ECE 327/627
Digital Hardware Systems

Lecture 10: FIFO-Based Dataflow Design

Andrew Boutros

andrew.boutros@uwaterloo.ca

Some Logistics

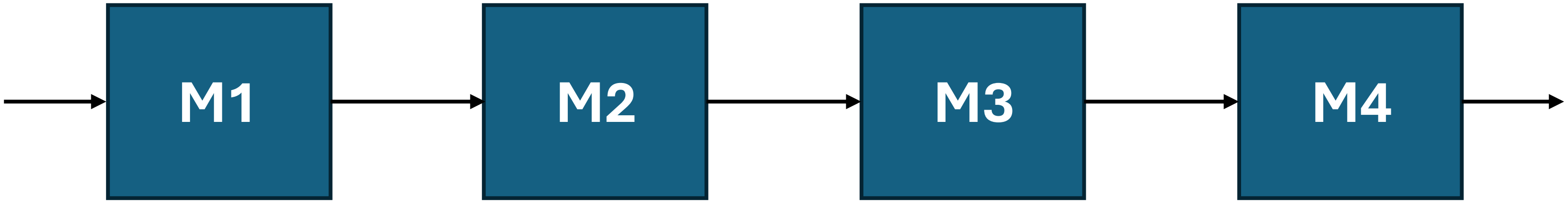
- Lab 3 was released on Friday
  - Worth **8%** of the ECE 327 total course grade
  - Worth **4%** of the ECE 327 total course grade
  - Deadline **March 6 @ 11:59 pm**
- ECE 627 Course Project handout posted last night
  - Project selection and approval due in a week (**Feb 16 @ 11:59 pm**)
- Lab 1 grades were released yesterday
  - Class average is **89%**
- Lab 2 individual assessment in this week's lab sessions
  - Worth **40%** of the lab 2 grade
  - Special schedule due to sharing the quiz room with another course – check Piazza and be on time
- You will not be tested on this lecture's content in the midterm
- No new content next lecture → midterm review

In the Previous Lecture …

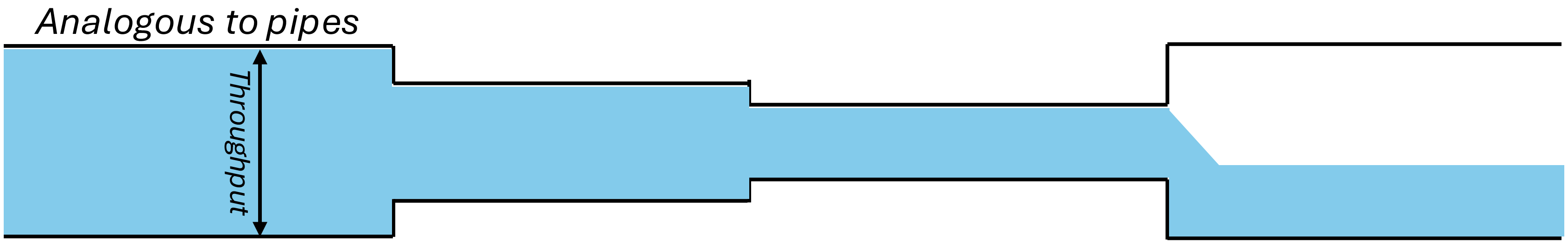
- Latency-insensitive design
  - Wire delays are becoming problematic in newer process technologies
  - Pipelining wires is not very straightforward if modules are latency-sensitive (i.e., expect inputs exactly at specific cycles)
  - Ready-valid handshakes
    - Sender produces data tagged with a valid bit
    - Receiver indicates whether ready to accept data or not
    - Transaction happens when valid & ready are both asserted
    - Regardless when that happens, functionality doesn’t break
  - This allows pipelining data & valid signals, but ready must remain combinational to freeze the entire pipeline if not ready → did not solve the wire delay issue
  - Simple pipeline regs are not enough → Relay Stations (Skid Buffers)
    - Pipeline ready signal going in opposite direction
    - Have auxiliary registers to hold 1 extra piece of data until ready is communicated to the upstream module

Throughput Balancing Revisited

- Many designs in practice have several cascaded modules processing data that flows from one module to the next → streaming architectures

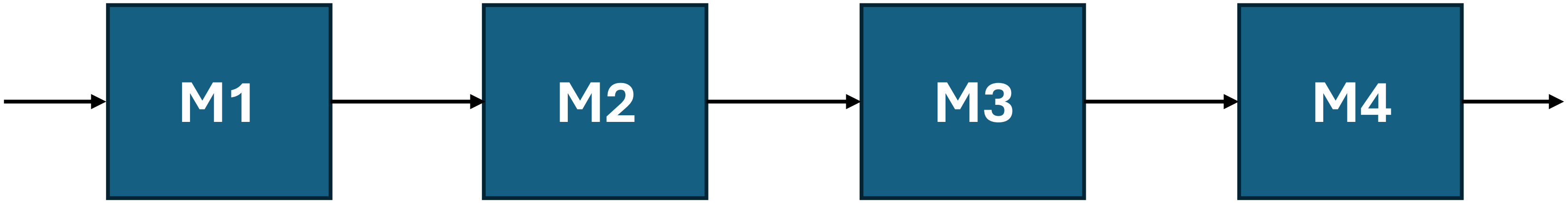


Throughput of the entire structure = Lowest throughput of all stages



Throughput Balancing Revisited

- Many designs in practice have several cascaded modules processing data that flows from one module to the next → streaming architectures



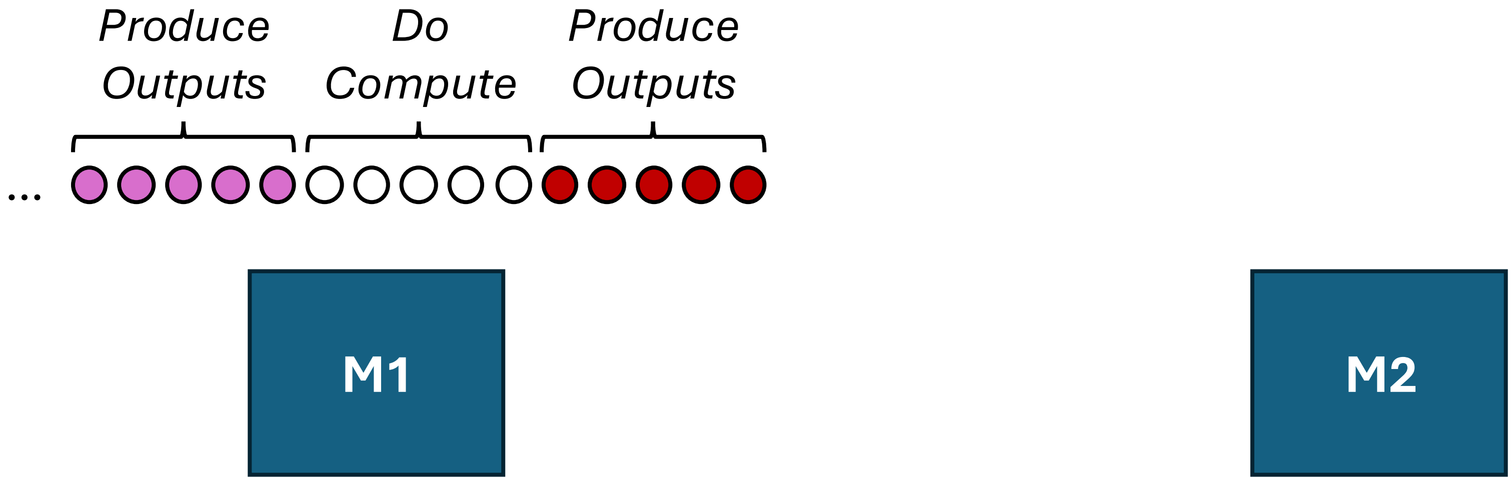
Throughput of the entire structure = Lowest throughput of all stages

Ideally, want throughput of all stages of the structure to be balanced

Throughput Balancing Revisited

Throughput Balancing Revisited

Produces 5 elements every 10 cycles!



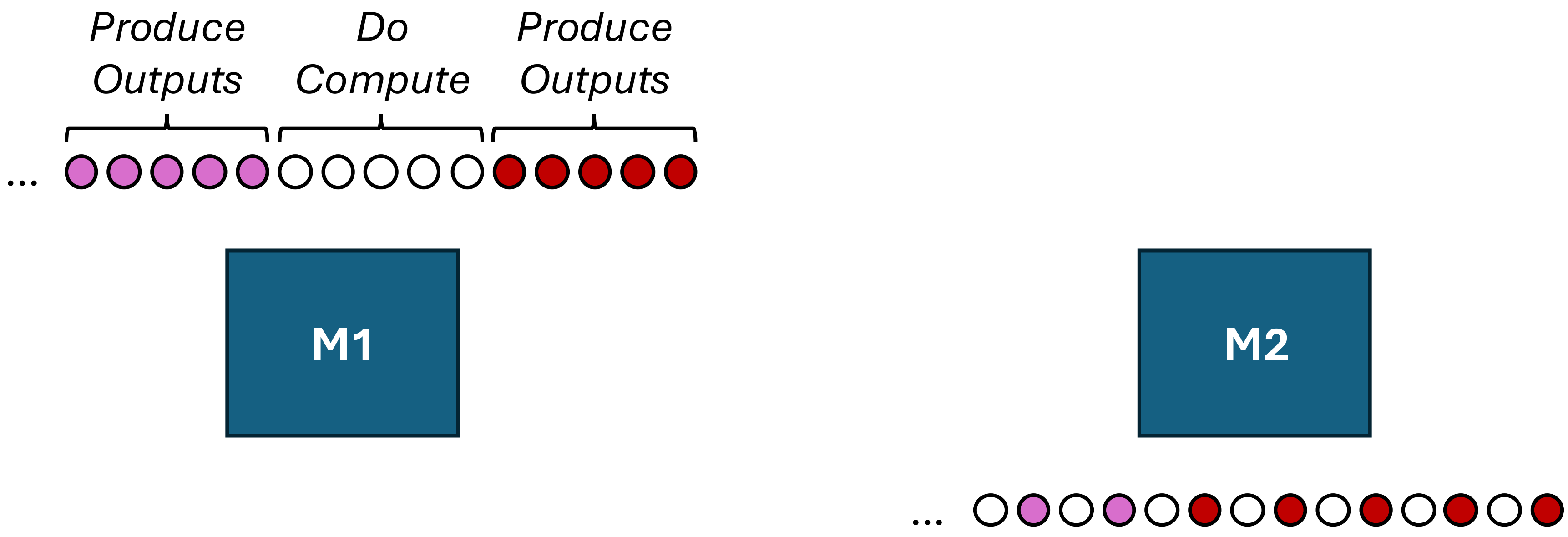
Produce Outputs

Do Compute

Produce Outputs

Throughput Balancing Revisited

Produces 5 elements every 10 cycles!



Produce Outputs

Do Compute

Produce Outputs

M1

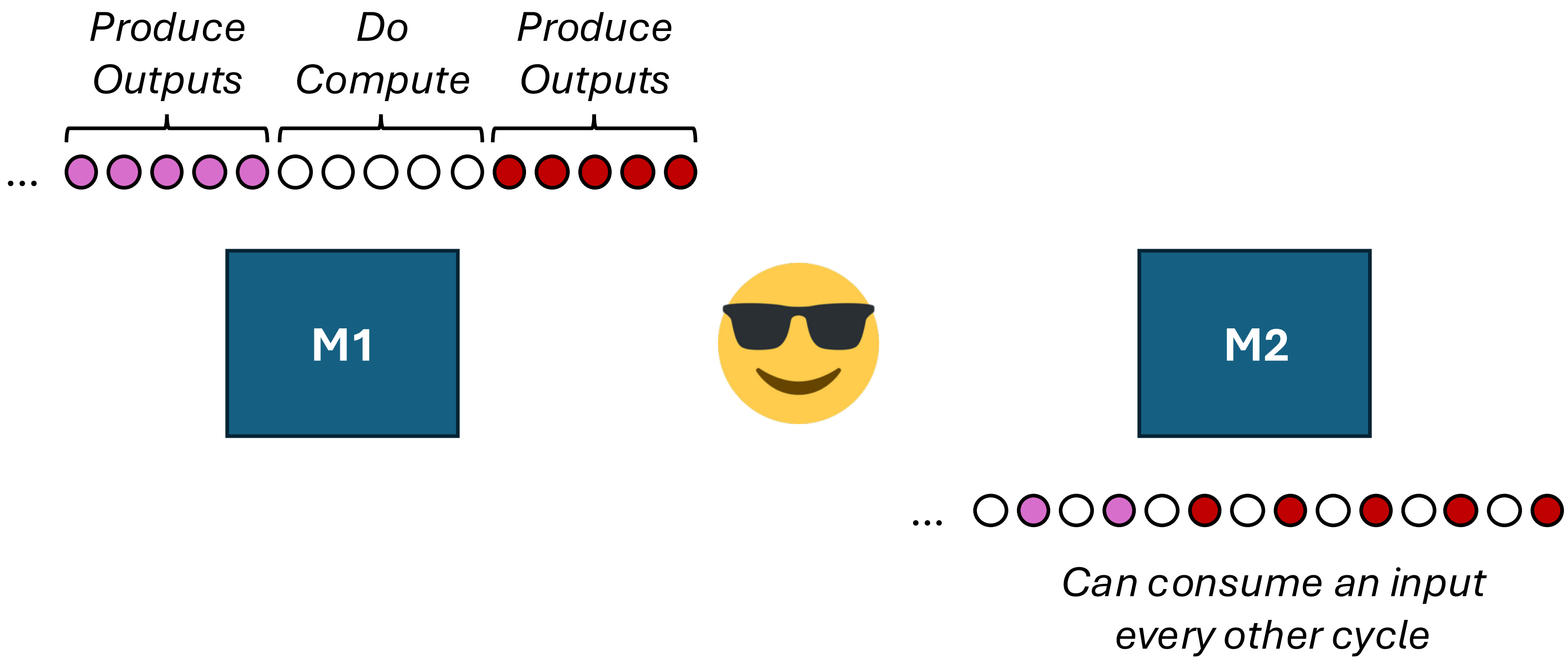
M2

Can consume an input every other cycle

Consumes 5 elements every 10 cycles!

Throughput Balancing Revisited

Produces 5 elements every 10 cycles!



Produce Outputs

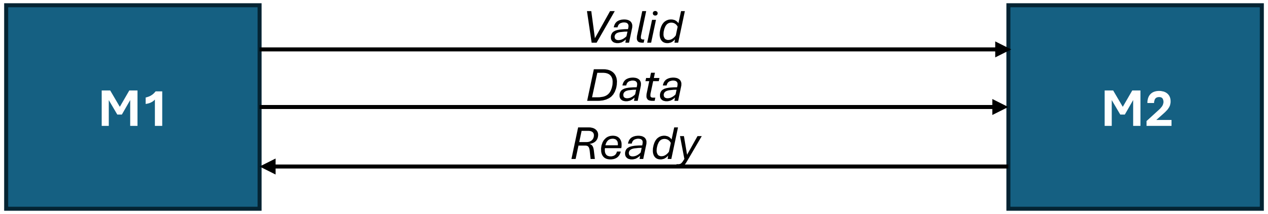
Do Compute

Produce Outputs

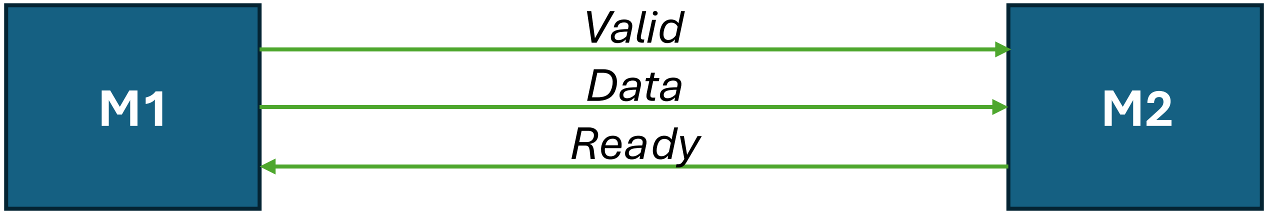
Can consume an input every other cycle

Consumes 5 elements every 10 cycles!

