ECE 327/627
Digital Hardware Systems

Lecture 7: Pipelining

Andrew Boutros

andrew.boutros@uwaterloo.ca

Some Logistics

- Lab 2 was released last Friday
  - Worth **8%** of the ECE 327 course grade
  - Worth **4%** of the ECE 627 course grade
  - Deadline on **Friday, Feb 6 @ 11:59 pm**

In the Previous Lecture …

- More FSM design examples:
  - FSM with multiple possible transitions per state – Chess Clock
  - Multiple interacting FSMs – Music Toy
  - FSM + Datapath (FSMD) – Polynomial on a multiply/add ALU

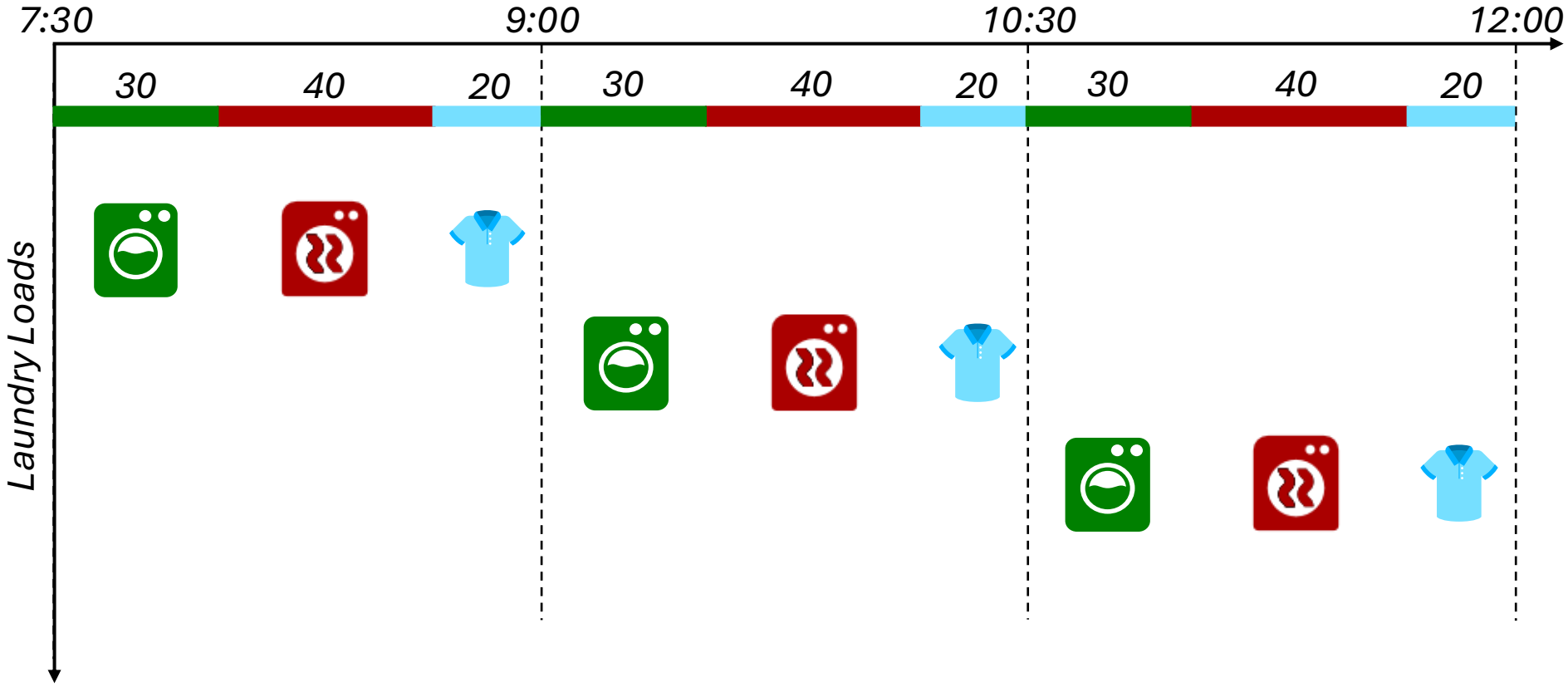
What is Pipelining?

What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity …

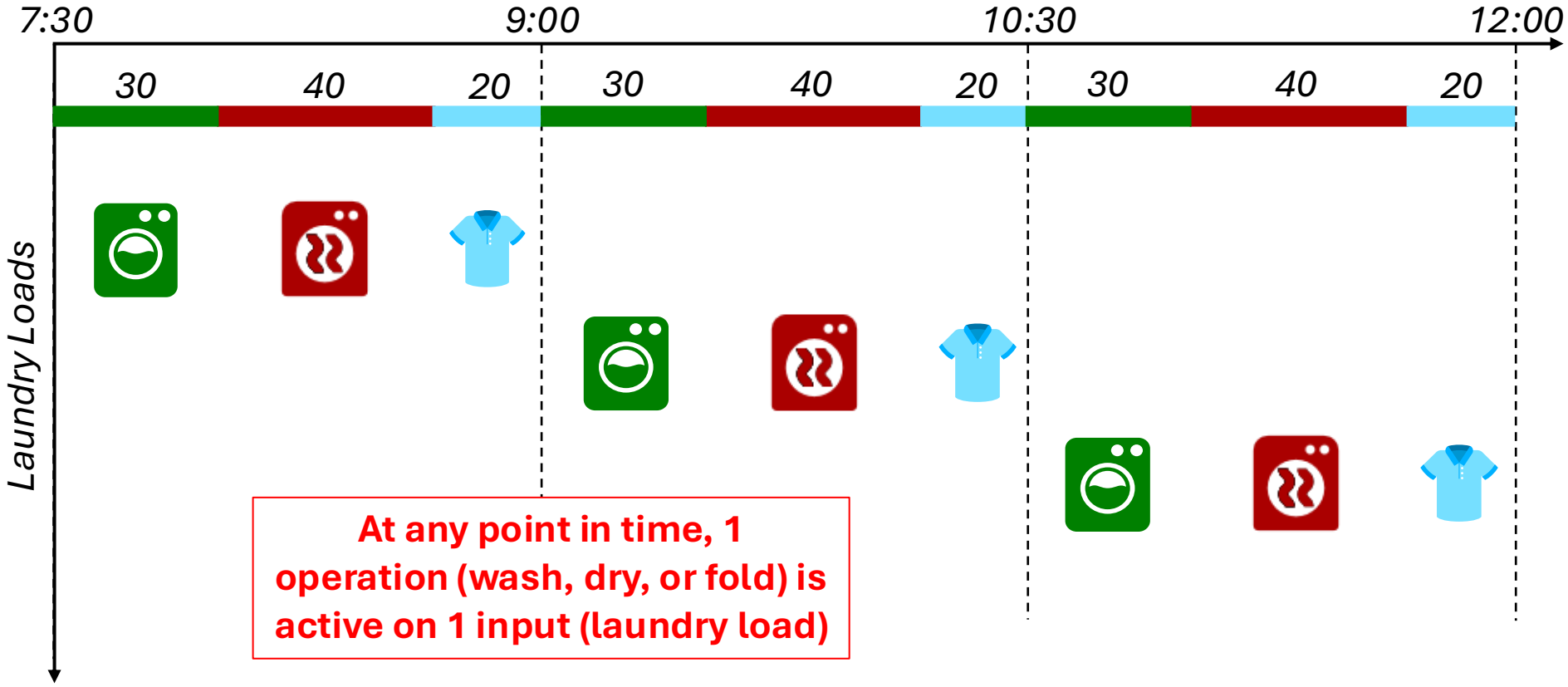
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



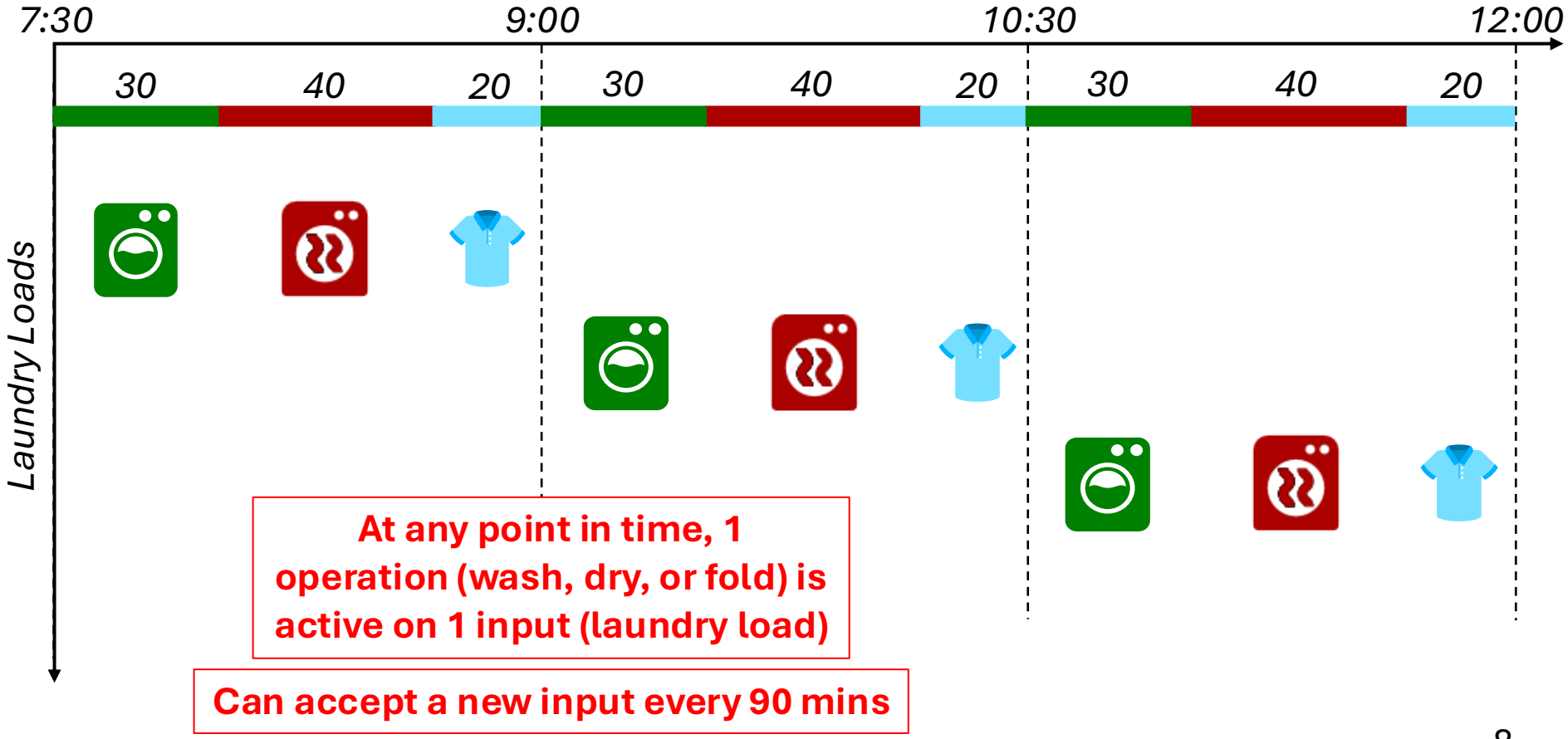
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



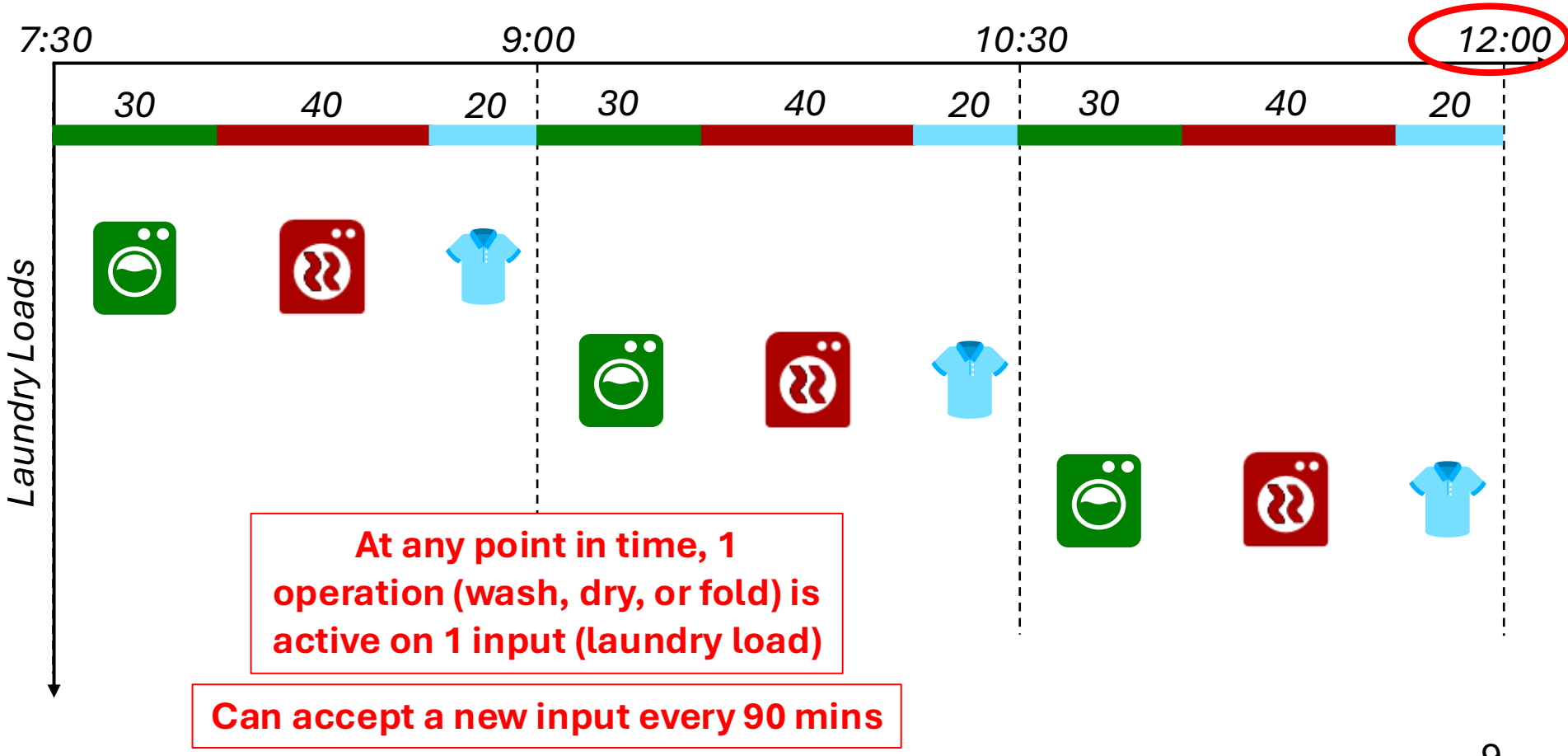
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



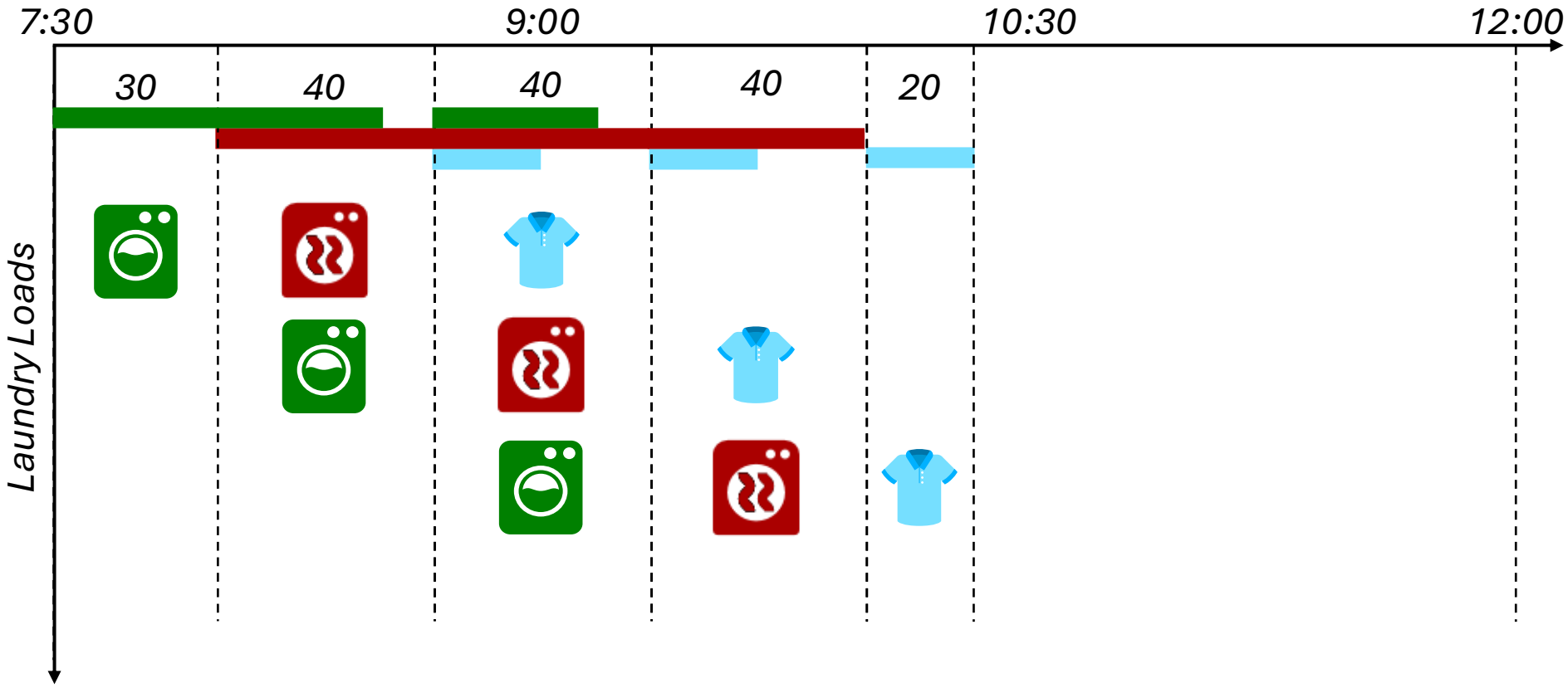
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



