



AN HONORS UNIVERSITY IN MARYLAND

Department of Computer Science and Electrical Engineering

CMPE 415

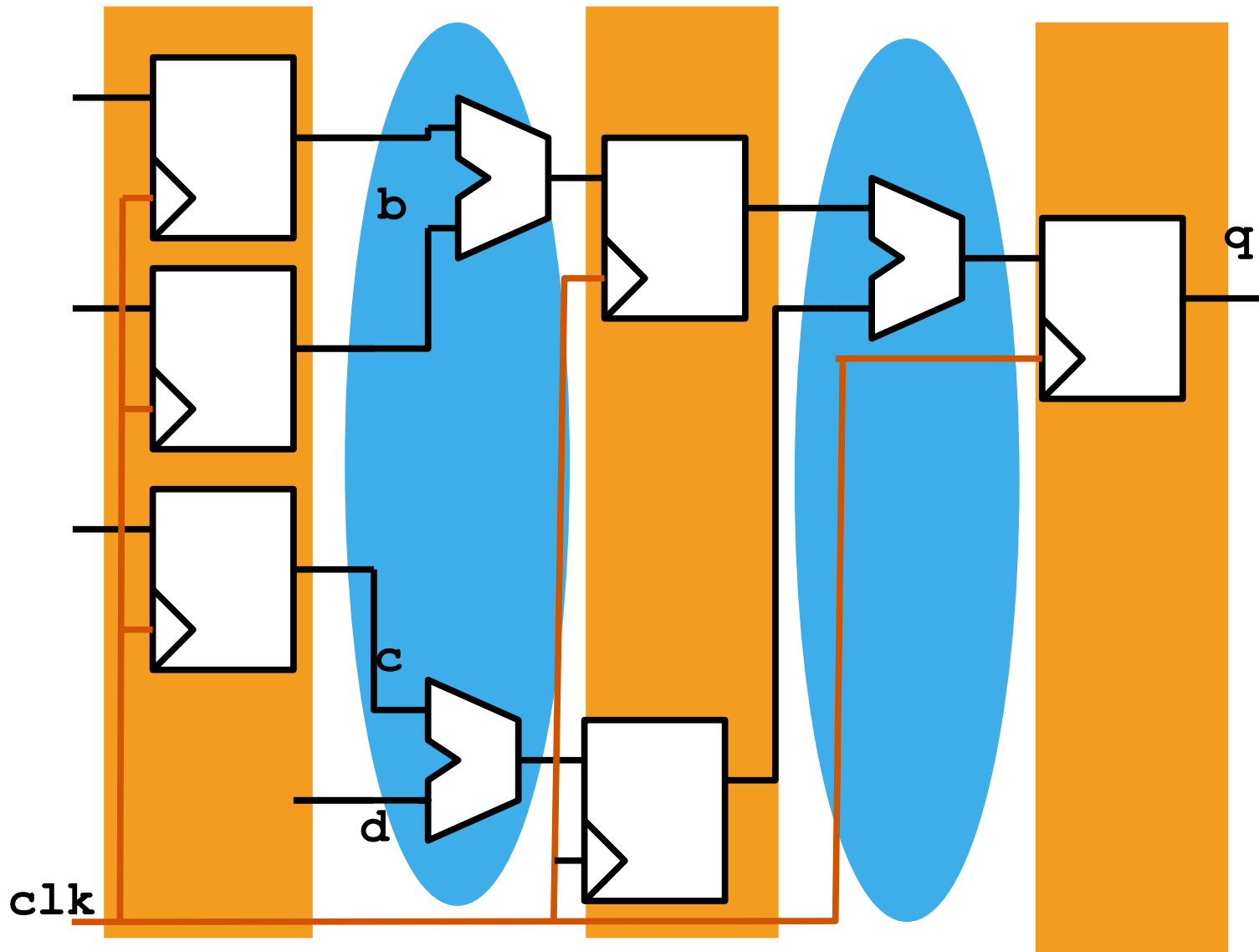
Suggested Coding and Design Practices

Prof. Ryan Robucci

Some examples and discussion are cited from Clifford E. Cummings' papers

<http://www.sunburst-design.com/papers>

Data Pipeline



Rule: No Continuous Feedback

Continuous assignments and logic should not have feedback unless you are adding delays and writing simulation-only modeling code

(i.e. not for synthesis...combinatorial logic reduction would be very confusing in this case)

Primitives

```
wire y, a, b, c;  
and (y, a, b, c)
```

Continuous assignments

```
wire y, a, b, c  
assign y=a&b&c;
```

```
wire y, d0, d1, sel  
assign y=sel?d1:d0;
```

Feedback

```
y=y&a&b&c;
```

```
y=a&b;
```

```
b=y&a;
```

```
y= (en & x) | (~en & y) ;
```

```
y=en?x:y;
```

```
nor n1 (q, r, q_n) ;
```

```
nor n1 (q_n, s, q) ;
```

SR Latch

```
q=~ (r | q_n) ;
```

```
q_n=~ (s | q) ;
```

Basic Register

A basic register

```
reg q;  
always @ (posedge clk)  
begin  
    q <= d;  
end
```

The sensitivity list includes only a timing control signal, the clock. Output is only updated on clock edges even if d is changing, this requires memory.

Register w/ Synchronous Reset/Clear

Synchronous reset

```
reg q;  
always @ (posedge clk)  
    if (reset)  
        q <= 0;  
    else  
        q <= d;
```

The sensitivity list only includes the clock, indicating that the set and clear only propagate to the output upon the rising clock edge.

Synchronous reset (active low)

```
reg q;  
always @ (posedge clk)  
    if (reset_n)  
        q <= 0;  
    else  
        q <= d;
```

Reg. w/ Asynchronous Reset/Clear

Asynchronous reset

```
reg q;  
always @(posedge clk, posedge clear )  
    if(reset)  
        q <= 0;  
    else  
        q <= d;
```

- The sensitivity list includes clear signal, indicating the clear should propagate to the output immediately. The edge specifier in the sensitivity list is required by some synthesizers, though not required for simulation.

