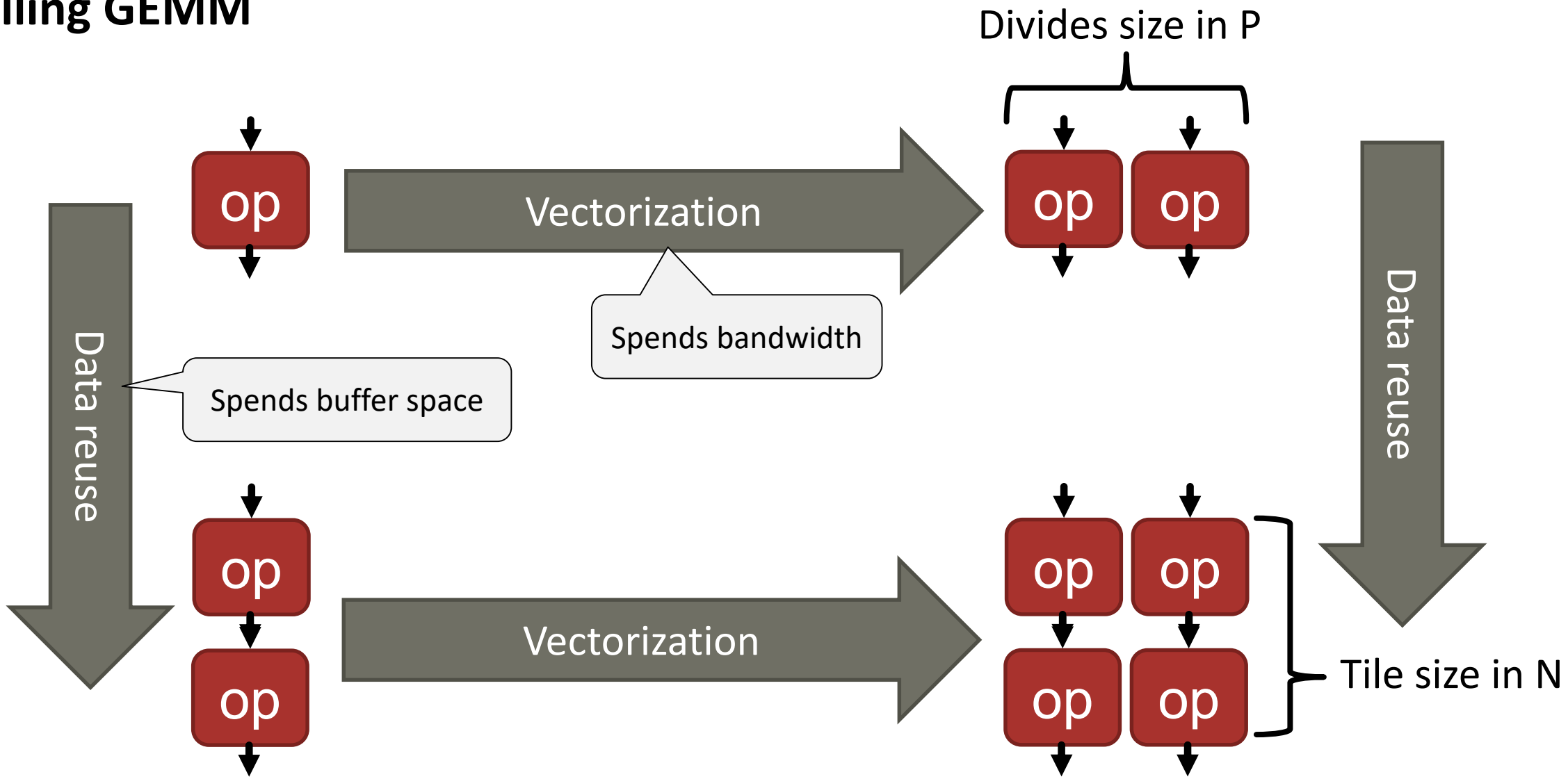
Unrolling GEMM



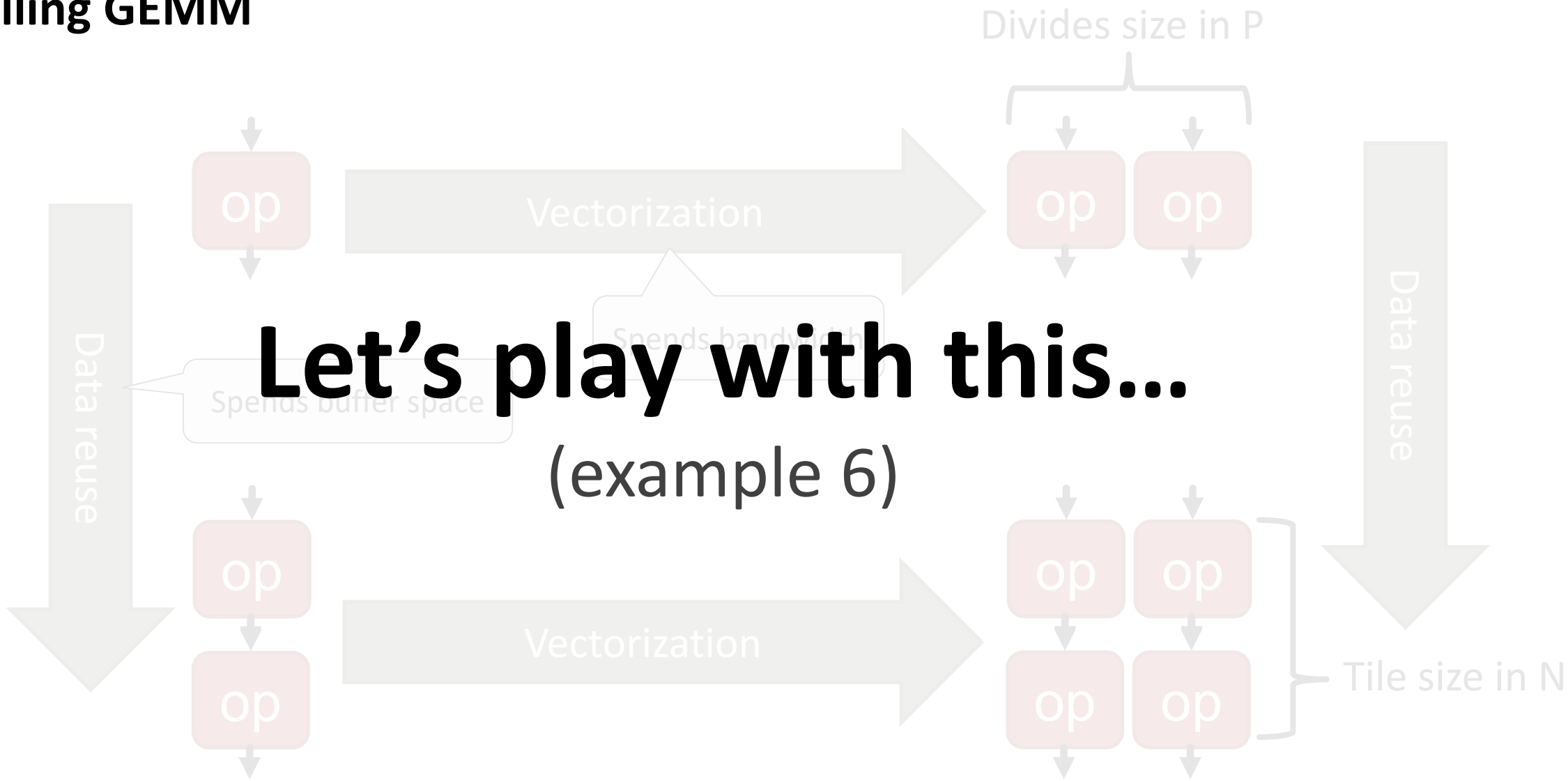
Unrolling GEMM



Unrolling GEMM



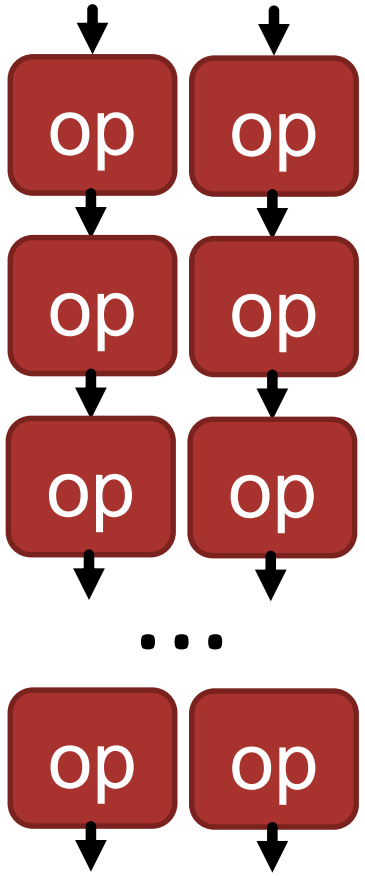
Unrolling GEMM



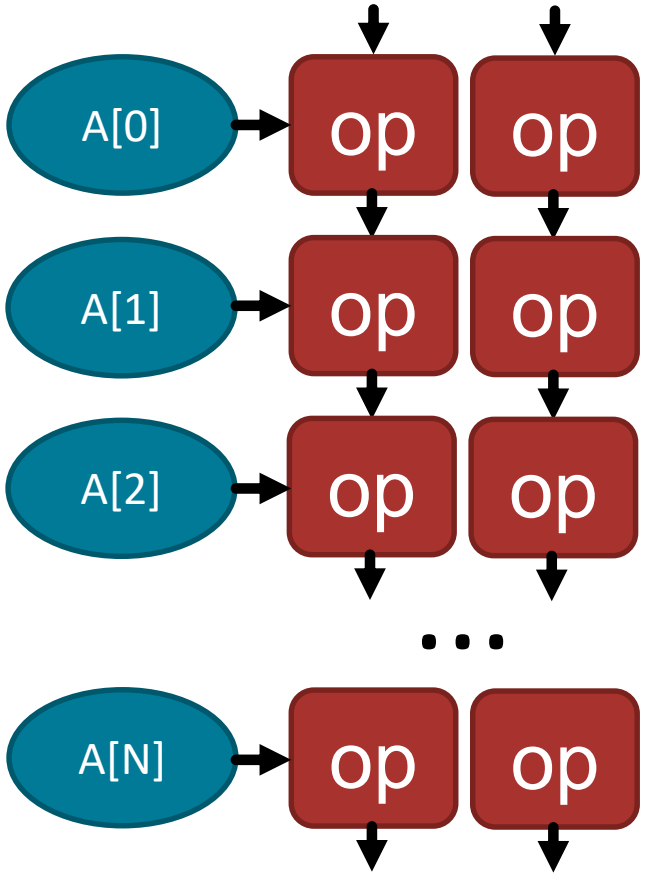
Let’s play with this…

(example 6)

