CDA 4253 FPGA System Design Optimization Techniques

Hao Zheng
Comp S ci & Eng
Univ of South Florida

Extracted from

Advanced FPGA Design

by Steve Kilts

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;
```

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

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

Latency: 3 cycles.

Critical path delay: 1 multiplier delay

Achieving High Throughput: Pipelining

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

Throughput: 1 data / cycle

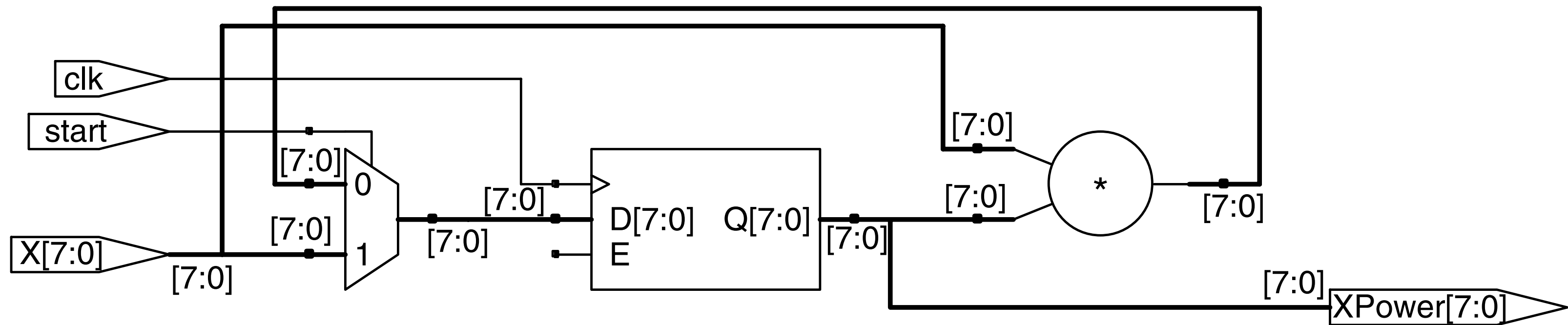
Latency: 3 cycles + register delays.

Critical path delay: 1 multiplier delay

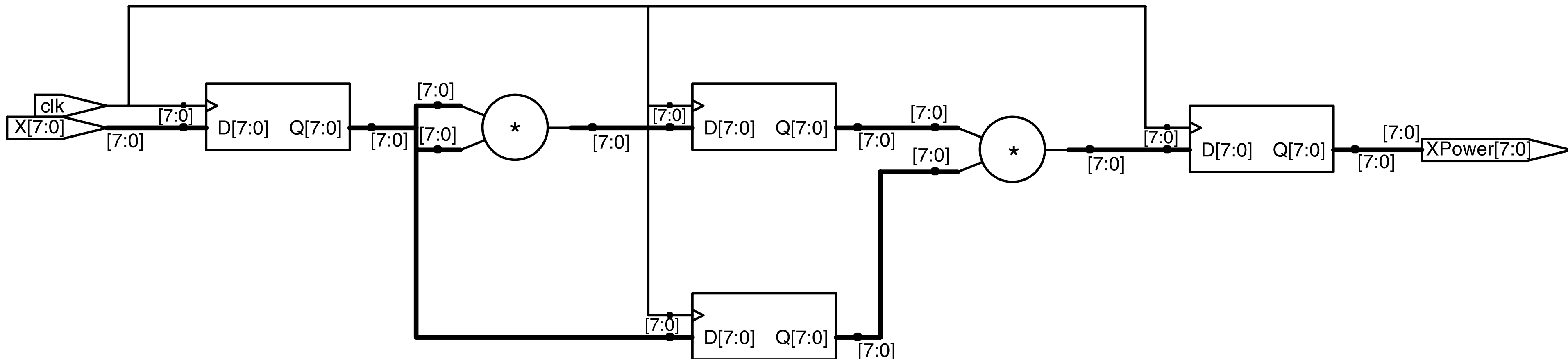
```
-- Pipelined version
process (clk, rst) begin
   if rising_edge(clk) then
      if start='1' then  -- stage 1
         x1 <= x;
         xpower1 <= x;
         done1 <= start;
      end if;
      -- stage 2
      x2 <= x1;
      xpower2 <= xpower1 * x1;
      done2 <= done1;
      -- stage 3
   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

