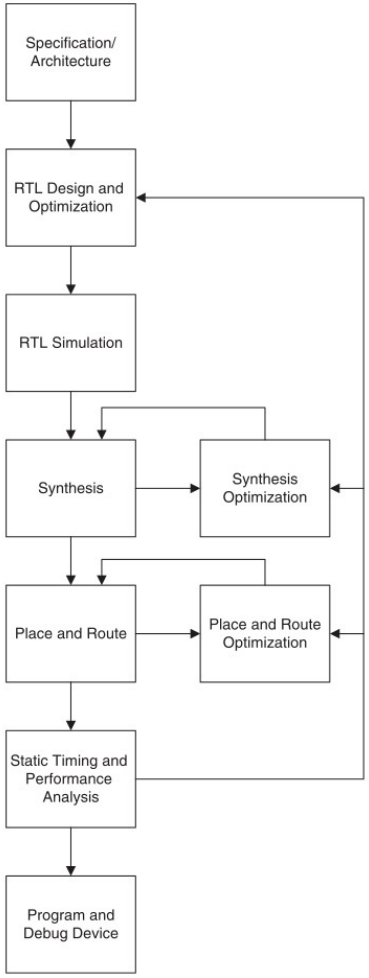
EEL 4783: HDL in Digital System Design

Lecture 3: Architeching Speed

Prof. Mingjie Lin

Flowchart of CAD



Digital Circuits: Definition of Speed

- Throughput
  - The amount of data that is processed per clock cycle.
  - A common metric for throughput is bits per second.
- Latency
  - The time between data input and processed data output
  - Typical metric: time or clock cycles
- Timing
  - Logic delays between sequential elements
  - Timing: critical path delay
  - Composed of comb. Delay, clk-to-out delay, routing delay, skew
  - Typical metric: clock period and frequency

Main Points

- High-throughput architecture
  - Maximizing number of bits processed per second.
- Latency
  - The time between data input and processed data output
  - Typical metric: time or clock cycles
- Timing
  - Logic delays between sequential elements
  - Timing: critical path delay
  - Composed of comb. Delay, clk-to-out delay, routing delay, skew
  - Typical metric: clock period and frequency

High Throughput

- A high-throughput design is one that is concerned with the steady-state data rate but less concerned about the time any specific piece of data requires to propagate through the design (latency).
- Analogy:
  - Ford came up with to manufacture automobiles in great quantities: an assembly line.
- Technique:
  - Pipelining
  - Price to pay?

Algorithmic Perspective of Pipeline

1. Unrolling the loop

a) Iterative

```
XPower = 1;
for (i=0;i < 3; i++)
  XPower = X * XPower;
```

```
module power3(
  output [7:0] XPower,
  output       finished,
  input  [7:0] X,
  input        clk, start); // the duration of start is a
                            single clock
  reg    [7:0] ncount;
  reg    [7:0] XPower;

  assign finished = (ncount == 0);

  always@(posedge clk)
    if(start) begin
      XPower <= X;
      ncount <= 2;
    end
    else if(!finished) begin
      ncount <= ncount - 1;
      XPower <= XPower * X;
    end
endmodule
```

Algorithmic Perspective of Pipeline

1. Unrolling the loop

a) Iterative

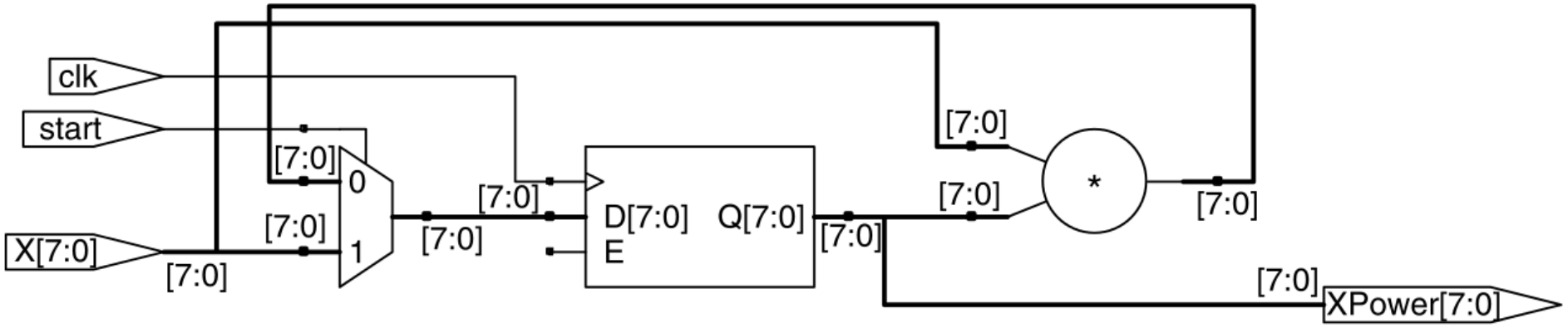
```
XPower = 1;
for (i=0;i < 3; i++)
  XPower = X * XPower;
```

```
module power3(
  output [7:0] XPower,
  output       finished,
  input  [7:0] X,
  input        clk, start); // the duration of start is a
                            single clock
  reg    [7:0] ncount;
  reg    [7:0] XPower;

  assign finished = (ncount == 0);

  always@(posedge clk)
    if(start) begin
      XPower <= X;
      ncount <= 2;
    end
    else if(!finished) begin
      ncount <= ncount - 1;
      XPower <= XPower * X;
    end
endmodule
```