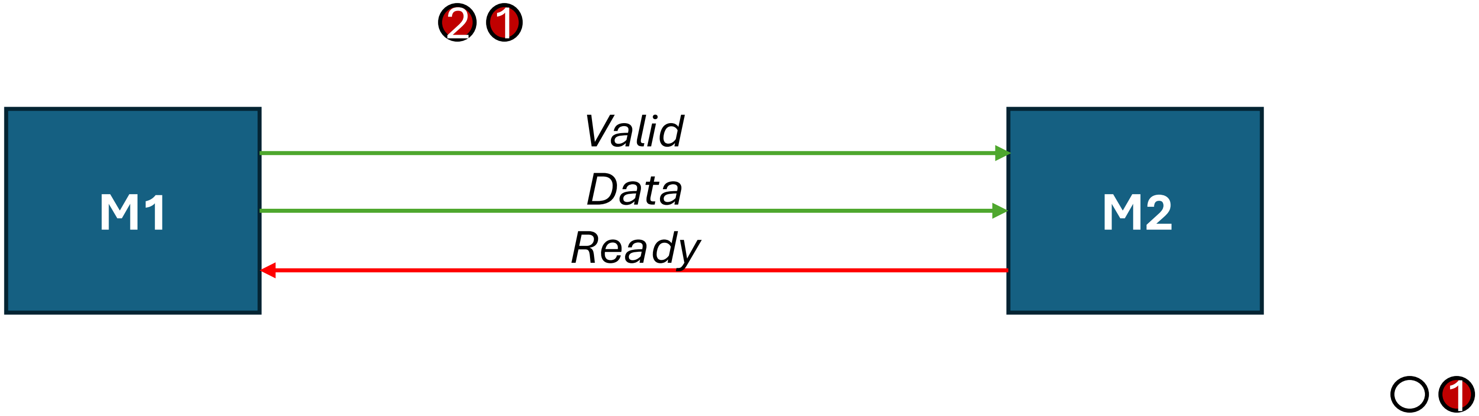
Throughput Balancing Revisited



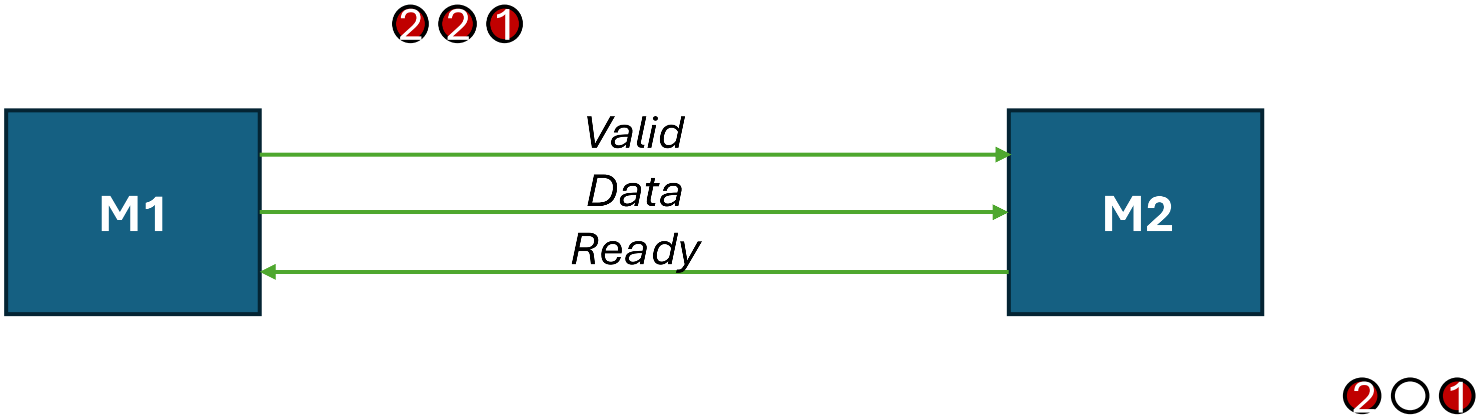
Throughput Balancing Revisited



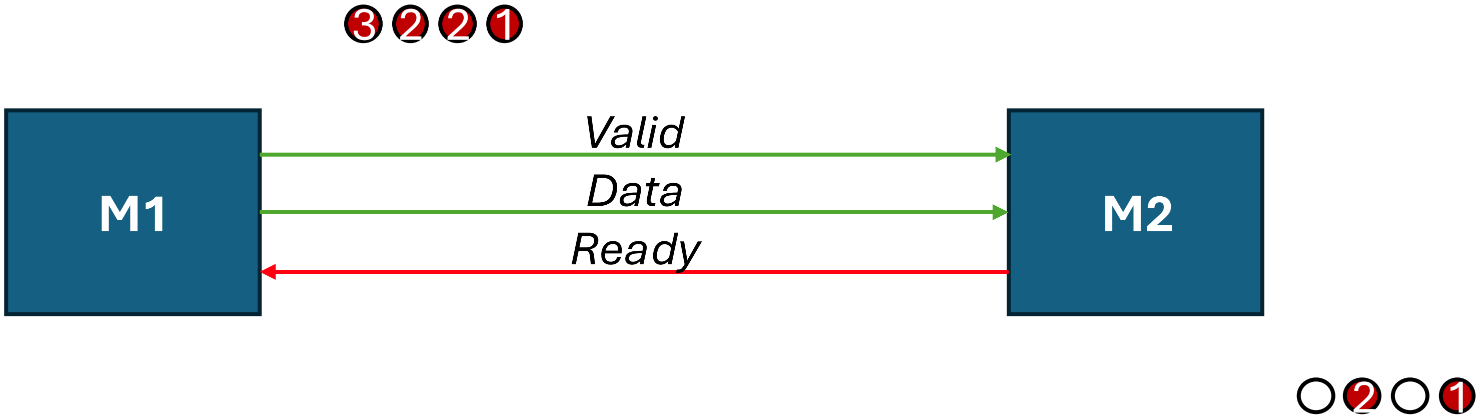
Throughput Balancing Revisited



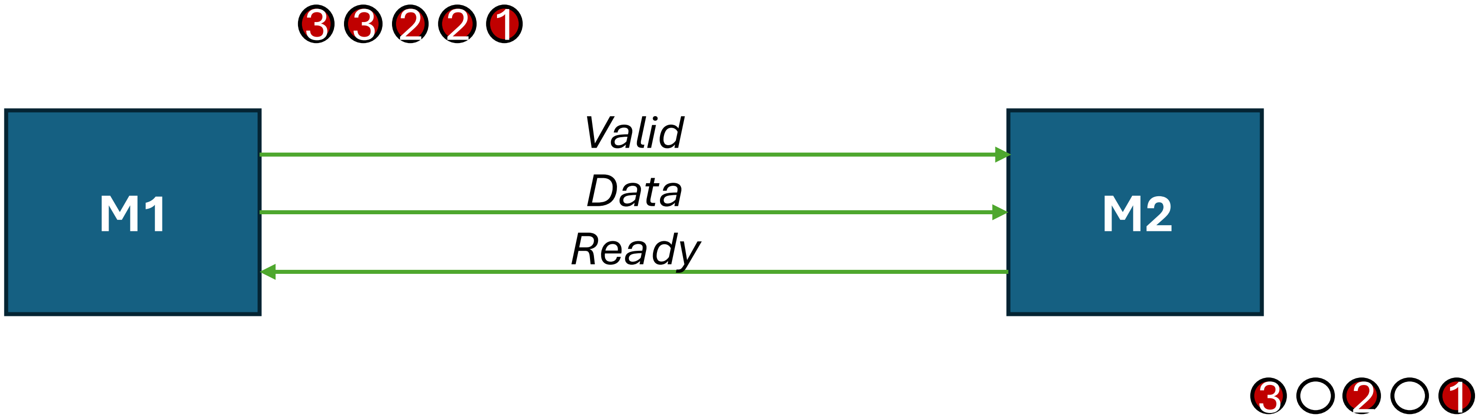
Throughput Balancing Revisited



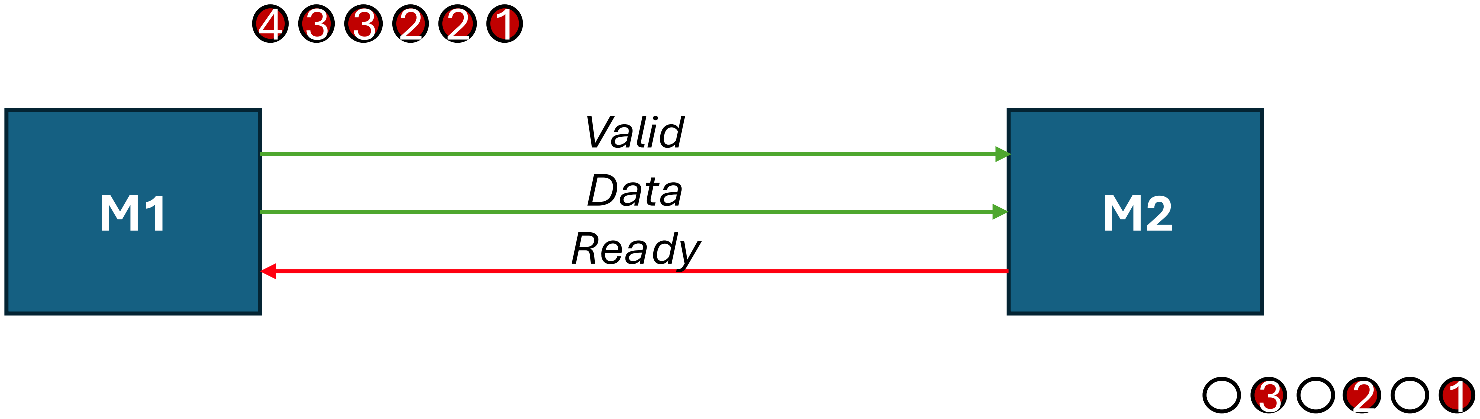
Throughput Balancing Revisited



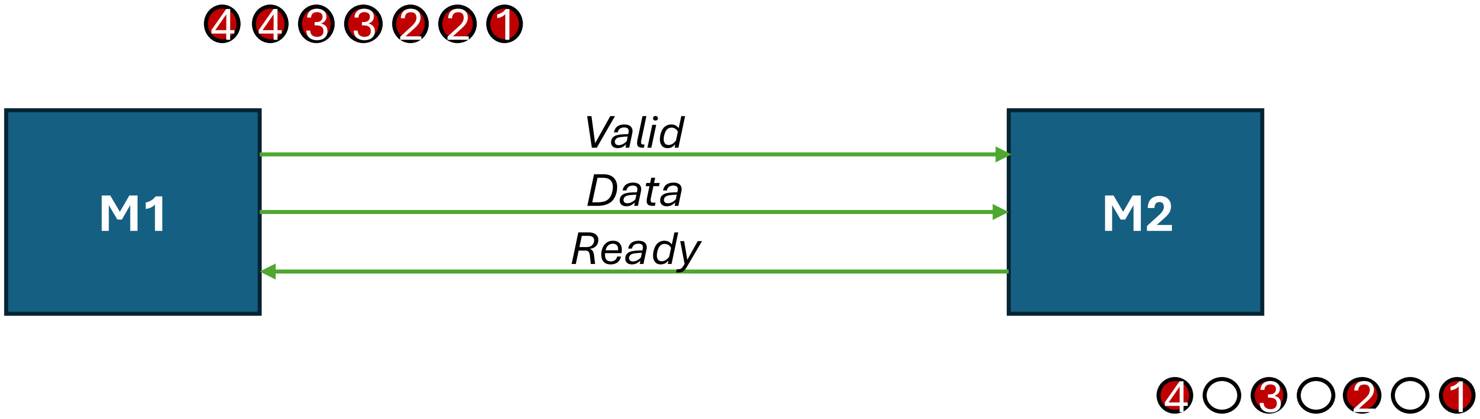
Throughput Balancing Revisited



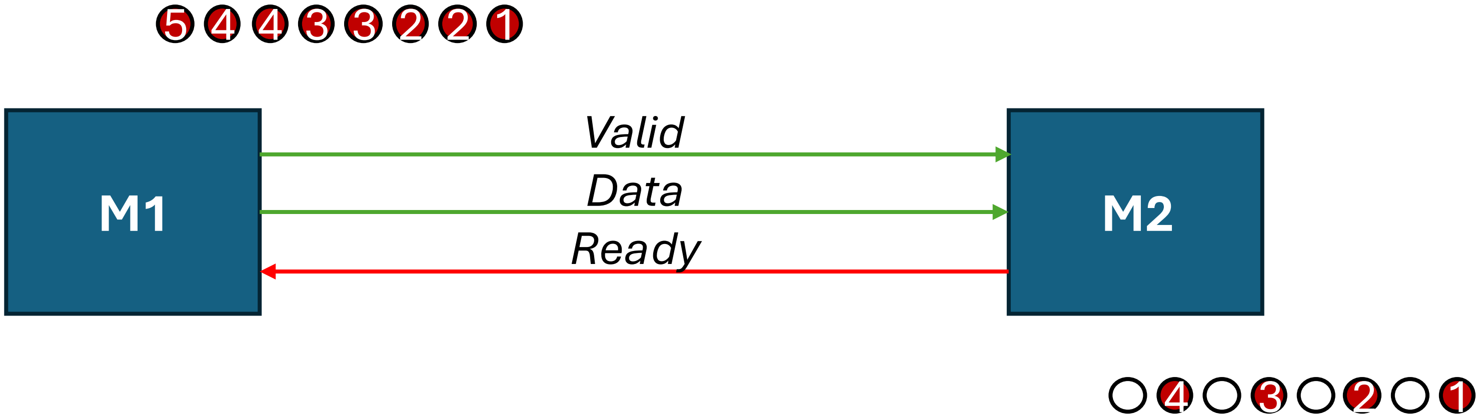
Throughput Balancing Revisited



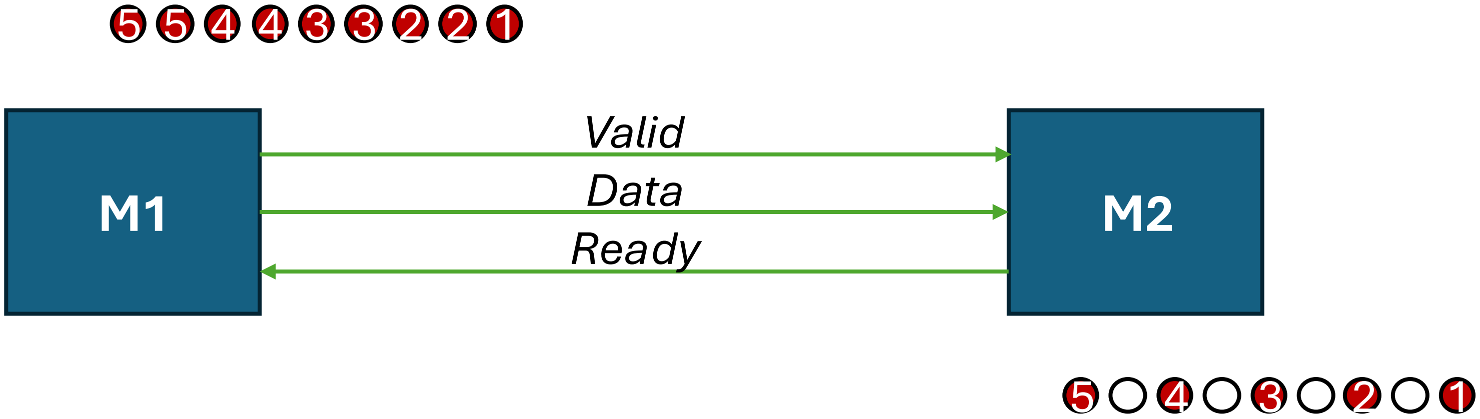
Throughput Balancing Revisited



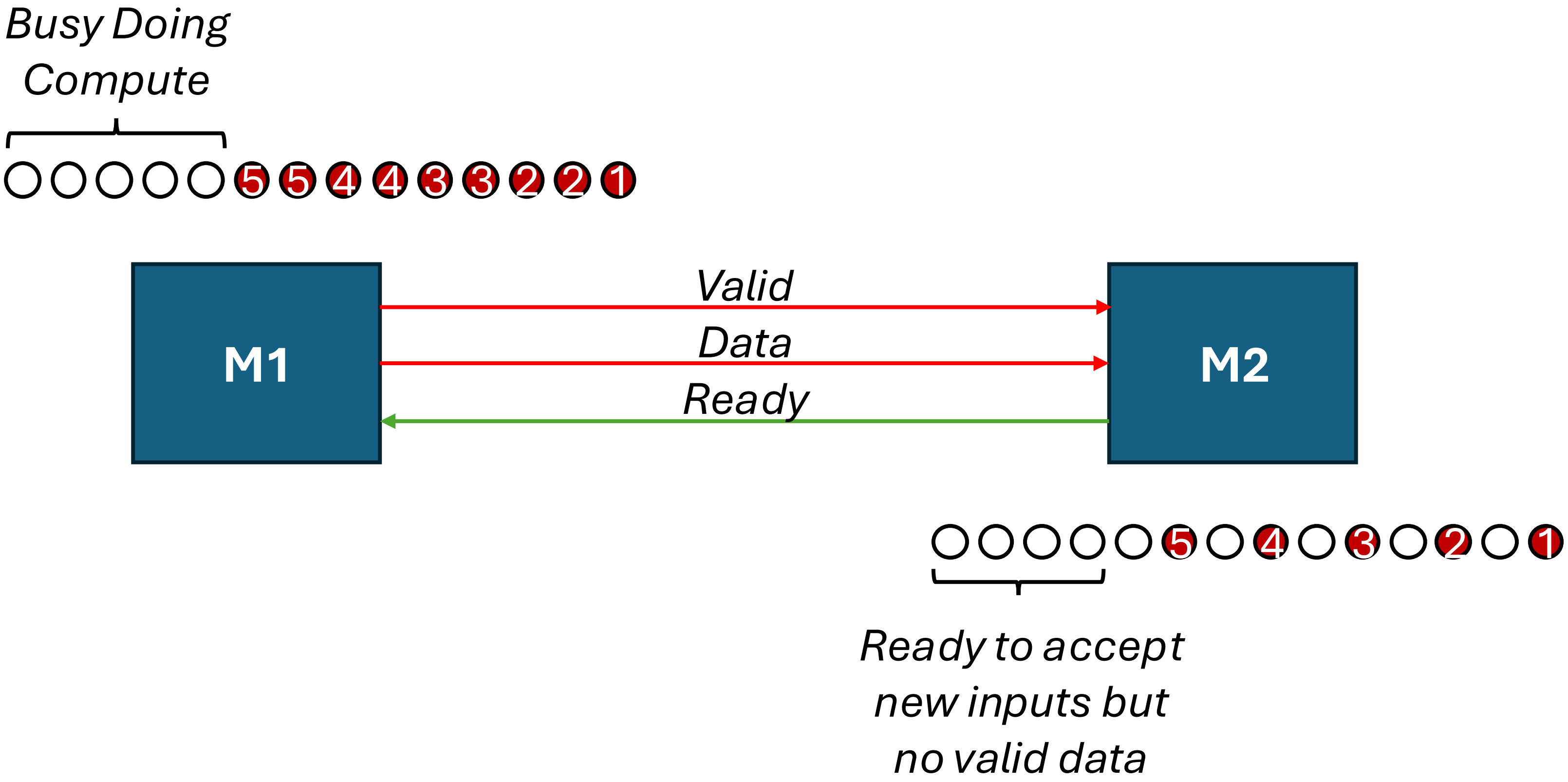
Throughput Balancing Revisited



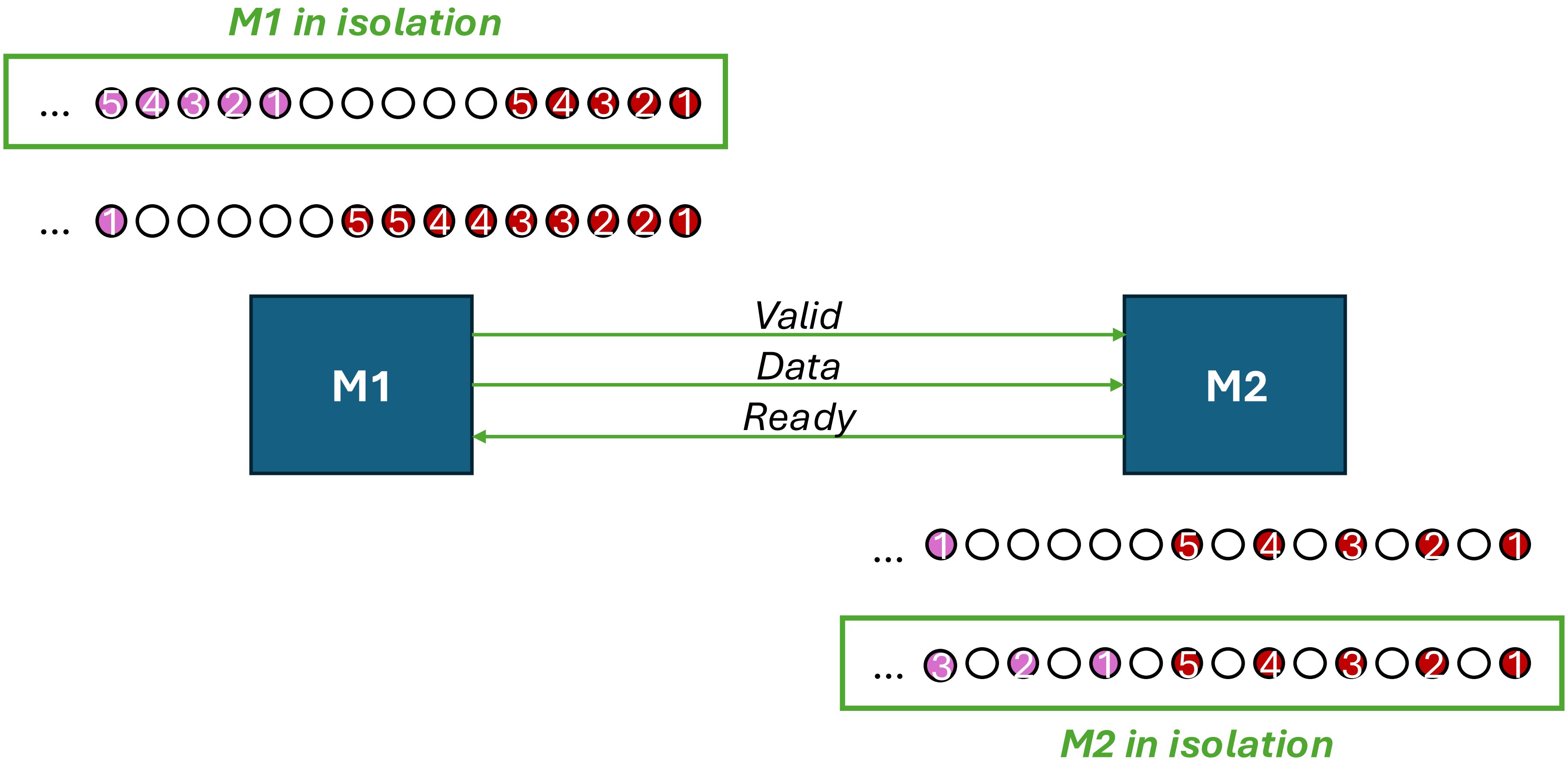
Throughput Balancing Revisited



Throughput Balancing Revisited



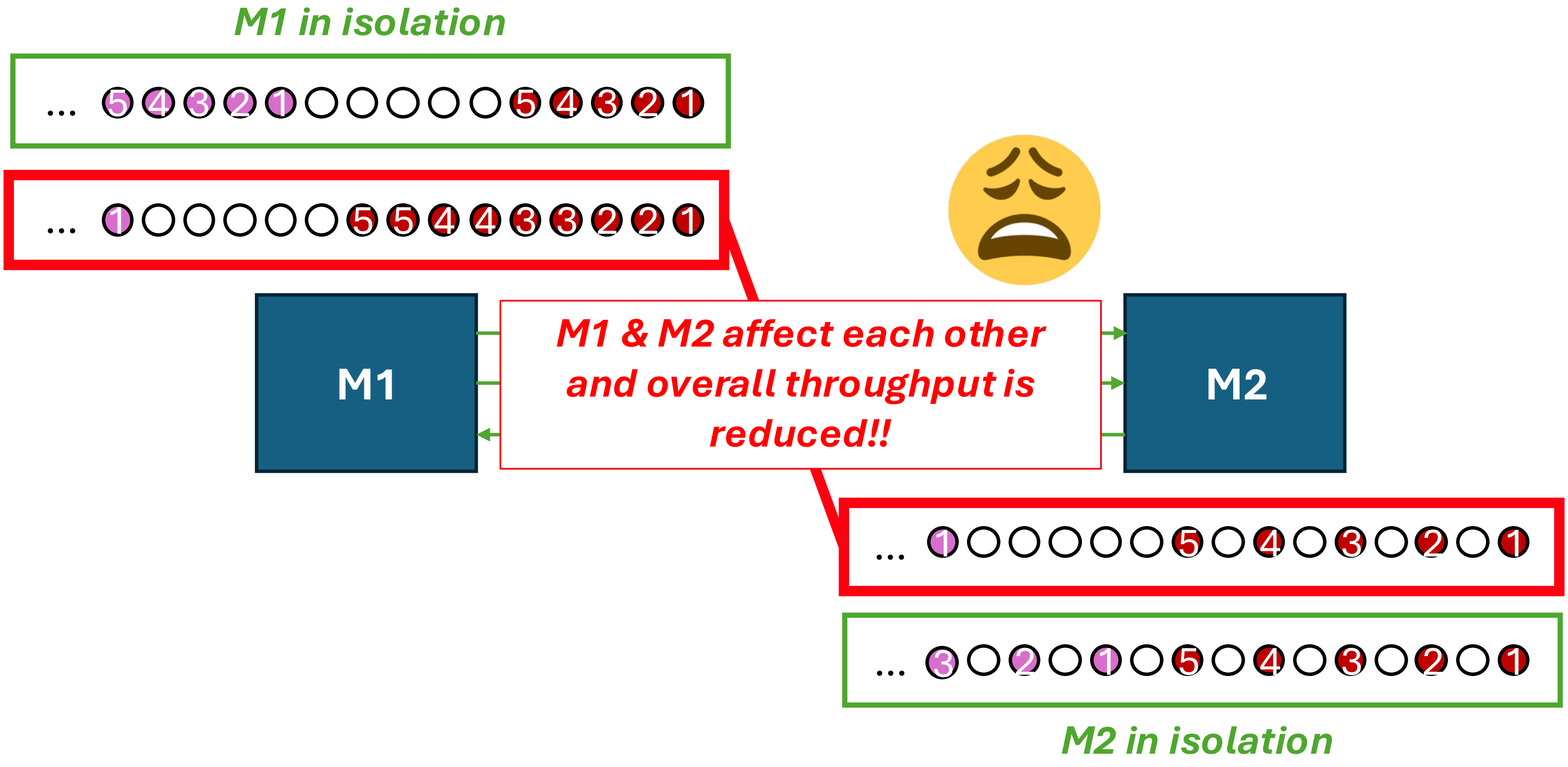
Throughput Balancing Revisited



M1 in isolation

M2 in isolation

Throughput Balancing Revisited

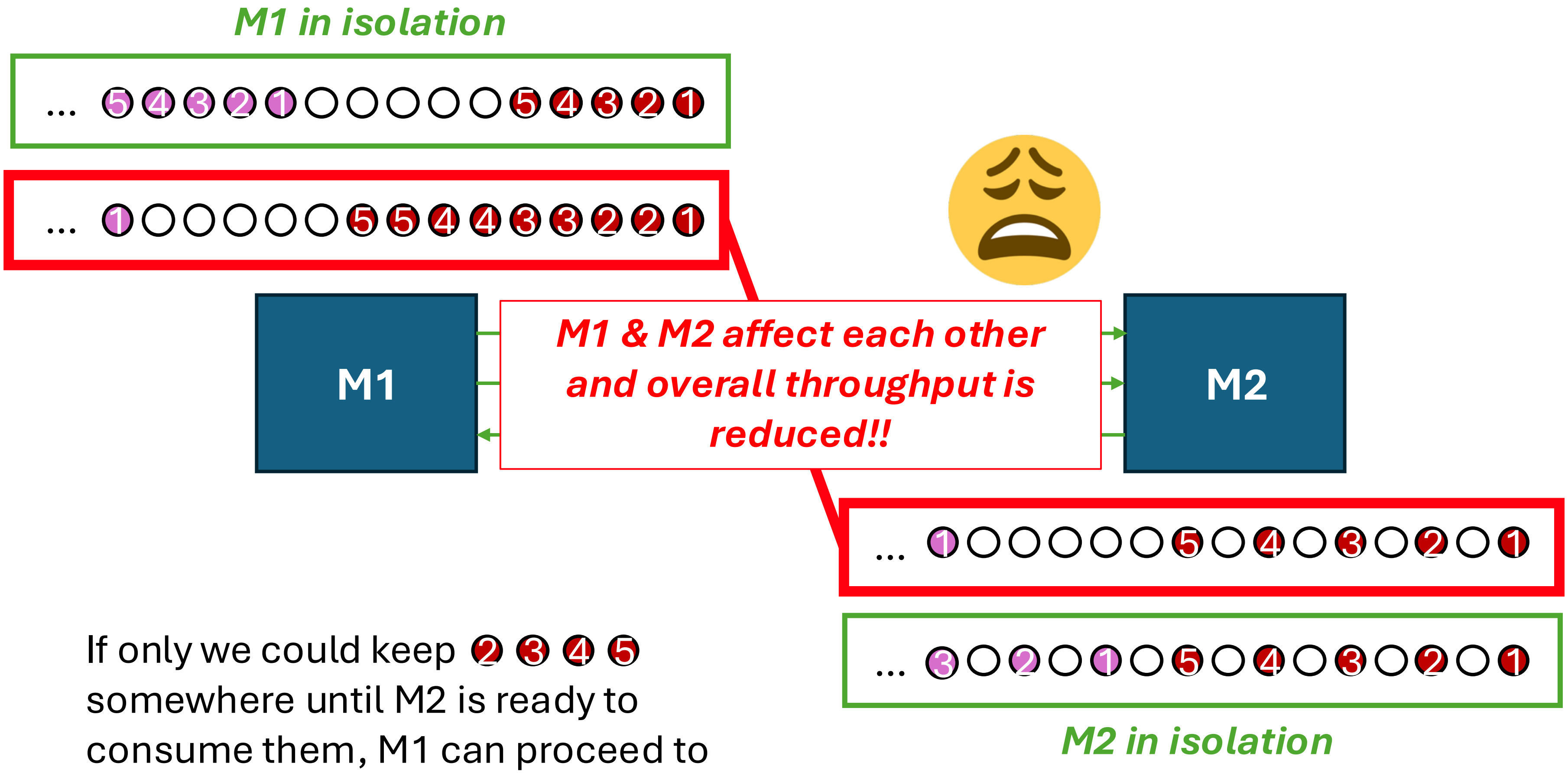


M1 in isolation

M1 & M2 affect each other and overall throughput is reduced!!

M2 in isolation