Comparison

Iterative implementation

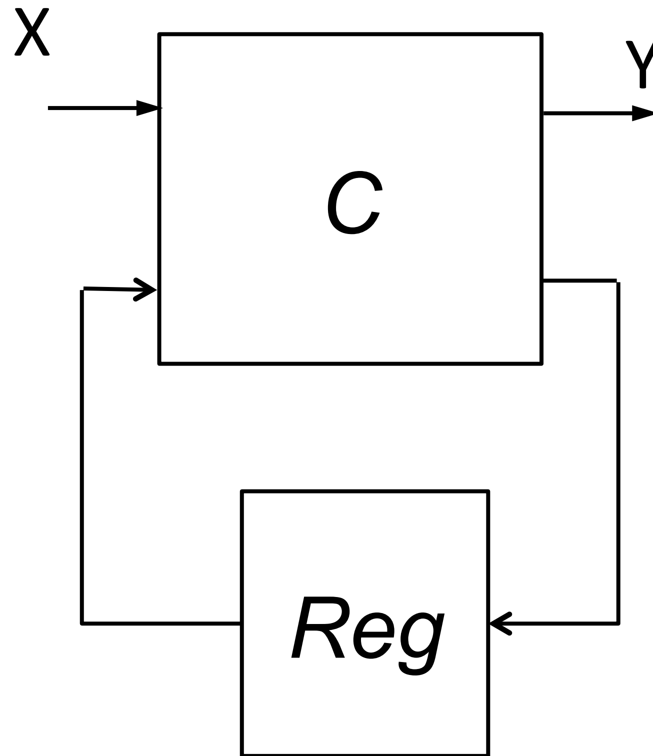


Pipelined implementation



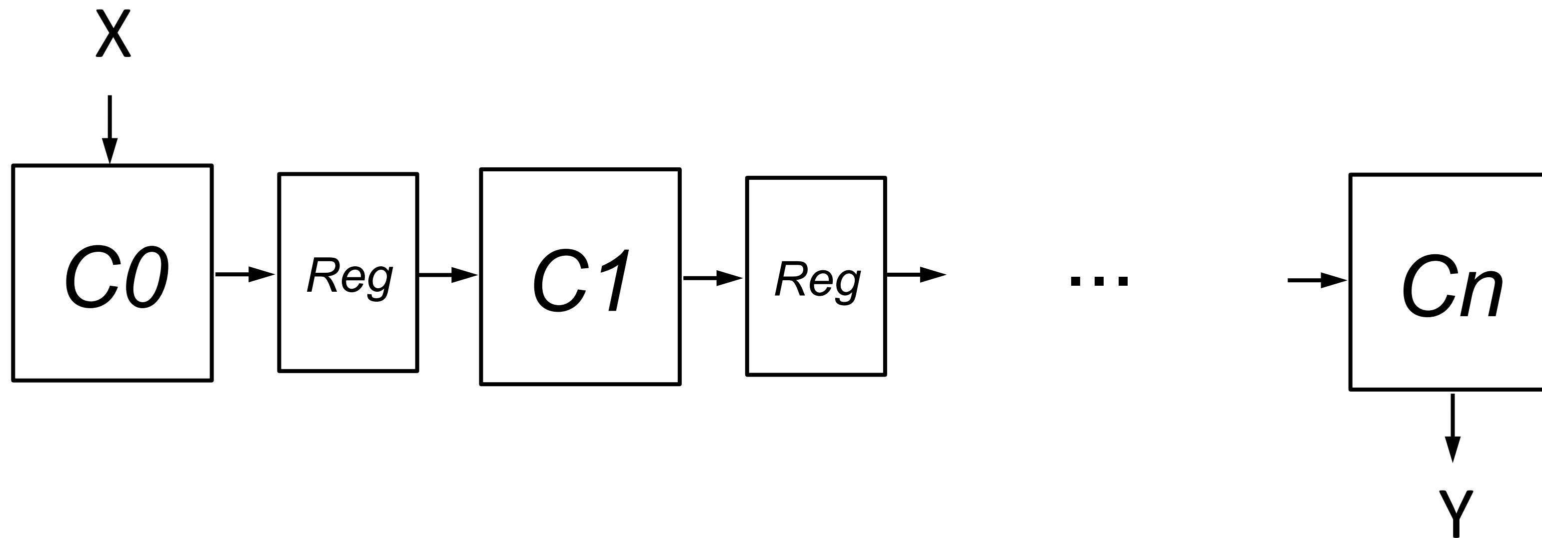
Achieving High Throughput: Pipelining

- Loop unrolling



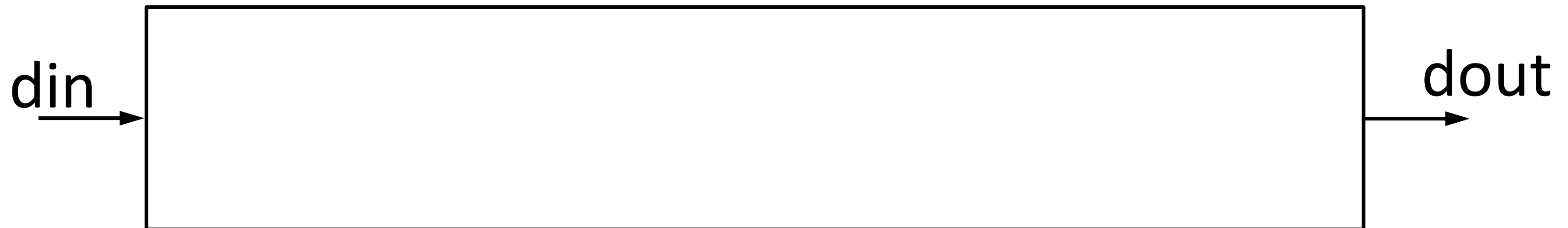
Achieving High Throughput: Pipelining

- Loop unrolling



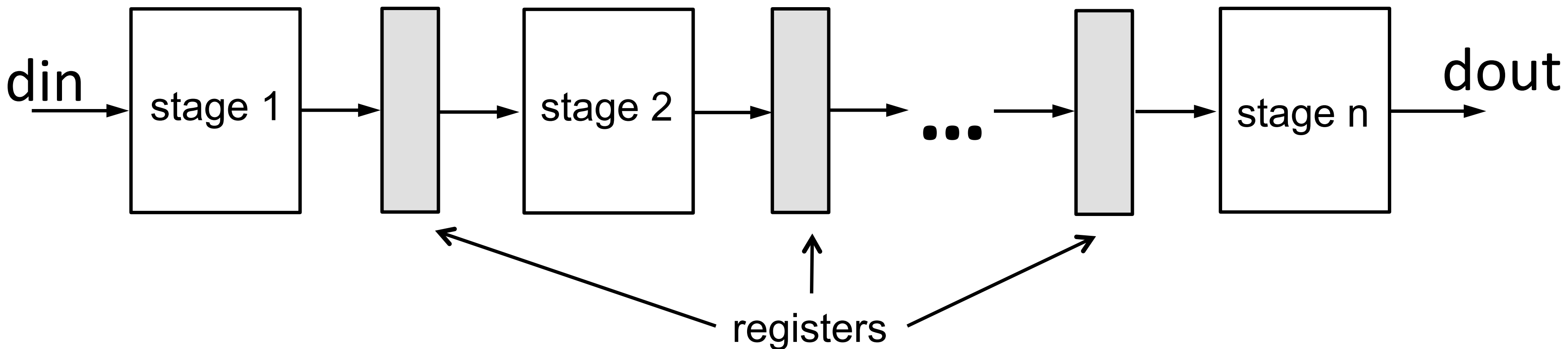
Achieving High Throughput: Pipelining

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



Achieving High Throughput: Pipelining

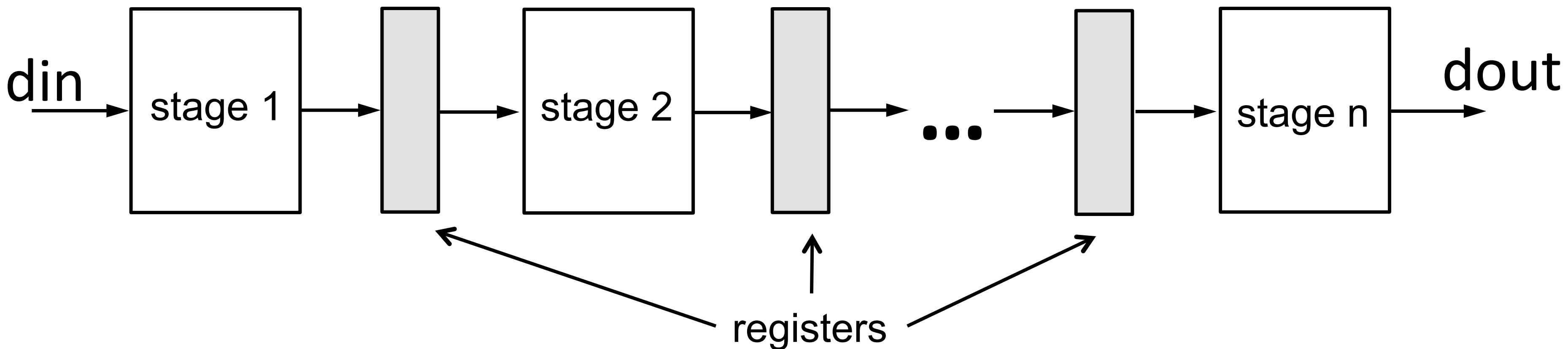
- Divide data processing into stages
- Process different data inputs in different stages simultaneously.



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

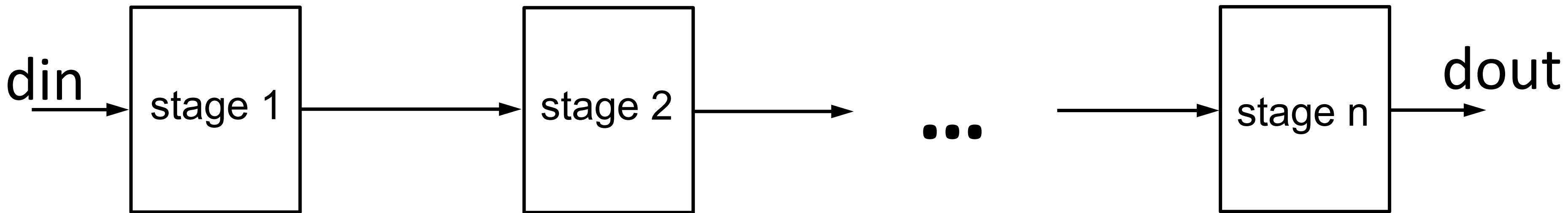
Reducing Latency

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



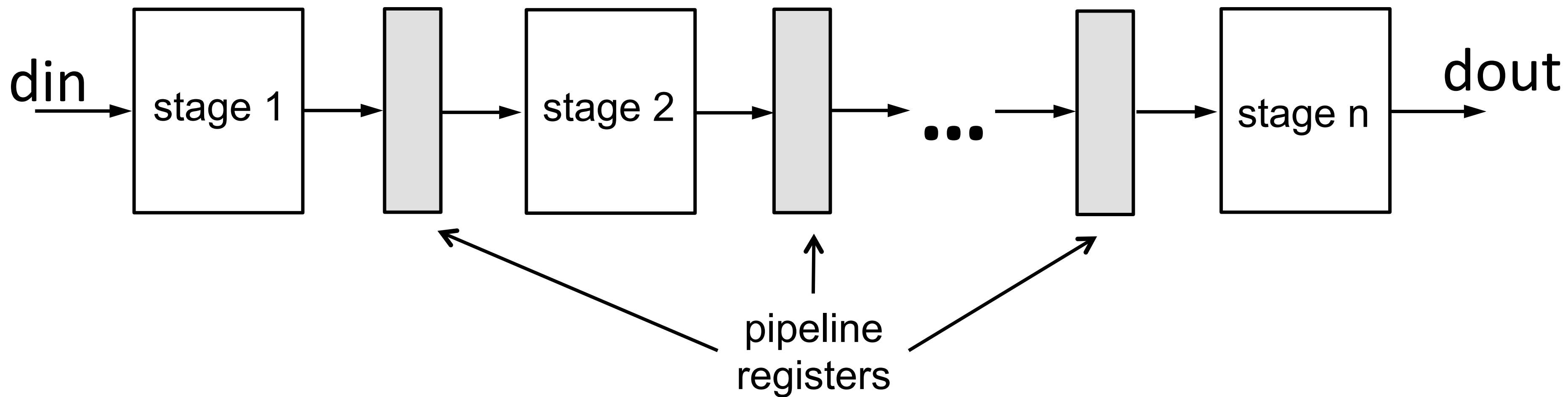
Reducing Latency

- 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 (clk, rst) begin
   if rising_edge(clk) then
      X1 := A + B;
      X2 := X1 + C;
      X  <= X2 + D;
   end if;
end process;
```

Critical path delay: 3 adders