Asynchronous reset (active low)

```
reg q;  
always @(posedge clk, negedge clear_n )  
    if(reset_n)  
        q <= 0;  
    else  
        q <= d;
```

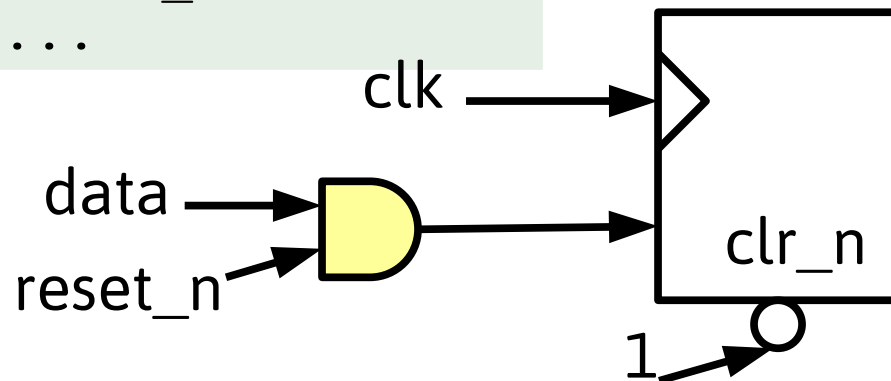
- As a general rule, review the synthesis tool's documentation regarding recommended coding templates. Some synthesizers are more flexible than others. Some may for instance require that the reset condition be handled first with an if else construct as shown.

Register Synthesis: Resets

- Often asynchronous resets are more efficient than synchronous resets since the inbuilt technology registers often have async. resets on them and synchronous resets would involve something like and and gate in the path of the input data
- Sometimes it is required that async. Reset be handle by if statements immediate following trigger statement

```
always @ (posedge clk, negedge clr_n)  
  if (clear==0) ...  
  else ...
```

```
always @ (posedge clk)  
  if (reset_n==0) ...  
  else ...
```



Note use of
negedge

Registered-Output Logic

Combinatorial Only includes all inputs:

```
reg y;  
always @ (a, b) //all comb. dependencies listed  
    y = a & b;
```

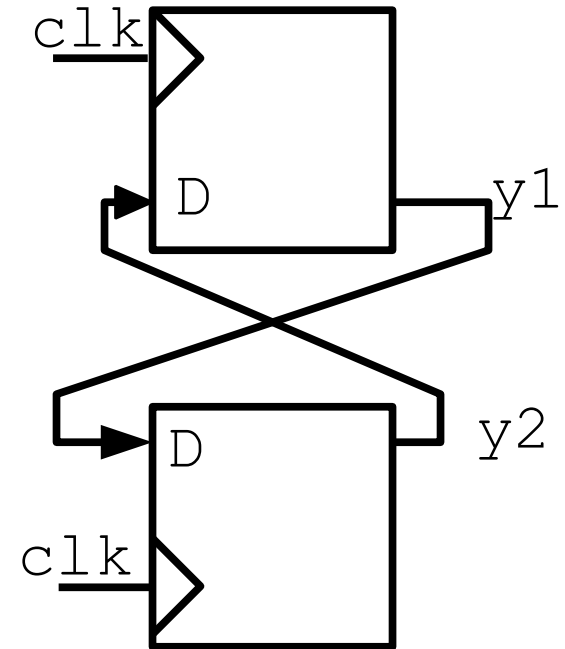
Could also have used `y<= a & b`; but we follow the practice of using blocking assignments for all combinatorial logic

Sequential (registered-output combinatorial logic):

```
reg q;  
always @ (posedge clk)  
    q <= a & b;
```


Bad Parallel Blocks

```
module fbosc1 (y1, y2, clk, rst);  
  output y1, y2;  
  input clk, rst;  
  reg y1, y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y1 = 0; // reset  
    else y1 = y2;  
  
  always @(posedge clk or posedge rst)  
    if (rst) y2 = 1; // preset  
    else y2 = y1;  
endmodule
```



This code attempts to model a swap of y1 and y2
Timing of execution of parallel always blocks is not guaranteed
in simulation – though synthesis will probably work since
synthesis approaches each always blocks somewhat
independently at first

Simulation of parallel blocks



Which one first? Does it even matter?

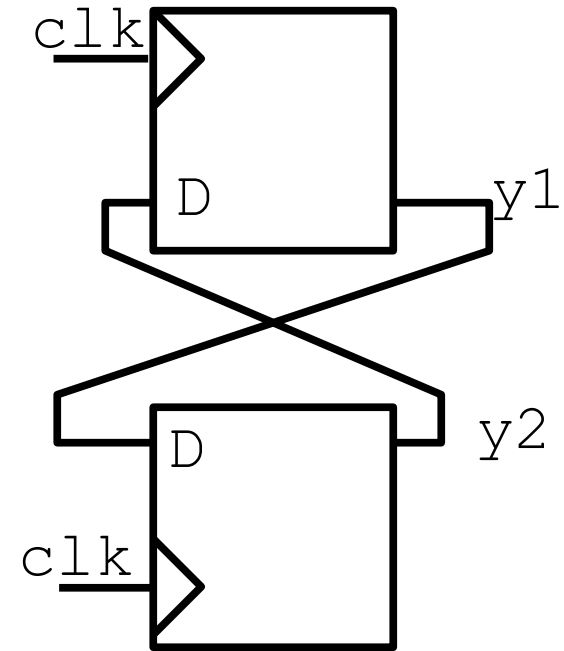
```
always @(posedge clk,  
        posedge rst)  
    if (rst) y2 = 1; // preset  
    else y2 = y1;
```

```
always @(posedge clk,  
        posedge rst)  
    if (rst) y1 = 0; // reset  
    else y1 = y2;
```

Good Parallel Blocks

Will not only synthesize correctly, but also simulate correctly:

```
module fbosc2 (y1, y2, clk, rst);  
    output y1, y2;  
    input clk, rst;  
    reg y1, y2;  
  
    always @(posedge clk or posedge rst)  
        if (rst) y1 <= 0; // reset  
        else y1 <= y2;  
  
    always @(posedge clk or posedge rst)  
        if (rst) y2 <= 1; // preset  
        else y2 <= y1;  
endmodule
```



Sim Example (Header)

```
module ambiguous_parallel_swap();  
    reg clk, rst;  
    reg y1, y2;  
    reg z1, z2;  
  
    initial clk = 0;  
    always #50 clk = ~clk;  
  
    initial begin  
        rst = 1;  
        #10;  
        rst = 0;  
    end  
  
    initial begin  
        #1000 $finish;  
    end
```

Sim Example (body code)

```
always @(posedge clk , posedge rst)
  if (rst) y1 = 0; // reset
  else y1 = y2;
```

