ECE 327/627
Digital Hardware Systems

Lecture 9: Latency Insensitive Design

Andrew Boutros

andrew.boutros@uwaterloo.ca

Some Logistics

- Lab 2 deadline is **tomorrow @ 11:59 pm**
  - Worth **8%** of the total ECE 327 grade
  - Worth **4%** of the total ECE 327 grade

- **[ECE 327]** Lab 2 assessment quiz in next week's lab sessions
  - Worth **40%** of the lab grade

In the Previous Lecture …

- Retiming
  - Algorithmic approach for improving frequency by moving around pipeline registers using simple rules
  - Preserves circuit behavior → does not add new registers
- Pipelining circuits with feedback loops
  - Possible but requires interleaving the computation of multiple streams of data to achieve correct functionality while having full throughput
- Throughput balancing in streaming architectures
  - Pipe and water analogy
  - Throughput of entire structure = the minimum throughput of all stages
  - Throughput balancing by module replication
  - Throughput balancing by multi-pumping

Chip-wide Interconnect Roadblock

- In the good old days, the delay of a digital circuit could be considered the sum of delays of cascaded logic gates

Chip-wide Interconnect Roadblock

- In the good old days, the delay of a digital circuit could be considered the sum of delays of cascaded logic gates
- This is no longer the case …
  - In modern process generations, wire delay is becoming dominant
  - Wires get thinner and pitch (i.e., spacing) between wires get smaller
    - Resistance & Capacitance increase for a constant wire length

Chip-wide Interconnect Roadblock

- In the good old days, the delay of a digital circuit could be considered the sum of delays of cascaded logic gates
- This is no longer the case …
  - In modern process generations, wire delay is becoming dominant
  - Wires get thinner and pitch (i.e., spacing) between wires get smaller
    - Resistance & Capacitance increase for a constant wire length
- Some interesting numbers to put things in perspective
  - At 60nm process technology, a signal can reach only 5% of the chip's length in a clock cycle [1]
  - A 1mm interconnect at 35nm process technology is ~100x longer than a transistor switching delay [2]
  - On a 14nm FPGA, chip-spanning connection delay is ~10ns [3]

[1] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?", IEEE Computer, 1997
[2] J. Davis, et al., "Interconnect Limits on Gigascale Integration in the 21st Century", Proceedings of the IEEE, 2001
[3] M. Abbas, V. Betz, "Latency Insensitive Design Styles for FPGAs", FPL, 2018

Chip-wide Interconnect Roadblock

- In the good old days, the delay of a digital circuit could be considered the sum of delays of cascaded logic gates
- This is no longer the case …
  - In modern process generations, wire delay is becoming dominant
  - Wires get thinner and pitch (i.e., spacing) between wires get smaller
    - Resistance & Capacitance increase for a constant wire length
- Some interesting numbers to put things in perspective
  - At 60nm process technology, a signal can reach only 5% of the chip's length in a clock cycle [1]
  - A 1mm interconnect at 35nm process technology is ~100x longer than a transistor switching delay [2]
  - On a 14nm FPGA, chip-spanning connection delay is ~10ns [3]
- Most problematic for "global" inter-module connections

[1] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?", IEEE Computer, 1997
[2] J. Davis, et al., "Interconnect Limits on Gigascale Integration in the 21st Century", Proceedings of the IEEE, 2001
[3] M. Abbas, V. Betz, "Latency Insensitive Design Styles for FPGAs", FPL, 2018

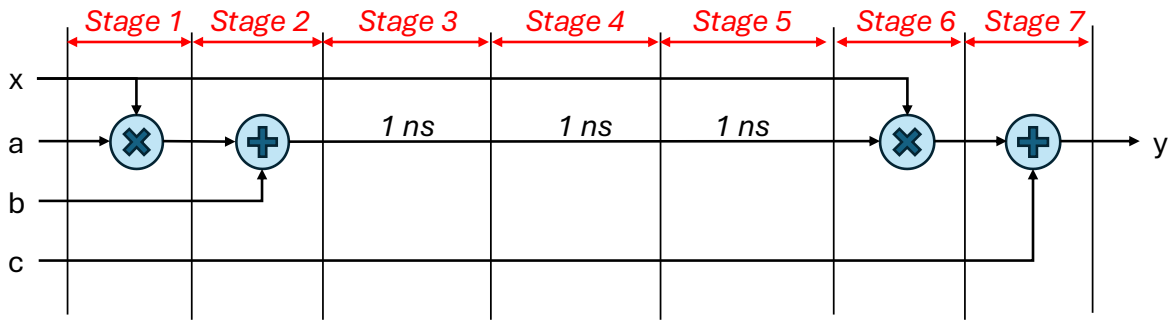
Luckily, We Know a Trick …

- Pipelining “global” wires to improve operating frequency is now a necessity

Luckily, We Know a Trick …

- Pipelining “global” wires to improve operating frequency is now a necessity

- We already know how to pipeline wires – it’s simple!

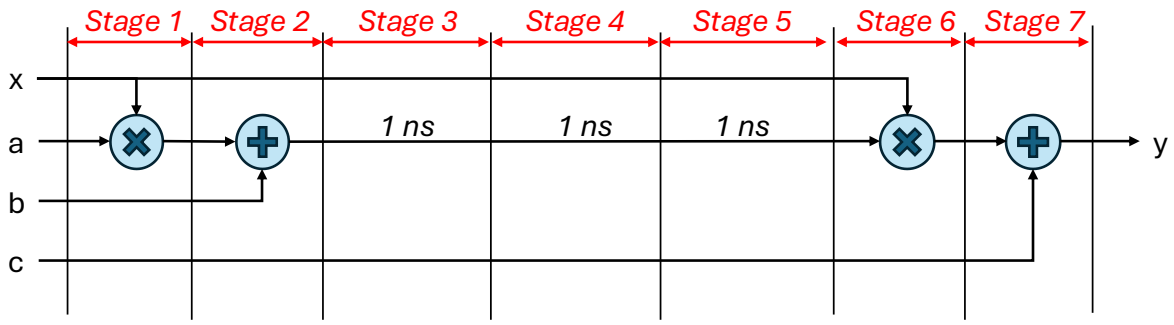



```
module wire_pipeline # (
      parameter DATAW = 8,
      parameter NUM_STAGES = 3
)(
      input  clk,
      input  rst,
      input  [DATAW-1:0] data_in,
      output [DATAW-1:0] data_out
);

logic [DATAW-1:0] r_data [0:NUM_STAGES-1];
integer i;

always_ff @ (posedge clk) begin
    if (rst) begin
        for (i=0; i<NUM_STAGES; i=i+1) begin
            r_data[i] <= 0;
        end
    end else begin
        r_data[0] <= data_in;
        for (i=1; i<NUM_STAGES; i=i+1) begin
            r_data[i] <= r_data[i-1];
        end
    end
end
assign data_out = r_data[NUM_STAGES-1];
endmodule
```

Luckily, We Know a Trick …

- Pipelining “global” wires to improve operating frequency is now a necessity

- We already know how to pipeline wires – it’s simple!

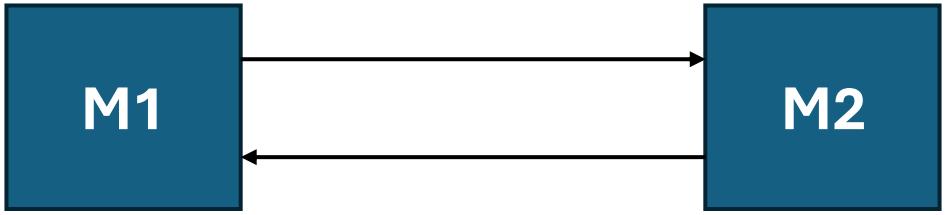



```
module wire_pipeline # (
      parameter DATAW = 8,
      parameter NUM_STAGES = 3
)(
      input  clk,
      input  rst,
      input  [DATAW-1:0] data_in,
      output [DATAW-1:0] data_out
);

logic [DATAW-1:0] r_data [0:NUM_STAGES-1];
integer i;

always_ff @ (posedge clk) begin
    if (rst) begin
        for (i=0; i<NUM_STAGES; i=i+1) begin
            r_data[i] <= 0;
        end
    end else begin
        r_data[0] <= data_in;
        for (i=1; i<NUM_STAGES; i=i+1) begin
            r_data[i] <= r_data[i-1];
        end
    end
end
assign data_out = r_data[NUM_STAGES-1];
endmodule
```

- As many other things in life, turns out it is not as simple as it seems!

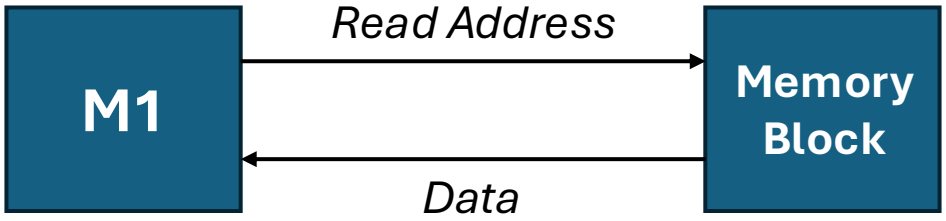
Problem with Pipelining Wires

Let's think about the simple case of communication between two modules on the chip …



Problem with Pipelining Wires

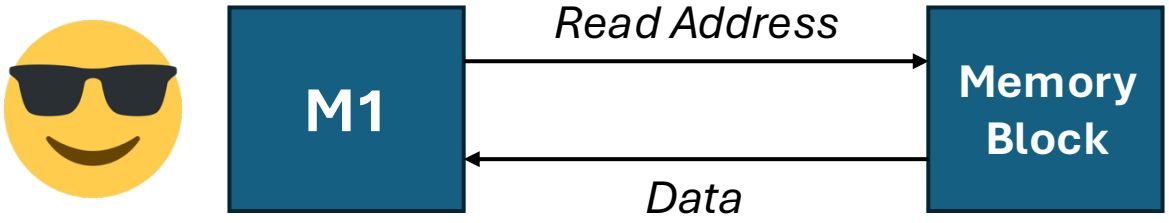
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M1 is designed to issue an address & exactly 2 cycles later use the data to perform some functionality

Problem with Pipelining Wires

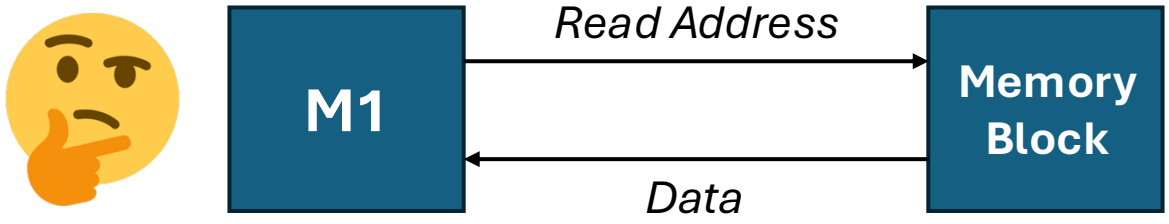
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M1 is designed to issue an address & exactly 2 cycles later use the data to perform some functionality

Problem with Pipelining Wires

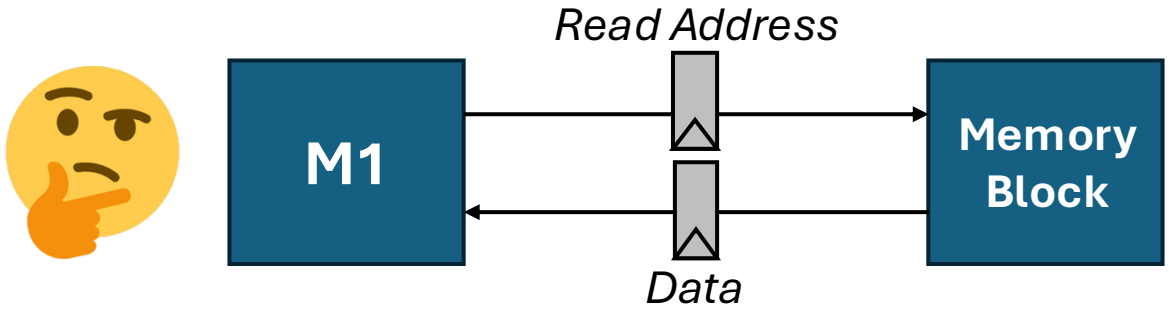
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M1 is designed to issue an address & exactly 2 cycles later use the data to perform some functionality
- Run through the implementation CAD tools → not meeting timing because of the wire delays between the two modules

Problem with Pipelining Wires

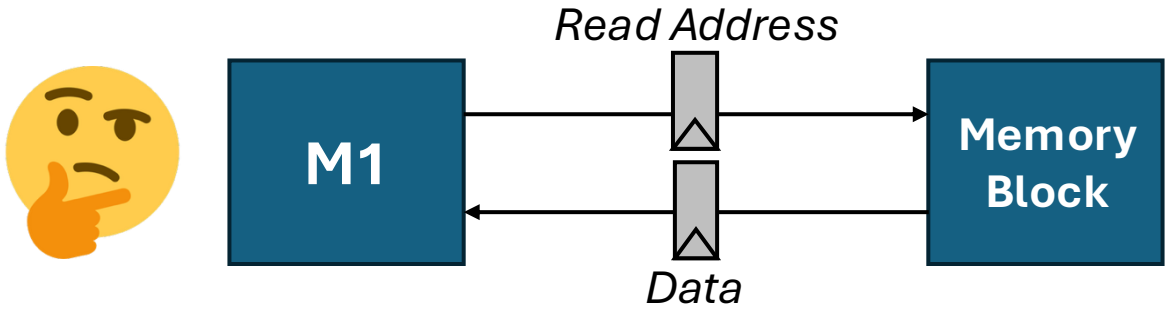
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M1 is designed to issue an address & exactly 2 cycles later use the data to perform some functionality
- Run through the implementation CAD tools → not meeting timing because of the wire delays between the two modules

Problem with Pipelining Wires

Let's think about the simple case of communication between two modules on the chip …

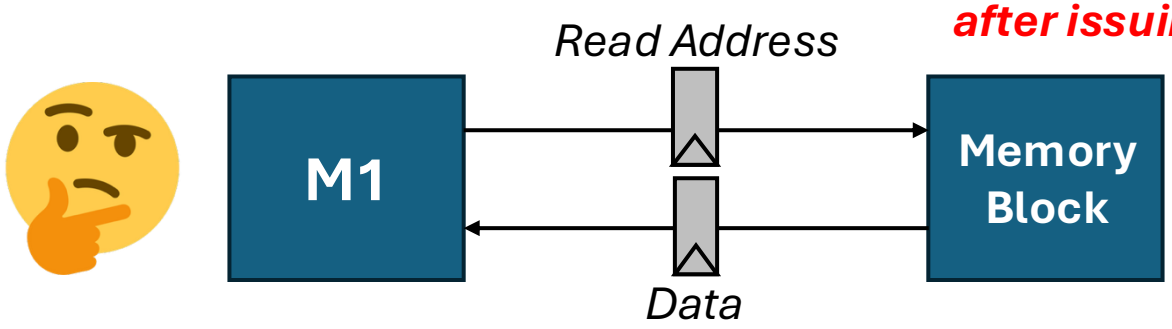


- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M1 is designed to issue an address & **exactly 2 cycles later** use the data to perform some functionality
- Run through the implementation CAD tools → not meeting timing because of the wire delays between the two modules

Problem with Pipelining Wires

Let's think about the simple case of communication between two modules on the chip …

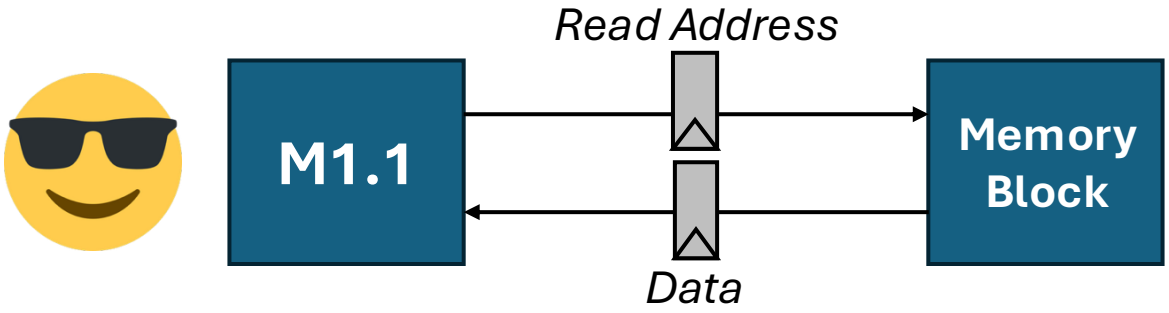
But now M1 should use data 4 cycles after issuing an address!



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M1 is designed to issue an address & **exactly 2 cycles later** use the data to perform some functionality
- Run through the implementation CAD tools → not meeting timing because of the wire delays between the two modules

Problem with Pipelining Wires

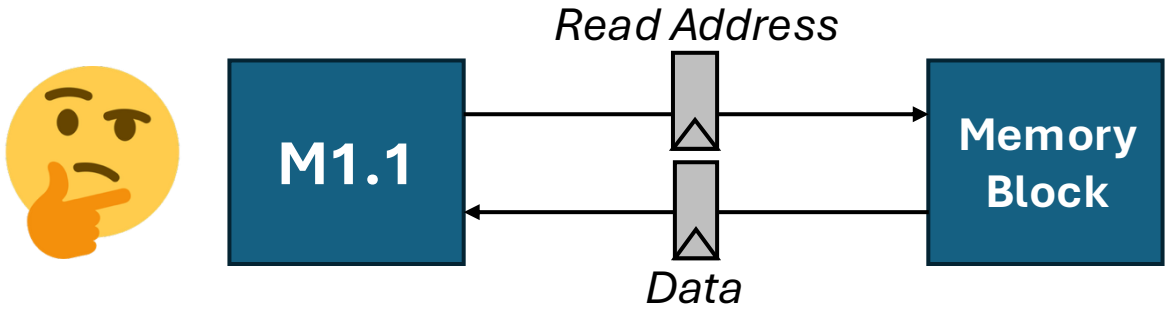
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M**1.1** is designed to issue an address & **exactly 4 cycles later** use the data to perform some functionality

Problem with Pipelining Wires

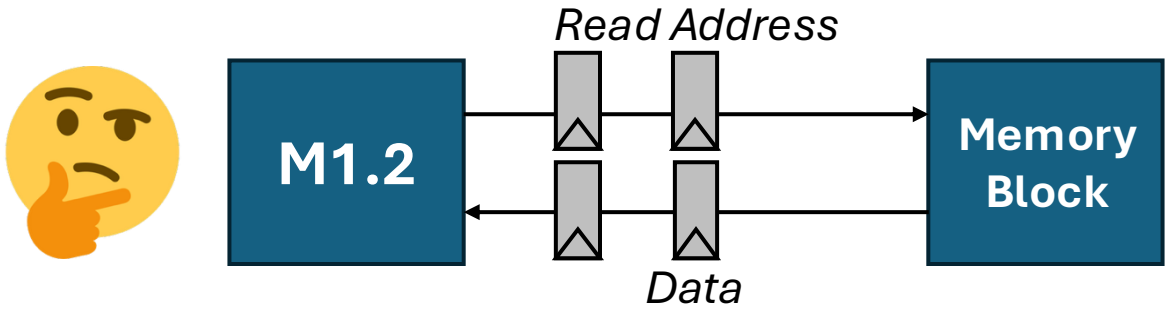
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M**1.1** is designed to issue an address & **exactly 4 cycles later** use the data to perform some functionality
- Run through the implementation CAD tools → not meeting timing because of the wire delays between the two modules

Problem with Pipelining Wires

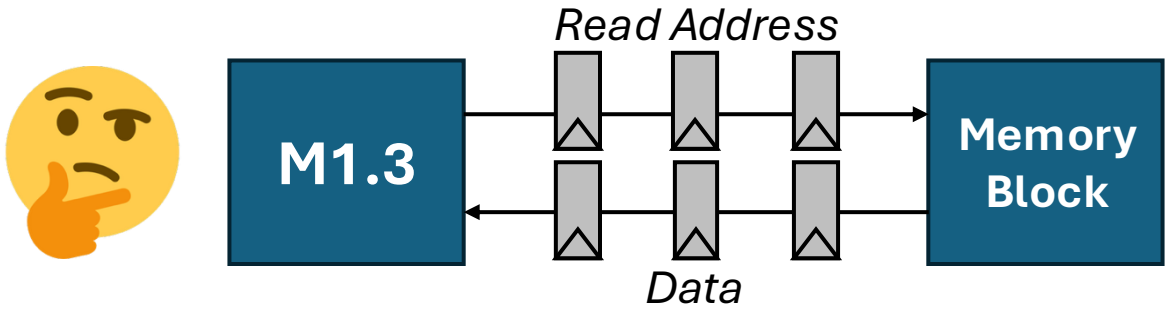
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M**1.2** is designed to issue an address & **exactly 6 cycles later** use the data to perform some functionality
- Run through the implementation CAD tools → not meeting timing because of the wire delays between the two modules

Problem with Pipelining Wires

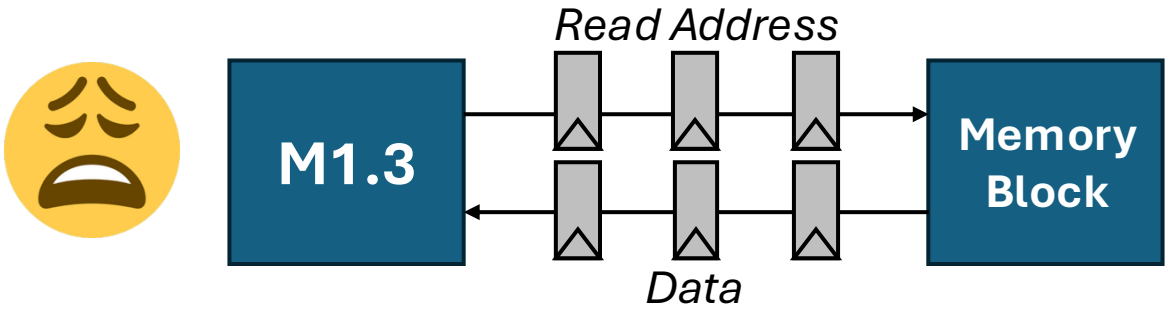
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M**1.3** is designed to issue an address & **exactly 8 cycles later** use the data to perform some functionality

Problem with Pipelining Wires

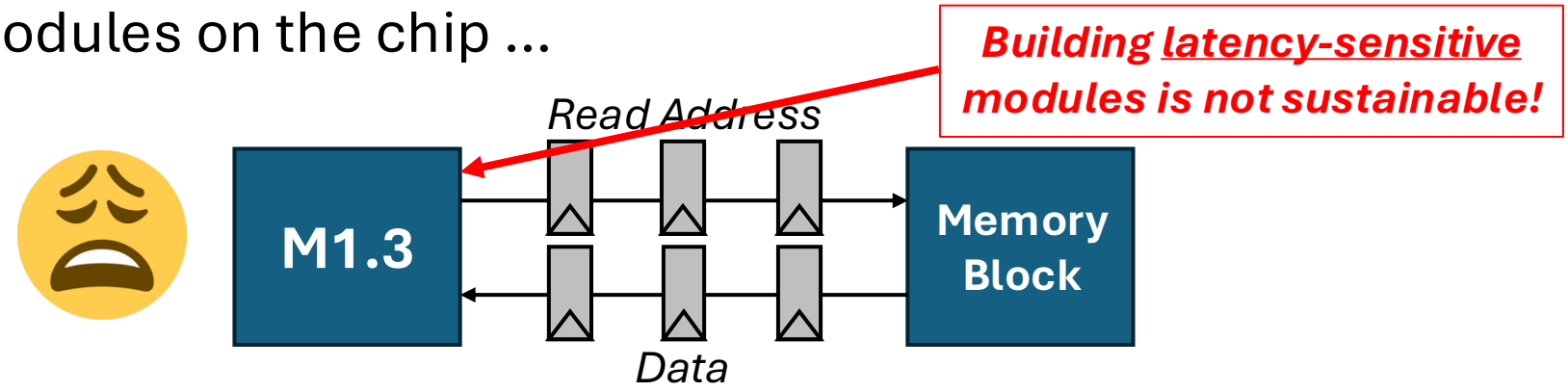
Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M**1.3** is designed to issue an address & **exactly 8 cycles later** use the data to perform some functionality
- Now it's time to move to different process generation or a different FPGA device!!