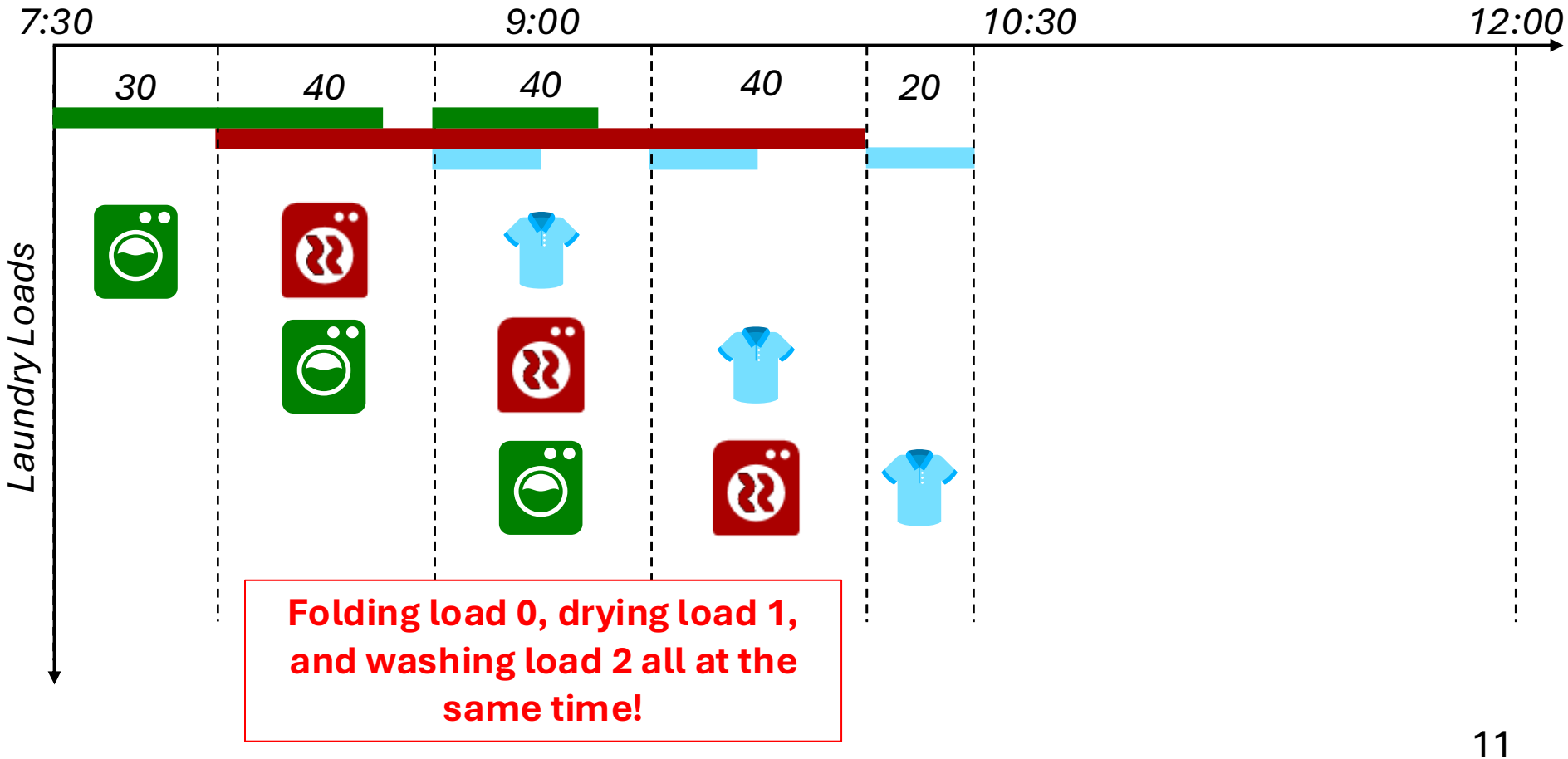
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



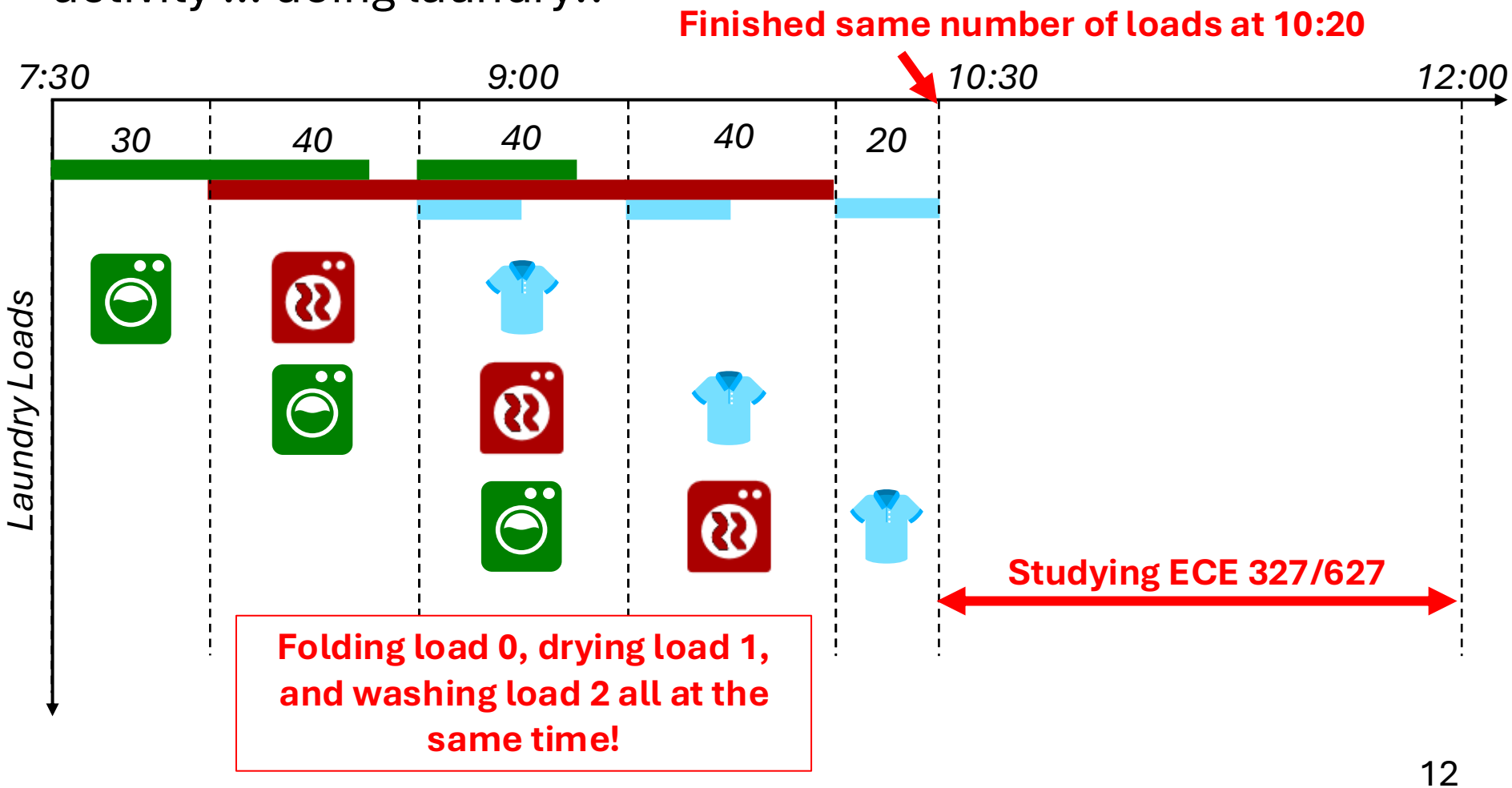
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



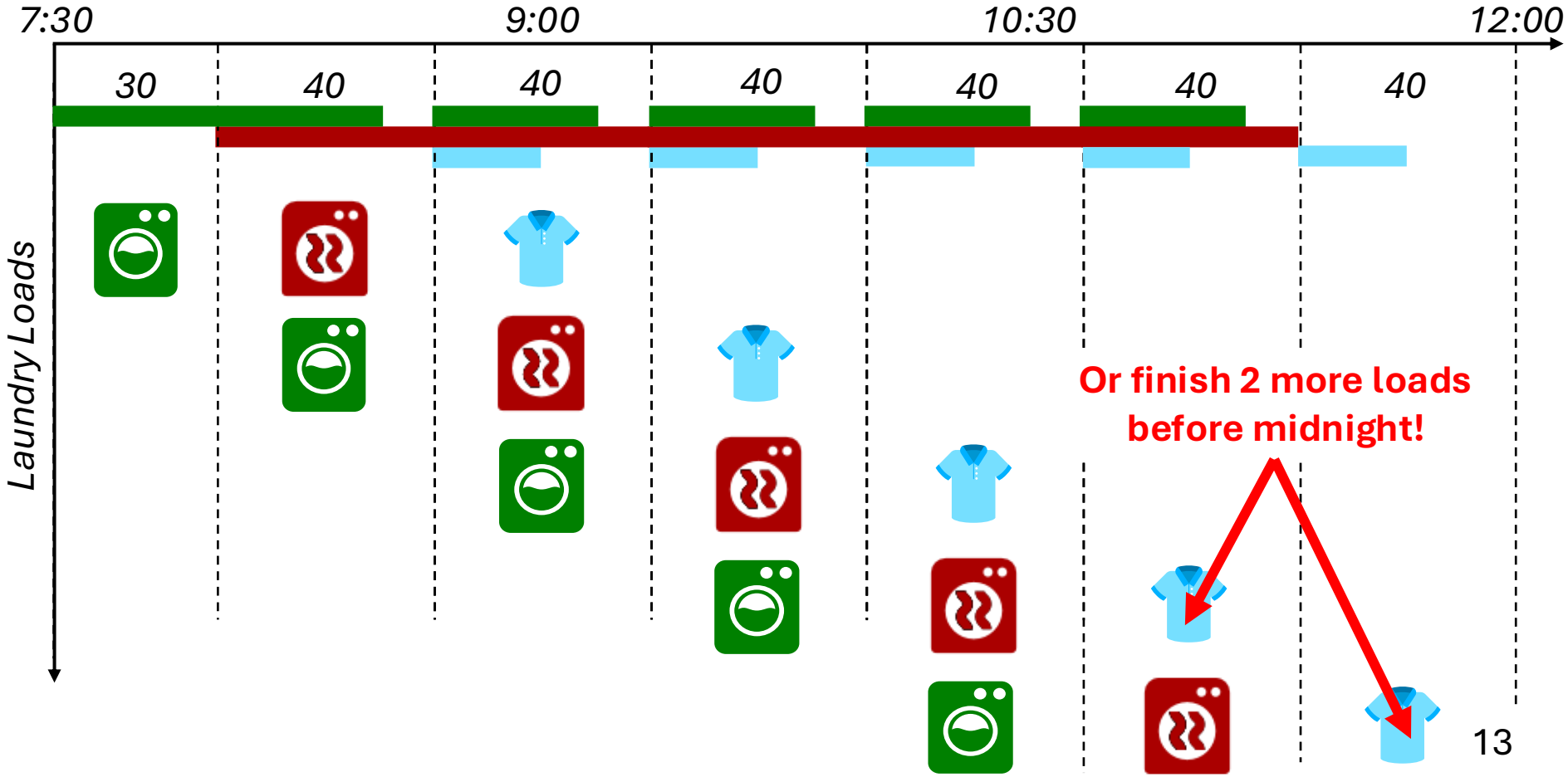
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



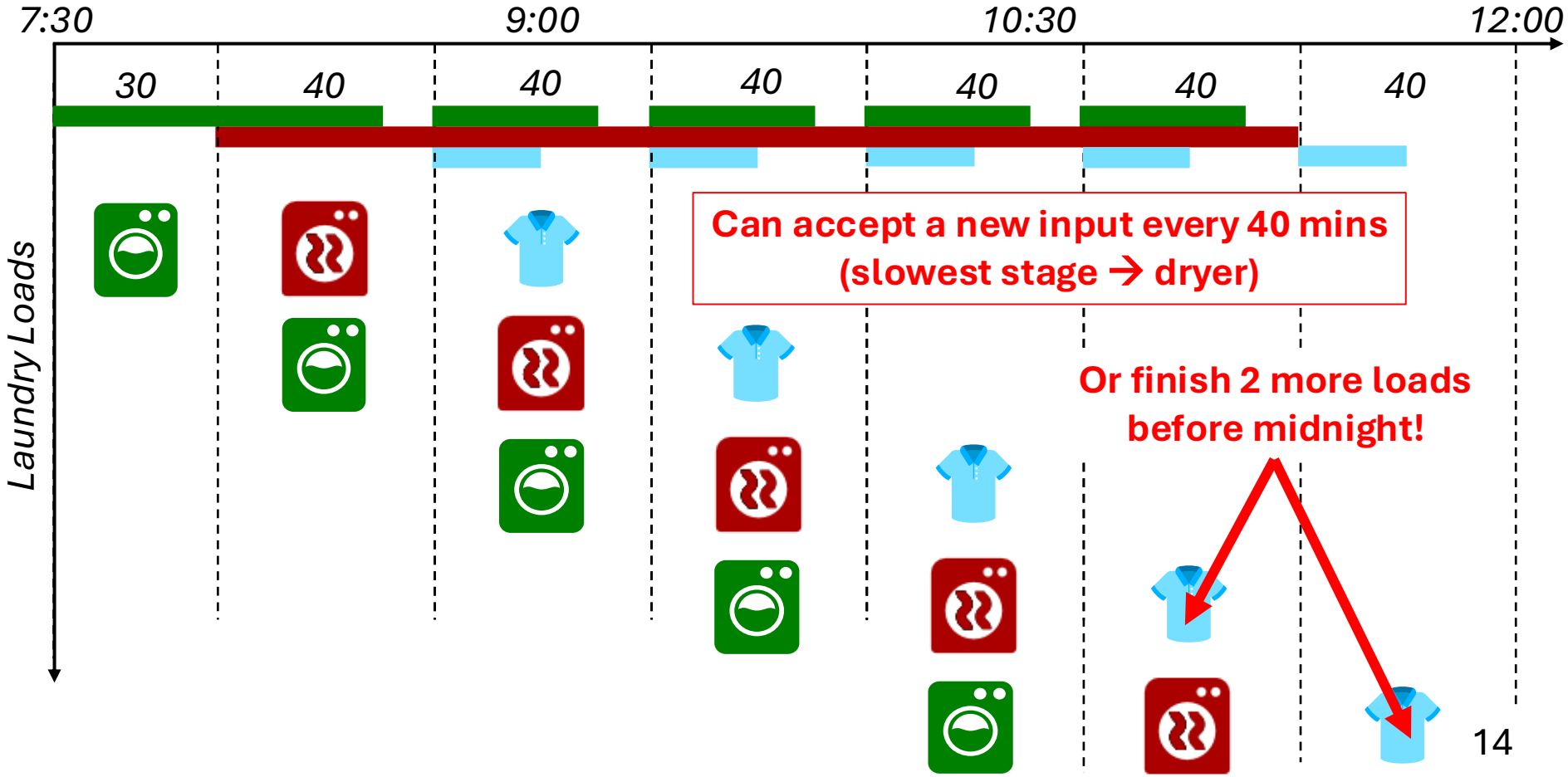
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



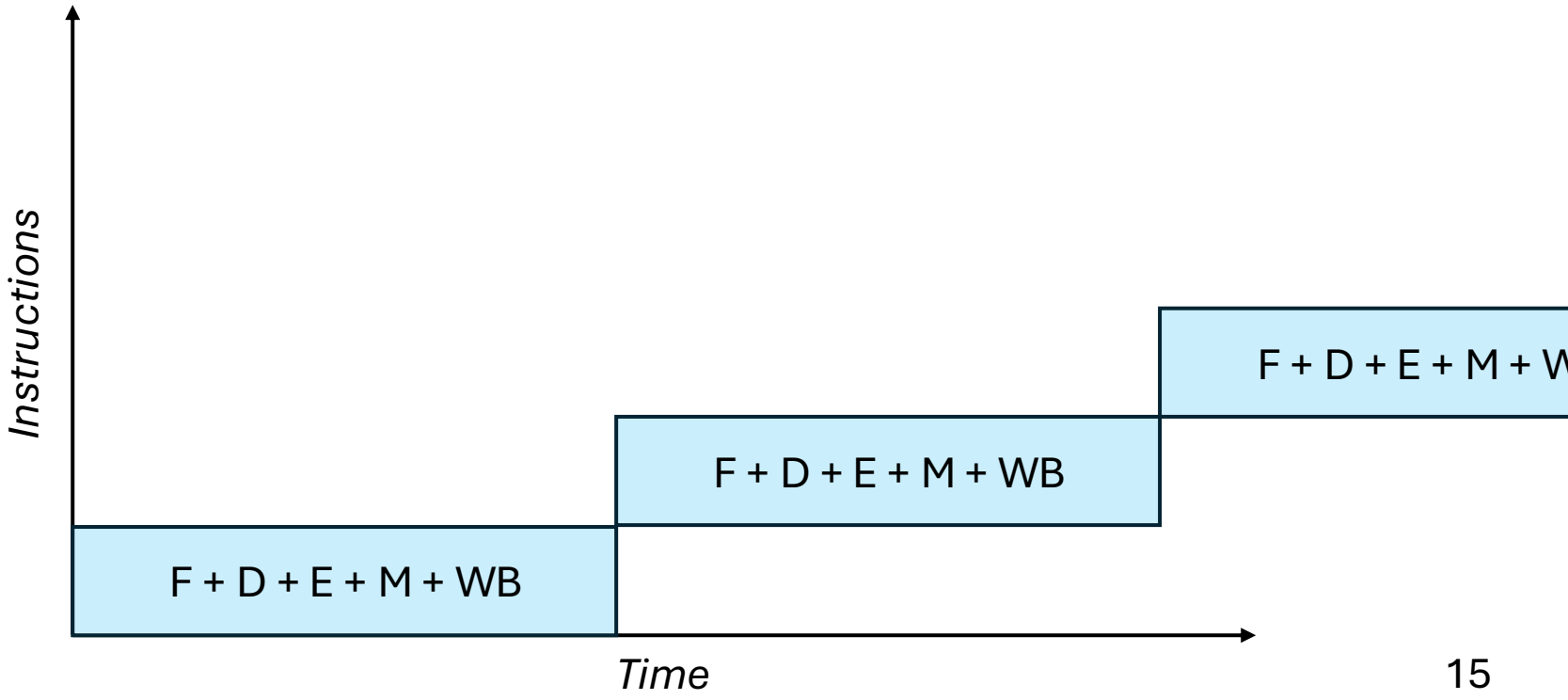
What is Pipelining?

Nice analogy to understand pipelining is everyone's favorite activity … doing laundry!!