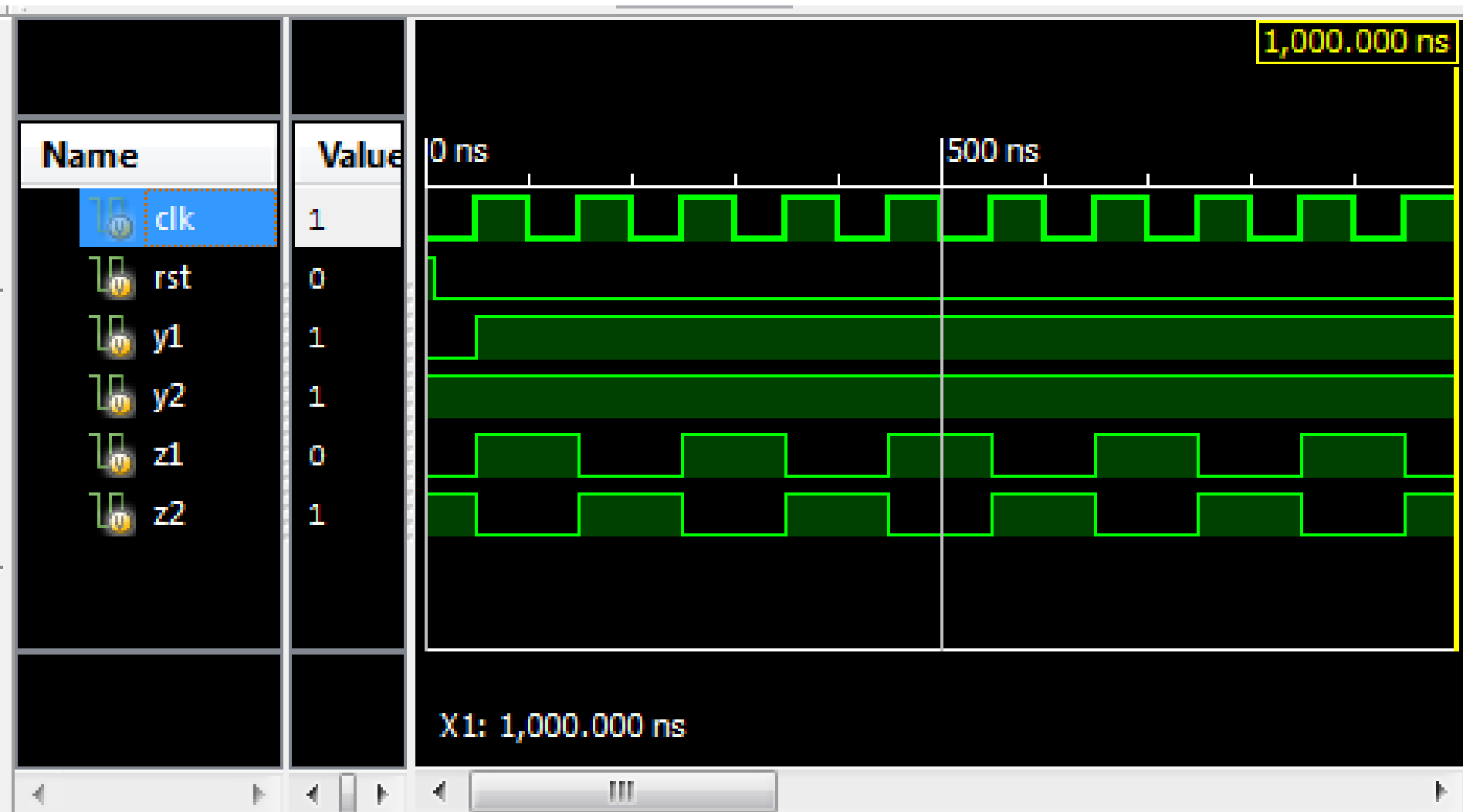
```
always @(posedge clk , posedge rst)
  if (rst) y2 = 1; // preset
  else y2 = y1;
```

```
always @(posedge clk , posedge rst)
  if (rst) z1 <= 0; // reset
  else z1 <= z2;
```

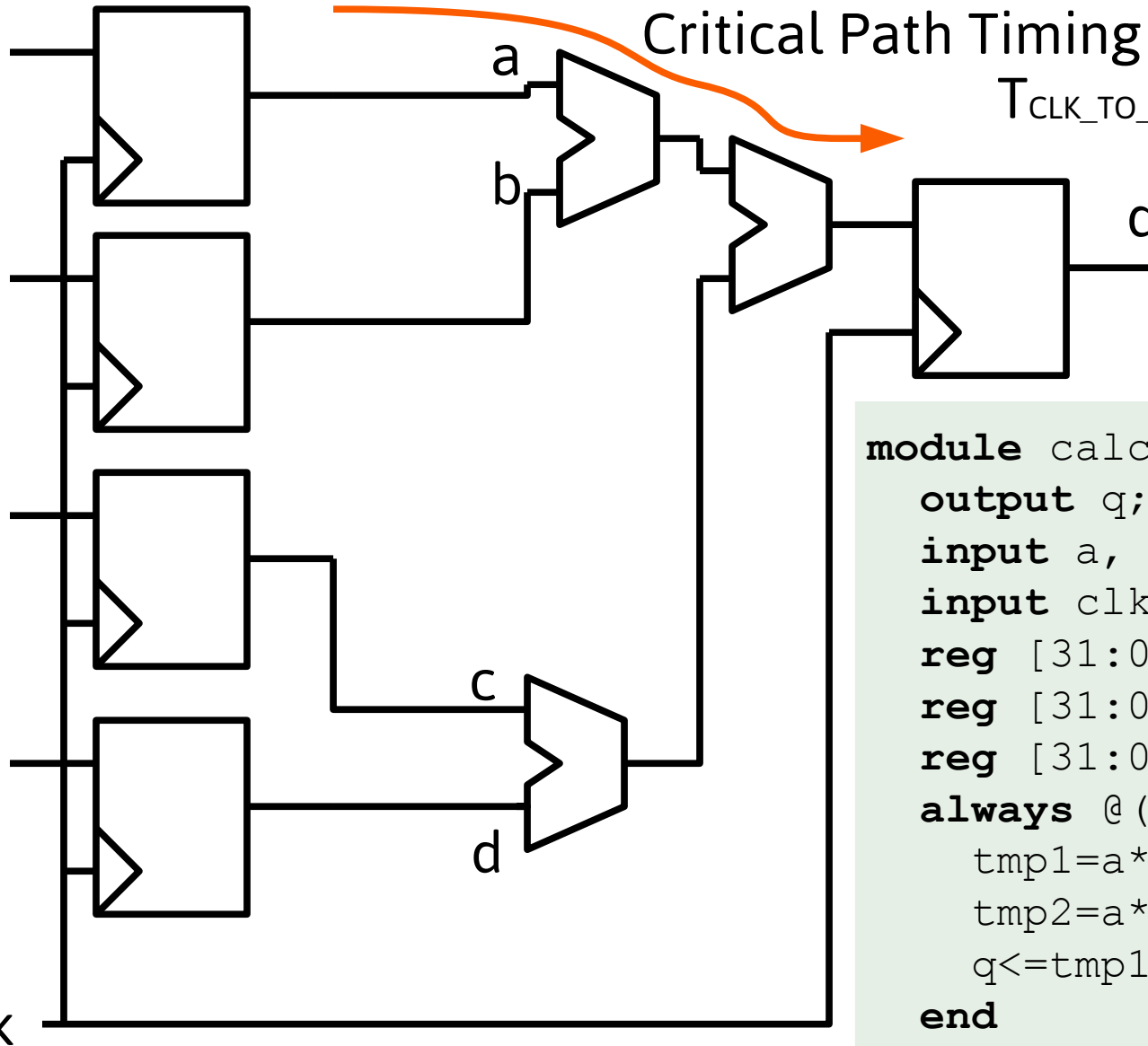
```
always @(posedge clk , posedge rst)
  if (rst) z2 <= 1; // preset
  else z2 <= z1;
```

```
endmodule
```