Performance



Throughput = 8/3, or 2.7 bits/clock

Latency = 3 clocks

Timing = One multiplier delay in the critical path

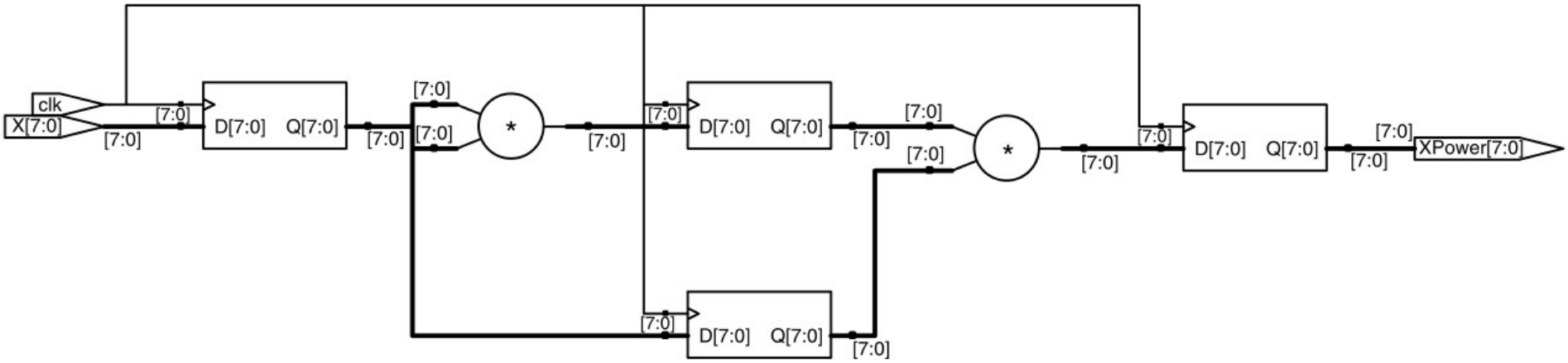
Pipelined Version

```
module power3(
  output reg [7:0] XPower,
  input            clk,
  input      [7:0] X
  );
  reg        [7:0] XPower1, XPower2;
  reg        [7:0] X1, X2;
  always @(posedge clk) begin
    // Pipeline stage 1
    X1      <= X;
    XPower1 <= X;

    // Pipeline stage 2
    X2      <= X1;
    XPower2 <= XPower1 * X1;

    // Pipeline stage 3
    XPower <= XPower2 * X2;
  end
endmodule
```

Circuit and Performance



Throughput = 8/1, or 8 bits/clock

Latency = 3 clocks

Timing = One multiplier delay in the critical path

Low-Latency

1. A low-latency design is one that passes the data from the input to the output as quickly as possible by minimizing the intermediate processing delays.
2. Oftentimes, a low-latency design will require parallelisms, removal of pipelining, and logical short cuts that may reduce the throughput or the max clock speed in a design.
3. No opportunity reducing latency in serial circuit, but there is in pipelined version.

Circuit

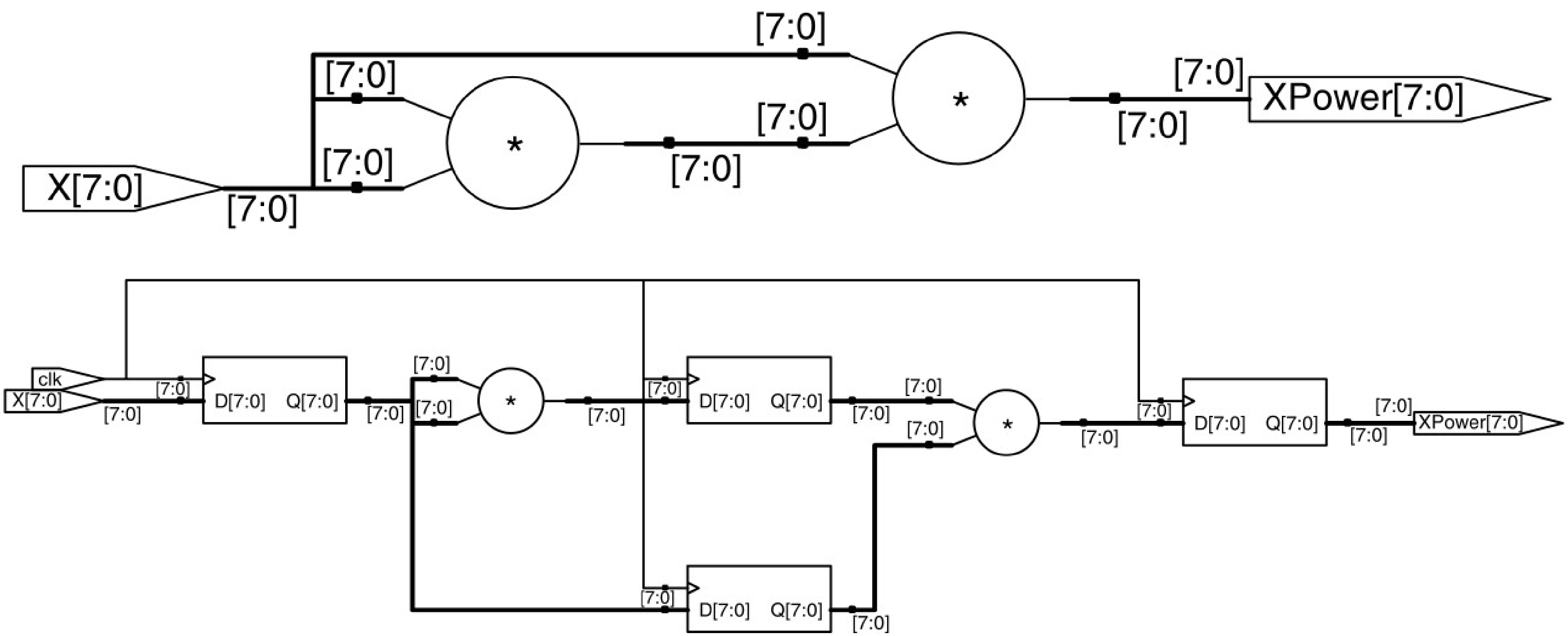
```
module power3(
  output [7:0] XPower,
  input  [7:0] X
  );
  reg    [7:0] XPower1, XPower2;
  reg    [7:0] X1, X2;

  assign XPower = XPower2 * X2;

  always @* begin
    X1      = X;
    XPower1 = X;
  end

  always @* begin
    X2      = X1;
    XPower2 = XPower1*X1;
  end
endmodule
```