Problem with Pipelining Wires

Let's think about the simple case of communication between two modules on the chip …



- Assume the memory block takes 2 cycles to output the data stored in a specific address
- M**1.3** is designed to issue an address & **exactly 8 cycles later** use the data to perform some functionality
- Now it's time to move to different process generation or a different FPGA device!!

Latency-Insensitive Design (LID)

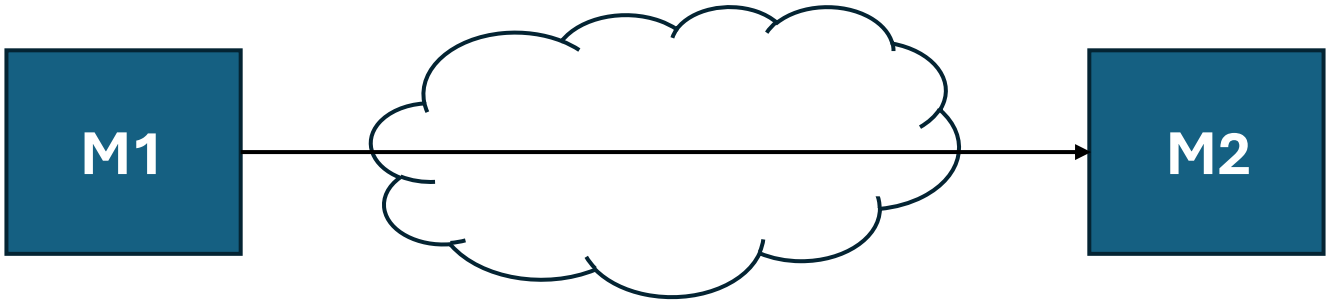
- Decouple compute functionality & communication

Latency-Insensitive Design (LID)

- Decouple compute functionality & communication
- Communication between modules happen over “channels”
  - Channels can have arbitrary latencies (hence the name “latency-insensitive” channel)
  - Modules perform computation whenever **valid** inputs arrive, and they are **ready** to accept it
  - Regardless of when that happens, it does not affect functionality

Latency-Insensitive Design (LID)

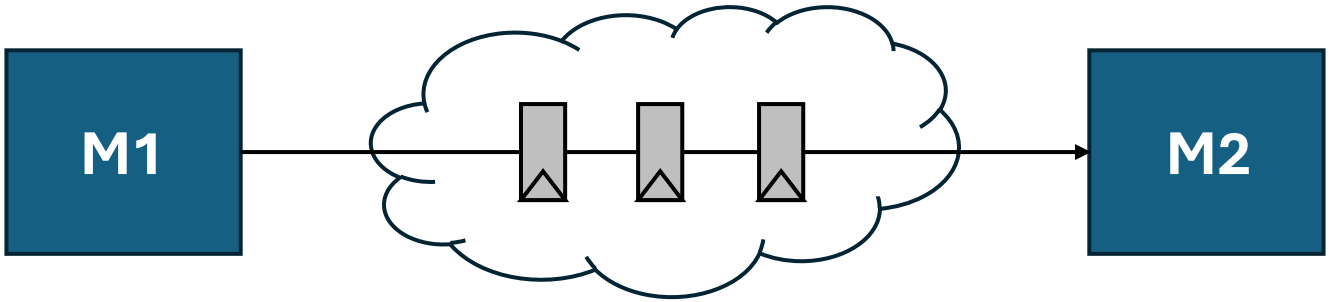
- Decouple compute functionality & communication
- Communication between modules happen over “channels”
  - Channels can have arbitrary latencies (hence the name “latency-insensitive” channel)
  - Modules perform computation whenever **valid** inputs arrive, and they are **ready** to accept it
  - Regardless of when that happens, it does not affect functionality



*A communication channel could be **simple wires***

Latency-Insensitive Design (LID)

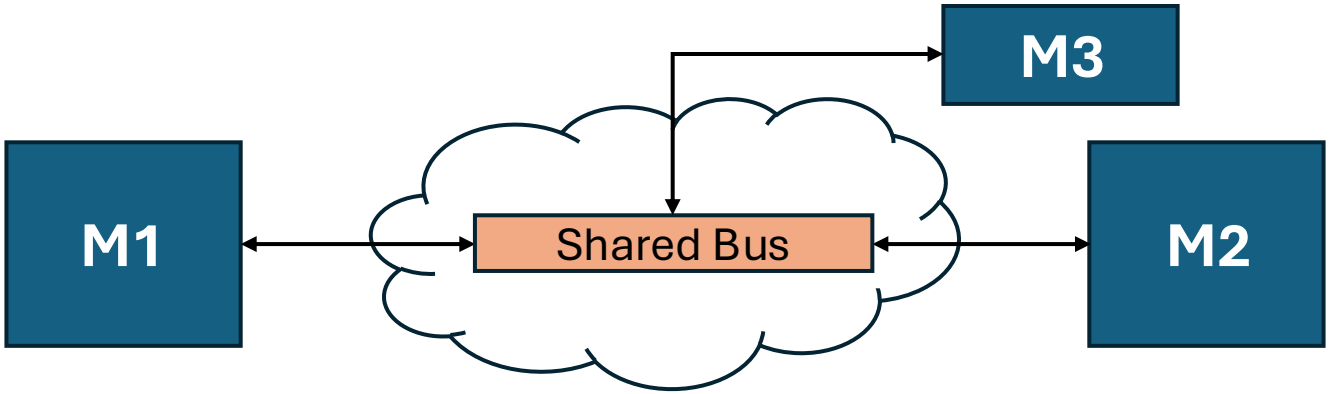
- Decouple compute functionality & communication
- Communication between modules happen over “channels”
  - Channels can have arbitrary latencies (hence the name “latency-insensitive” channel)
  - Modules perform computation whenever **valid** inputs arrive, and they are **ready** to accept it
  - Regardless of when that happens, it does not affect functionality



*A communication channel could be **pipelined wires***

Latency-Insensitive Design (LID)

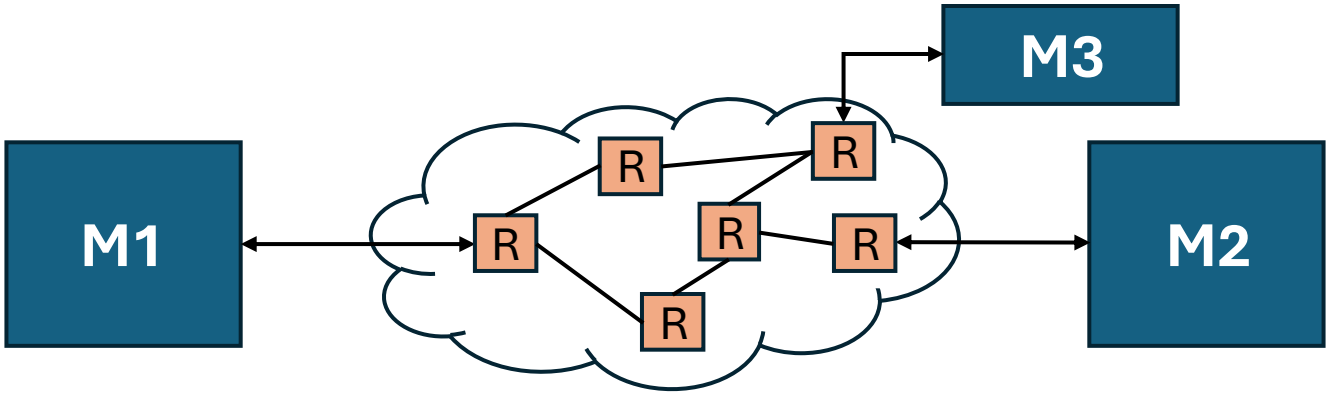
- Decouple compute functionality & communication
- Communication between modules happen over “channels”
  - Channels can have arbitrary latencies (hence the name “latency-insensitive” channel)
  - Modules perform computation whenever **valid** inputs arrive, and they are **ready** to accept it
  - Regardless of when that happens, it does not affect functionality



*A communication channel could be **shared bus***

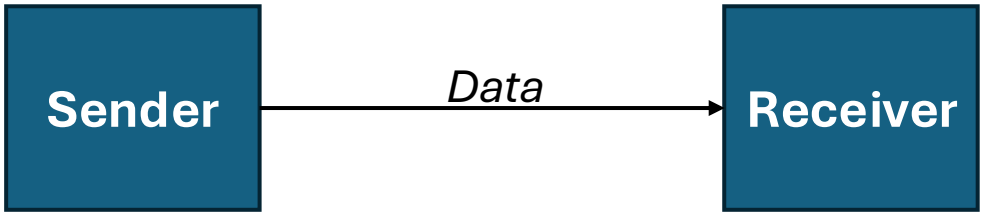
Latency-Insensitive Design (LID)

- Decouple compute functionality & communication
- Communication between modules happen over “channels”
  - Channels can have arbitrary latencies (hence the name “latency-insensitive” channel)
  - Modules perform computation whenever **valid** inputs arrive, and they are **ready** to accept it
  - Regardless of when that happens, it does not affect functionality



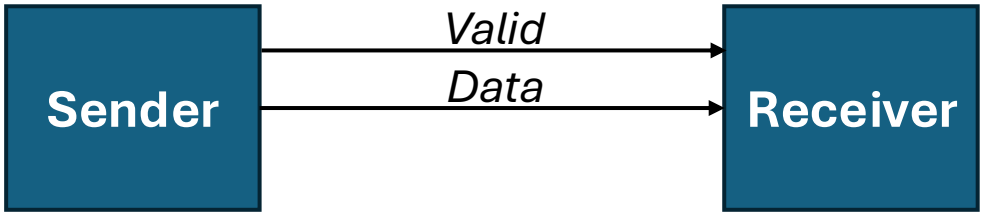
*A communication channel could be **network-on-chip (NoC)***

Ready-Valid Handshake



Since there is no strict requirement on when data arrives …

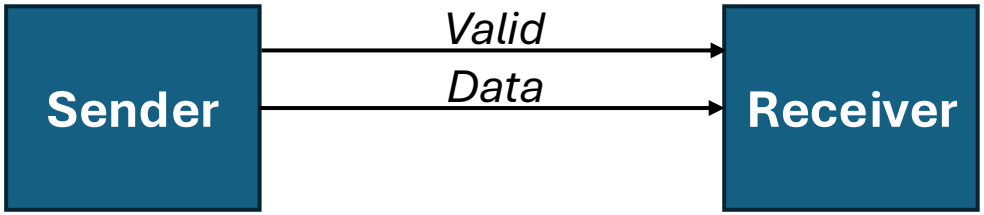
Ready-Valid Handshake



Since there is no strict requirement on when data arrives …

- Sender must be able to indicate that it is safe for the receiver to read and use the data → **Valid** bit

Ready-Valid Handshake

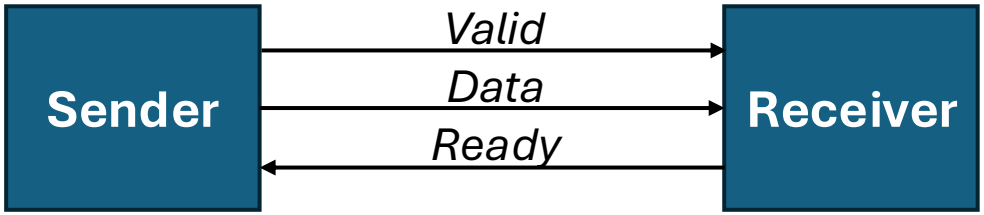


Since there is no strict requirement on when data arrives …

- Sender must be able to indicate that it is safe for the receiver to read and use the data → **Valid** bit

Similar to what we discussed when handling pipeline bubbles!

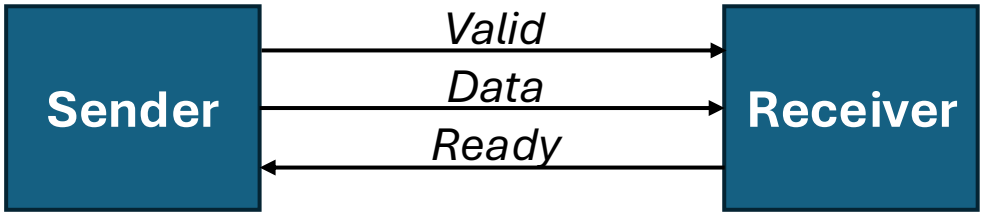
Ready-Valid Handshake



Since there is no strict requirement on when data arrives …

- Sender must be able to indicate that it is safe for the receiver to read and use the data → **Valid** bit
- If receiver cannot accept data, sender must be notified to hold the message → **Ready** bit

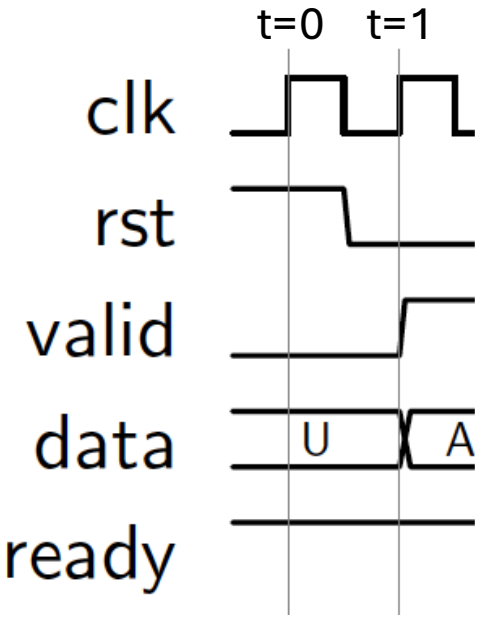
Ready-Valid Handshake



Since there is no strict requirement on when data arrives …

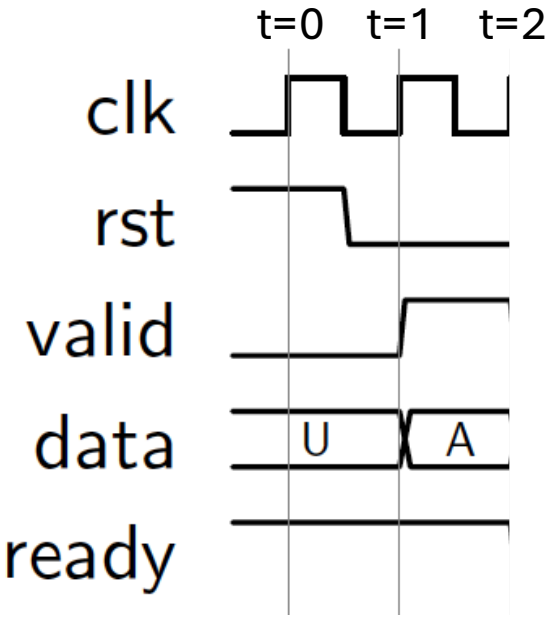
- Sender must be able to indicate that it is safe for the receiver to read and use the data → **Valid** bit
- If receiver cannot accept data, sender must be notified to hold the message → **Ready** bit
- States of the handshake:
  - If valid == 1 & ready == 1 → transmission happens
  - If valid == 1 & ready == 0 → sender is holding the data
  - If valid == 0 & ready == X → there is nothing to send (channel is idle)

Ready-Valid Handshake Expected Behavior



- At t=1, Sender produced data ‘A’ and tagged it as valid

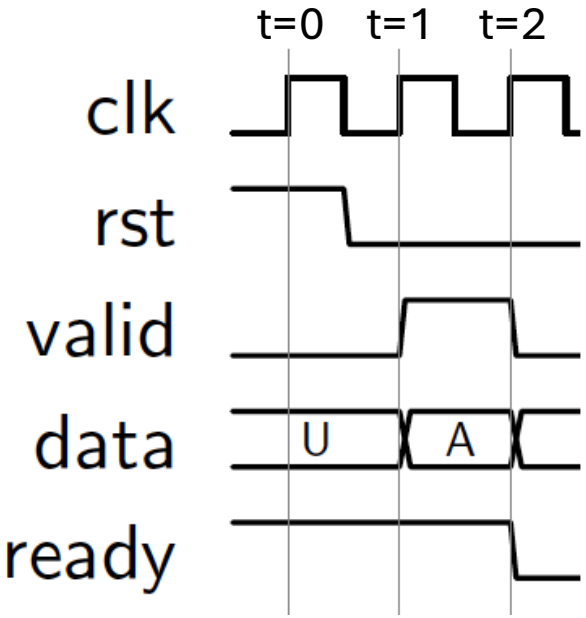
Ready-Valid Handshake Expected Behavior



- At t=1, Sender produced data ‘A’ and tagged it as valid

- At t=2, transmission of ‘A’ happens (valid && ready)

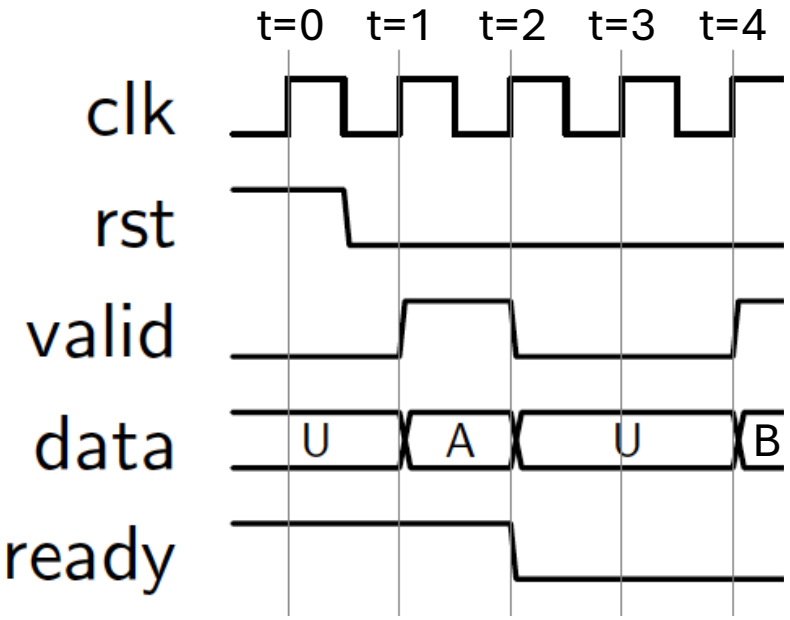
Ready-Valid Handshake Expected Behavior



- At t=1, Sender produced data ‘A’ and tagged it as valid

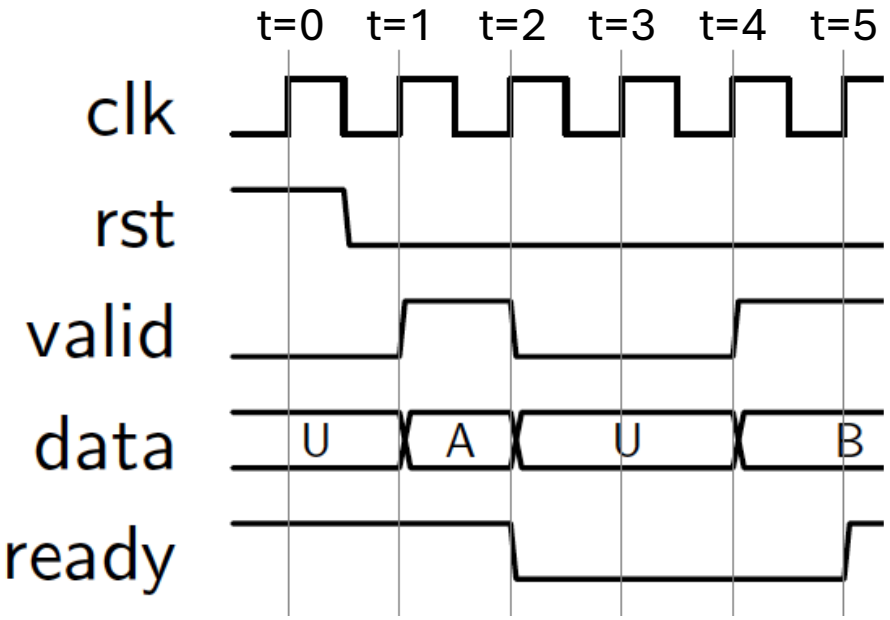
- At t=2, transmission of ‘A’ happens (valid && ready)
  - Sender has no more data to send → valid is de-asserted
  - Receiver not ready to accept more data → ready is de-asserted

Ready-Valid Handshake Expected Behavior



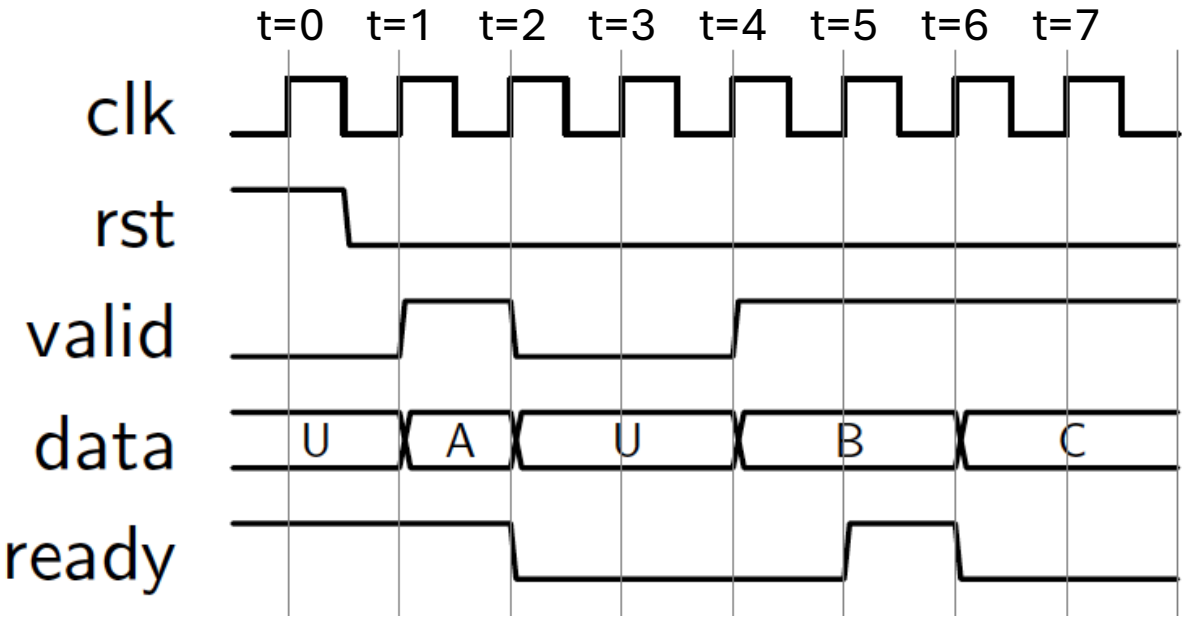
- At t=4, although receiver is not ready, sender produced data ‘B’ and tagged it as valid

Ready-Valid Handshake Expected Behavior



- At t=4, although receiver is not ready, sender produced data ‘B’ and tagged it as valid
- At t=5, receiver became ready again!