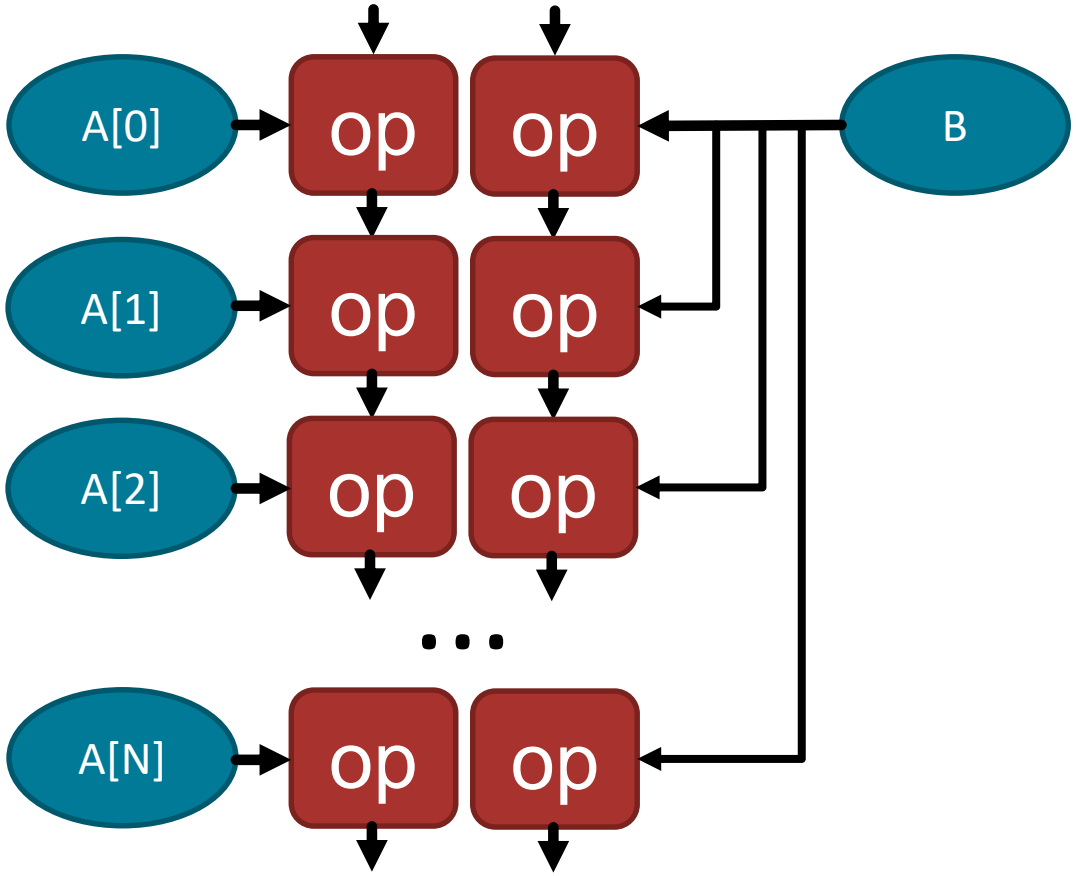
Fanout issue



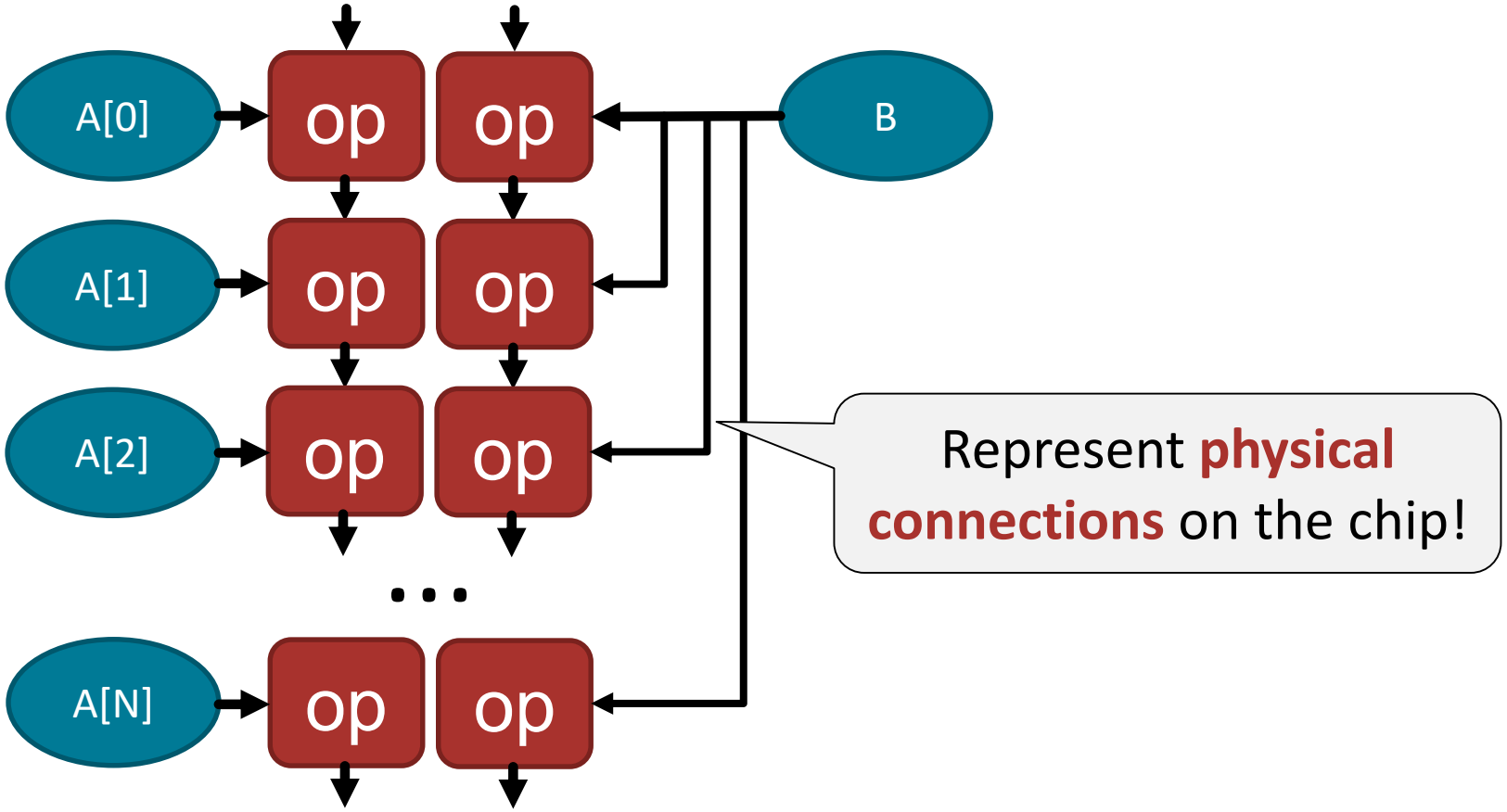
Fanout issue



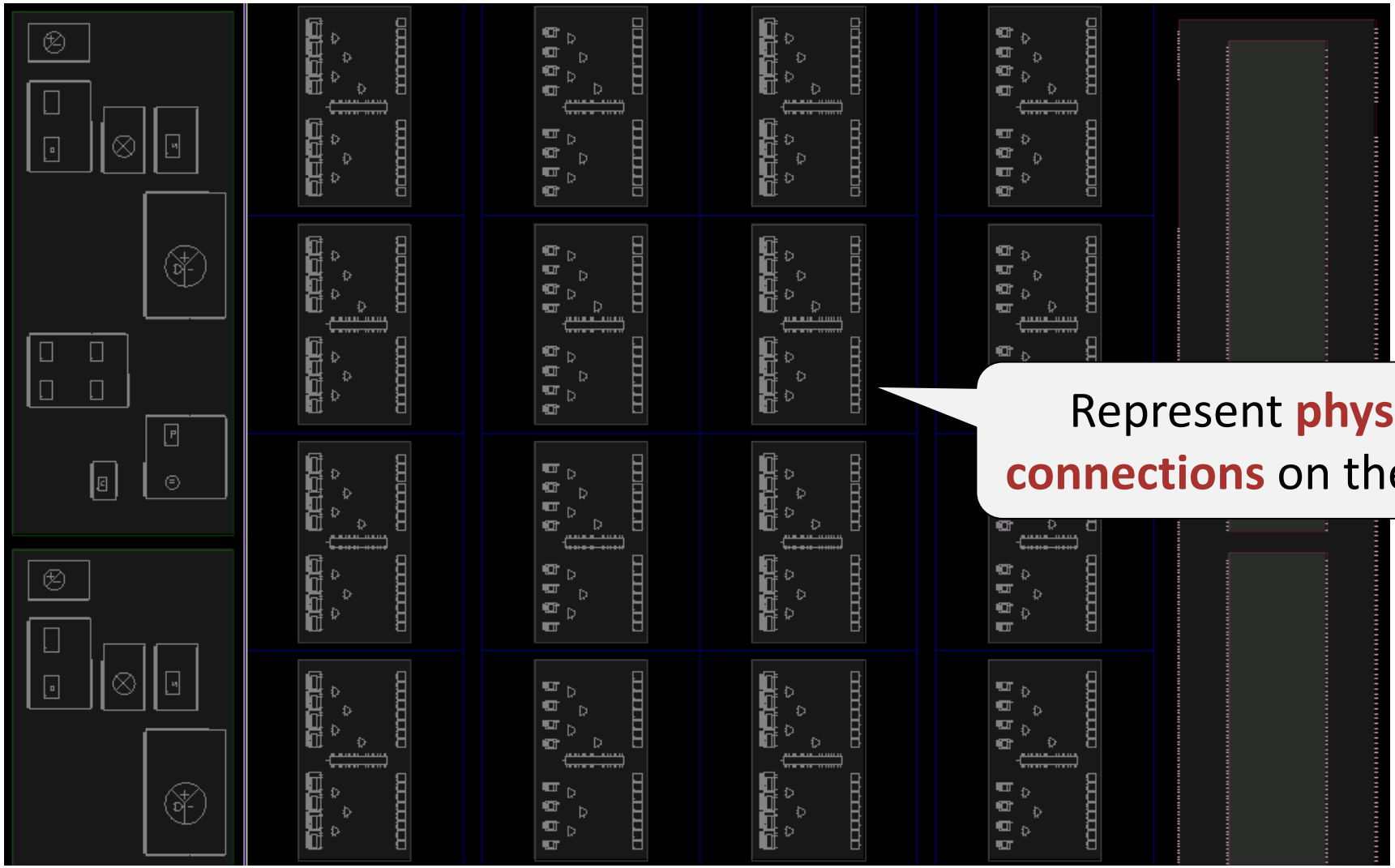
Fanout issue



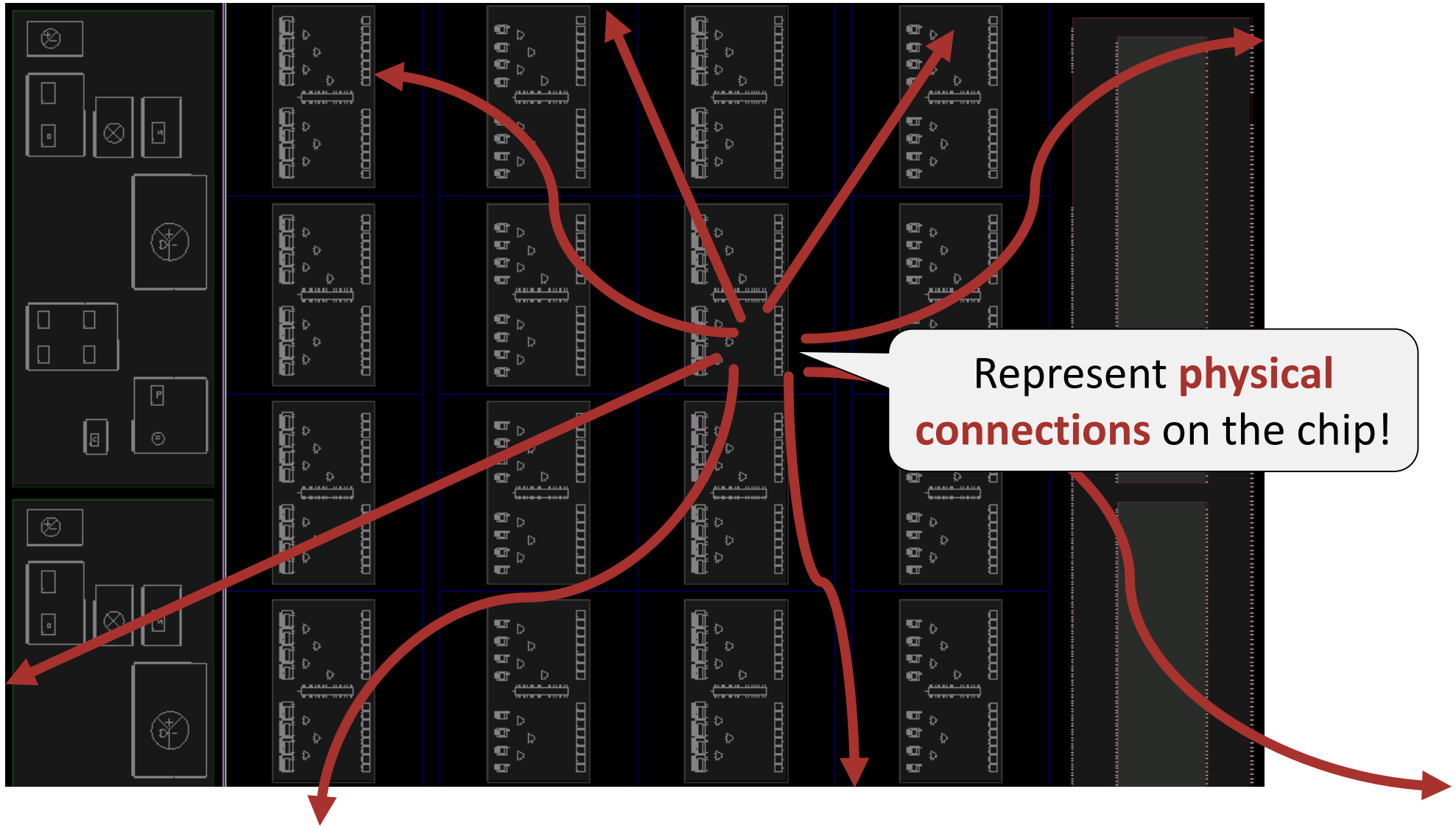
Fanout issue



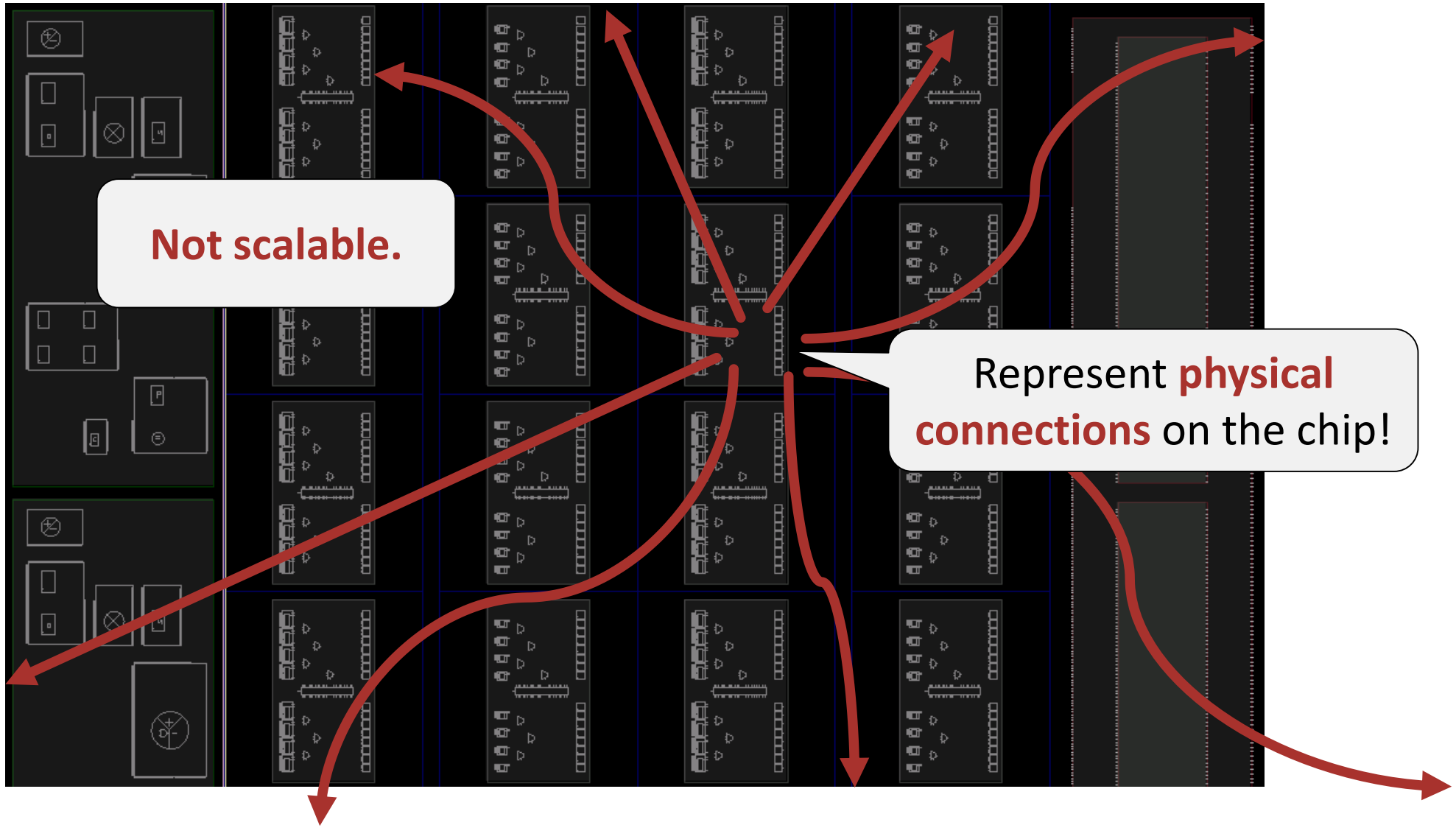
Fanout issue



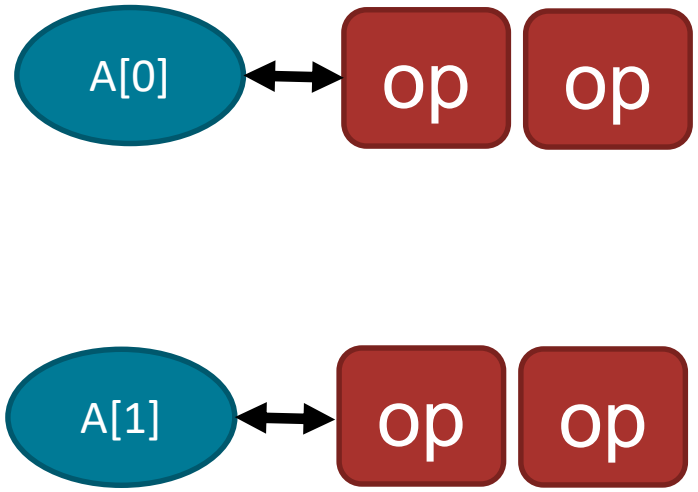
Fanout issue



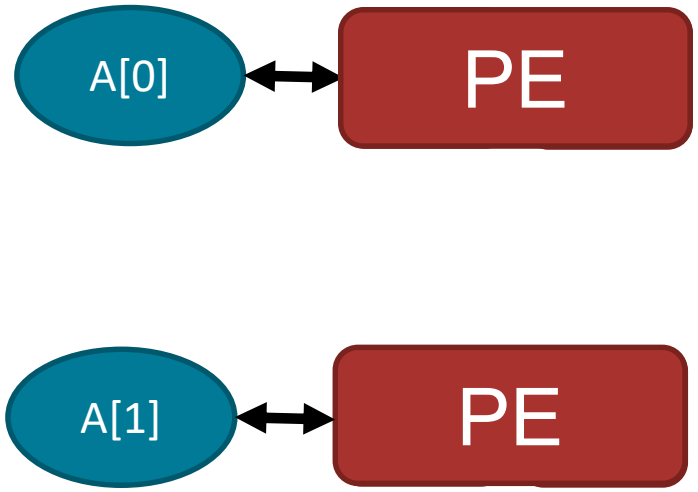
Fanout issue



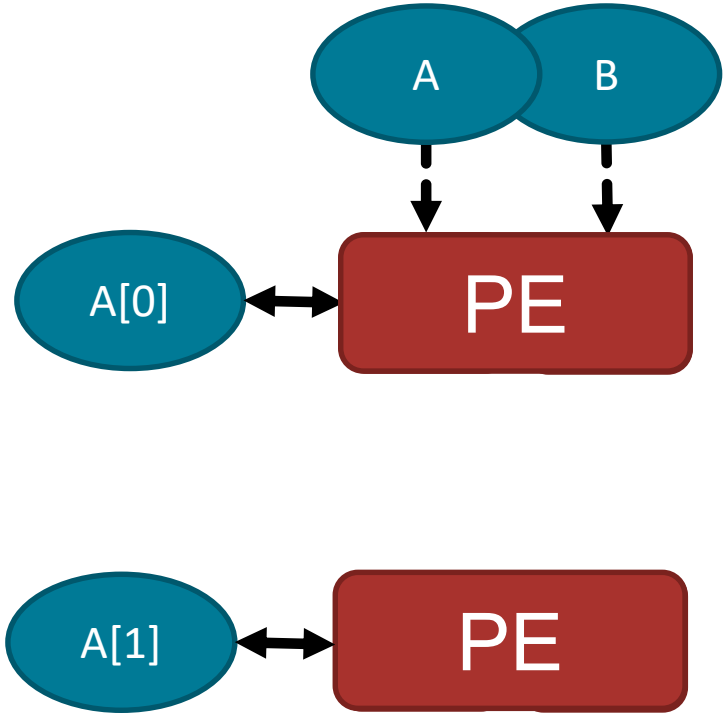