```
process (clk, rst) begin
   if rising_edge(clk) then
      X1 := A + B;
      X2 := C + D;
      X  <= X1 + X2;
   end if;
end process;
```

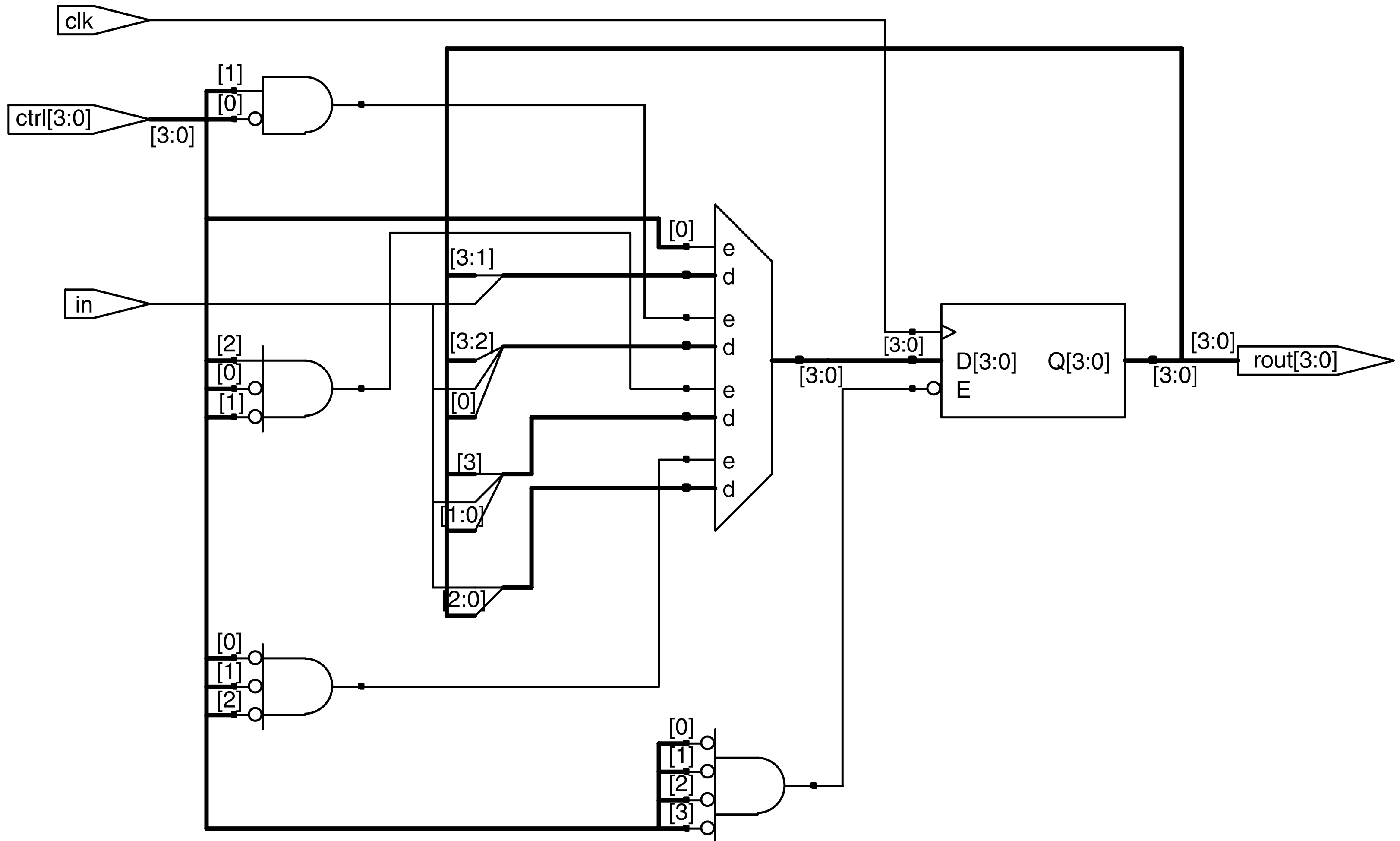
Critical path delay: 2 adders

Timing Optimization: Avoid Unwanted Priority

```
process (clk, rst) begin
    if rising_edge(clk) then
        if     c[0]='1' then r[0] <= din;
        elsif  c[1]='1' then r[1] <= din;
        elsif  c[2]='1' then r[2] <= din;
        elsif  c[3]='1' then r[3] <= din;
        end if;
    end if;
end process;
```

Critical path delay: 3-input AND gate + 4x1 MUX.

Timing Optimization: Avoid Unwanted Priority



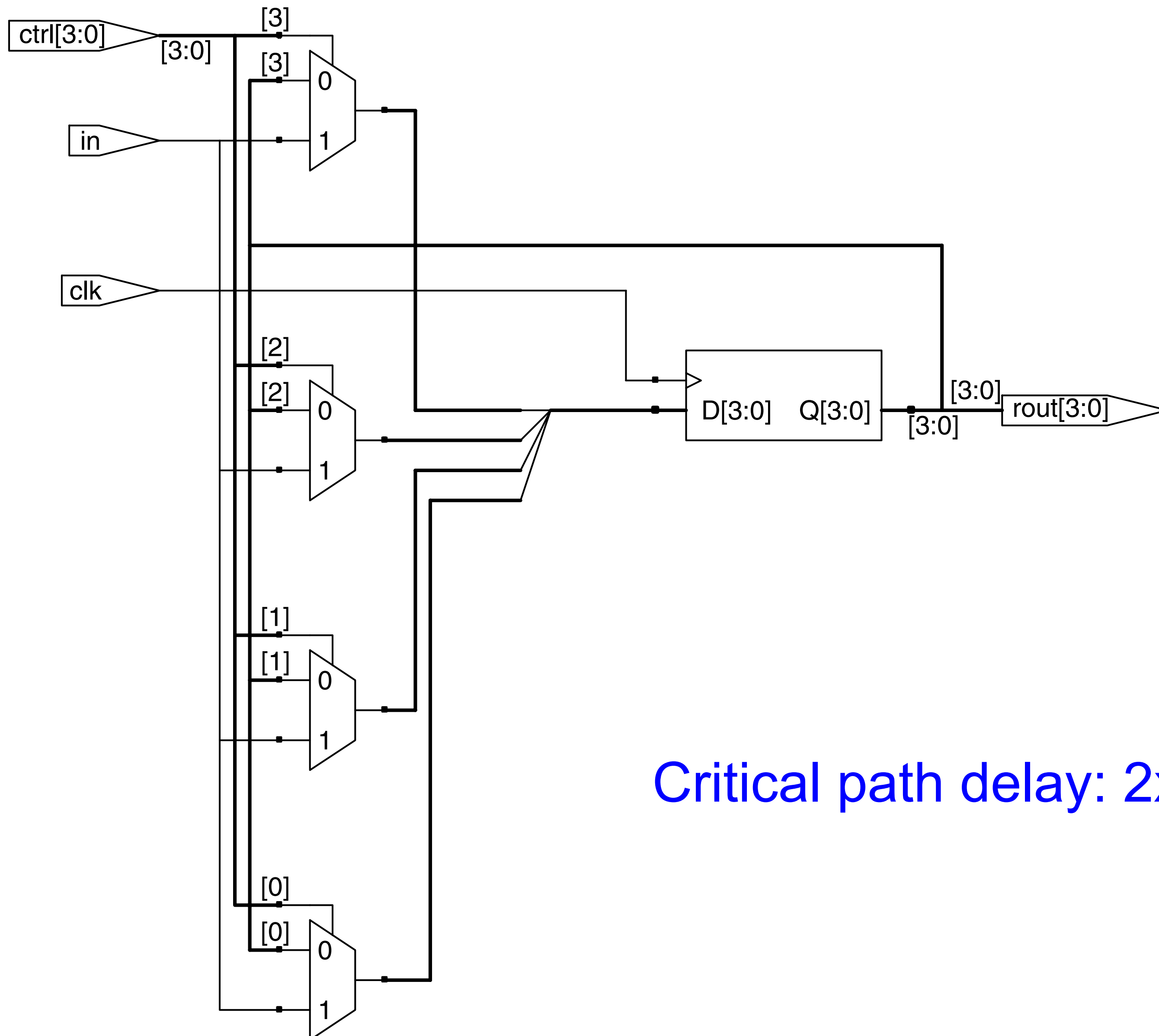
Critical path delay: 3-input AND gate + 4x1 MUX.

Timing Optimization: Avoid Unwanted Priority