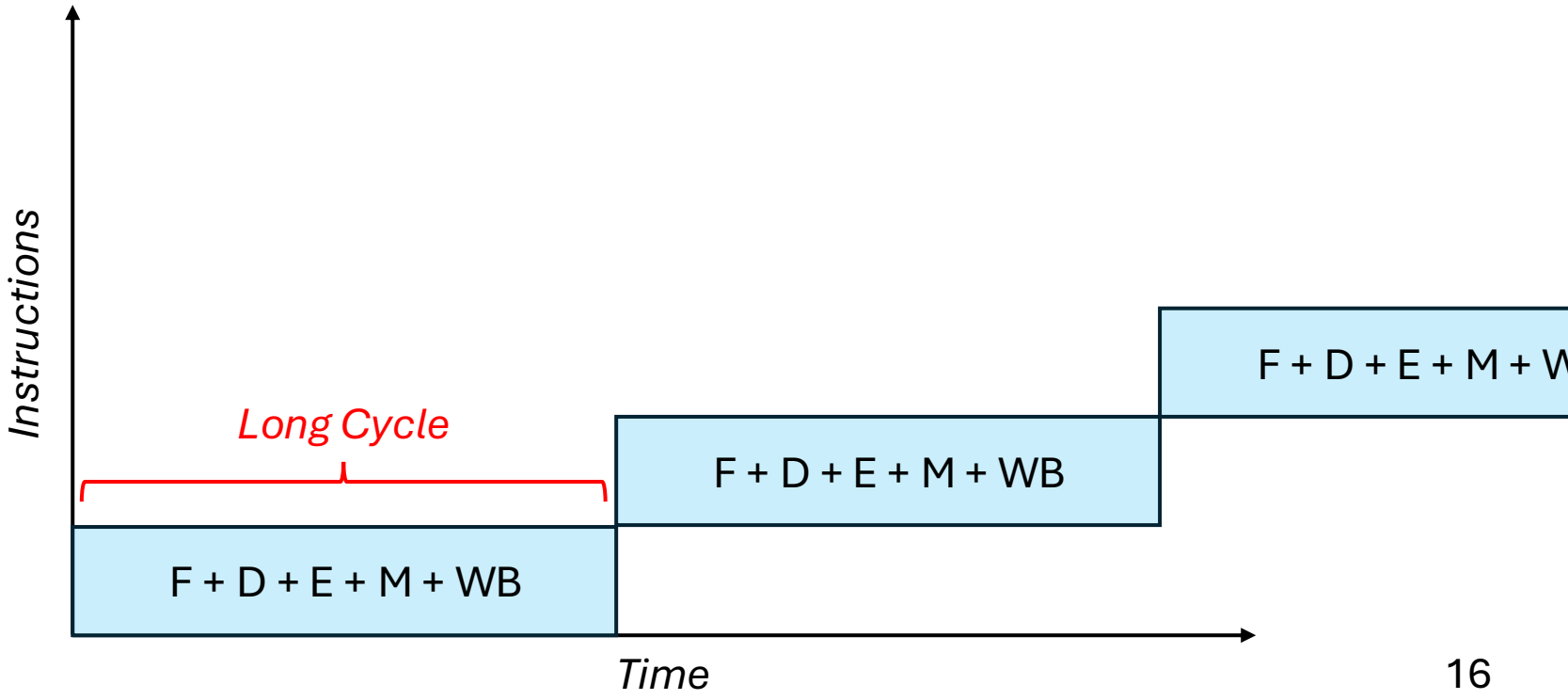
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back



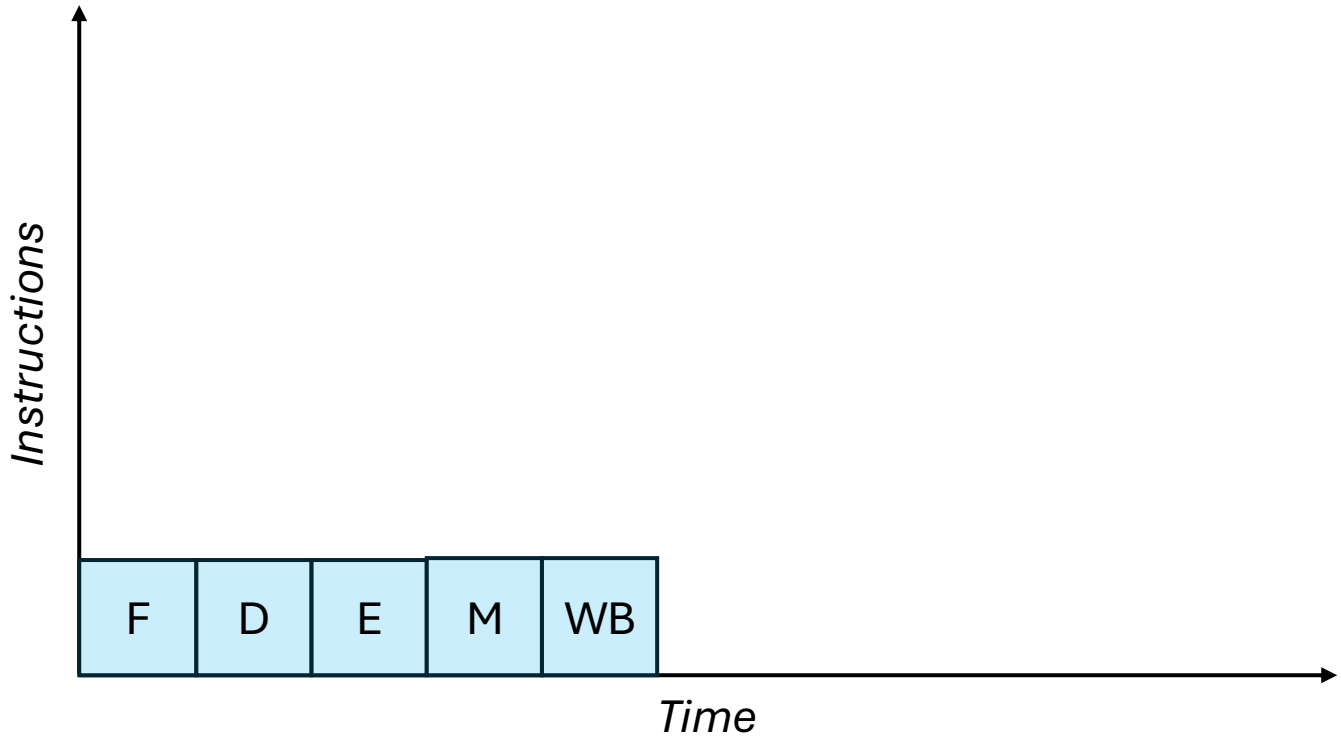
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back



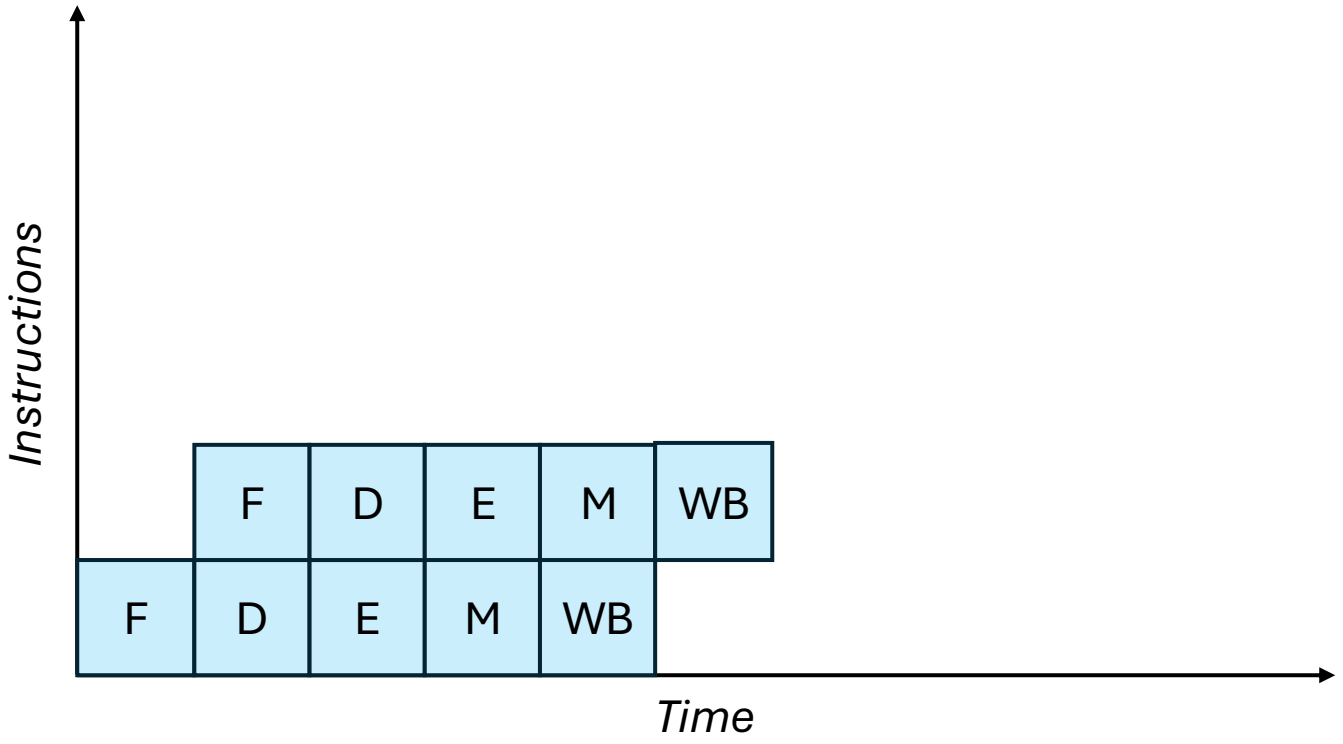
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



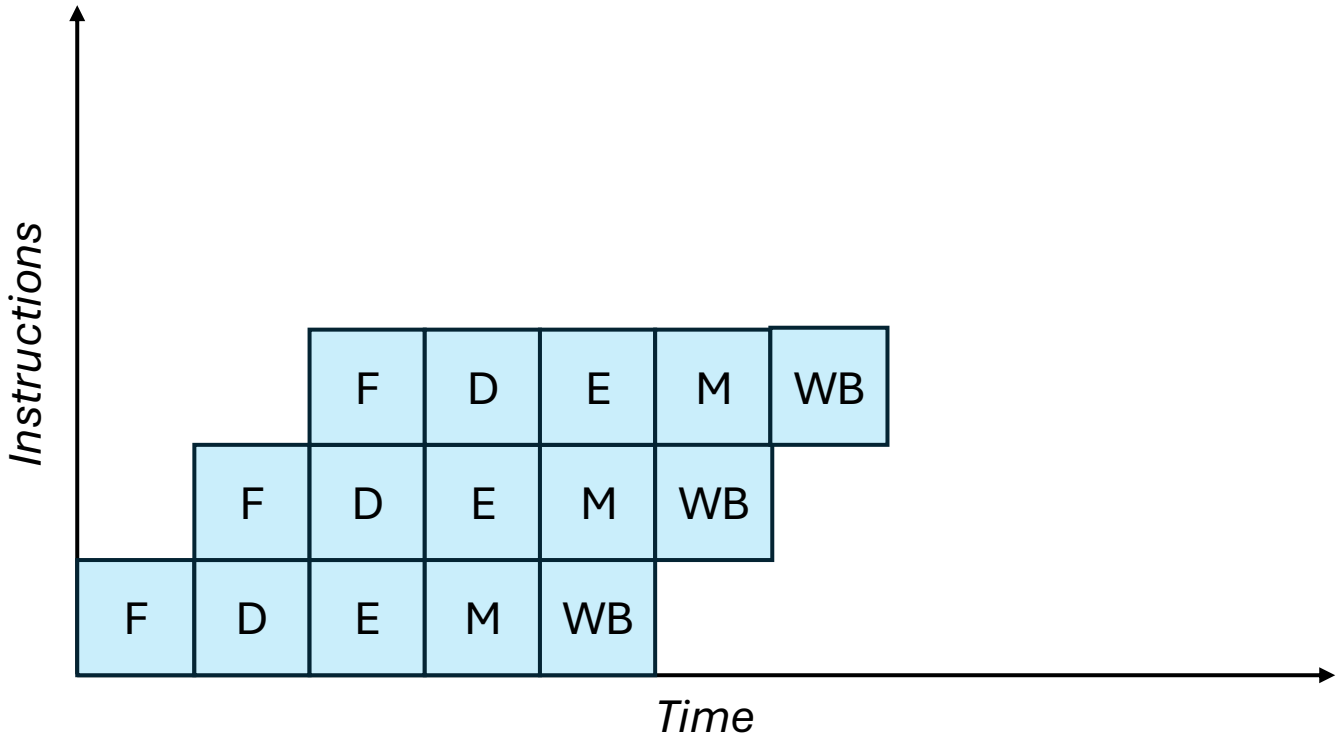
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



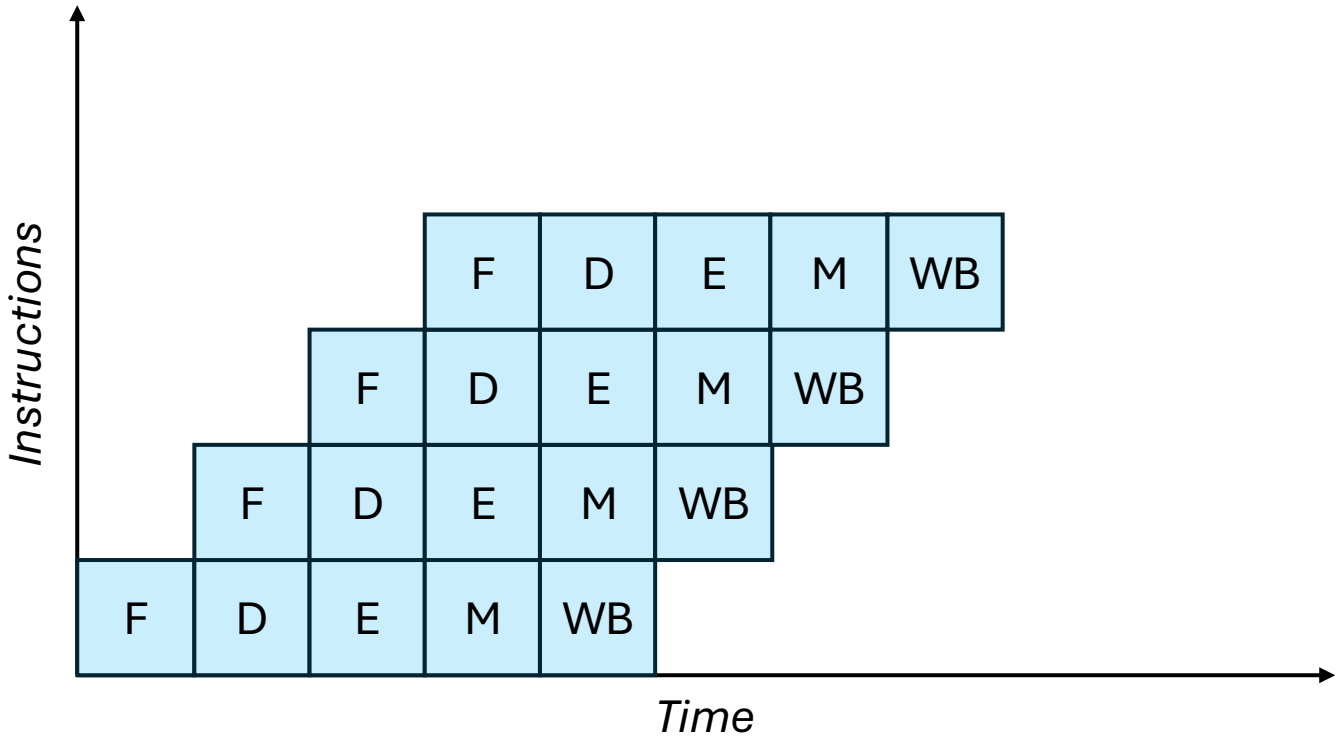
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



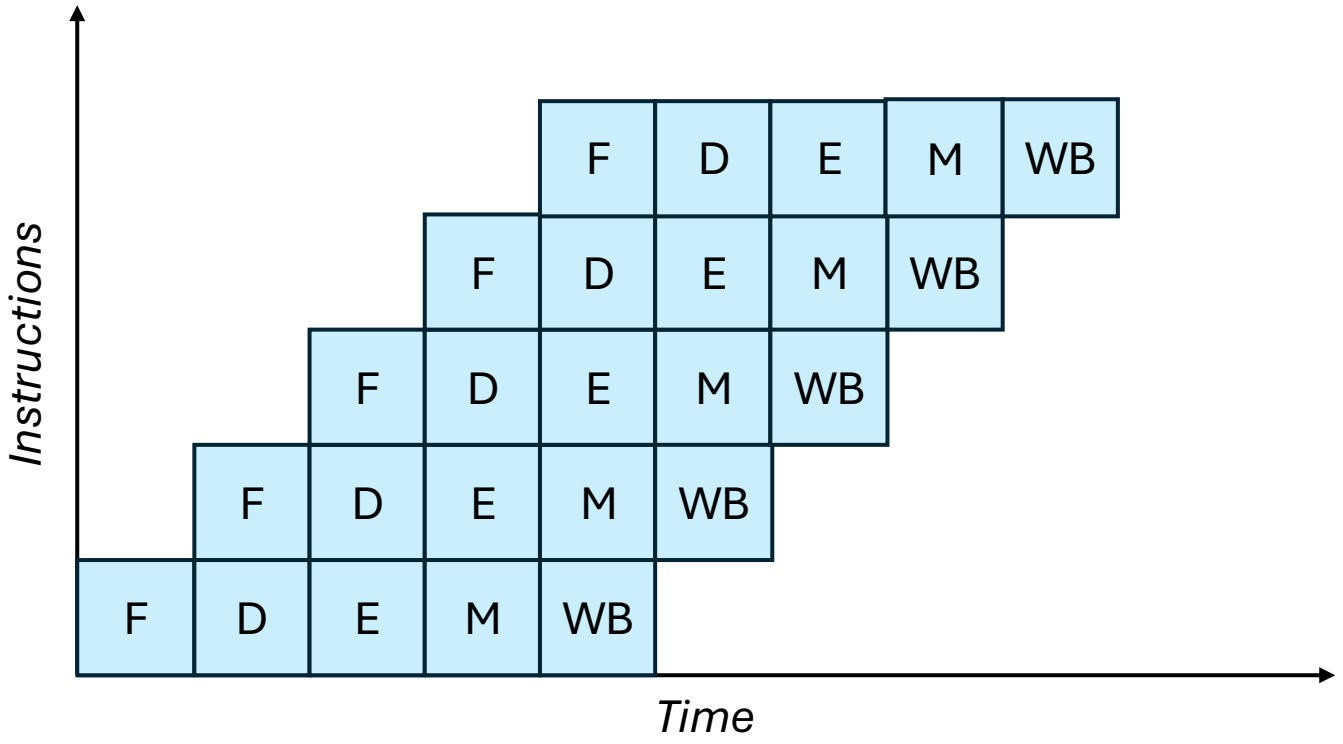
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