Breaking into processing elements



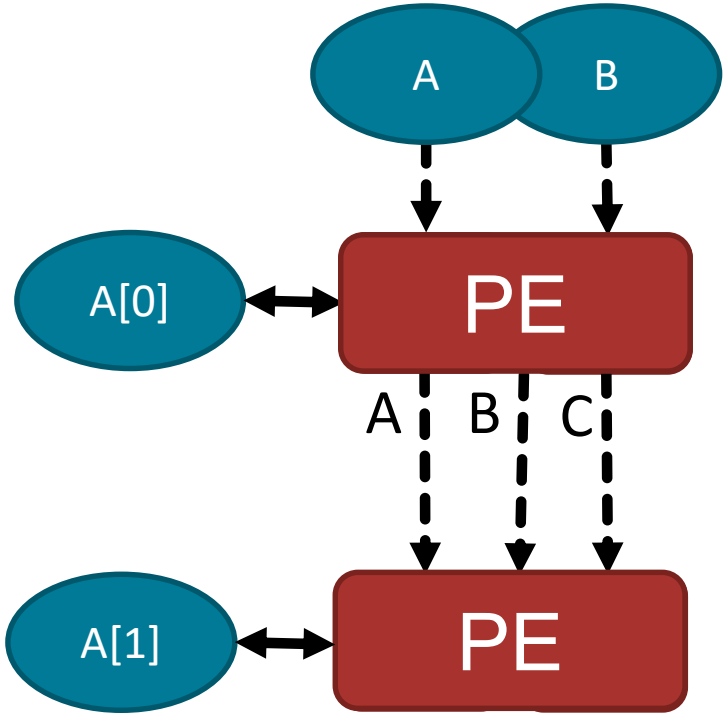
Breaking into processing elements



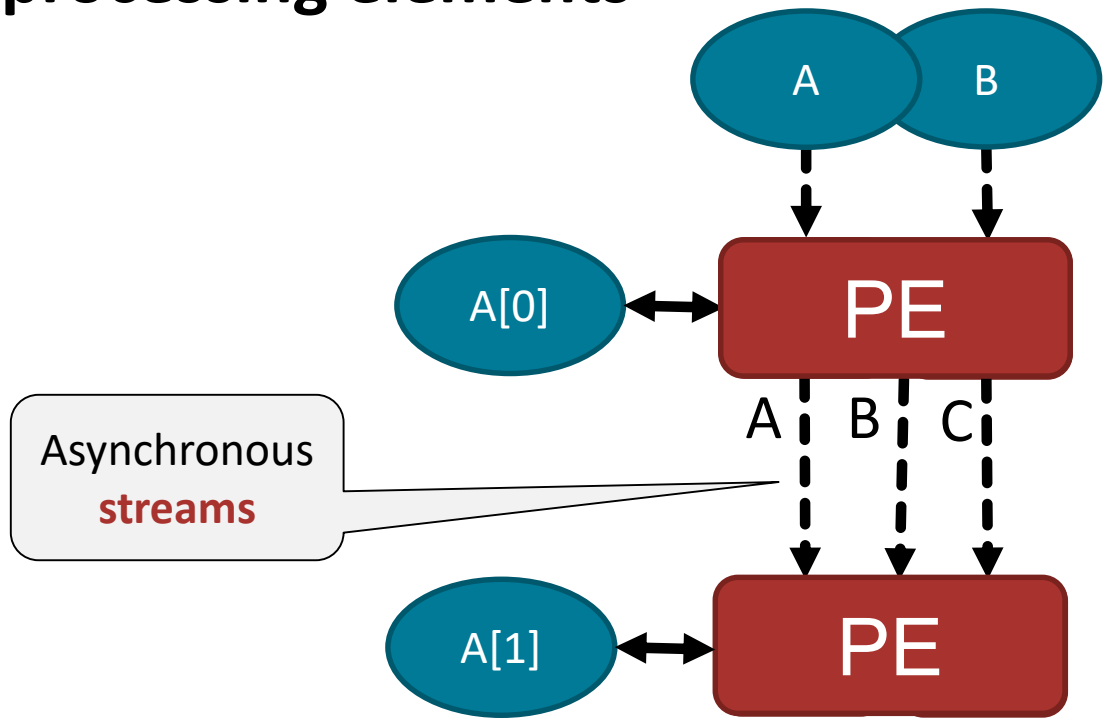
Breaking into processing elements



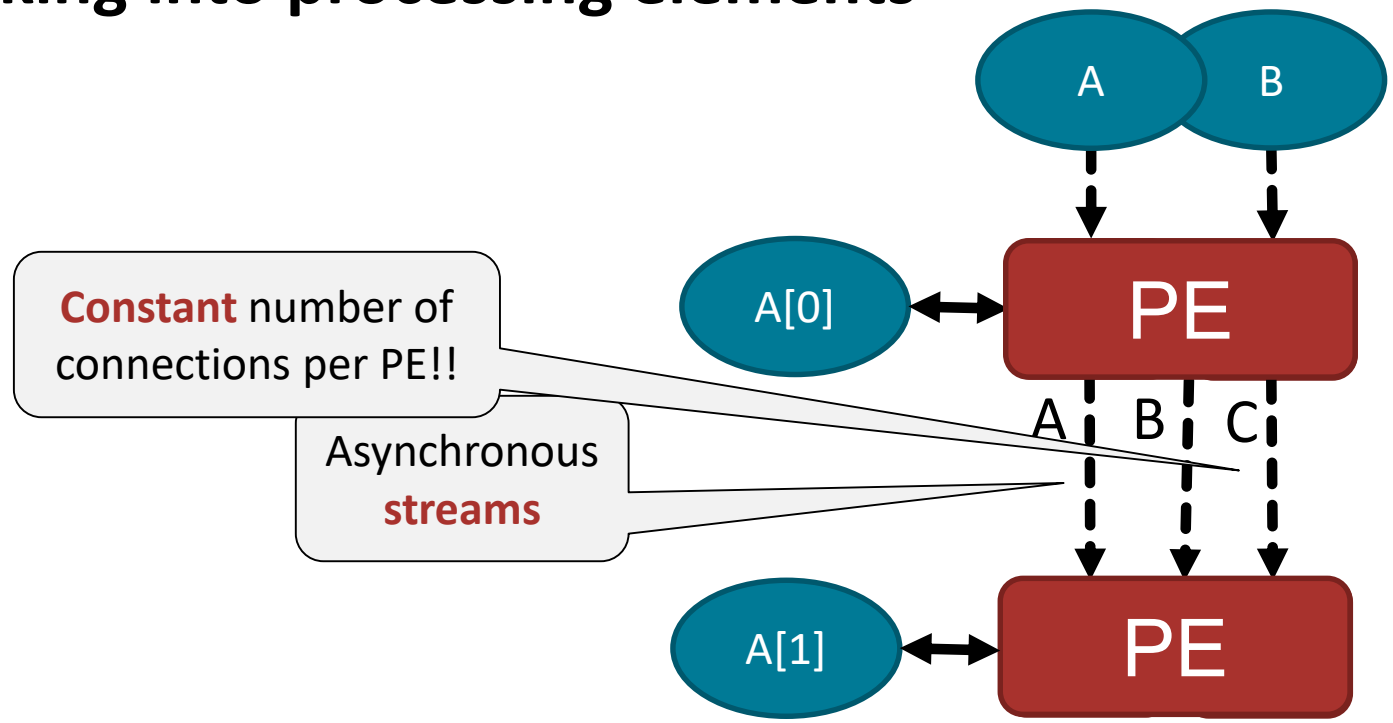
Breaking into processing elements



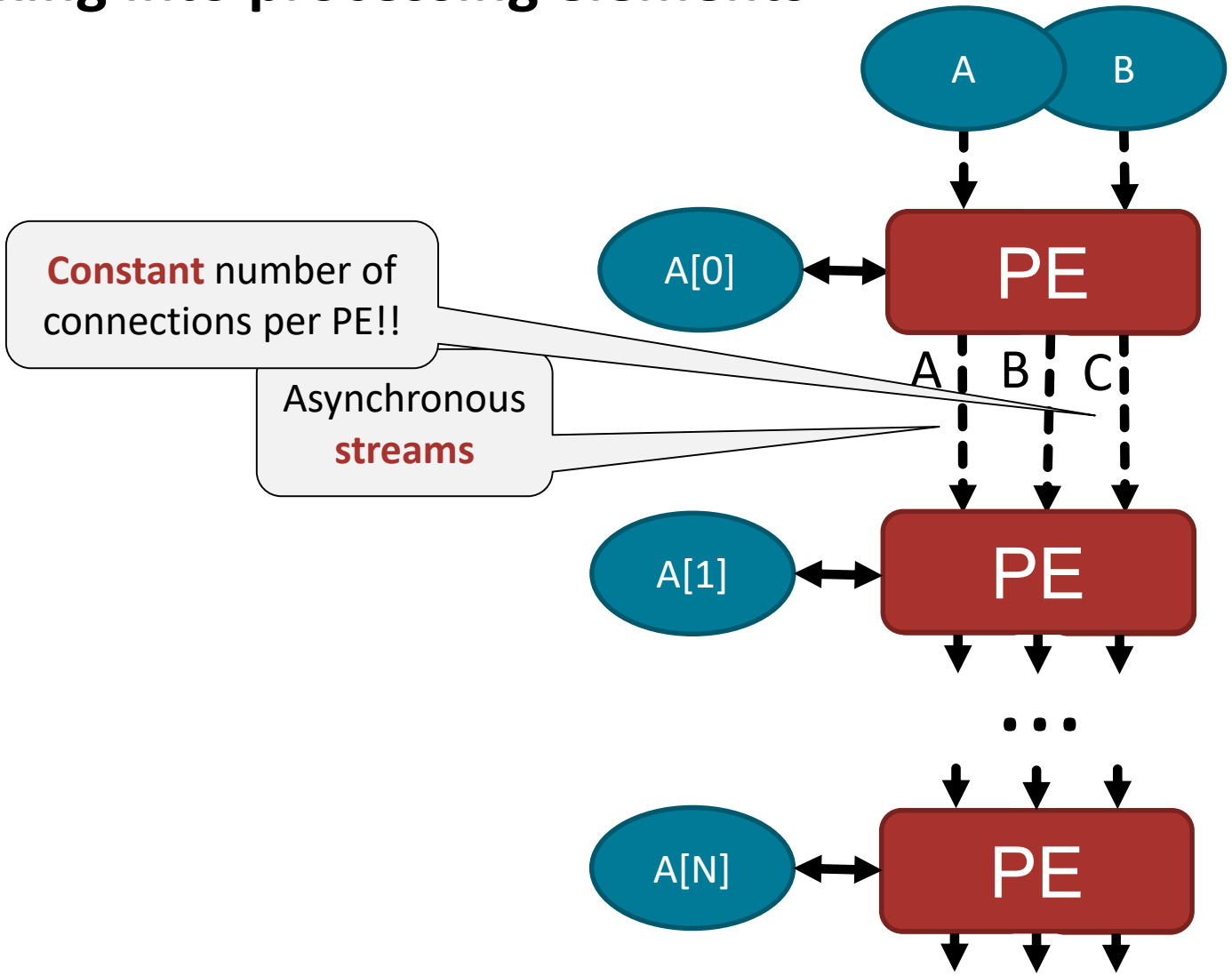
Breaking into processing elements



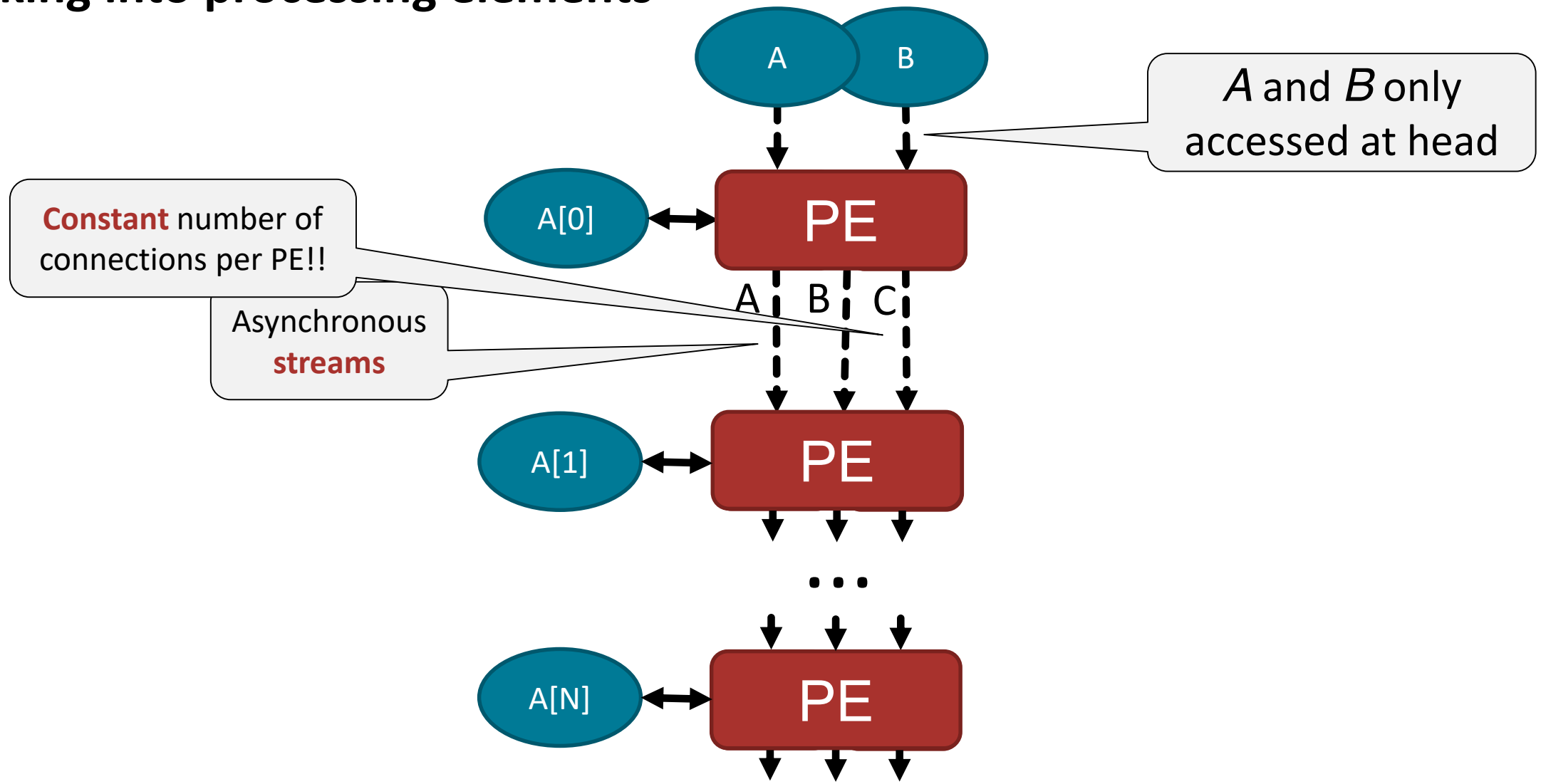
Breaking into processing elements