Sim Example (results)



Intentional Pipeline for Timing



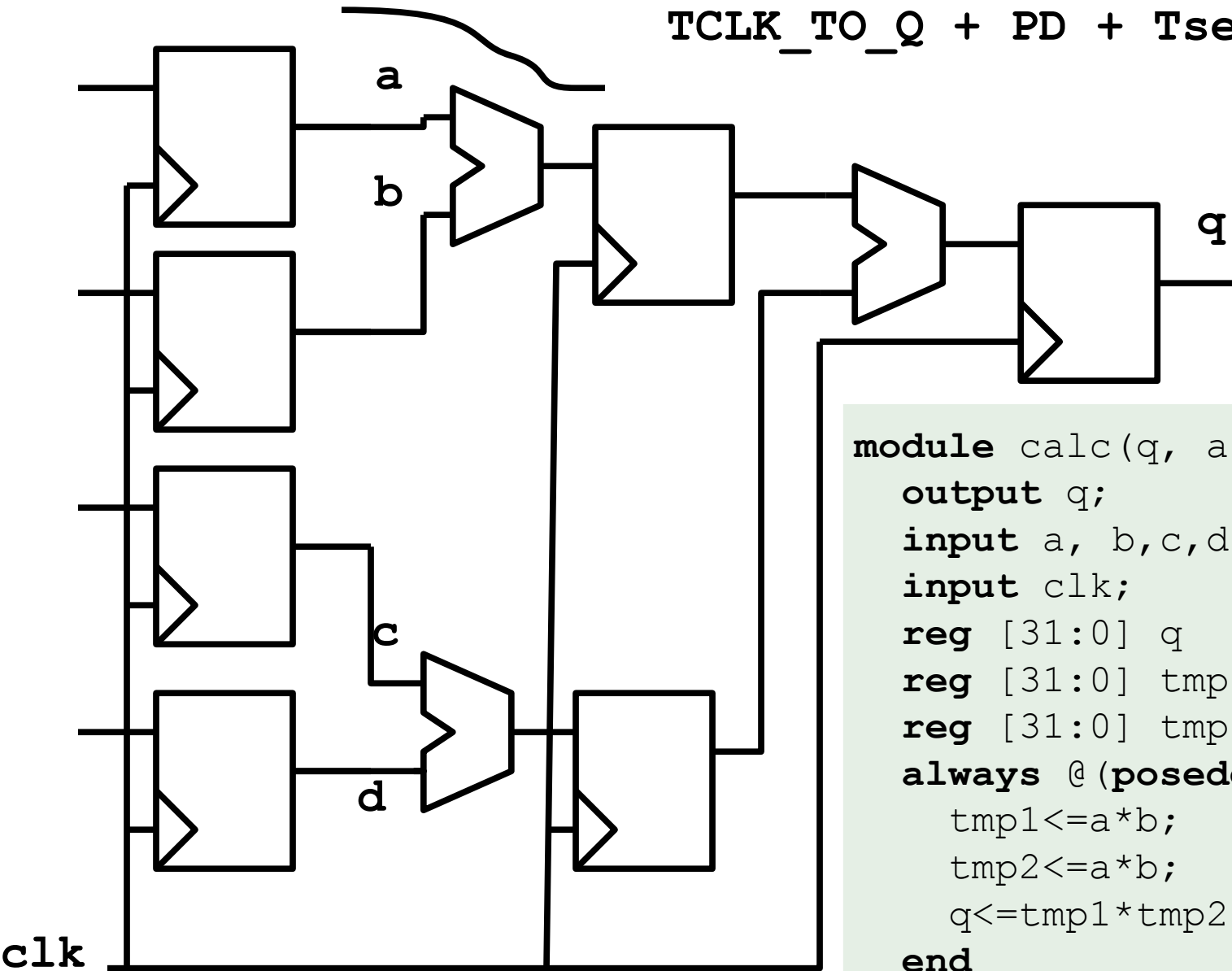
What if
timing requirement
is not satisfied?

```
module calc(q, a, b, c, d, clk);  
    output q;  
    input a, b, c, d;  
    input clk;  
    reg [31:0] q;  
    reg [31:0] tmp1;  
    reg [31:0] tmp2;  
    always @(posedge clk) begin  
        tmp1 = a * b;  
        tmp2 = c * d;  
        q <= tmp1 * tmp2;  
    end  
endmodule
```

- Can reduce the clock speed
 - But this slows the entire system
- Can introduce pipelining
 - Overall propagation of computation is longer (two clock cycles incurring multiple setup and hold times)
 - Maintains fast system clock
- Alternatively states, may be able to introduce pipelining in the critical path of a system in order to increase the clock rate and therefore overall system throughput

Critical Path

$$T_{CLK_TO_Q} + PD + T_{setup} < T_{clk}$$



```
module calc(q, a, b, c, d, clk);  
    output q;  
    input a, b, c, d;  
    input clk;  
    reg [31:0] q;  
    reg [31:0] tmp1;  
    reg [31:0] tmp2;  
    always @(posedge clk) begin  
        tmp1 <= a * b;  
        tmp2 <= c * d;  
        q <= tmp1 * tmp2;  
    end  
endmodule
```

pipeline

Data Pipeline

```
module pipeb2(q3,,clk);  
output [7:0] q3;  
input [7:0] d; input clk;  
reg [7:0] q3, q2, q1;  
always @(posedge clk) begin  
    q3 = q2;  
    q2 = q1;  
    q1 = d;  
end  
endmodule
```

```
module pipeb1 (q3, d, clk);  
    output [7:0] q3;  
    input [7:0] d; input clk;  
    reg [7:0] q3, q2, q1;  
    always @(posedge clk) begin  
        q1 = d;  
        q2 = q1;  
        q3 = q2;  
    end  
endmodule
```