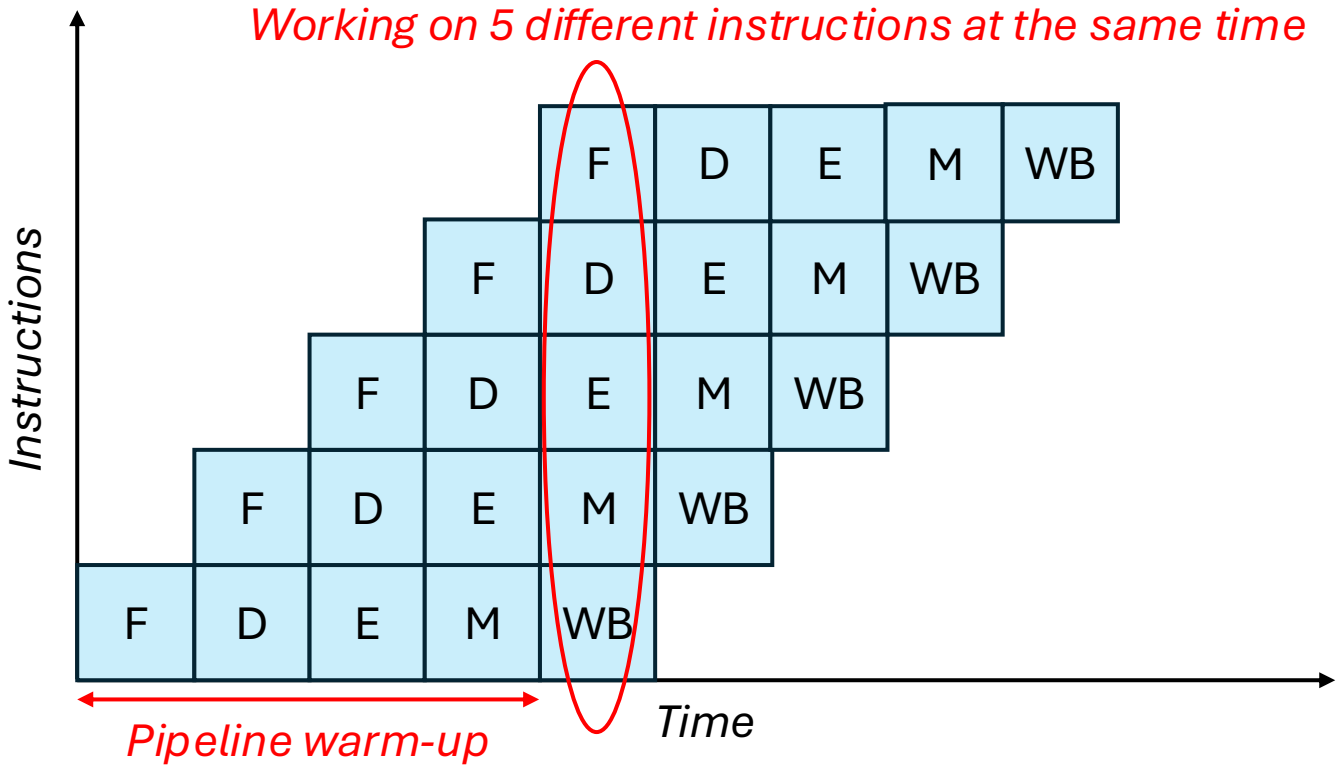
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



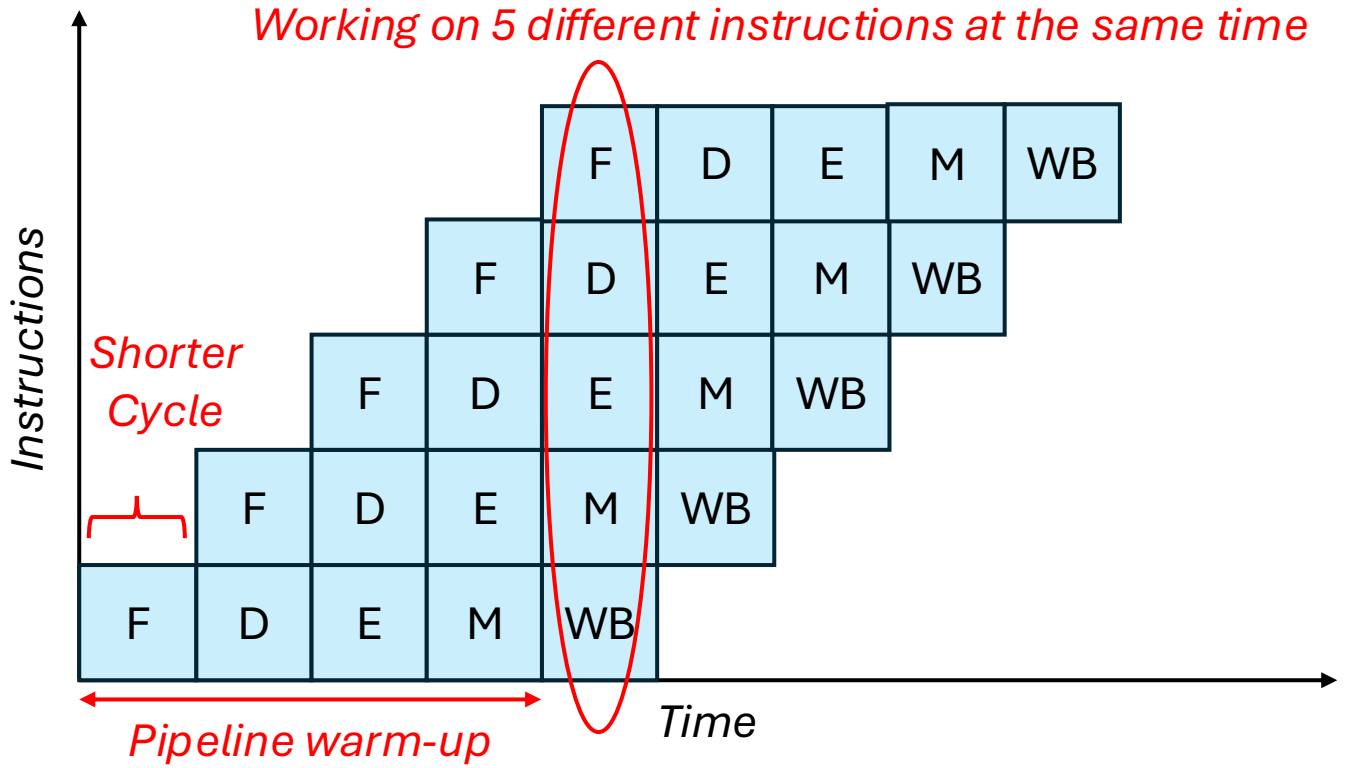
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



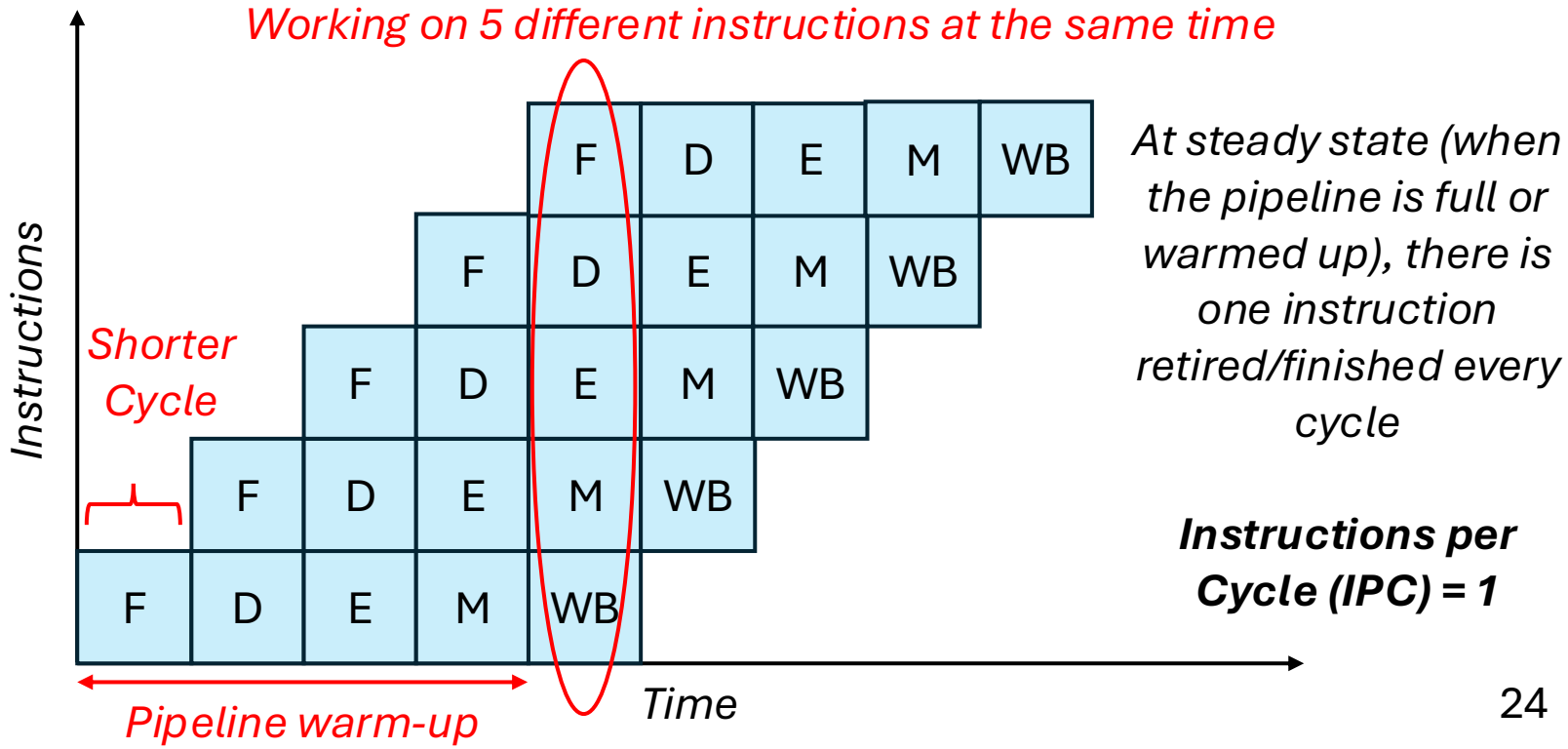
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



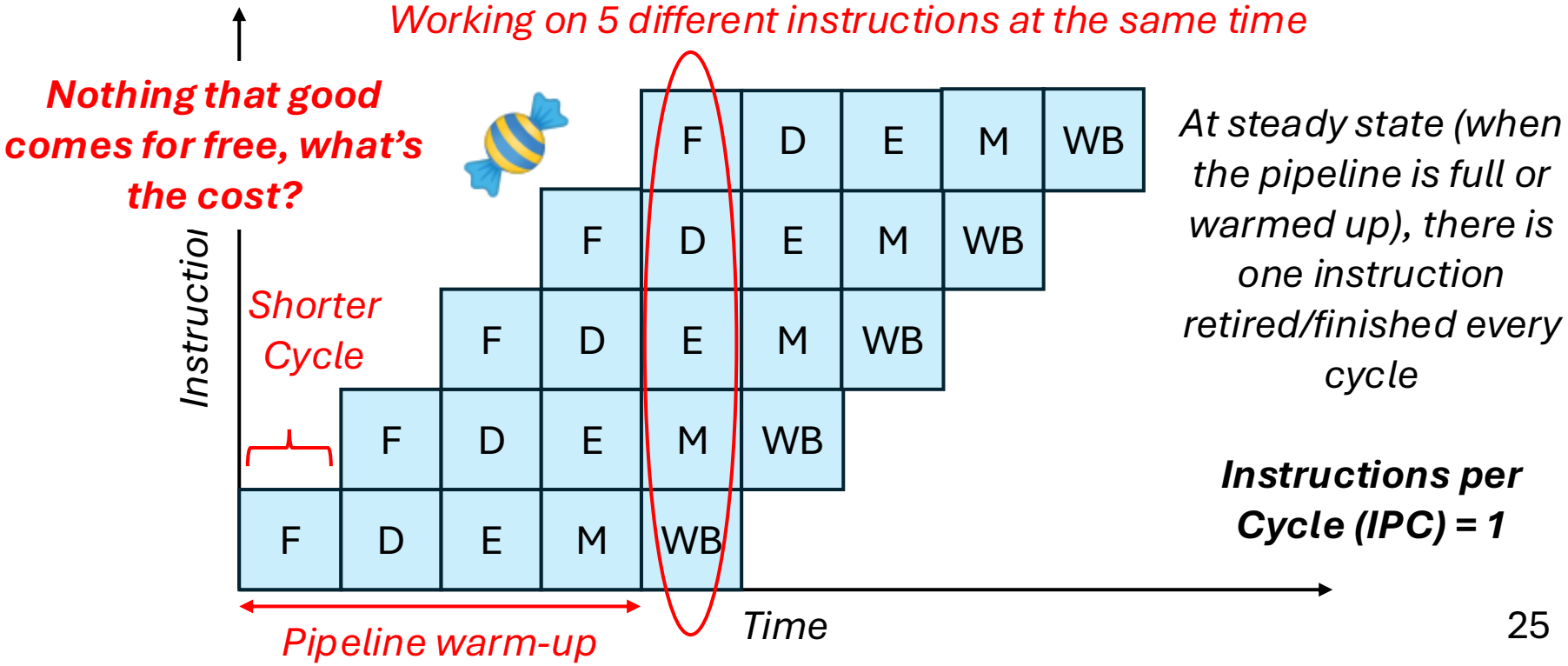
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



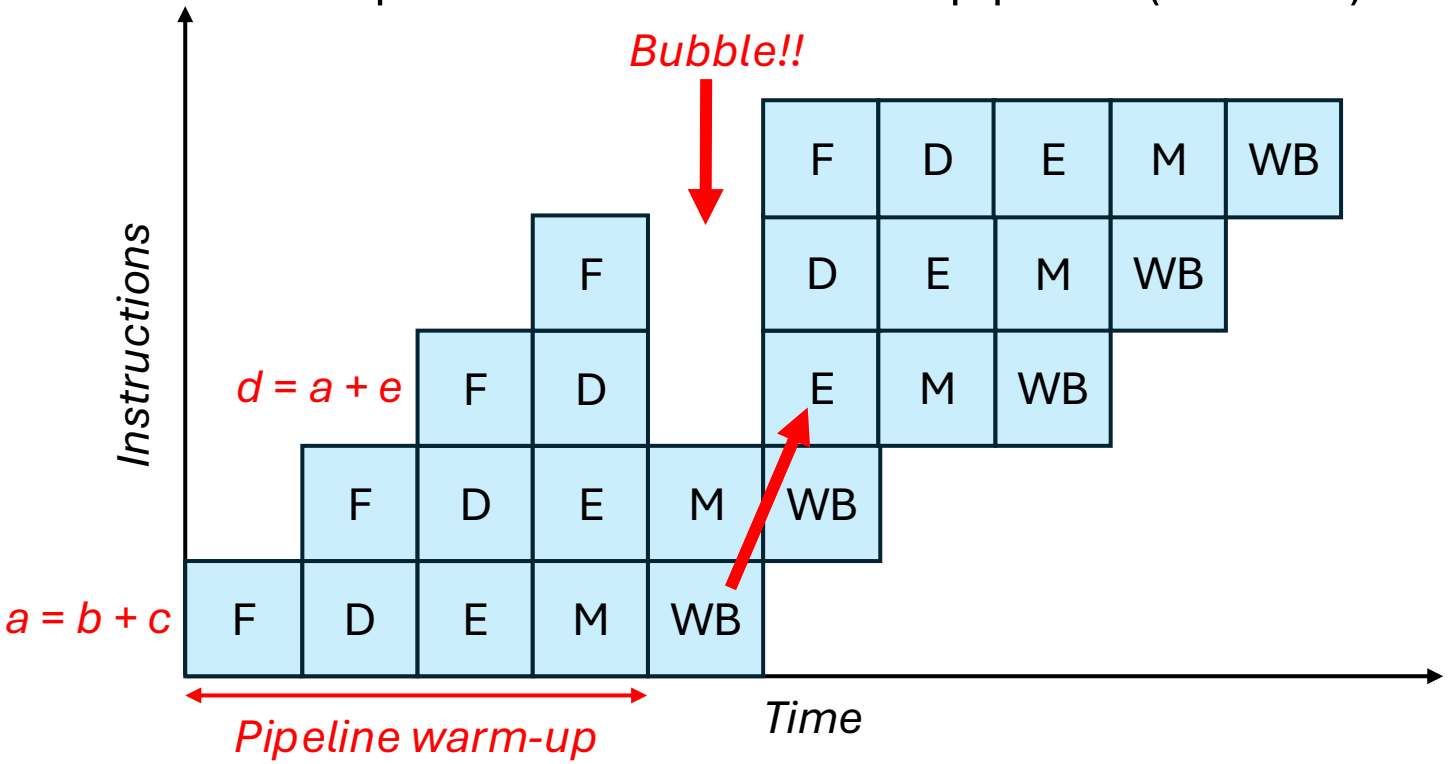
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle



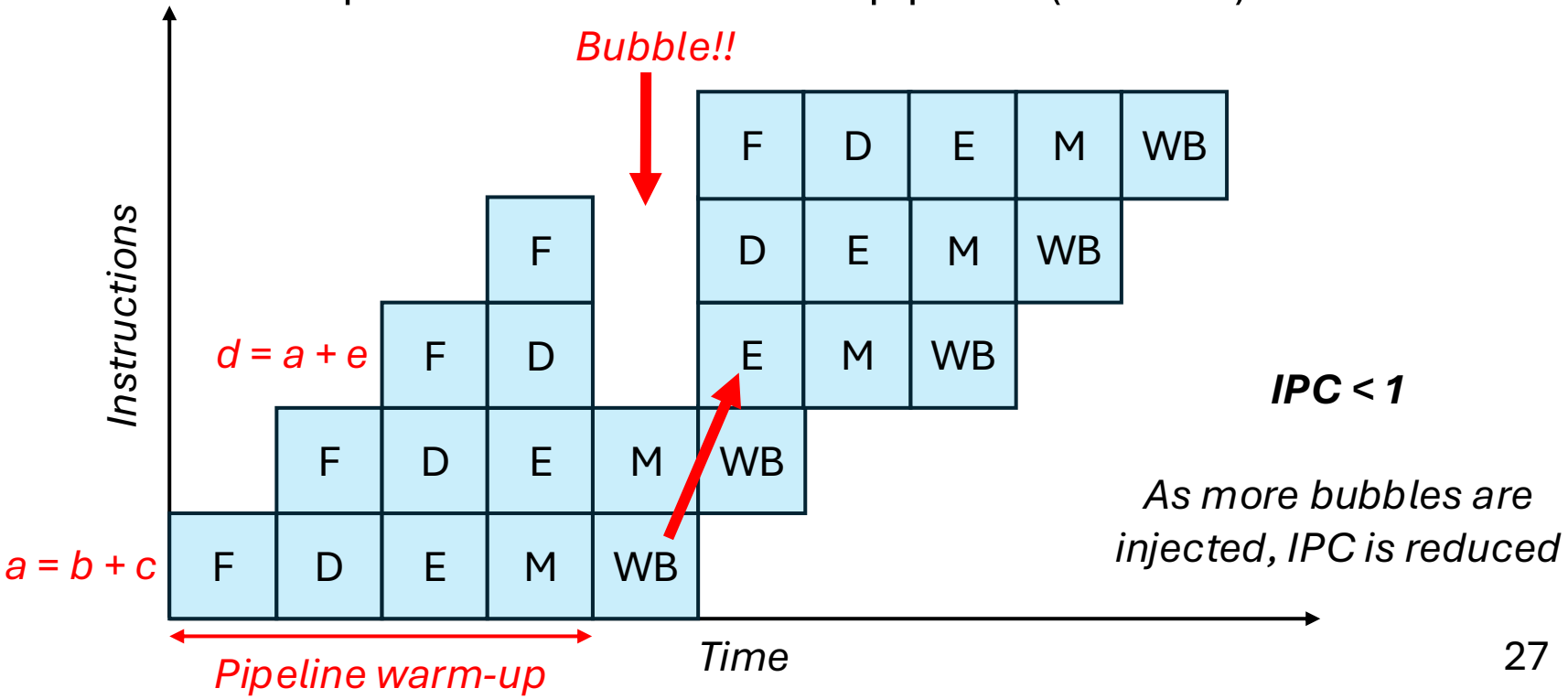
Pipelining in Modern CPUs

- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle
  - Check for dependencies and stall the pipeline (bubbles)