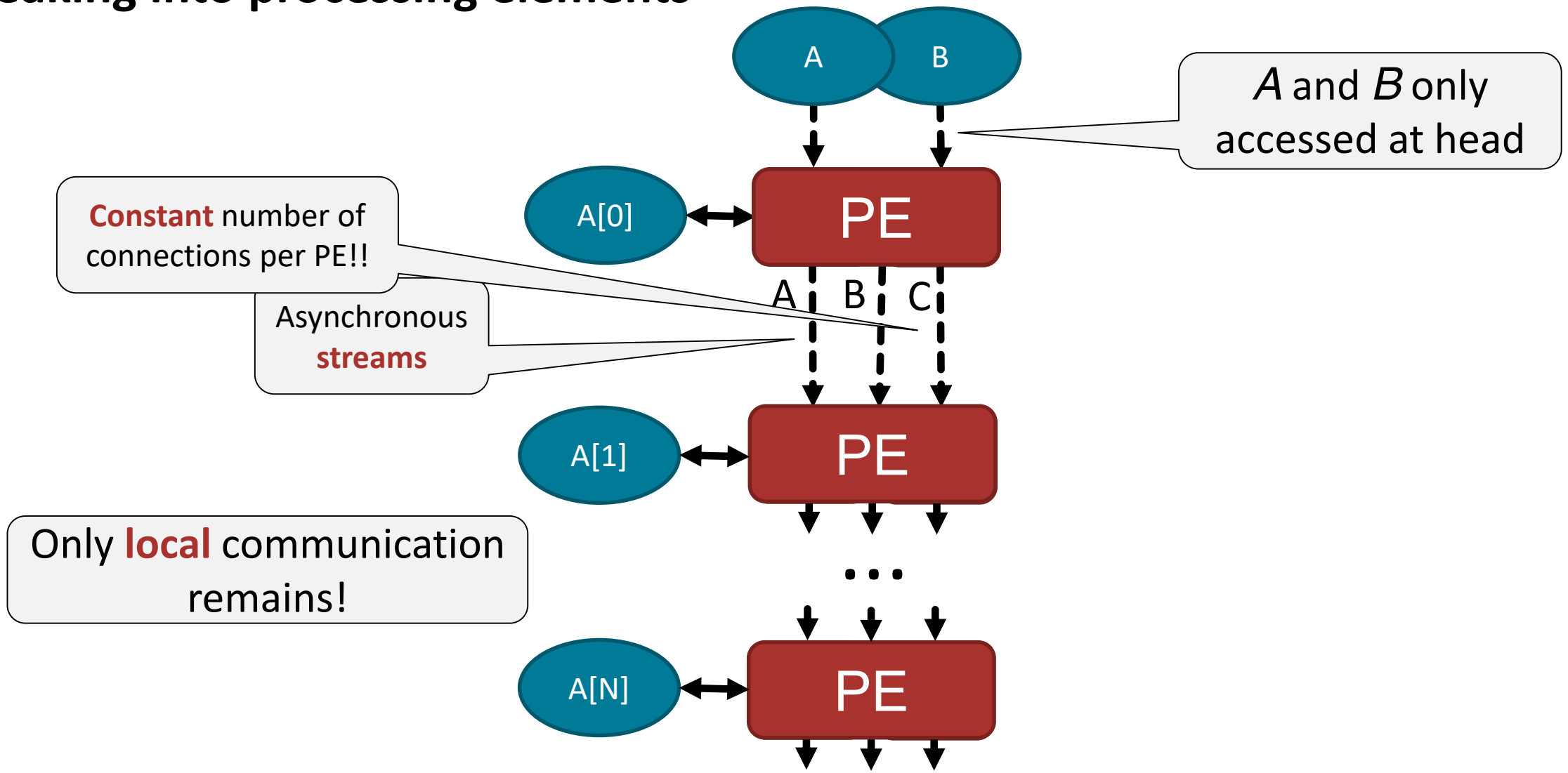
Breaking into processing elements



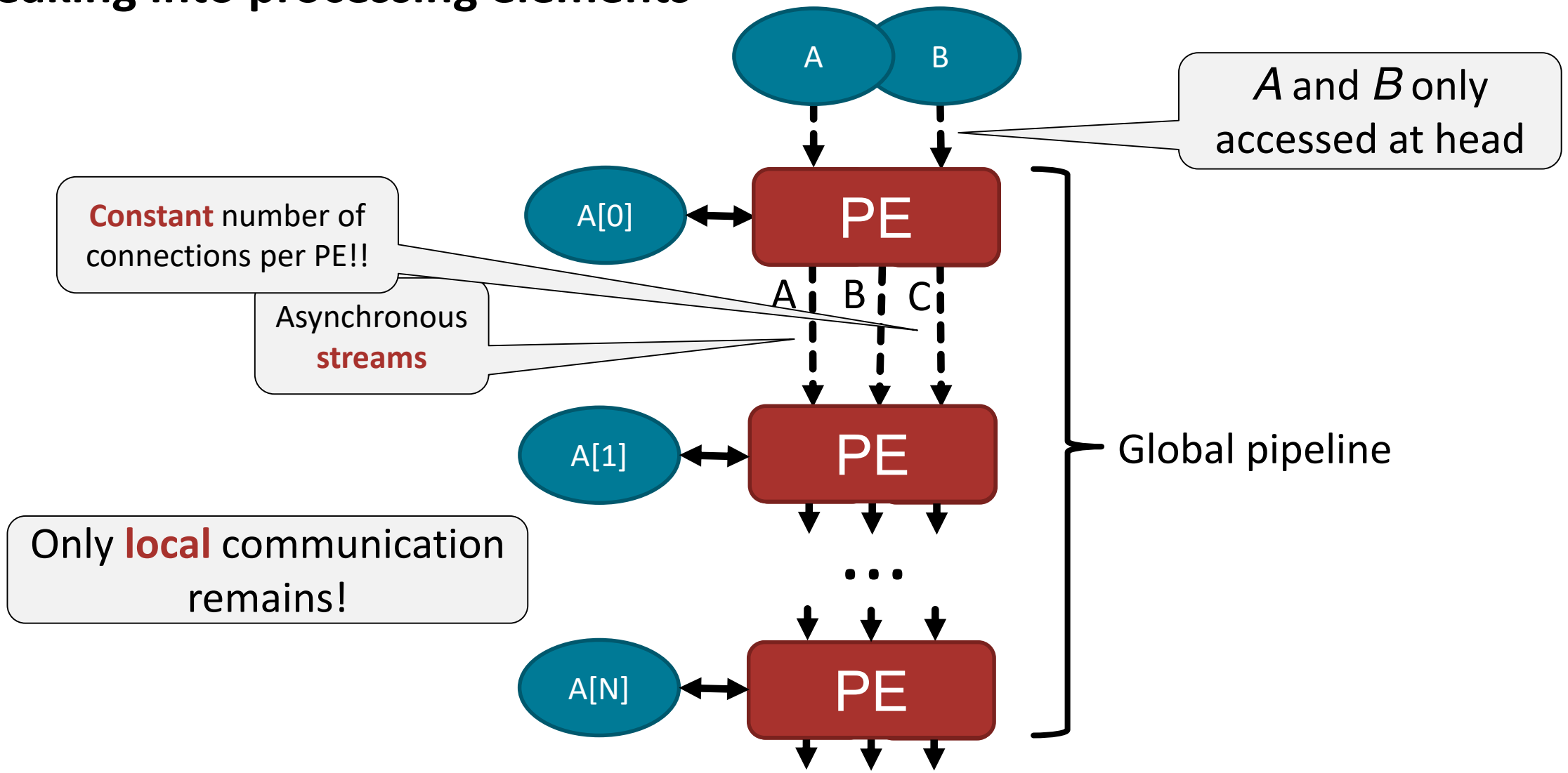
Breaking into processing elements



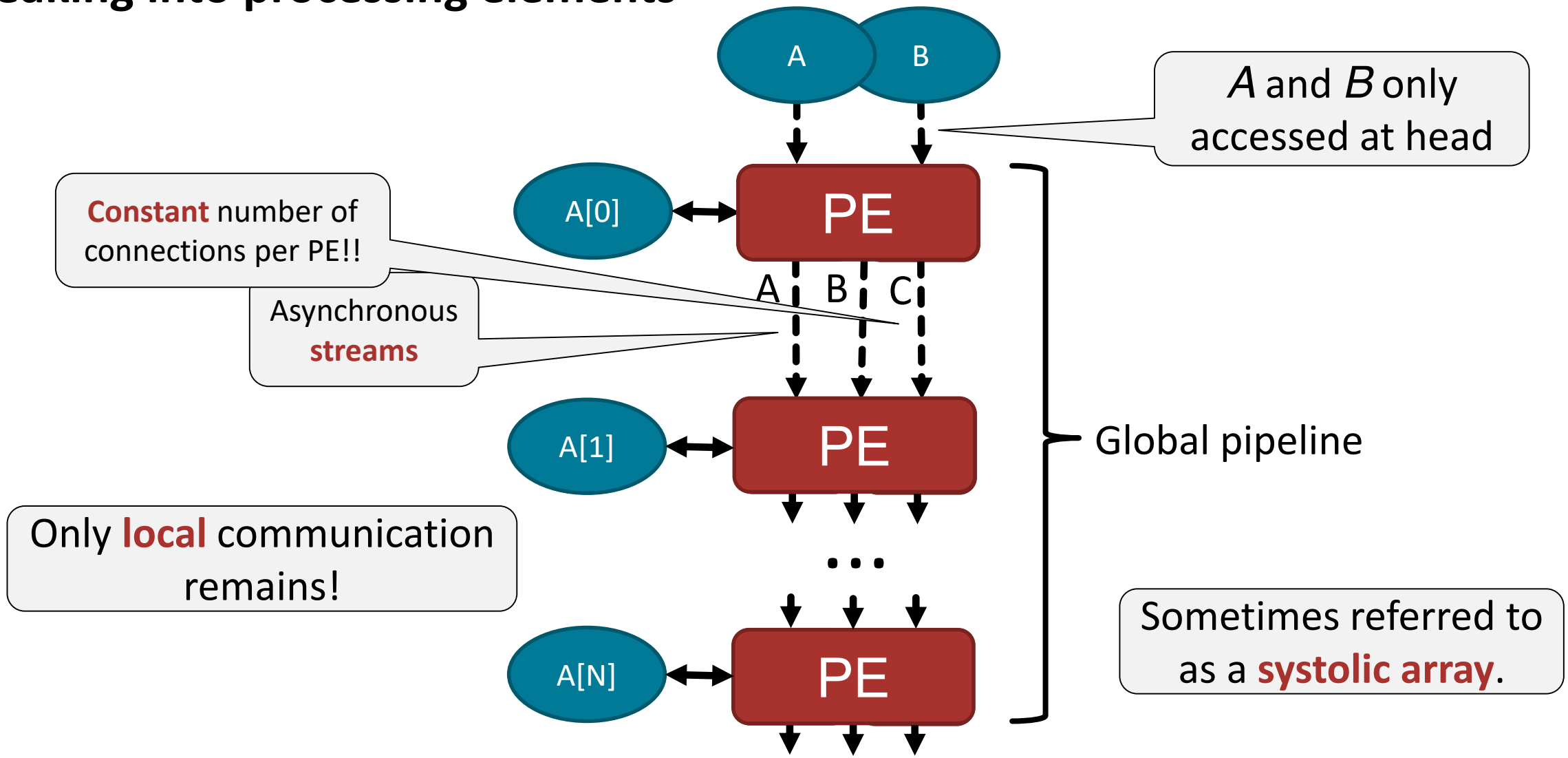
Breaking into processing elements



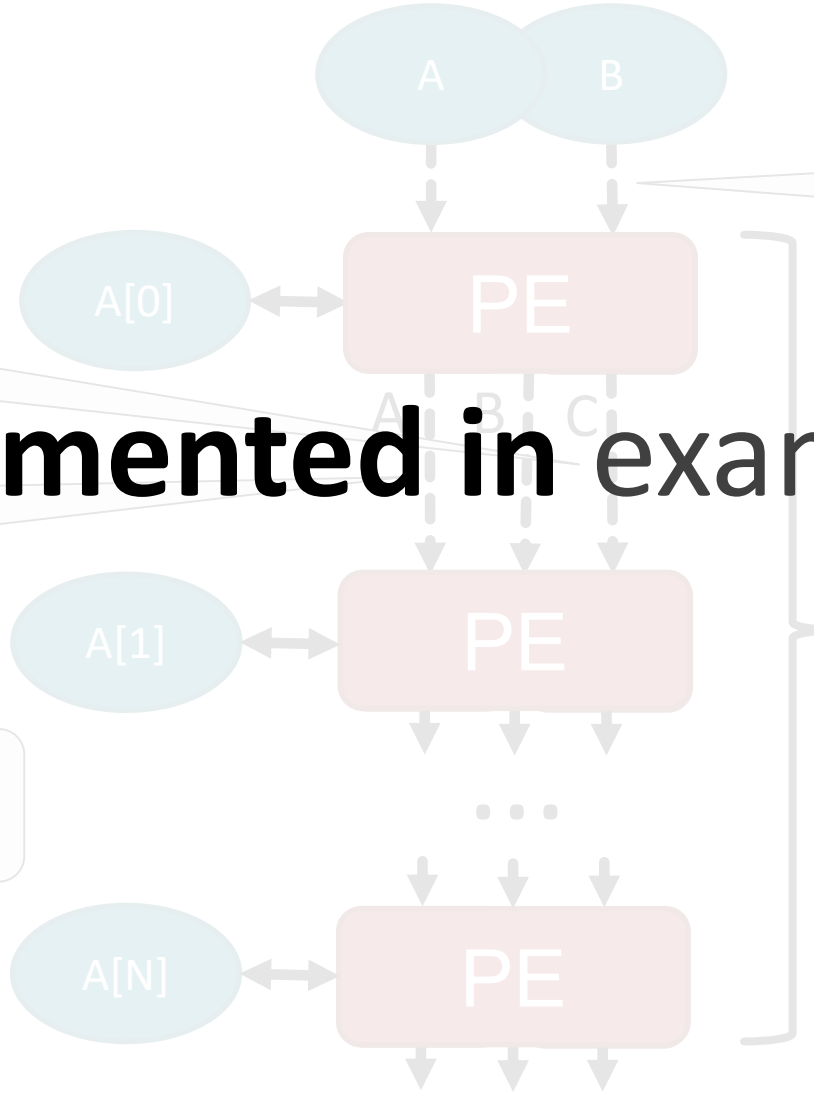
Breaking into processing elements



Breaking into processing elements



Breaking into processing elements



A and *B* only accessed at head

Constant number of connections per PE!!