Throughput Balancing Revisited

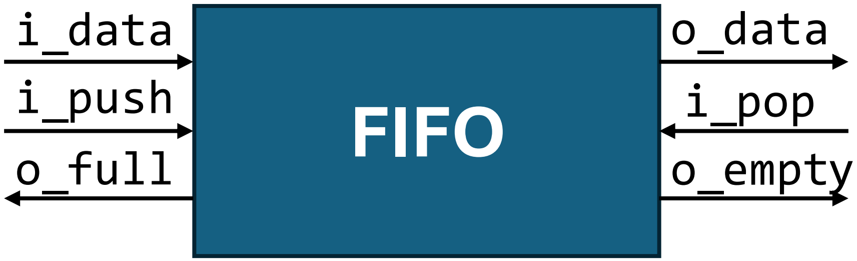


M1 & M2 affect each other and overall throughput is reduced!!

If only we could keep ② ③ ④ ⑤ somewhere until M2 is ready to consume them, M1 can proceed to the next compute in the meantime!

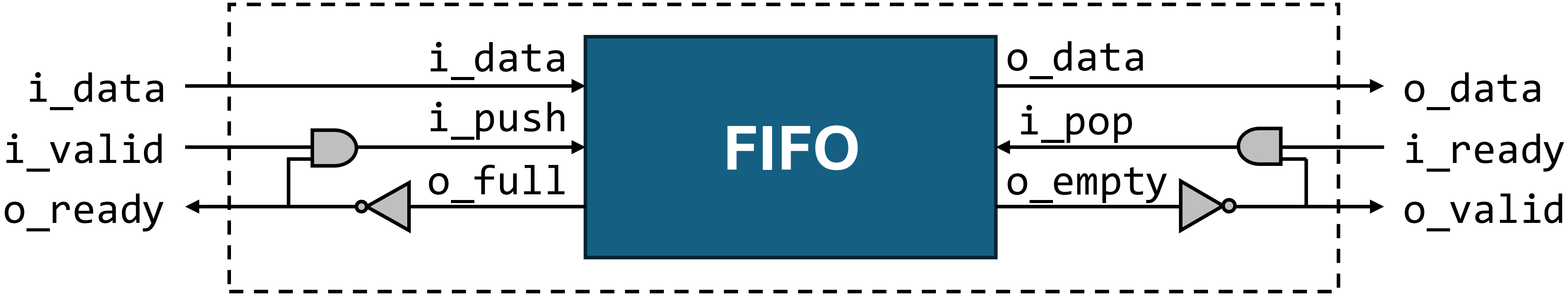
FIFOs can help!

- A **FIFO** is a **first-in first-out** buffer with a push/pop interface
  - Has status output signals (e.g., `empty` and `full`)
  - As long as not full → can push data into it
  - As long as not empty → can pop data from it
  - Similar to a queue data structure in software
  - Bitwidth of input/output data & FIFO depth are design parameters



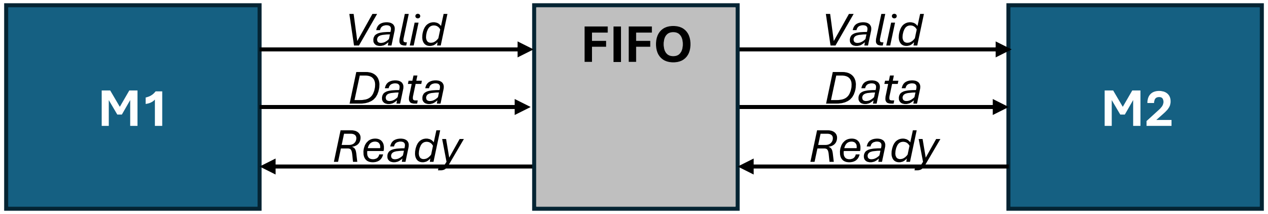
FIFOs can help!

- A **FIFO** is a **first-in first-out** buffer with a push/pop interface
  - Has status output signals (e.g., empty and full)
  - As long as not full → can push data into it
  - As long as not empty → can pop data from it
  - Similar to a queue data structure in software
  - Bitwidth of input/output data & FIFO depth are design parameters