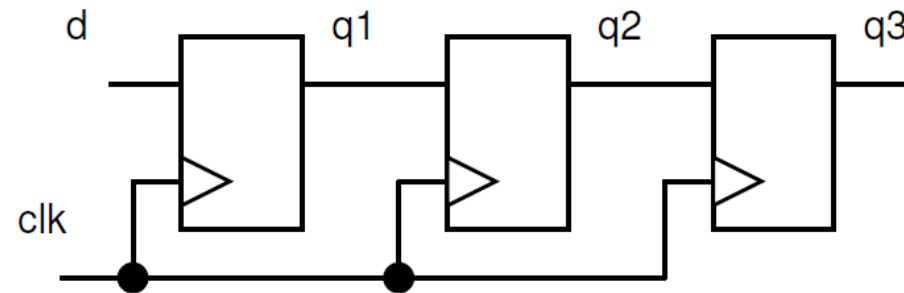


Figure 2 - Sequential pipeline register

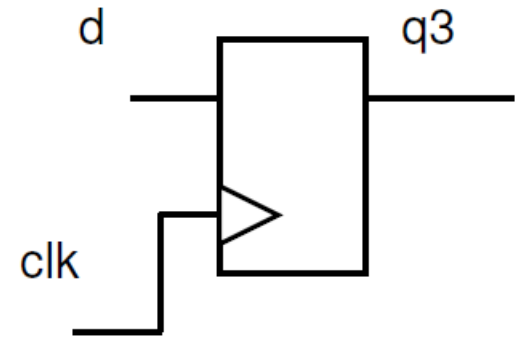


Figure 3 - Actual synthesized result!

Bad Parallel Block Pipeline Implementation

```
module pipeb3 (q3, d, clk);  
    output [7:0] q3;  
    input [7:0] d;  
    input clk;  
    reg [7:0] q3, q2, q1;  
    always @(posedge clk) q1=d;  
    always @(posedge clk) q2=q1;  
    always @(posedge clk) q3=q2;  
endmodule
```

```
module pipeb4 (q3, d, clk);  
    output [7:0] q3;  
    input [7:0] d;  
    input clk;  
    reg [7:0] q3, q2, q1;  
    always @(posedge clk) q2=q1;  
    always @(posedge clk) q3=q2;  
    always @(posedge clk) q1=d;  
endmodule
```

These may synthesize correctly, but simulation may not match

Good Pipeline Implementations

Use non-blocking statements for registers

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input [7:0] d;
  input clk;
  reg [7:0] q3, q2, q1;
  always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

★ Order doesn't matter

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input [7:0] d;
  input clk;
  reg [7:0] q3, q2, q1;
  always @(posedge clk) begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
  end
endmodule
```

★

```
module pipen4 (q3, d, clk);
  output [7:0] q3;
  input [7:0] d;
  input clk;
  reg [7:0] q3, q2, q1;
  always @(posedge clk) q2<=q1;
  always @(posedge clk) q3<=q2;
  always @(posedge clk) q1<=d;
endmodule
```

★

```
module pipen3 (q3, d, clk);
  output [7:0] q3;
  input [7:0] d;
  input clk;
  reg [7:0] q3, q2, q1;
  always @(posedge clk) q1<=d;
  always @(posedge clk) q2<=q1;
  always @(posedge clk) q3<=q2;
endmodule
```

★

Cascading Combinatorial Logic

Ex: AND-OR

```
module ao4 (y, a, b, c, d);  
output y;  
input a, b, c, d;  
reg y, tmp1, tmp2;  
always @(a or b or c or d) begin  
    tmp1 <= a & b;  
    tmp2 <= c & d;  
    y <= tmp1 | tmp2;  
end  
endmodule
```



guideline avoids this problem

Guideline: When modeling combinatorial logic with an always block, use blocking assignments.

- y reflects old values
- y may not be updated correctly until next change triggers another evaluation

- Works, but requires multiple passes in simulation

```
module ao2 (y, a, b, c, d);  
output y;  
input a, b, c, d;  
reg y, tmp1, tmp2;  
always @(a or b or c or d) begin  
    tmp1 = a & b;  
    tmp2 = c & d;  
    y = tmp1 | tmp2;  
end  
endmodule
```



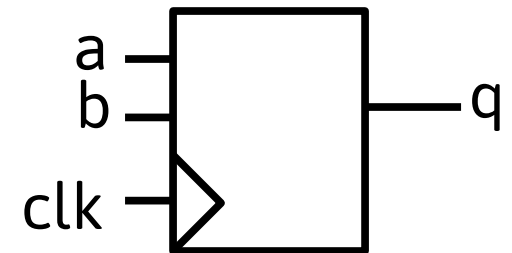
- efficient sim

```
module ao5 (y, a, b, c, d);  
output y;  
input a, b, c, d;  
reg y, tmp1, tmp2;  
always @(a,b,c,d,tmp1,tmp2)  
begin  
    tmp1 <= a & b;  
    tmp2 <= c & d;  
    y <= tmp1 | tmp2;  
end  
endmodule
```

Mixing Comb. and Sequential Example: xor-DFF

```
module nbex1 (q, a, b, clk, rst_n);  
  output q;  
  input clk, rst_n;  
  input a, b;  
  reg q, y;  
  always @(a or b)  
    y = a ^ b;  
  
  always @(posedge clk or negedge rst_n)  
    if (!rst_n) q <= 1'b0;  
    else q <= y;  
endmodule
```

Separation of sequential
And combinatorial



```
module nbex2 (q, a, b, clk, rst_n);  
  output q;  
  input clk, rst_n;  
  input a, b;  
  reg q;  
  always @(posedge clk or negedge rst_n)  
    if (!rst_n) q <= 1'b0;  
    else q <= a ^ b;  
endmodule
```

mix

Mixing Comb and Sequential Example: xor-DFF

Use of local declarations for temporary and trimmed signals

```
module ba_nba2 (q, a, b, clk, rst_n);  
  output q;  
  input a, b, rst_n;  
  input clk;  
  reg q;  
  always @(posedge clk or negedge rst_n) begin: ff  
    reg tmp;  
    if (!rst_n)  
      q <= 1'b0;  
    else begin  
      tmp = a & b;  
      q <= tmp;  
    end  
  end  
endmodule
```

Required Block name

Recommend Coding:
Local variable declared in a
named block allowed in Xilinx ISE*
Prevents accidental use outside block

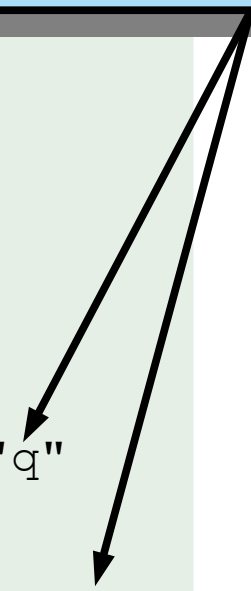
Mix of blocking for intermediate/
combinatorial logic and non-blocking
for sequential

*WARNING:Xst:646 - Signal <ff/tmp> is assigned but never used. This unconnected signal will be trimmed during the optimization process.

DFF: Poor and possibly unsupported style

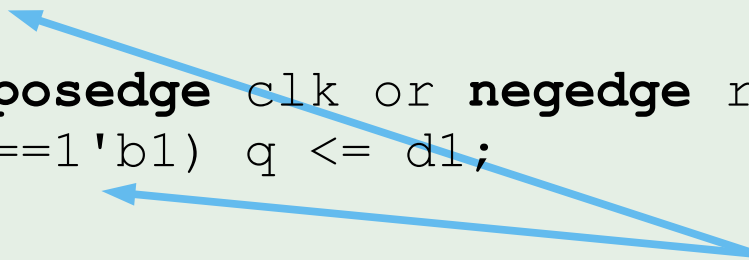
Mix of blocking and non-blocking
To same variable.