```
process (clk, rst) begin
     if rising_edge(clk) then
         if c[0]='1' then r[0] <= din; end if;
         if c[1]='1' then r[1] <= din; end if;
         if c[2]='1' then r[2] <= din; end if;
         if c[3]='1' then r[3] <= din; end if;
    end if;
end process;
```

Critical path delay: 2x1 MUX

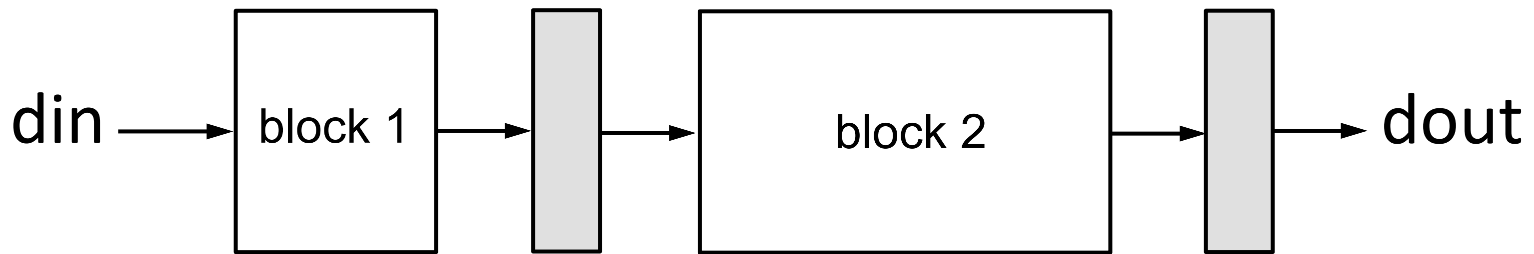
Timing Optimization: Avoid Unwanted Priority

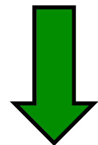


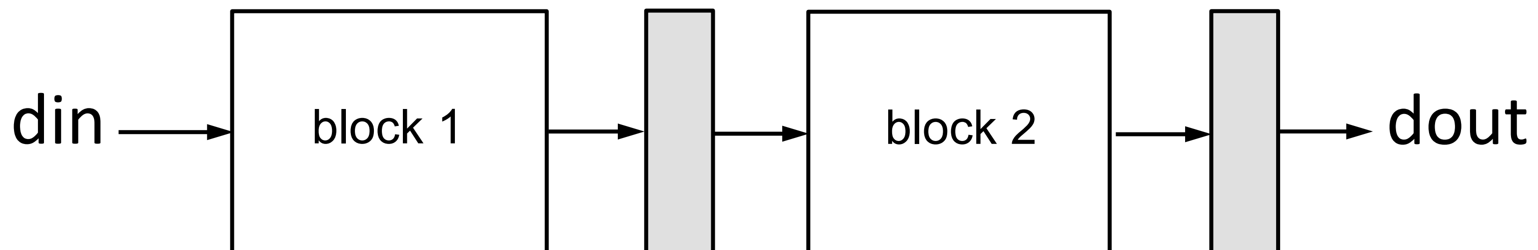
Critical path delay: 2x1 MUX

Timing Optimization: Register Balancing

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







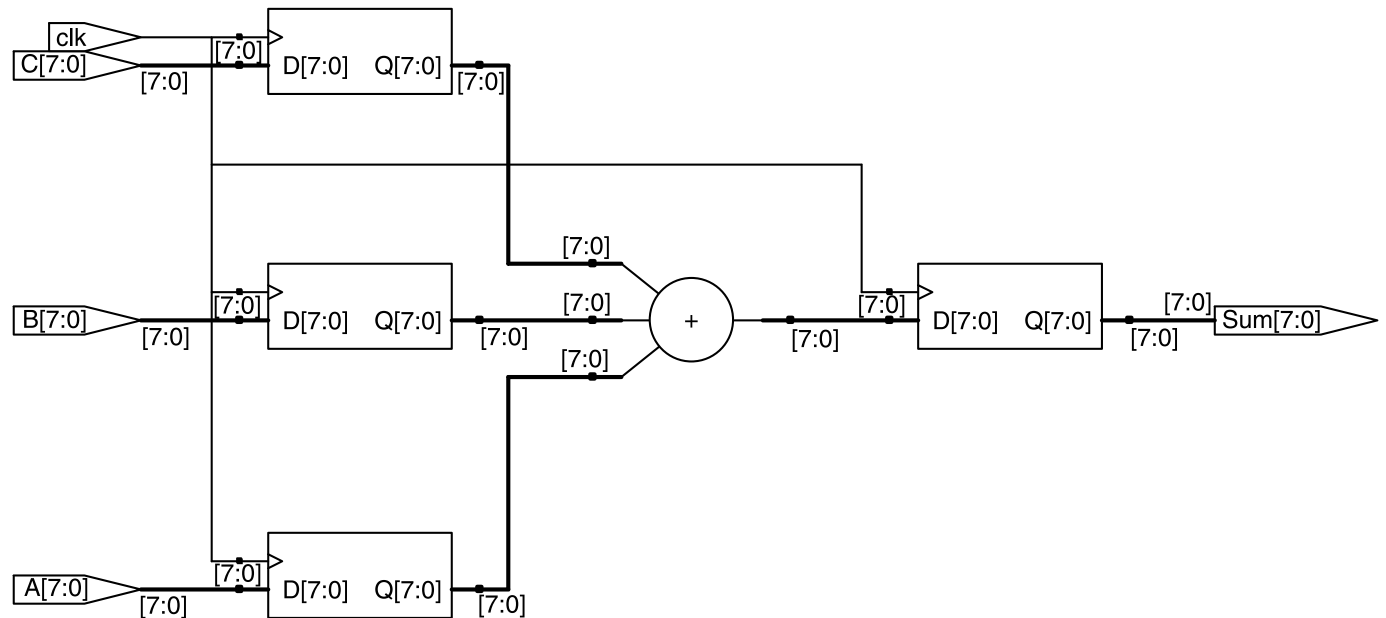
Timing Optimization: Register Balancing

