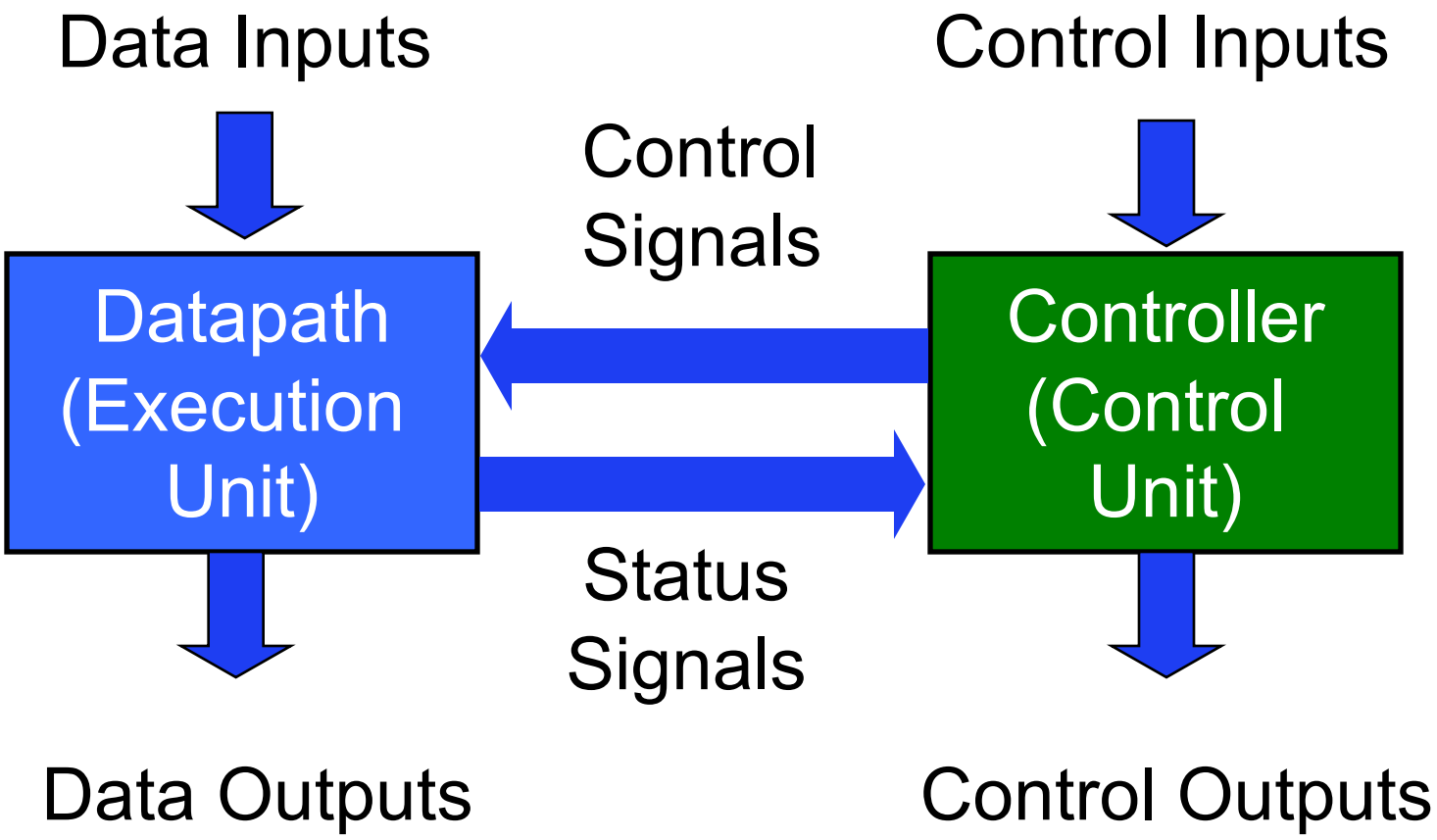
CDA 4253/CIS 6930 FPGA System Design

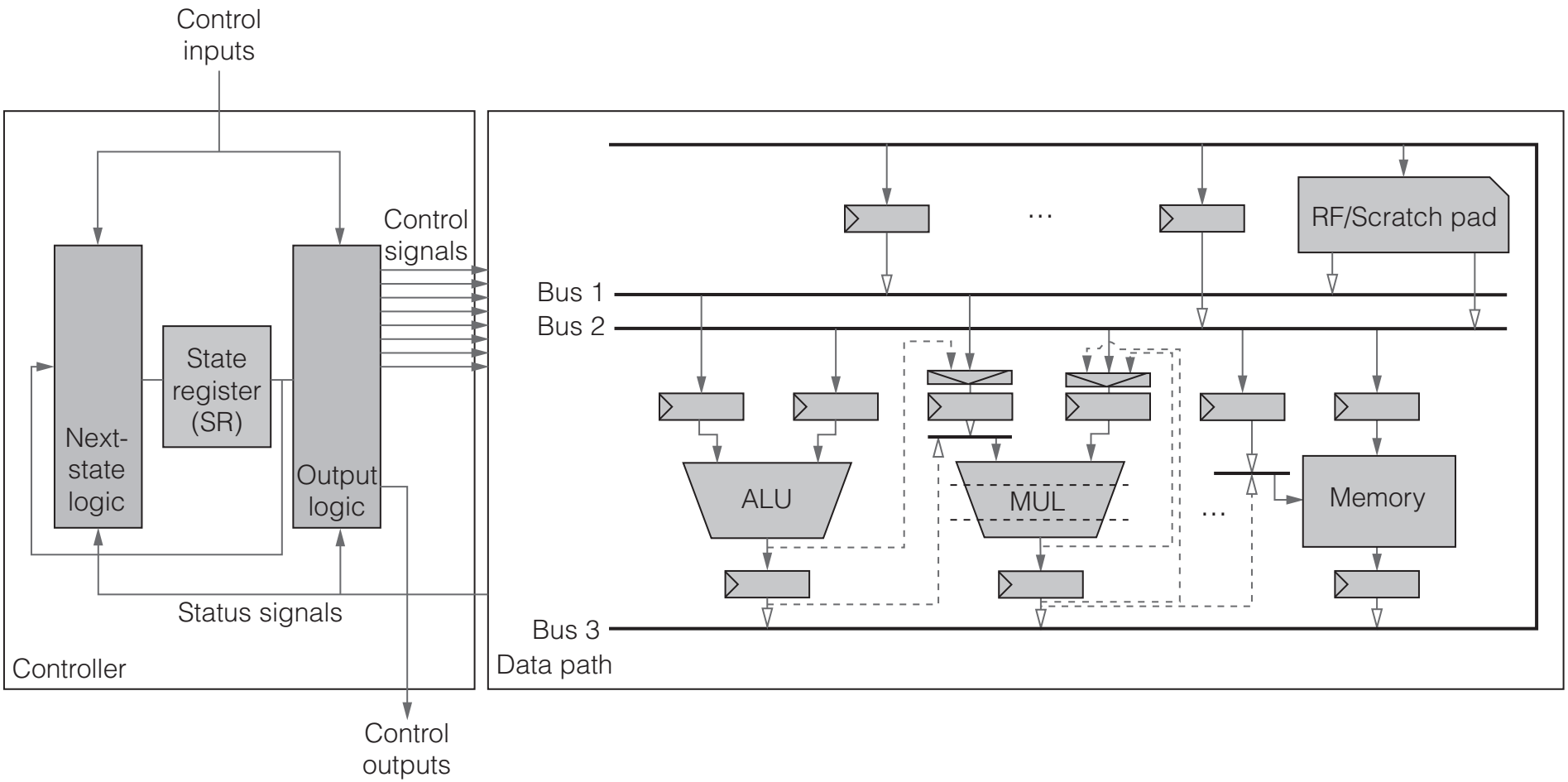
RTL Design Methodology

Hao Zheng
Comp S ci & Eng
Univ of South Florida

Structure of a Typical Digital Design



Hardware Design with RTL VHDL



Steps of the Design Process

1. Text description
2. Define interface
3. Describe the functionality using pseudo-code
4. Convert pseudo-code to FSM in state diagram
    1. Define states and state transitions
    2. Define datapath operations in each state.
5. Develop VHDL code to implement FSM
6. Develop testbench for simulation and debugging
7. Implementation and timing simulation
    - Timing simulation can reveal more bugs than pre-synthesis simulation
8. Test the implementation on FPGA boards

Min_Max_Average

Pseudocode

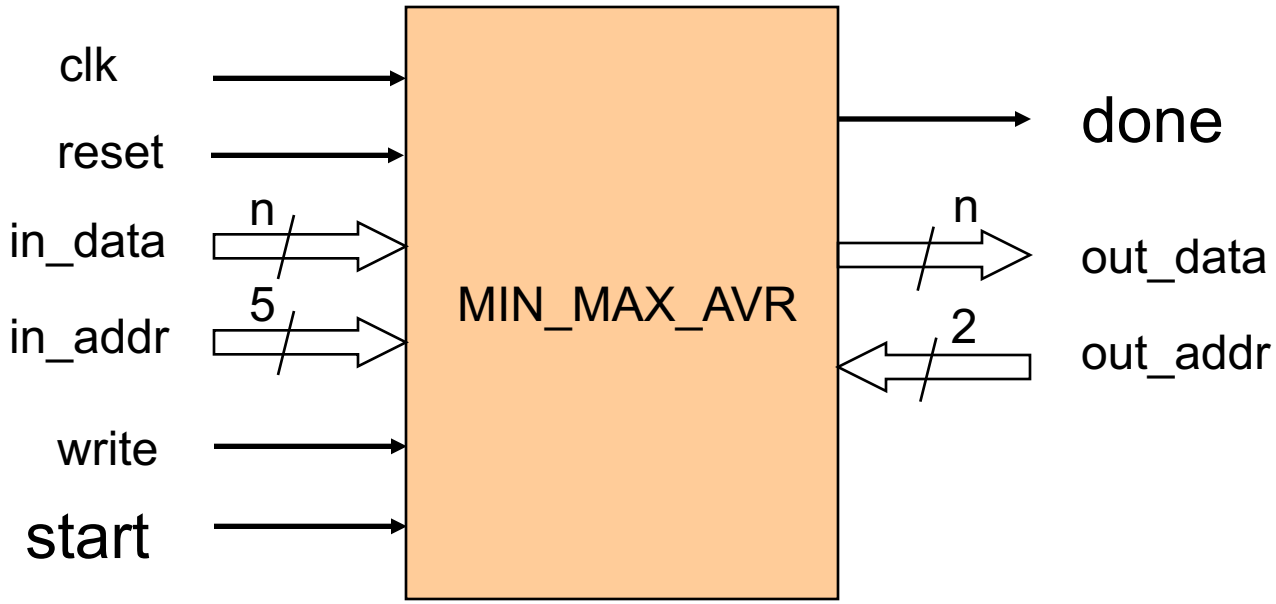
Data M[i] are stored in memory.

Results are stored in the internal registers.