Circuit and Performance



Throughput = 8 bits/clock (assuming one new input per clock)

Latency = Between one and two multiplier delays, 0 clocks

Timing = Two multiplier delays in the critical path

Timing

1. Timing refers to the clock speed of a design.
2. The maximum delay between any two sequential elements in a design will determine the max clock speed.

$$F_{max} = \frac{1}{T_{clk-q} + T_{logic} + T_{routing} + T_{setup} - T_{skew}} \quad (1.1)$$

where F_{max} is maximum allowable frequency for clock; $T_{clk\text{-}q}$ is time from clock arrival until data arrives at Q; T_{logic} is propagation delay through logic between flip-flops; $T_{routing}$ is routing delay between flip-flops; T_{setup} is minimum time data must arrive at D before the next rising edge of clock (setup time); and T_{skew} is propagation delay of clock between the launch flip-flop and the capture flip-flop.

How to Improve Timing? 1

1. The first strategy for architectural timing improvements is to add intermediate layers of registers to the critical path.
2. This technique should be used in highly pipelined designs where an additional clock cycle latency does not violate the design specifications, and the overall functionality will not be affected by the further addition of registers.

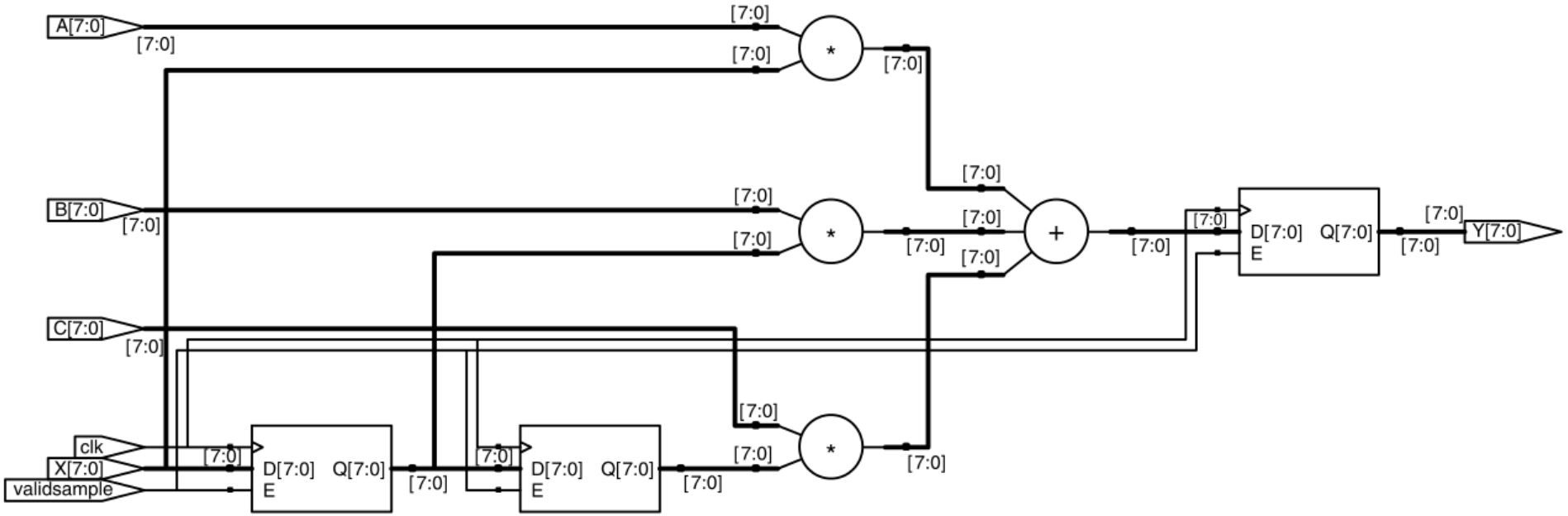
FIR Filter Example

```
module fir(
  output [7:0] Y,
  input  [7:0] A, B, C, X,
  input        clk,

 input         validsample);
 reg    [7:0] X1, X2, Y;

 always @(posedge clk)
   if(validsample) begin
     X1 <= X;
     X2 <= X1;
     Y <= A* X+B* X1+C* X2;
   end
endmodule
```