```
module ba_nba6 (q, a, b, clk, rst_n);  
  output q;  
  input a, b, rst_n;  
  input clk;  
  reg q, tmp;  
  always @(posedge clk or negedge rst_n)  
  if (!rst_n)  
    q = 1'b0; // blocking assignment to "q"  
  else begin  
    tmp = a & b;  
    q <= tmp; // nonblocking assignment to "q"  
  end  
endmodule
```



Assignments to the same variable from multiple always blocks

```
module badcode1 (q, d0, d1, sel, clk,);  
  output q;  
  input d0, d1, clk, rst_n;  
  reg q;  
  
  always @(posedge clk or negedge rst_n)  
    if (sel==1'b0) q <= d0;  
  
  always @(posedge clk or negedge rst_n)  
    if (sel==1'b1) q <= d1;  
endmodule
```



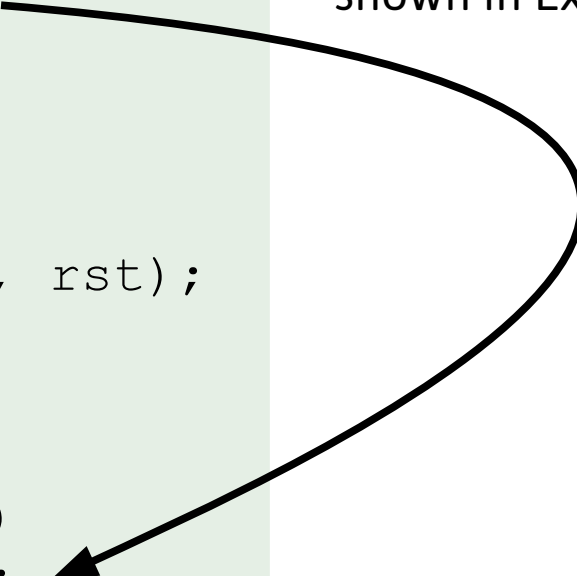
Mutually exclusive
cases

These blocks are make mutually exclusive assignments
May make sense. May sim, but synth. usually complains
of multiple drivers.

Guideline: Use non-blocking for EVERY register

```
module dffb (q, d, clk, rst);  
    output q;  
    input d, clk, rst;  
    reg q;  
    always @(posedge clk)  
        if (rst) q = 1'b0;  
        else q = d;  
endmodule
```

```
module dffx (q, d, clk, rst);  
    output q;  
    input d, clk, rst;  
    reg q;  
    always @(posedge clk)  
        if (rst) q <= 1'b0;  
        else q <= d;  
endmodule
```

- It is better to develop the habit of coding all sequential always blocks, even simple single-block modules, using nonblocking assignments as shown in Example 14.
- 

Swapping Example

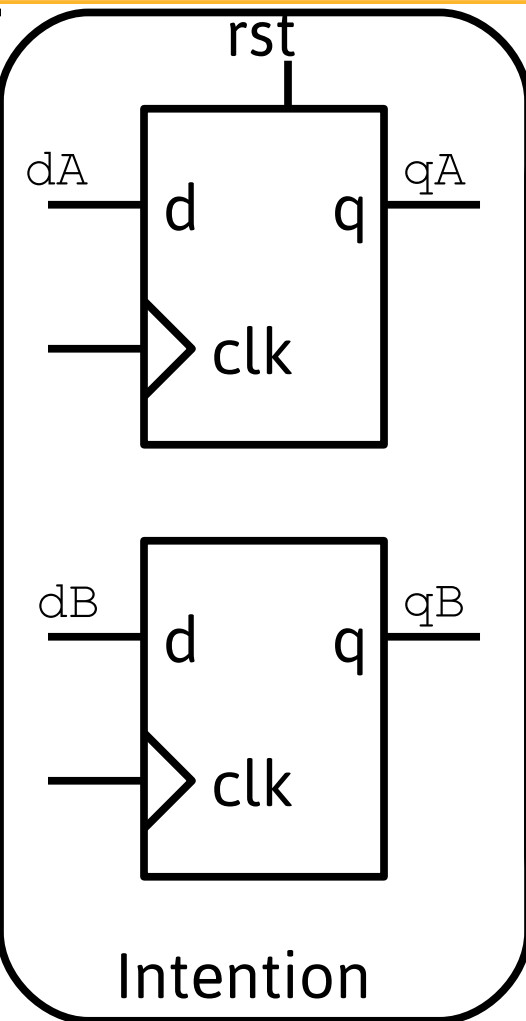
Non-blocking

```
always @(posedge clk , posedge rst) begin
    if (rst) begin
        z1 <= 0; // reset
    end else begin
        z1 <= z2; } Order doesn't matter
        z2 <= z1;
    end
end
```

Blocking

```
always @(posedge clk , posedge rst) begin
    if (rst) begin
        y1 <= 0; // reset
    end else begin
        temp = y1; ← Temp was declared as
        y1 = y2;      reg and doesn't exist in
        y2 <= temp;   synthesized design
    end
end
```

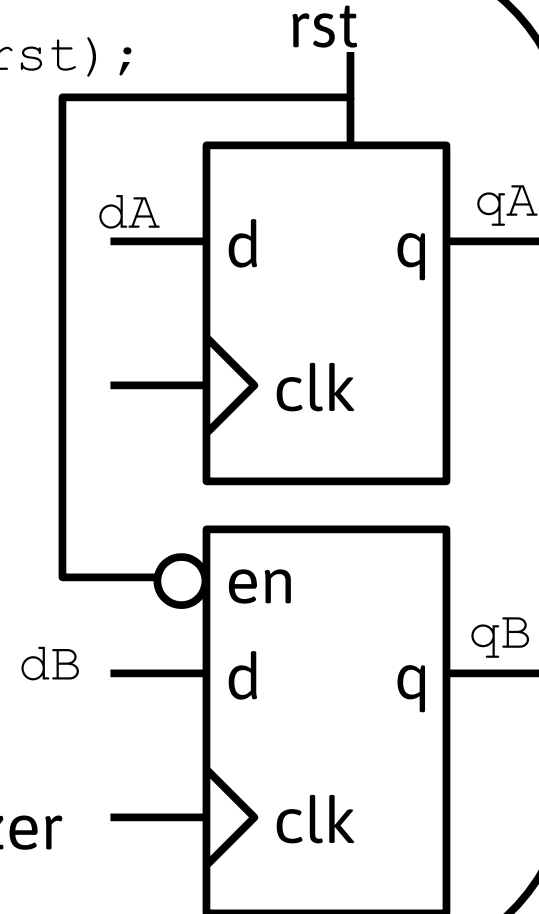
General warning for Sloppy combinations