Input: M[i]
Outputs: max, min, average

```
max = 0
min = MAX   // the maximal constant
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32
```

Circuit Interface



Interface Table

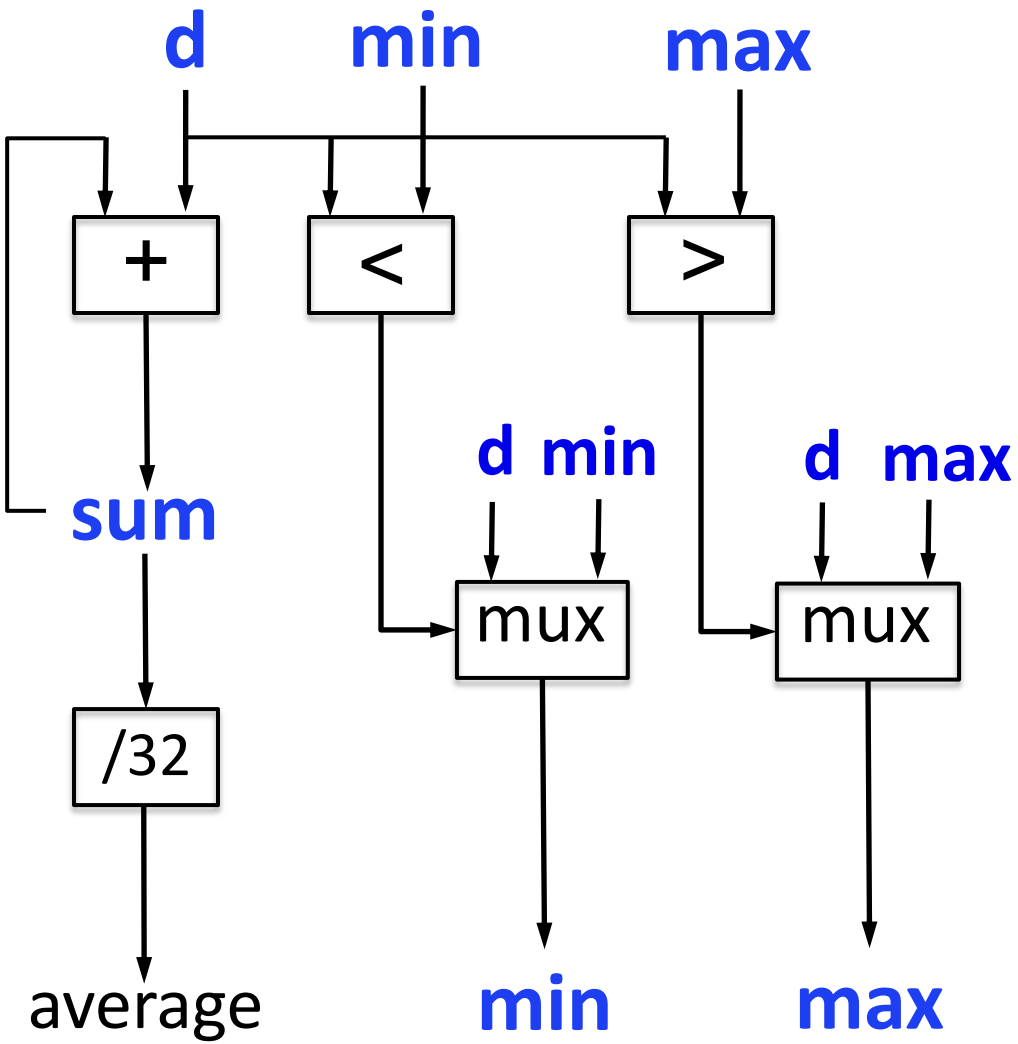
Port	Width	Meaning
clk	1	System clock
reset	1	System reset – clears internal registers
in_data	n	Input data bus
in_addr	5	Address of the internal memory where input data is stored
write	1	Synchronous write control signal – validity of in_data
start	1	Starts the computations
done	1	Asserted when all results are ready
out_data	n	Output data bus used to read results
out_addr	2	01 – reading minimum 10 – reading maximum 11 – reading average

Datapath

Input: M[i]
Output: max, min, average

```
max = 0
min = max
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32
```

Datapath



Input: M[i]
Output: max, min, average

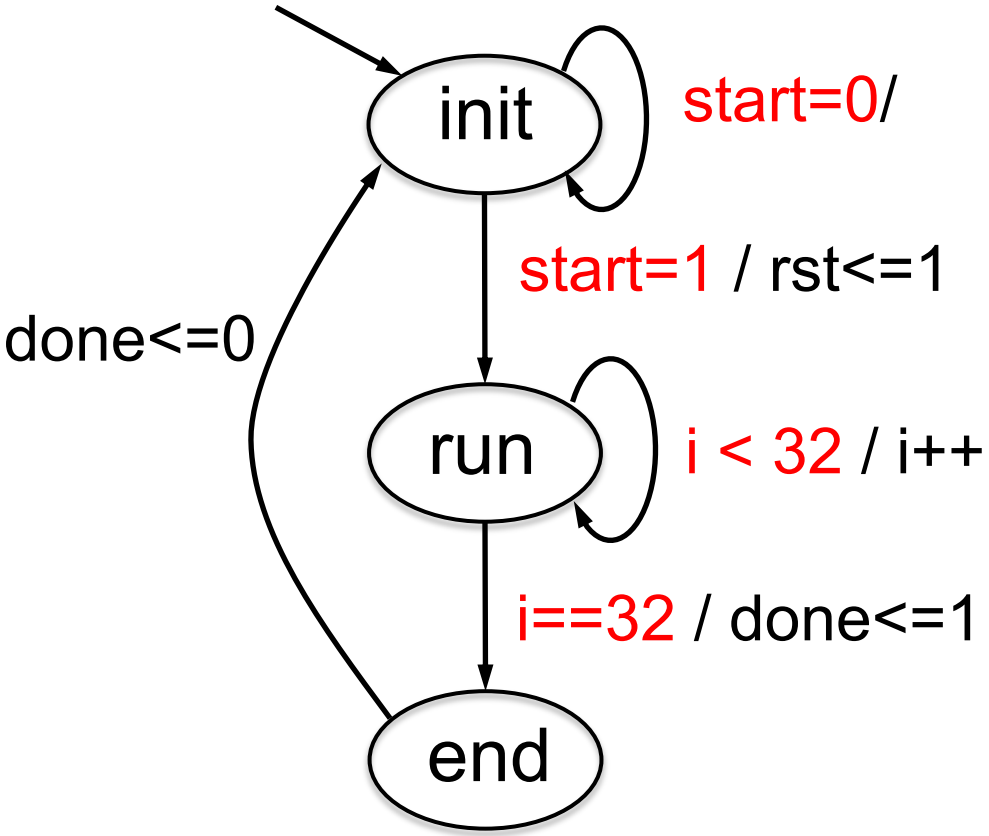
```
max = 0
min = max
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32
```

State Diagram for Controller

Input: M[i]
Outputs: max, min, average

```
max = 0
min = MAX
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32
```

State Diagram for Controller



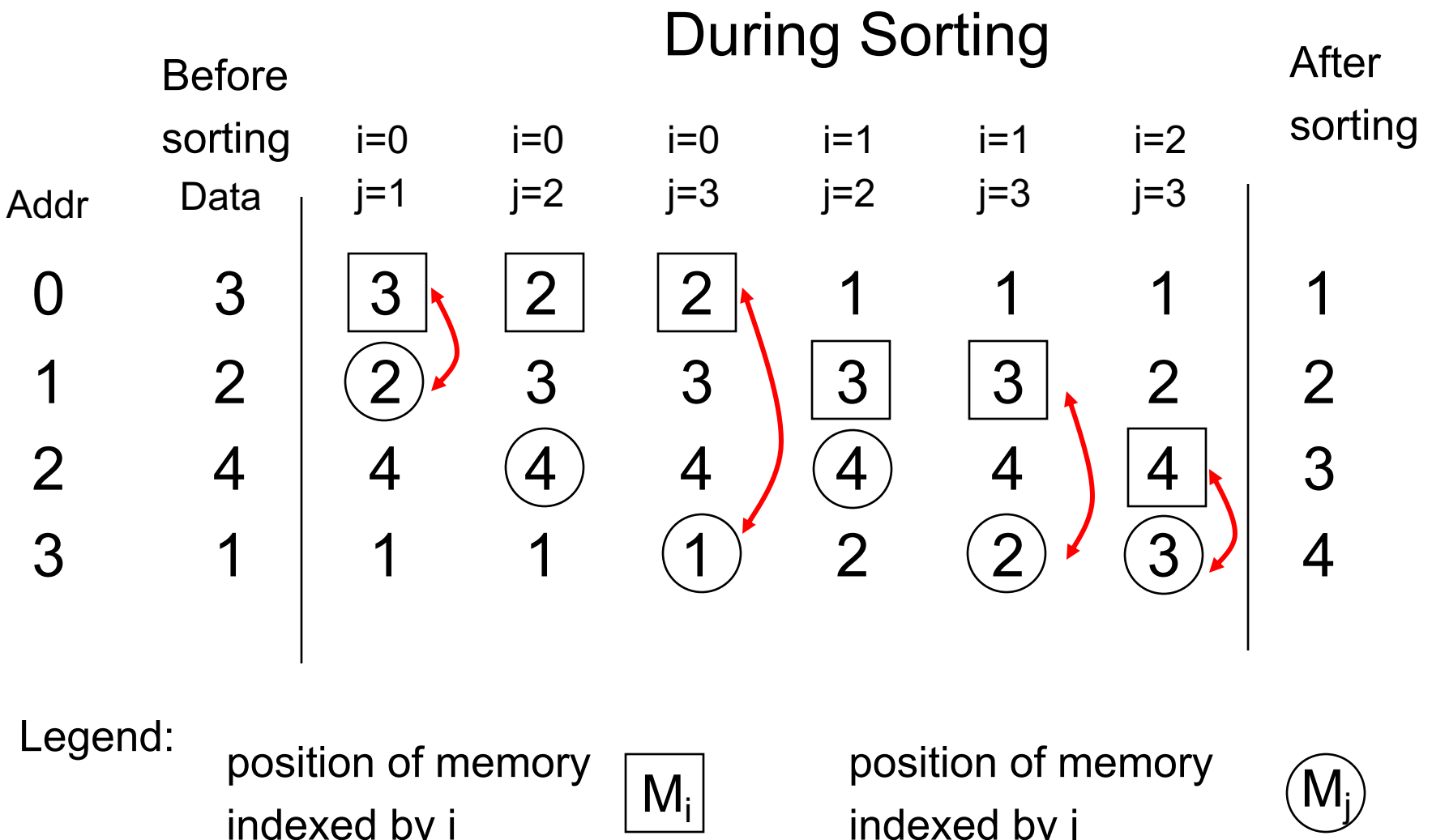
Output logic: in_addr <= i;
out_data <= ...