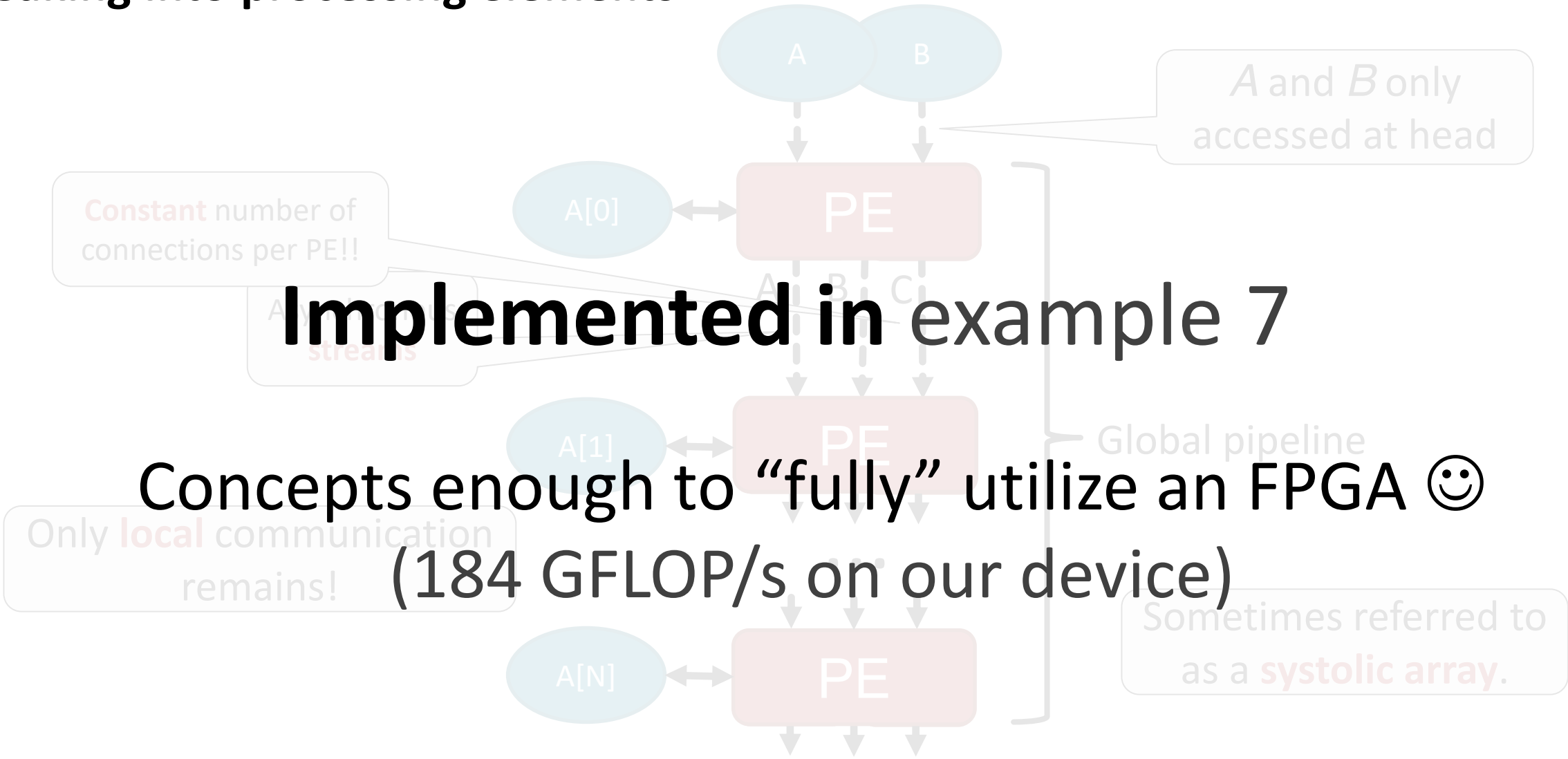
Only **local** communication remains!

Global pipeline

Sometimes referred to as a **systolic array**.

Implemented in example 7

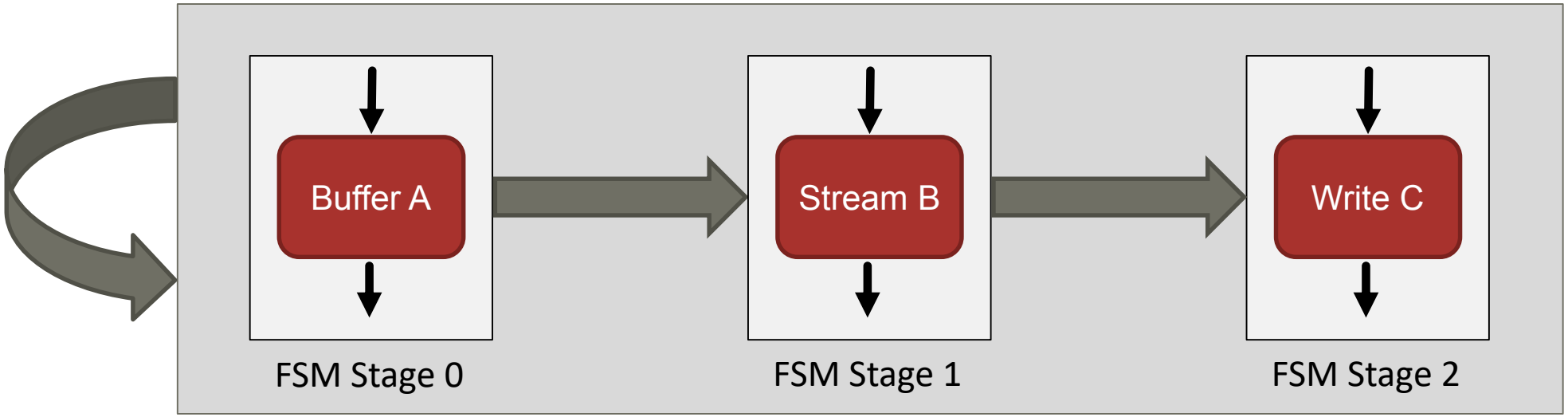
Breaking into processing elements



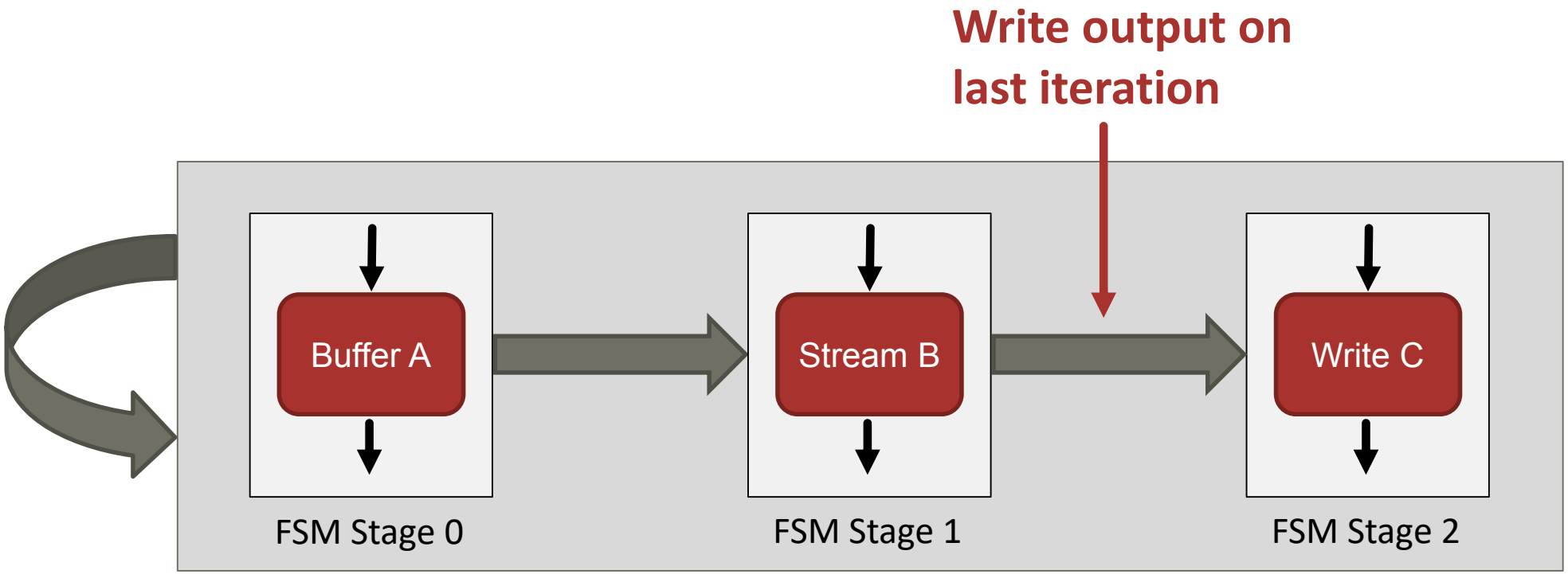
Implemented in example 7

Concepts enough to “fully” utilize an FPGA ☺
(184 GFLOP/s on our device)