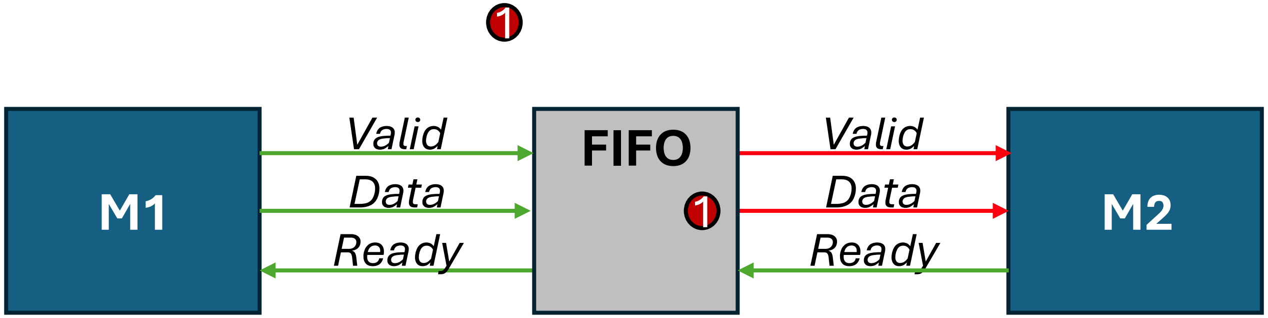


Can be wrapped to present a latency-insensitive ready-valid interface

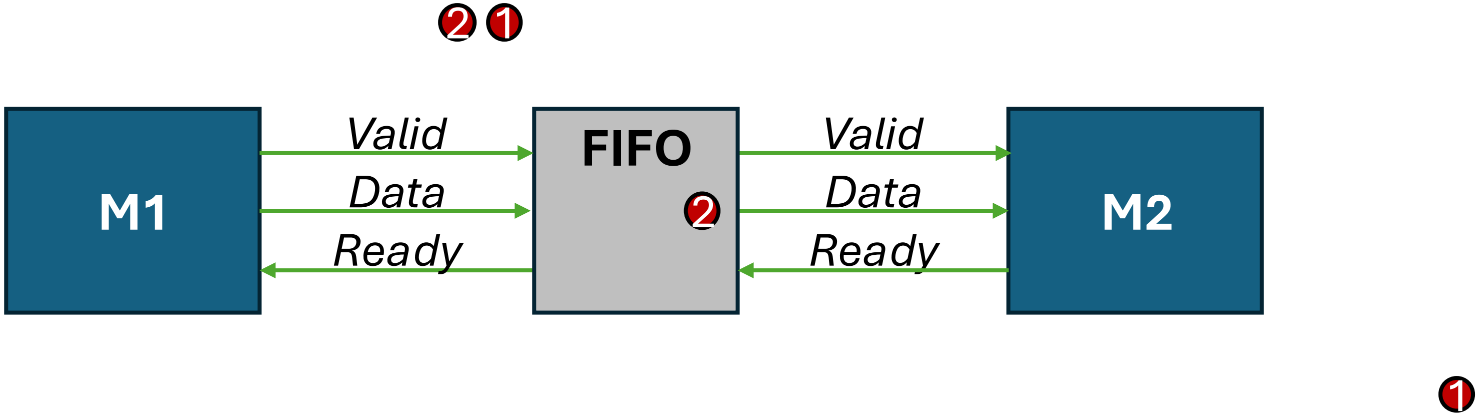
FIFO-Based Design



FIFO-Based Design



FIFO-Based Design



M1

Valid

Data

Ready

FIFO

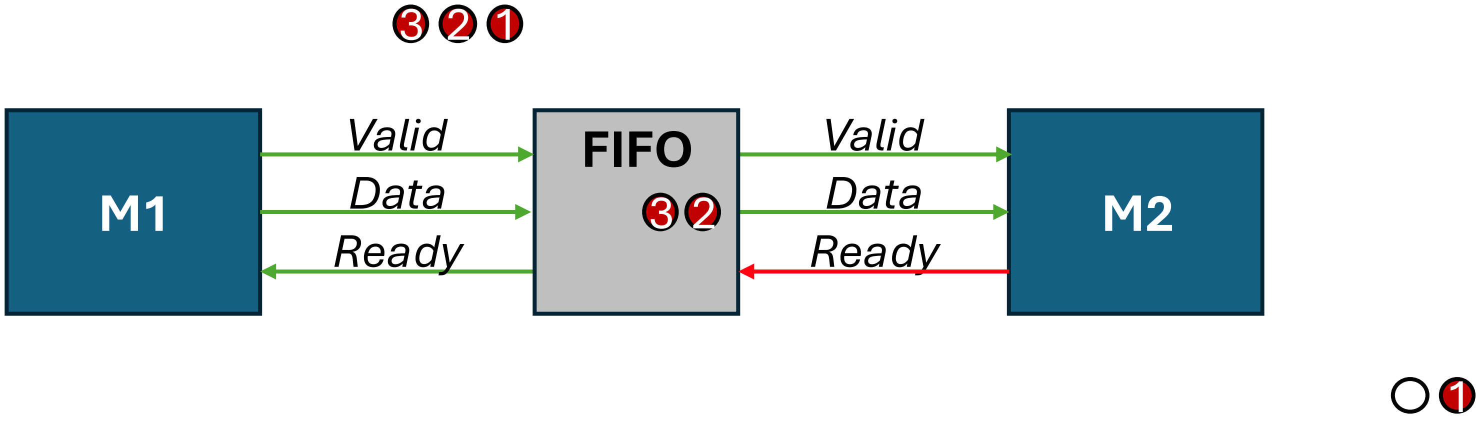
Valid

Data

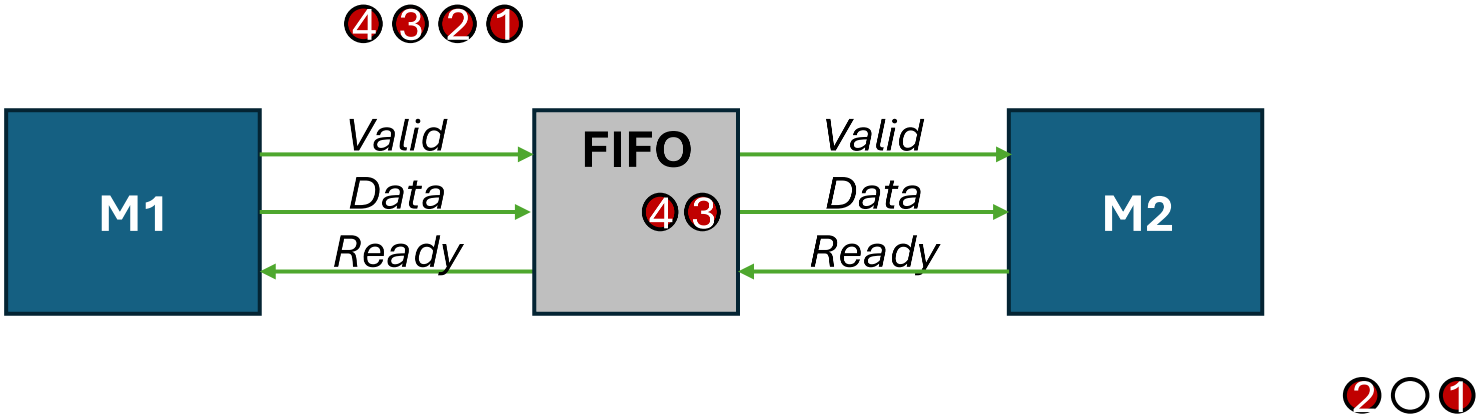
Ready

M2

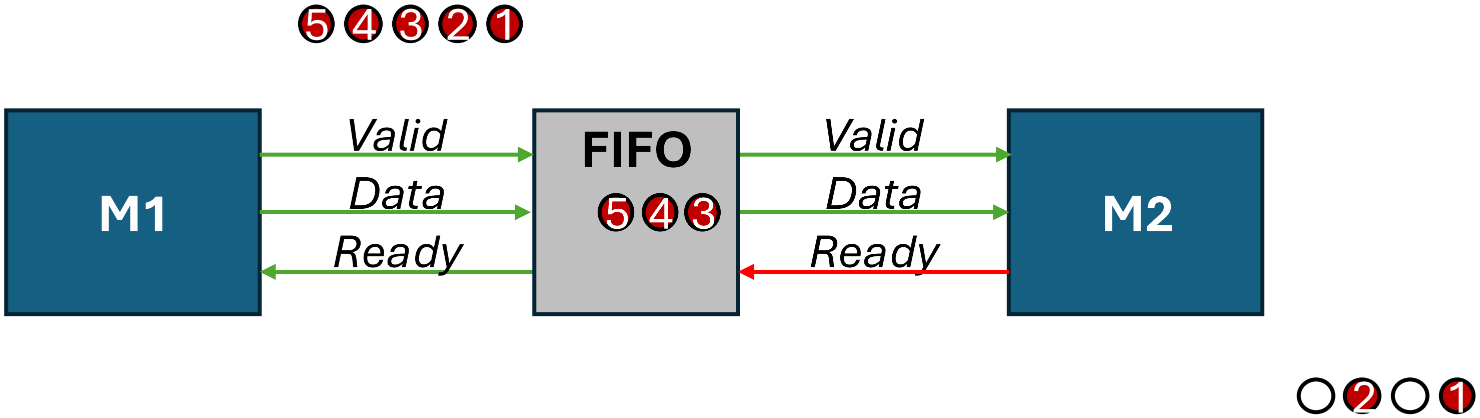
FIFO-Based Design



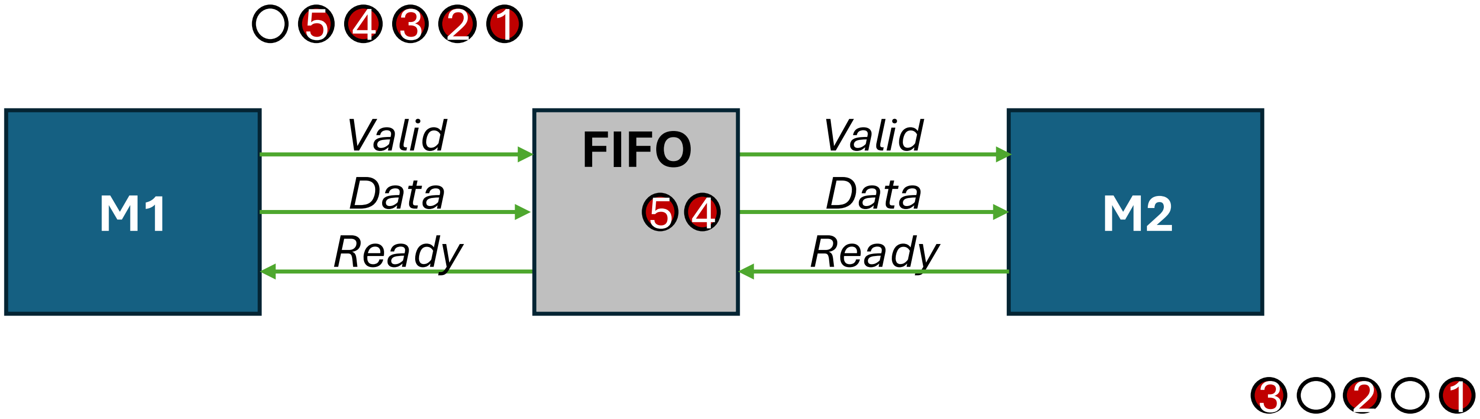
FIFO-Based Design



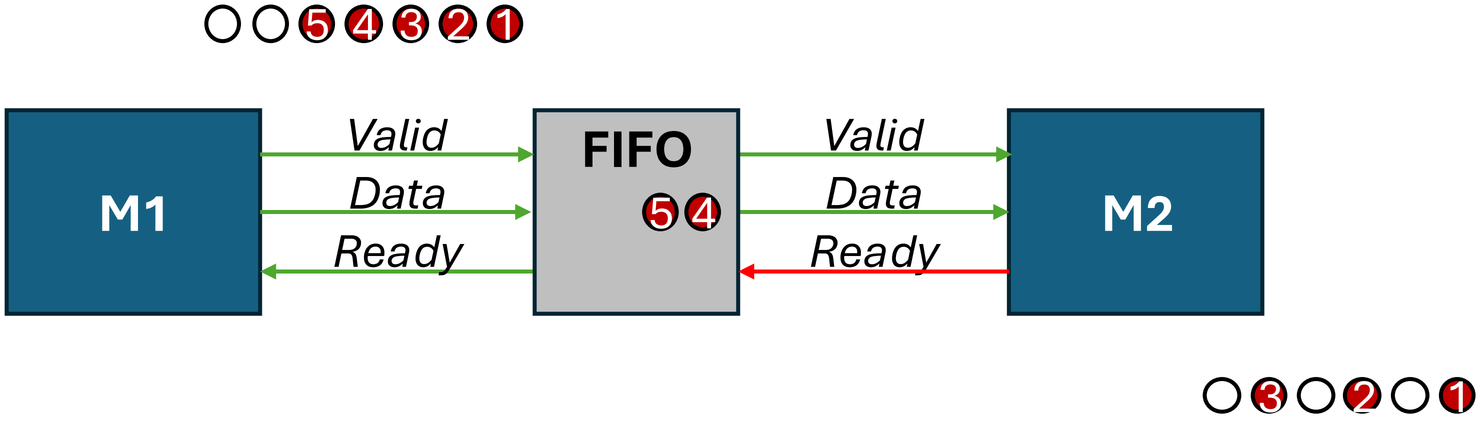
FIFO-Based Design



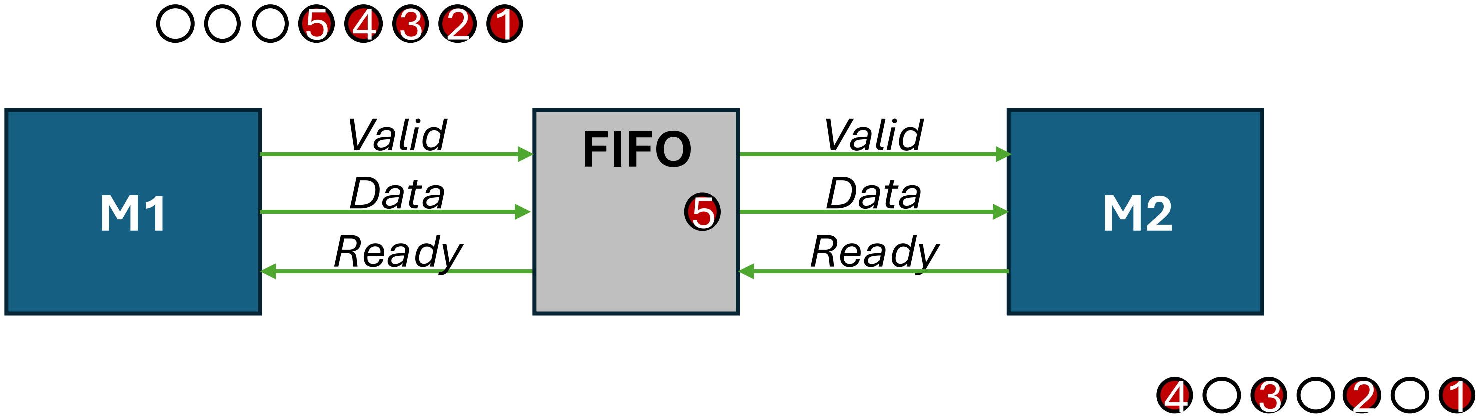
FIFO-Based Design



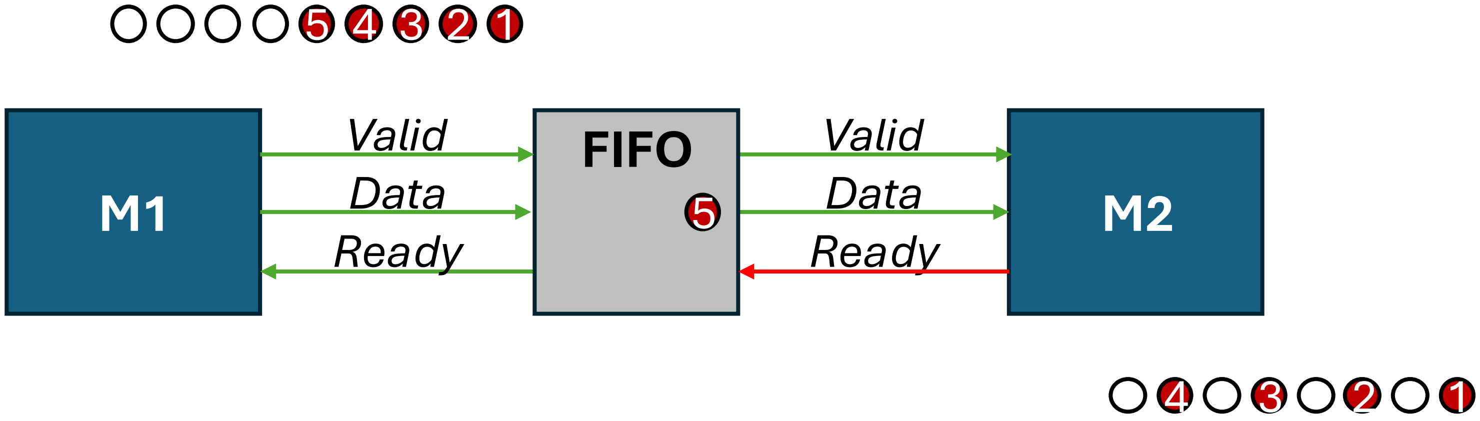
FIFO-Based Design



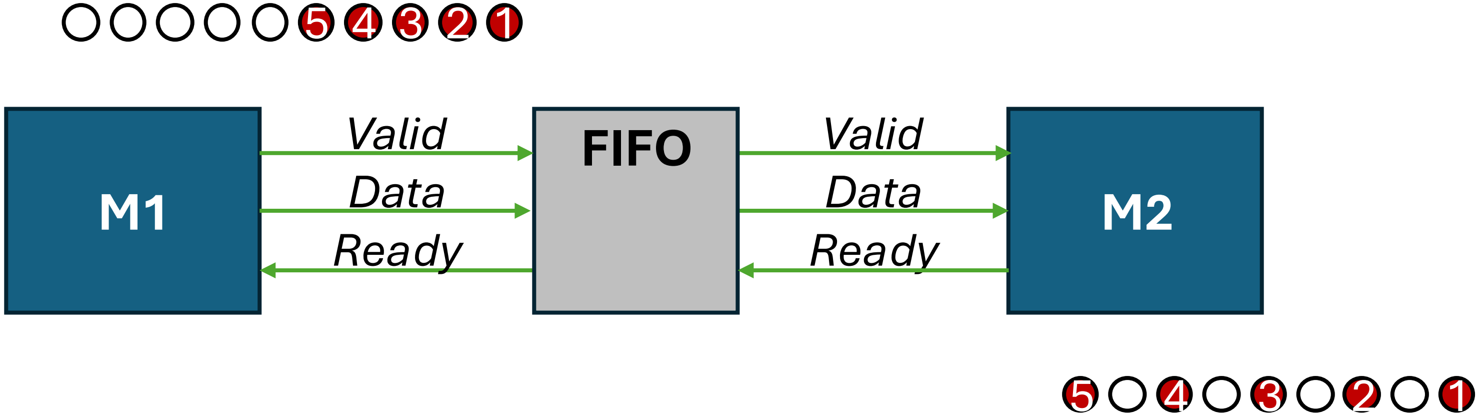
FIFO-Based Design



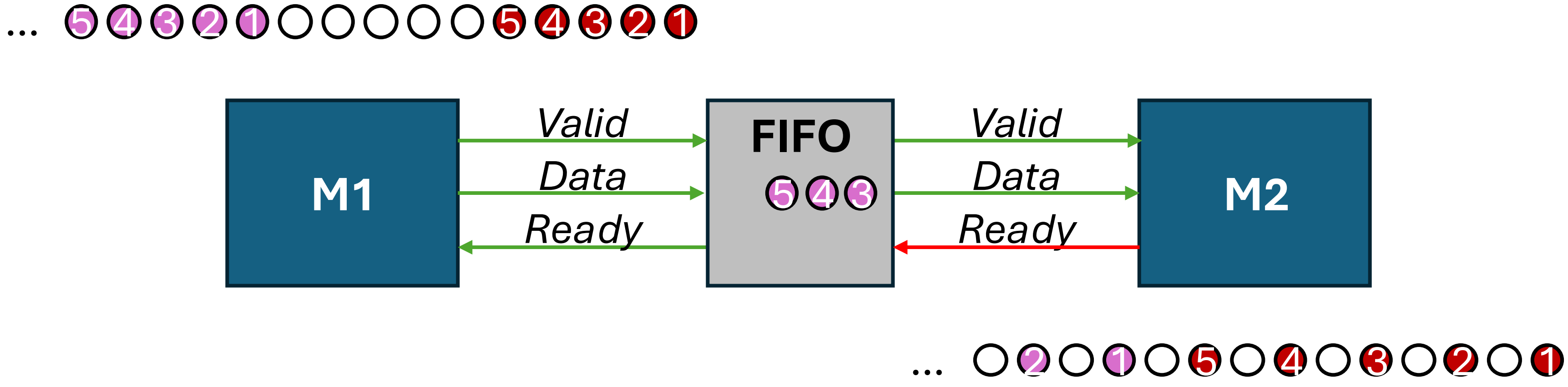
FIFO-Based Design



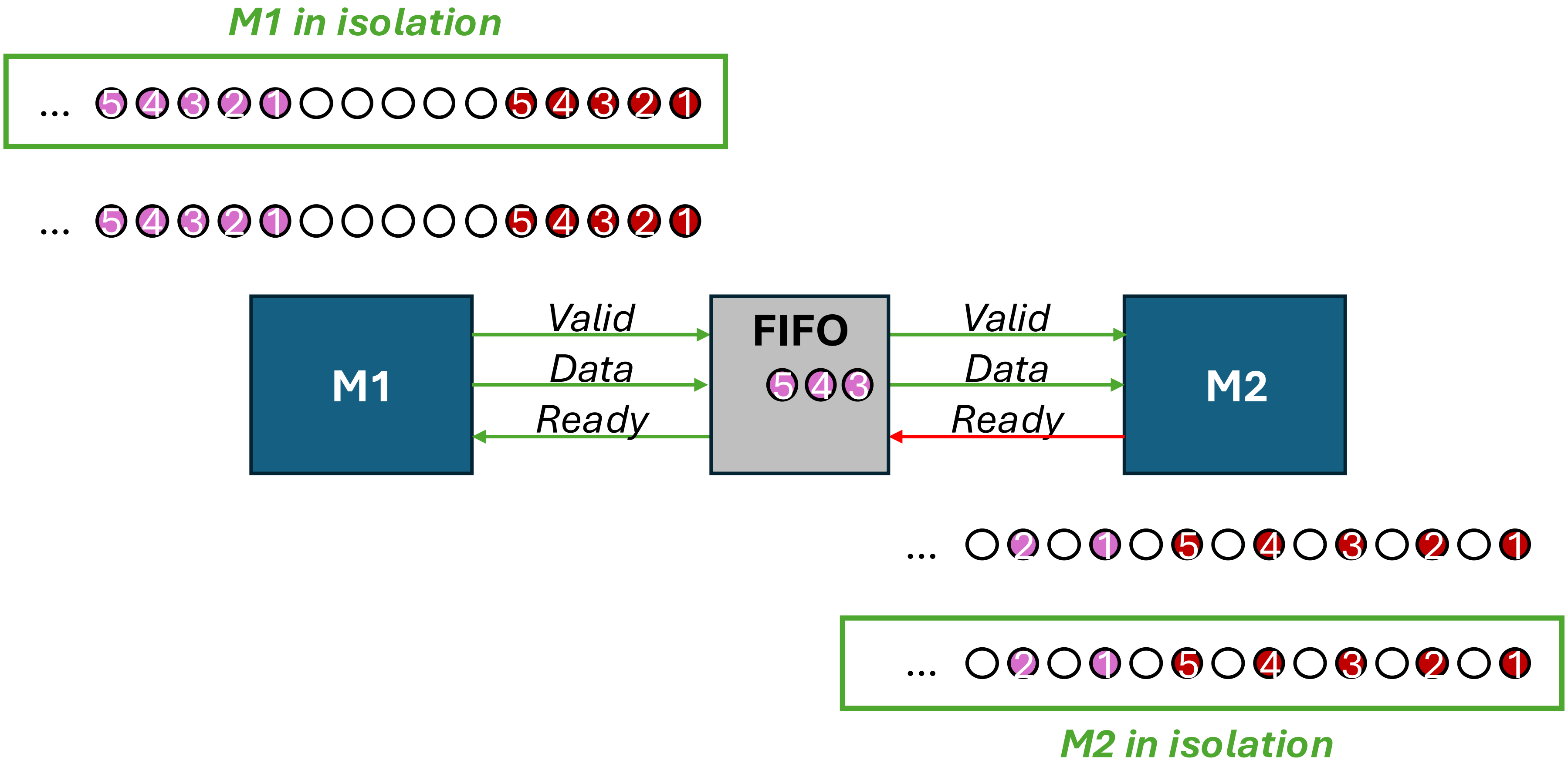
FIFO-Based Design



FIFO-Based Design



FIFO-Based Design



M1 in isolation

… 5 4 3 2 1 ○ ○ ○ ○ ○ 5 4 3 2 1

… 5 4 3 2 1 ○ ○ ○ ○ ○ 5 4 3 2 1

M1

Valid

Data

Ready

FIFO

5 4 3

Valid

Data

Ready

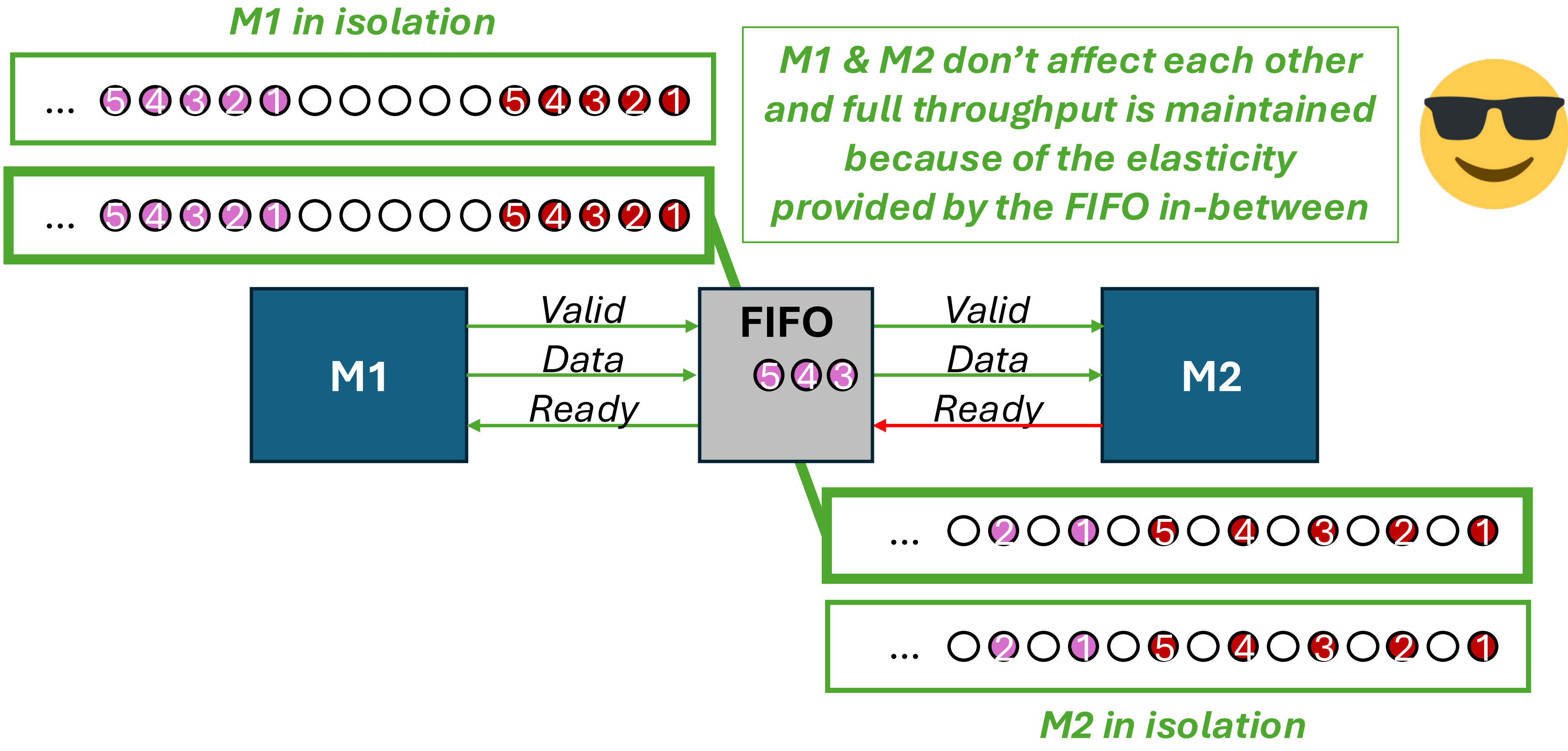
M2

… ○ 2 ○ 1 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

… ○ 2 ○ 1 ○ 5 ○ 4 ○ 3 ○ 2 ○ 1

M2 in isolation

FIFO-Based Design



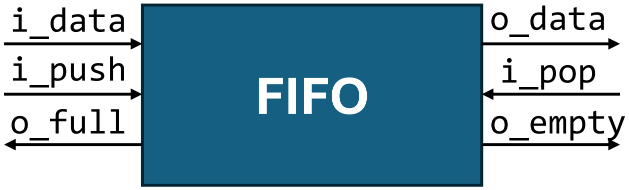
M1 & M2 don't affect each other and full throughput is maintained because of the elasticity provided by the FIFO in-between

FIFO Implementation (Vanilla Interface)

FIFO Implementation (Vanilla Interface)

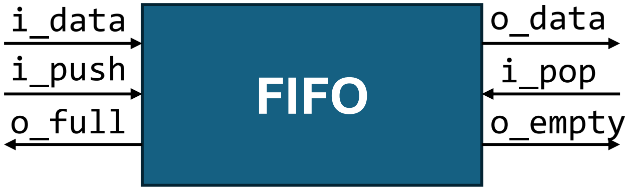
```
module fifo # (

)(

);
```

```
endmodule
```



FIFO Implementation (Vanilla Interface)

```
module fifo # (
    parameter DATAW = 32,
    parameter DEPTH = 512,
    parameter ADDRW = $clog2(DEPTH)
)(









);
```

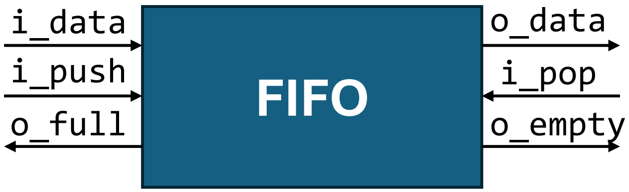


```
endmodule
```

FIFO Implementation (Vanilla Interface)

```
module fifo # (
    parameter DATAW = 32,
    parameter DEPTH = 512,
    parameter ADDRW = $clog2(DEPTH)
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_push,
    output o_full,
    output logic [DATAW-1:0] o_data,
    input  i_pop,
    output o_empty
);
```

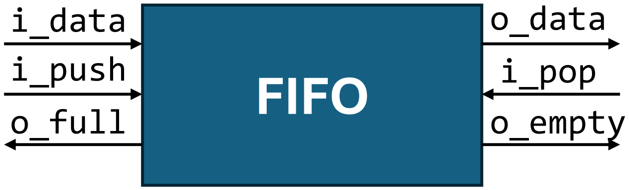
```
endmodule
```



FIFO Implementation (Vanilla Interface)

```
module fifo # (
    parameter DATAW = 32,
    parameter DEPTH = 512,
    parameter ADDRW = $clog2(DEPTH)
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_push,
    output o_full,
    output logic [DATAW-1:0] o_data,
    input  i_pop,
    output o_empty
);

logic [DATAW-1:0] mem [0:DEPTH-1];
logic [ADDRW-1:0] rd_ptr, wr_ptr;
```

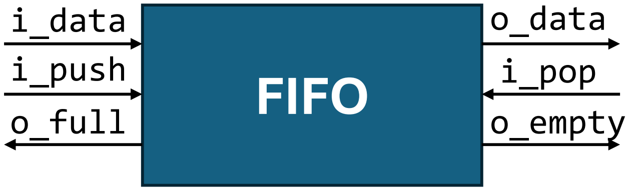
```
endmodule
```



FIFO Implementation (Vanilla Interface)

```
module fifo # (
    parameter DATAW = 32,
    parameter DEPTH = 512,
    parameter ADDRW = $clog2(DEPTH)
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_push,
    output o_full,
    output logic [DATAW-1:0] o_data,
    input  i_pop,
    output o_empty
);

logic [DATAW-1:0] mem [0:DEPTH-1];