Input: M[i]
Outputs: max, min, average

```
max = 0
min = MAX
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32
```

Sorting

Sorting - Example



Pseudocode

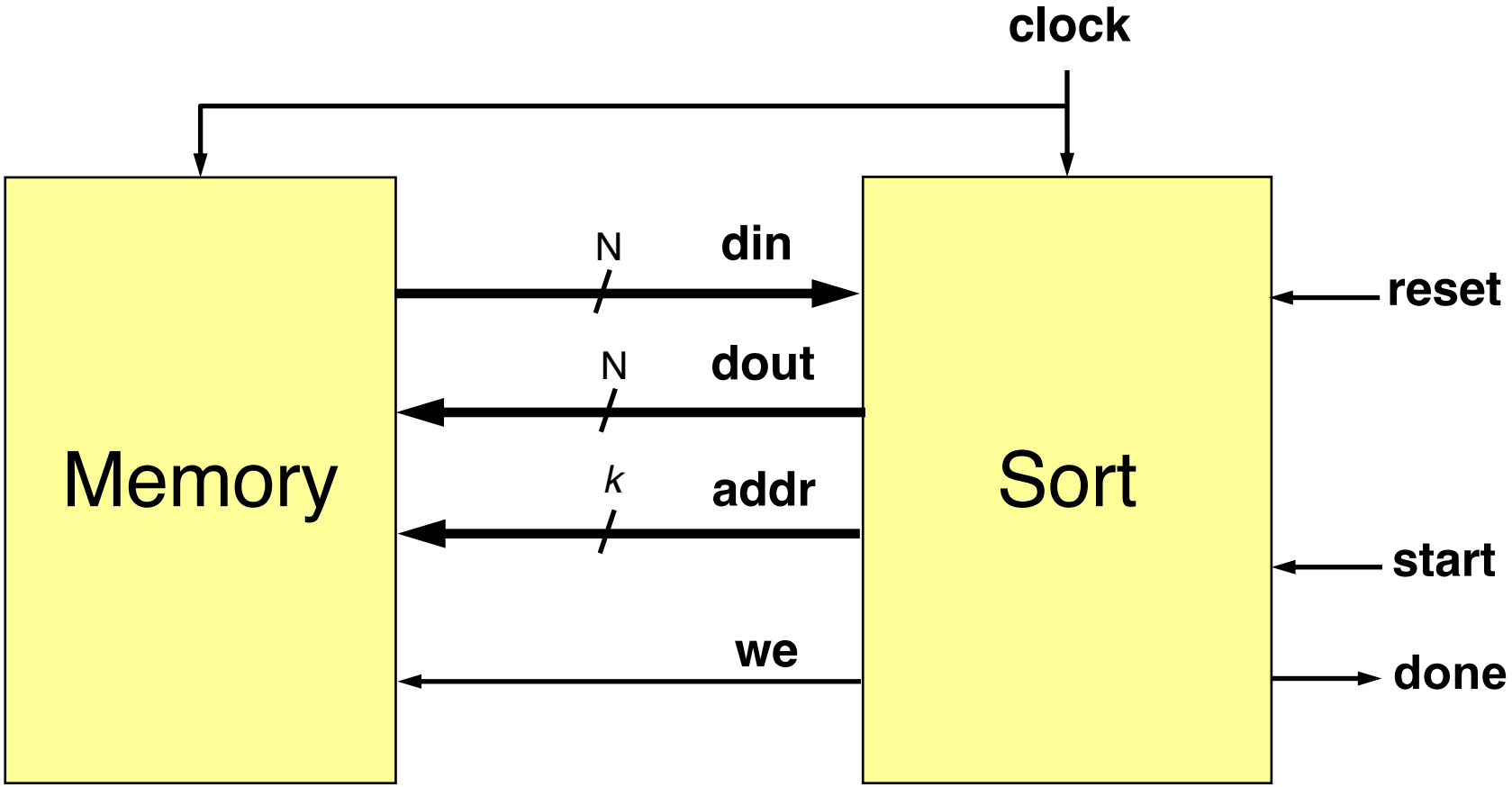
```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

K is a constant, the number of integers to be sorted in memory

M denotes memory.

Memory address is either *i* or *j*.

Sorting – Interface

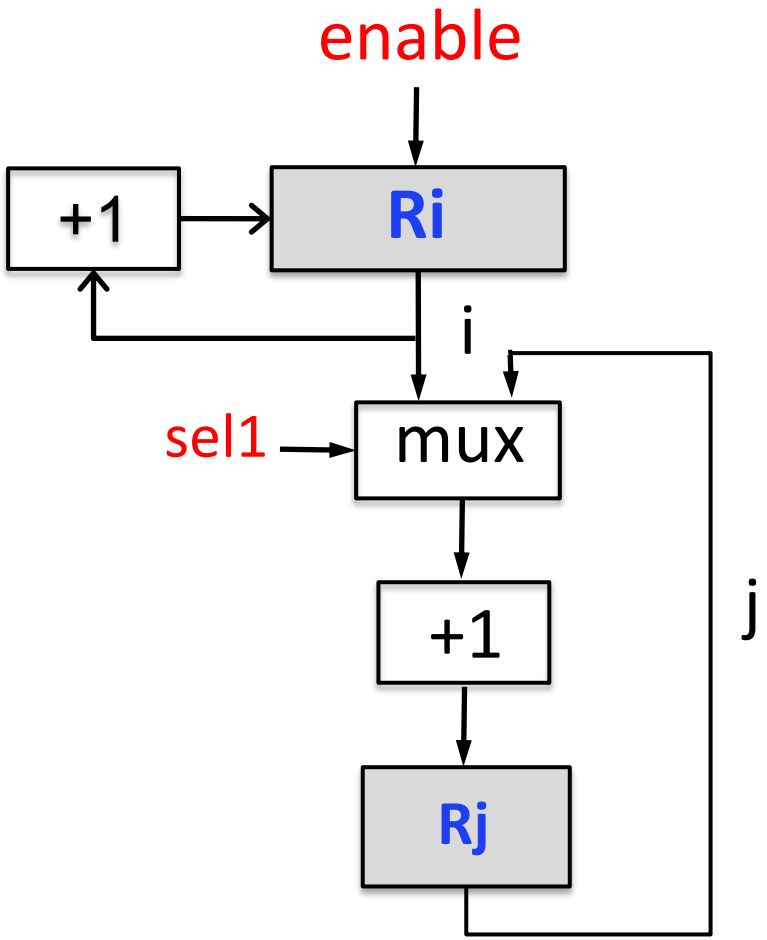


Sorting – Datapath

- Registers to hold A, B,
- Memory addresses i and j
- Incrementor
- Comparator

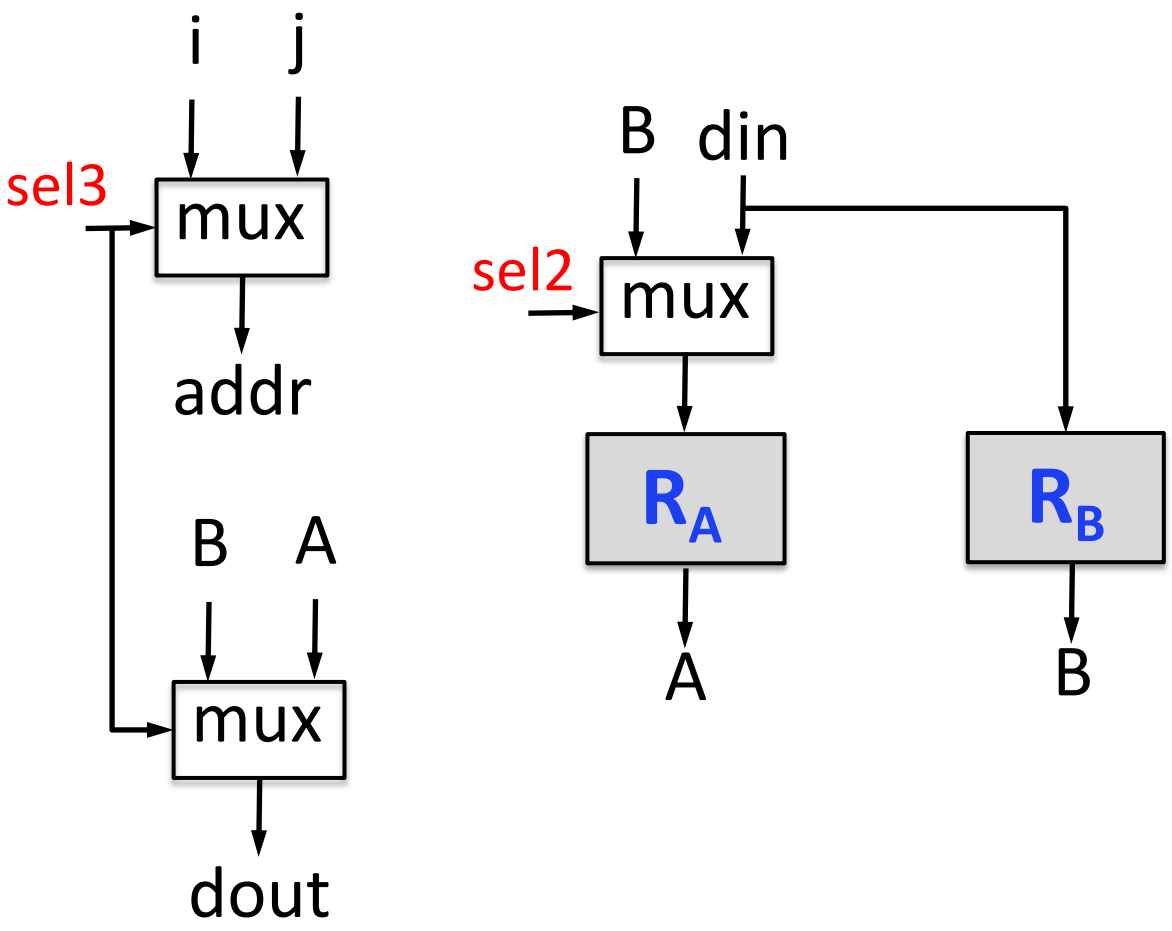
```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

Sorting – Datapath



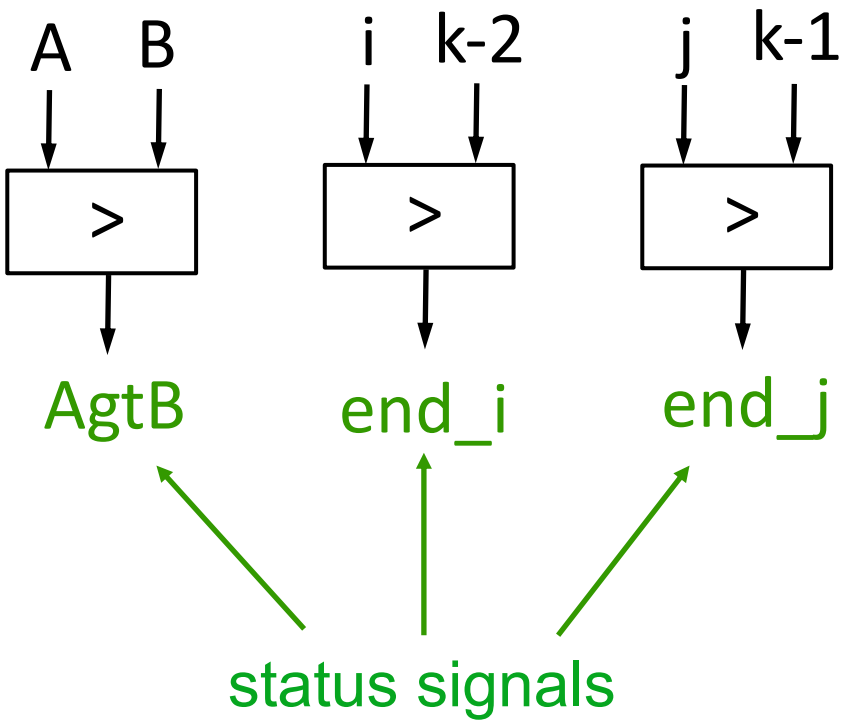

```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

Sorting – Datapath