Ready-Valid Handshake Expected Behavior



- At t=4, although receiver is not ready, sender produced data ‘B’ and tagged it as valid – has to be held until receiver is ready
- At t=5, receiver became ready again!
- At t=6, transmission of ‘B’ happens (valid && ready)
  - Sender has more data ‘C’ but receiver is not ready to accept it

If Receiver not Ready, Stall the Pipeline!

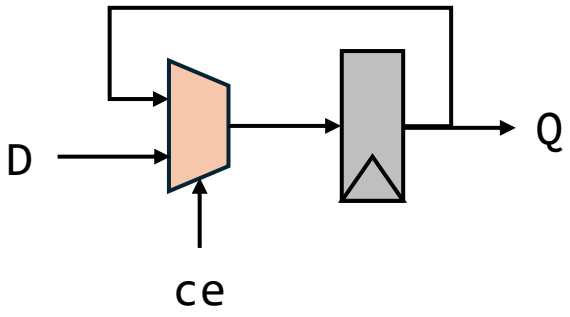
- A pipelined module (or a pipelined communication channel) has multiple pieces of data “in-flight” at different stages of the pipeline

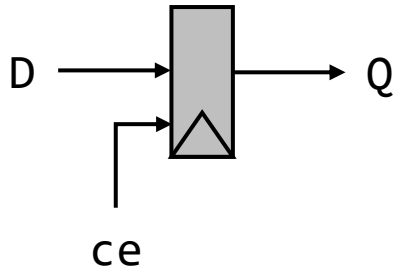
If Receiver not Ready, Stall the Pipeline!

- A pipelined module (or a pipelined communication channel) has multiple pieces of data “in-flight” at different stages of the pipeline
- If receiver is not ready, we need to freeze the pipeline in-place so data is not lost
  - This is known as “stalling” the pipeline

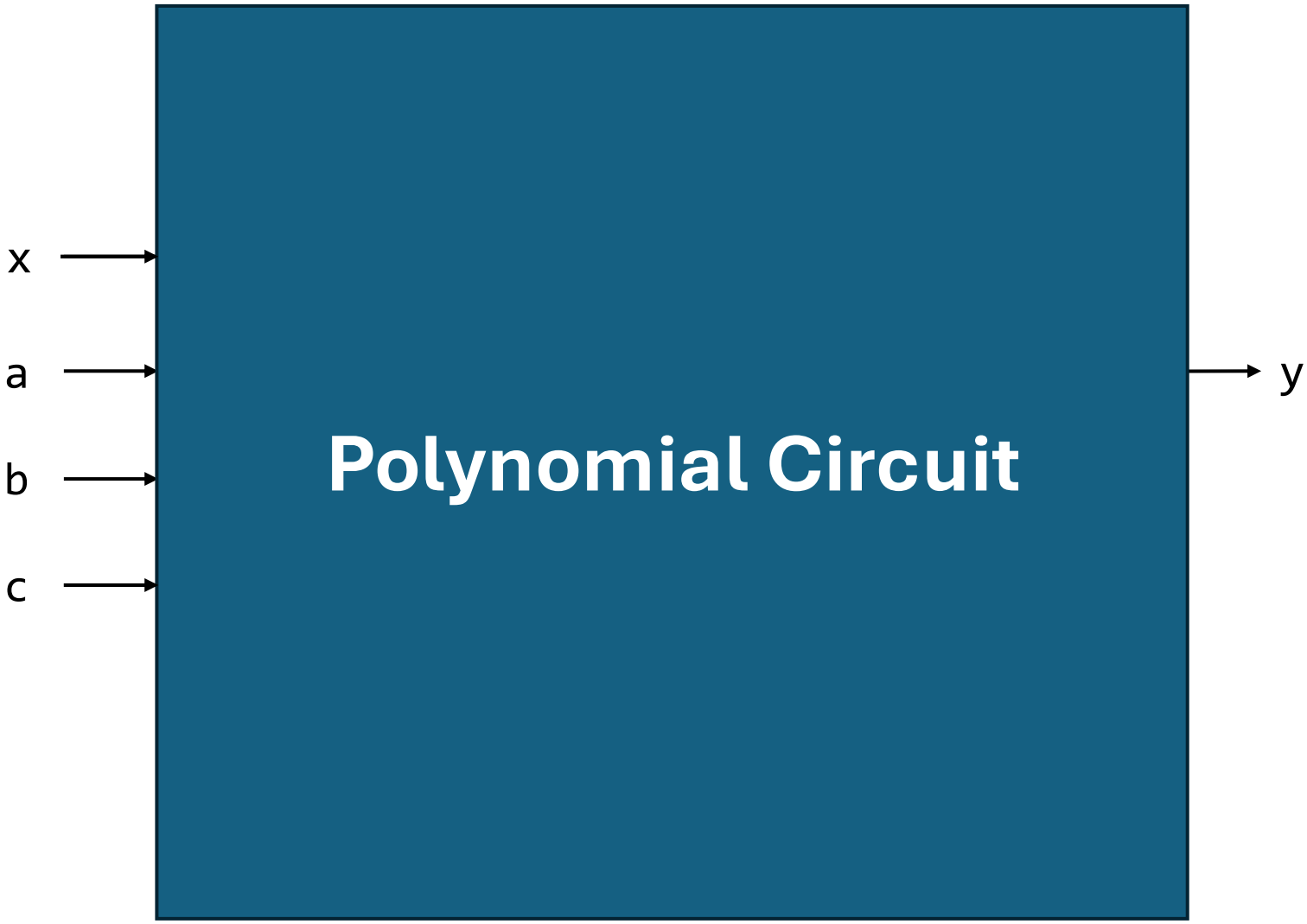
If Receiver not Ready, Stall the Pipeline!

- A pipelined module (or a pipelined communication channel) has multiple pieces of data “in-flight” at different stages of the pipeline
- If receiver is not ready, we need to freeze the pipeline in-place so data is not lost
  - This is known as “stalling” the pipeline
- Registers typically have a clock enable (ce) signal
  - If ce = 1, input data is registered at clock edge
  - If ce = 0, register keeps its value (i.e., nothing happens at clock edge)

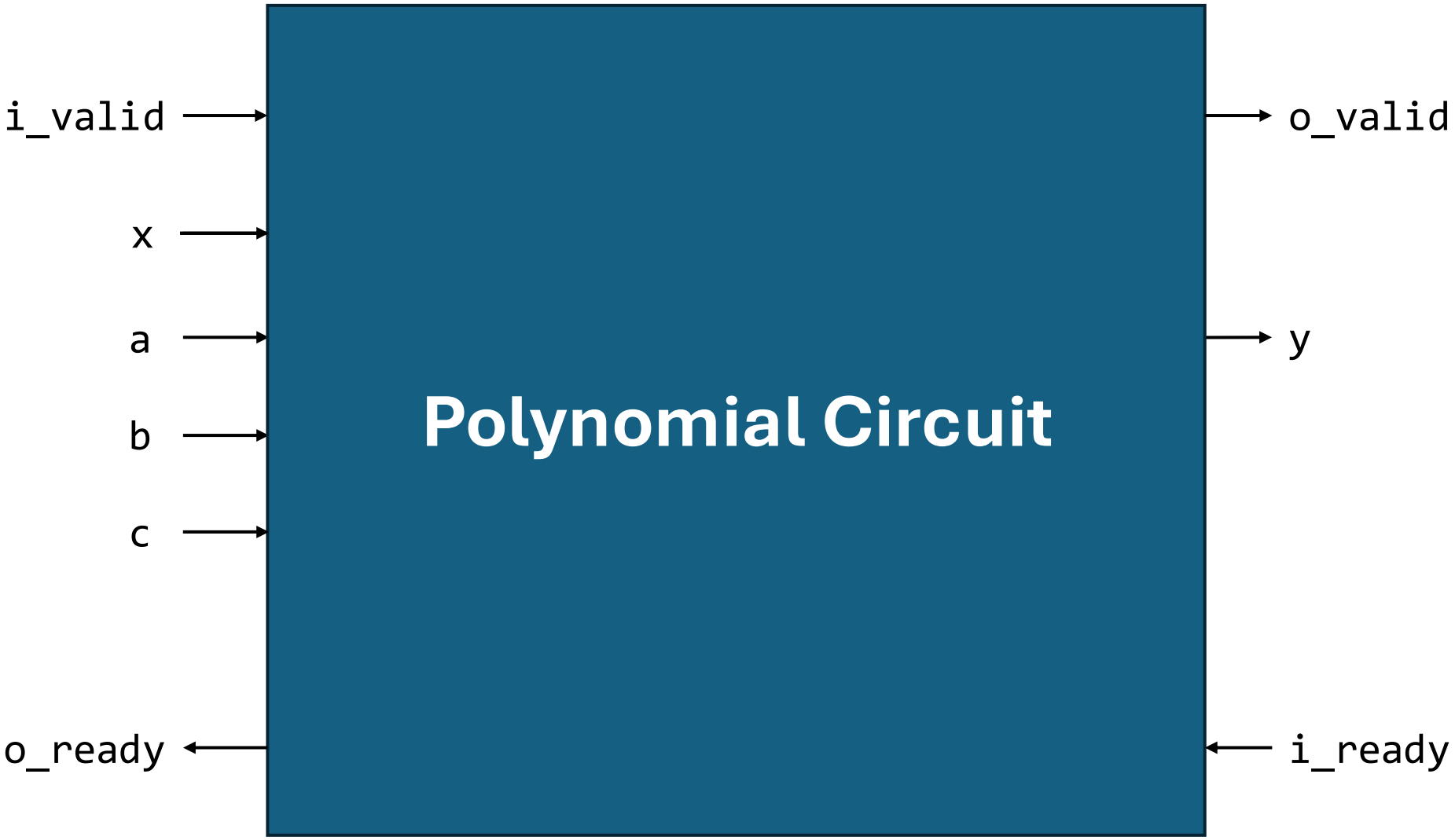




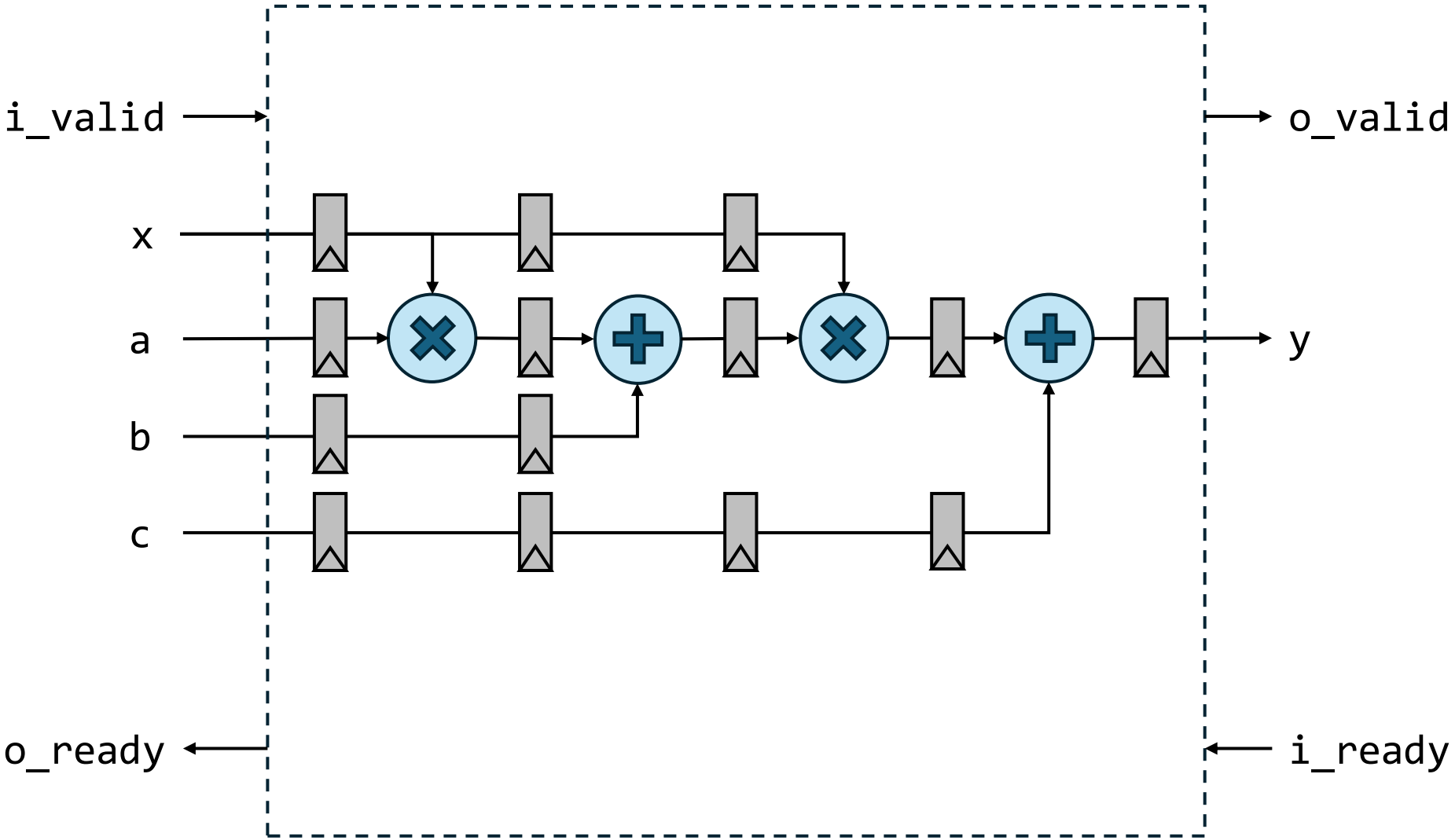
Example: Latency Insensitive Polynomial Circuit



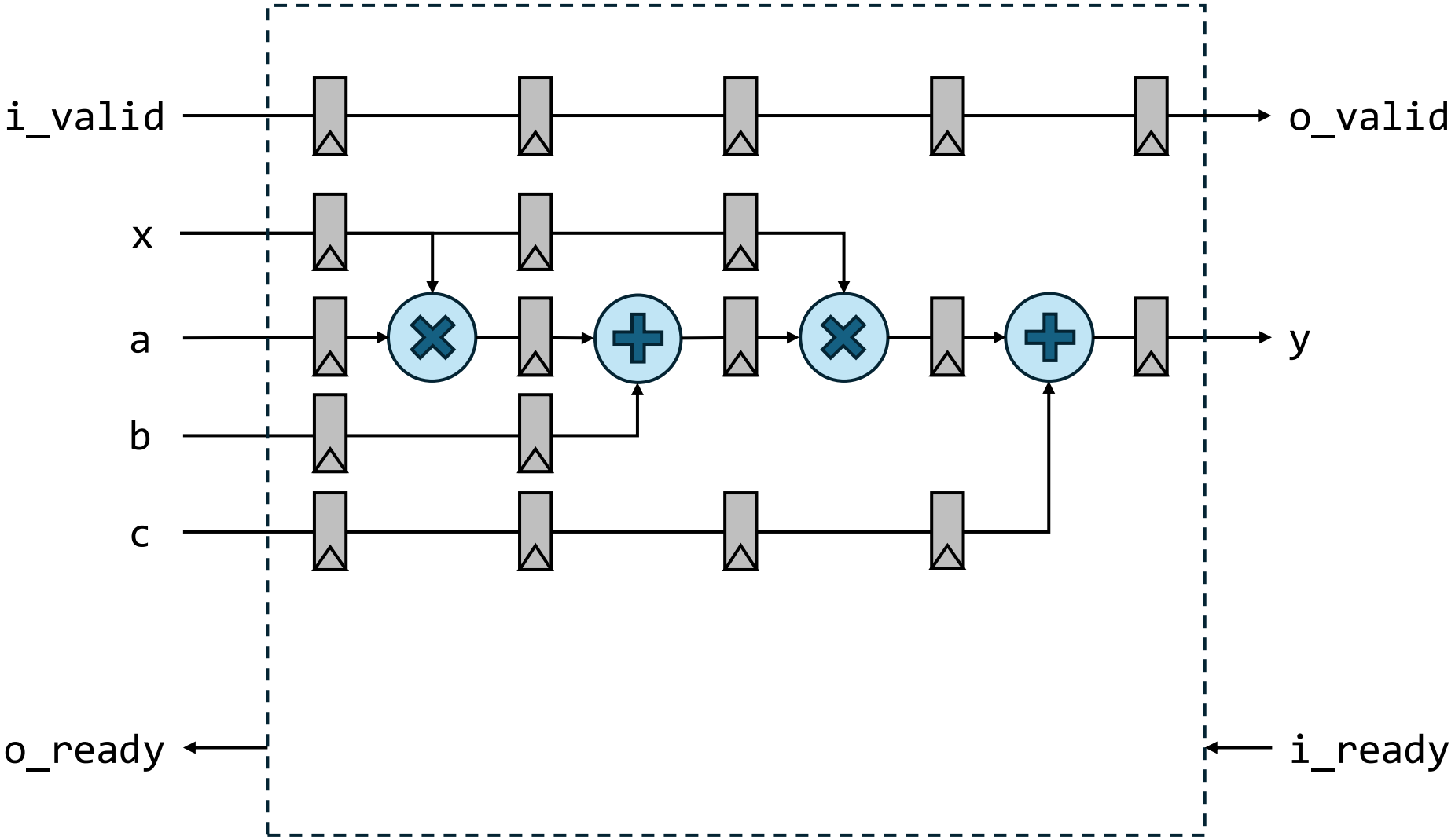
Example: Latency Insensitive Polynomial Circuit



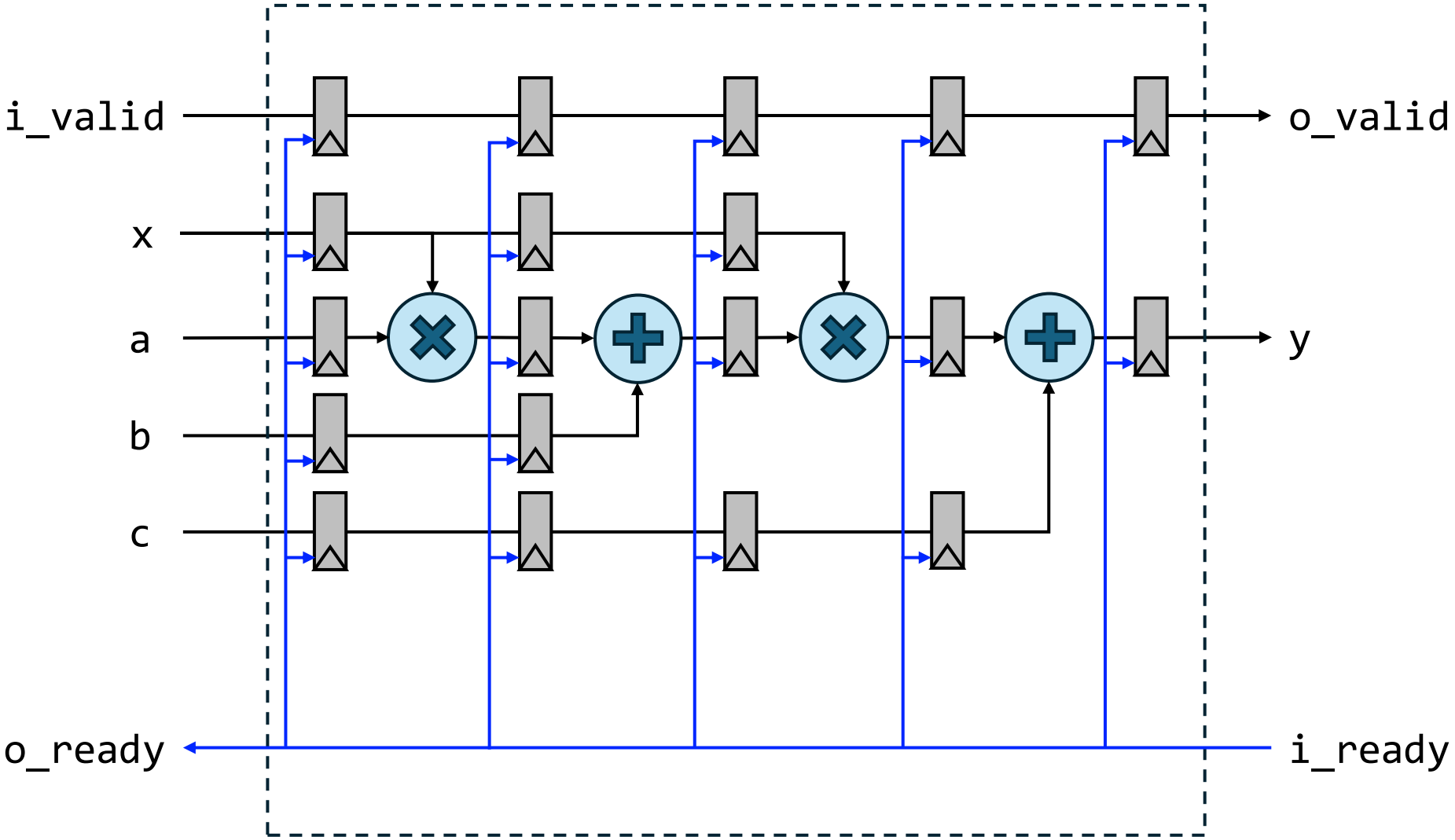
Example: Latency Insensitive Polynomial Circuit