Diagram

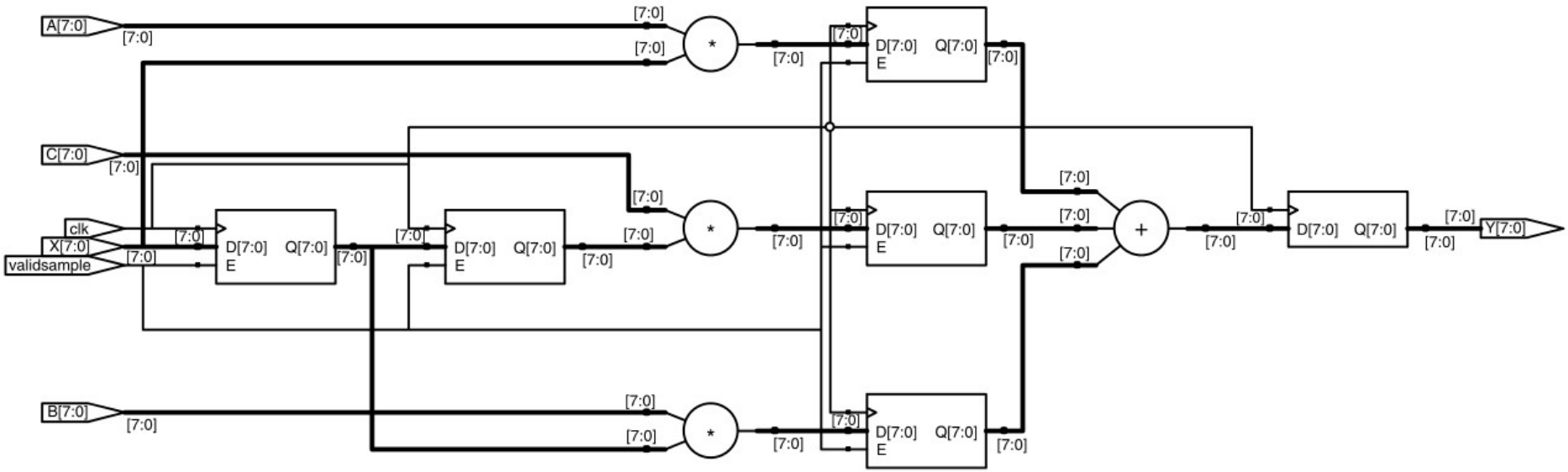


Improve Timing

```
module fir(
  output [7:0] Y,
  input  [7:0] A, B, C, X,
  input        clk,
  input        validsample);
  reg    [7:0] X1, X2, Y;
  reg    [7:0] prod1, prod2, prod3;

  always @ (posedge clk) begin
    if(validsample) begin
      X1    <= X;
      X2    <= X1;
      prod1 <= A * X;
      prod2 <= B * X1;
      prod3 <= C * X2;
    end
    Y <= prod1 + prod2 + prod3;
  end
endmodule
```

Diagram

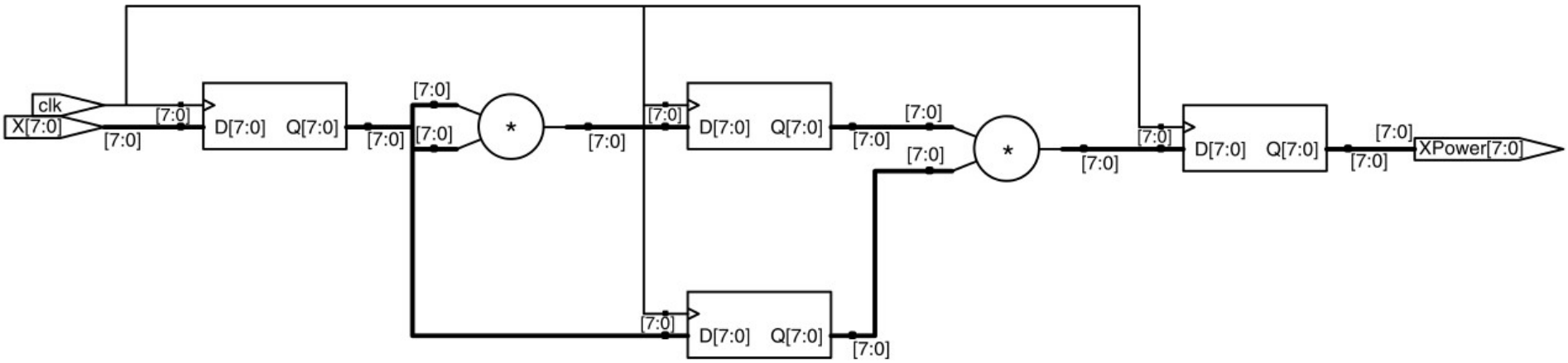


Adding register layers improves timing by dividing the critical path into two paths of smaller delay.

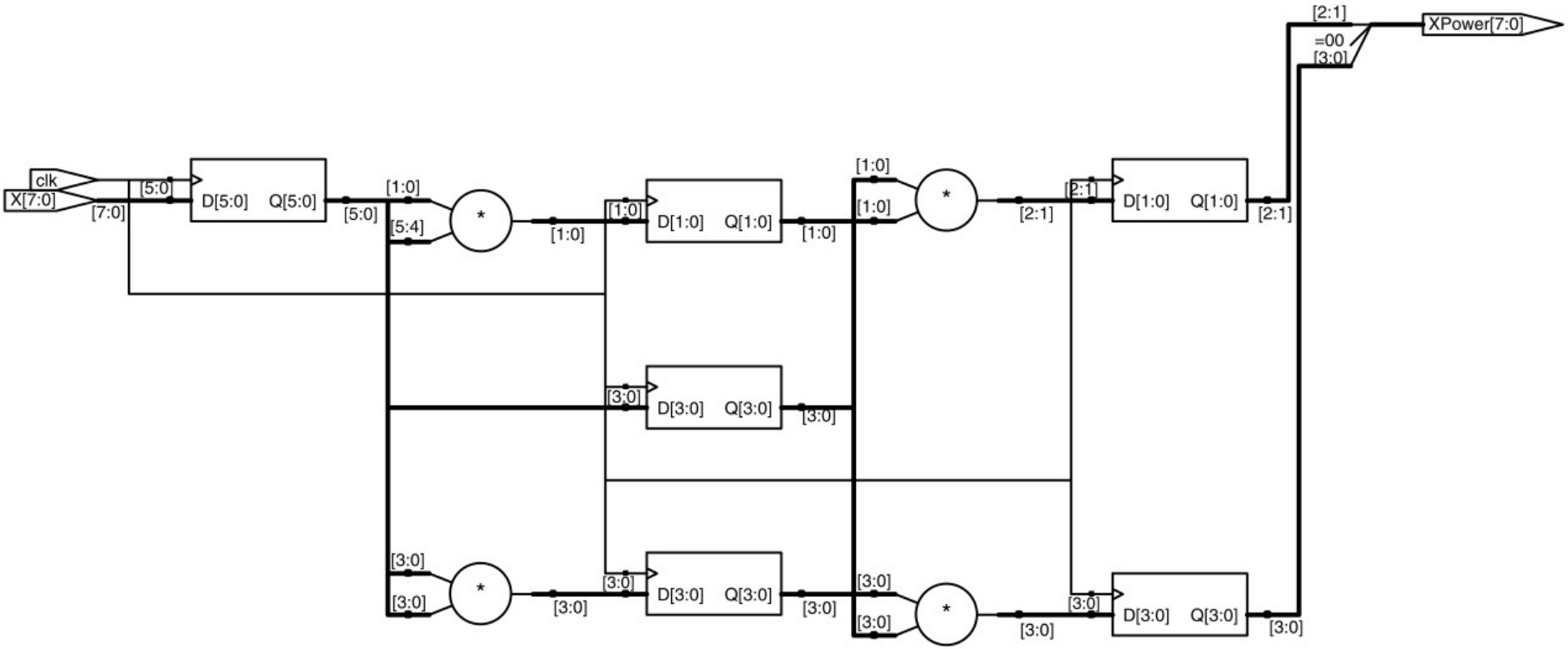
How to Improve Timing? 2

1. The second strategy for architectural timing improvements is to reorganize the critical path such that logic structures are implemented in parallel.
2. This technique should be used whenever a function that currently evaluates through a serial string of logic can be broken up and evaluated in parallel.