```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

Sorting – Datapath



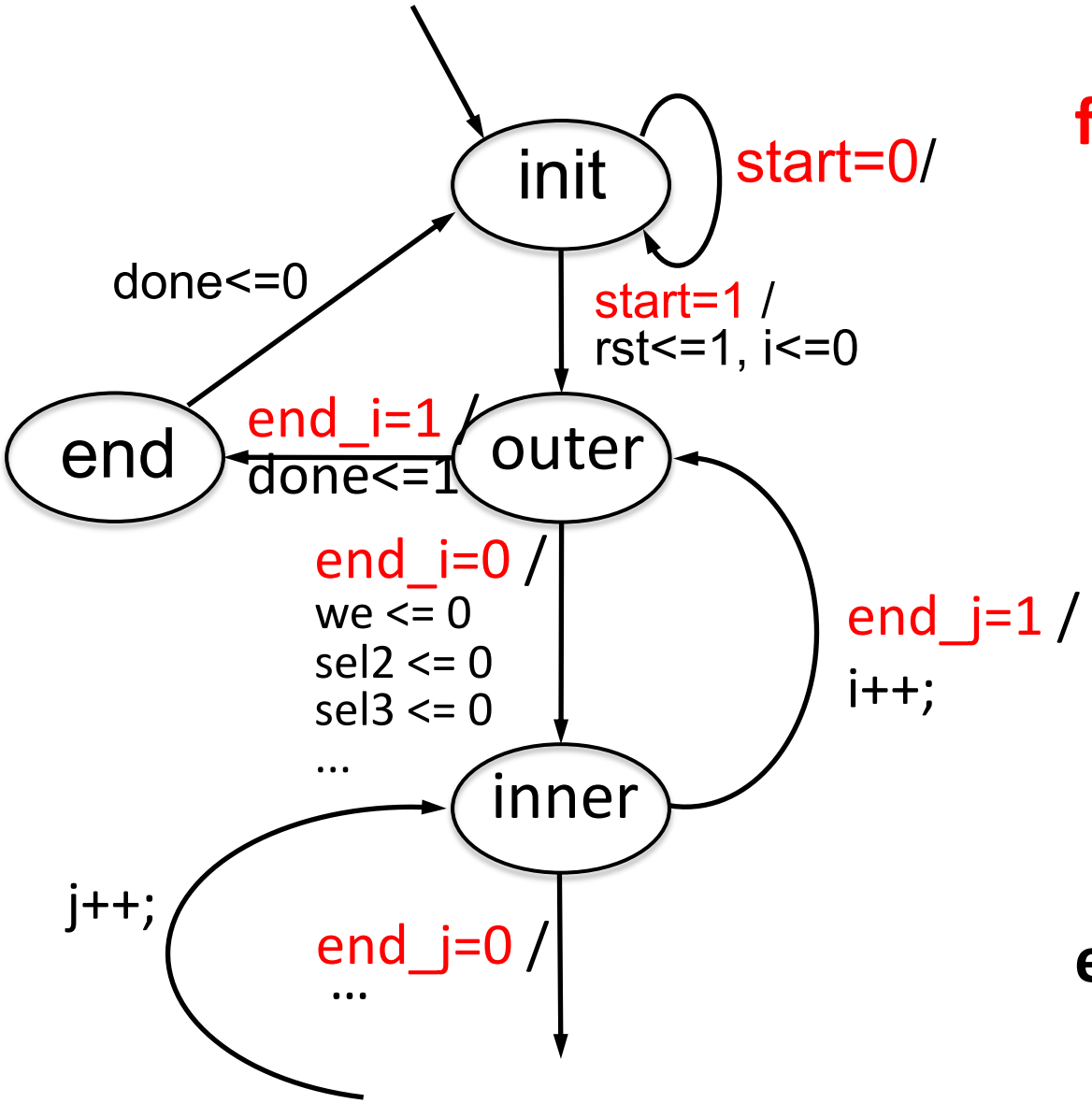

```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

Sorting – Controller

- Nested loops by two FSMs: one for the outer loop controls the one for the inner loop.
- Reuse the FSM for the single for loop in the previous example.

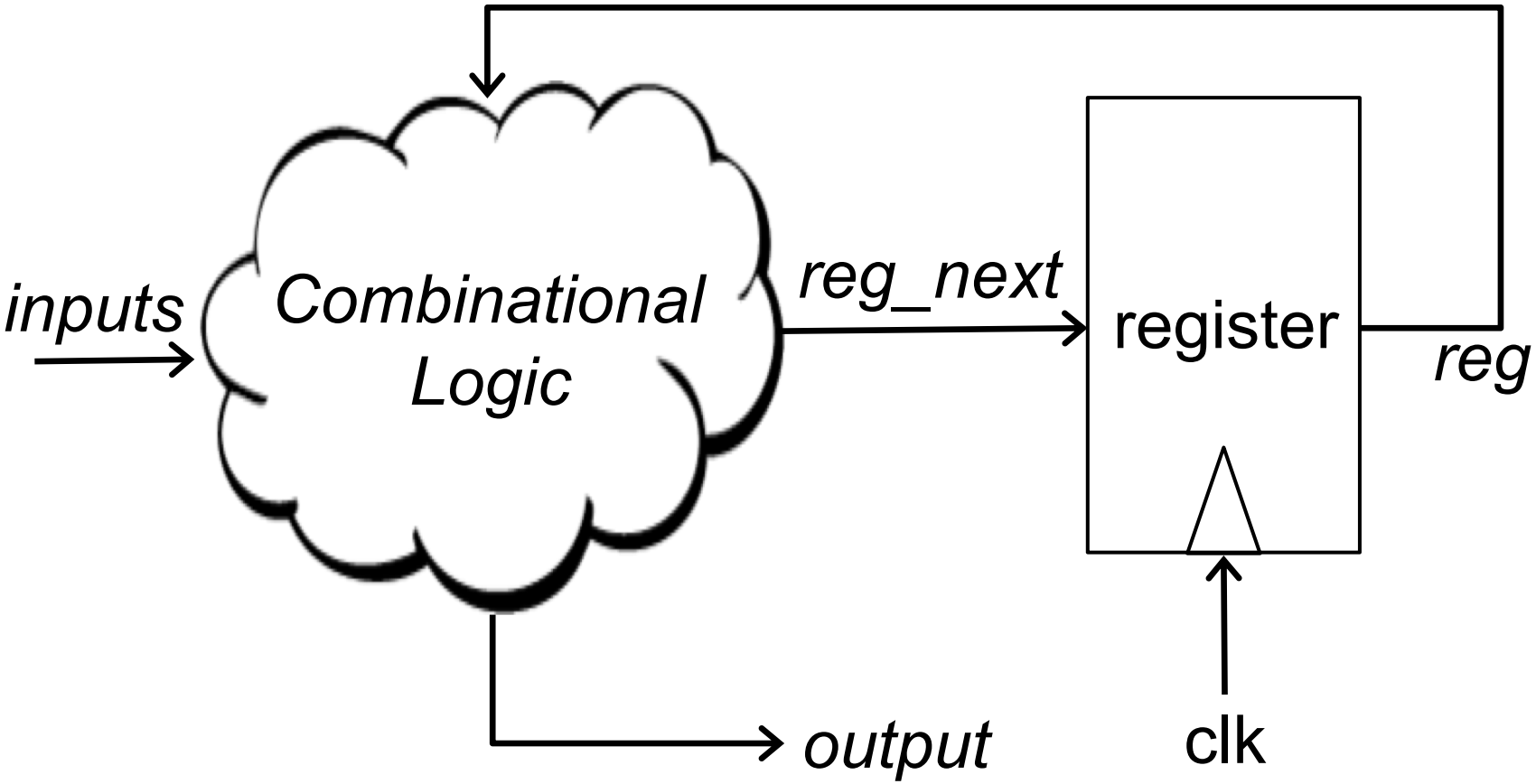
```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

Sorting – Controller




```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

Behavioral Level Design

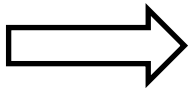


FSMD

```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```

FSMD

```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```



```
1   i = 0;
2   while i < k-1 do
3       addr = i
4       A = M[addr]
5       j=i+1
6           while j < k do
7           addr = j
8           B = M[addr]
9           if A > B then
10                  addr = i
11                  M[addr] = B
12                  addr = j
13                  M[addr] = A
14                  A = B
15          end if
16          j=j+1
17      end while
18      i = i+1;
19  end while
```

FSMD

```
1   i = 0;
2   while i < k-1 do
3       addr = i
4       A = M[addr]
5       j=i+1
6           while j < k do
7           addr = j
8           B = M[addr]
9           if A > B then
10                  addr = i
11                  M[addr] = B
12                  addr = j
13                  M[addr] = A
14                  A = B
15          end if
16          j=j+1
17      end while
18      i = i+1;
19  end while
```

FSMD

Current State	Next State	Cond	Operations
1	2	start='1'	i <= 0
2	3	i < k-1	null
2	18	!(i<k-1)	done <= '1'
3	6	true	addr <= i, A <= M[addr]; j <= j+1;
6	7	j < k	null
6	17	!(j<k)	null
7	10	true	j++; addr <= j; B <= M[addr];
10	16	A > B	addr <= i; M[addr] <= B;
10	16	!(A > B)	null
16	6	true	null
17	2	true	null
...	...	...	...

```
1   i = 0;
2   while i < k-1 do
3       addr = i
4       A = M[addr]
5       j = i+1;
6       while j < k do
7           j = j+1
8           addr = j
9           B = M[addr]
10          if A > B then
11              addr = i
12              M[addr] = B
13              addr = j
14              M[addr] = A
15              A = B
16          end if
17      end while
18  end while
```

FSMD

Current State	Next State	Cond	Operations
s0	s1	start='1'	i <= 0
s1	s2	i < k-1	addr <= i, A <= M[addr]; j <= i+1;
s1	s0	!(i<k-1)	done <= '1'
s2	s3	j < k	addr <= j; B <= M[addr];
s2	s1	!(j<k)	i <= i+1

s3	s2	A > B	addr <= i; M[addr] <= B;
			addr <= j; M[addr] <= A;
			A <= B; j <= j+1;

s3	s2	!(A > B)	j <= j+1;

```
1   i = 0;
2   while i < k-1 do
3       addr = i
4       A = M[addr]
5       j = i+1
6       while j < k do
7           addr = j
8           B = M[addr]
9           if A > B then
10              addr = i
11              M[addr] = B
12              addr = j
13              M[addr] = A
14              A = B
15          end if
16          j = j+1
17      end while
18      i = i + 1
19  end while
```

Optimization for Performance

Performance Definitions

- **Throughput**: the number of inputs processed per unit time.
- **Latency**: the amount of time for an input to be processed.
- Maximizing throughput and minimizing latency in conflict.
- Both require timing optimization:
  - Reduce delay of the critical path

Achieving High Throughput: Pipelining

- Divide data processing into stages
- Process different data inputs in different stages simultaneously.