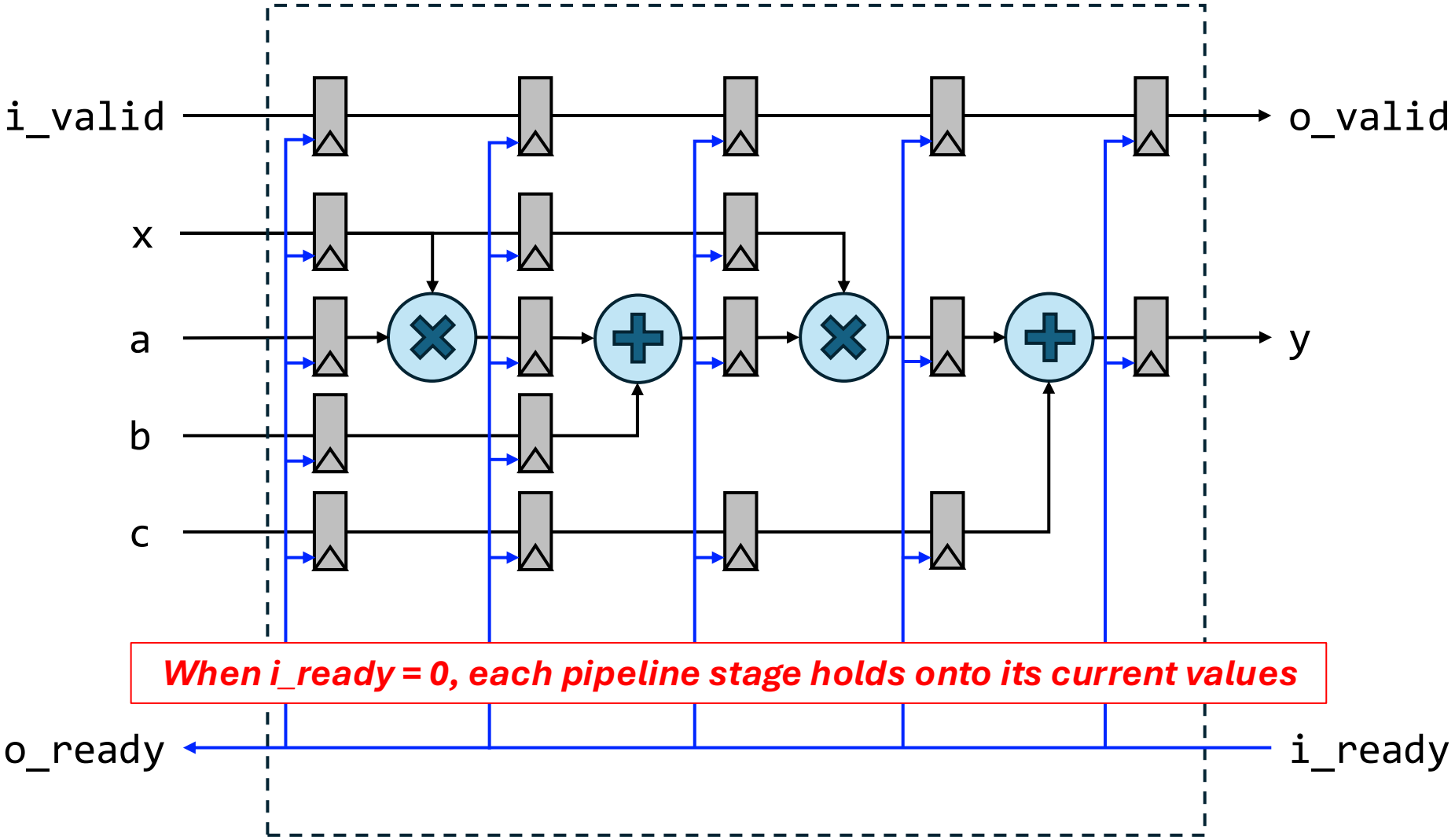
Example: Latency Insensitive Polynomial Circuit



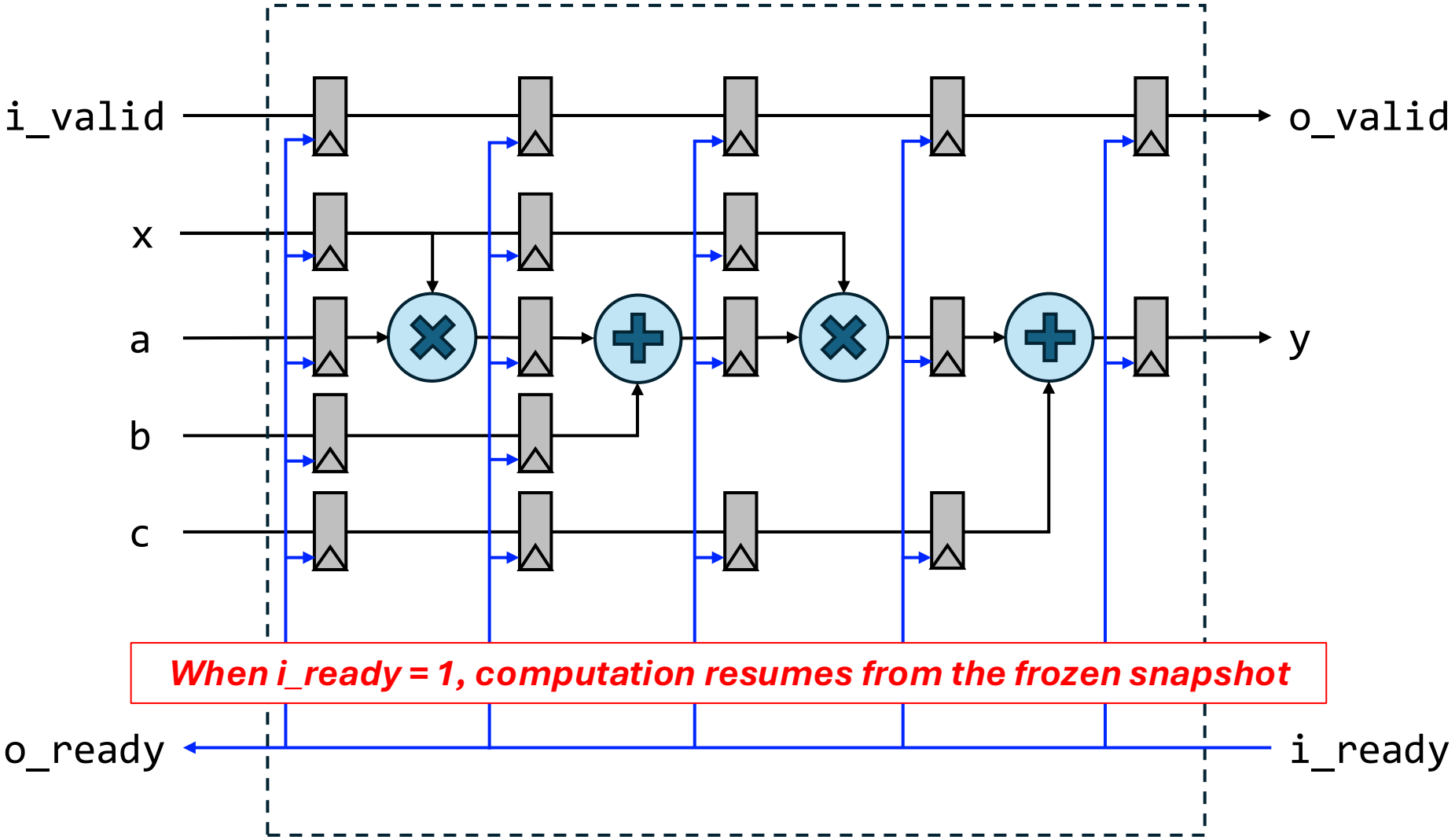
Example: Latency Insensitive Polynomial Circuit



Example: Latency Insensitive Polynomial Circuit



Example: Latency Insensitive Polynomial Circuit



Example: Latency Insensitive Polynomial Circuit

```
module poly_li (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y




);

// Input registers
logic [7:0] r_a, r_b, r_c, r_x;
// Stage 1 registers
logic [15:0] r_y1;
logic [7:0] r_x1, r_b1, r_c1;
// Stage 2 registers
logic [15:0] r_y2;
logic [7:0] r_x2, r_c2;
// Stage 3 registers
logic [23:0] r_y3;
logic [7:0] r_c3;



always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 0; r_y2 <= 0; r_y3 <= 0;
        r_x1 <= 0; r_b1 <= 0; r_c1 <= 0;
        r_x2 <= 0; r_c2 <= 0;
        r_c3 <= 0; y <= 0;


    end else begin
        // Input registers
        r_a <= a; r_b <= b;
        r_c <= c; r_x <= x;

        // Stage 1
        r_y1 <= (r_a * r_x);
        r_x1 <= r_x; r_b1 <= r_b; r_c1 <= r_c;

        // Stage 2
        r_y2 <= r_y1 + r_b1;
        r_x2 <= r_x1; r_c2 <= r_c1;

        // Stage 3
        r_y3 <= r_y2 * r_x2;
        r_c3 <= r_c2;

        // Stage 4
        y <= r_y3 + r_c3;

    end
end



endmodule
```

Example: Latency Insensitive Polynomial Circuit

```
module poly_li (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y,
      input i_valid,
      input i_ready,
      output logic o_valid,
      output o_ready
);

// Input registers
logic [7:0] r_a, r_b, r_c, r_x;
// Stage 1 registers
logic [15:0] r_y1;
logic [7:0] r_x1, r_b1, r_c1;
// Stage 2 registers
logic [15:0] r_y2;
logic [7:0] r_x2, r_c2;
// Stage 3 registers
logic [23:0] r_y3;
logic [7:0] r_c3;



always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 0; r_y2 <= 0; r_y3 <= 0;
        r_x1 <= 0; r_b1 <= 0; r_c1 <= 0;
        r_x2 <= 0; r_c2 <= 0;
        r_c3 <= 0; y <= 0;


    end else begin
        // Input registers
        r_a <= a; r_b <= b;
        r_c <= c; r_x <= x;

        // Stage 1
        r_y1 <= (r_a * r_x);
        r_x1 <= r_x; r_b1 <= r_b; r_c1 <= r_c;

        // Stage 2
        r_y2 <= r_y1 + r_b1;
        r_x2 <= r_x1; r_c2 <= r_c1;

        // Stage 3
        r_y3 <= r_y2 * r_x2;
        r_c3 <= r_c2;

        // Stage 4
        y <= r_y3 + r_c3;

    end
end



endmodule
```

Example: Latency Insensitive Polynomial Circuit

```
module poly_li (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y,
      input i_valid,
      input i_ready,
      output logic o_valid,
      output o_ready
);

// Input registers
logic [7:0] r_a, r_b, r_c, r_x;
// Stage 1 registers
logic [15:0] r_y1;
logic [7:0] r_x1, r_b1, r_c1;
// Stage 2 registers
logic [15:0] r_y2;
logic [7:0] r_x2, r_c2;
// Stage 3 registers
logic [23:0] r_y3;
logic [7:0] r_c3;
// Valid signal registers
logic r_valid, r_valid1, r_valid2, r_valid3;

always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 0; r_y2 <= 0; r_y3 <= 0;
        r_x1 <= 0; r_b1 <= 0; r_c1 <= 0;
        r_x2 <= 0; r_c2 <= 0;
        r_c3 <= 0; y <= 0;


    end else begin
        // Input registers
        r_a <= a; r_b <= b;
        r_c <= c; r_x <= x;

        // Stage 1
        r_y1 <= (r_a * r_x);
        r_x1 <= r_x; r_b1 <= r_b; r_c1 <= r_c;

        // Stage 2
        r_y2 <= r_y1 + r_b1;
        r_x2 <= r_x1; r_c2 <= r_c1;

        // Stage 3
        r_y3 <= r_y2 * r_x2;
        r_c3 <= r_c2;

        // Stage 4
        y <= r_y3 + r_c3;

    end
end



endmodule
```

Example: Latency Insensitive Polynomial Circuit

```
module poly_li (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y,
      input i_valid,
      input i_ready,
      output logic o_valid,
      output o_ready
);

// Input registers
logic [7:0] r_a, r_b, r_c, r_x;
// Stage 1 registers
logic [15:0] r_y1;
logic [7:0] r_x1, r_b1, r_c1;
// Stage 2 registers
logic [15:0] r_y2;
logic [7:0] r_x2, r_c2;
// Stage 3 registers
logic [23:0] r_y3;
logic [7:0] r_c3;
// Valid signal registers
logic r_valid, r_valid1, r_valid2, r_valid3;

always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 0; r_y2 <= 0; r_y3 <= 0;
        r_x1 <= 0; r_b1 <= 0; r_c1 <= 0;
        r_x2 <= 0; r_c2 <= 0;
        r_c3 <= 0; y <= 0;
        r_valid <= 0; r_valid1 <= 0;
        r_valid2 <= 0; r_valid3 <= 0;
    end else begin
        // Input registers
        r_a <= a; r_b <= b;
        r_c <= c; r_x <= x;
        r_valid <= i_valid;
        // Stage 1
        r_y1 <= (r_a * r_x);
        r_x1 <= r_x; r_b1 <= r_b; r_c1 <= r_c;
        r_valid1 <= r_valid;
        // Stage 2
        r_y2 <= r_y1 + r_b1;
        r_x2 <= r_x1; r_c2 <= r_c1;
        r_valid2 <= r_valid1;
        // Stage 3
        r_y3 <= r_y2 * r_x2;
        r_c3 <= r_c2;
        r_valid3 <= r_valid2;
        // Stage 4
        y <= r_y3 + r_c3;
        o_valid <= r_valid3;
    end
end



endmodule
```

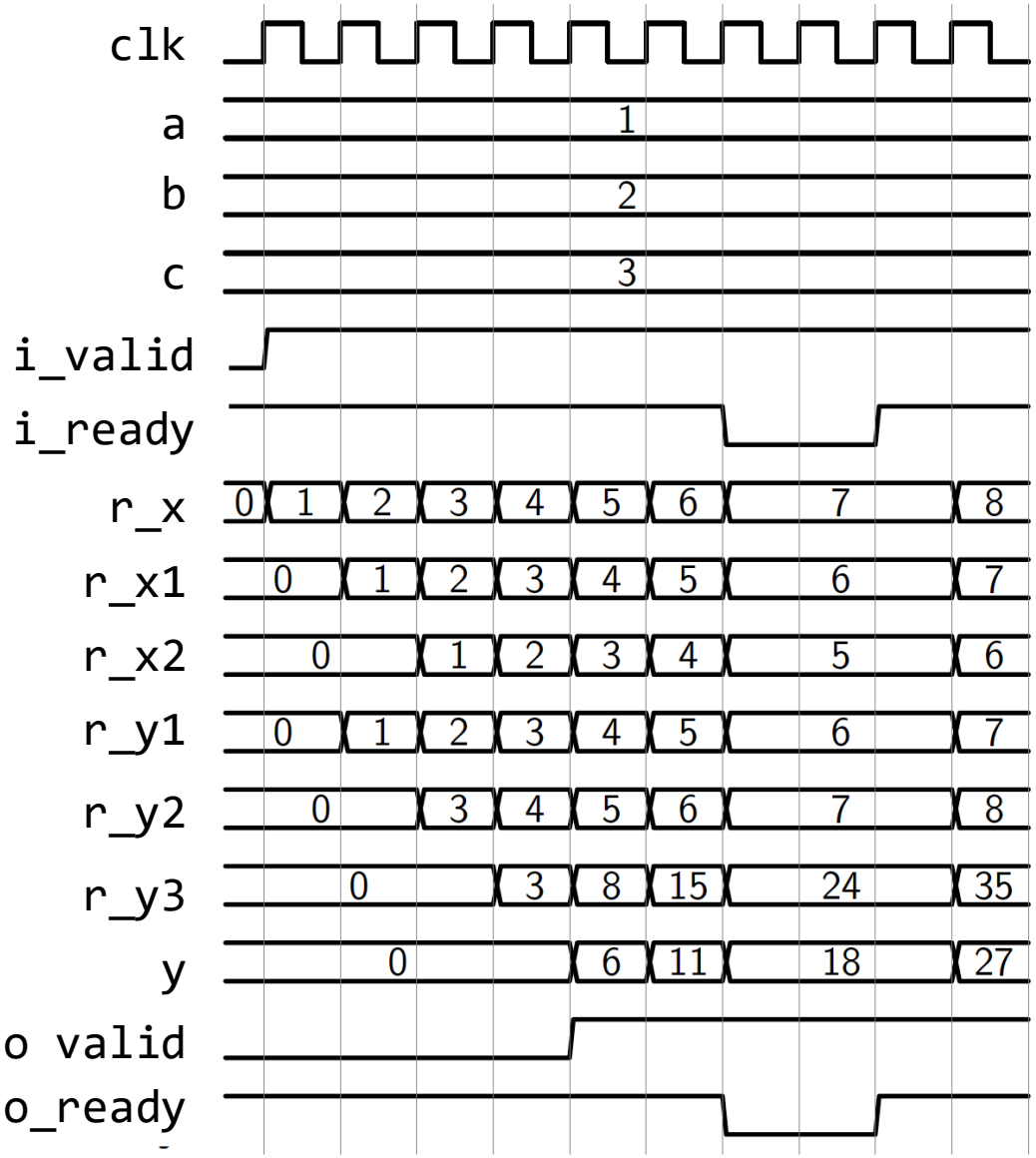
Example: Latency Insensitive Polynomial Circuit

```
module poly_li (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y,
      input i_valid,
      input i_ready,
      output logic o_valid,
      output o_ready
);

// Input registers
logic [7:0] r_a, r_b, r_c, r_x;
// Stage 1 registers
logic [15:0] r_y1;
logic [7:0] r_x1, r_b1, r_c1;
// Stage 2 registers
logic [15:0] r_y2;
logic [7:0] r_x2, r_c2;
// Stage 3 registers
logic [23:0] r_y3;
logic [7:0] r_c3;
// Valid signal registers
logic r_valid, r_valid1, r_valid2, r_valid3;

always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 0; r_y2 <= 0; r_y3 <= 0;
        r_x1 <= 0; r_b1 <= 0; r_c1 <= 0;
        r_x2 <= 0; r_c2 <= 0;
        r_c3 <= 0; y <= 0;
        r_valid <= 0; r_valid1 <= 0;
        r_valid2 <= 0; r_valid3 <= 0;
    end else begin
        // Input registers
        r_a <= a; r_b <= b;
        r_c <= c; r_x <= x;
        r_valid <= i_valid;
        // Stage 1
        r_y1 <= (r_a * r_x);
        r_x1 <= r_x; r_b1 <= r_b; r_c1 <= r_c;
        r_valid1 <= r_valid;
        // Stage 2
        r_y2 <= r_y1 + r_b1;
        r_x2 <= r_x1; r_c2 <= r_c1;
        r_valid2 <= r_valid1;
        // Stage 3
        r_y3 <= r_y2 * r_x2;
        r_c3 <= r_c2;
        r_valid3 <= r_valid2;
        // Stage 4
        y <= r_y3 + r_c3;
        o_valid <= r_valid3;
    end
end

assign o_ready = i_ready;

endmodule
```

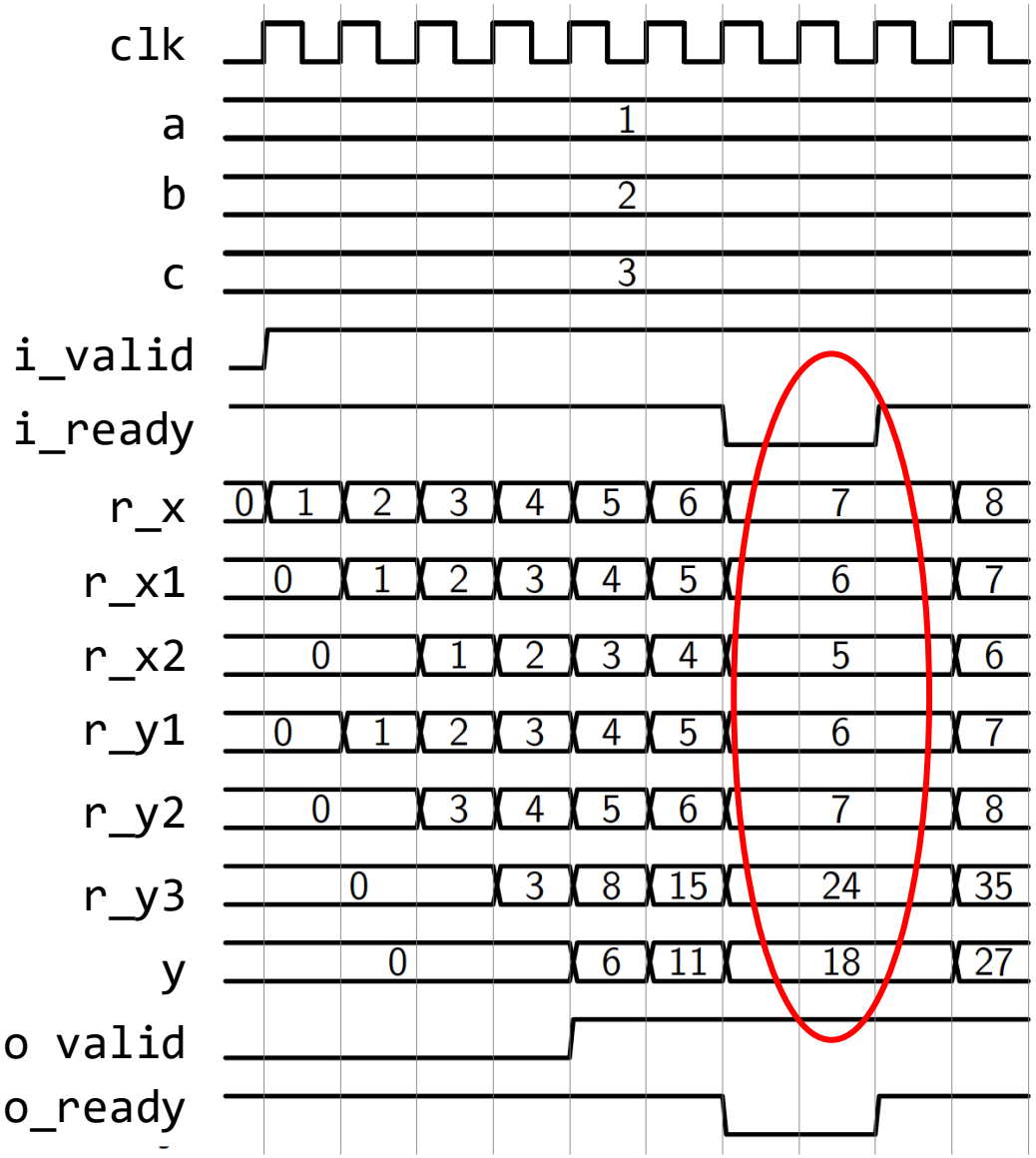
Example: Latency Insensitive Polynomial Circuit

```
module poly_li (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y,
      input i_valid,
      input i_ready,
      output logic o_valid,
      output o_ready
);

// Input registers
logic [7:0] r_a, r_b, r_c, r_x;
// Stage 1 registers
logic [15:0] r_y1;
logic [7:0] r_x1, r_b1, r_c1;
// Stage 2 registers
logic [15:0] r_y2;
logic [7:0] r_x2, r_c2;
// Stage 3 registers
logic [23:0] r_y3;
logic [7:0] r_c3;
// Valid signal registers
logic r_valid, r_valid1, r_valid2, r_valid3;

always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 0; r_y2 <= 0; r_y3 <= 0;
        r_x1 <= 0; r_b1 <= 0; r_c1 <= 0;
        r_x2 <= 0; r_c2 <= 0;
        r_c3 <= 0; y <= 0;
        r_valid <= 0; r_valid1 <= 0;
        r_valid2 <= 0; r_valid3 <= 0;
    end else if (i_ready) begin
        // Input registers
        r_a <= a; r_b <= b;
        r_c <= c; r_x <= x;
        r_valid <= i_valid;
        // Stage 1
        r_y1 <= (r_a * r_x);
        r_x1 <= r_x; r_b1 <= r_b; r_c1 <= r_c;
        r_valid1 <= r_valid;
        // Stage 2
        r_y2 <= r_y1 + r_b1;
        r_x2 <= r_x1; r_c2 <= r_c1;
        r_valid2 <= r_valid1;
        // Stage 3
        r_y3 <= r_y2 * r_x2;
        r_c3 <= r_c2;
        r_valid3 <= r_valid2;
        // Stage 4
        y <= r_y3 + r_c3;
        o_valid <= r_valid3;
    end
end

assign o_ready = i_ready;

endmodule
```