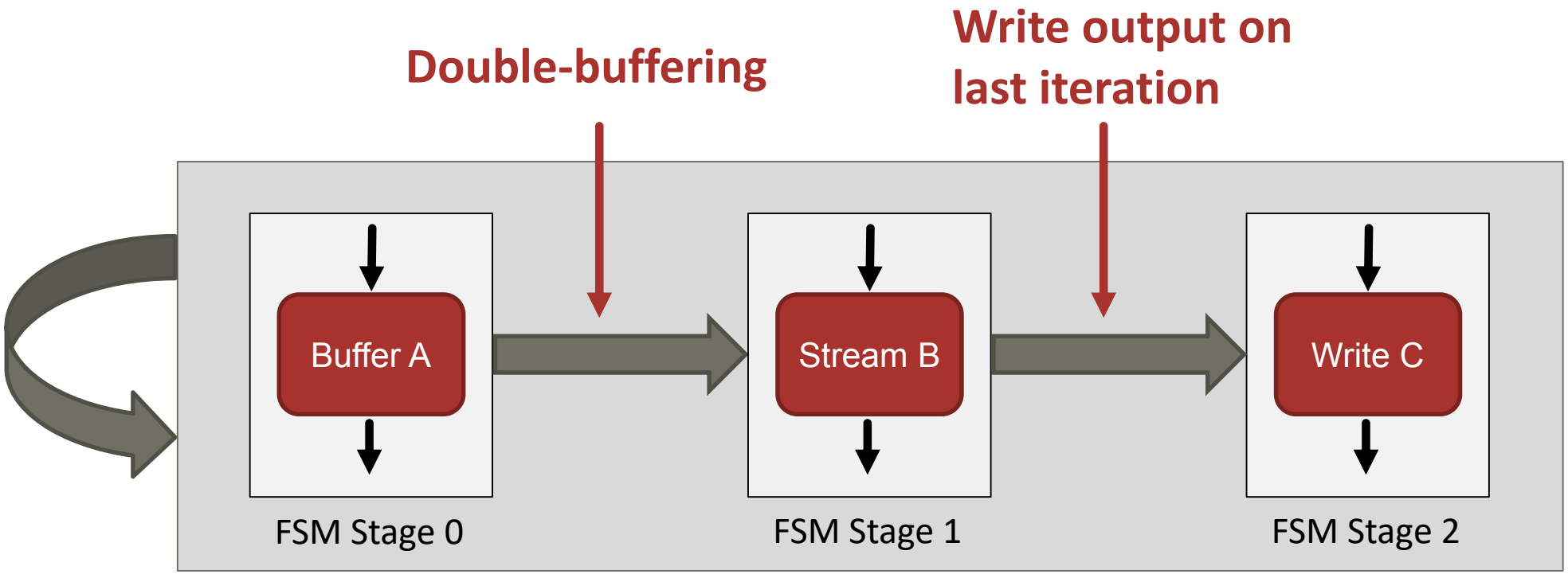
Remaining optimization potential



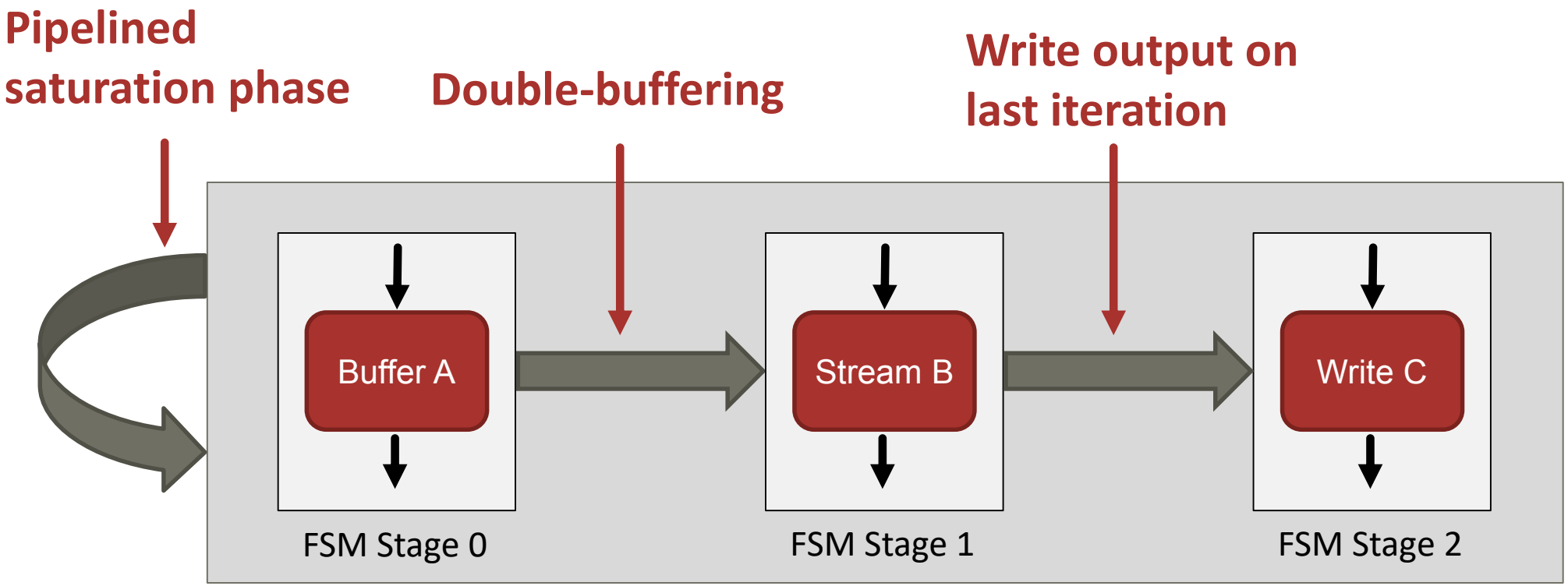
Remaining optimization potential



Remaining optimization potential



Remaining optimization potential



Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Summary

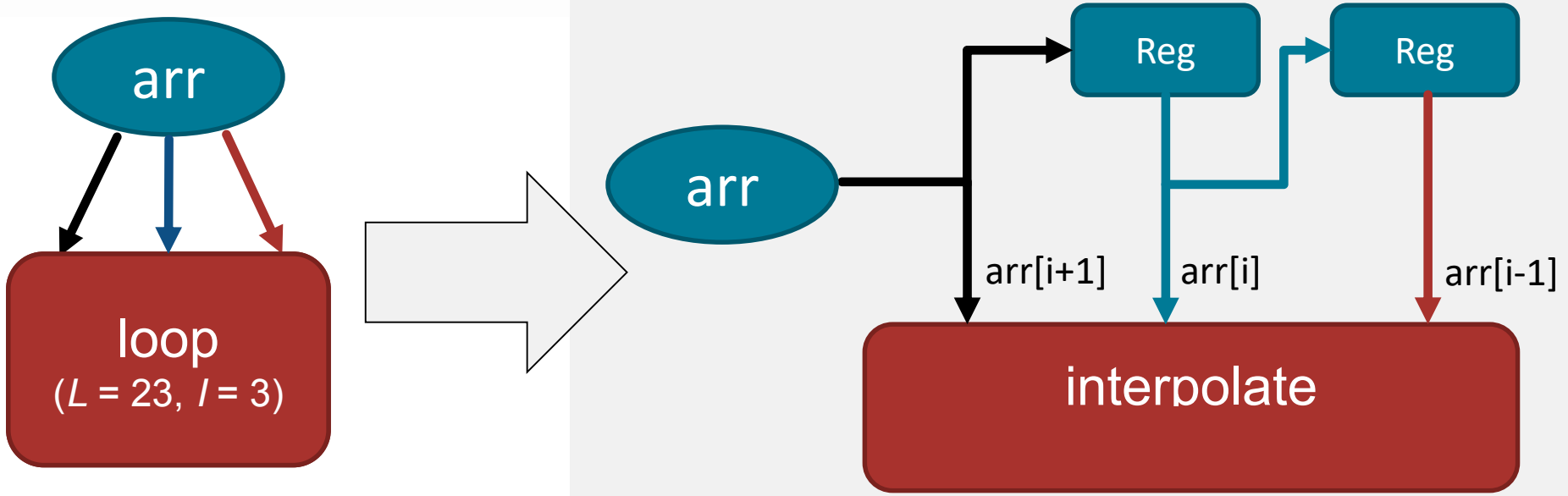
Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.

Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.



Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.

Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.

Challenge: Designing for maximum throughput for competitive performance.

Summary

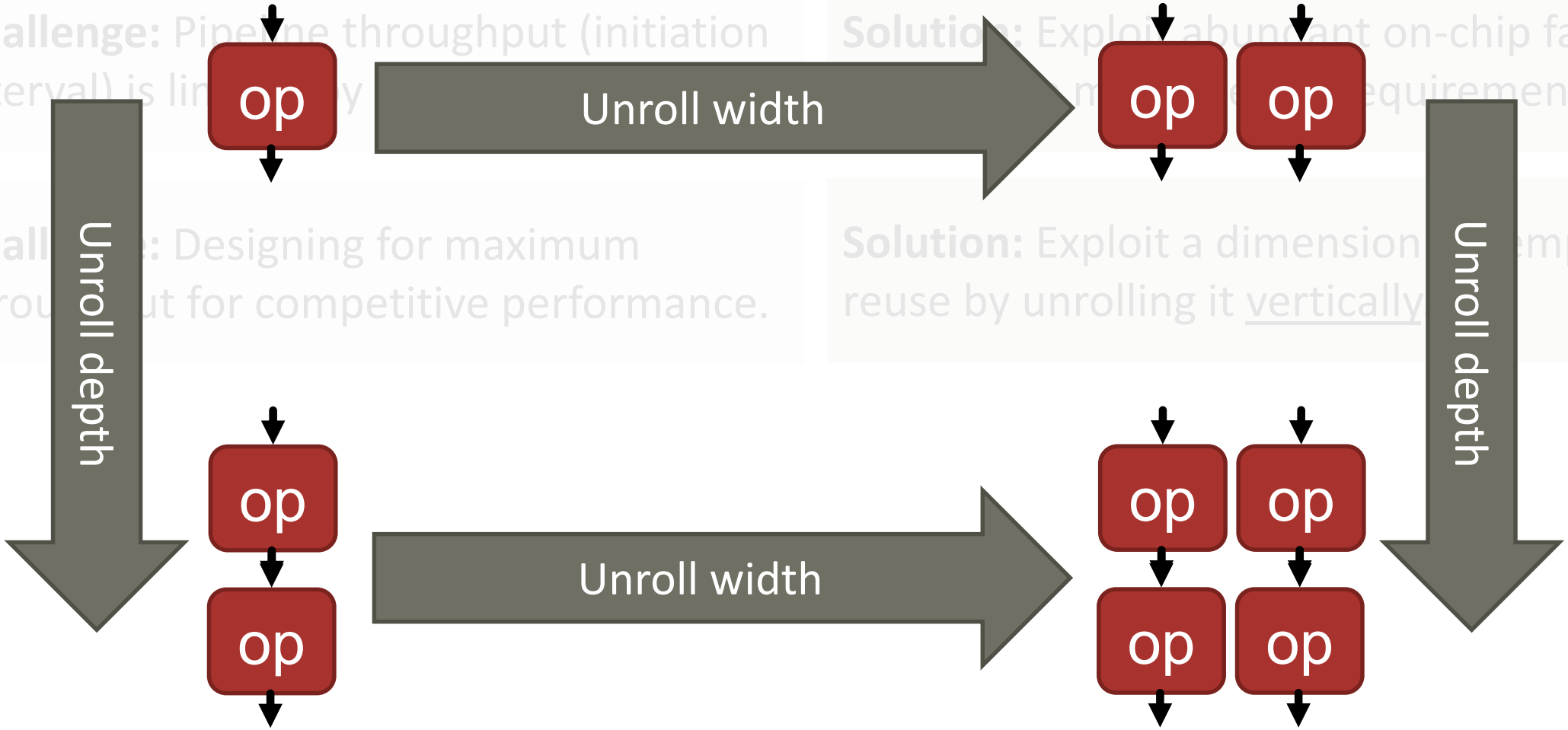
Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.

Challenge: Designing for maximum throughput for competitive performance.

Solution: Exploit a dimension of temporal reuse by unrolling it vertically.

Summary


op
Unroll width
op
op
Unroll depth
op
op
Unroll width
op
op
op
op
Unroll depth

Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.

Challenge: Designing for maximum throughput for competitive performance.

Solution: Exploit a dimension of temporal reuse by unrolling it vertically.

Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.

Challenge: Designing for maximum throughput for competitive performance.

Solution: Exploit a dimension of temporal reuse by unrolling it vertically.

Challenge: Loop-carried dependencies cause pipeline stalls that kill performance.

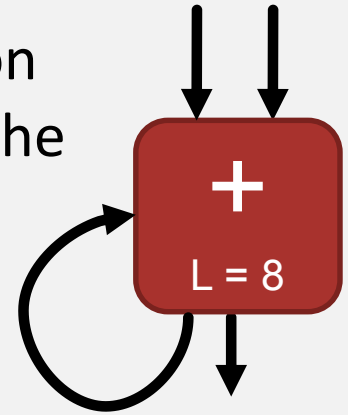
Summary

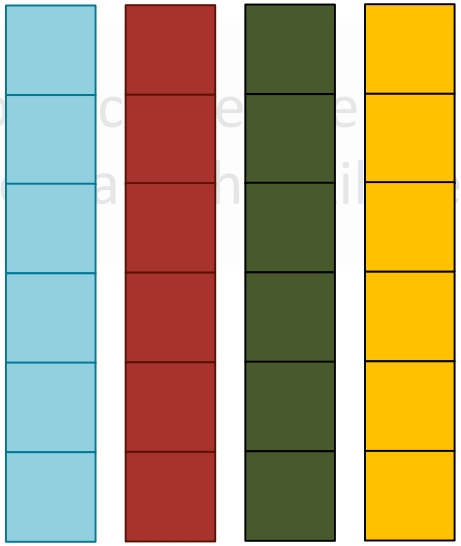
Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.	**Solution:** Exploit abundant on-chip fast memory to minimize I/O requirements.
Challenge: Designing for maximum throughput for competitive performance.	**Solution:** Exploit a dimension of temporal reuse by unrolling it vertically.
Challenge: Loop-carried dependencies cause pipeline stalls that kill performance.	**Solution:** Transpose iteration space to delay inter-iteration dependencies.

Summary

Every iteration depends on the previous:

$I = 8$





```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += stream.Pop();
  }
}
```

```
float acc[M];
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < M; ++j) {
    acc[M] += stream.Pop();
  }
}
```

Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.	**Solution:** Exploit abundant on-chip fast memory to minimize I/O requirements.
Challenge: Designing for maximum throughput for competitive performance.	**Solution:** Exploit a dimension of temporal reuse by unrolling it vertically.
Challenge: Loop-carried dependencies cause pipeline stalls that kill performance.	**Solution:** Transpose iteration space to delay inter-iteration dependencies.

Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.

Solution: Exploit abundant on-chip fast memory to minimize I/O requirements.

Challenge: Designing for maximum throughput for competitive performance.

Solution: Exploit a dimension of temporal reuse by unrolling it vertically.

Challenge: Loop-carried dependencies cause pipeline stalls that kill performance.

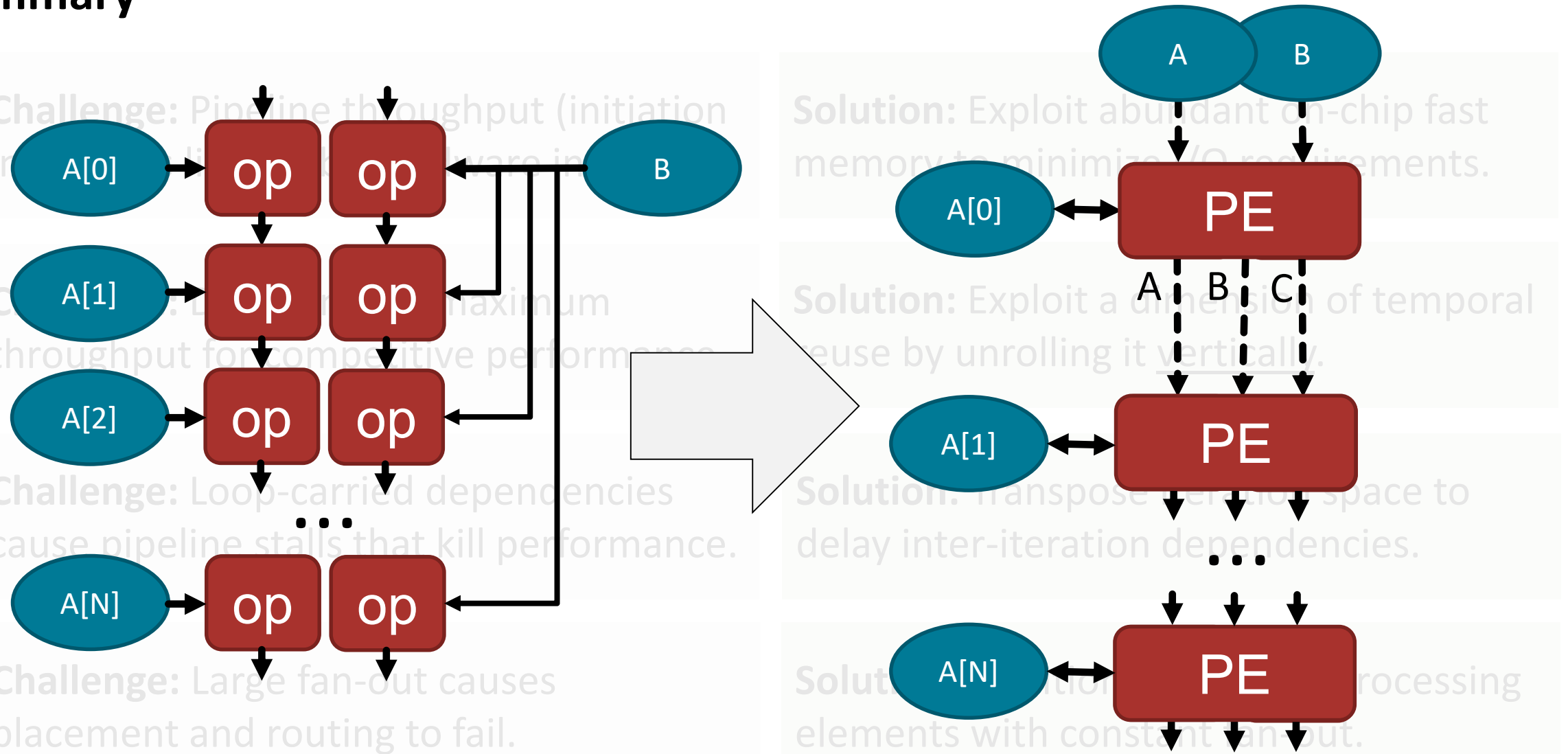
Solution: Transpose iteration space to delay inter-iteration dependencies.

Challenge: Large fan-out causes placement and routing to fail.

Summary

Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.	**Solution:** Exploit abundant on-chip fast memory to minimize I/O requirements.
Challenge: Designing for maximum throughput for competitive performance.	**Solution:** Exploit a dimension of temporal reuse by unrolling it vertically.
Challenge: Loop-carried dependencies cause pipeline stalls that kill performance.	**Solution:** Transpose iteration space to delay inter-iteration dependencies.
Challenge: Large fan-out causes placement and routing to fail.	**Solution:** Partition into async. processing elements with constant fan-out.