```
xpower = 1;
for (i = 0; i < 3; i++)
   xpower = x * xpower;
```

Throughput: 1 data / 3 cycles = 0.33 data / cycle .

Latency: 3 cycles.

Critical path delay: 1 multiplier delay

```
process (clk) begin
   if rising_edge(clk) then
      if start='1' then
         cnt <= 3;
         done <= '0';
      elsif cnt > 0 then
         cnt <= cnt - 1;
         xpower <= xpower * x;
      elsif cnt = 0 then
         done <= '1';
      end if;
end process;
```

Achieving High Throughput: Pipelining

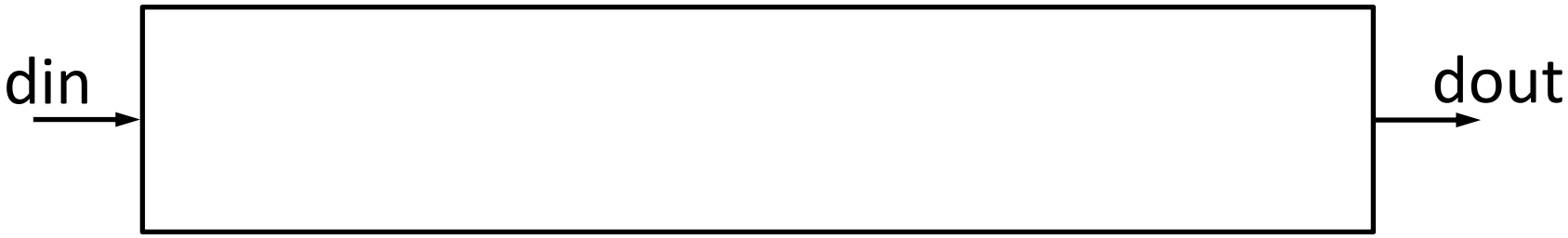
```
xpower = 1;
for (i = 0; i < 3; i++)
    xpower = x * xpower;
```

Throughput: 1 data / cycle
Latency: 3 cycles + register delays.
Critical path delay: 1 multiplier delay

```
process (clk, rst) begin
    if rising_edge(clk) then
        if start='1' then  -- stage 1
            x1 <= x;
            xpower1 <= x;
            done1 <= start;
        end if;
        -- stage 2
        x2 <= x1;
        xpower2 <= xpower1 * x1;
        done2 <= done1;
        -- stage 3
        xpower <= xpower2 * x2;
        done <= done2;
    end if;
end process;
```

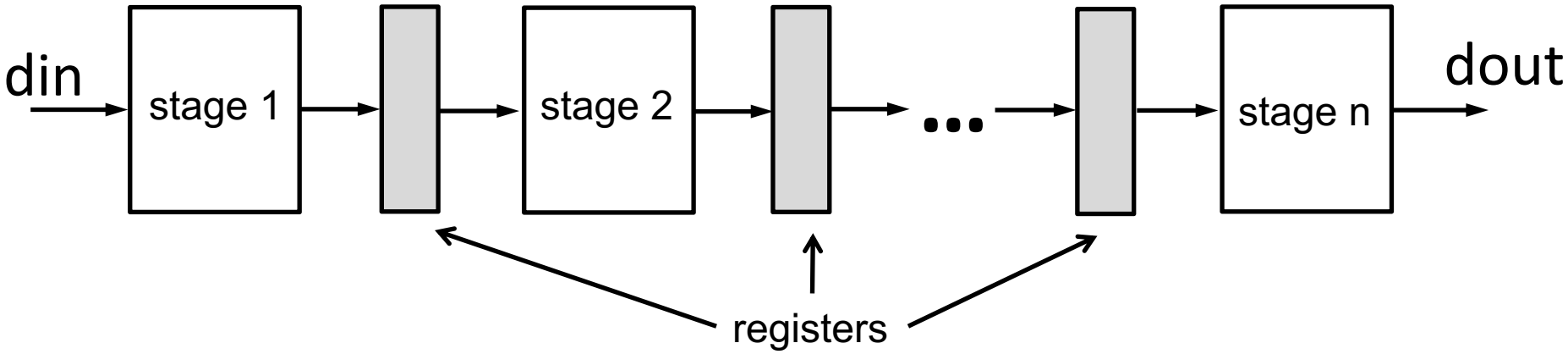
Achieving High Throughput: Pipelining

- Divide data processing into stages
- Process different data inputs in different stages simultaneously.



Achieving High Throughput: Pipelining

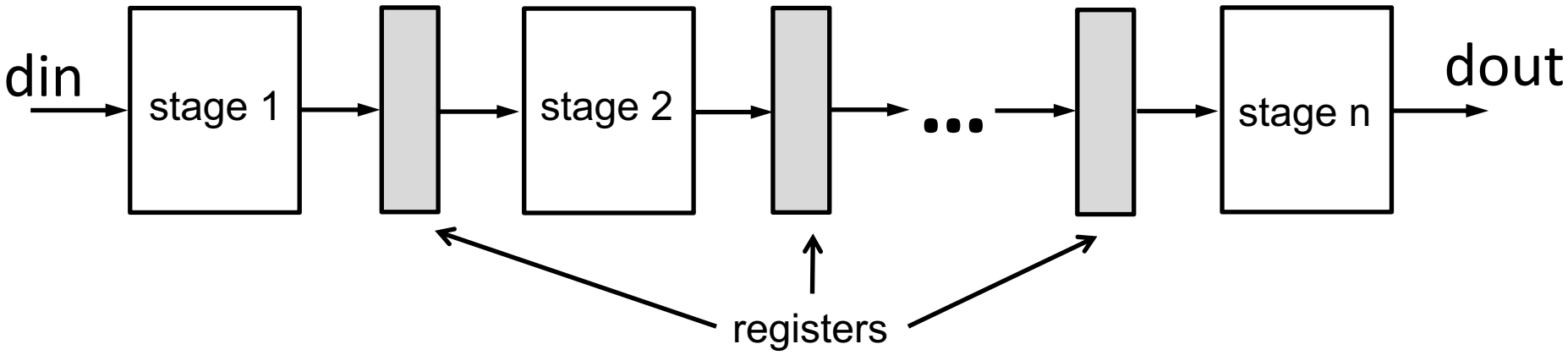
- Divide data processing into stages
- Process different data inputs in different stages simultaneously.



Penalty: increase in area as logic needs to be duplicated for different stages

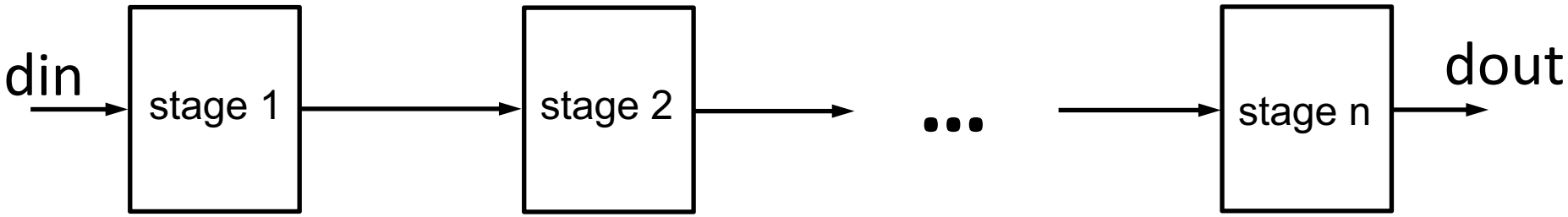
Reducing Latency

- Closely related to reducing critical path delay.
- Reducing pipeline registers reduces latency.



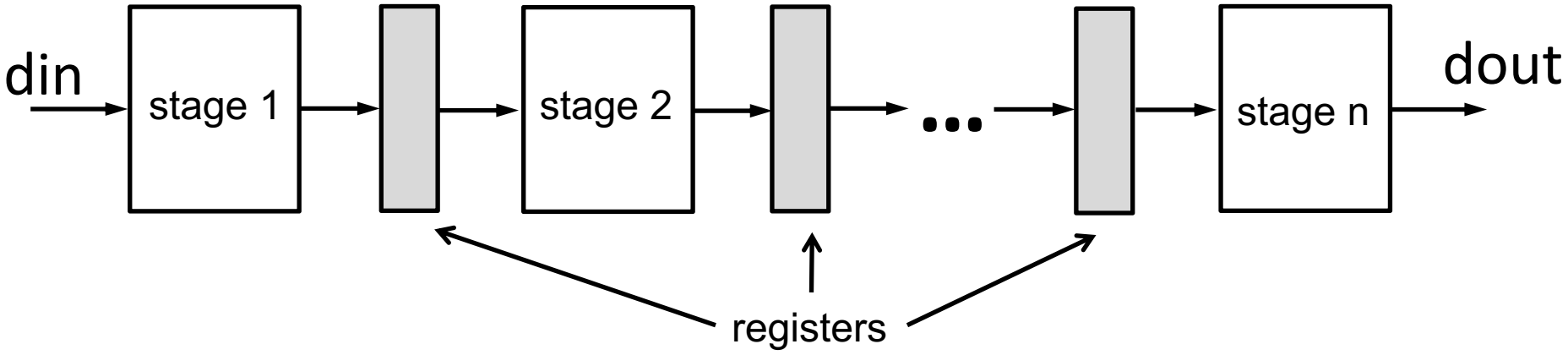
Reducing Latency

- Closely related to reducing critical path delay.
- Reducing pipeline registers reduces latency.



Timing Optimization

- Maximal clock frequency determined by the longest path delay in any combinational logic blocks.
- Pipelining is one approach.



Timing Optimization: Spatial Computing

- Extract independent operations
- Execute independent operations in parallel.

$$X = A + B + C + D$$

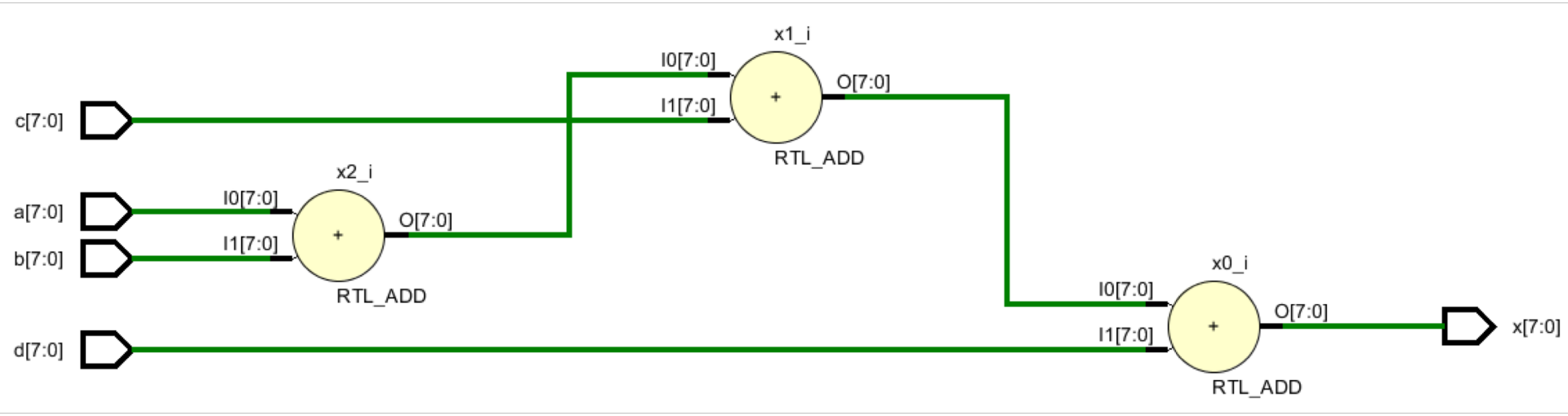
```
process (a, b, c, d) begin
        X1 := A + B;
        X2 := X1 + C;
        X  <= X2 + D;
end process;
```

```
process (a, b, c, d) begin
        X1 <= A + B;
        X2 <= C + D;
        X  <= X1 + X2;
end process;
```

Timing Optimization: Spatial Computing

X = A + B + C + D