Example: Latency Insensitive Polynomial Circuit

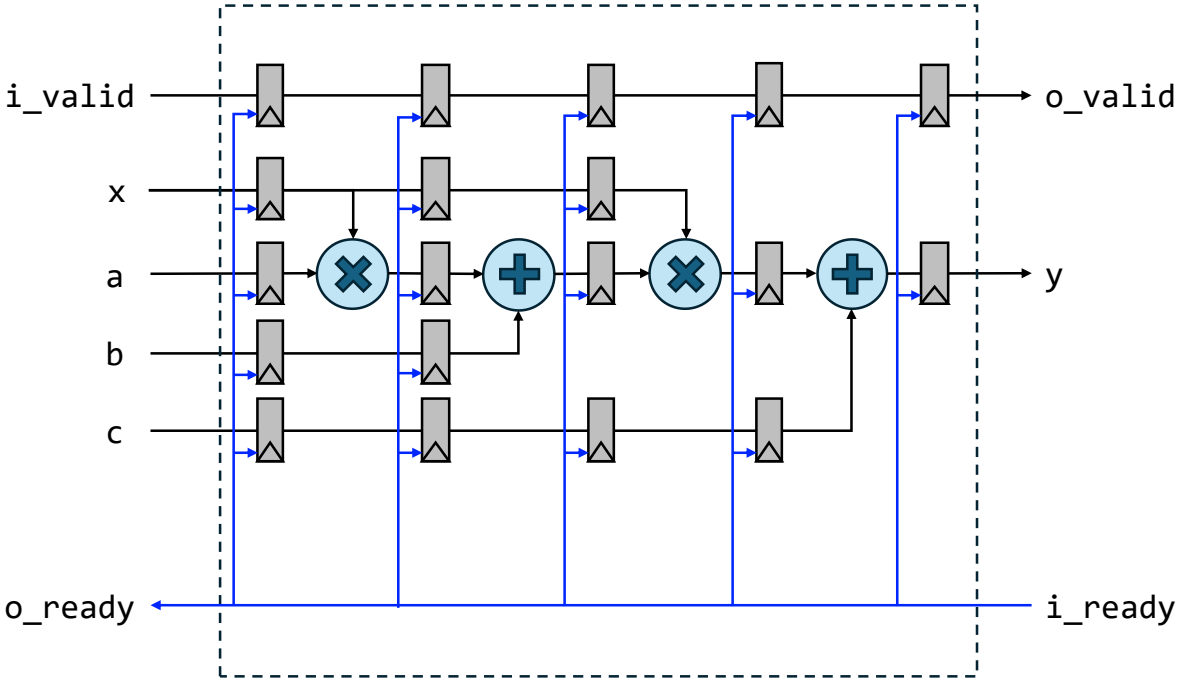


Example: Latency Insensitive Polynomial Circuit



All pipeline registers are frozen!

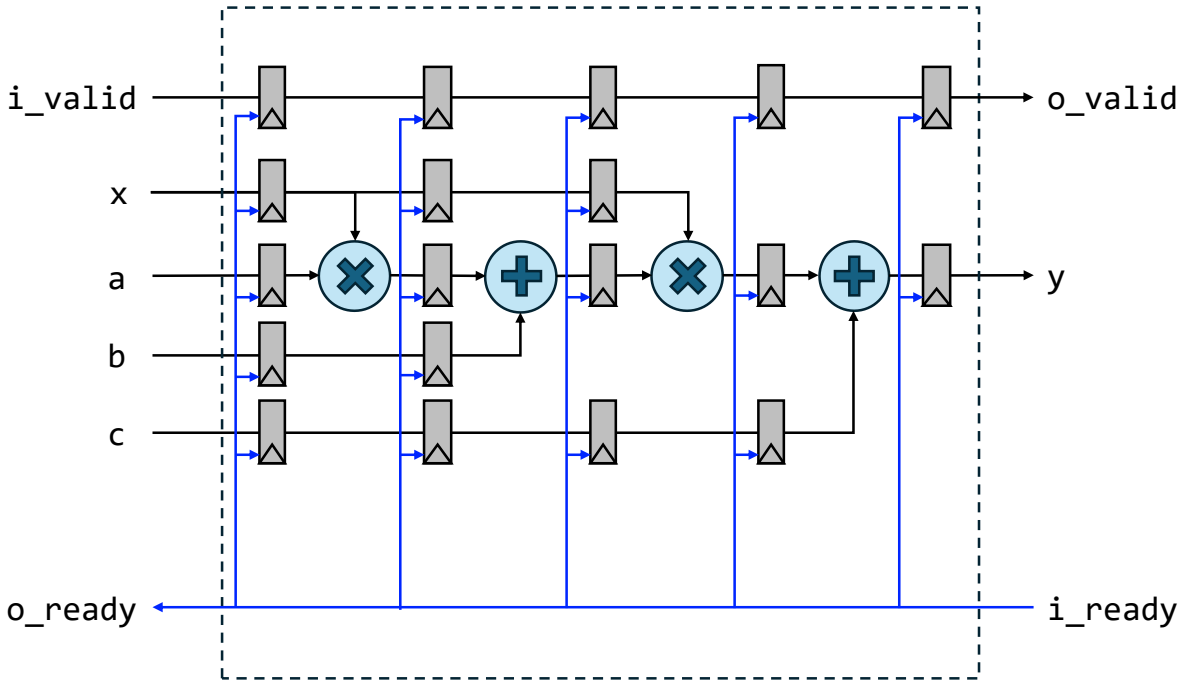
“Global Stalling” is Simple, but …



“Global Stalling” is Simple, but …

There are two major issues with this approach …

1. Ready signal has large fanout – it must connect to all clock enable inputs of all registers
2. Ready signal must be purely combinational

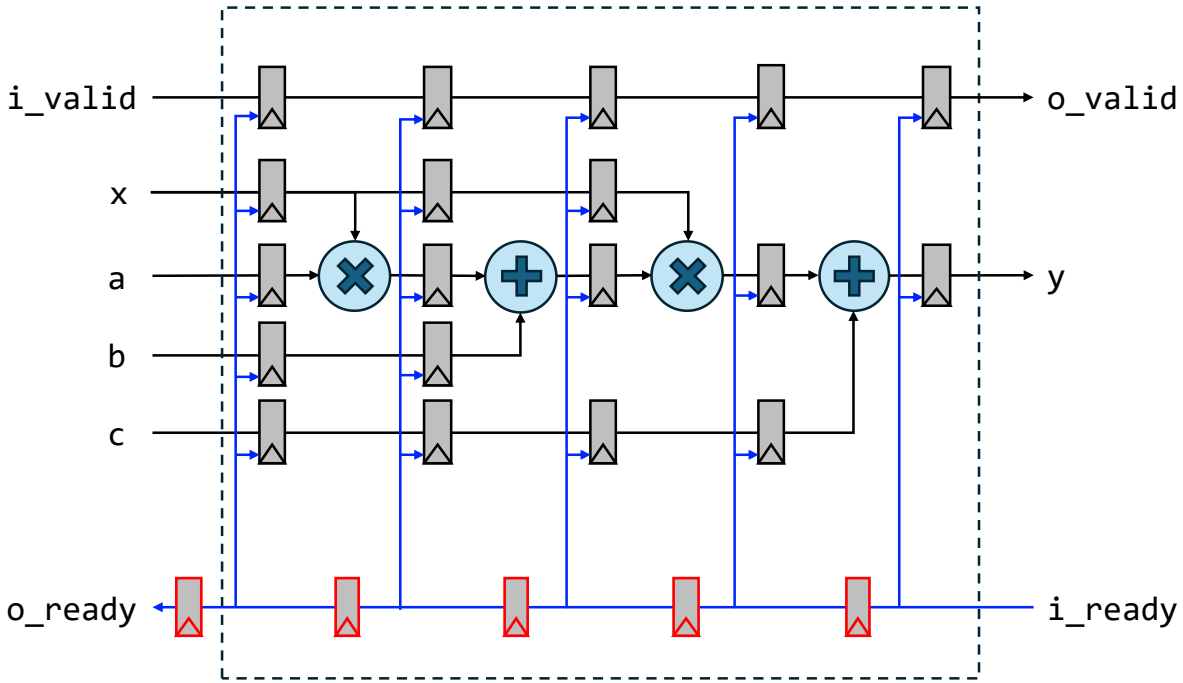


“Global Stalling” is Simple, but …

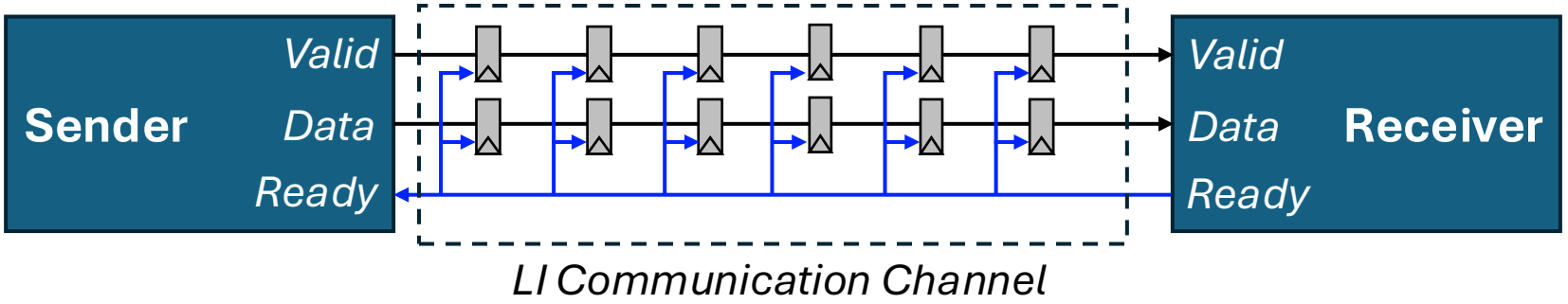
There are two major issues with this approach …

1. Ready signal has large fanout – it must connect to all clock enable inputs of all registers
2. Ready signal must be purely combinational

Why can’t we just pipeline the ready signal going backwards?



Well … That didn't solve our problem!

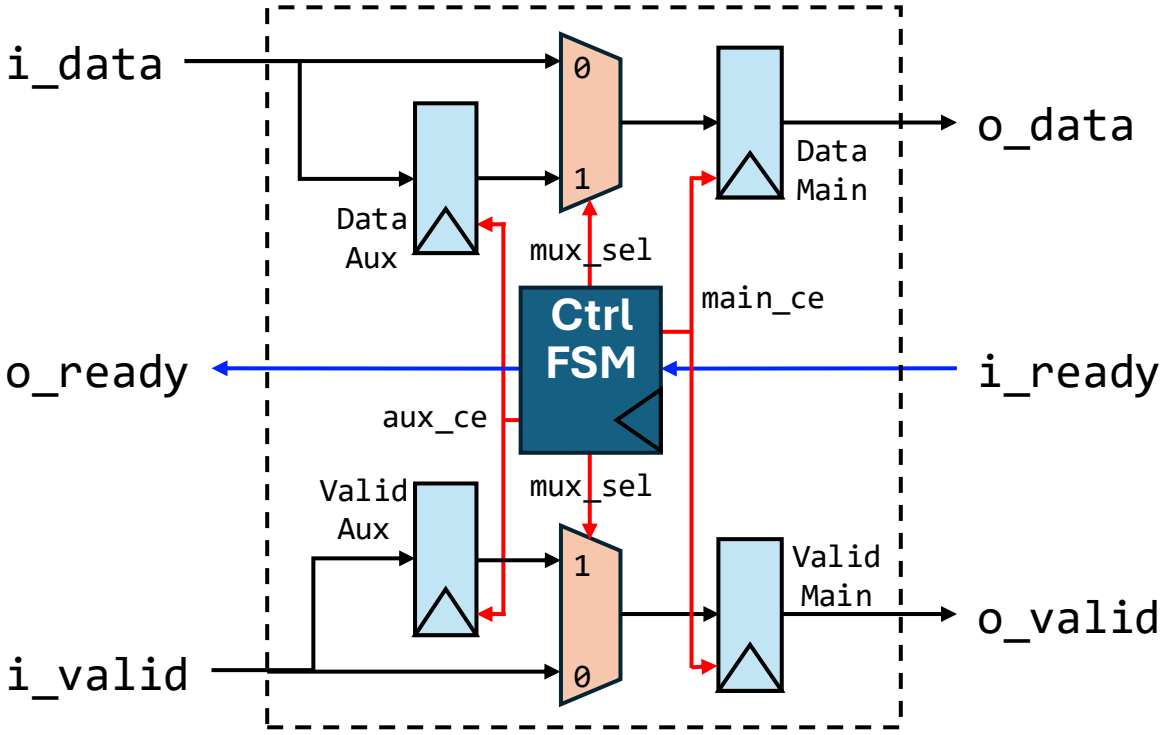


- We can pipeline the data & valid signals as much as we want without needing to change the internal logic of the receiver

- However, we cannot pipeline the ready signal to avoid losing data in case of a backpressure

Simple pipeline registers are not enough!!

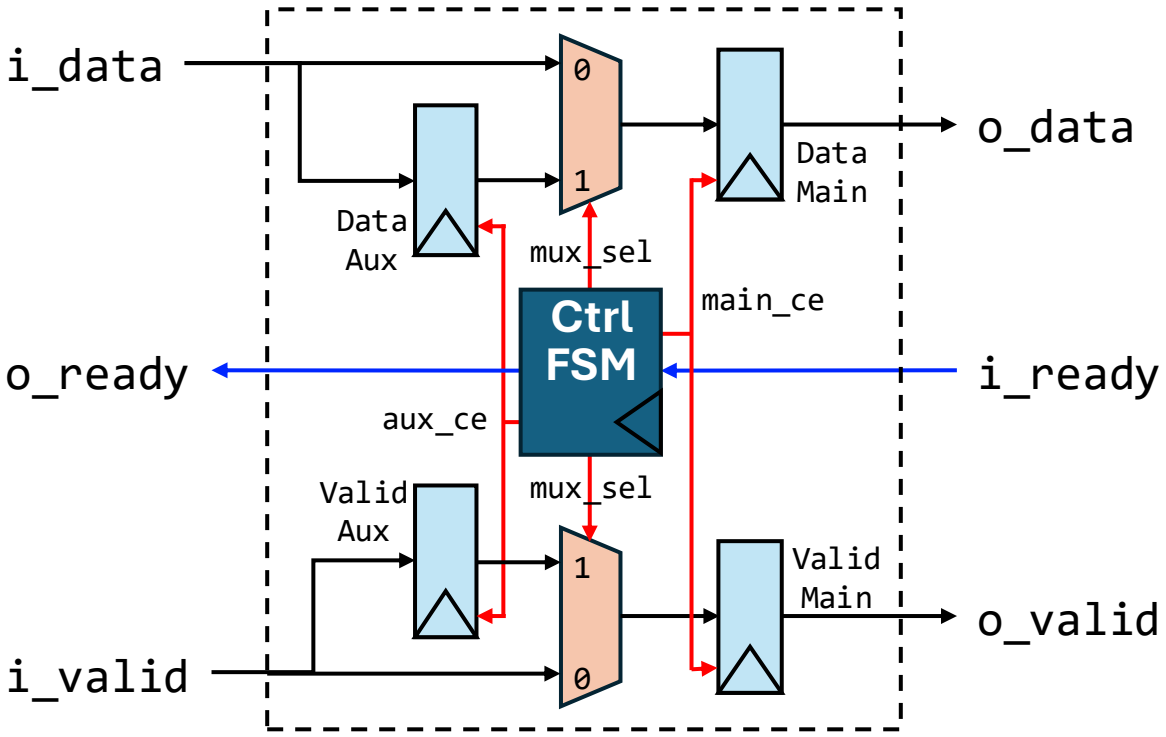
Relay Station



Relay Station

Consists of ...

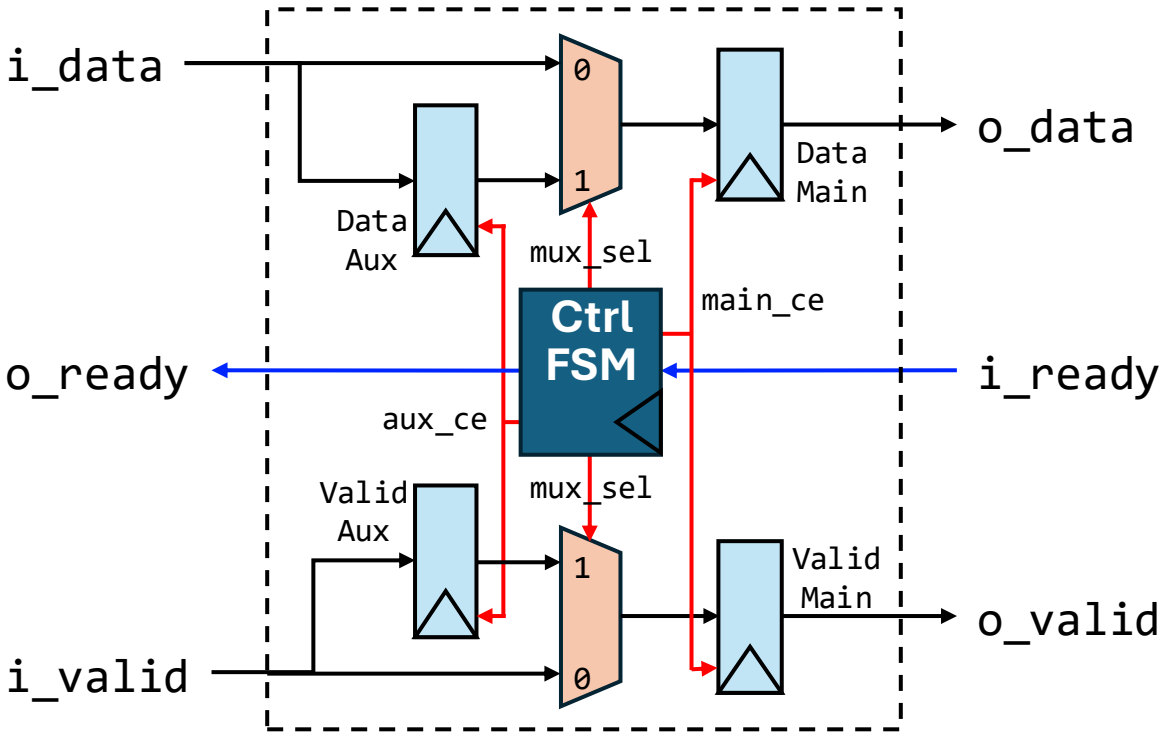
- Main and auxiliary registers for data and valid signals



Relay Station

Consists of ...