logic [ADDRW-1:0] rd_ptr, wr_ptr;
```

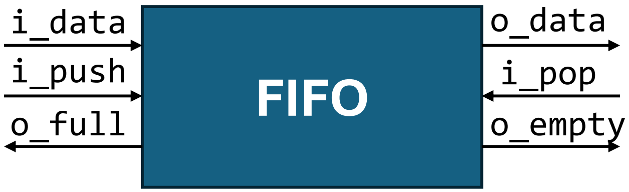
```
always_ff @ (posedge clk) begin
    if (rst) begin
        rd_ptr <= 'd0;
        wr_ptr <= 'd0;
    end else begin
        if (~o_empty && i_pop) begin
            o_data <= mem[rd_ptr];
            rd_ptr <= rd_ptr + 1;
        end

        if (~o_full && i_push) begin
            mem[wr_ptr] <= i_data;
            wr_ptr <= wr_ptr + 1;
        end
    end
end




endmodule
```



FIFO Implementation (Vanilla Interface)

```
module fifo # (
    parameter DATAW = 32,
    parameter DEPTH = 512,
    parameter ADDRW = $clog2(DEPTH)
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_push,
    output o_full,
    output logic [DATAW-1:0] o_data,
    input  i_pop,
    output o_empty
);

logic [DATAW-1:0] mem [0:DEPTH-1];
logic [ADDRW-1:0] rd_ptr, wr_ptr;
```

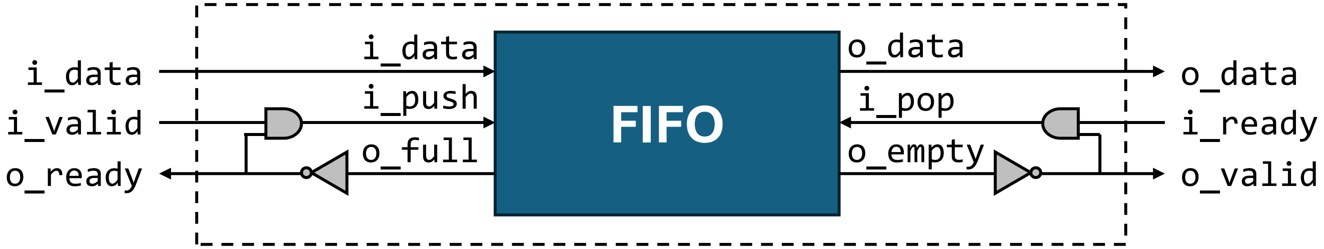
```
always_ff @ (posedge clk) begin
    if (rst) begin
        rd_ptr <= 'd0;
        wr_ptr <= 'd0;
    end else begin
        if (~o_empty && i_pop) begin
            o_data <= mem[rd_ptr];
            rd_ptr <= rd_ptr + 1;
        end

        if (~o_full && i_push) begin
            mem[wr_ptr] <= i_data;
            wr_ptr <= wr_ptr + 1;
        end
    end
end

assign o_full = (wr_ptr +1) == rd_ptr;
assign o_empty = wr_ptr == rd_ptr;

endmodule
```



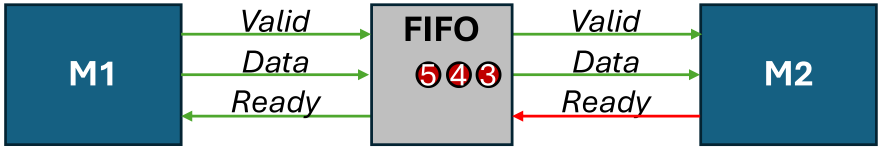
FIFO Implementation (Ready-Valid Interface)

```
module fifo_ready_valid # (
    parameter DATAW = 32,
    parameter DEPTH = 512
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_valid,
    output o_ready,
    output logic [DATAW-1:0] o_data,
    output o_valid,
    input  i_ready
);

logic fifo_full, fifo_empty, fifo_push, fifo_pop;
```

```
fifo # (
    .DATAW(DATAW),
    .DEPTH(DEPTH)
) fifo_inst (
    .clk(clk),
    .rst(rst),
    .i_data(i_data),
    .i_push(fifo_push),
    .o_full(fifo_full),
    .o_data(o_data),
    .i_pop(fifo_pop),
    .o_empty(fifo_empty)
);

assign fifo_push = i_valid && o_ready;
assign fifo_pop = o_valid && i_ready;
assign o_ready = ~fifo_full;
assign o_valid = ~fifo_empty;

endmodule
```

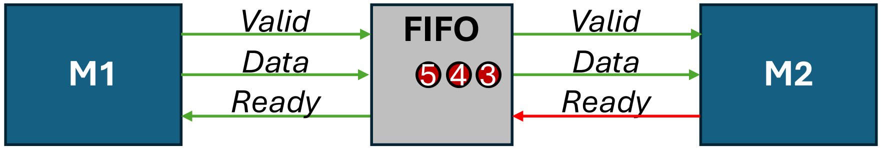


Design Considerations



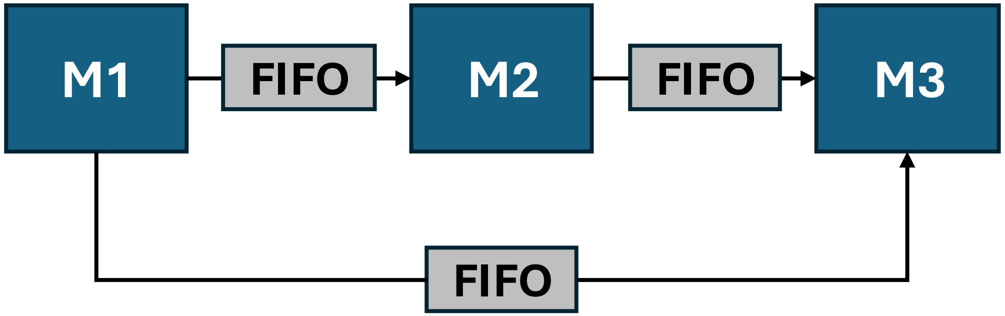
- FIFO depth is a key design choice …
  - A very shallow FIFO would backpressure frequently affecting overall throughput negatively
  - A very deep FIFO is more expensive in terms of silicon area footprint
  - **Goal:** minimize FIFO size and provide enough elasticity to maintain full throughput

Design Considerations

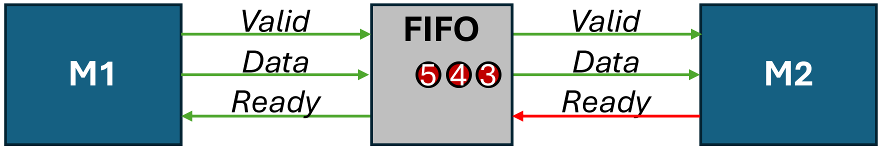


- FIFO depth is a key design choice …
  - A very shallow FIFO would backpressure frequently affecting overall throughput negatively
  - A very deep FIFO is more expensive in terms of silicon area footprint
  - **Goal:** minimize FIFO size and provide enough elasticity to maintain full throughput

- FIFOs have to be carefully sized to avoid deadlocks!