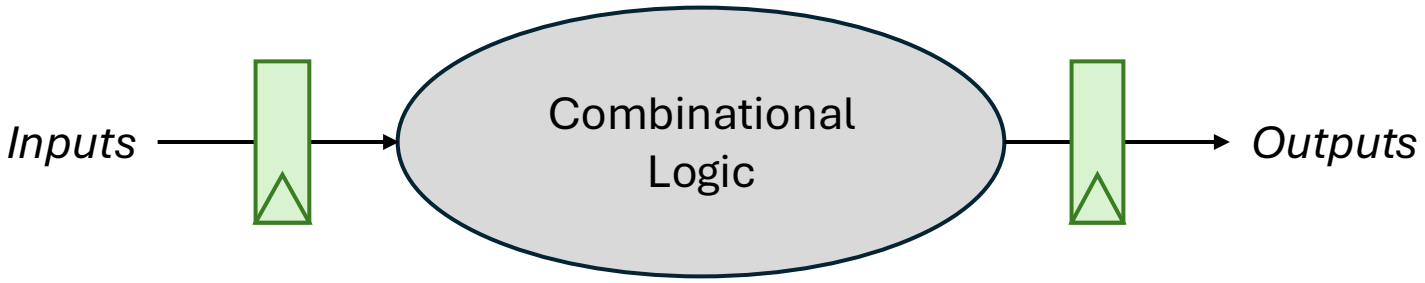


Pipelining in Modern CPUs

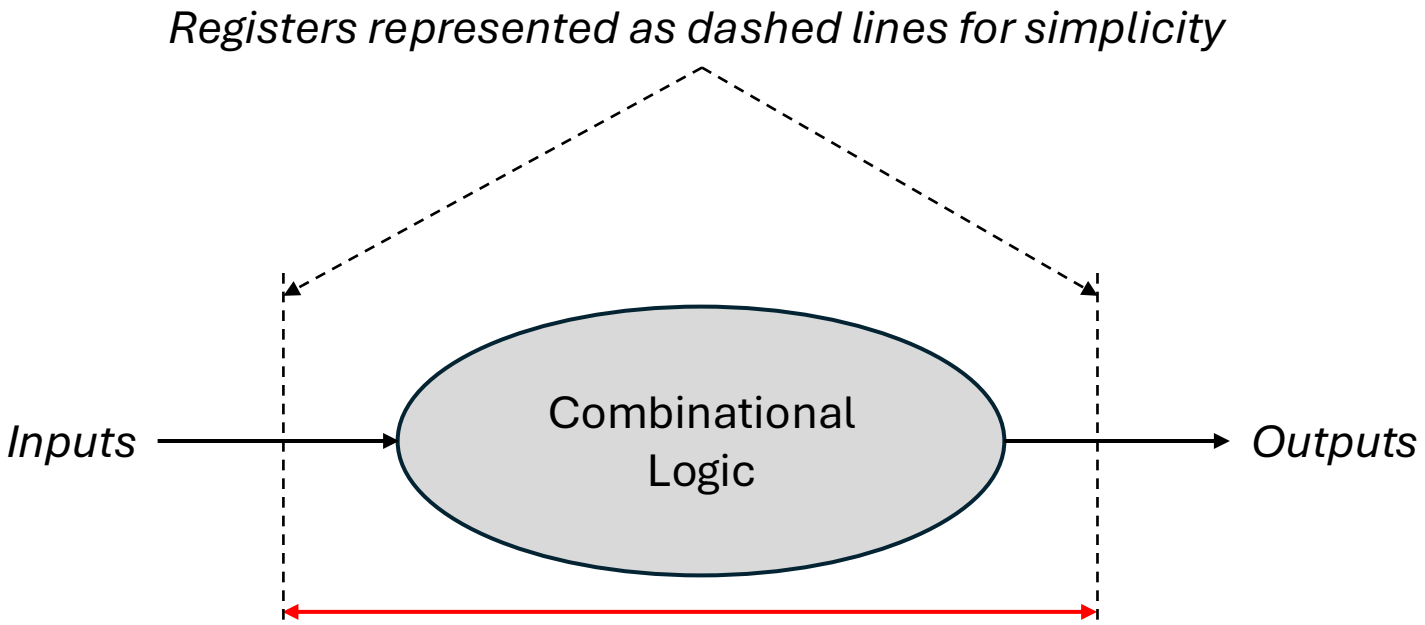
- A simple CPU can execute an entire instruction in 1 cycle
  - Fetch → Decode → Execute → Memory → Write back
- Better CPUs pipeline each state into a cycle
  - Overlap execution of different instructions in the same clock cycle
  - Check for dependencies and stall the pipeline (bubbles)



Pipelining Digital Circuits



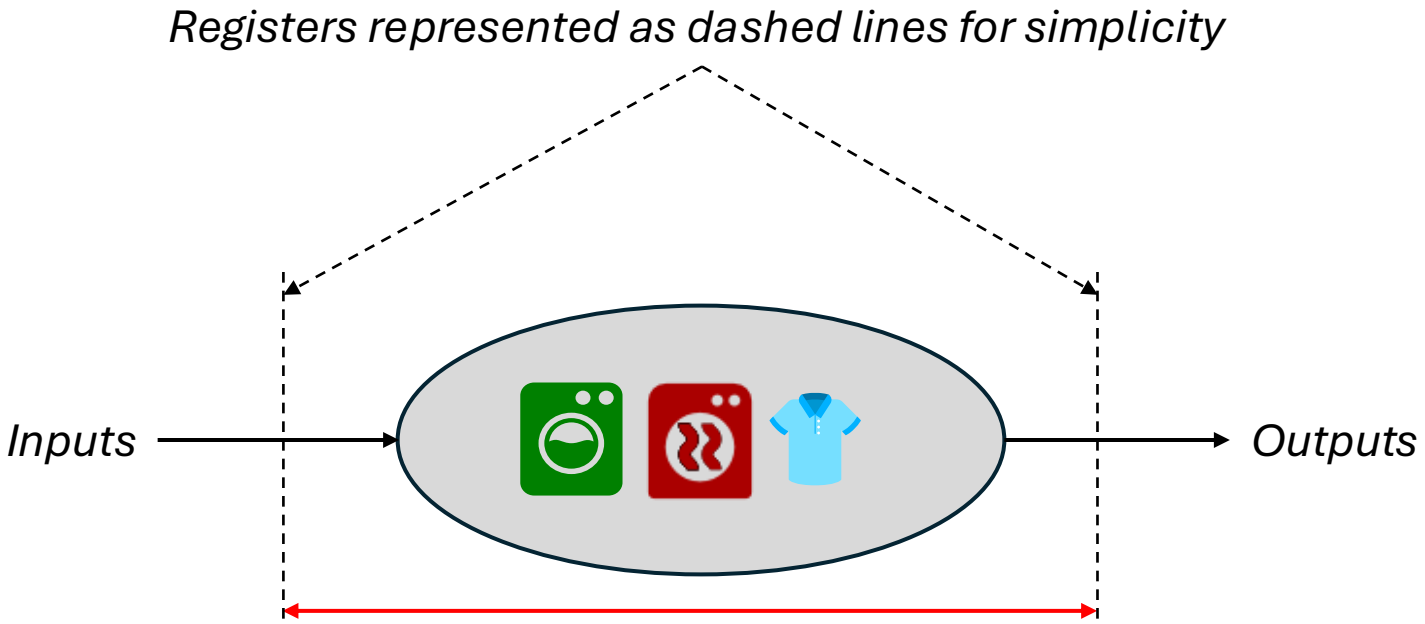
Pipelining Digital Circuits



Clock period must be >= longest path between two registers

(Critical Path)

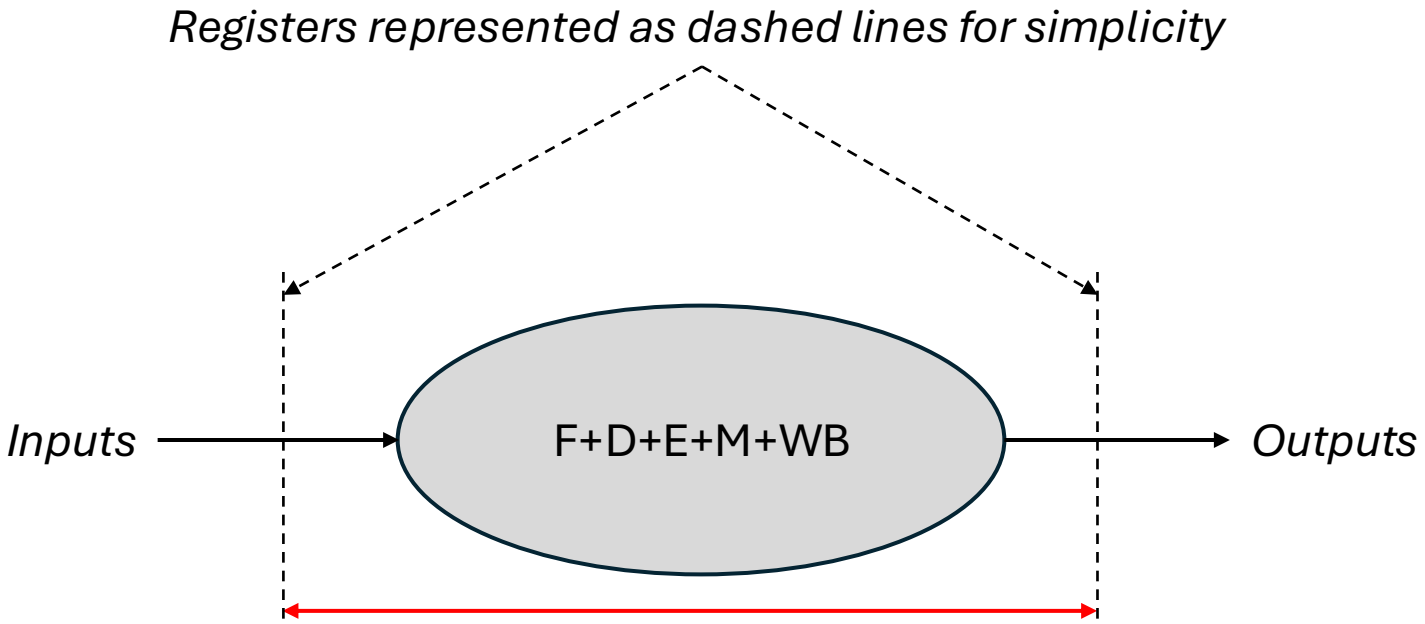
Pipelining Digital Circuits



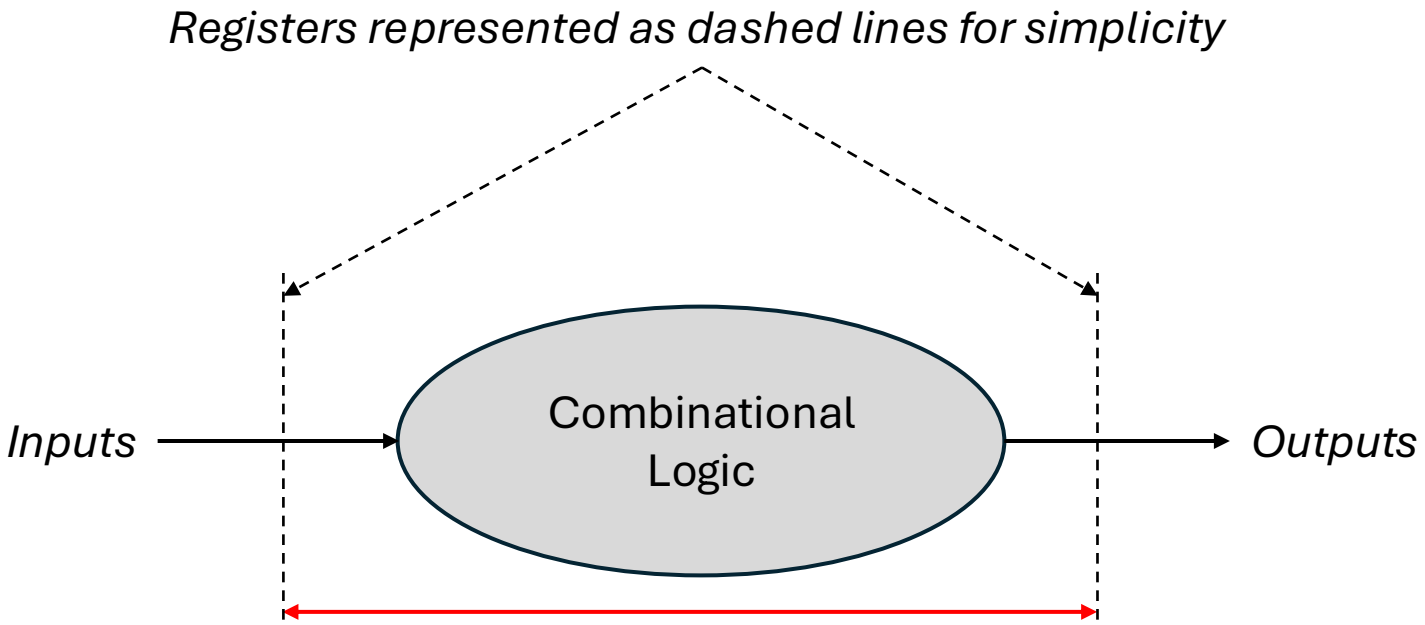
Clock period must be >= longest path between two registers

(Critical Path)

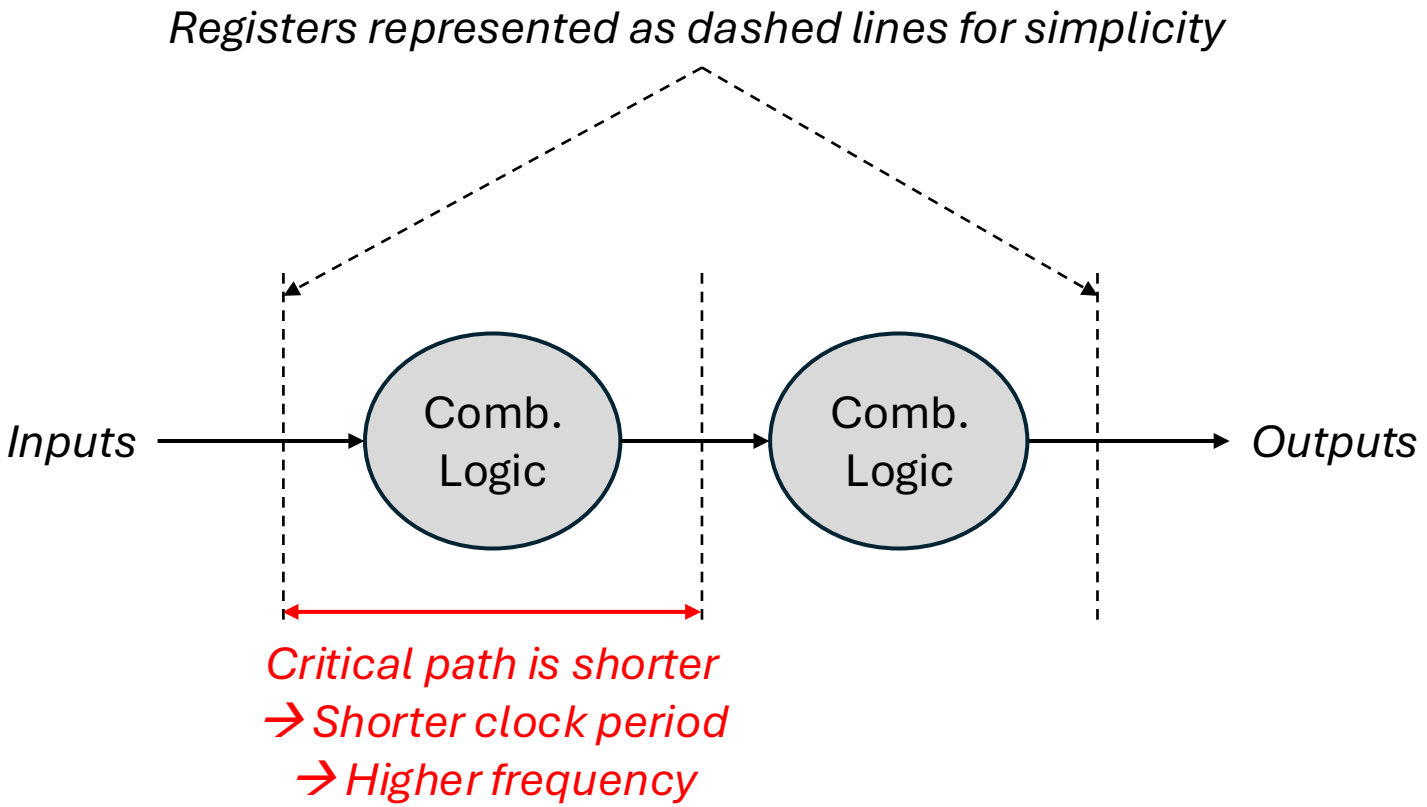
Pipelining Digital Circuits



Pipelining Digital Circuits



Pipelining Digital Circuits



Pipelining Terminology

- **Throughput:** Operations performed per second (ops/sec)
  - Operations per cycle x cycles per second (frequency)

Pipelining Terminology

- **Throughput:** Operations performed per second (ops/sec)
  - Operations per cycle x cycles per second (frequency)

- **Latency:** Time taken for an input to propagate through the circuit and produce an output (sec)
  - No. of cycles x clock period

Pipelining Terminology

- **Throughput:** Operations performed per second (ops/sec)
  - Operations per cycle x cycles per second (frequency)

- **Latency:** Time taken for an input to propagate through the circuit and produce an output (sec)
  - No. of cycles x clock period

- **Clock Period:** Time separation between clock edges
  - Faster clock frequency → smaller clock period
  - Must run the circuit as fast as required by the design specifications

Pipelining Terminology

- **Throughput:** Operations performed per second (ops/sec)
  - Operations per cycle x cycles per second (frequency)

- **Latency:** Time taken for an input to propagate through the circuit and produce an output (sec)
  - No. of cycles x clock period

- **Clock Period:** Time separation between clock edges
  - Faster clock frequency → smaller clock period
  - Must run the circuit as fast as required by the design specifications

- **Critical Path:** The delay of the longest path between registers
  - Determines the smallest safe clock period to operate at

Pipelining Effect

For a given circuit, more pipelining means …

Critical Path ↓

Clock Period ↓

Frequency ↑

No. of Cycles ↑

Pipelining Effect

For a given circuit, more pipelining means …

Critical Path ↓

Clock Period ↓

Frequency ↑

No. of Cycles ↑

Throughput = Operations per cycle x Frequency

Pipelining Effect

For a given circuit, more pipelining means …

Critical Path ↓

Clock Period ↓

Frequency ↑

No. of Cycles ↑

Throughput = Operations per cycle x Frequency

Latency = No. of Cycles x Clock Period

Pipelining Effect

For a given circuit, more pipelining means …

Critical Path ↓

Clock Period ↓

Frequency ↑

No. of Cycles ↑

Throughput = Operations per cycle x Frequency

Latency = No. of Cycles x Clock Period

If that's the full picture, should we just pipeline indefinitely?!