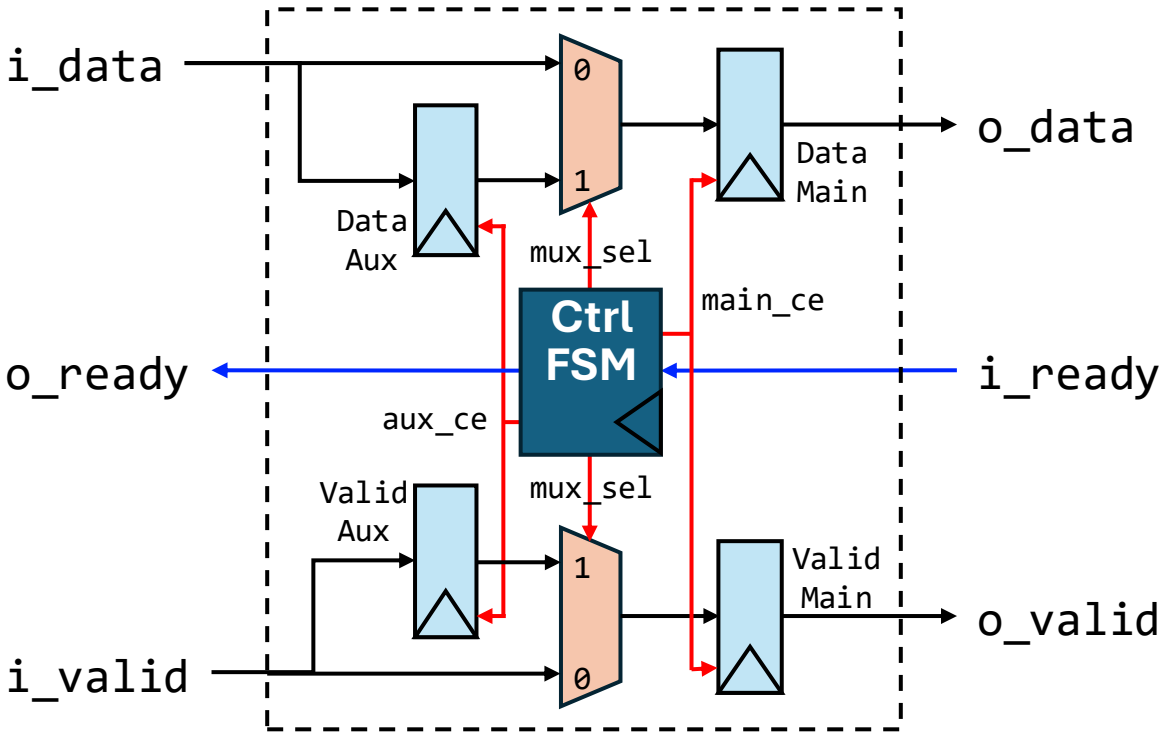
- Main and auxiliary registers for data and valid signals
- MUXes to allow writing to main registers from inputs or aux registers



Relay Station

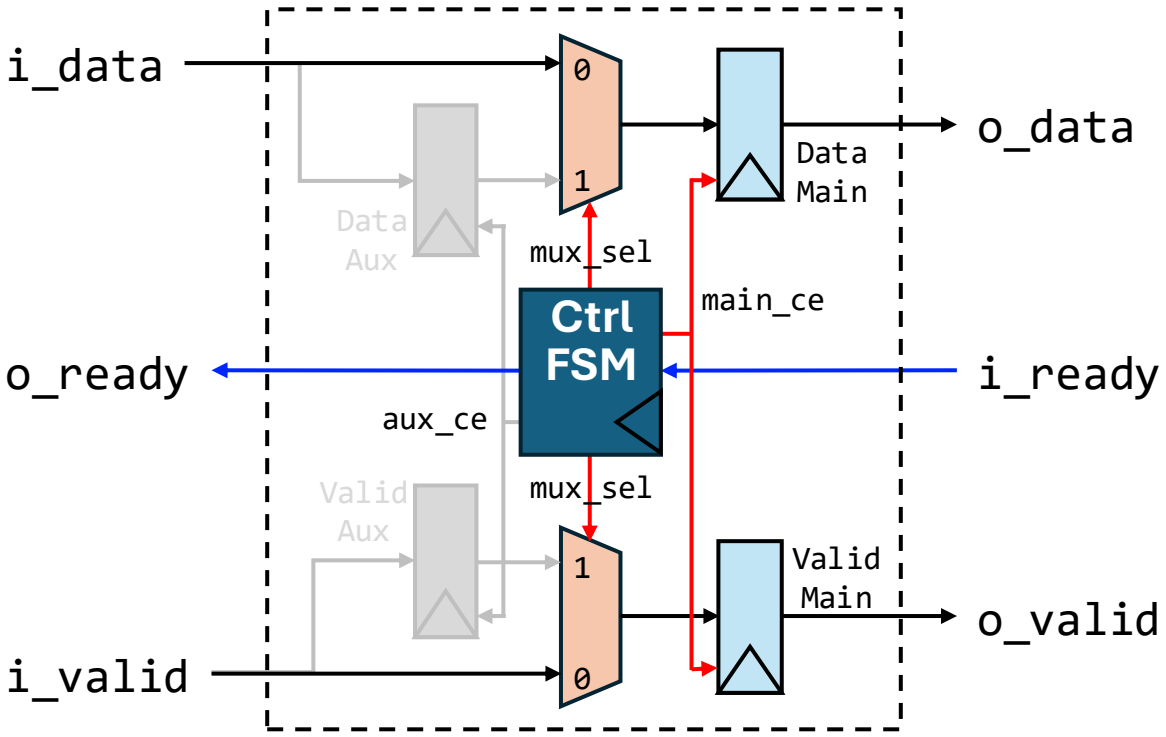
Consists of …

- Main and auxiliary registers for data and valid signals
- MUXes to allow writing to main registers from inputs or aux registers
- Control FSM to issue register clock enables and MUX select lines
  - Remember – the FSM has registered outputs



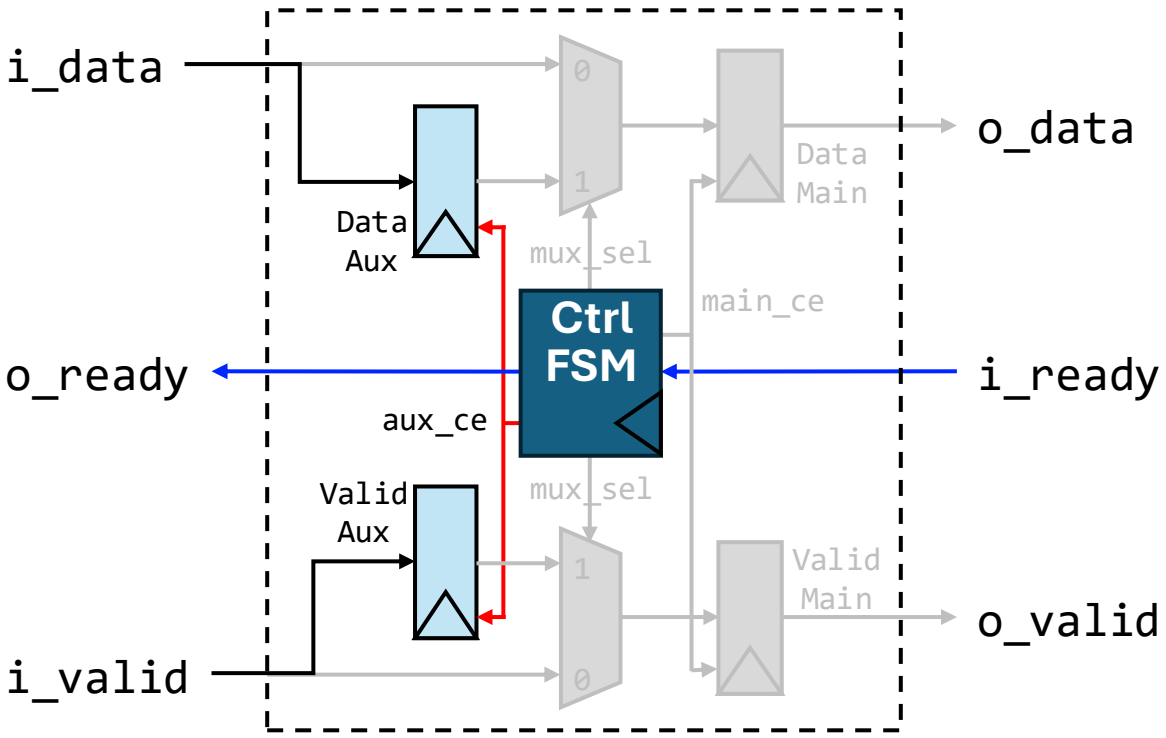
Relay Station

- If receiver is ready, the main registers are used to propagate the data and valid signals



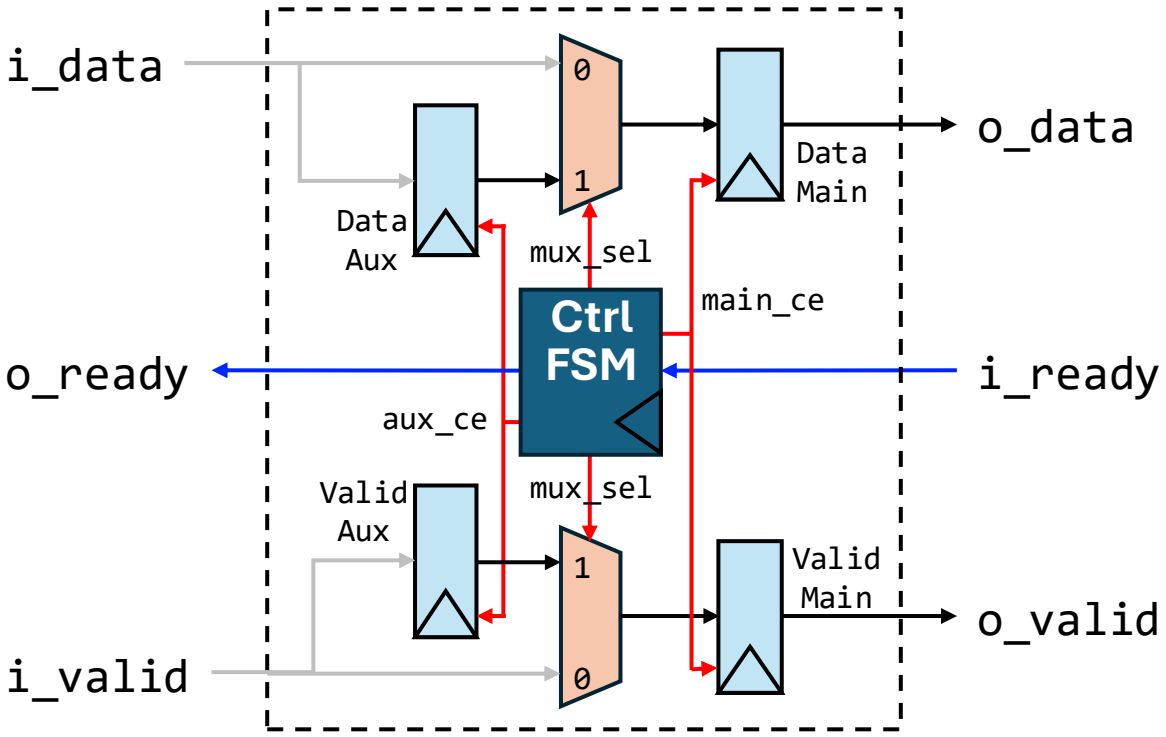
Relay Station

- If receiver is not ready, the main registers are stalled
- Since i_ready takes 1 cycle to propagate to o_ready, sender has already sent a new data/valid, hold them in aux registers



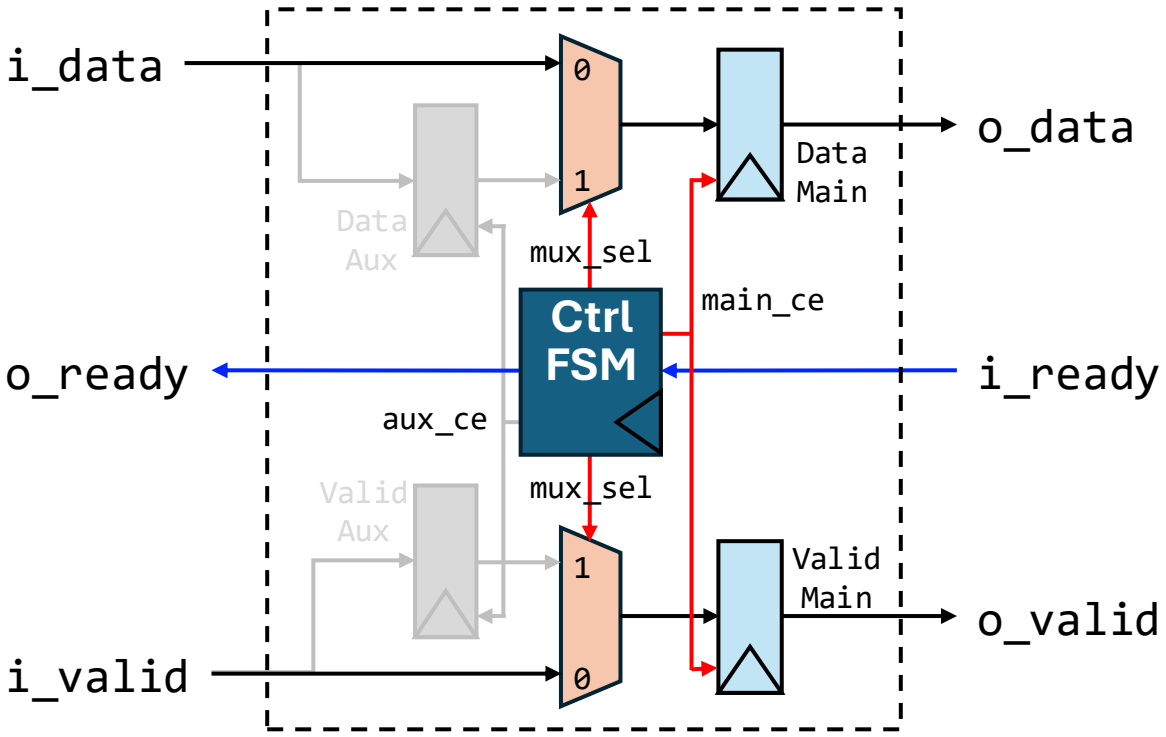
Relay Station

- When receiver is ready again, first read out data/valid from main registers and push data/valid from aux registers to main

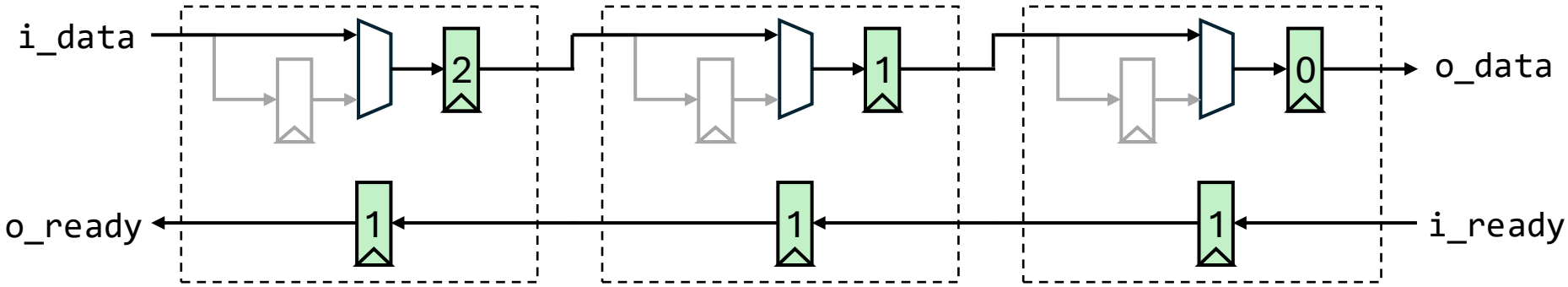


Relay Station

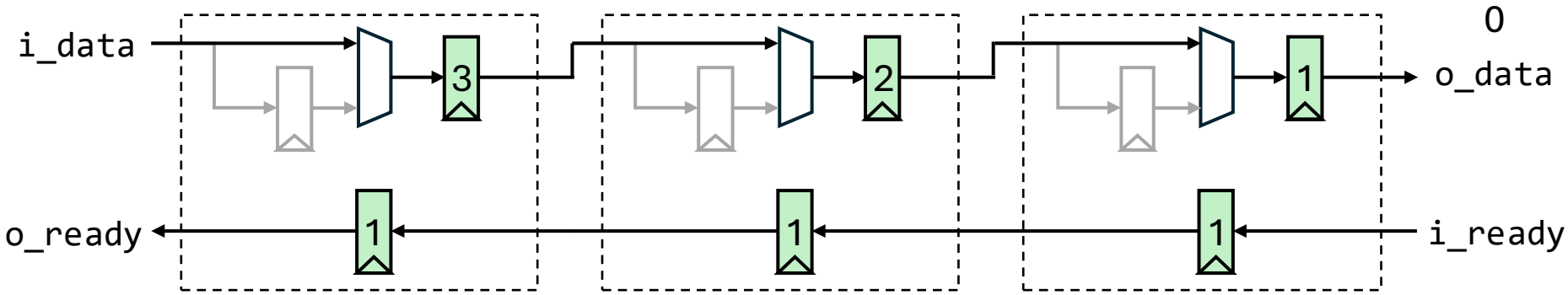
- When receiver is ready again, first read out data/valid from main registers and push data/valid from aux registers to main
- Then back to normal operation …



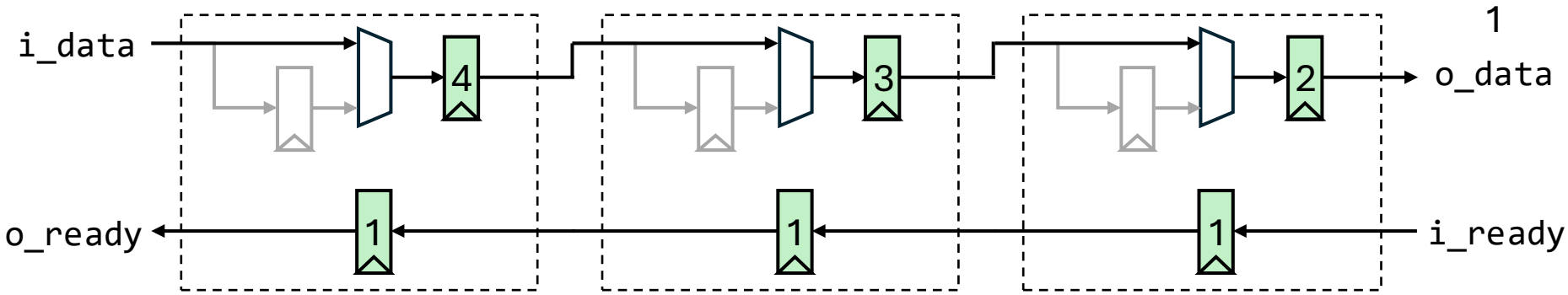
Example Operation of Relay Stations



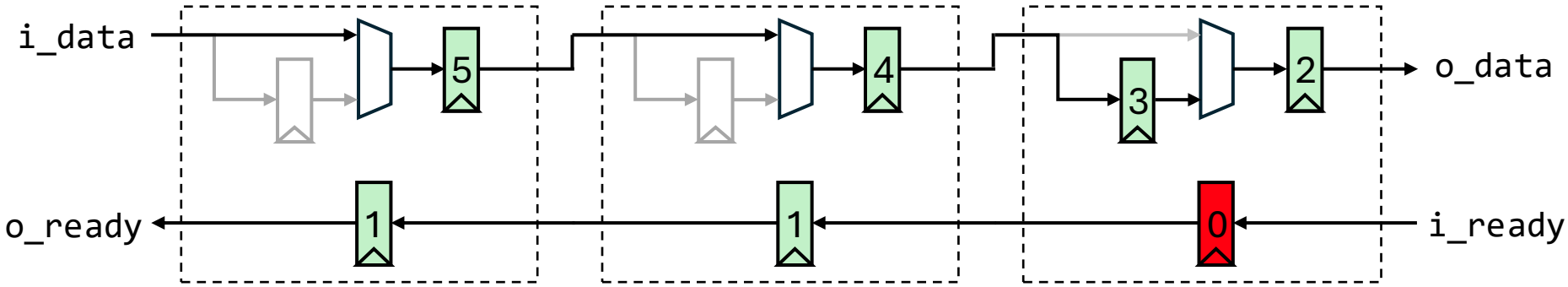
Example Operation of Relay Stations



Example Operation of Relay Stations

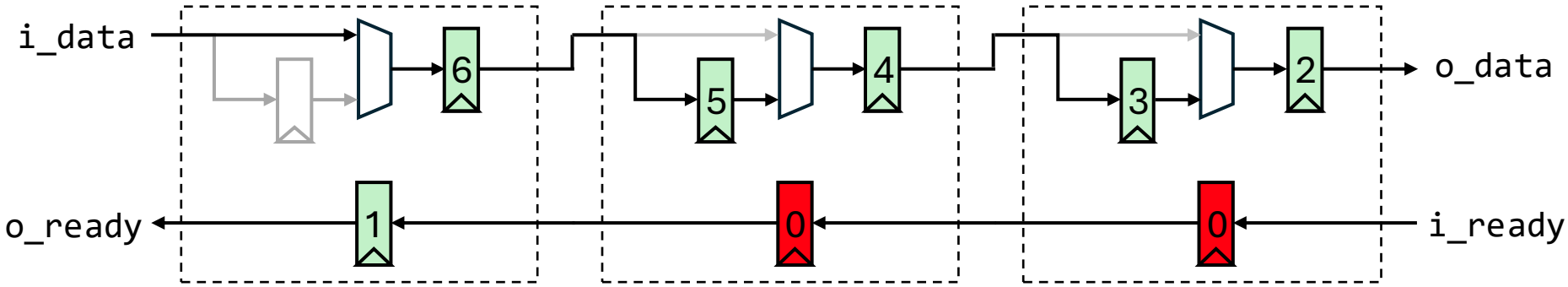


Example Operation of Relay Stations



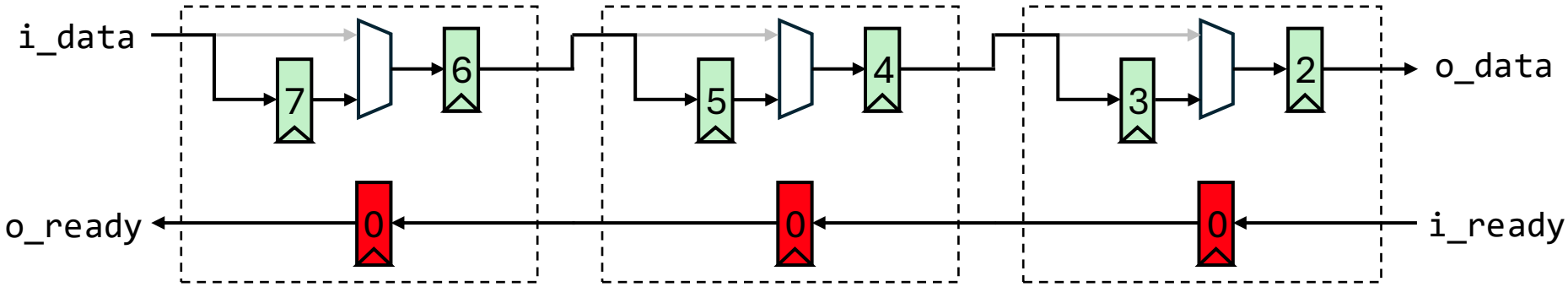
Main register is stalled & aux register saves it from being overwritten!

Example Operation of Relay Stations



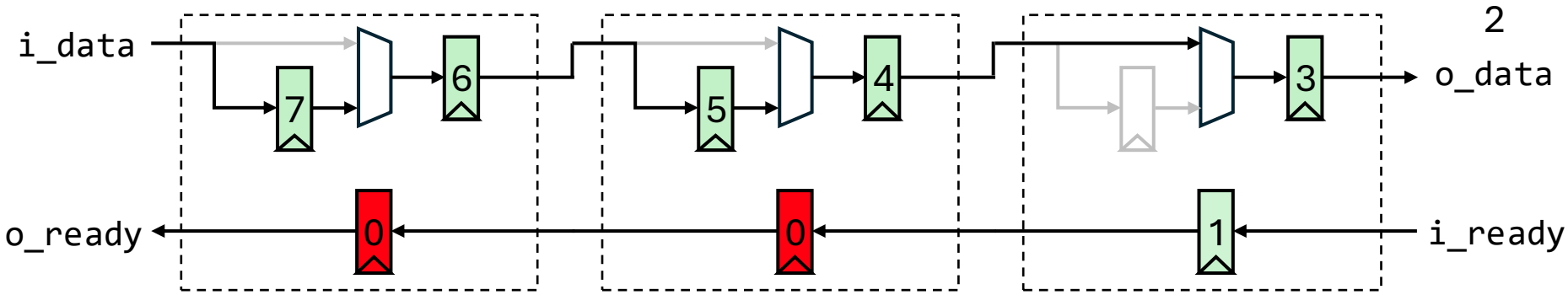
Main register is stalled &
aux register saves it
from being overwritten!