Summary

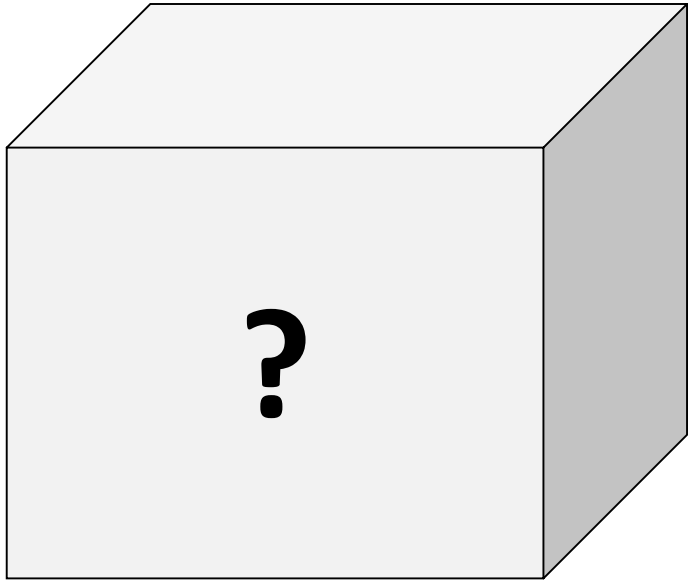


Summary

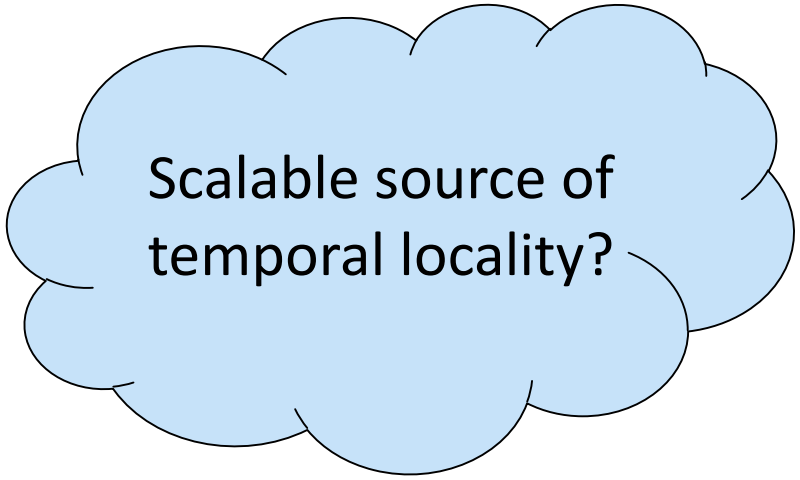
Challenge: Pipeline throughput (initiation interval) is limited by hardware interfaces.	**Solution:** Exploit abundant on-chip fast memory to minimize I/O requirements.
Challenge: Designing for maximum throughput for competitive performance.	**Solution:** Exploit a dimension of temporal reuse by unrolling it vertically.
Challenge: Loop-carried dependencies cause pipeline stalls that kill performance.	**Solution:** Transpose iteration space to delay inter-iteration dependencies.
Challenge: Large fan-out causes placement and routing to fail.	**Solution:** Partition into async. processing elements with constant fan-out.

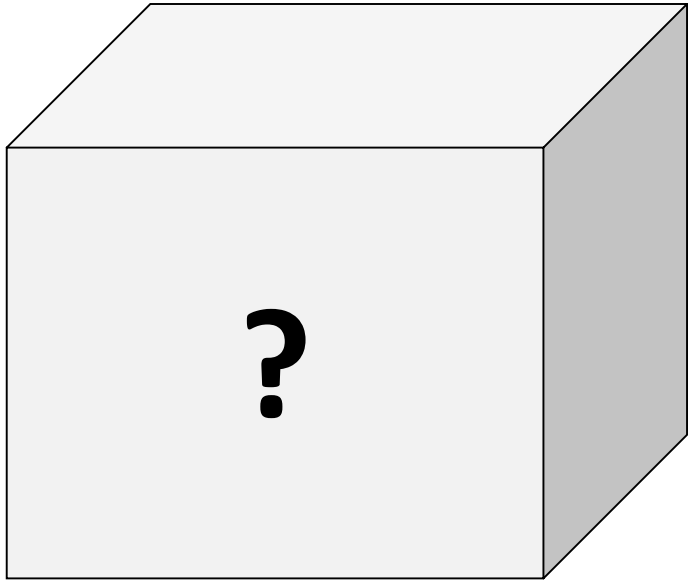
Thank you for your attention ☺

Approaching a new problem

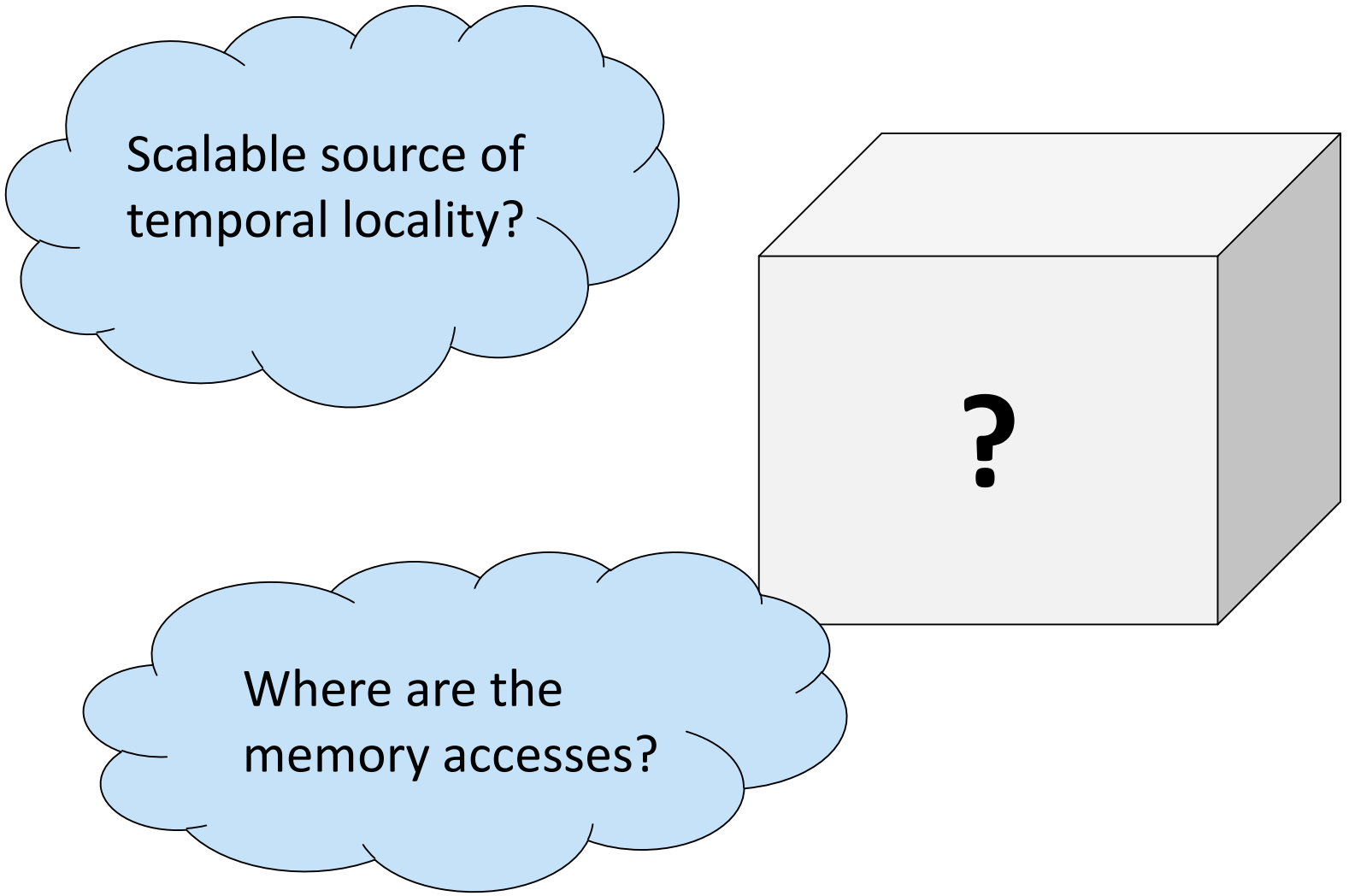


Approaching a new problem

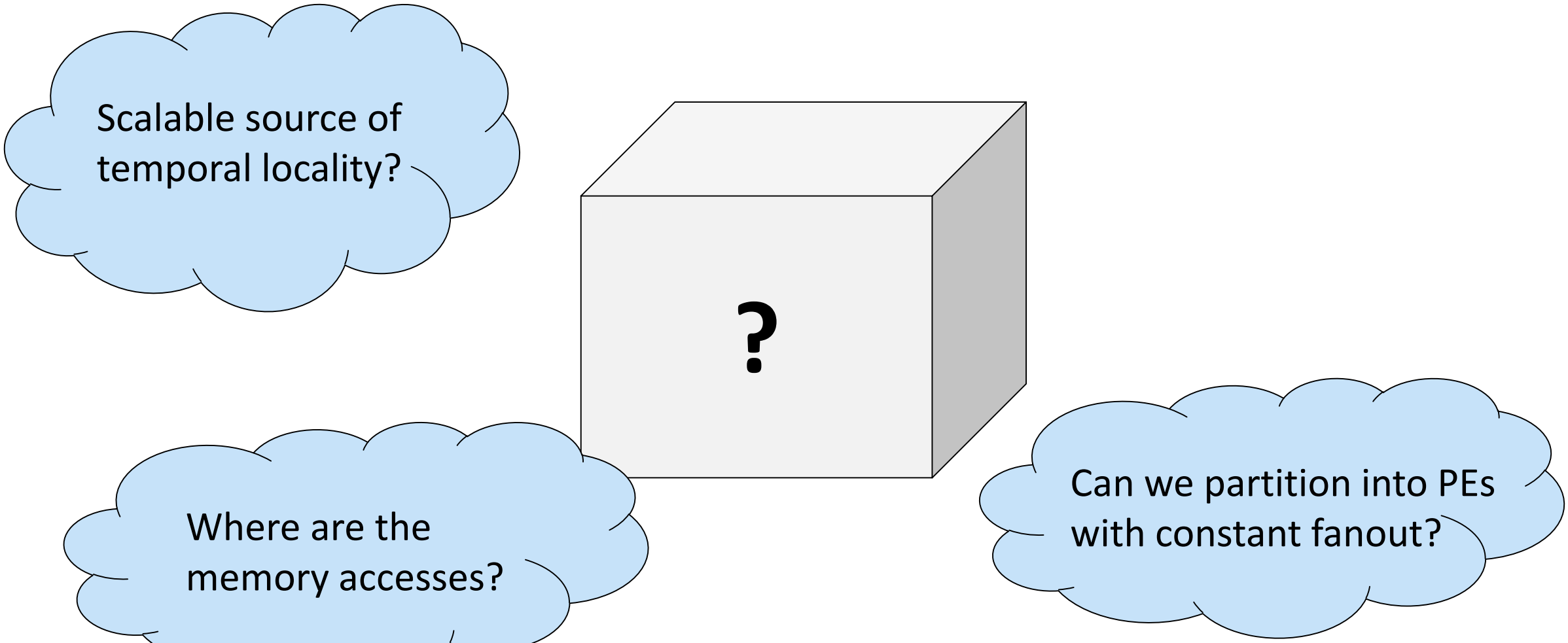




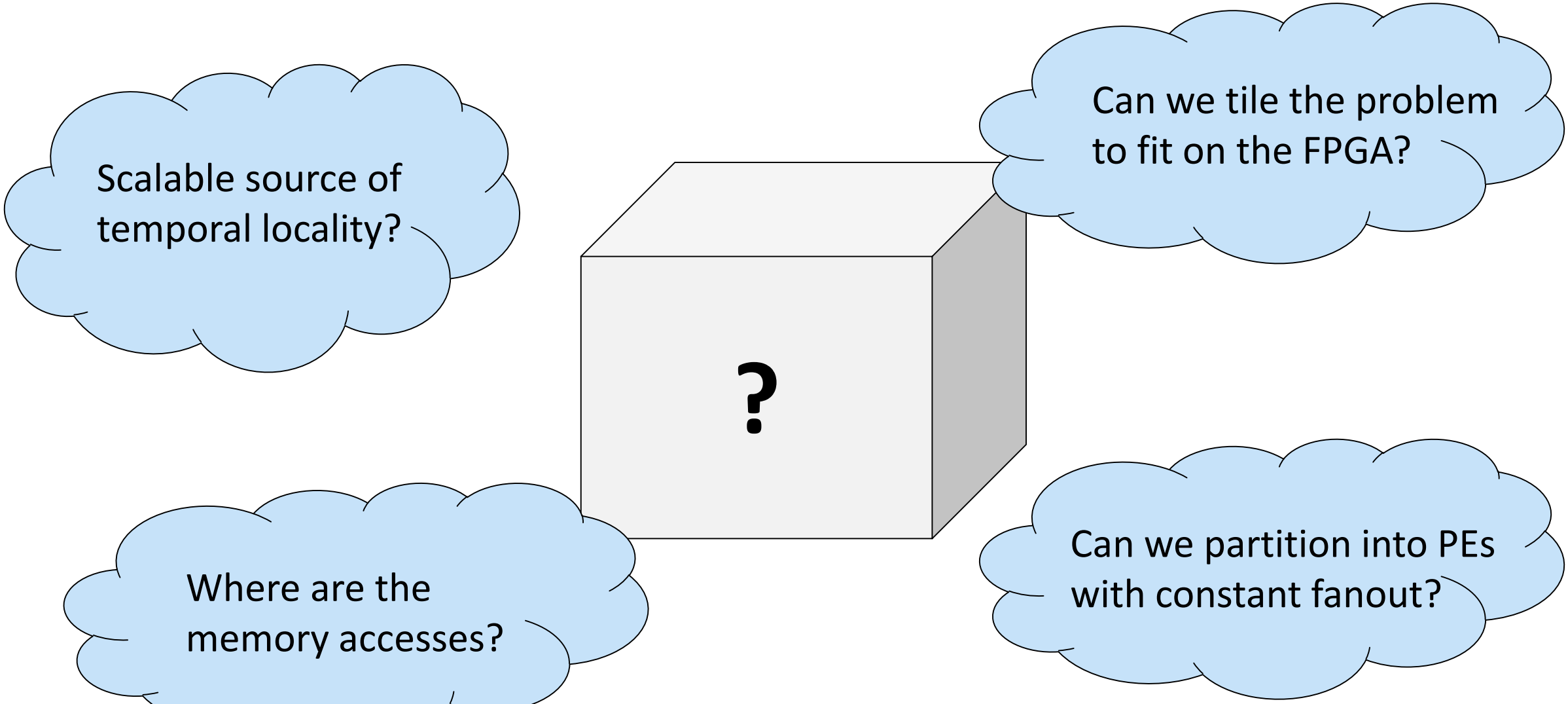
Approaching a new problem



Approaching a new problem



Approaching a new problem



When HDL should be involved

```
always @(posedge ACLK) begin
    if (~ARESETN | system_reset) begin
        write_state <= WRITE_IDLE;
        write_addr <= 0;
        start_kernel_signal <= 1'b0;
        SC_reset <= 1'b0;
        icap_wr <= 1'b0;
        host2device_wr <= 1'b0;
        host2device <= 32'h0;
        system_reset <= 1'b0;
    end
    else begin
      AWREADY <= 1'b1;
      WREADY <= 1'b0;
      BVALID <= 1'b0;
      system_reset <= 1'b0;
```

When HDL should be involved

```
always @(posedge ACLK) begin
    if (~ARESETN | system_reset) begin
        write_state <= WRITE_IDLE;
        write_addr <= 0;
        start_kernel_signal <= 1'b0;
        SC_reset <= 1'b0;
        icap_wr <= 1'b0;
        host2device_wr <= 1'b0;
        host2device <= 32'h0;
        system_reset <= 1'b0;
    end
    else begin
      AWREADY <= 1'b1;
      WREADY <= 1'b0;
      BVALID <= 1'b0;
      system_reset <= 1'b0;
```

Latency critical optimizations

When HDL should be involved

```
always @(posedge ACLK) begin
    if (~ARESETN | system_reset) begin
        write_state <= WRITE_IDLE;
        write_addr <= 0;
        start_kernel_signal <= 1'b0;
        SC_reset <= 1'b0;
        icap_wr <= 1'b0;
        host2device_wr <= 1'b0;
        host2device <= 32'h0;
        system_reset <= 1'b0;
    end
    else begin
      AWREADY <= 1'b1;
      WREADY <= 1'b0;
      BVALID <= 1'b0;
      system_reset <= 1'b0;
```

