

# EE201L

Verilog Lecture  
by  
Gandhi Puvvada, USC

**always statements,  
Coding a Flip-Flop  
Counters, Basics of Data Path,  
blocking and non-blocking  
assignments**

# *always* statement for clocked logic

- Flip-Flops, Counters, ...
- State Machines, ...
- Data Registers, ...

In short, any clocked logic

posedge clk  
negedge clk

## Logic with no reset

```
always @(posedge clk)
begin    : NO_RESET
    // statements;
end
```

active-high reset

# asynchronous reset

```
always @(posedge Clk, posedge Reset)
begin : ASYNC_R
    if (Reset)
        // statements;
    else
        // statements;
end
```

# synchronous reset

```
always @(posedge Clk)
begin : SYNC_R
    if (Reset)
        // statements;
    else
        // statements;
end
```

# asynchronous reset

active-low reset

```
always @(posedge Clk, negedge Reset)
begin : ASYNC_R
    if (~Reset)           // statements;
    else                   // statements;
end
```

# synchronous reset

```
always @(posedge Clk)
begin : SYNC_R
    if (~Reset)           // statements;
    else                   // statements;
end
```

# Flip-Flop coding

- Q\_no\_r = FF with no reset at all
- Q\_async\_r = FF with asynchronous reset
- Q\_sync\_r = FF with synchronous reset

- Q\_no\_r\_de = FF with Data Enable but no reset at all  
(suitable for data registers)
- Q\_async\_r\_de = FF with asynchronous reset with Data Enable
- Q\_sync\_r\_de = FF with synchronous reset with Data Enable

- Q\_bad\_r = The BAD coding results in treating the RESET as a Data-enable control.
- Q\_bad\_r\_made\_good = BAD coding fixed!