Example Operation of Relay Stations



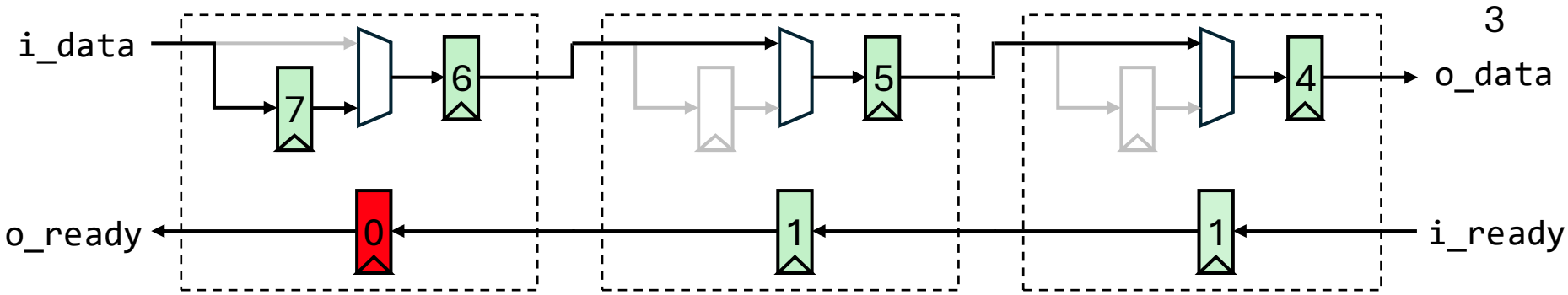
Main register is stalled & aux register saves it from being overwritten!

Example Operation of Relay Stations



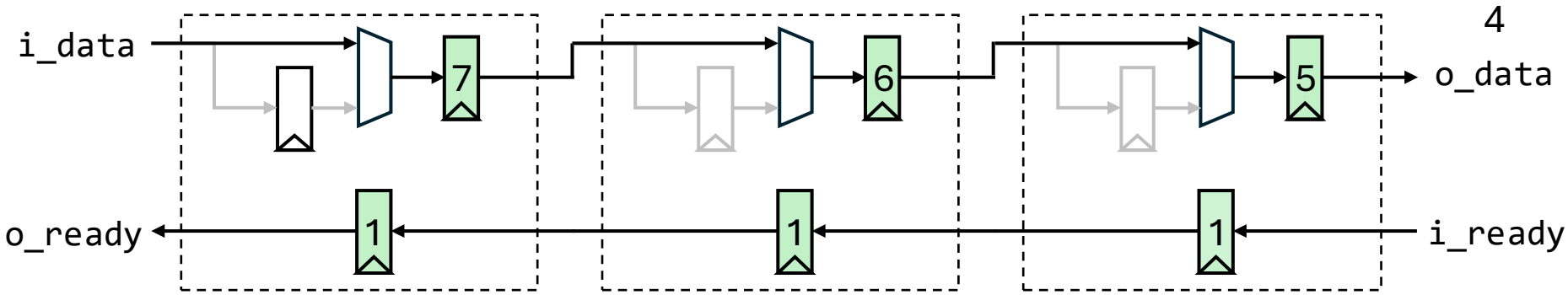
Read out data from main register & push data from aux register to main register!

Example Operation of Relay Stations



Read out data from main register & push data from aux register to main register!

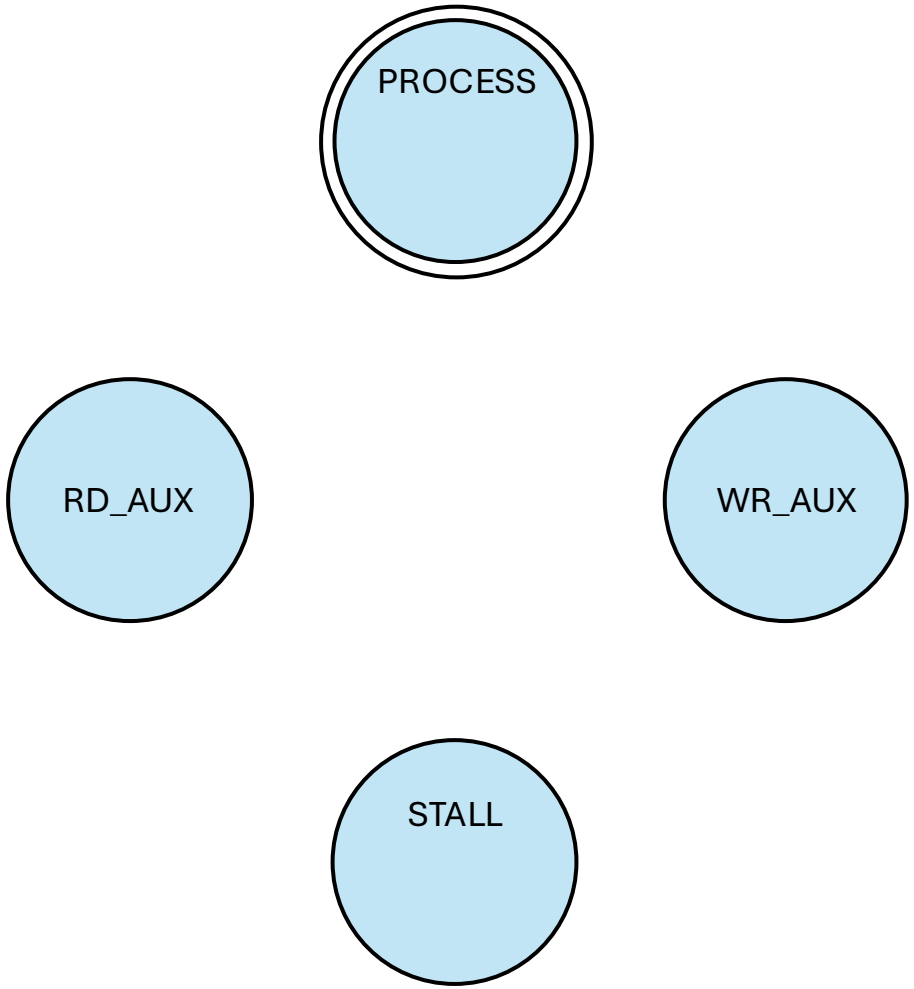
Example Operation of Relay Stations



Read out data from main register & push data from aux register to main register!

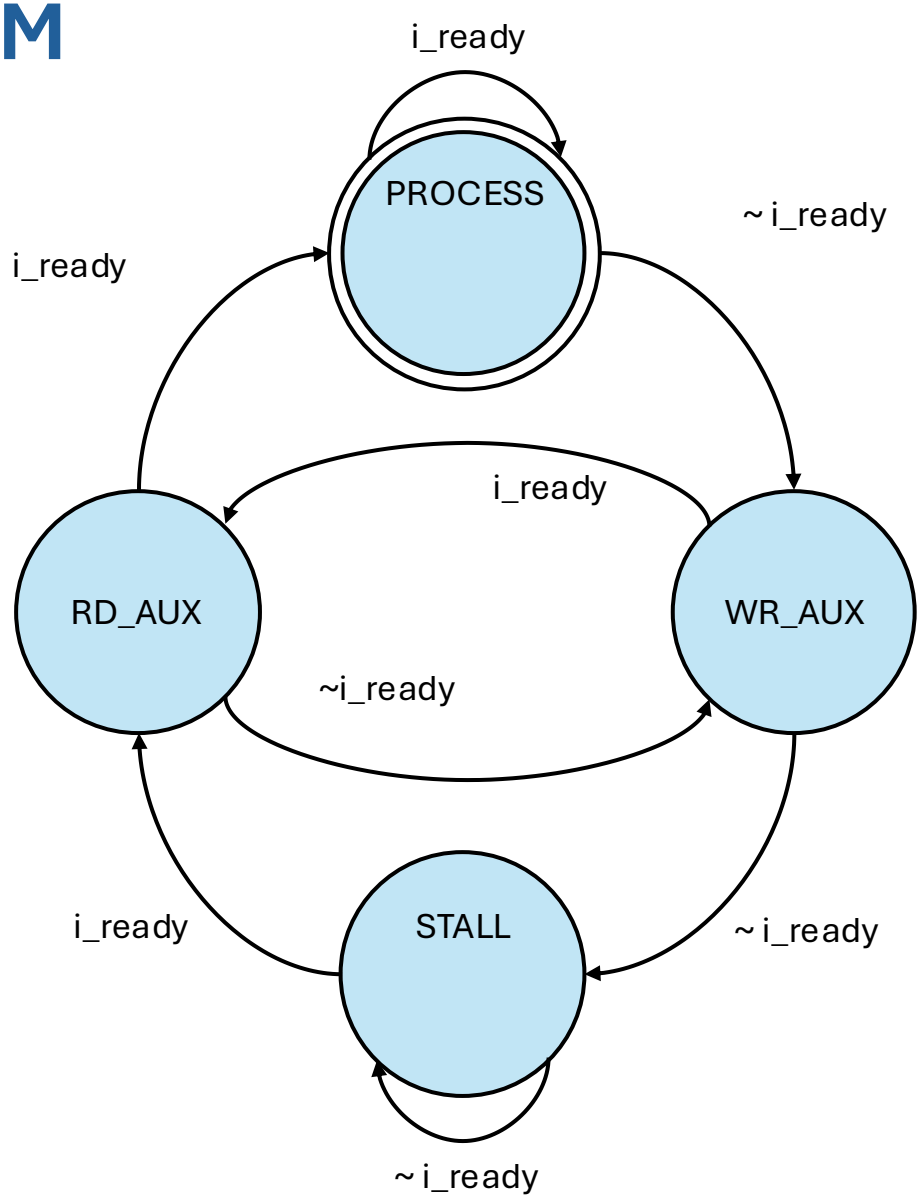
Relay Station Control FSM

- FSM has 4 states …
  - **PROCESS**: normal operation
  - **WR_AUX**: write to aux reg when i_ready goes from 1→0
  - **RD_AUX**: read from aux reg when i_ready goes from 0→1
  - **STALL**: relay station is stalled



Relay Station Control FSM

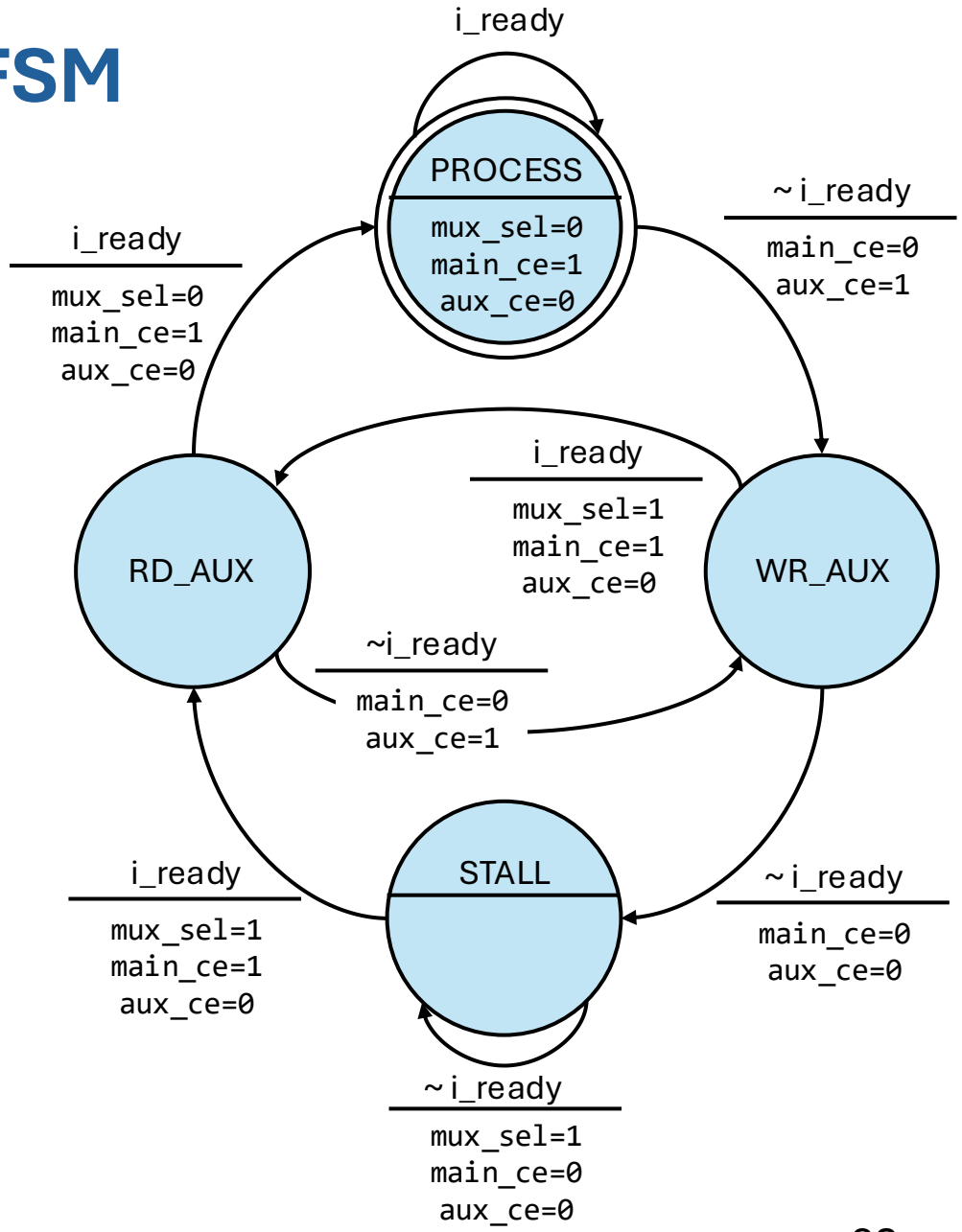
- FSM has 4 states …
  - **PROCESS**: normal operation
  - **WR_AUX**: write to aux reg when i_ready goes from 1→0
  - **RD_AUX**: read from aux reg when i_ready goes from 0→1
  - **STALL**: relay station is stalled
- Transition between states based on value of i_ready



Relay Station Control FSM

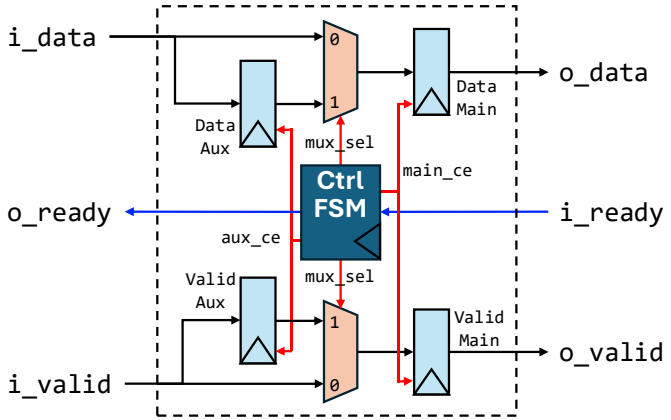
Set MUX select lines and register clock enables:

- If writing to aux regs or stalling, freeze main regs (`main_ce = 0`), otherwise clock enable them (`main_ce = 1`)
- If writing to aux regs, clock enable them (`aux_ce = 1`), otherwise freeze them (`aux_ce = 0`)
- MUX select chooses aux register as a source if stalling or coming out of a stall (`mux_sel = 1`), otherwise chooses input data as a source (`mux_sel = 0`)



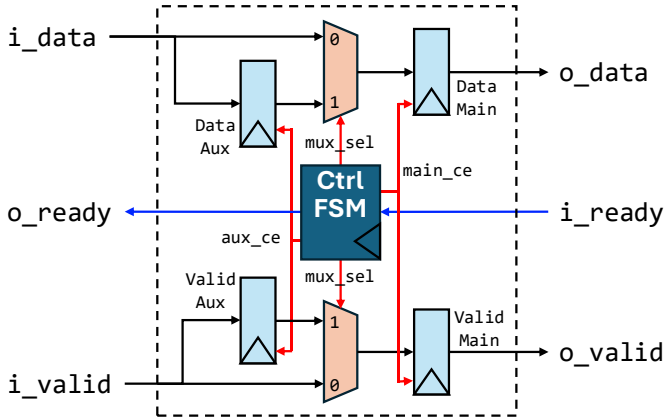
Relay Station SystemVerilog Code

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



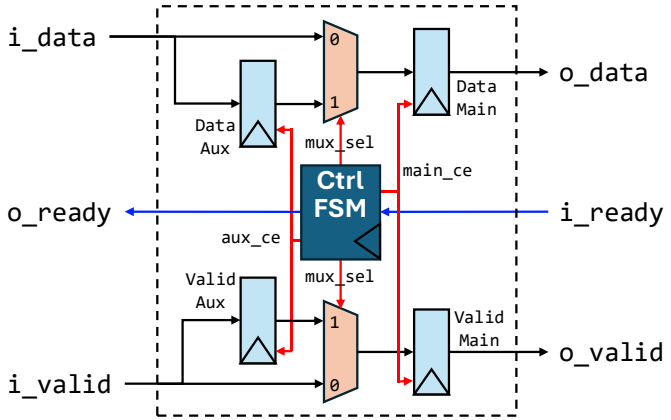
Relay Station SystemVerilog Code

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


logic [DATAW-1:0] data_aux;
logic valid_aux;
enum {PROCESS, WR_AUX, RD_AUX, STALL} state, next_state;
logic main_ce_w, aux_ce_w, mux_sel_w;
```



Relay Station SystemVerilog Code

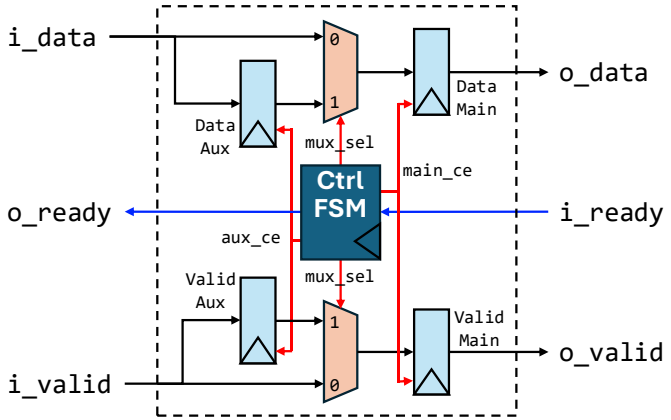
```
module relay_station # (
    parameter DATAW = 32
)(
    input clk,
    input rst,
    input [DATAW-1:0] i_data,
    input i_valid,
    input i_ready,
    output logic [DATAW-1:0] o_data,
    output logic o_valid,
    output logic o_ready
);


logic [DATAW-1:0] data_aux;
logic valid_aux;
enum {PROCESS, WR_AUX, RD_AUX, STALL} state, next_state;
logic main_ce_w, aux_ce_w, mux_sel_w;


always_ff @ (posedge clk) begin
  if (rst) begin
    o_data <= 'd0; data_aux <= ‘d0;
    o_valid <= 1'b0; valid_aux <= 1'b0;
  end else begin
    if (main_ce_w) begin
      o_data <= (mux_sel_w)? data_aux : i_data;
      o_valid <= (mux_sel_w)? valid_aux : i_valid;
    end
    if (aux_ce_w) begin
      data_aux <= i_data;
      valid_aux <= i_valid;
    end
  end
end
```



Relay Station SystemVerilog Code

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


logic [DATAW-1:0] data_aux;
logic valid_aux;
enum {PROCESS, WR_AUX, RD_AUX, STALL} state, next_state;
logic main_ce_w, aux_ce_w, mux_sel_w;


always_ff @ (posedge clk) begin
  if (rst) begin
    o_data <= 'd0; data_aux <= ‘d0;
    o_valid <= 1'b0; valid_aux <= 1'b0;
  end else begin
    if (main_ce_w) begin
      o_data <= (mux_sel_w)? data_aux : i_data;
      o_valid <= (mux_sel_w)? valid_aux : i_valid;
    end
    if (aux_ce_w) begin
      data_aux <= i_data;
      valid_aux <= i_valid;
    end
  end
end
```

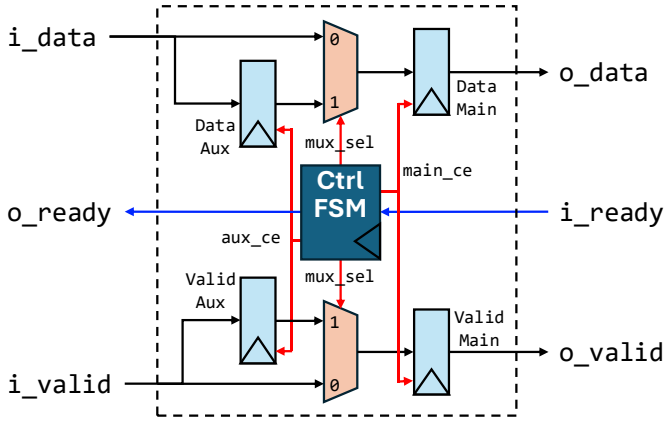
```
always_ff @ (posedge clk) begin
    if (rst) begin
        state <= PROCESS;
        o_ready <= 1’b0;
    end else begin
        state <= next_state;
        o_ready <= i_ready;
    end
end
```



Relay Station SystemVerilog Code

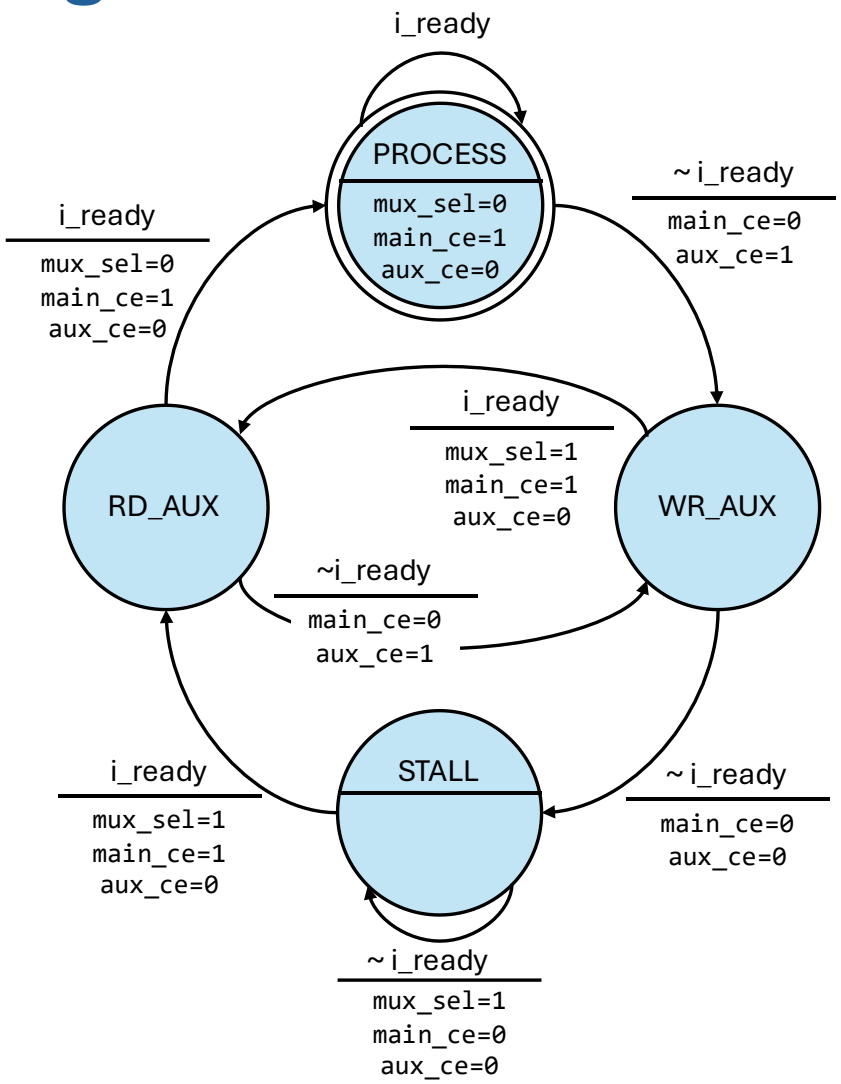
```
module relay_station # (
    parameter DATAW = 32
)(
    input clk,
    input rst,
    input [DATAW-1:0] i_data,
    input i_valid,
    input i_ready,
    output logic [DATAW-1:0] o_data,
    output logic o_valid,
    output logic o_ready
);


logic [DATAW-1:0] data_aux;
logic valid_aux;
enum {PROCESS, WR_AUX, RD_AUX, STALL} state, next_state;
logic main_ce_w, aux_ce_w, mux_sel_w;


always_ff @ (posedge clk) begin
  if (rst) begin
    o_data <= 'd0; data_aux <= ‘d0;
    o_valid <= 1'b0; valid_aux <= 1'b0;
  end else begin
    if (main_ce_w) begin
      o_data <= (mux_sel_w)? data_aux : i_data;
      o_valid <= (mux_sel_w)? valid_aux : i_valid;
    end
    if (aux_ce_w) begin
      data_aux <= i_data;
      valid_aux <= i_valid;
    end
  end
end
```

```
always_ff @ (posedge clk) begin
    if (rst) begin
        state <= PROCESS;
        o_ready <= 1’b0;
    end else begin
        state <= next_state;
        o_ready <= i_ready;
    end
end

always_comb begin: state_decoder
  case (state)
    PROCESS: next_state = (!i_ready)? WR_AUX : PROCESS;
    WR_AUX: next_state = (!i_ready)? STALL : RD_AUX;
    RD_AUX: next_state = (!i_ready)? WR_AUX : PROCESS;
    STALL: next_state = (!i_ready)? STALL : RD_AUX;
    default: next_state = PROCESS;
  endcase
end
```