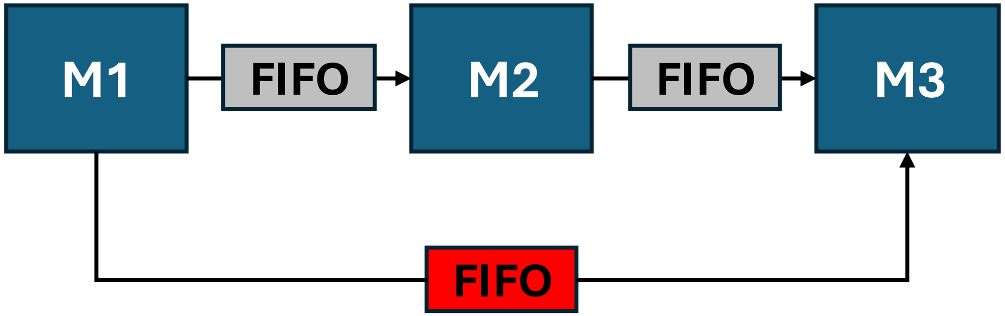


Design Considerations



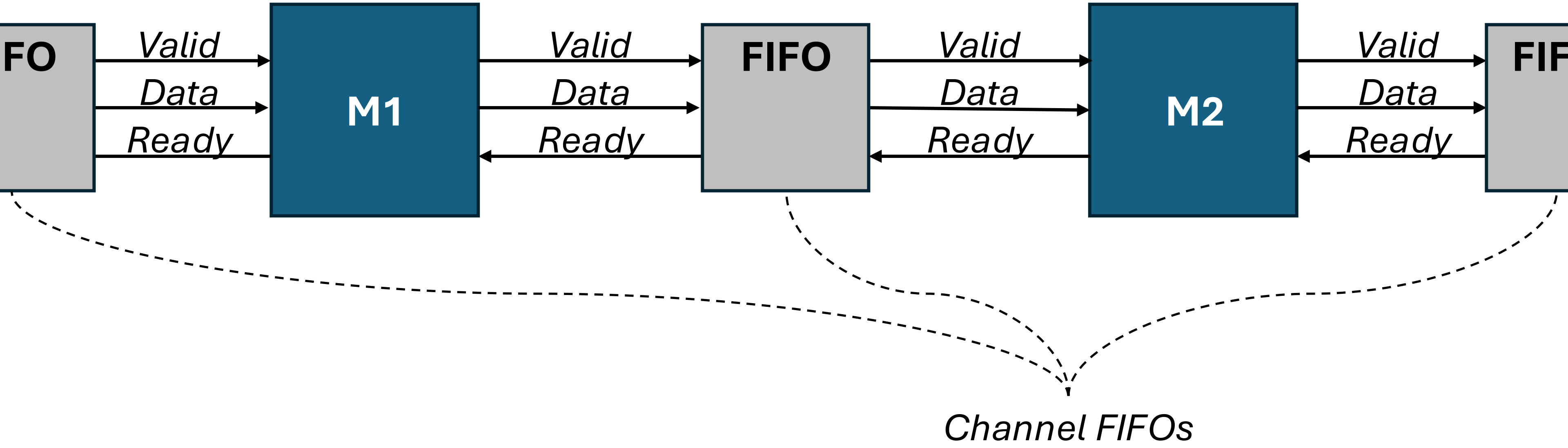
- FIFO depth is a key design choice …
  - A very shallow FIFO would backpressure frequently affecting overall throughput negatively
  - A very deep FIFO is more expensive in terms of silicon area footprint
  - **Goal:** minimize FIFO size and provide enough elasticity to maintain full throughput

- FIFOs have to be carefully sized to avoid deadlocks!

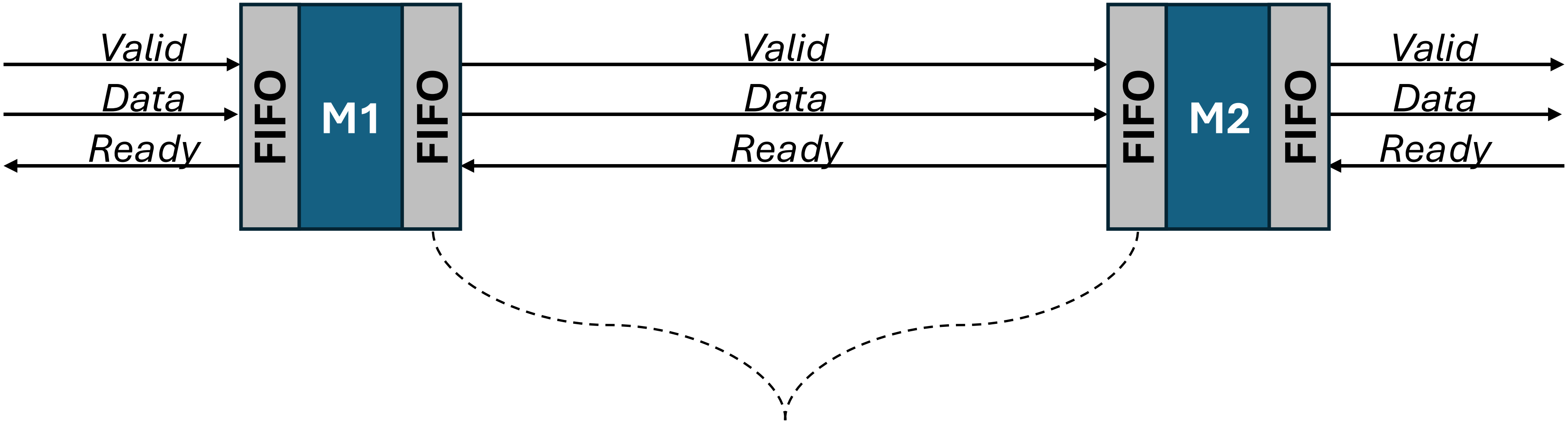


If this FIFO gets full, M1 will be stalled.
If this happens before M2 gets enough data elements from M1 to produce outputs, the design will deadlock!

Coming Full Circle: FIFOs & LI Channels



Coming Full Circle: FIFOs & LI Channels

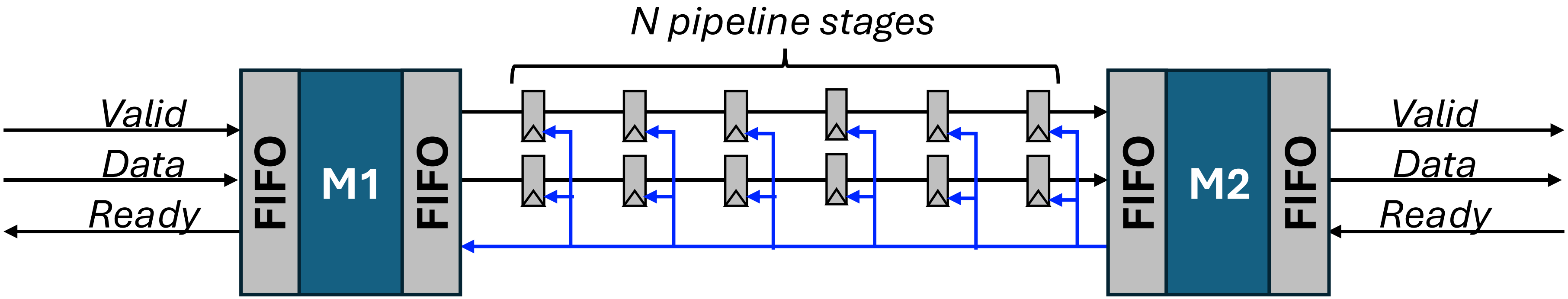


Can split channel FIFOs such that each of the modules has FIFOs on its inputs and outputs

Depths of split FIFOs combined = Depth of channel FIFO

FIFO control logic replicated (not very expensive)

Coming Full Circle: FIFOs & LI Channels

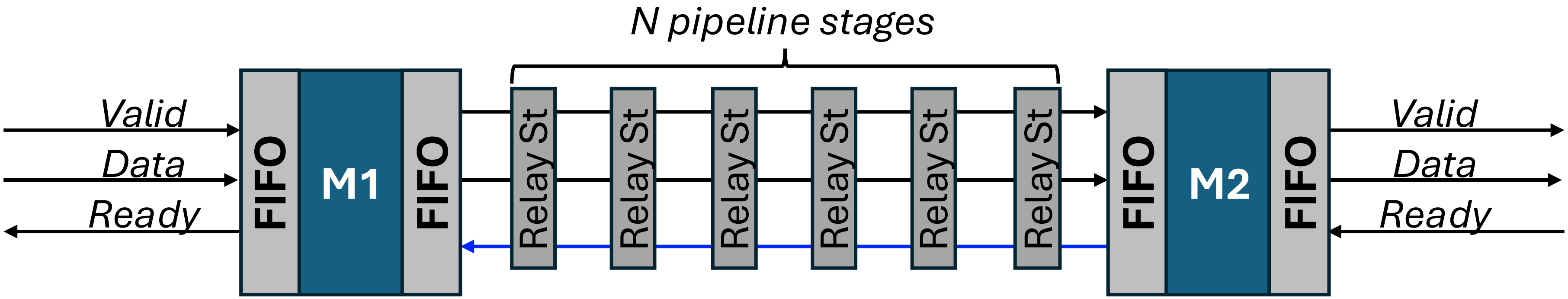


If we want to pipeline the latency-insensitive channel to optimize timing, ready signal has to stay combinational to …

(1) Freeze the pipeline

(2) Stall the sender module

Coming Full Circle: FIFOs & LI Channels



If we want to pipeline the latency-insensitive channel to optimize timing, ready signal has to stay combinational to ...

(1) Freeze the pipeline

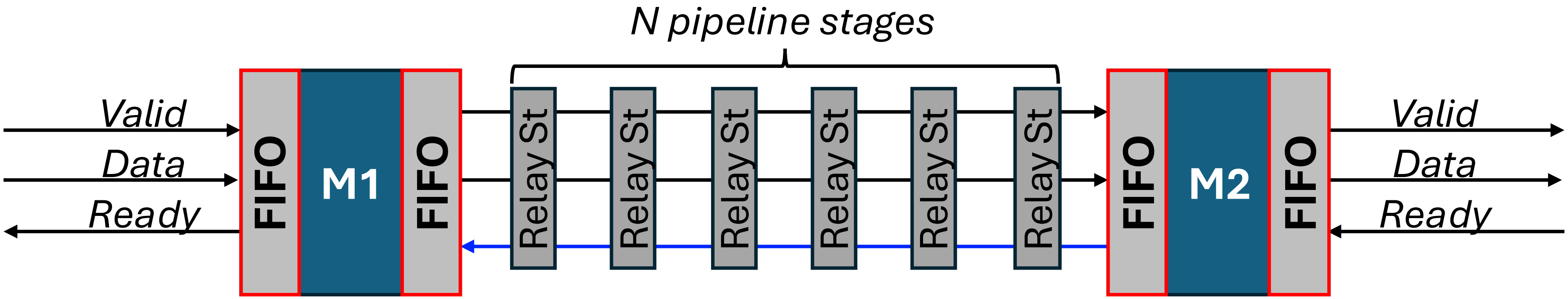
(2) Stall the sender module

Or use relay stations instead of simple registers

→ higher cost (2x the registers + control FSM per stage)

Coming Full Circle: FIFOs & LI Channels

These FIFOs can provide a different solution!



If we want to pipeline the latency-insensitive channel to optimize timing, ready signal had to stay combinational to …

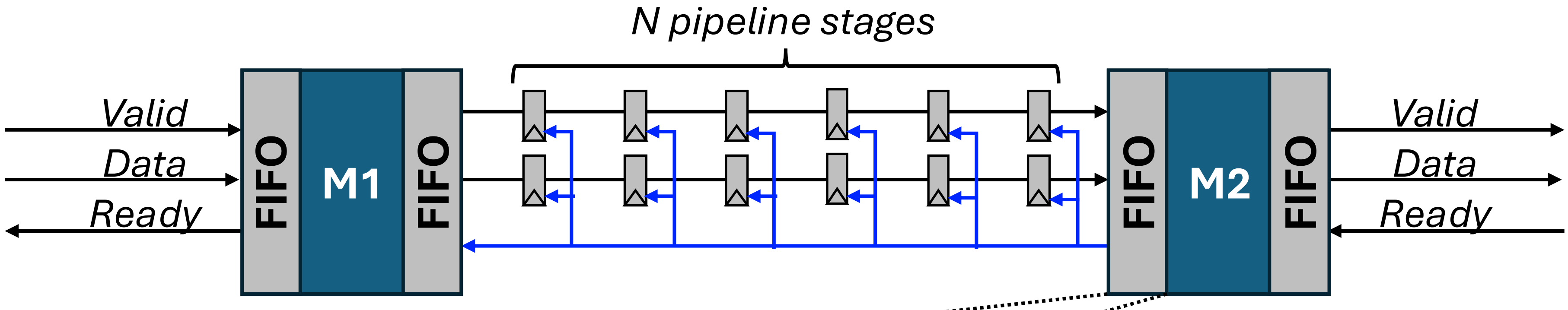
(1) Freeze the pipeline

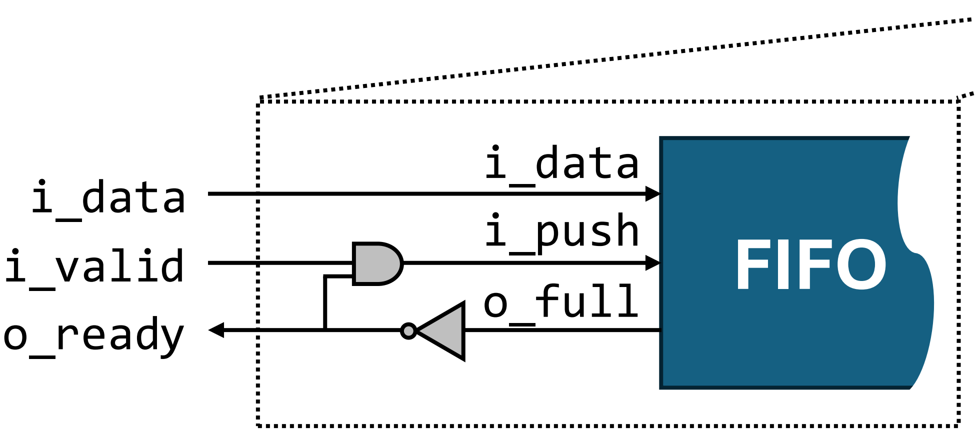
(2) Stall the sender module

Or use relay stations instead of simple registers

→ higher cost (2x the registers + control FSM per stage)

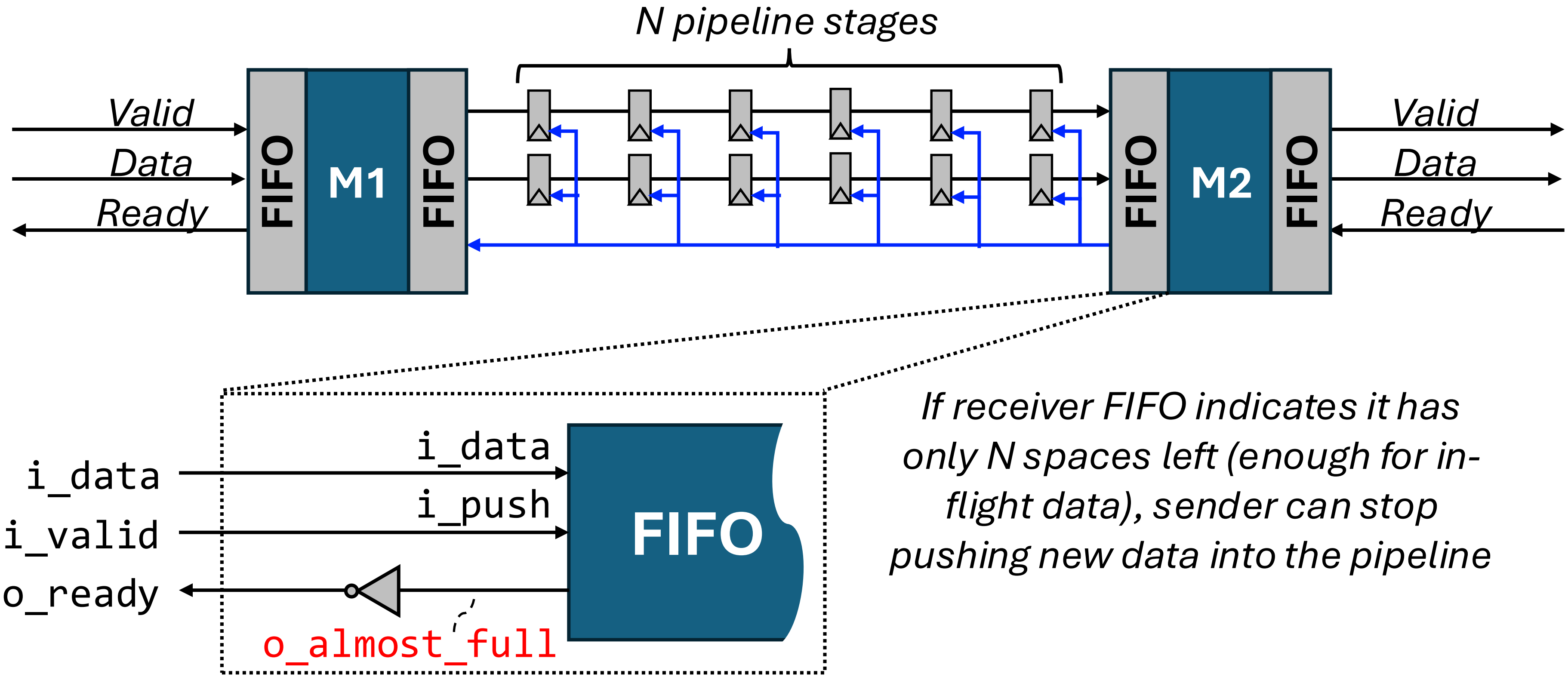
Coming Full Circle: FIFOs & LI Channels





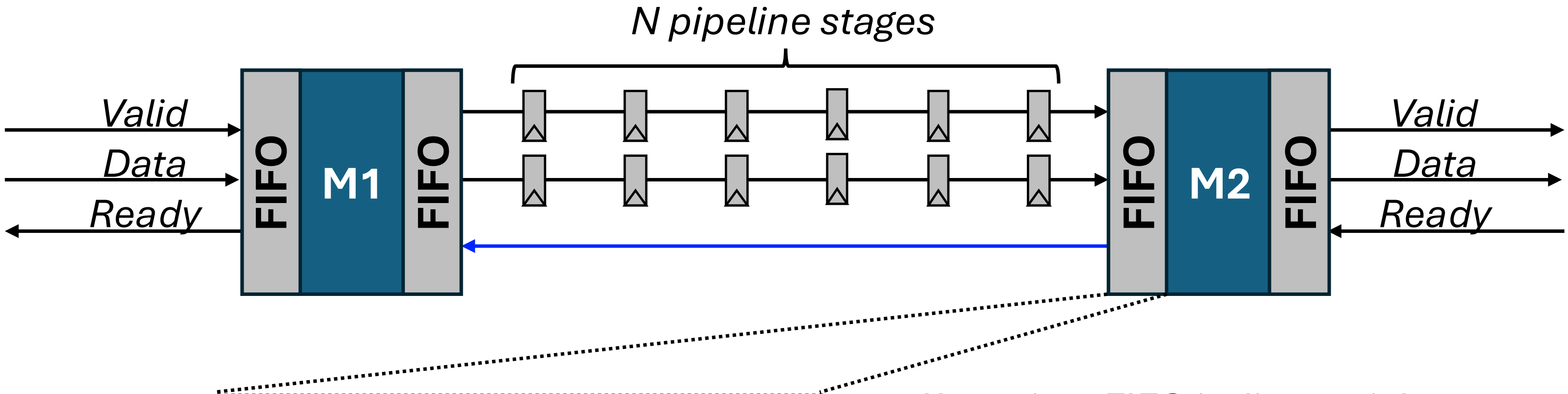
If receiver FIFO indicates it has only N spaces left (enough for in-flight data), sender can stop pushing new data into the pipeline