Pipelining Effect

For a given circuit, more pipelining means …

Critical Path ↓

Clock Period ↓

Frequency ↑

No. of Cycles ↑

Throughput = Operations per cycle x Frequency

Latency = No. of Cycles x Clock Period

In practice, you typically want to optimize one while meeting a certain spec on the other

If that's the full picture, should we just pipeline indefinitely?!

Pipelining Effect

For a given circuit, more pipelining means …

Critical Path ↓

Clock Period ↓

Frequency ↑

No. of Cycles ↑

Throughput = Operations per cycle x Frequency

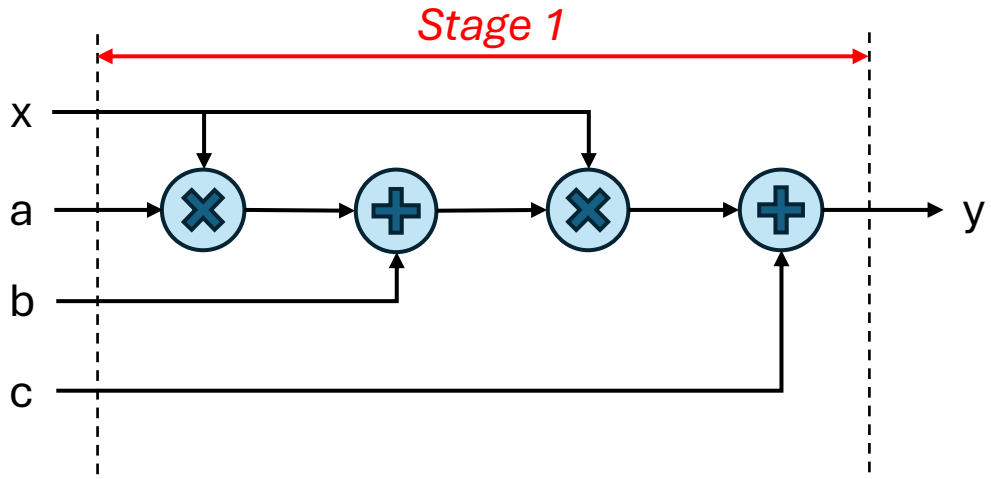
Latency = No. of Cycles x Clock Period

Silicon Area ↑

Power ↑

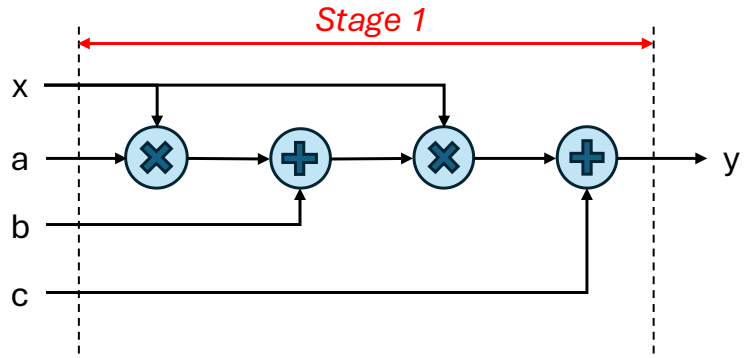
Clock Distribution Network Design Complexity ↑

Example: Polynomial Circuit

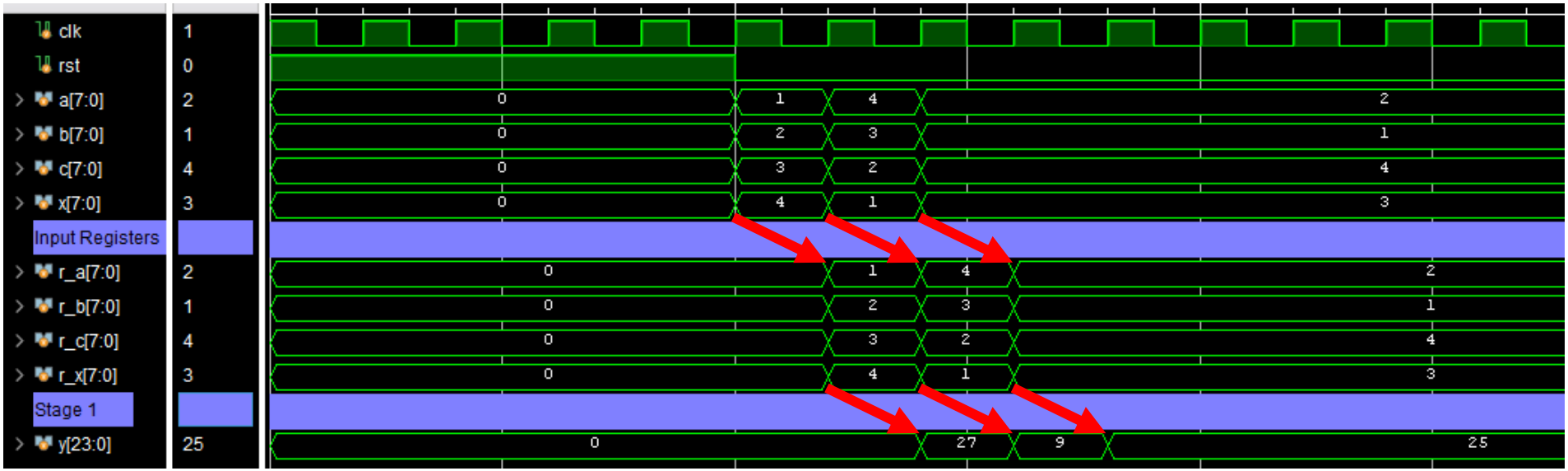



```
module poly_1stage (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y
);

logic [7:0] r_a, r_b, r_c, r_x;

always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        y <= 0;
    end else begin
        // Input registers
        r_a <= a; r_b <= b;
        r_c <= c; r_x <= x;
        // Stage 1
        y <= ((r_a * r_x) + r_b) * r_x + r_c;
    end
end

endmodule
```

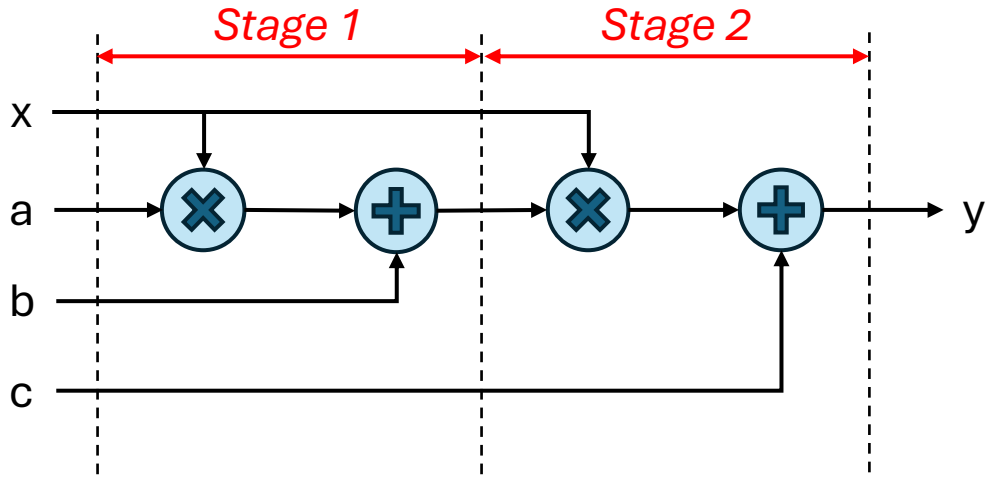
Example: Polynomial Circuit



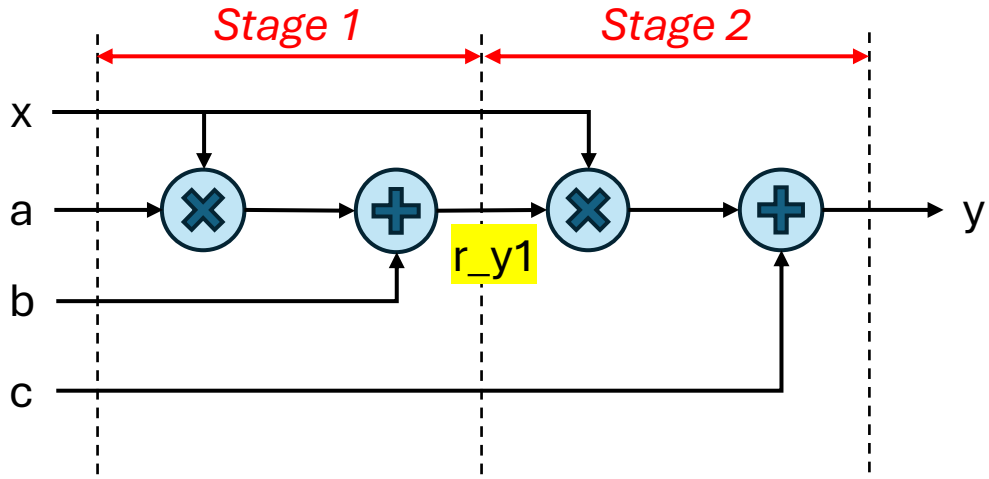
Clock Period = 11.4ns



Example: Polynomial Circuit

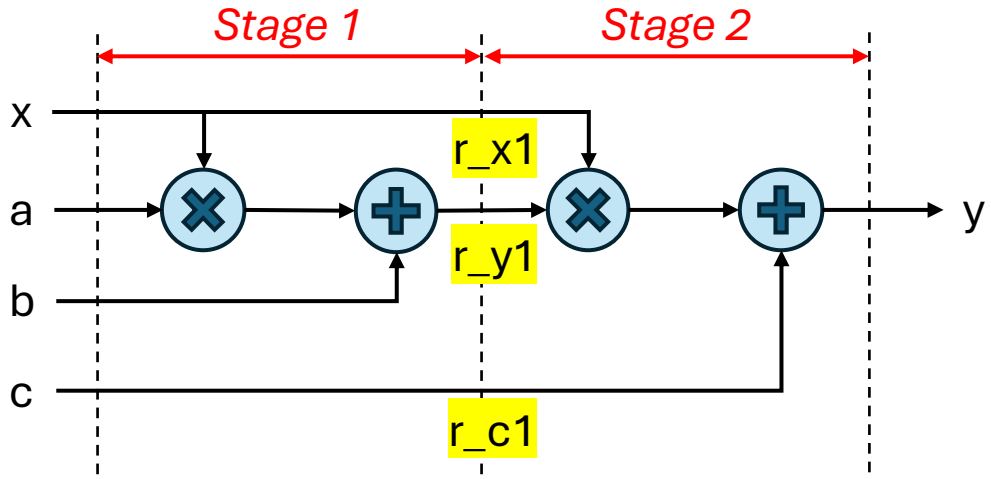


Example: Polynomial Circuit



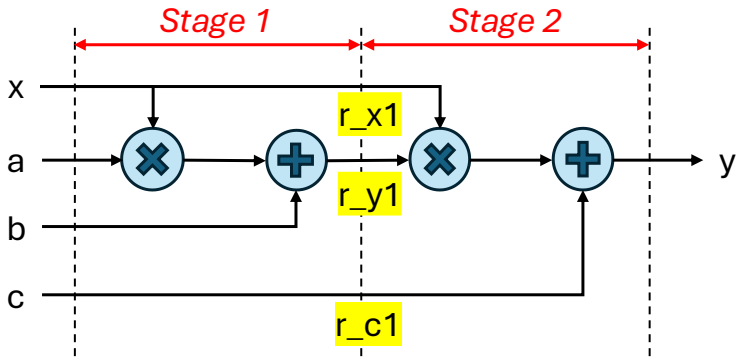

```
module poly_2stage (
       input clk,
       input rst,
       input [7:0] a,
       input [7:0] b,
       input [7:0] c,
       input [7:0] x,
       output logic [23:0] y
);

logic [7:0] r_a, r_b, r_c, r_x;
logic [15:0] r_y1;


always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 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_b;

        // Stage 2
        y <= (r_y1 * r_x) + r_c;
    end
end

endmodule
```

Example: Polynomial Circuit

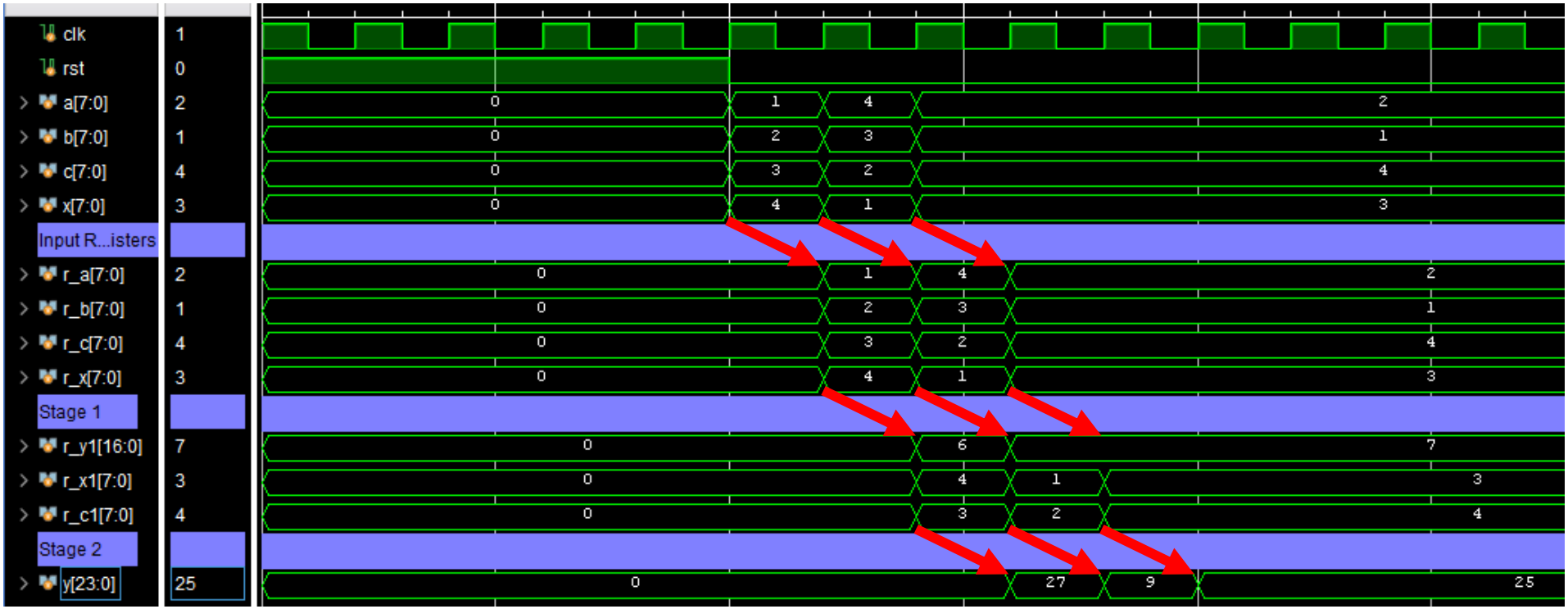



```
module poly_2stage (
       input clk,
       input rst,
       input [7:0] a,
       input [7:0] b,
       input [7:0] c,
       input [7:0] x,
       output logic [23:0] y
);

logic [7:0] r_a, r_b, r_c, r_x;
logic [15:0] r_y1;
logic [7:0] r_x1, r_c1;

always_ff @ (posedge clk) begin
    if (rst) begin
        r_a <= 0; r_b <= 0;
        r_c <= 0; r_x <= 0;
        r_y1 <= 0; r_x1 <= 0; r_c1 <= 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_b;
        r_x1 <= r_x; r_c1 <= r_c;
        // Stage 2
        y <= (r_y1 * r_x1) + r_c1;
    end
end

endmodule
```