```
process (a, b, c, d) begin
        X1 := A + B;
        X2 := X1 + C;
        X   <= X2 + D;
end process;
```

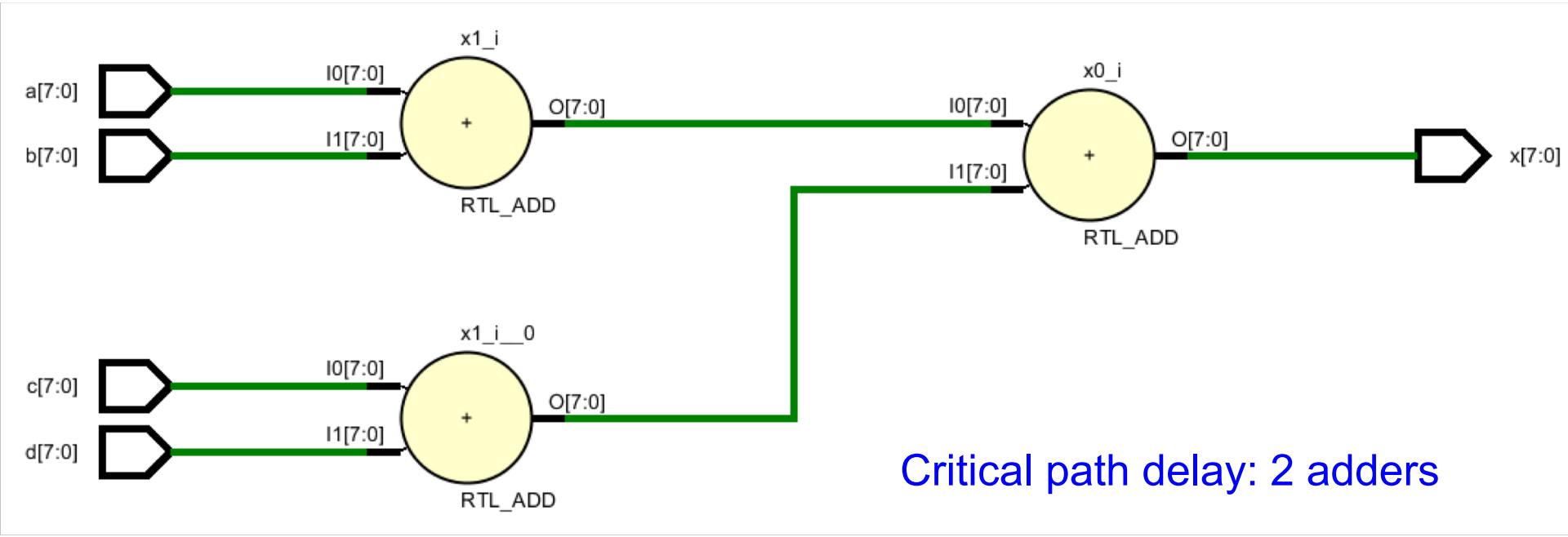


Critical path delay: 3 adders

Timing Optimization: Spatial Computing

X = A + B + C + D

```
process (a, b, c, d) begin
        X1 <= A + B;
        X2 <= C + D;
        X  <= X1 + X2;
end process;
```



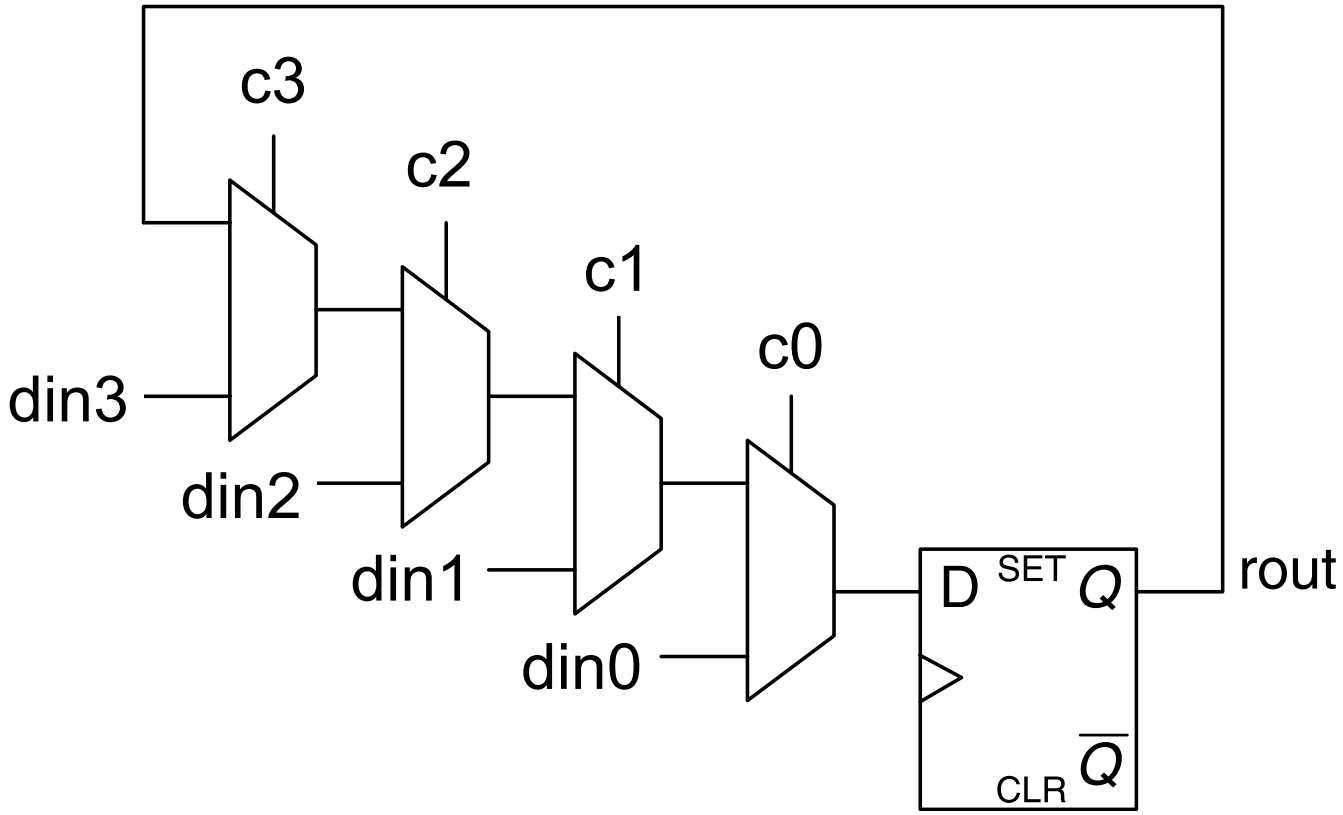
Critical path delay: 2 adders

Timing Optimization: Avoid Unwanted Priority

```
process (clk, rst) begin
    if rising_edge(clk) then
        if     c0='1' then rout <= din1;
        elsif  c1='1' then rout <= din2;
        elsif  c2='1' then rout <= din3;
        elsif  c3='1' then rout <= din4;
        end if;
    end if;
end process;
```

Critical path delay: 4 2x1MUX.

Timing Optimization: Avoid Unwanted Priority



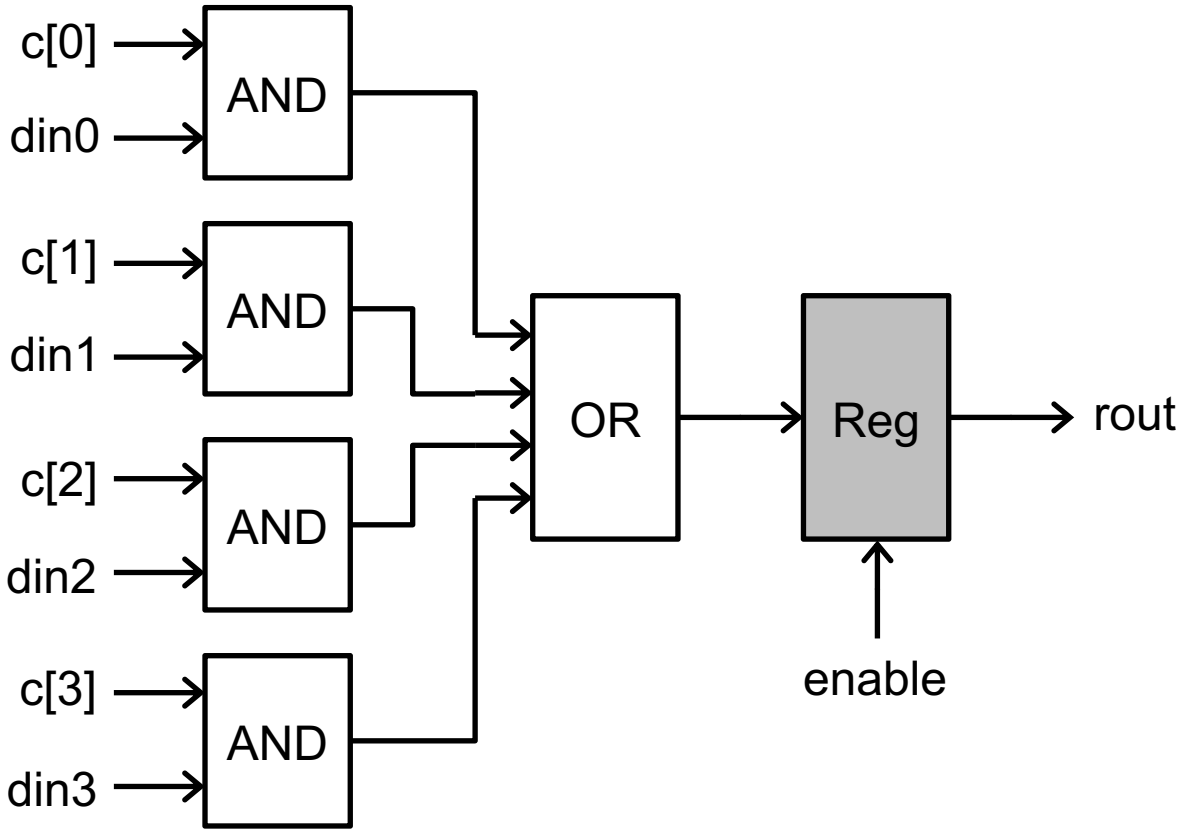
Critical path delay: 4 2x1 MUX.

cont'd from previous slide

Timing Optimization: Avoid Unwanted Priority