```
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;
```

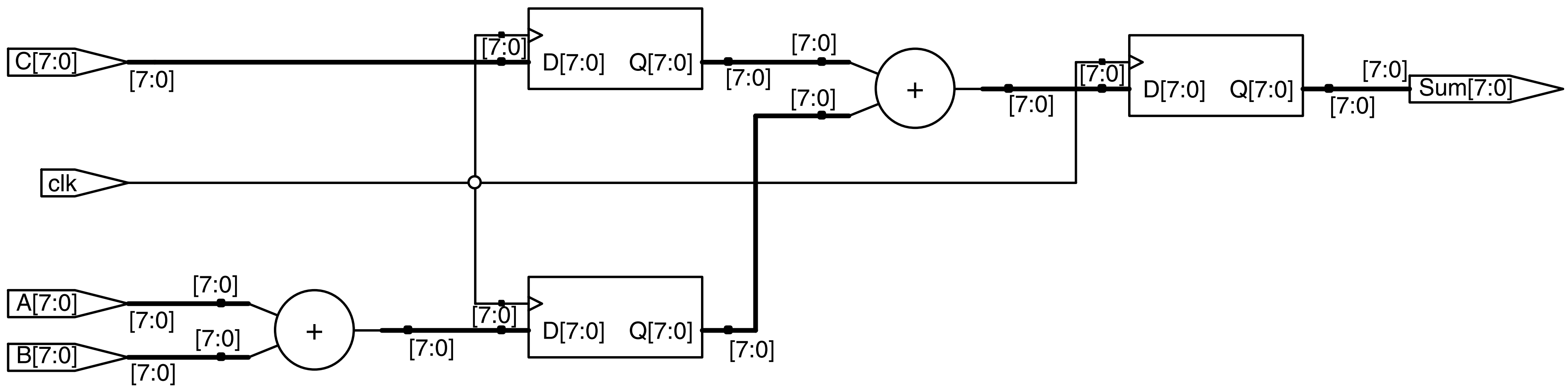
Timing Optimization: Register Balancing

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



Timing Optimization: Register Balancing

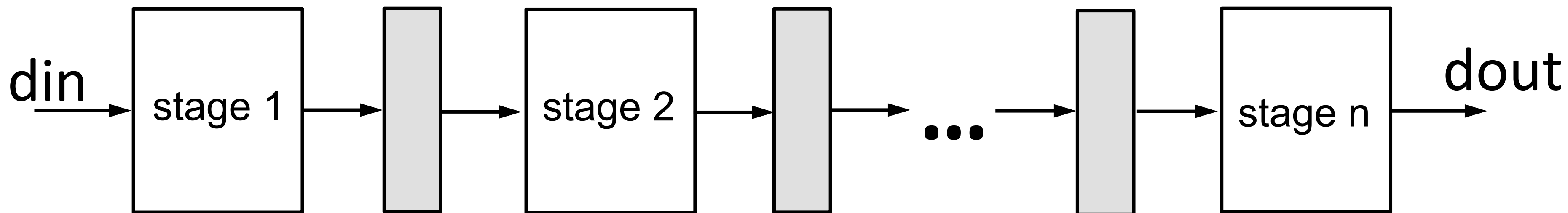
```
process (clk, rst) begin
  if rising_edge(clk) then
    sumAB <= A + B;
    rC <= C;
    sum <= sumAB + rC;
  end if;
end process;
```

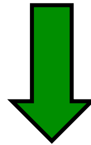


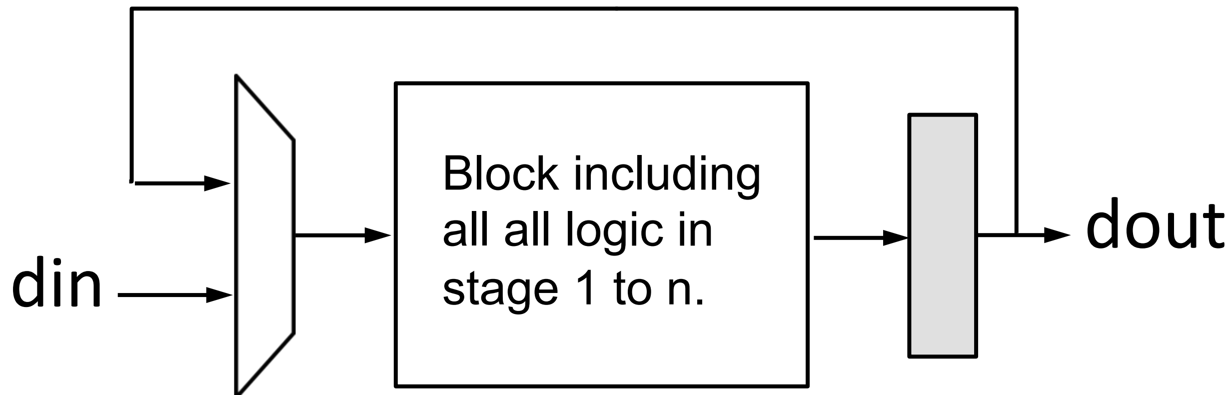
Optimization for Area

Area Optimization: Resource Sharing

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



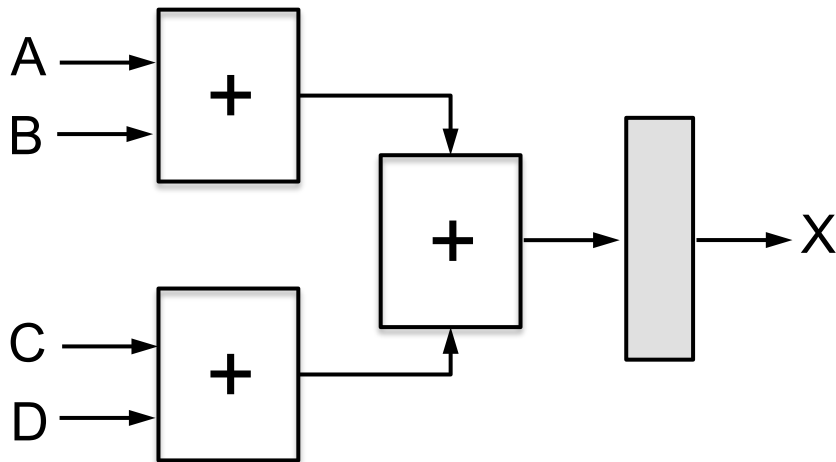




Area Optimization: Resource Sharing

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

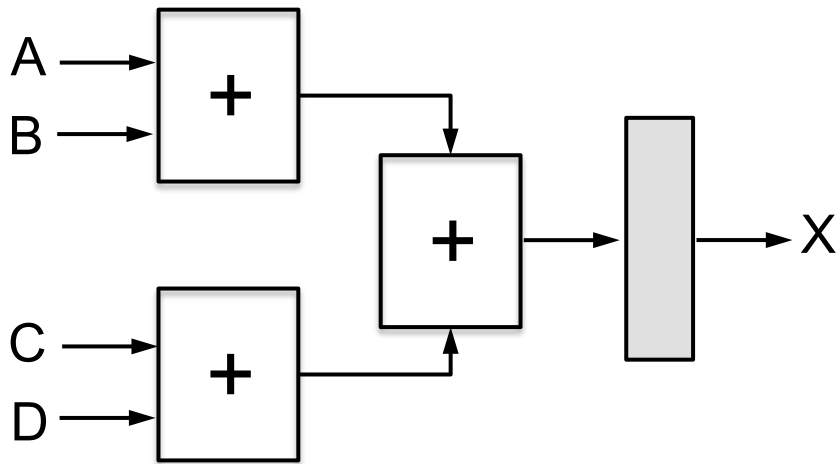
$$X = A + B + C + D$$

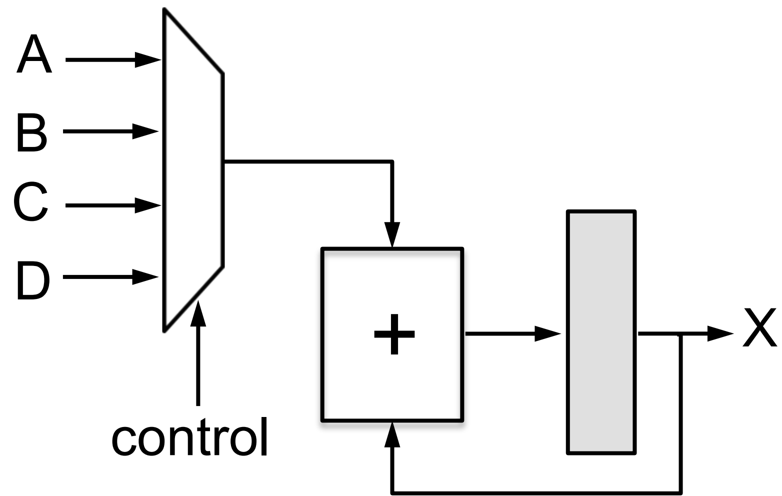


Area Optimization: Resource Sharing

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

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

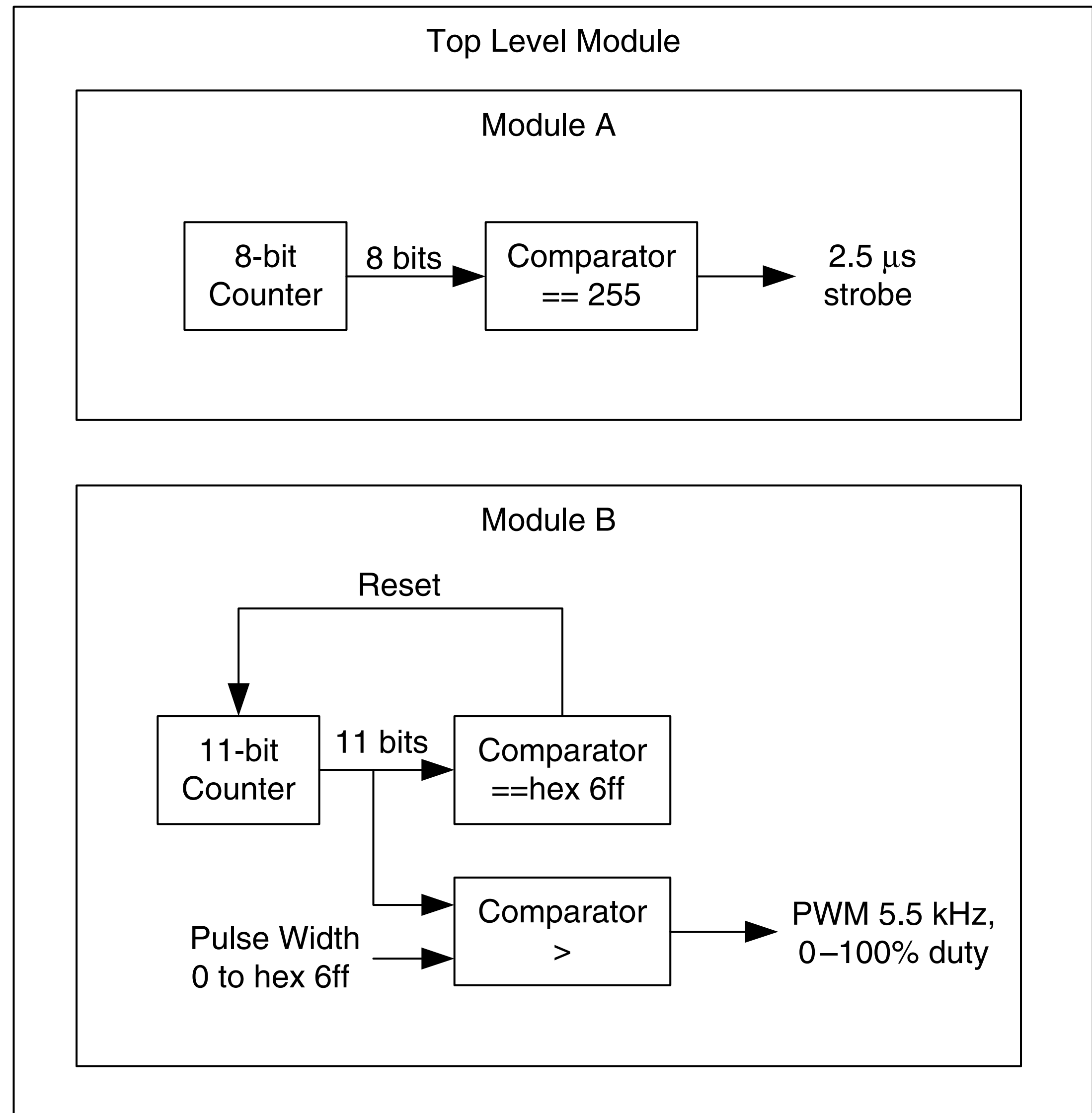