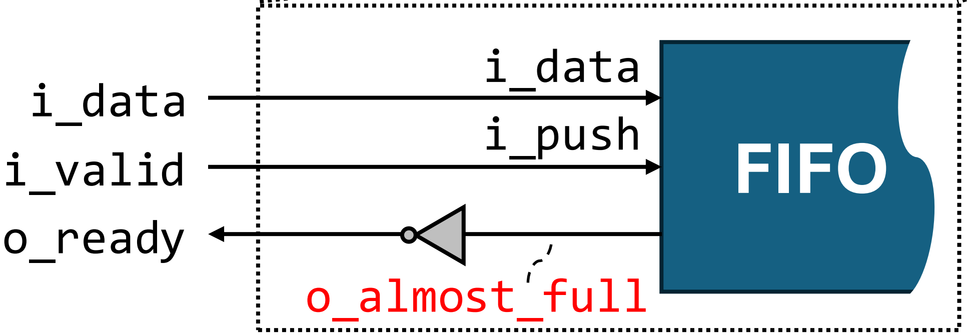
Coming Full Circle: FIFOs & LI Channels



If receiver FIFO indicates it has only N spaces left (enough for in-flight data), sender can stop pushing new data into the pipeline

Coming Full Circle: FIFOs & LI Channels

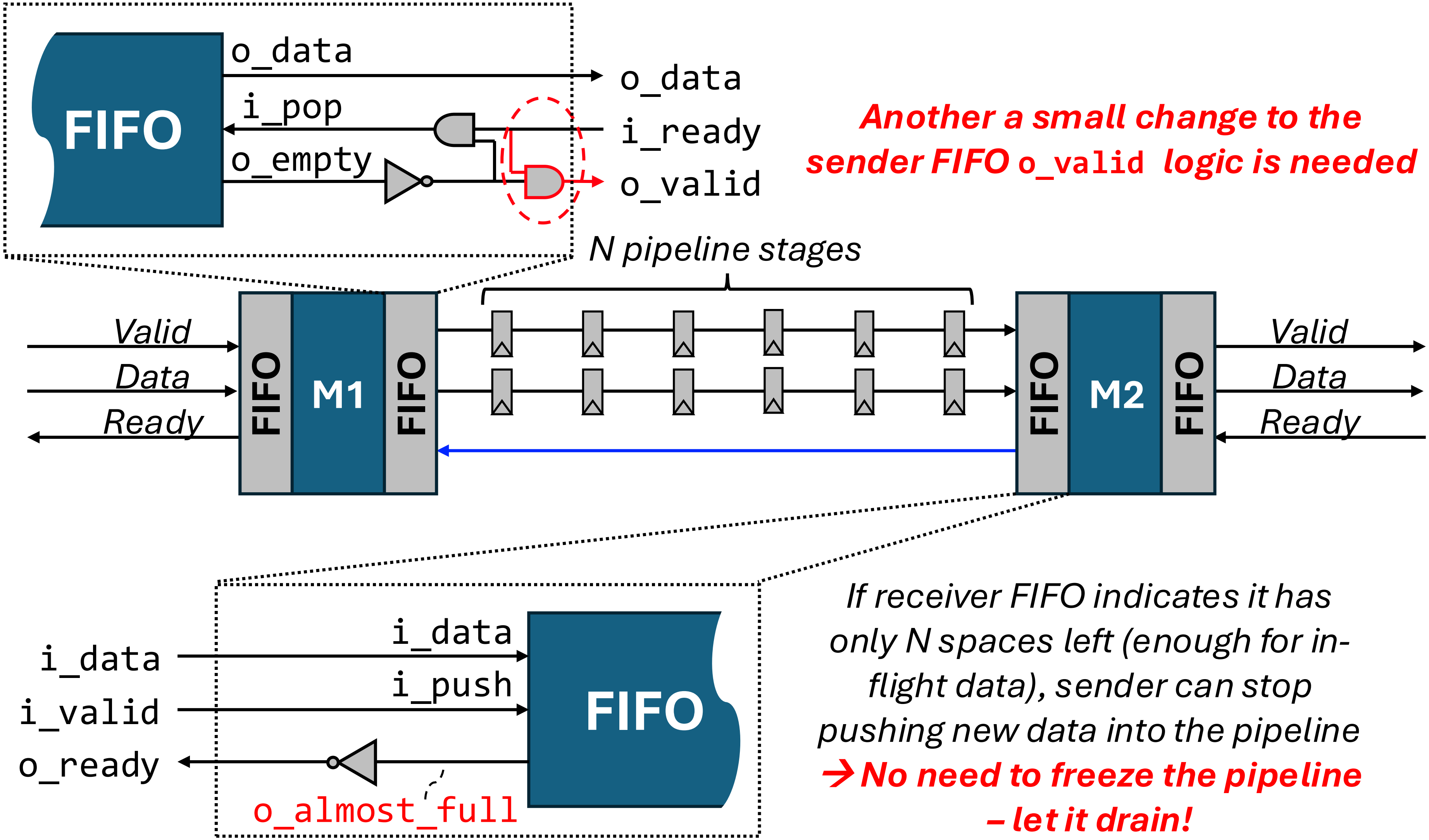




If receiver FIFO indicates it has only N spaces left (enough for in-flight data), sender can stop pushing new data into the pipeline

→No need to freeze the pipeline – let it drain!

Coming Full Circle: FIFOs & LI Channels

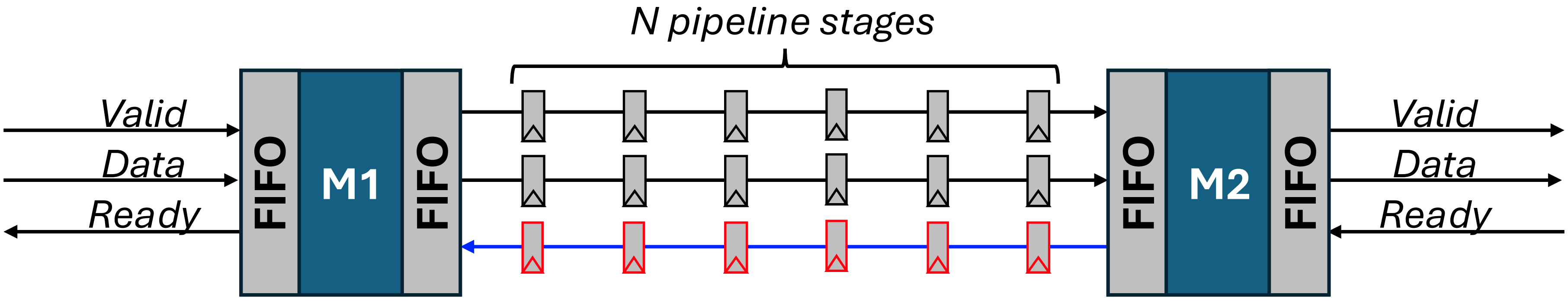


Another a small change to the sender FIFO `o_valid` logic is needed

If receiver FIFO indicates it has only N spaces left (enough for in-flight data), sender can stop pushing new data into the pipeline

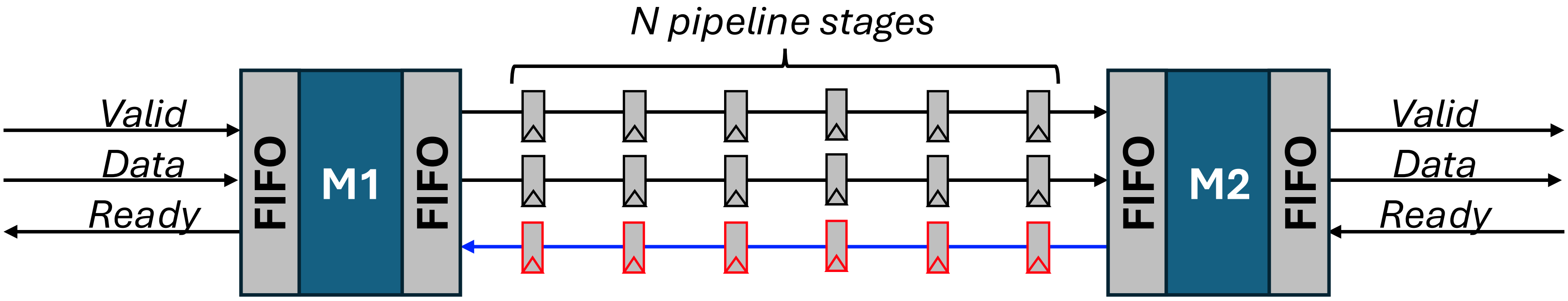
→ No need to freeze the pipeline – let it drain!

Coming Full Circle: FIFOs & LI Channels



Now if we pipeline the ready signal, it takes another N cycles to reach the sender

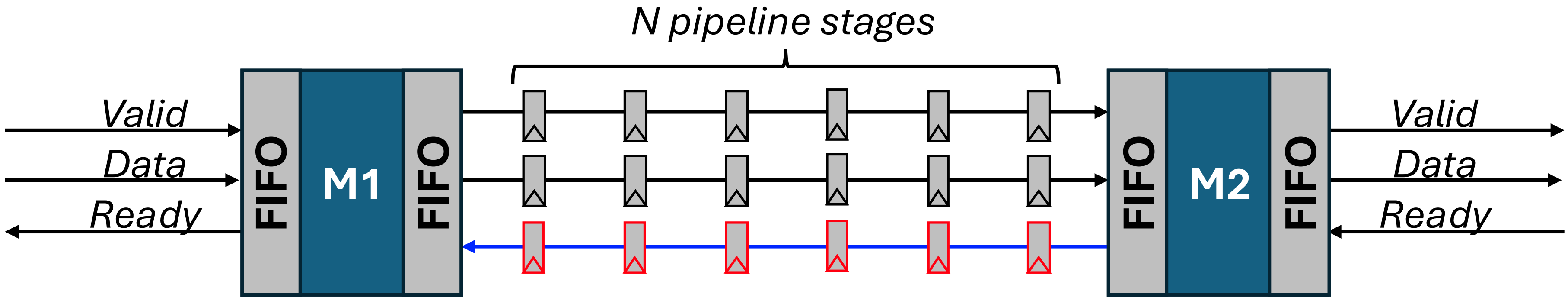
Coming Full Circle: FIFOs & LI Channels



Now if we pipeline the ready signal, it takes another N cycles to reach the sender

→ If receiver FIFO indicates it is almost full when it has 2N spaces left, works fine!

Coming Full Circle: FIFOs & LI Channels

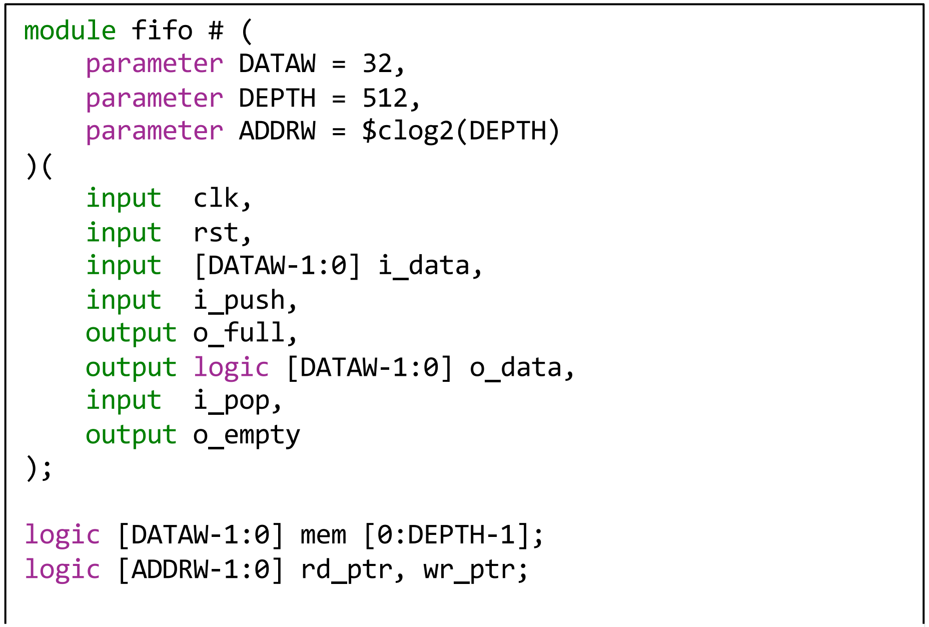


Now if we pipeline the ready signal, it takes another N cycles to reach the sender

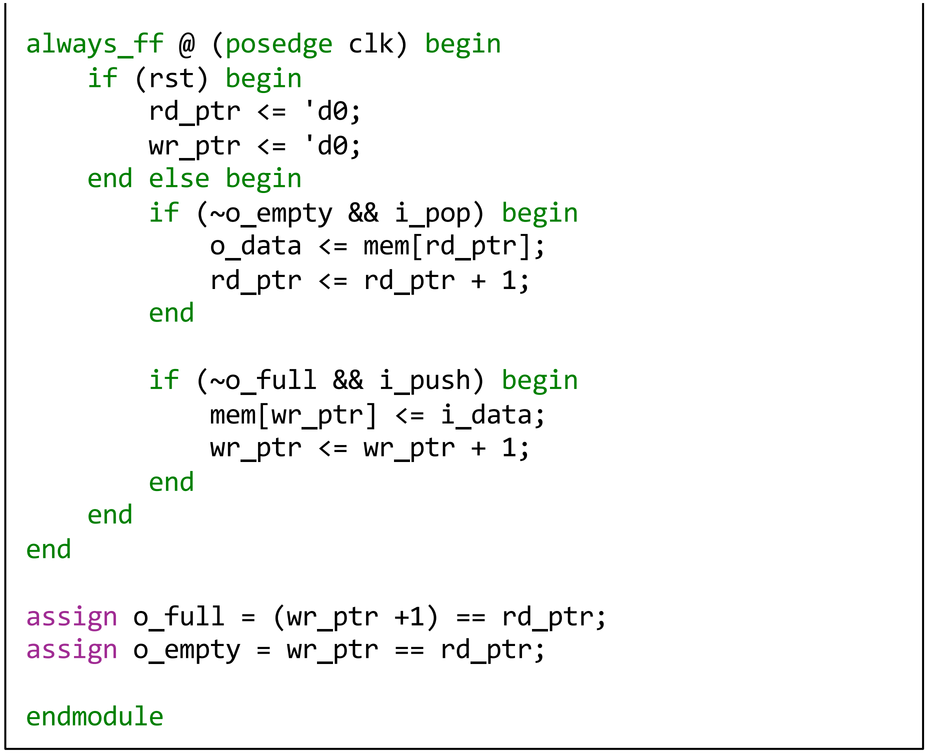
→ If receiver FIFO indicates it is almost full when it has 2N spaces left, works fine!

There is one subtle catch … Any idea? 🍬

FIFO Implementation with almost_full Status



```
module fifo # (
    parameter DATAW = 32,
    parameter DEPTH = 512,
    parameter ADDRW = $clog2(DEPTH)
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_push,
    output o_full,
    output logic [DATAW-1:0] o_data,
    input  i_pop,
    output o_empty
);

logic [DATAW-1:0] mem [0:DEPTH-1];
logic [ADDRW-1:0] rd_ptr, wr_ptr;
```



```
always_ff @ (posedge clk) begin
    if (rst) begin
        rd_ptr <= 'd0;
        wr_ptr <= 'd0;
    end else begin
        if (~o_empty && i_pop) begin
            o_data <= mem[rd_ptr];
            rd_ptr <= rd_ptr + 1;
        end

        if (~o_full && i_push) begin
            mem[wr_ptr] <= i_data;
            wr_ptr <= wr_ptr + 1;
        end
    end
end

assign o_full = (wr_ptr +1) == rd_ptr;
assign o_empty = wr_ptr == rd_ptr;

endmodule
```

What needs to change in the FIFO implementation to support almost_full?