```
process (clk, rst) begin
     if rising_edge(clk) then
        case c is
          when “0001” =>
               rout <= din0;
          when “0010” =>
               rout <= din1;
          when “0100” =>
               rout <= din2;
          when “1000” =>
               rout <= din3;
          when others => null;
     end if;
end process;
```

Timing Optimization: Avoid Unwanted Priority

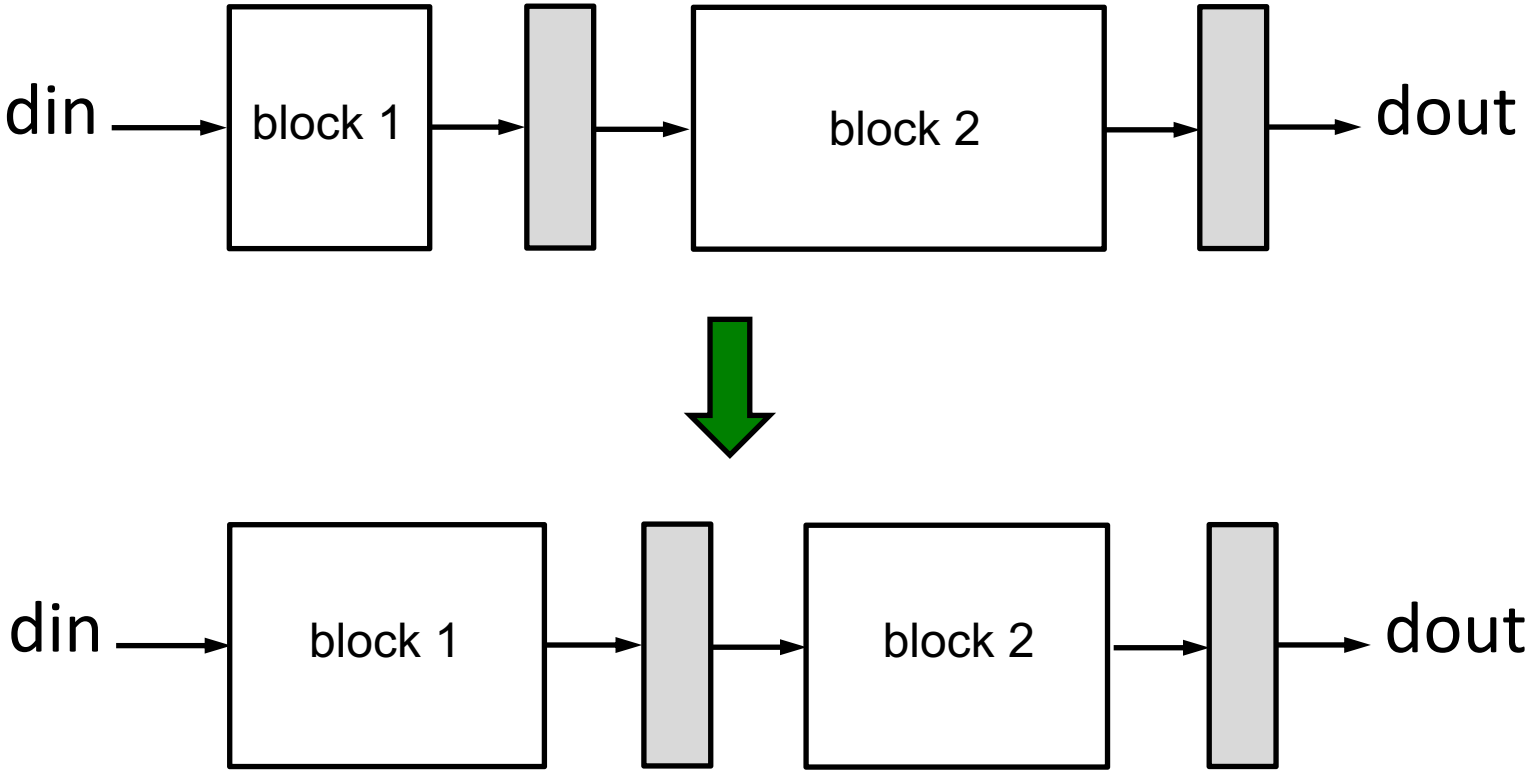


Critical path delay: 2-AND plus 4-OR.

cont'd from previous slide

Timing Optimization: Register Balancing

- Maximal clock frequency determined by the longest path delay in any combinational logic blocks.



Timing Optimization: Register Balancing

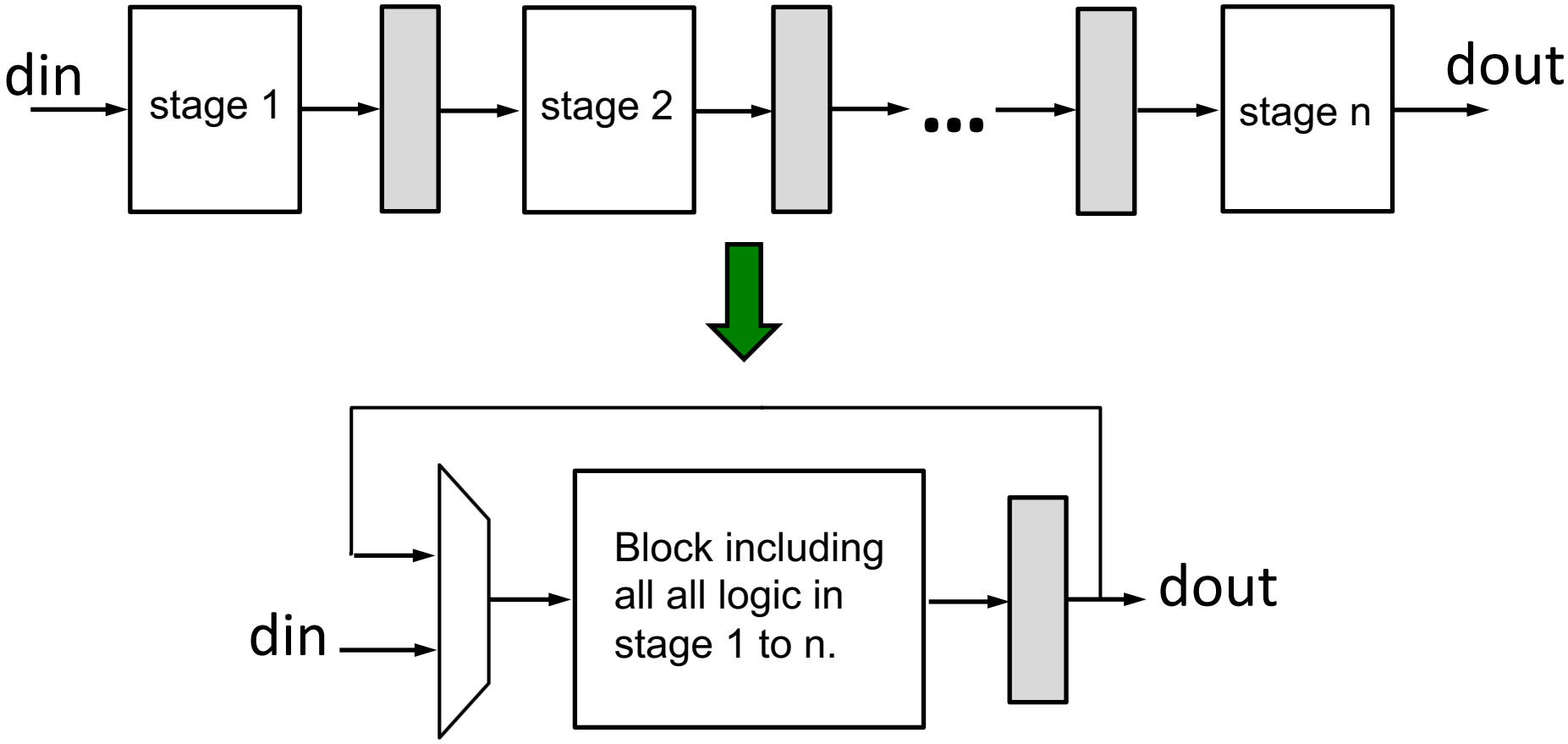
```
process (clk, rst) begin
  if rising_edge(clk) then
    rA <= A;
    rB <= B;
    rC <= C;
    sum <= rA + rB + rC;
  end if;
end process;
```