A, B, C, D need to hold steady until X is processed

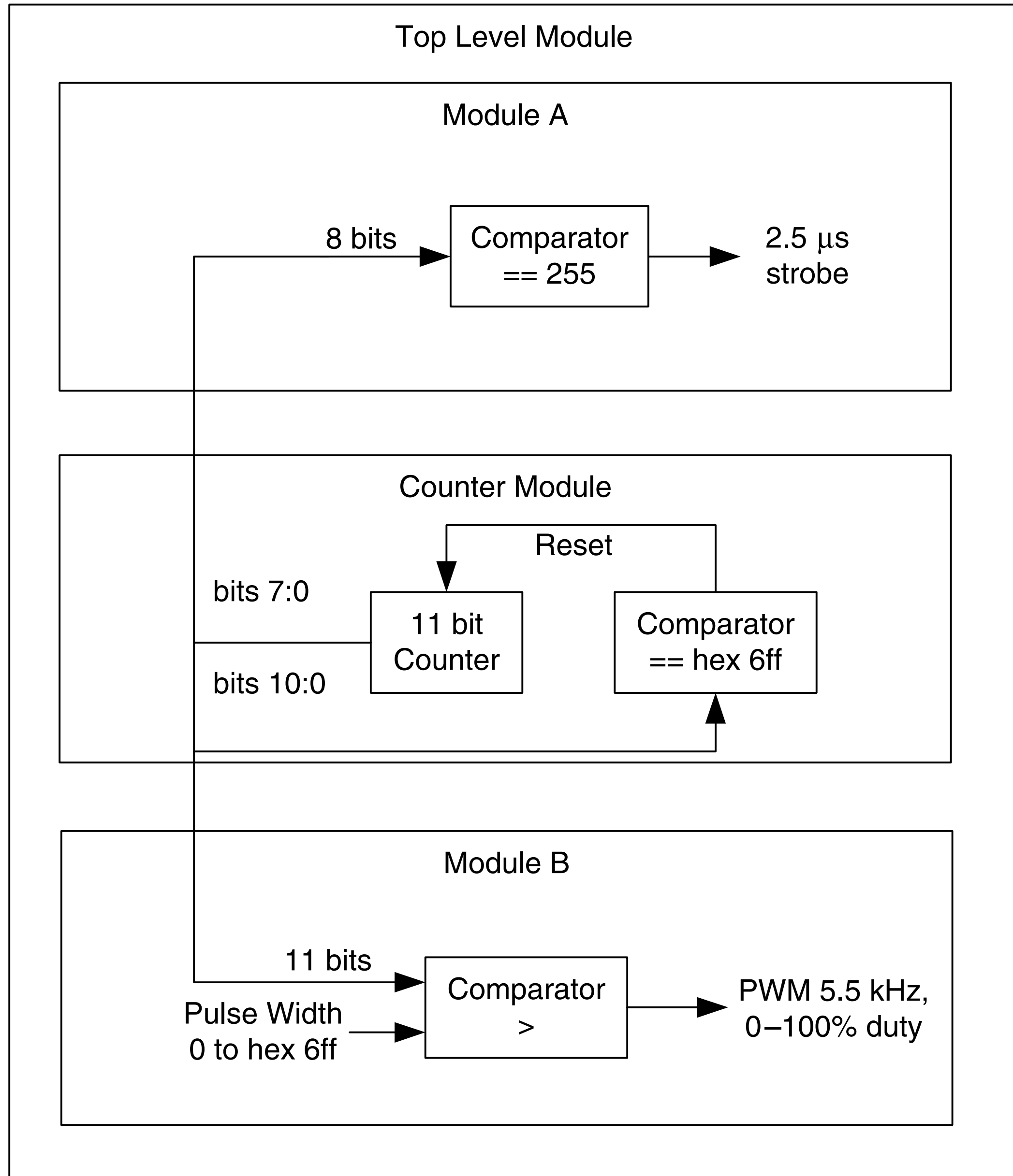
Area Optimization: Resource Sharing

Merge duplicate components together



Area Optimization: Resource Sharing

Merge duplicate components together – reduces a 8-bit counter



Impact of Reset on Area (Xilinx Specific)

Do not set or reset Registers asynchronously.

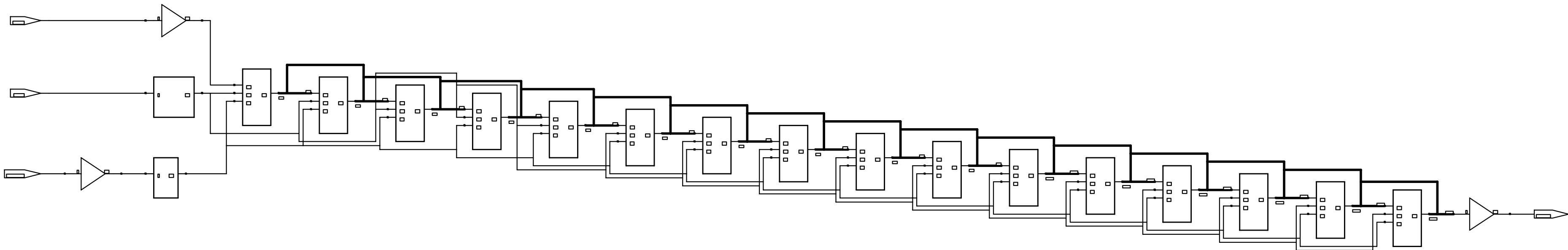
- Control set remapping becomes impossible.
- Sequential functionality in device resources such as block RAM components and DSP blocks can be set or reset synchronously only.
- You will be unable to leverage device resources resources, or they will be configured sub-optimally.
- Use synchronous initialization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

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

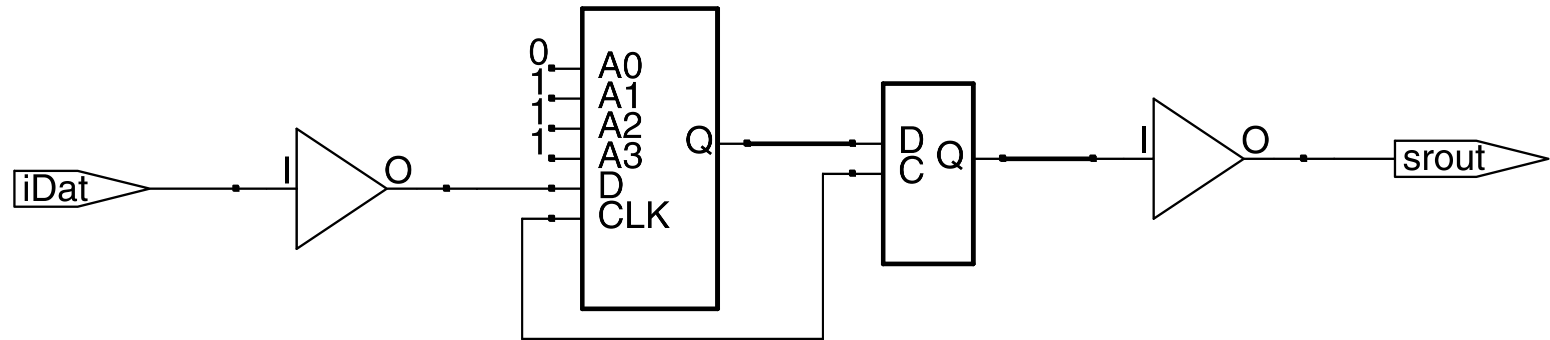
- No Flip-Flop primitives feature both a set and a reset, whether synchronous or asynchronous.
- If not rejected by the software, Flip-Flop primitives featuring both a set and a reset may adversely affect area and performance.

Reset or No Reset?