Recall


clk
X[7:0]
[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]
[7:0]
[7:0]
*
[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]
[7:0]
[7:0]
*
[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]
[7:0]
XPower[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]

Parallelize



Separating a logic function into a number of smaller functions that can be evaluated in parallel reduces the path delay to the longest of the substuctures.

How to Improve Timing? 3

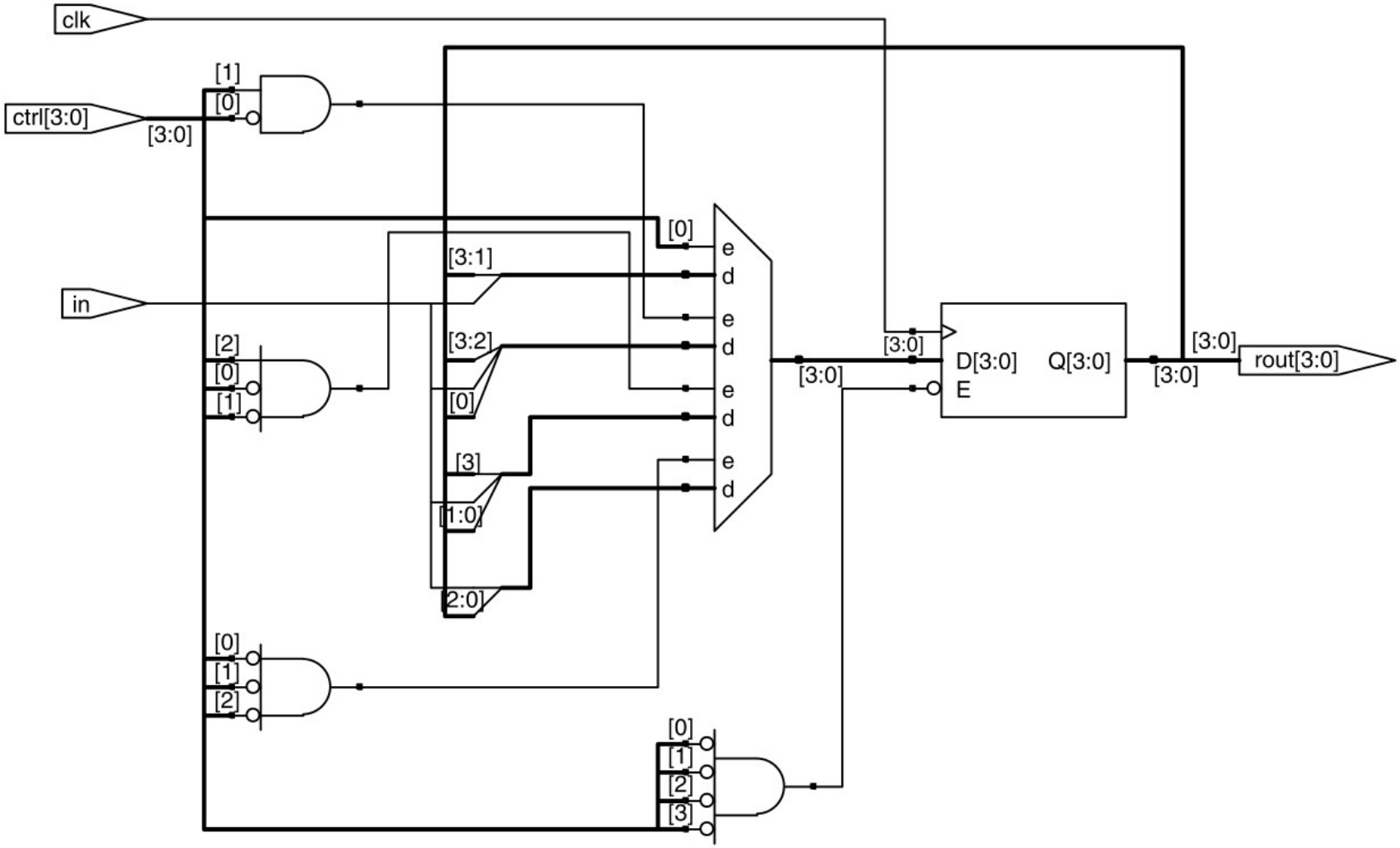
1. The third strategy for architectural timing improvements is to flatten logic structures.
2. This is closely related to the idea of parallel structures, but applies specifically to logic that is chained due to priority encoding. Typically, synthesis and layout tools are smart enough to duplicate logic to reduce fanout, but they are not smart enough to break up logic structures that are coded in a serial fashion, nor do they have enough information relating to the priority requirements of the design.