FIFO Implementation with almost_full Status

```
module fifo_ready_valid # (
    parameter DATAW = 32,
    parameter DEPTH = 512
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_valid,
    output o_ready,
    output logic [DATAW-1:0] o_data,
    output logic o_valid,
    input  i_ready
);

logic fifo_full, fifo_empty, fifo_push, fifo_pop;
```

```
fifo # (
    .DATAW(DATAW),
    .DEPTH(DEPTH)
) fifo_inst (
    .clk(clk),
    .rst(rst),
    .i_data(i_data),
    .i_push(fifo_push),
    .o_full(fifo_full),
    .o_data(o_data),
    .i_pop(fifo_pop),
    .o_empty(fifo_empty)
);

assign fifo_push = i_valid && o_ready;
assign fifo_pop = o_valid && i_ready;
assign o_ready = ~fifo_full;
assign o_valid = ~fifo_empty;

endmodule
```

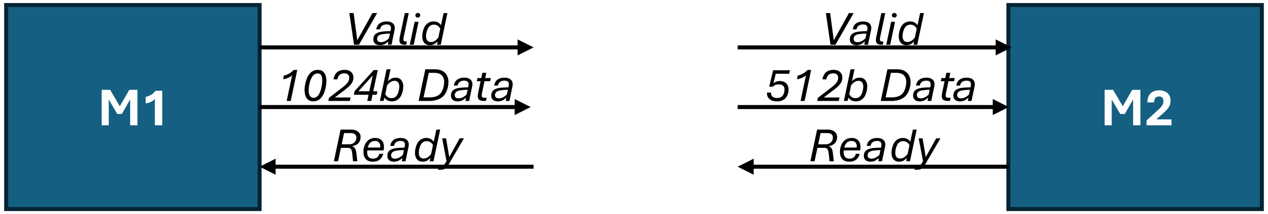
FIFO Implementation with almost_full Status

```
module fifo_ready_valid # (
    parameter DATAW = 32,
    parameter DEPTH = 512,
    parameter ALMOST_FULL_DEPTH = DEPTH - 2
)(
    input  clk,
    input  rst,
    input  [DATAW-1:0] i_data,
    input  i_valid,
    output o_ready,
    output logic [DATAW-1:0] o_data,
    output logic o_valid,
    input  i_ready
);

logic fifo_full, fifo_empty, fifo_push, fifo_pop;
logic fifo_almost_full;
```

```
fifo # (
    .DATAW(DATAW),
    .DEPTH(DEPTH),
    .ALMOST_FULL_DEPTH(ALMOST_FULL_DEPTH)
) fifo_inst (
    .clk(clk),
    .rst(rst),
    .i_data(i_data),
    .i_push(fifo_push),
    .o_full(fifo_full),
    .o_almost_full(fifo_almost_full),
    .o_data(o_data),
    .i_pop(fifo_pop),
    .o_empty(fifo_empty)
);

assign fifo_push = i_valid;
assign fifo_pop = o_valid && i_ready;
assign o_ready = ~fifo_almost_full;
assign o_valid = ~fifo_empty && i_ready;

endmodule
```

Data Width Adaptation

- In many cases, sender/receiver modules produce/consume data words that have different bitwidths

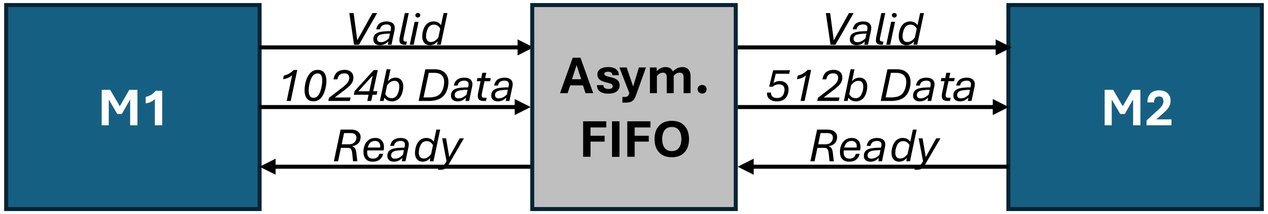


M1 produces a 1024-bit data word every other cycle

M2 can consume a 512-bit data word every cycle

Data Width Adaptation

- In many cases, sender/receiver modules produce/consume data words that have different bitwidths
- An asymmetric FIFO can be used to perform width adaptation

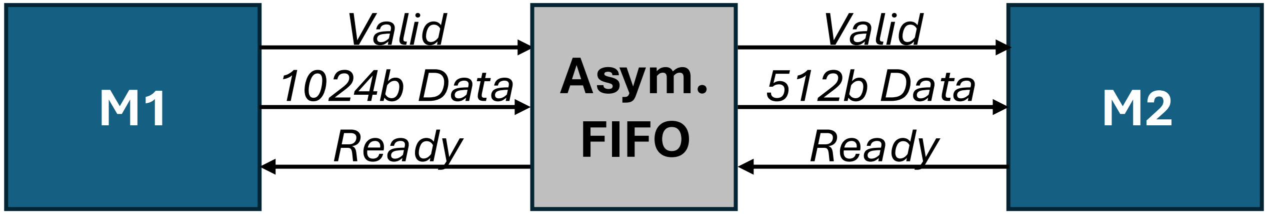


M1 produces a 1024-bit data word every other cycle

M2 can consume a 512-bit data word every cycle

Data Width Adaptation

- In many cases, sender/receiver modules produce/consume data words that have different bitwidths
- An asymmetric FIFO can be used to perform width adaptation



M1 produces a 1024-bit data word every other cycle

M2 can consume a 512-bit data word every cycle

- An asymmetric FIFO has different data bitwidth for the push and pop interfaces (usually integer multiples of each other)
- Can be implemented using multiple narrower FIFOs
  - **Wide to Narrow Conversion:** Push in parallel and pop sequentially
  - **Narrow to Wide Conversion:** Push sequentially and pop in parallel

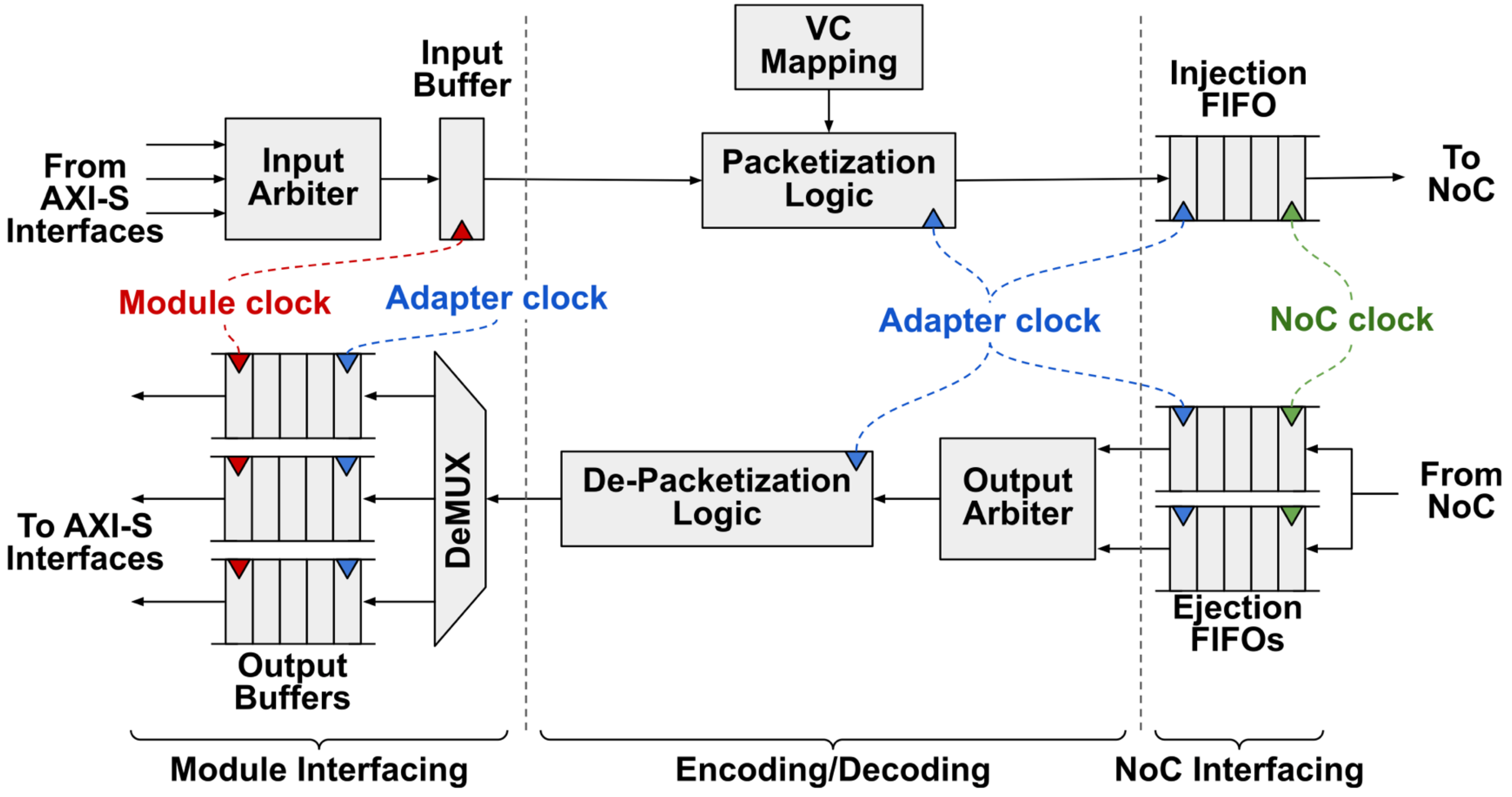
Asym. FIFO Implementation (Wide to Narrow)

```
module asym_fifo # (
    parameter IDATAW = 32,
    parameter ODATAW = 8,
    parameter DEPTH = 512,
    parameter NUM_FIFOS = IDATAW / ODATAW
)(
    input  clk,
    input  rst,
    input  [IDATAW-1:0] i_data,
    input  i_valid,
    output o_ready,
    output logic [ODATAW-1:0] o_data,
    output o_valid,
    input  i_ready
);

logic fifo_full, fifo_almost_full, fifo_empty;
logic fifo_push, fifo_pop;
logic [IDATAW-1:0] fifo_odata;
logic [$clog2(NUM_FIFOS)-1:0] cntr;

fifo # (
    .DATAW(IDATAW),
    .DEPTH(DEPTH)
) fifo_inst (
    .clk(clk),
    .rst(rst),
    .i_data(i_data),
    .i_push(fifo_push),
    .o_full(fifo_full),
    .o_almost_full(fifo_almost_full),
    .o_data(fifo_odata),
    .i_pop(fifo_pop),
    .o_empty(fifo_empty)
);
```

```
always_ff @ (posedge clk) begin
    if (rst) begin
        cntr <= 0;
    end else begin
        if (o_valid && i_ready) begin
            if (cntr == NUM_FIFOS-1) cntr <= 0;
            else cntr <= cntr + 1;
        end
    end
end

always_comb begin
    o_data = fifo_odata[cntr * ODATAW +: ODATAW];
end

assign fifo_push = i_valid && o_ready;
assign fifo_pop = o_valid && i_ready
                  && (cntr==NUM_FIFOS-1);
assign o_ready = ~fifo_almost_full;
assign o_valid = ~fifo_empty;

endmodule
```

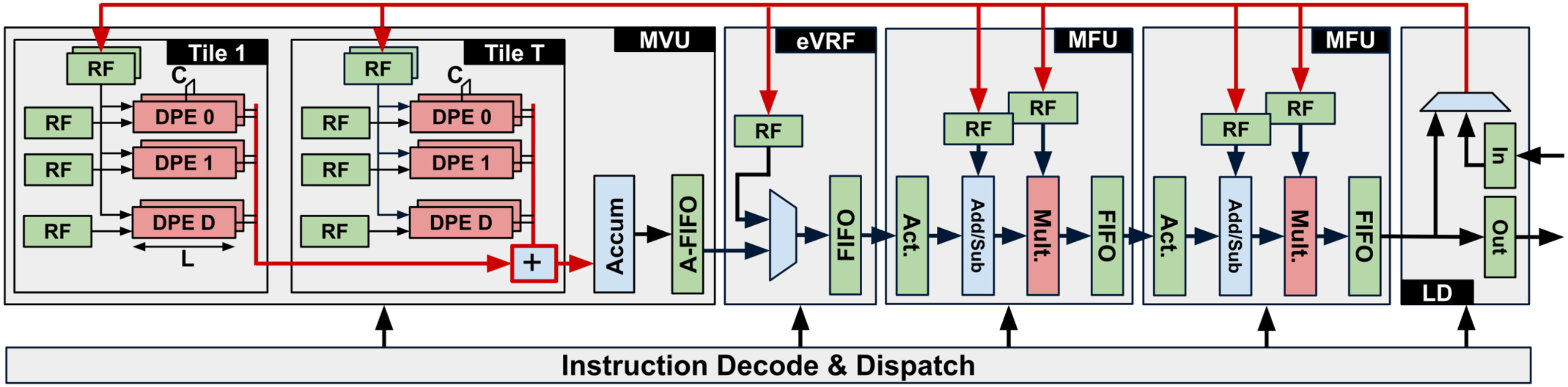
FIFO-Based Design in Practice

- In network-on-chip (NoC) router ports



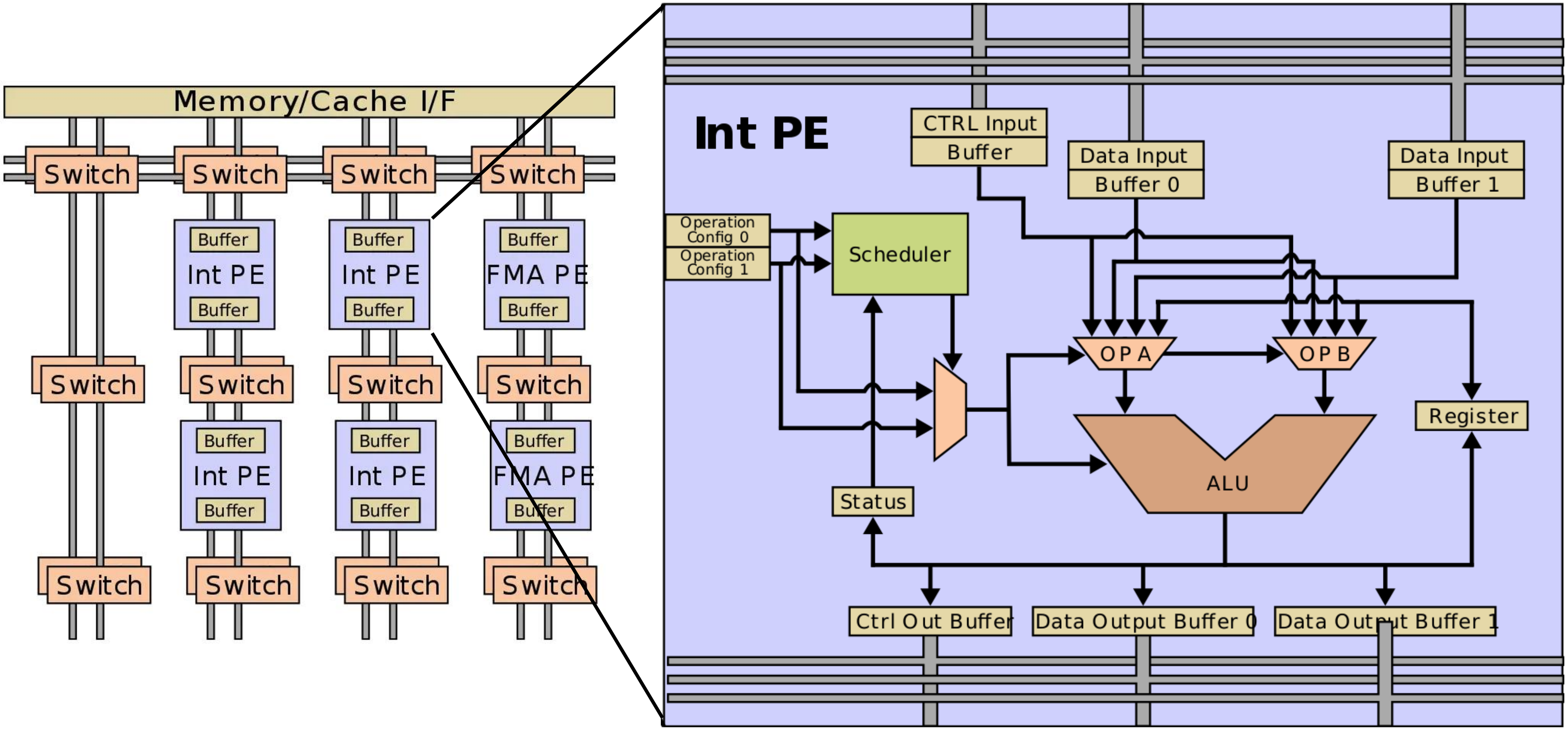
FIFO-Based Design in Practice

- In many accelerator designs (e.g., Microsoft Brainwave)



FIFO-Based Design in Practice

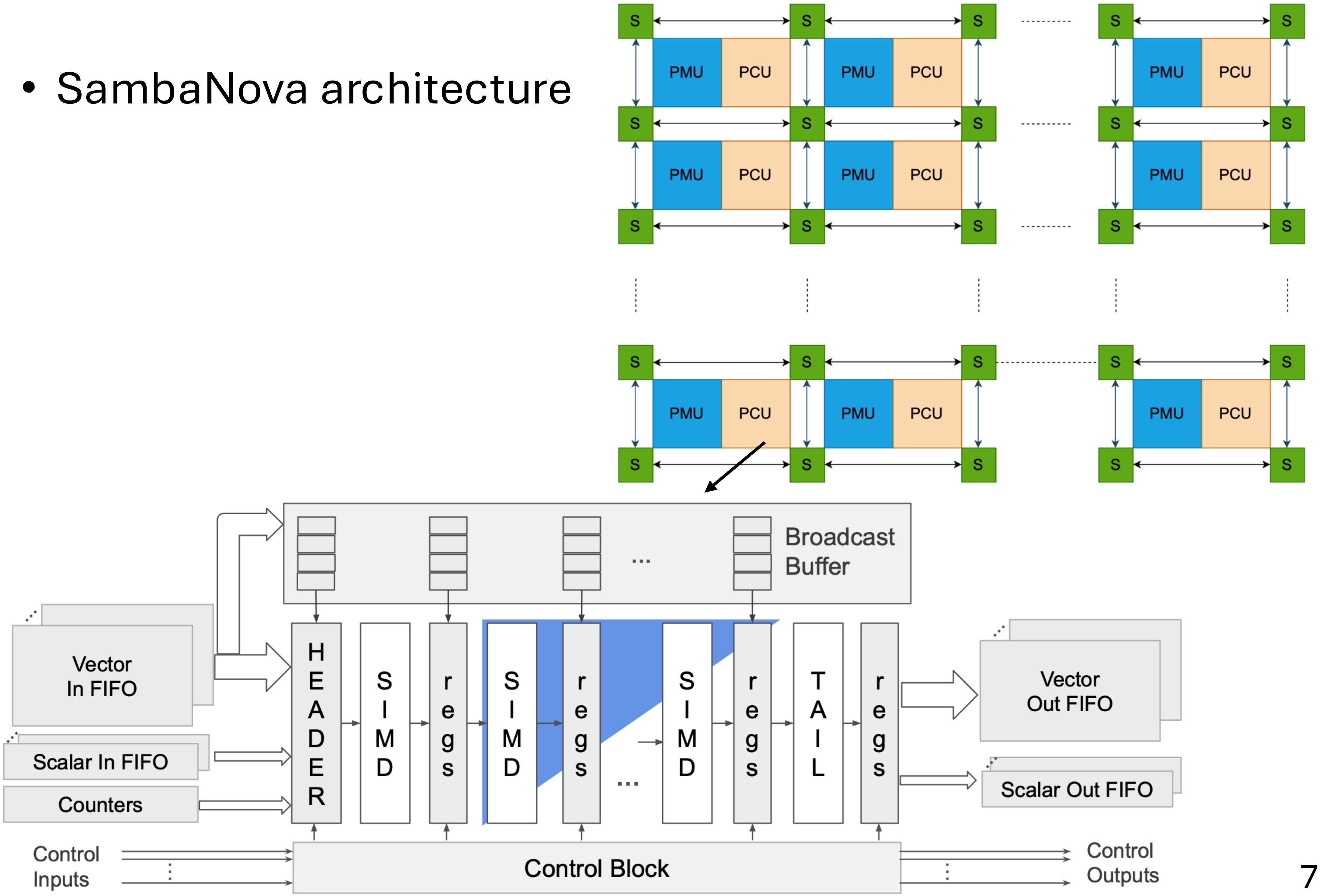
- Dataflow architectures (e.g., Intel Configurable Spatial Array)



Source: https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator

FIFO-Based Design in Practice

- SambaNova architecture


S
PMU
PCU
Broadcast Buffer
...
Vector In FIFO
Scalar In FIFO
Counters
HEADER
SIMD
regs
SIMD
regs
...
SIMD
regs
TAIL
regs
Vector Out FIFO
Scalar Out FIFO
Control Inputs
Control Block
Control Outputs