```
process (clk) begin
  if rising_edge(clk) then
     if rst = '0' then
        sr <= (others <= '0');
     else
        sr <= din & sr(14 downto 0);
     end if;
  end if;
end process;
```



Reset or No Reset?




```
process (clk) begin
  if rising_edge(clk) then
    sr <= din & sr(14 downto 0);
  end if;
end process;
```

Reset or No Reset?

Table 2.1 Resource Utilization for Shift Register Implementations

Implementation	Slices slice	Flip-flops
Resets defined	9	16
No resets defined	1	1

Resetting Block RAM

- 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

VHDL model should match features offered by FPGA building blocks in order for those devices instantiated in the implementation.

Utilizing Set/Reset FF Pins

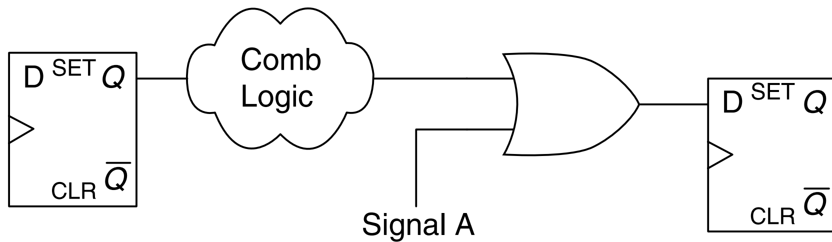


Figure 2.11 Simple synchronous logic with OR gate.

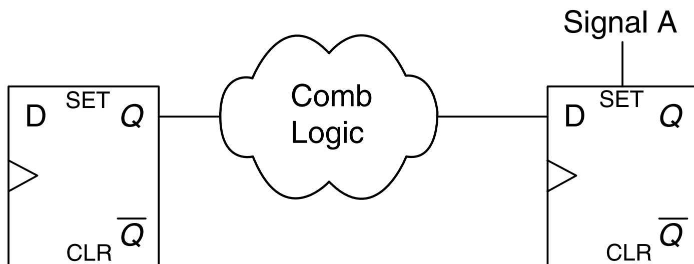


Figure 2.12 OR gate implemented with set pin.

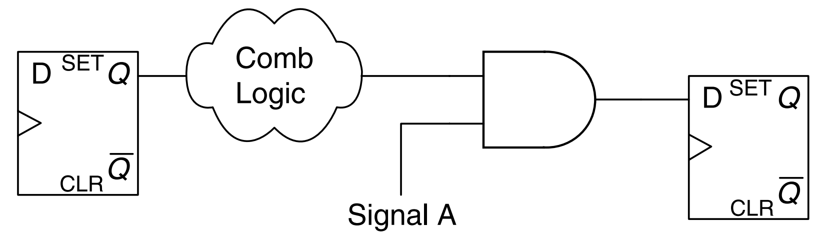


Figure 2.13 Simple synchronous logic with AND gate.

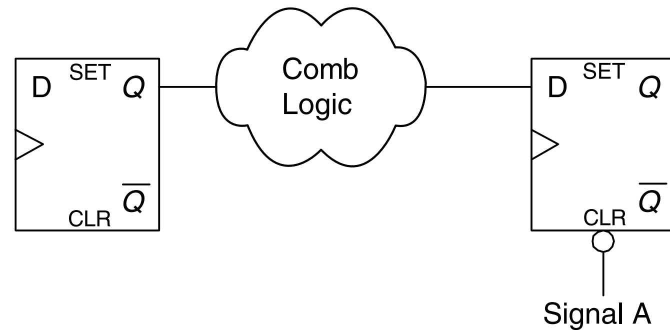


Figure 2.14 AND gate implemented with CLR pin.

Utilizing Set/Reset FF Pins – Example

```
process (clk, reset)
begin
  if reset='0' then
    oDat <= '0';
  else
    oDat <= iDat1 | iDat2;
  end if;
end process;
```

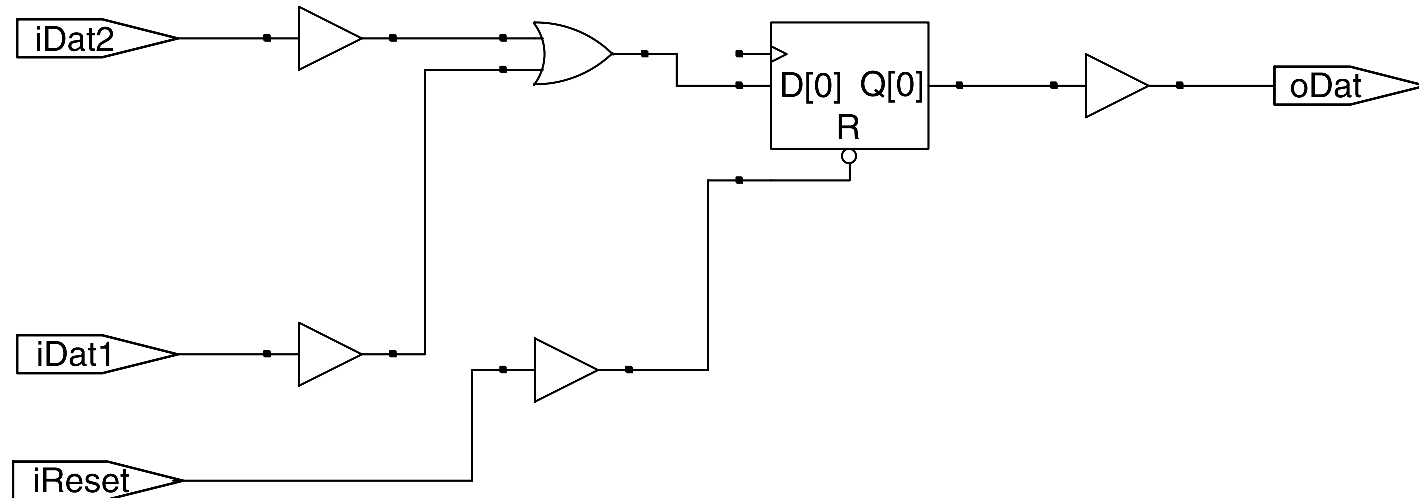


Figure 2.15 Simple asynchronous reset.

Utilizing Set/Reset FF Pins – Example