Example

```
module regwrite(
   output reg [3:0] rout,
   input            clk, in,
   input      [3:0] ctrl);

   always @(posedge clk)
     if(ctrl[0])      rout[0] <= in;
     else if(ctrl[1]) rout[1] <= in;
     else if(ctrl[2]) rout[2] <= in;
     else if(ctrl[3]) rout[3] <= in;
endmodule
```

Circuit

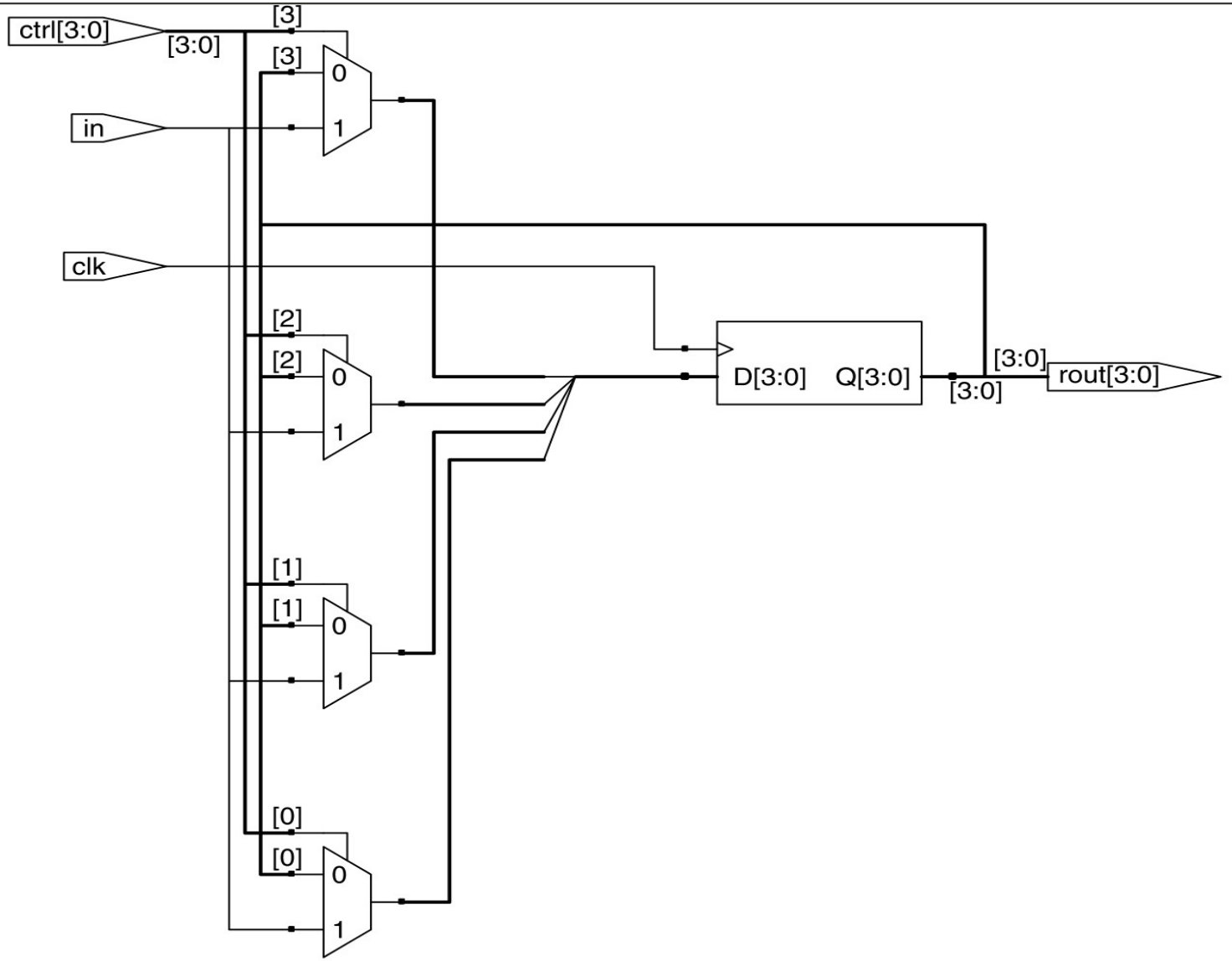

clk
ctrl[3:0]
[3:0]
[1]
[0]
in
[2]
[0]
[1]
[0]
[1]
[2]
[0]
[3:1]
[3:2]
[0]
[3]
[1:0]
[2:0]
[0]
[1]
[2]
[3]
e
d
e
d
e
d
e
d
[3:0]
[3:0]
D[3:0]
E
Q[3:0]
[3:0]
[3:0]
rout[3:0]

Improved

```
module regwrite(
   output reg [3:0] rout,
   input            clk, in,
   input      [3:0] ctrl);

   always @(posedge clk) begin
     if(ctrl[0]) rout[0] <= in;
     if(ctrl[1]) rout[1] <= in;
     if(ctrl[2]) rout[2] <= in;
     if(ctrl[3]) rout[3] <= in;
   end
endmodule
```