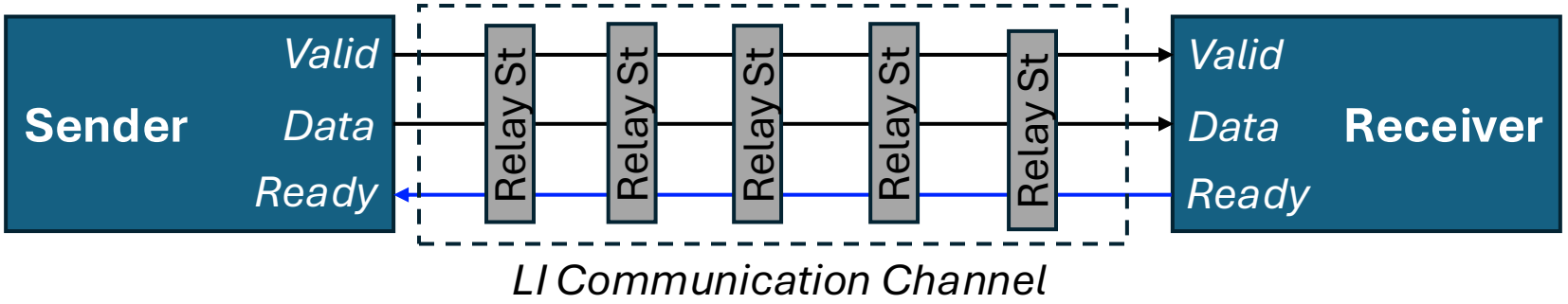


Relay Station SystemVerilog Code

```
always_comb begin: output_decoder
    case (state)
        PROCESS: begin
            mux_sel_w = 1'b0;
            main_ce_w = (!i_ready)? 1'b0 : 1'b1;
            aux_ce_w = (!i_ready)? 1'b1 : 1'b0;
        end
        WR_AUX: begin
            mux_sel_w = 1'b1;
            main_ce_w = (!i_ready)? 1'b0 : 1'b1;
            aux_ce_w = 1'b0;
        end
        RD_AUX: begin
            mux_sel_w = 1'b0;
            main_ce_w = (!i_ready)? 1'b0 : 1'b1;
            aux_ce_w = (!i_ready)? 1'b1 : 1'b0;
        end
        STALL: begin
            mux_sel_w = 1'b1;
            main_ce_w = (!i_ready)? 1'b0: 1'b1;
            aux_ce_w = 1'b0;
        end
        default: begin
            mux_sel_w = 1'b0;
            main_ce_w = (!i_ready)? 1'b0 : 1'b1;
            aux_ce_w = (!i_ready)? 1'b1 : 1'b0;
        end
    endcase
end

endmodule
```



Relay Stations Save the Day!



- We can add an arbitrary number of relay stations between sender and receiver
  - No combinational paths → easier to close timing!
  - No need to change logic in sender/receiver as we change number of relay stations on the communication channel

- Relay stations are sometimes referred to as “skid buffers” or “latency-insensitive register slice”

Advanced eXtensible Interface (AXI) Protocol

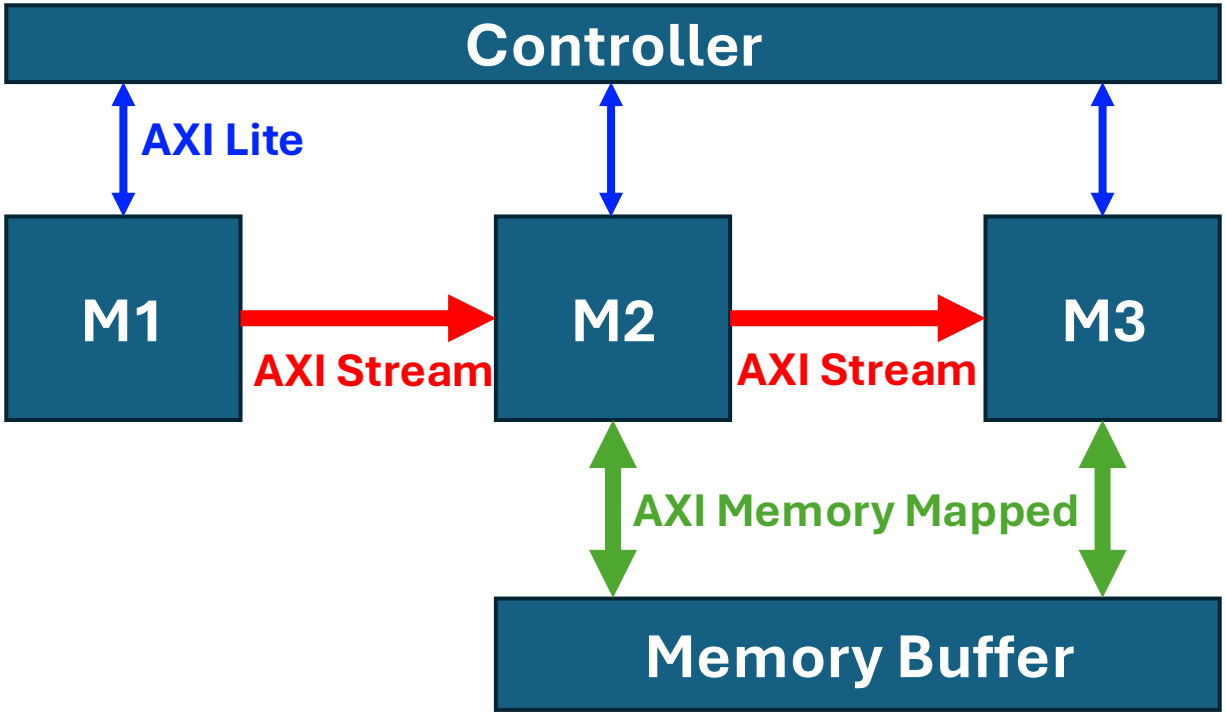
- AXI is a latency-insensitive interface protocol
  - Instead of the simple data/valid/ready interface we have been using so far, AXI defines different types of interfaces for different use cases
  - Introduced by ARM in 2003 as part of the Advanced Microcontroller Bus Architecture (AMBA) standard
  - The industry standard for latency-insensitive communication between different modules in a digital system
  - Full specification can be found at this link
- Designing an intellectual property (IP) block that speaks AXI enables others to easily integrate it in their systems
- For FPGA development, AMD Xilinx adopted AXI while Altera developed a similar protocol called Avalon – now gradually shifting to AXI for wider compatibility

Types of AXI Interfaces

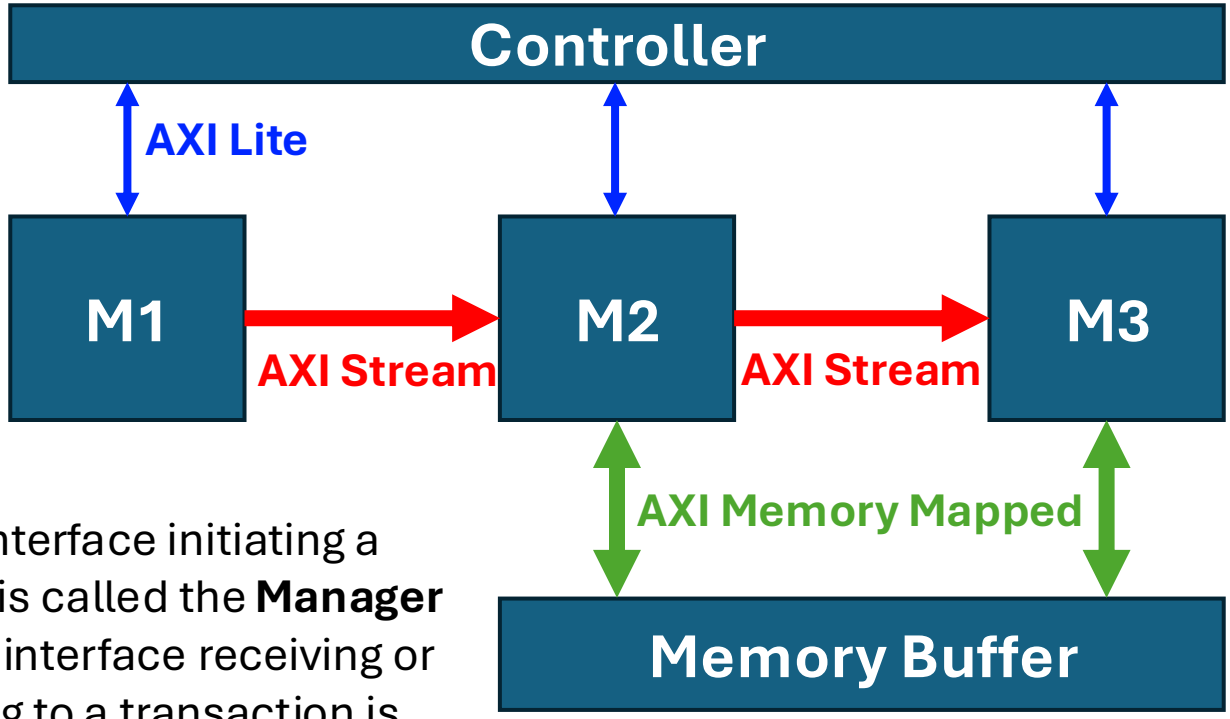
There are 3 types of AXI interfaces used for different purposes …

1. AXI Memory Mapped (typically referred to as just “AXI”)
   - Used to read/write data to/from a specific address
   - Has 5 channels: read address (AR), read data (R), write address (AW), write data (W), and write response (B)
2. AXI Lite
   - Simplified version of AXI Memory Mapped
   - Used for simple, low-throughput, memory-mapped communication such as reading or writing from Control and Status Registers (CSRs)
3. AXI Stream
   - Used for data flowing/streaming between two end points
   - No notion of addresses

Types of AXI Interfaces

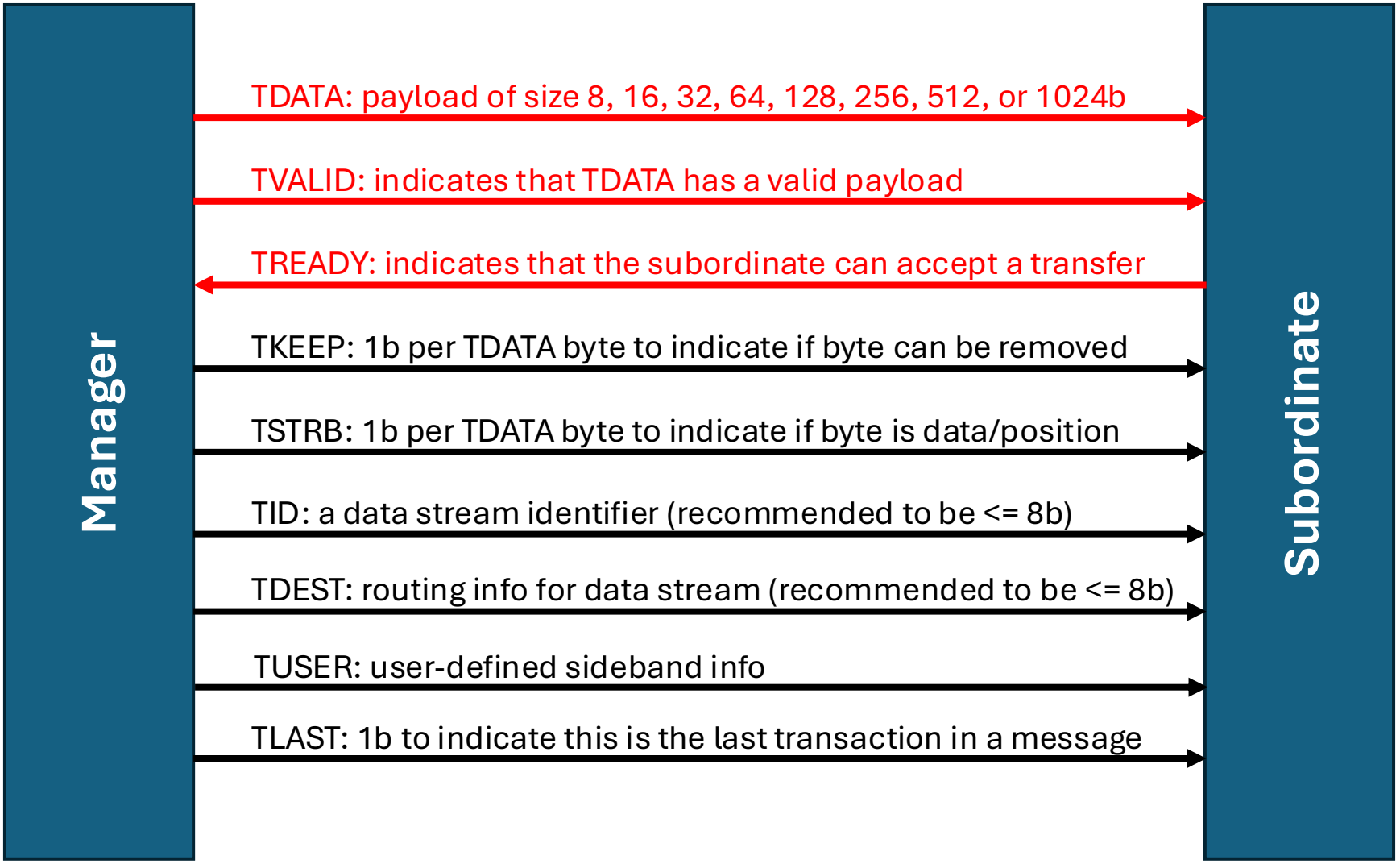


Types of AXI Interfaces

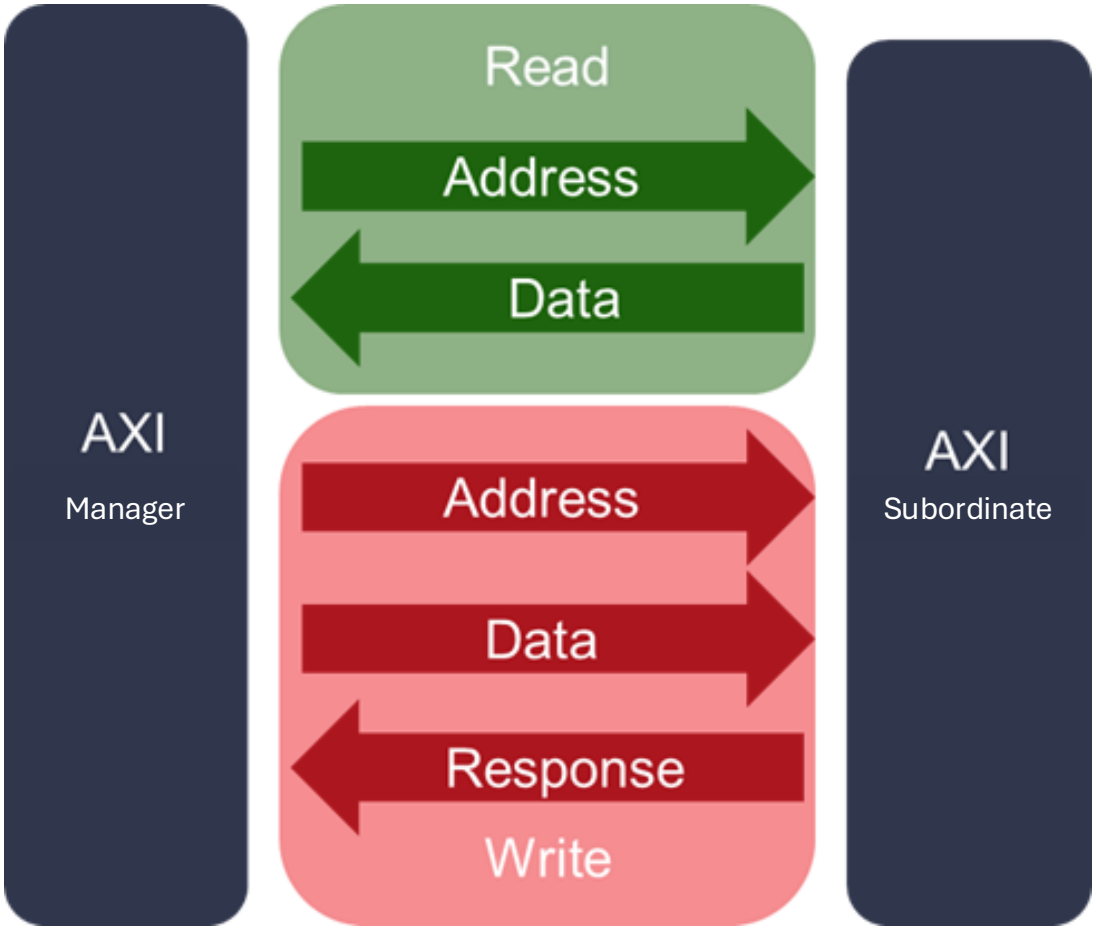


The AXI interface initiating a transaction is called the **Manager** and the AXI interface receiving or responding to a transaction is called the **Subordinate**

AXI Stream Interface Signals

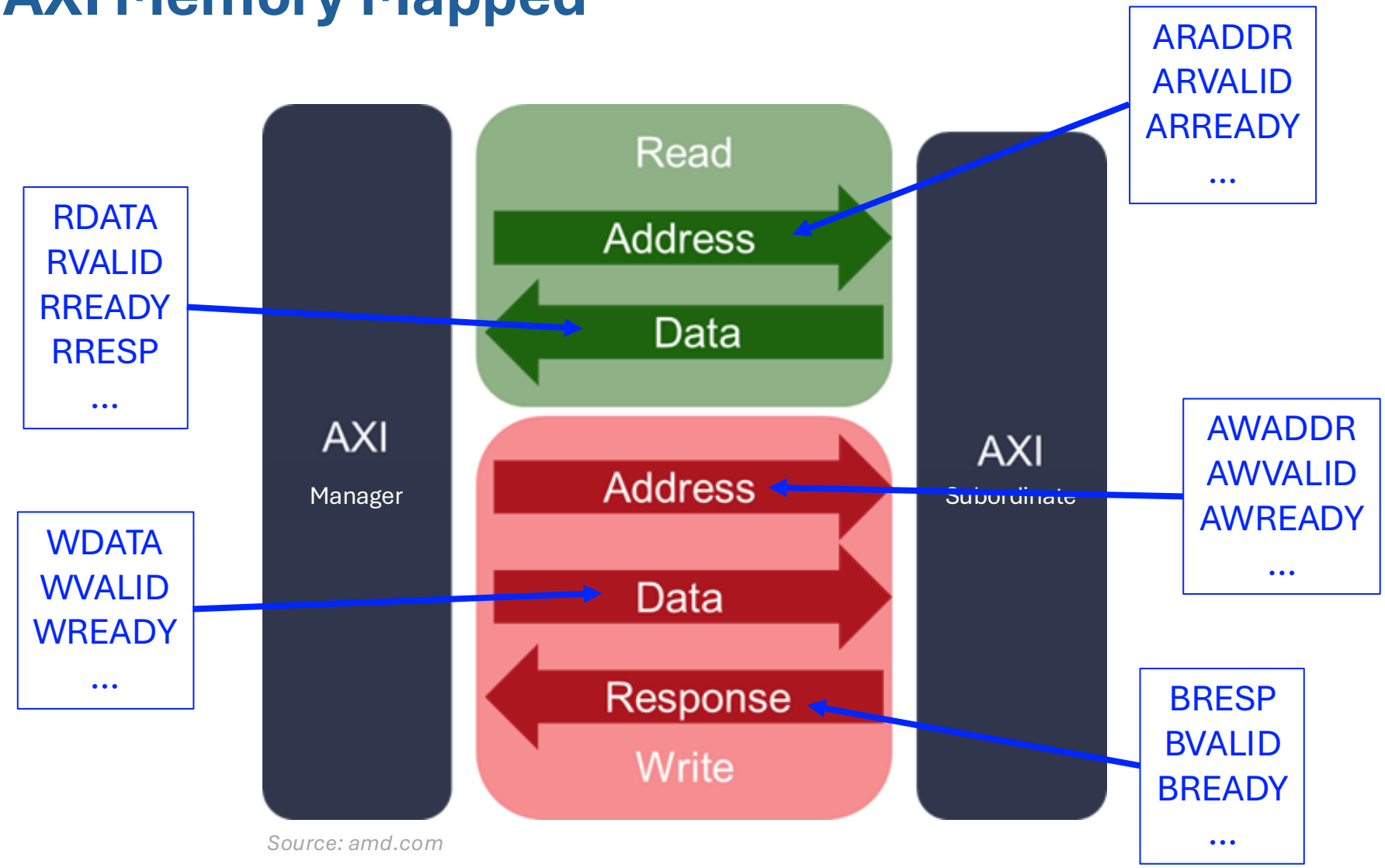


AXI Memory Mapped



Source: amd.com

AXI Memory Mapped



Source: amd.com

Next Lecture ...

FIFO-based design ...