**Q\_no\_r**  
**= FF with no reset at all**

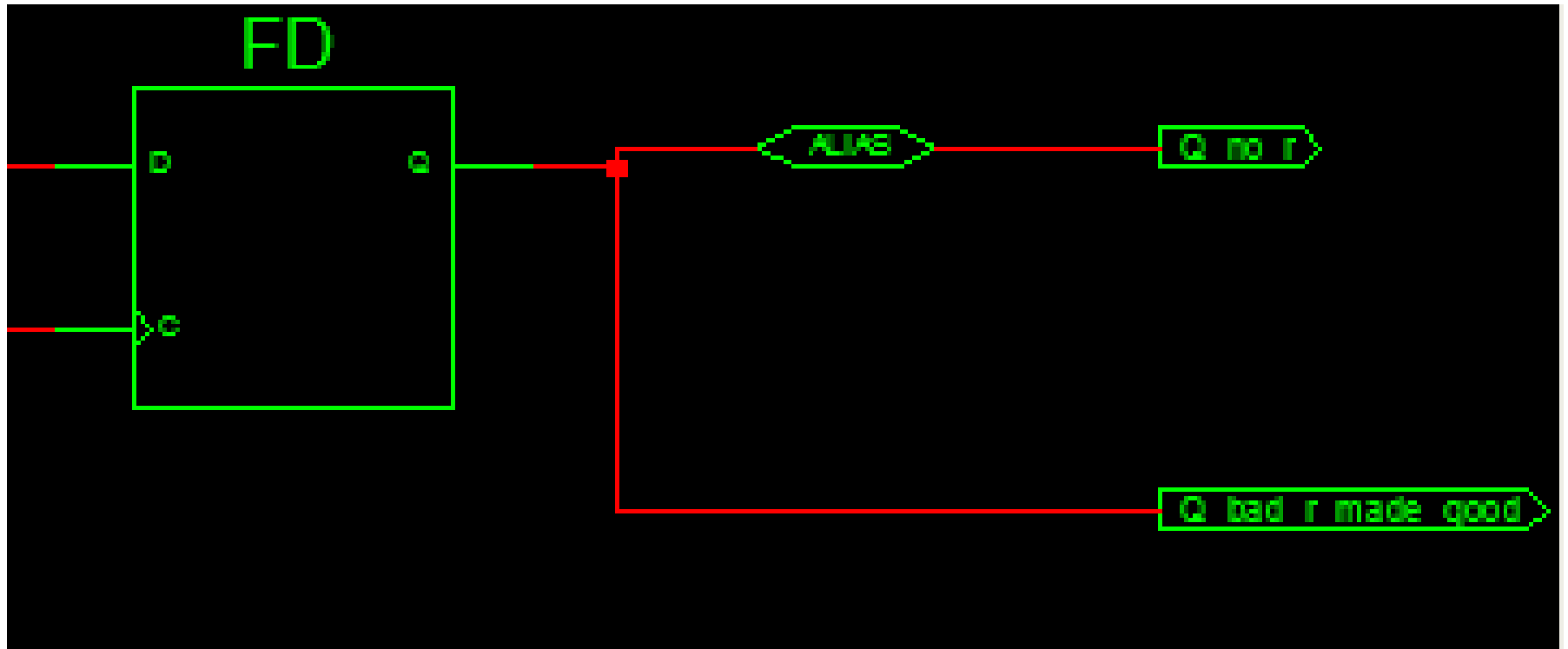
**always @(posedge clk)**

**begin : FF\_NO\_R**

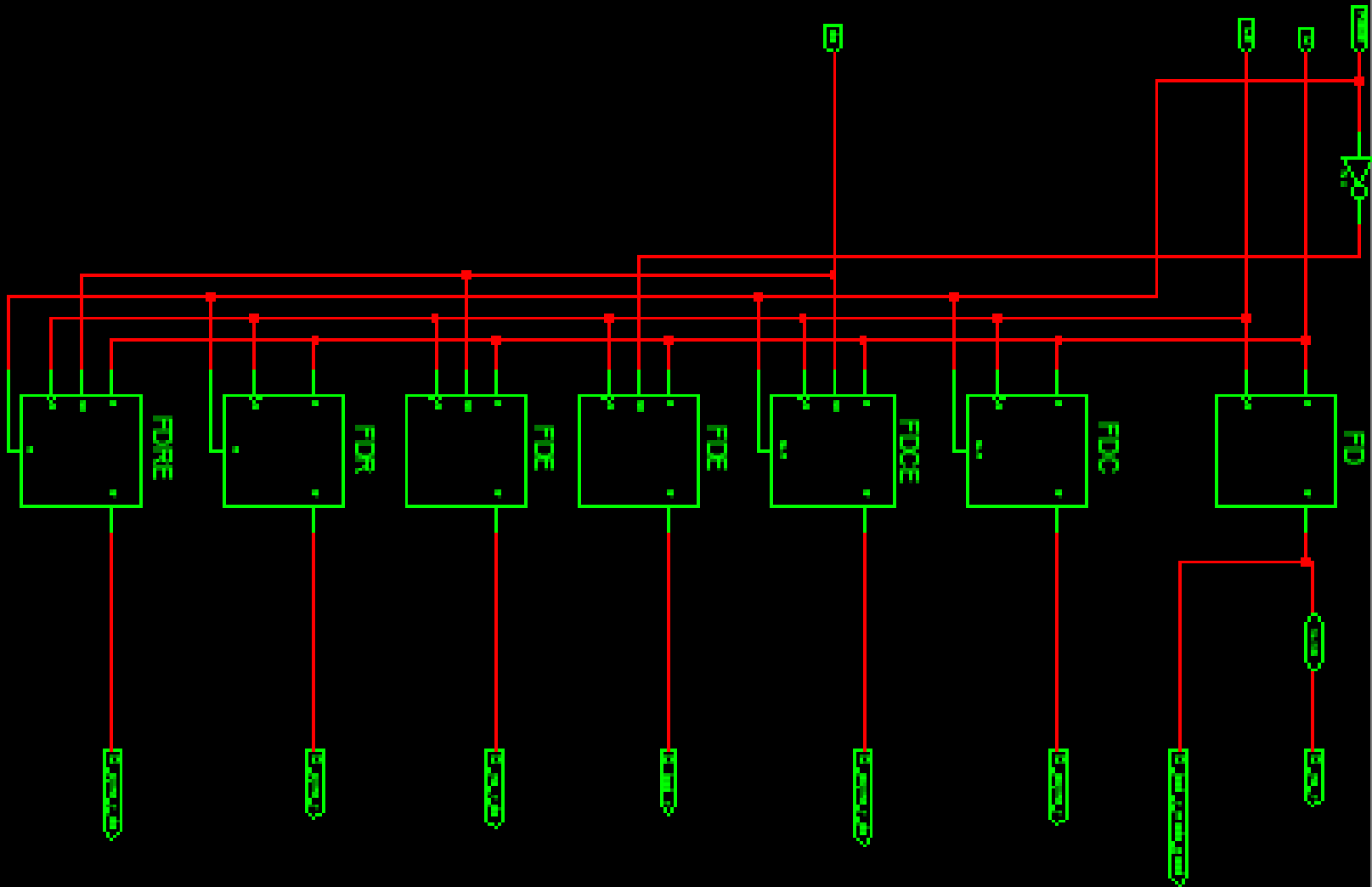
**Q\_no\_r <= D;**

**end**

Q\_no\_r  
= FF with no reset at all



# RTL schematic



**Q\_async\_r**  
**= FF with asynchronous reset**

**always @(posedge Clk, posedge Reset)**

**begin : FF\_ASYNC\_R**

**if (Reset)**

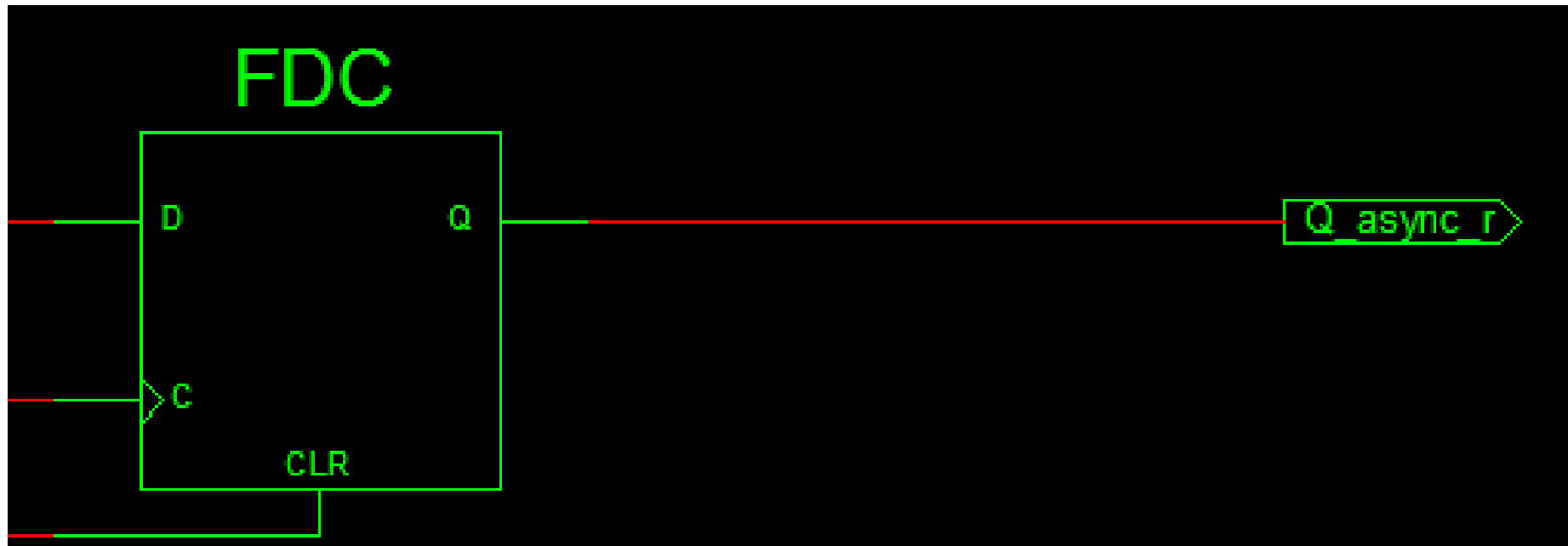
**Q\_async\_r <= 1'b0;**

**else**

**Q\_async\_r <= D;**

**end**

Q\_async\_r  
= FF with asynchronous reset



**Q\_sync\_r**  
**= FF with synchronous reset**

**always @(posedge clk)**

**begin : FF\_SYNC\_R**

**if (Reset)**

**Q\_sync\_r <= 1'b0;**

**else**

**Q\_sync\_r <= D;**

**end**

Q\_no\_r\_de = FF with Data Enable  
but no reset at all  
(suitable for data registers)

```
always @(posedge clk)
```

```
begin : FF_NO_R_DE
```

```
if (de)
```

```
    Q_no_r_de <= D;
```

```
end
```