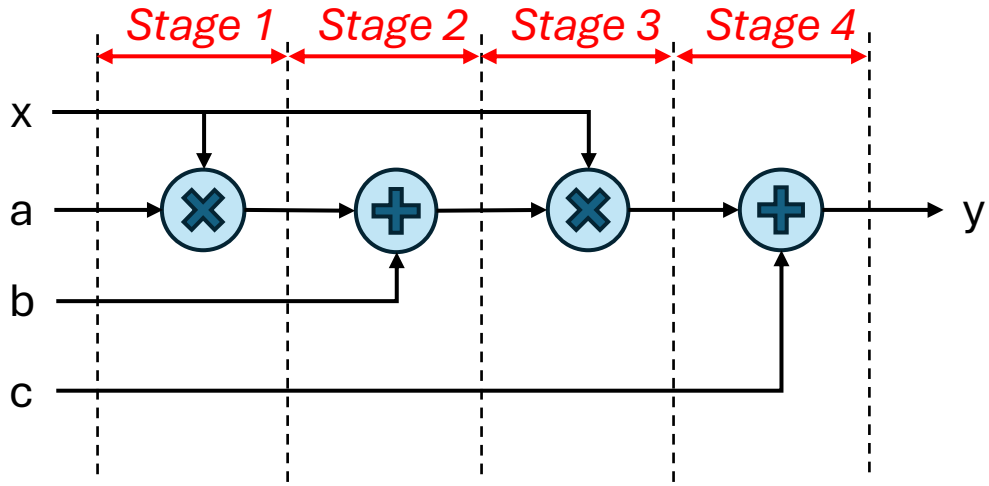
Example: Polynomial Circuit



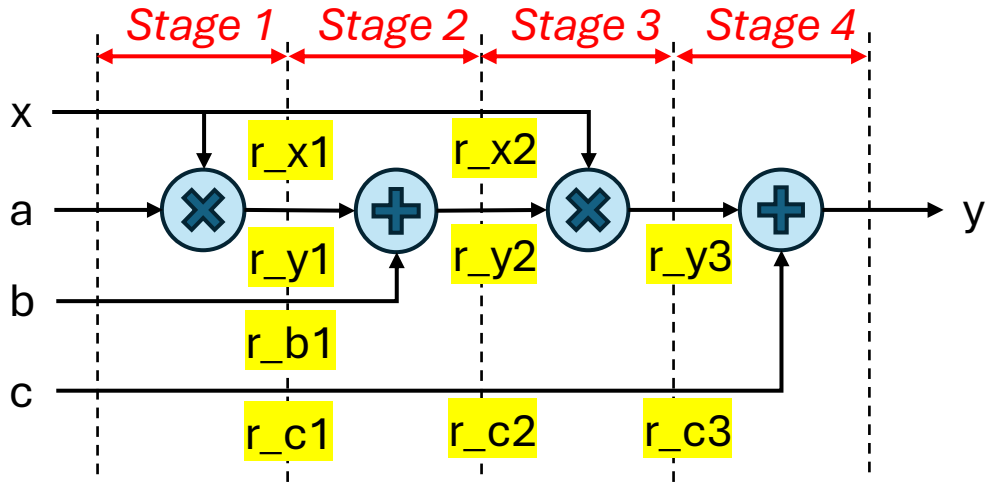
Clock Period = 6.9ns



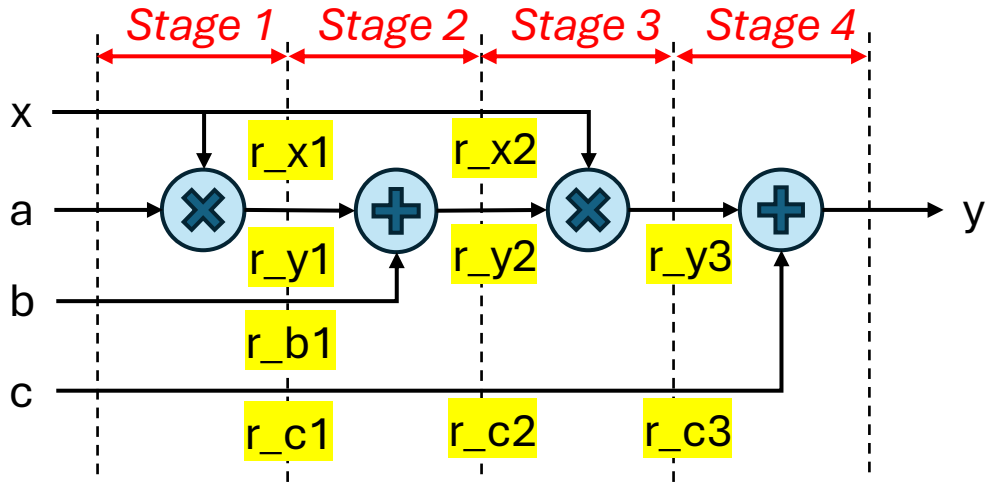
Example: Polynomial Circuit



Example: Polynomial Circuit



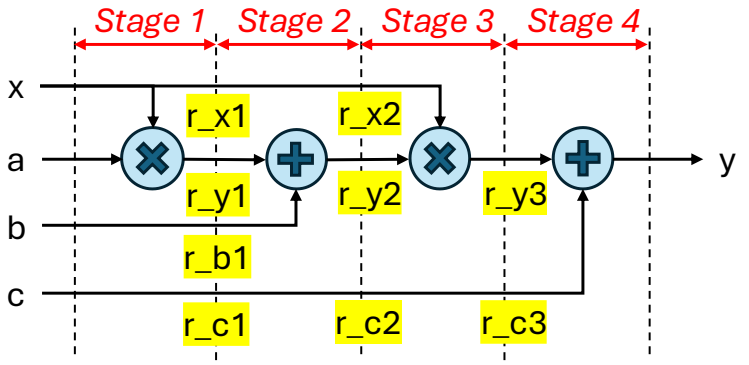
Example: Polynomial Circuit



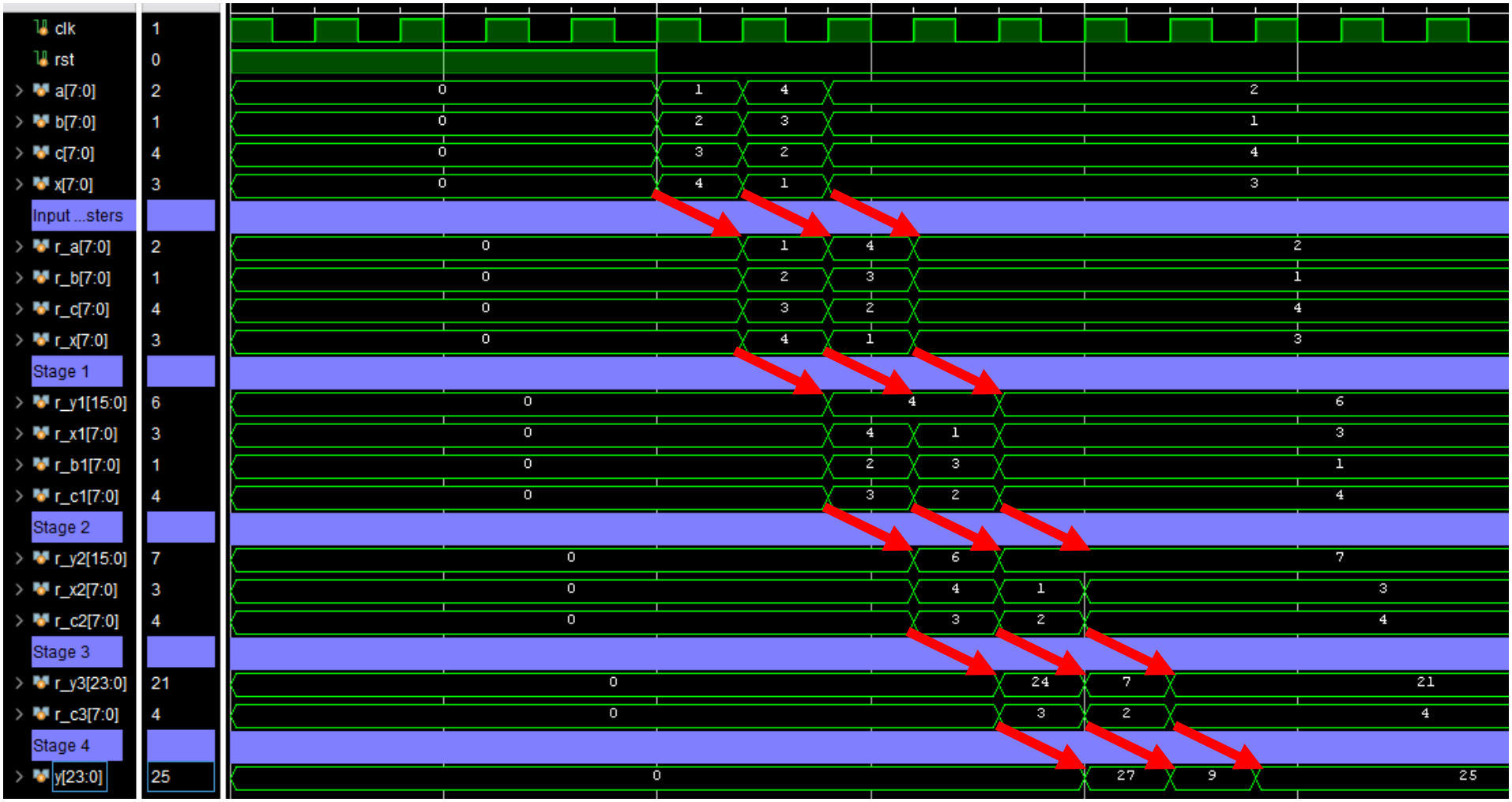

```
module poly_4stage (
      input clk,
      input rst,
      input [7:0] a,
      input [7:0] b,
      input [7:0] c,
      input [7:0] x,
      output logic [23:0] y
);

logic [7:0] r_a, r_b, r_c, r_x;
logic [15:0] r_y1;
logic [7:0] r_x1, r_b1, r_c1;
logic [15:0] r_y2;
logic [7:0] r_x2, r_c2;
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: Polynomial Circuit



Clock Period = 5.4ns

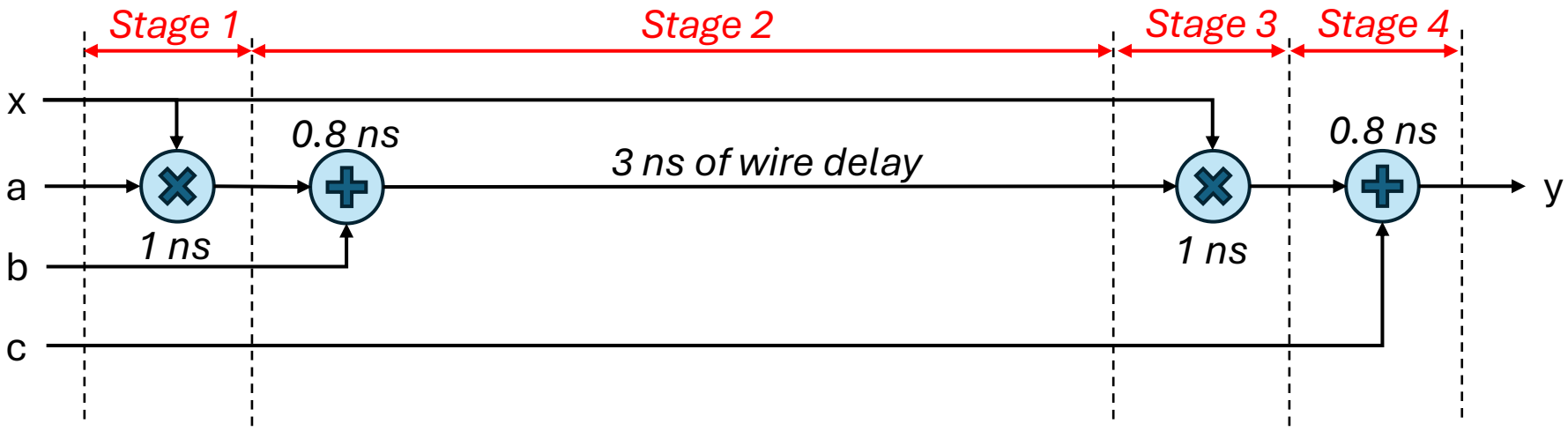


Pipelined Polynomial Circuit Analysis

1 Stage	2 Stages	4 Stages
Clock Period = 11.4ns	Clock Period = 6.9ns	Clock Period = 5.4ns
Frequency = 88 MHz	Frequency = 145 MHz	Frequency = 185 MHz
No. of Ops = 4	No. of Ops = 4	No. of Ops = 4
Throughput = 352 MOPS	Throughput = 580 MOPS	Throughput = 740 MOPS
Latency = 2 x 11.4ns = 22.8ns	Latency = 3 x 6.9ns = 20.7ns	Latency = 5 x 5.4ns = 27ns
Registers = (4x8)+24= 56	Registers = (6x8)+16+24= 88	Registers = (10x8)+(2x16)+(2x24)= 160

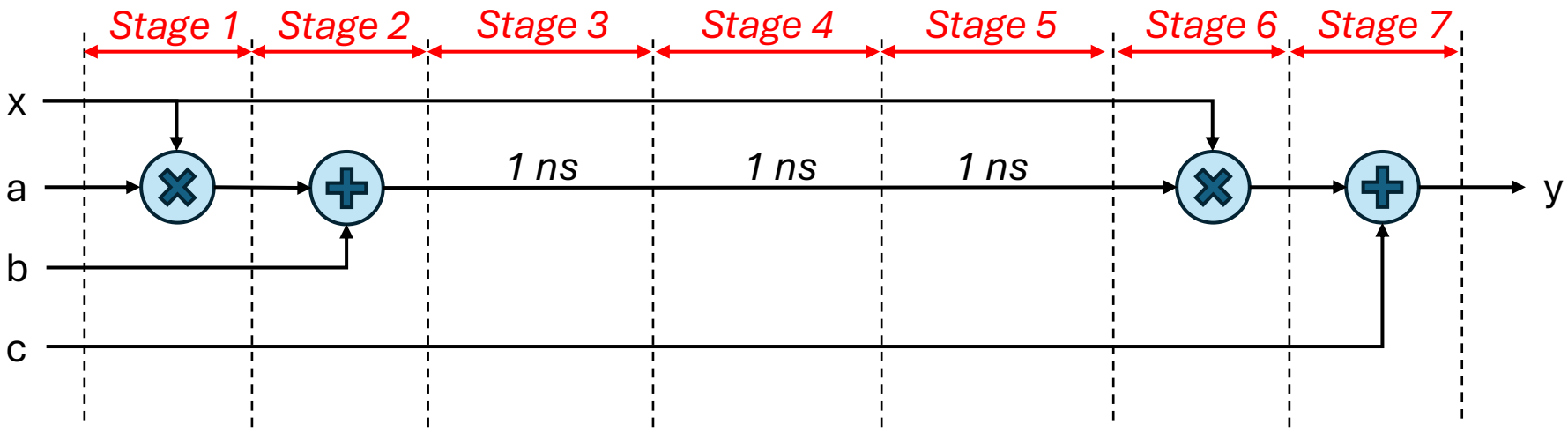
Wire Pipelining

- Remember that wires do not have zero delay
- In modern semiconductor process nodes, most of the delay is in the wires and not the logic



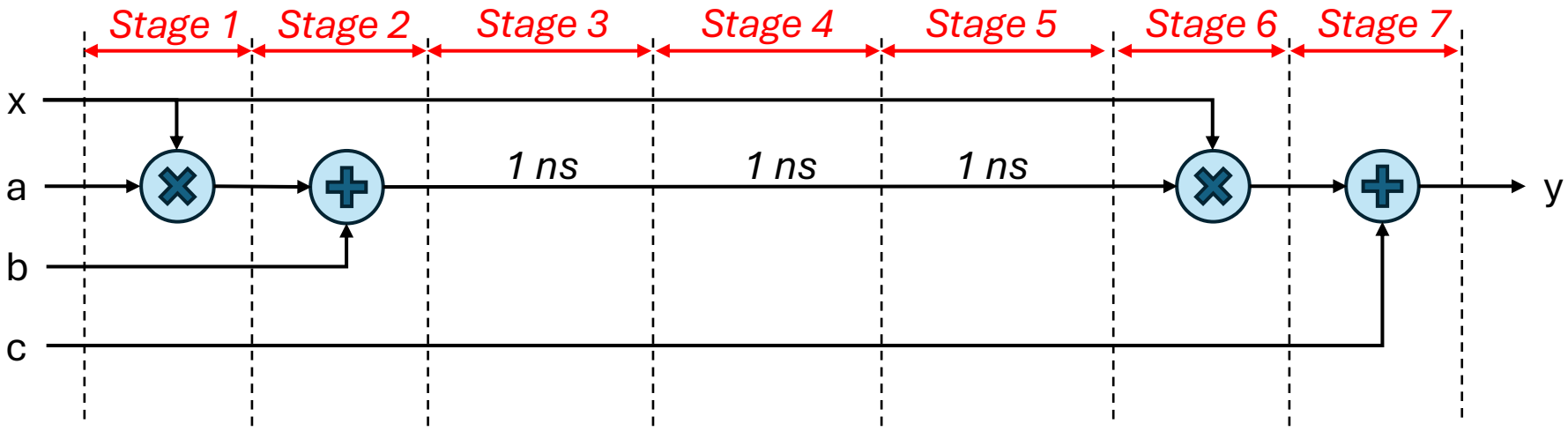
Wire Pipelining

- Remember that wires do not have zero delay
- In modern semiconductor process nodes, most of the delay is in the wires and not the logic
- Sometimes we need to pipeline wires to improve frequency



Wire Pipelining

- Remember that wires do not have zero delay
- In modern semiconductor process nodes, most of the delay is in the wires and not the logic
- Sometimes we need to pipeline wires to improve frequency



How do I know which wires to add pipeline registers to?

Wire Pipelining

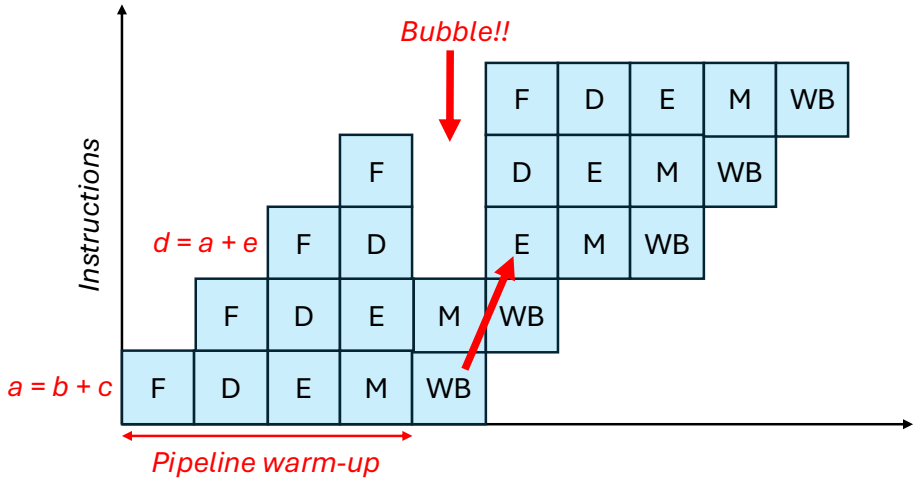
- To avoid frequent code changes when experimenting with different wire pipelining depths, we can implement a parameterizable wire pipelining “module”

- Instantiate between design modules and change `NUM_STAGES` parameter to experiment with different pipeline depths