Circuit



How to Improve Timing? 4

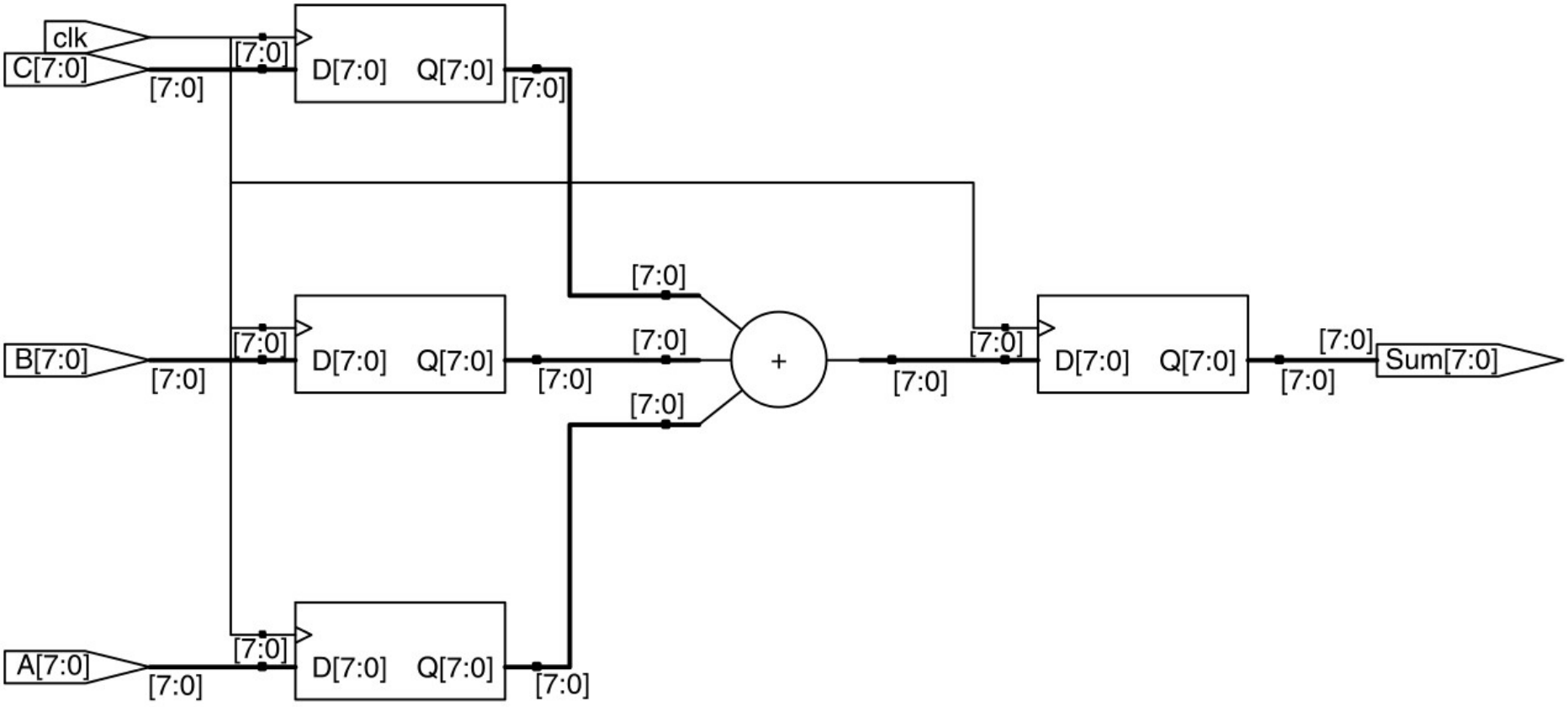
1. The fourth strategy is called register balancing.
2. Conceptually, the idea is to redistribute logic evenly between registers to minimize the worst-case delay between any two registers.
3. This technique should be used whenever logic is highly imbalanced between the critical path and an adjacent path. Because the clock speed is limited by only the worst-case path, it may only take one small change to successfully rebalance the critical logic.

Example

```
module adder(
   output reg [7:0] Sum,
   input      [7:0] A, B, C,
   input            clk);
   reg        [7:0] rA, rB, rC;

   always @(posedge clk) begin
     rA   <= A;
     rB   <= B;
     rC   <= C;
     Sum  <= rA + rB + rC;
   end
endmodule
```

Circuit

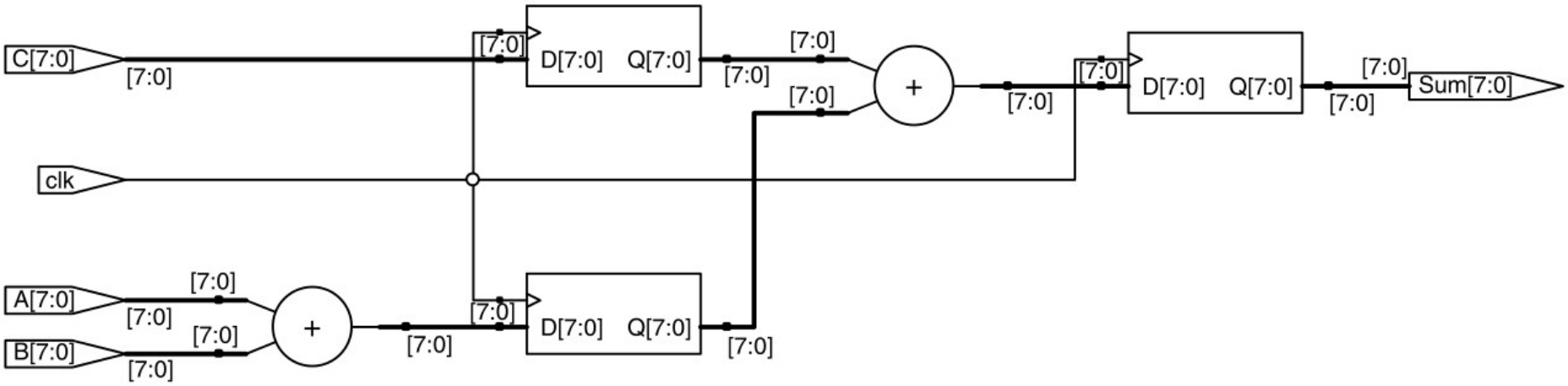

clk
C[7:0]
[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]
B[7:0]
[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]
[7:0]
[7:0]
+
[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]
Sum[7:0]
A[7:0]
[7:0]
[7:0]
D[7:0] Q[7:0]
[7:0]

Improved

```
module adder(
   output reg [7:0] Sum,
   input      [7:0] A, B, C,
   input            clk);
   reg        [7:0] rABSum, rC;

   always @(posedge clk) begin
      rABSum <= A + B;
      rC     <= C;
      Sum    <= rABSum + rC;
   end
endmodule
```

Improved



Register balancing improves timing by moving combinatorial logic from the critical path to an adjacent path.