```
process (clk, reset)
begin
  oDat <= iDat1 | iDat2;
end process;
```

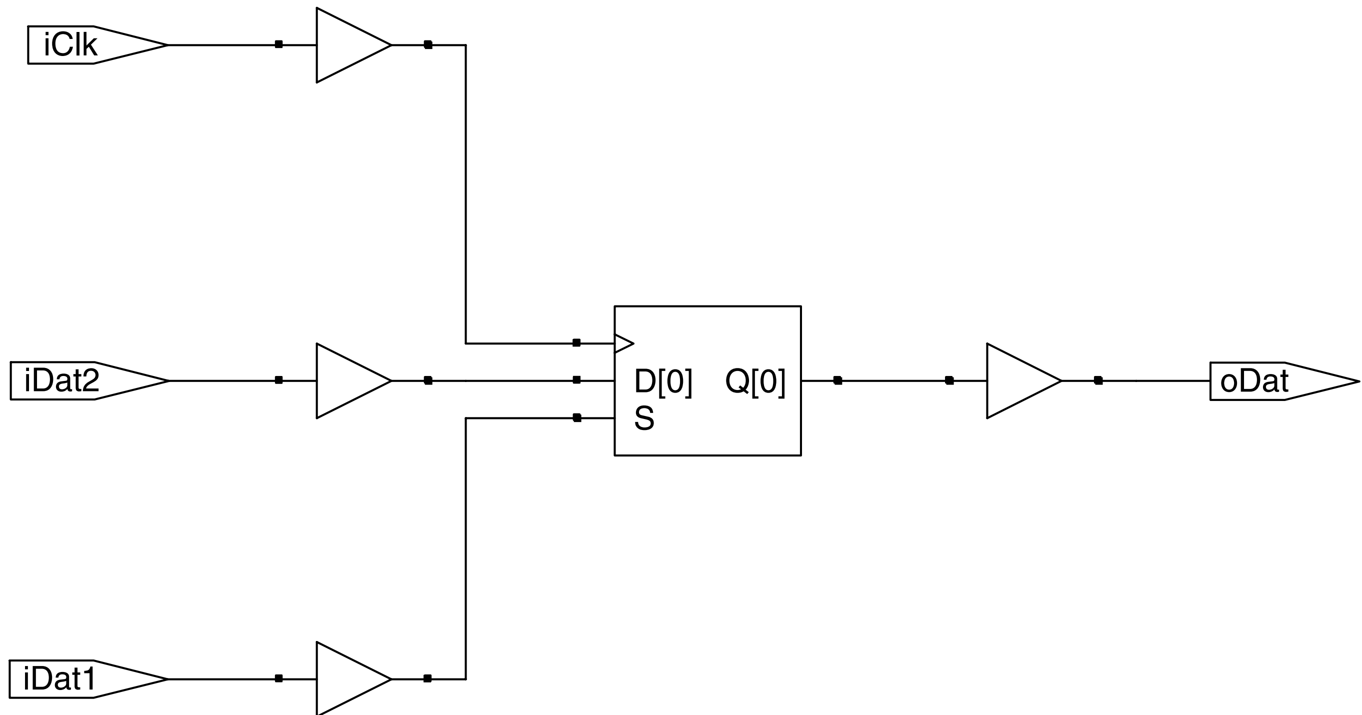


Figure 2.16 Optimization without reset.

Optimization for Power

Power Reduction Techniques

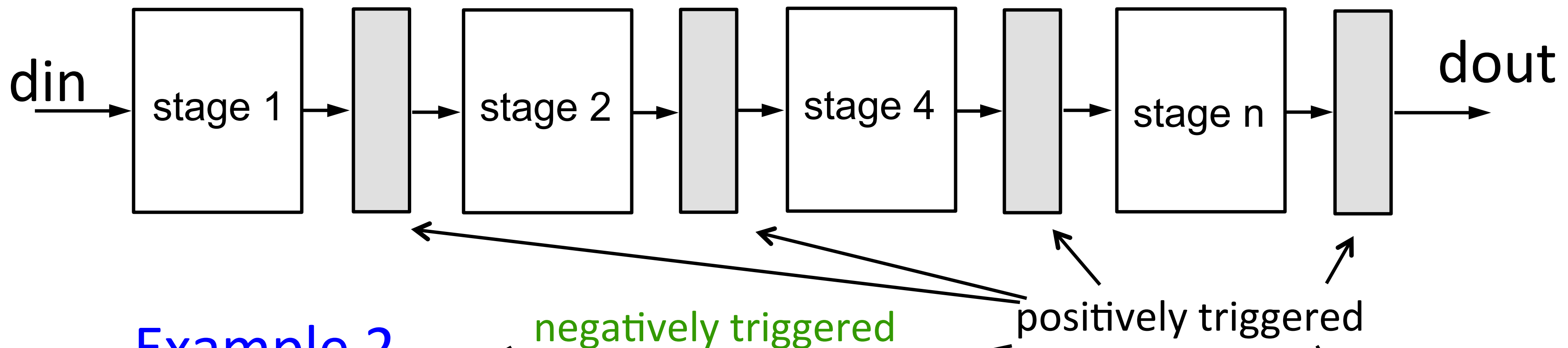
- In general, FPGAs are power hungry.
- Power consumption is determined by

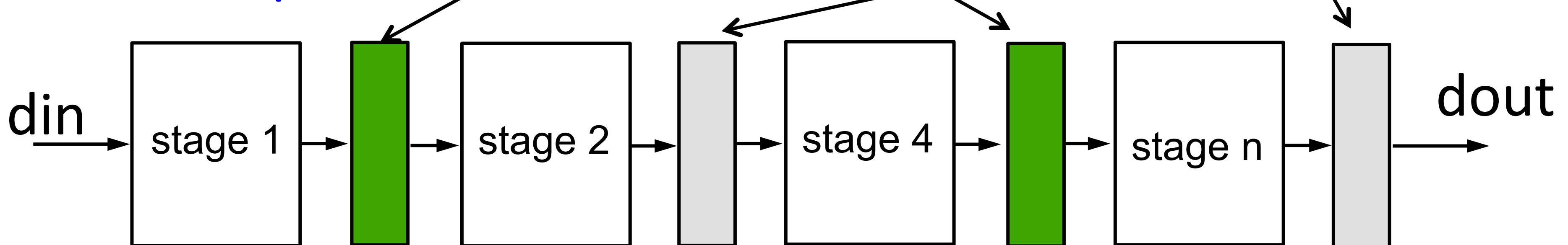
$$P = V^2 \cdot C \cdot f$$

  where *V* is voltage, *C* is load capacitance, and *f* is switching frequency
- In FPGAs, *V* is usually 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%.





Dual-EdgeTriggered FFs – Example

```
process(clk)
begin
   if (rising_edge(clk)) then
            reg(0) <= din;
            reg(2) <= reg(1);
   end if;
end process;

process(clk)
begin
   if(rising_edge(clk)) then
            reg(1) <= reg(0);
            reg(3) <= reg(2);
   end if;
end process;
```

Synthesizable using Vivado 2016.2