```
module dff2 (q, d, clk, rst);  
  output q;  
  input d, clk, rst;  
  reg q;  
  always @(posedge clk)  
    if (rst) begin  
      qA <= 1'b0;  
    end else begin  
      qA <= dA;  
      qB <= dB;  
    end  
endmodule
```

Code

What
Synthesizer
sees

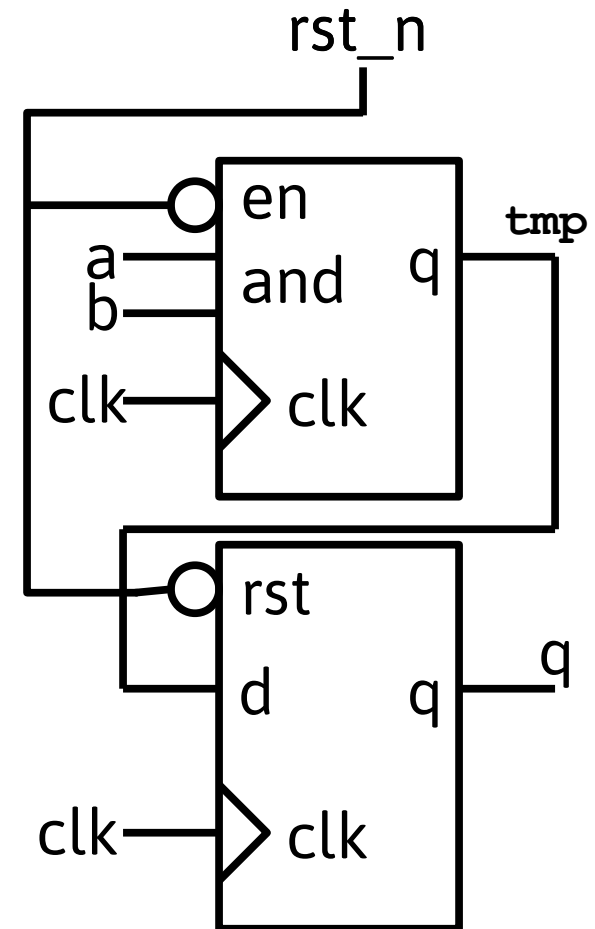


Moral of the story: Be very careful to consider every output for every path in the decision tree.

Accidental Pipeline

```
module andReg(q, a, b, clk, rst_n);  
  output q;  
  input a, b, rst_n;  
  input clk;  
  reg q, tmp;  
  always @(posedge clk , negedge rst_n)  
  if (!rst_n)  
    q <= 1'b0;  
  else begin  
    tmp <= a & b;  
    q <= tmp;  
  end  
endmodule
```

Non-blocking assignment
breaks our rules



Multiple Clock Domains

As a beginner, avoid the creation of additional clock domains caused by using various signals as a clock

A basic register

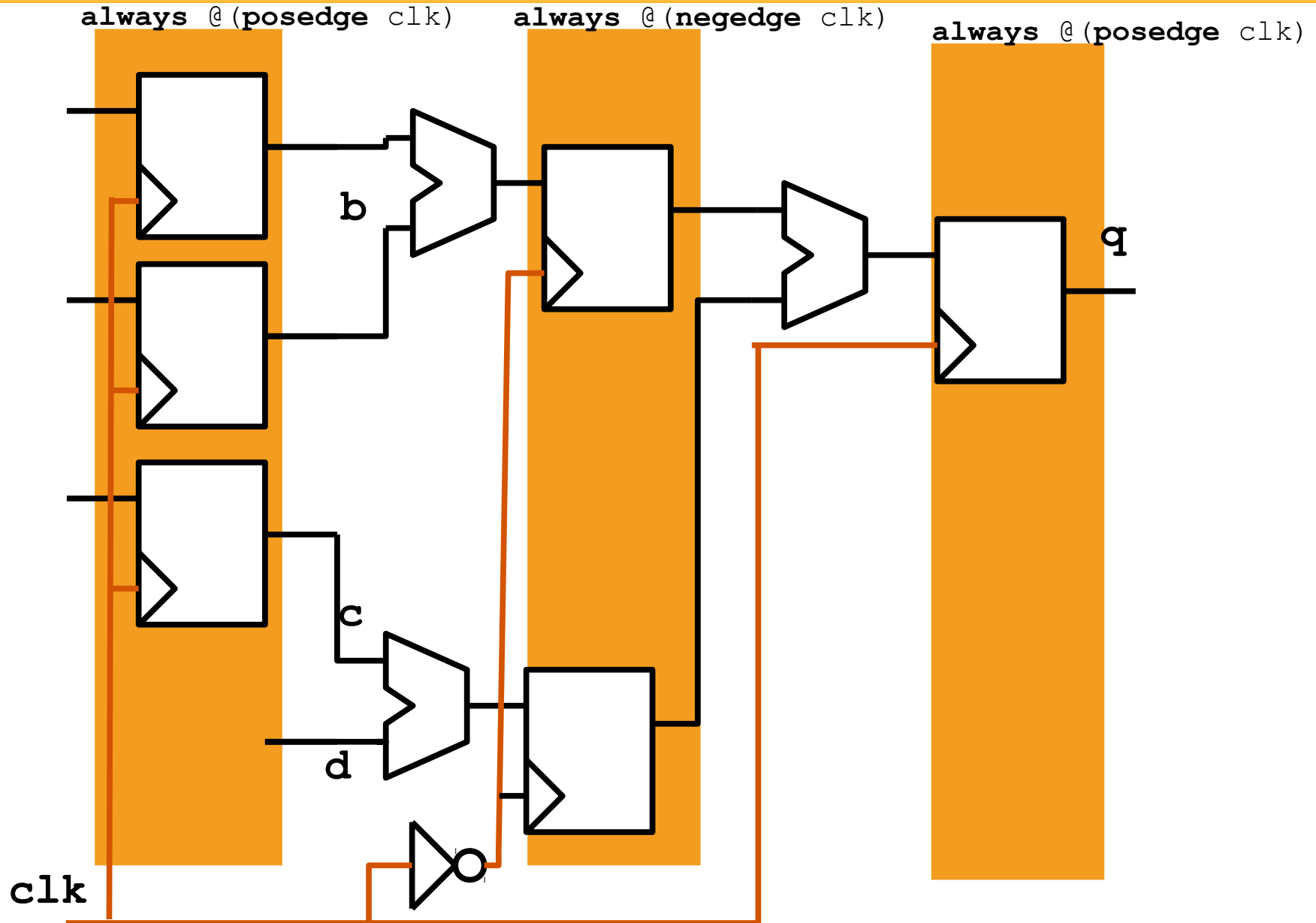
```
reg q;  
always @ (posedge clk)  
begin  
    q <= d;  
end
```

Use of a logic signal as a clock