```
process (clk, rst) begin
  if rising_edge(clk) then
    sumAB <= A + B;
    rC <= C;
    sum <= sumAB + rC;
  end if;
end process;
```

Optimization for Area

Area Optimization: Resource Sharing

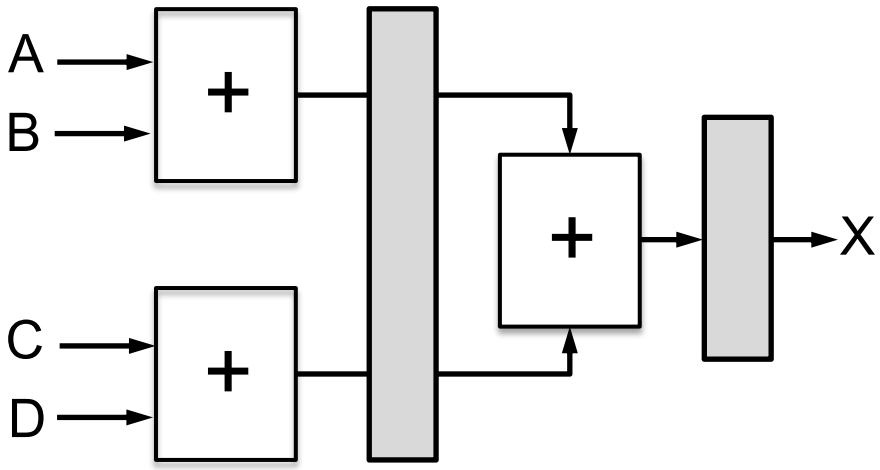
- Rolling up pipeline: share common resources at different time – a form of temporal computing



Area Optimization: Resource Sharing

- Use registers to hold inputs
- Develop FSM to select which inputs to process in each cycle.

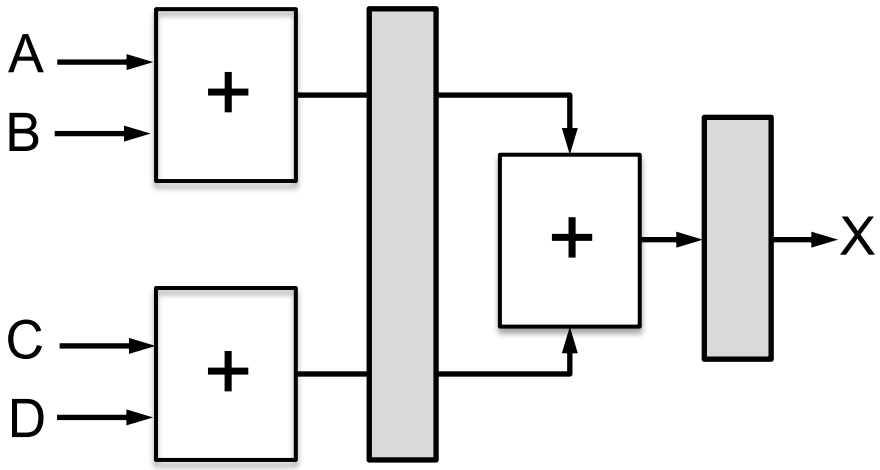
$$X = A + B + C + D$$

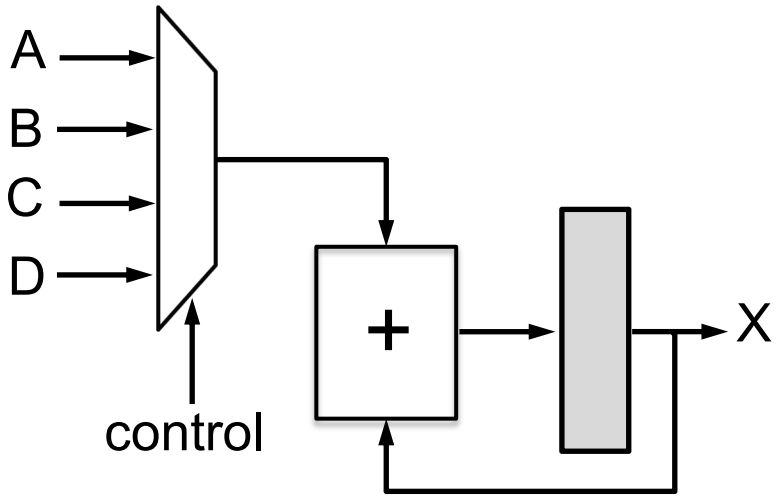


Area Optimization: Resource Sharing

- Use registers to hold inputs
- Develop FSM to select which inputs to process in each cycle.

$$X = A + B + C + D$$

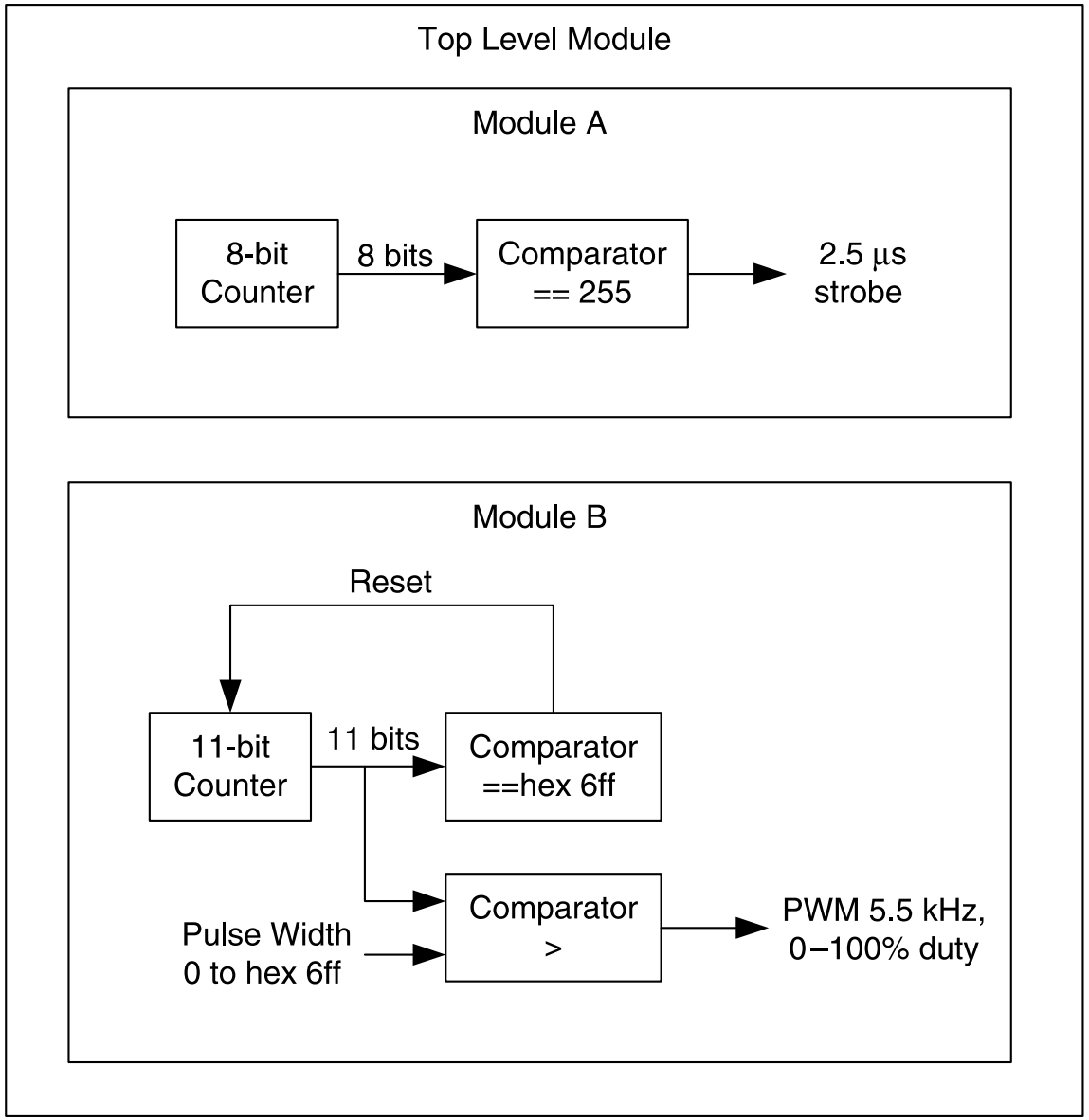




A, B, C, D need to hold steady until X is processed

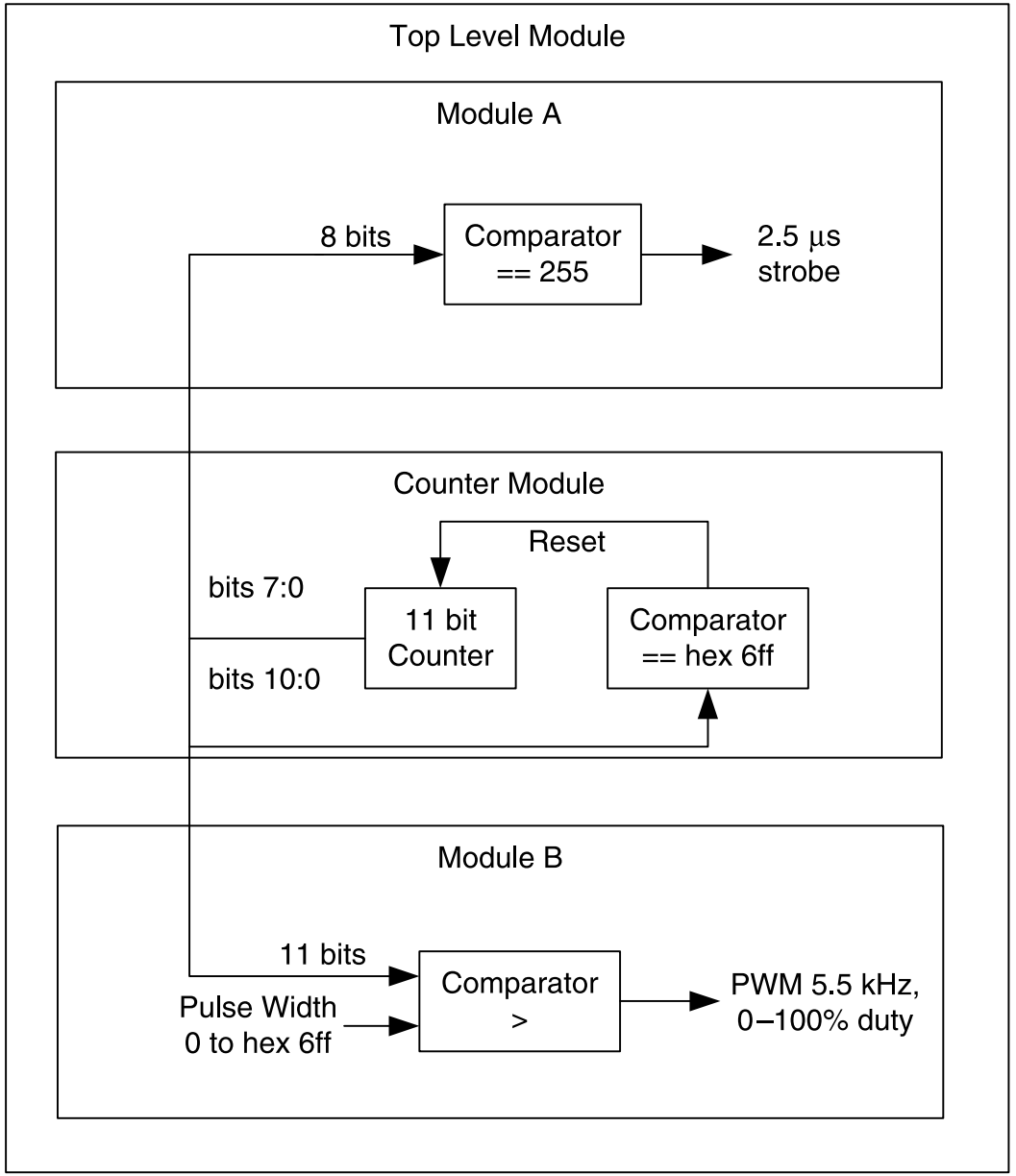
Area Optimization: Resource Sharing

Merge duplicate components together



Area Optimization: Resource Sharing

Merge duplicate components together



Impact of Reset on Area – Xilinx Specific

Do not set or reset Registers asynchronously.

- Control set remapping becomes impossible.
- Sequential functionality in device resources such as block RAM components and DSP blocks can be set or reset synchronously only.
- You will be unable to leverage device resources resources, or they will be configured sub-optimally.
- Use synchronous initialization instead.

Do not describe Flip-Flops with both a set and a reset.

- No Flip-Flop primitives feature both a set and a reset, whether synchronous or asynchronous.
- If not rejected by the software, Flip-Flop primitives featuring both a set and a reset may adversely affect area and performance.

Resetting Block RAM

- On-chip block RAM only supports synchronous reset.
- Suppose that Mem is 256x16b RAM.
- Implementations of Mem with synchronous and asynchronous reset on Xilinx Virtex-4.

Implementation	Slices slice	Flip-flops	4 Input LUTs	BRAMs
Asynchronous reset	3415	4112	2388	0
Synchronous reset	0	0	0	1

Optimization for Power

Power Reduction Techniques

- In general, FPGAs are power hungry.
- Power consumption is determined by

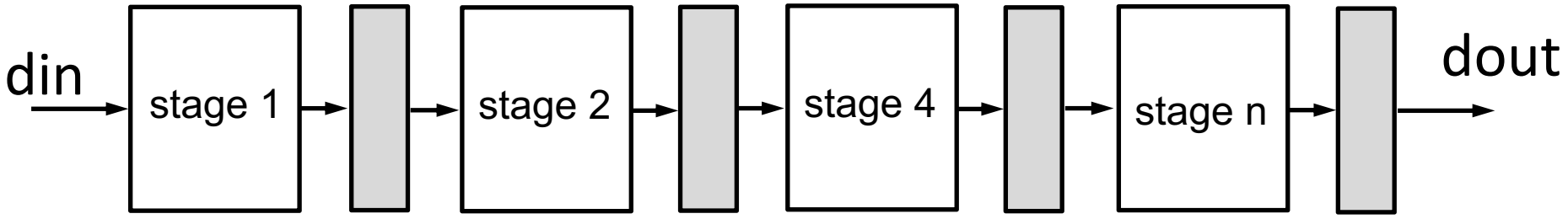
$$P = V^2 \cdot C \cdot f$$

  where *V* is voltage, *C* is load capacitance, and *f* is switching frequency
- In FPGAs, *V* is fixed, *C* depends on the number of switching gates and length of wires connecting all gates.
- To reduce power,
  - turn off gates not actively used,
  - have multiple clock domains,
  - reduce *f*.

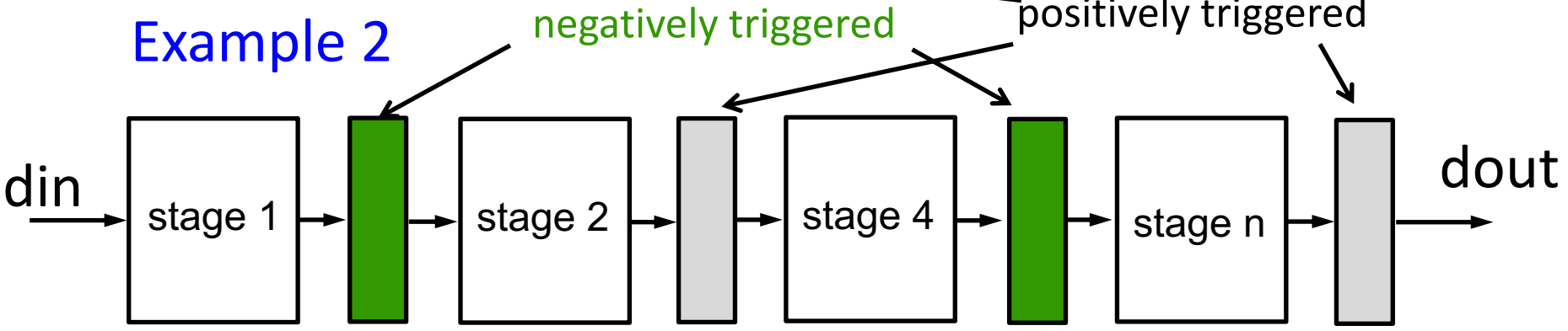
Dual-EdgeTriggered FFs

- A design that is active on both clock edges can reduce clock frequency by 50%.

Example 1



Example 2



Backup

FSMD

Input: M[i]
Outputs: max, min, average

```
max = 0
min = MAX
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32
```