```
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
```

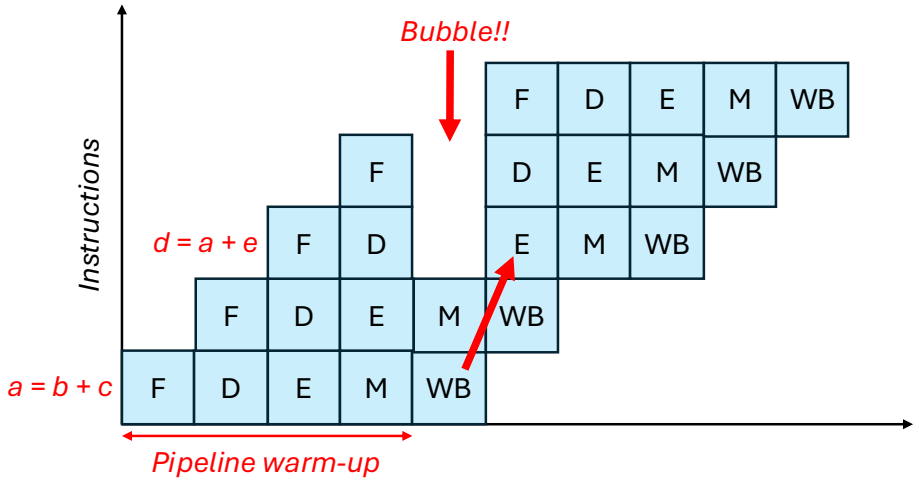
Pipeline Bubbles Revisited

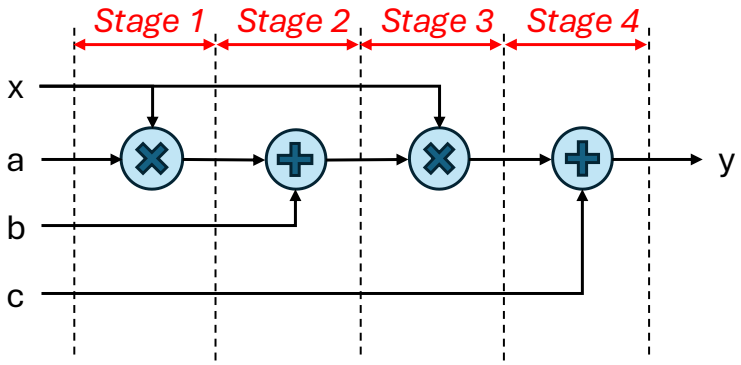
- In a CPU processing pipeline, bubbles are injected due to stalls caused by data dependencies



Pipeline Bubbles Revisited

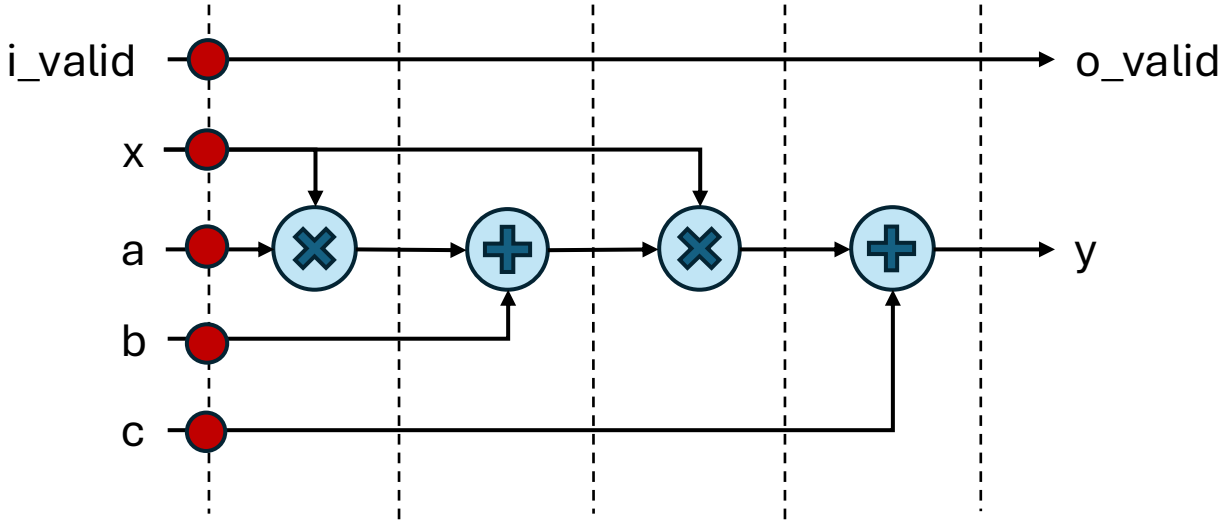
- In a CPU processing pipeline, bubbles are injected due to stalls caused by data dependencies
- In custom datapaths (e.g., our polynomial circuit), bubbles can be injected due to absence of input data
  - We implicitly assume that new inputs will arrive every cycle
  - This is not necessarily the case in many designs
  - Data arrival uncertainty is not known at design time





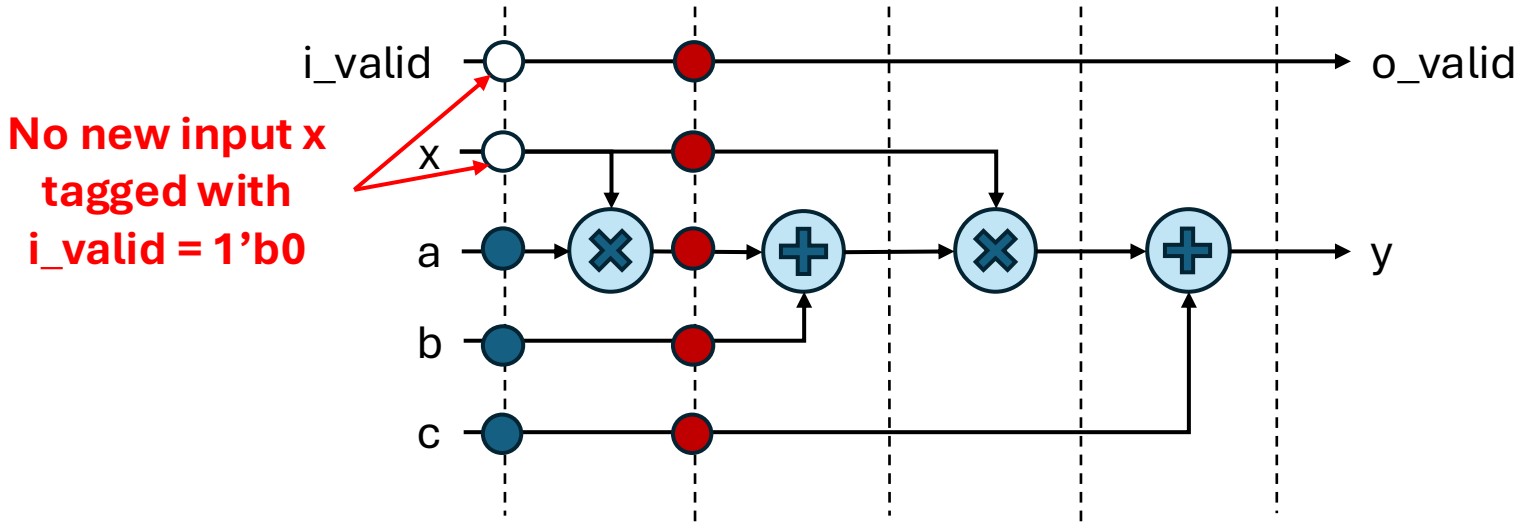
Handling Bubbles in Custom Datapaths

- When no new inputs, **invalid** data is injected in the pipeline
- We need to know the output corresponding to invalid inputs such that downstream modules can ignore it:
  - Add new input and output "valid" ports
  - Each input arrives coupled with a valid tag
  - Valid tag propagates through the circuit from the input valid port to the output valid port with a delay matching the datapath pipeline latency



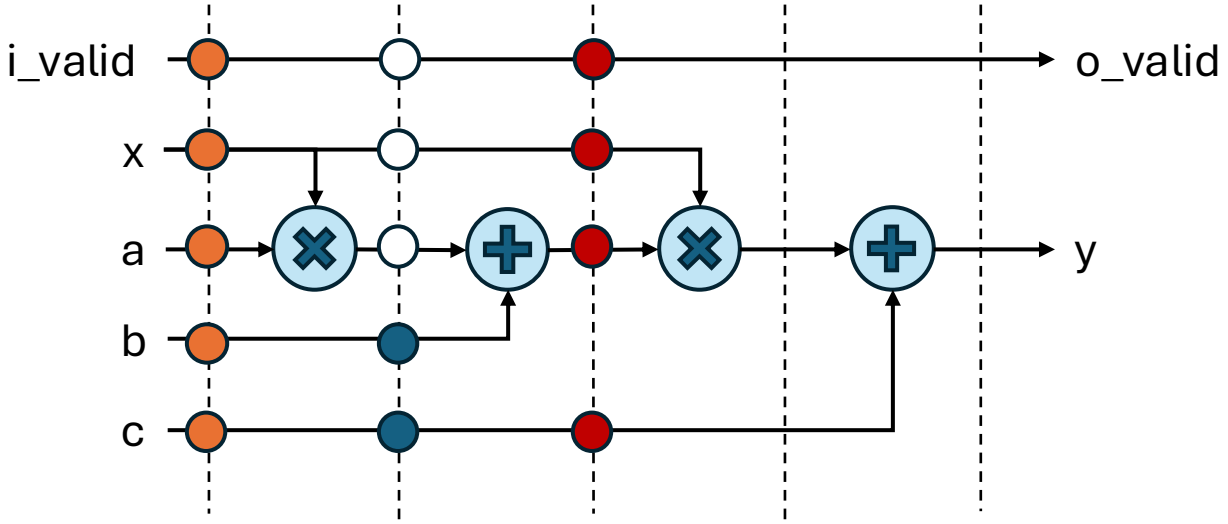
Handling Bubbles in Custom Datapaths

- When no new inputs, **invalid** data is injected in the pipeline
- We need to know the output corresponding to invalid inputs such that downstream modules can ignore it:
  - Add new input and output "valid" ports
  - Each input arrives coupled with a valid tag
  - Valid tag propagates through the circuit from the input valid port to the output valid port with a delay matching the datapath pipeline latency



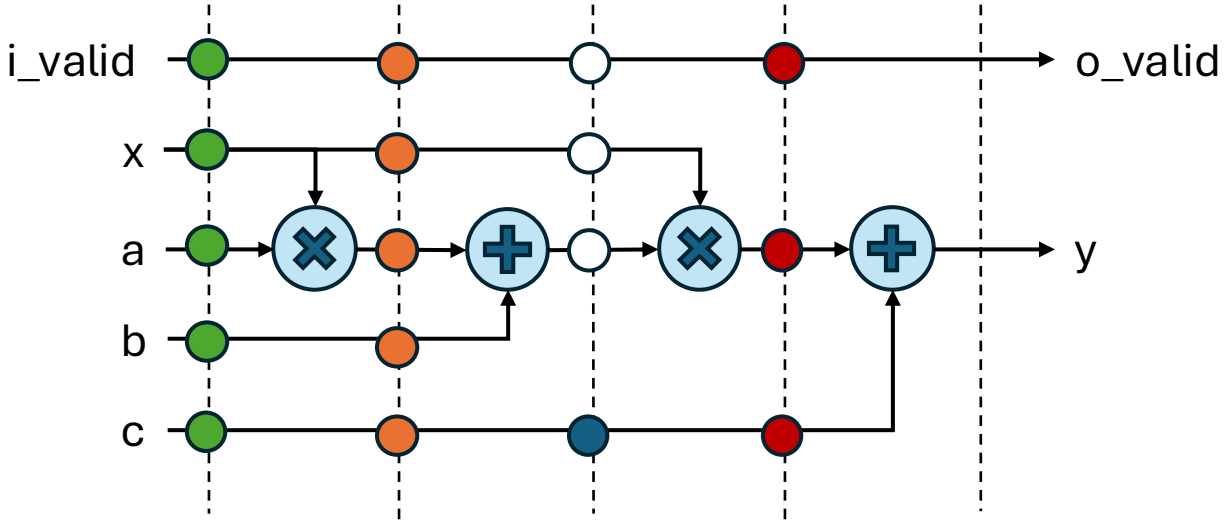
Handling Bubbles in Custom Datapaths

- When no new inputs, **invalid** data is injected in the pipeline
- We need to know the output corresponding to invalid inputs such that downstream modules can ignore it:
  - Add new input and output "valid" ports
  - Each input arrives coupled with a valid tag
  - Valid tag propagates through the circuit from the input valid port to the output valid port with a delay matching the datapath pipeline latency



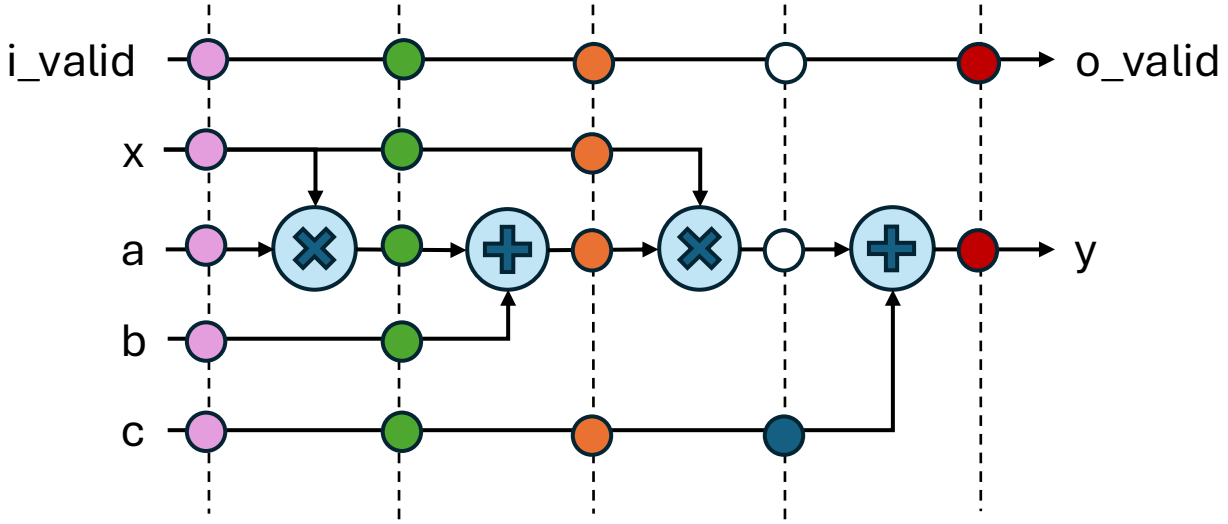
Handling Bubbles in Custom Datapaths

- When no new inputs, **invalid** data is injected in the pipeline
- We need to know the output corresponding to invalid inputs such that downstream modules can ignore it:
  - Add new input and output "valid" ports
  - Each input arrives coupled with a valid tag
  - Valid tag propagates through the circuit from the input valid port to the output valid port with a delay matching the datapath pipeline latency



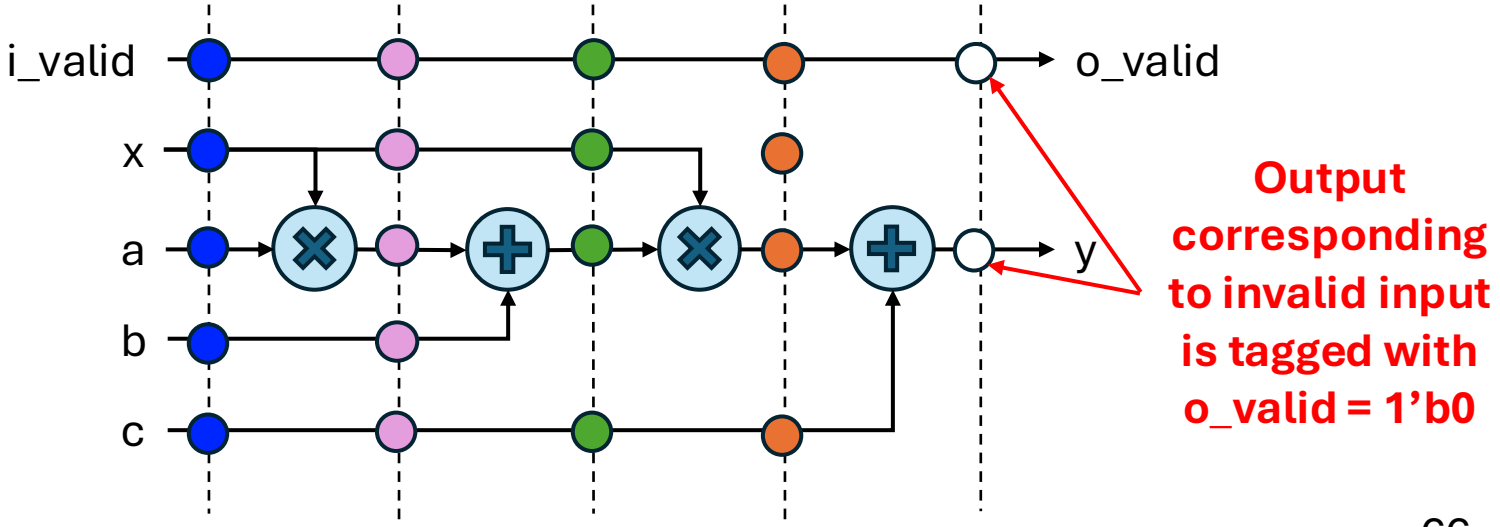
Handling Bubbles in Custom Datapaths

- When no new inputs, **invalid** data is injected in the pipeline
- We need to know the output corresponding to invalid inputs such that downstream modules can ignore it:
  - Add new input and output "valid" ports
  - Each input arrives coupled with a valid tag
  - Valid tag propagates through the circuit from the input valid port to the output valid port with a delay matching the datapath pipeline latency



Handling Bubbles in Custom Datapaths

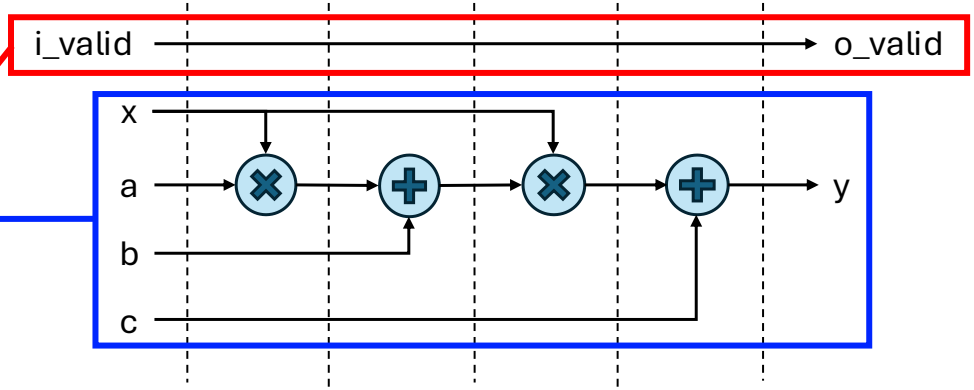
- When no new inputs, **invalid** data is injected in the pipeline
- We need to know the output corresponding to invalid inputs such that downstream modules can ignore it:
  - Add new input and output "valid" ports
  - Each input arrives coupled with a valid tag
  - Valid tag propagates through the circuit from the input valid port to the output valid port with a delay matching the datapath pipeline latency



Handling Bubbles in Custom Datapaths

```
module poly_4stage_bubbles (
      input  clk,
      input  rst,
      input  [7:0] a,
      input  [7:0] b,
      input  [7:0] c,
      input  [7:0] x,
      input  i_valid,
      output [23:0] y,
      output o_valid
);

poly_4stage poly_inst (
      .clk(clk),
      .rst(rst),
      .a(a),
      .b(b),
      .c(c),
      .x(x),
      .y(y)
);

wire_pipeline # (
      .DATAW(1),
      .NUM_STAGES(5)
) valid_pipeline (
      .clk(clk),
      .rst(rst),
      .data_in(i_valid),
      .data_out(o_valid)
);

endmodule
```

This is just a wire pipeline with 1-bit data and 5 pipeline registers on the path



This is the same 4-stage polynomial circuit we implemented before

Handling Bubbles in Custom Datapaths

Next Lecture

Can we pipeline any circuit?!