Latency critical optimizations

Interfacing

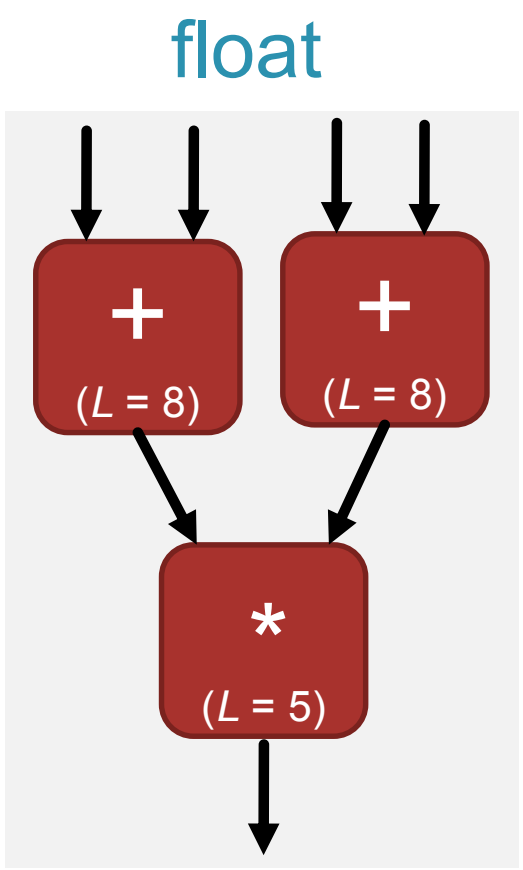
When HDL should be involved

```
always @(posedge ACLK) begin
    if (~ARESETN | system_reset) begin
        write_state <= WRITE_IDLE;
        write_addr <= 0;
        start_kernel_signal <= 1'b0;
        SC_reset <= 1'b0;
        icap_wr <= 1'b0;
        host2device_wr <= 1'b0;
        host2device <= 32'h0;
        system_reset <= 1'b0;
    end
    else begin
      AWREADY <= 1'b1;
      WREADY <= 1'b0;
      BVALID <= 1'b0;
      system_reset <= 1'b0;
```

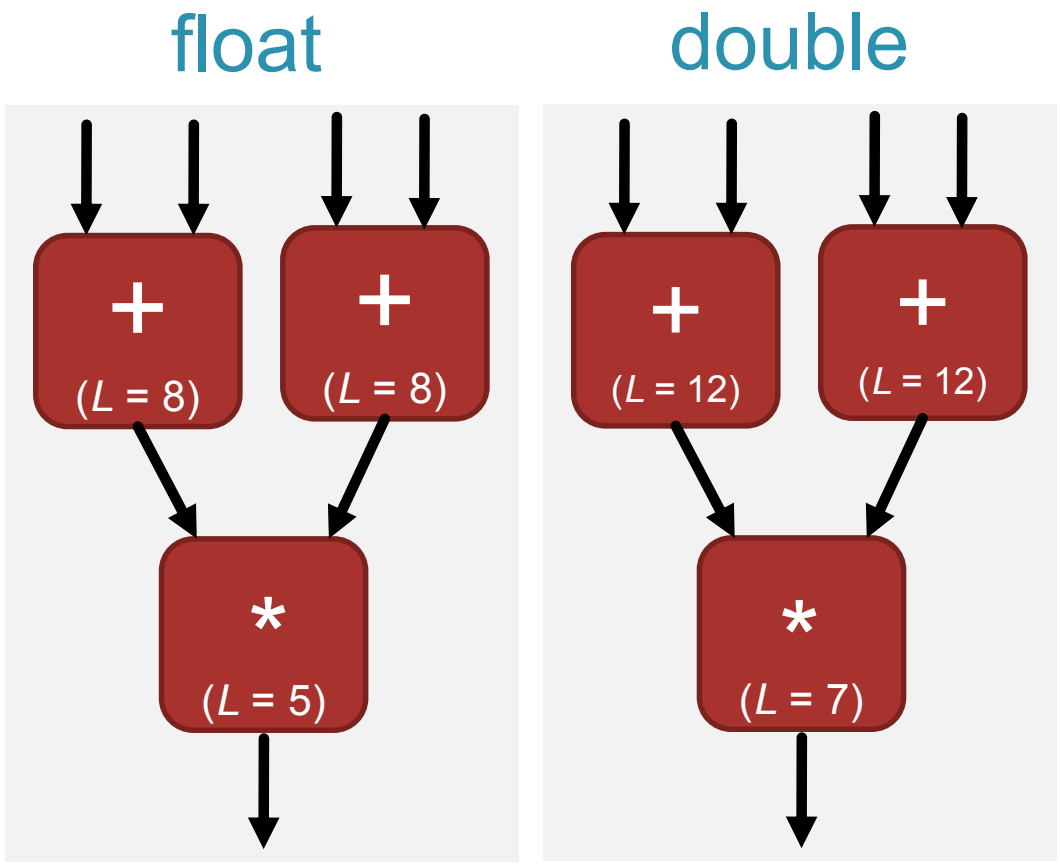
HDL and HLS can (and do often) happily co-exist!

```
for (int i = 1; i < N - 1; ++i) {
  for (int j = 0; j < M; ++j) {
    #pragma HLS PIPELINE II=1


    const auto above = above_buffer.read();
    const auto center = center_buffer.read();
                                mory_in[(i + 1)*M + j];

                              = 0.3333;

        factor * (above + center + below);

    above_buffer.write(center);
    center_buffer.write(below);

    memory_out[i * M + j] = average;
  }
}
```

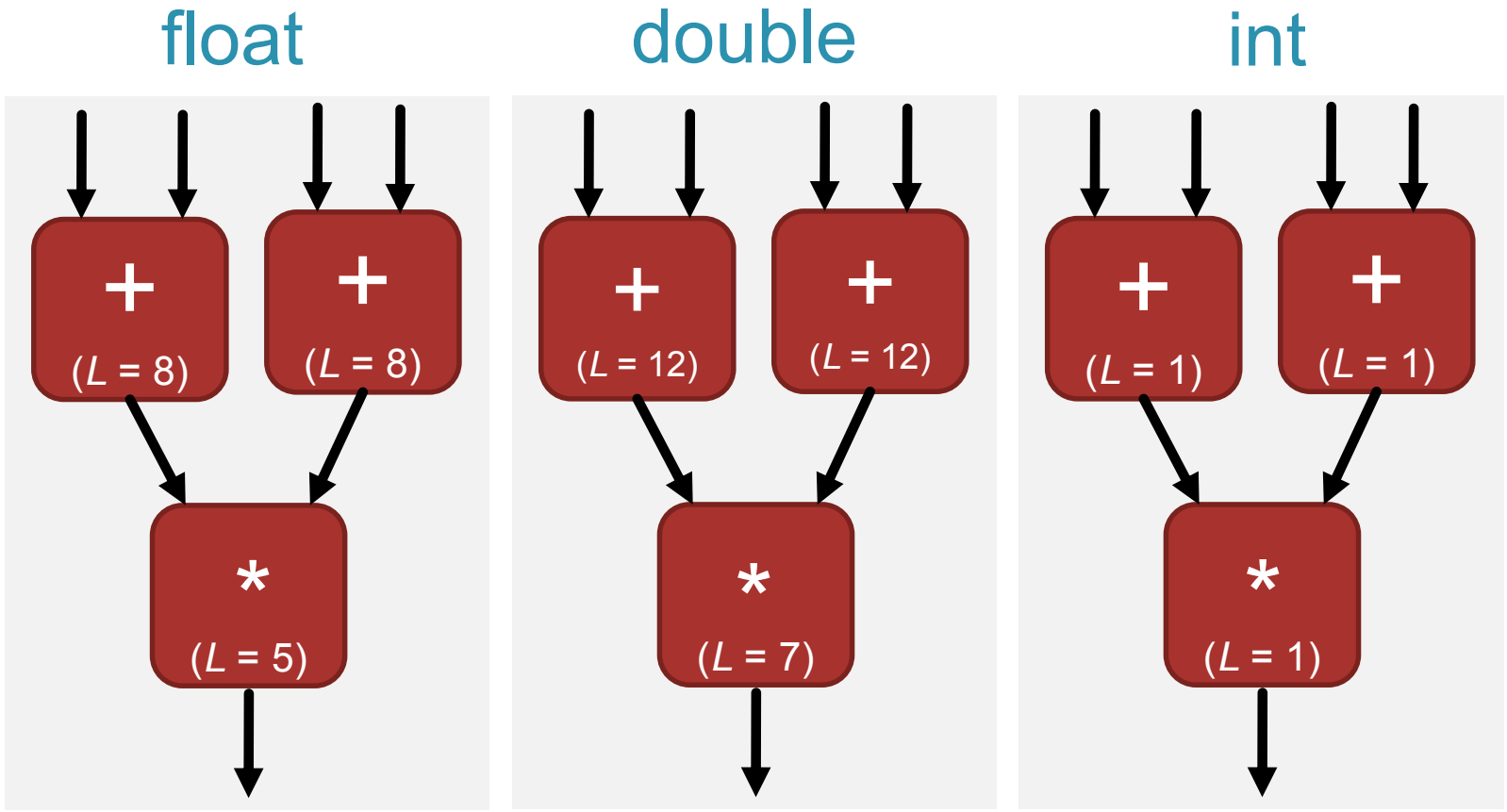
Types on FPGA



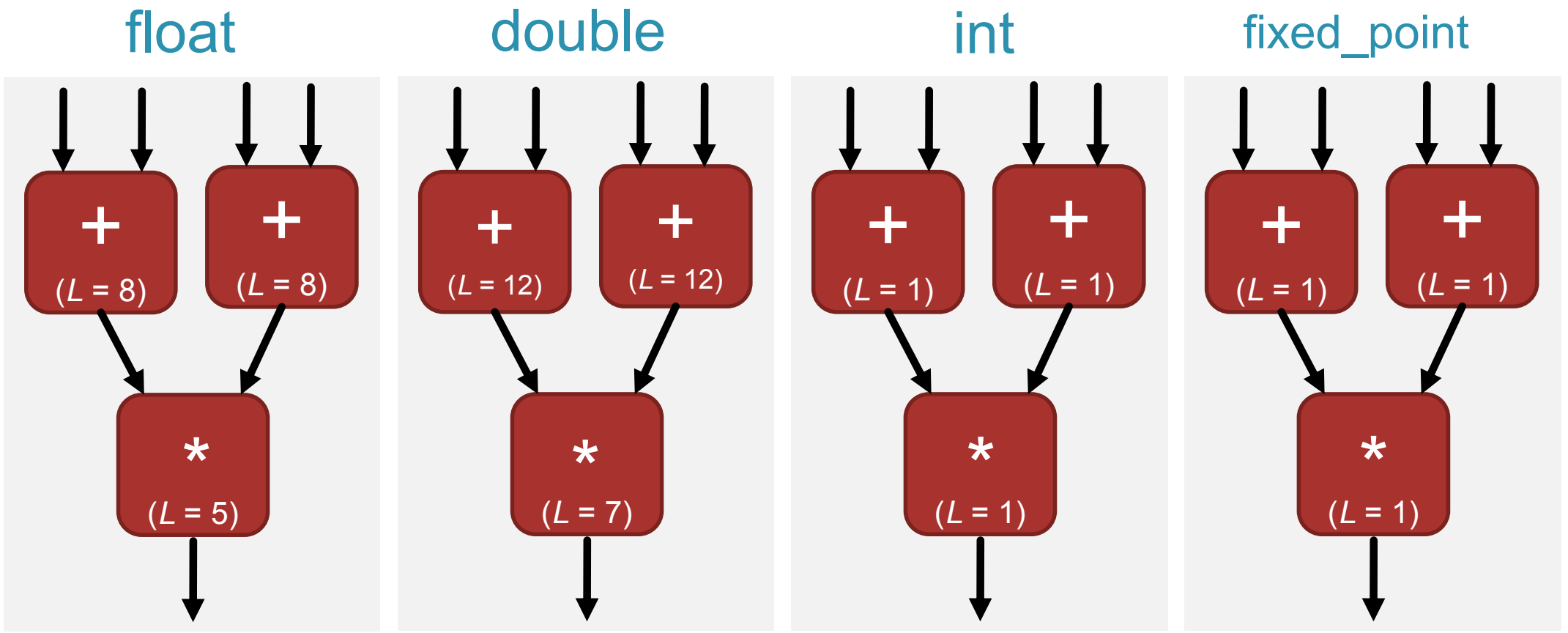
Types on FPGA



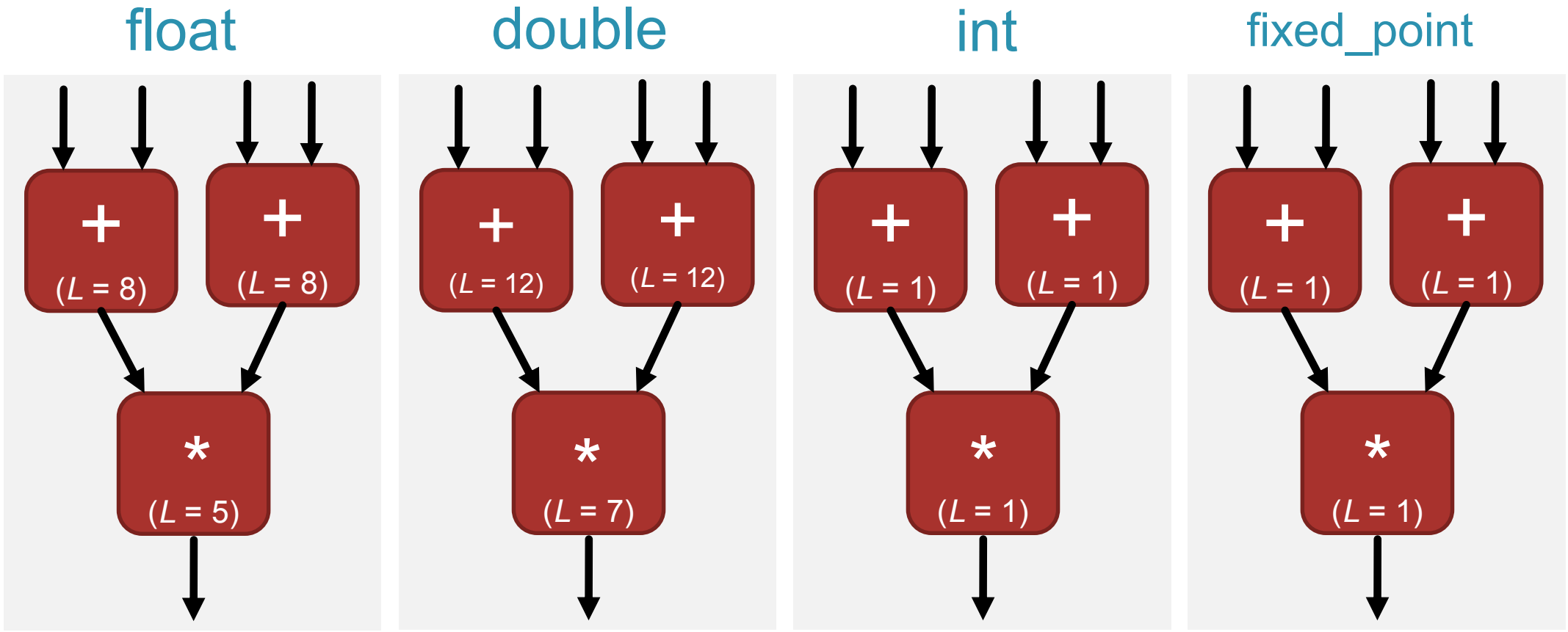
Types on FPGA



Types on FPGA



Types on FPGA



Same concepts, different latencies (some problems go away at $L = 1$).