```
assign gt = a>b;  
reg q;  
always @ (posedge gt)  
begin  
    a <= b;  
end
```

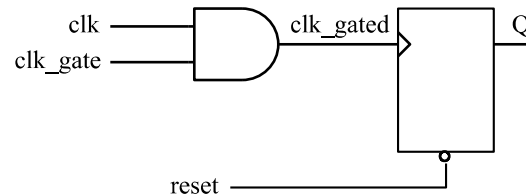
A synthesizer may wish to partition its task by organizing logic into clock domains, optimizing the logic within them, and then performing timing analysis (e.g. critical path propagation delay, setup time and hold time checks) within the domain and attempt to handle signals which **cross** clock domains. Furthermore, FPGAs use a special hardware routing network for clocks that is distinct from general logic signals. Creation of a additional clock domains requires care and should be avoided at this time.

No Double-Edge Clocking in this Course



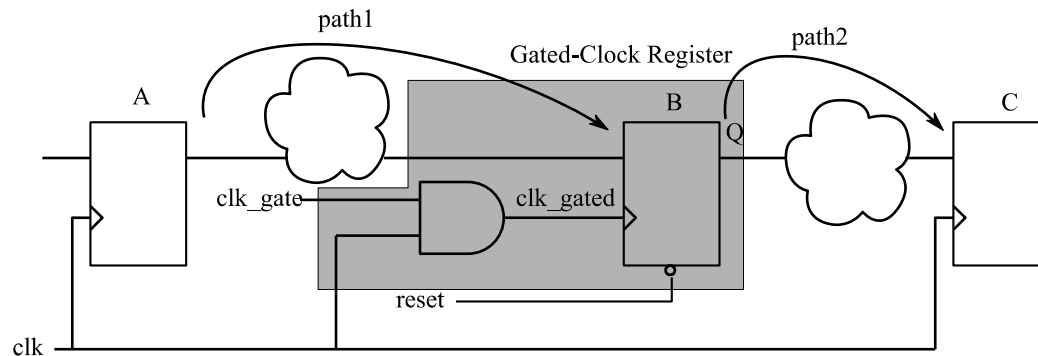
Gated Clocks

- These avoid unnecessary switching in a system and reduce power
- Gating introduces clock skew between parts, complicating timing by introducing **additional clock domains**



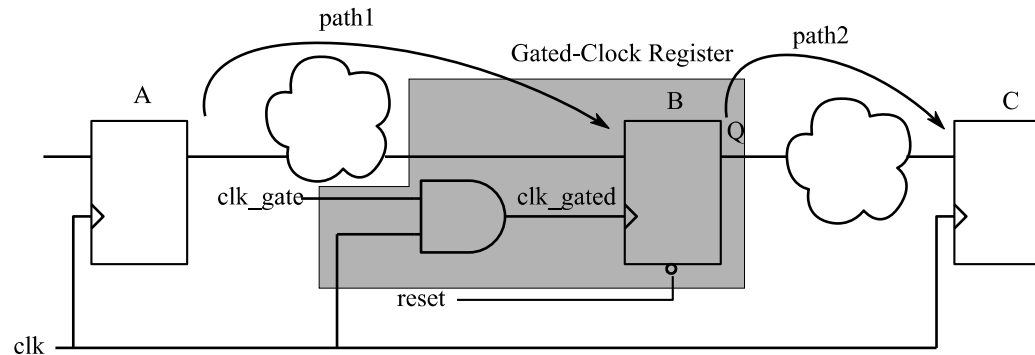
- It is best to use special hardware to create additional clocks which have not yet been taught
- For now, consider it safer to implement an enable
- For those that want to read ahead:
<http://www.xilinx.com/support/answers/38099.html>

Gated Clock Issues



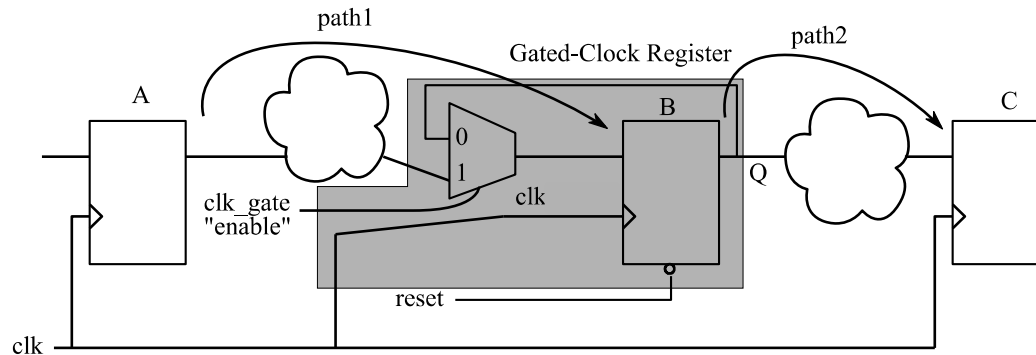
- `clk_gated` is delayed from `clk`
 - Increased opportunity for a race condition from register A to register B
 - Increase delay from `clk` edge to updated Q on B registered reduces allowed propagation time in logic on path2
- Timing analysis and tools must be able to account for this

Gated Clock Issues



```
module gated_clock (clk, reset_, clk_gate, data, Q);  
  input clk, reset_, clk_gate, data;  
  output Q;  
  reg Q.  
  wire clk_gated = clk && clk_gate;  
  always @ (posedge clk_gated , negedge reset_)  
    if (reset_==0) Q<=0; else Q<= data;  
  end  
endmodule
```

Gated Clock Functionality using Enable



- Safer, and simpler for timing analysis and optimization tools to work with

```
module not_gated_clock (clk, reset_, data_gate, data,Q);  
  input clk, reset_,data_gate,data;  
  output Q;  
  reg Q;  
  always @ (posedge clk or negedge reset_)  
    if (reset_==0) Q<=0; else if (data_gate) Q<= data;  
    //else assignment to previous value is inferred  
  end  
endmodule
```