How to Improve Timing? 5

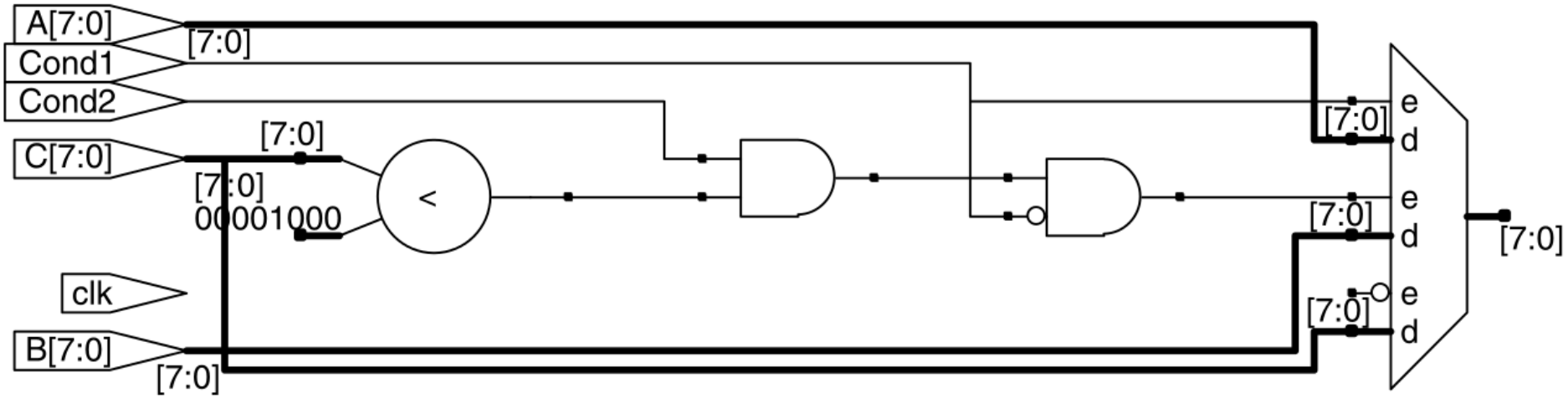
1. The fifth strategy is to reorder the paths in the data flow to minimize the critical path.
2. This technique should be used whenever multiple paths combine with the critical path, and the combined path can be reordered such that the critical path can be moved closer to the destination register.
3. With this strategy, we will only be concerned with the logic paths between any given set of registers.

Example

```
module randomlogic(
  output reg [7:0] Out,
  input      [7:0] A, B, C,
  input            clk,
  input            Cond1, Cond2);

  always @(posedge clk)
    if(Cond1)
    Out <= A;
  else if(Cond2 && (C < 8))
    Out <= B;
  else
    Out <= C;
endmodule
```

Circuit

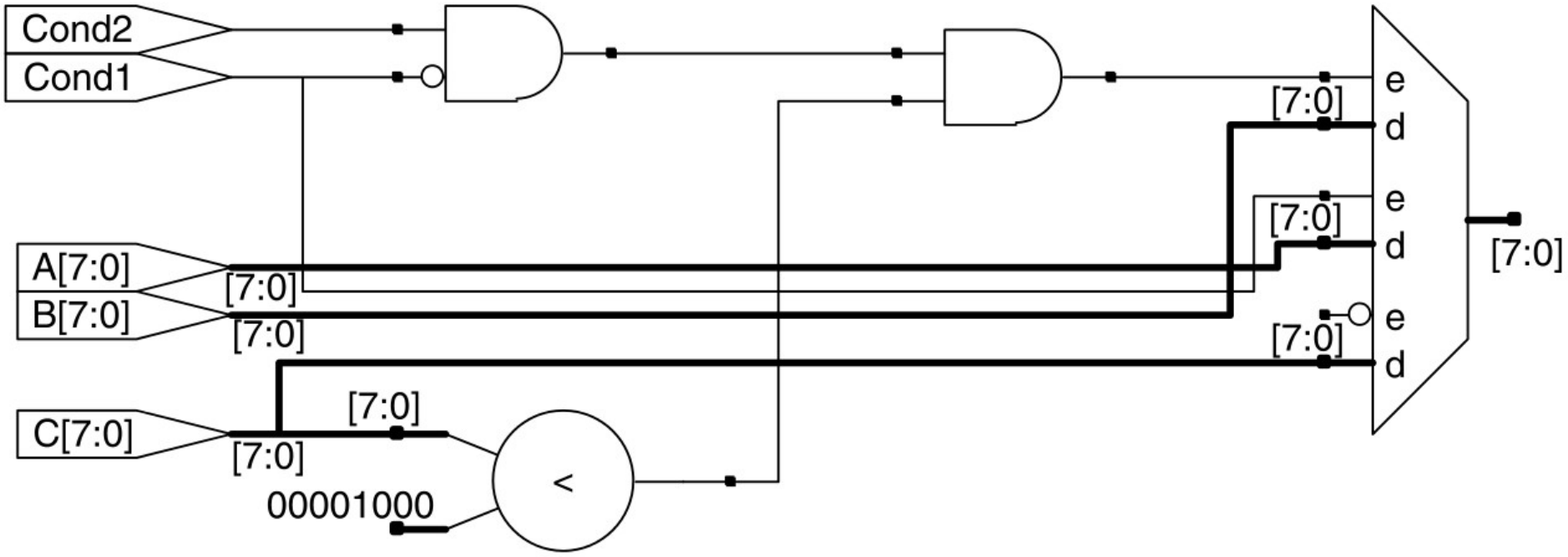

A[7:0]
Cond1
Cond2
C[7:0]
clk
B[7:0]
[7:0]
[7:0]
[7:0]
00001000
<
[7:0]
[7:0]
[7:0]
e
d
e
d
e
d
[7:0]

Improved

```
module randomlogic(
   output reg [7:0] Out,
   input      [7:0] A, B, C,
   input            clk,
   input            Cond1, Cond2);
   wire CondB = (Cond2 & !Cond1);

   always @(posedge clk)
     if(CondB && (C < 8))
       Out <= B;
     else if(Cond1)
       Out <= A;
     else
       Out <= C;
endmodule
```

Circuit



Timing can be improved by reordering paths that are combined with the critical path in such a way that some of the critical path logic is placed closer to the destination register.

Final issues

- Please fill out the student info sheet before leaving
- Come by my office hours (right after class)
- Any questions or concerns?