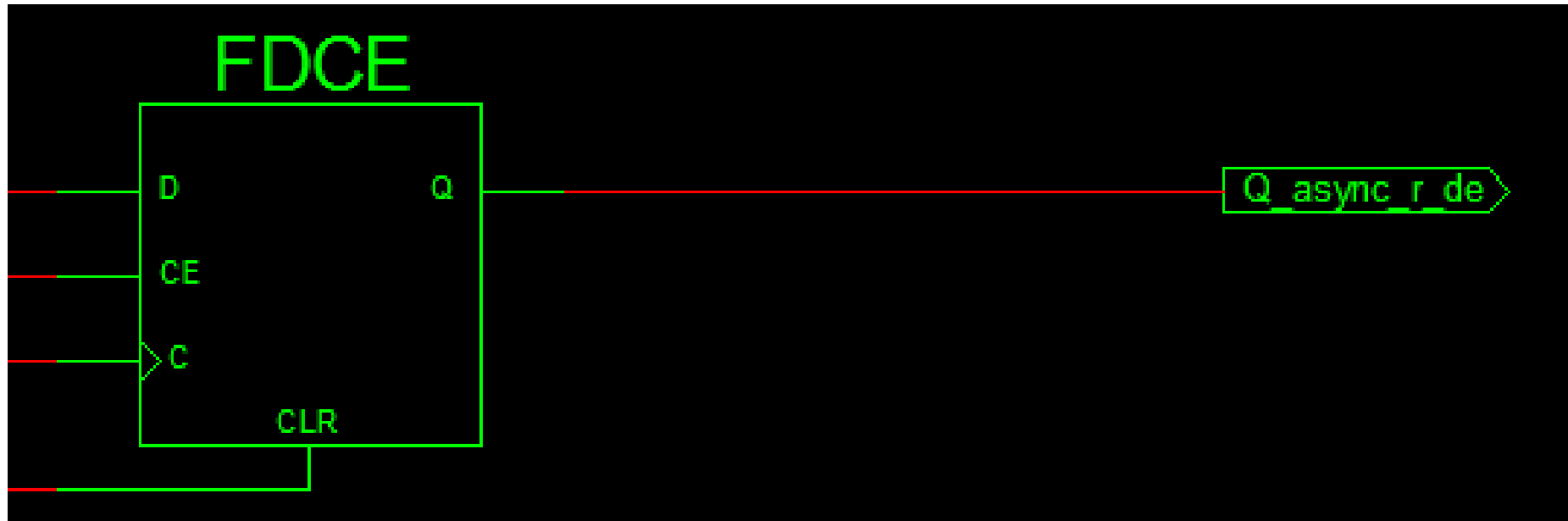
# Q\_async\_r\_de = FF with asynchronous reset with Data Enable

```
always @(posedge Clk, posedge Reset)

begin      : FF_ASYNC_R_DE

    if (Reset)
        Q_async_r_de  <= 1'b0;
    else if (de)
        Q_async_r_de  <= D;
end
```

# Q\_async\_r\_de = FF with asynchronous reset with Data Enable



# Q\_sync\_r\_de = FF with synchronous reset with Data Enable

```
always @(posedge clk)

begin      : FF_SYNC_R_DE

    if (Reset)
        Q_sync_r_de  <= 1'b0;
    else if (de)
        Q_sync_r_de  <= D;
end
```

**Q\_bad\_r = The BAD coding  
results in treating the RESET  
as a Data-enable control**

**always @(posedge Clk, posedge Reset)**

**begin : FF\_BAD\_R**

**if (Reset)**

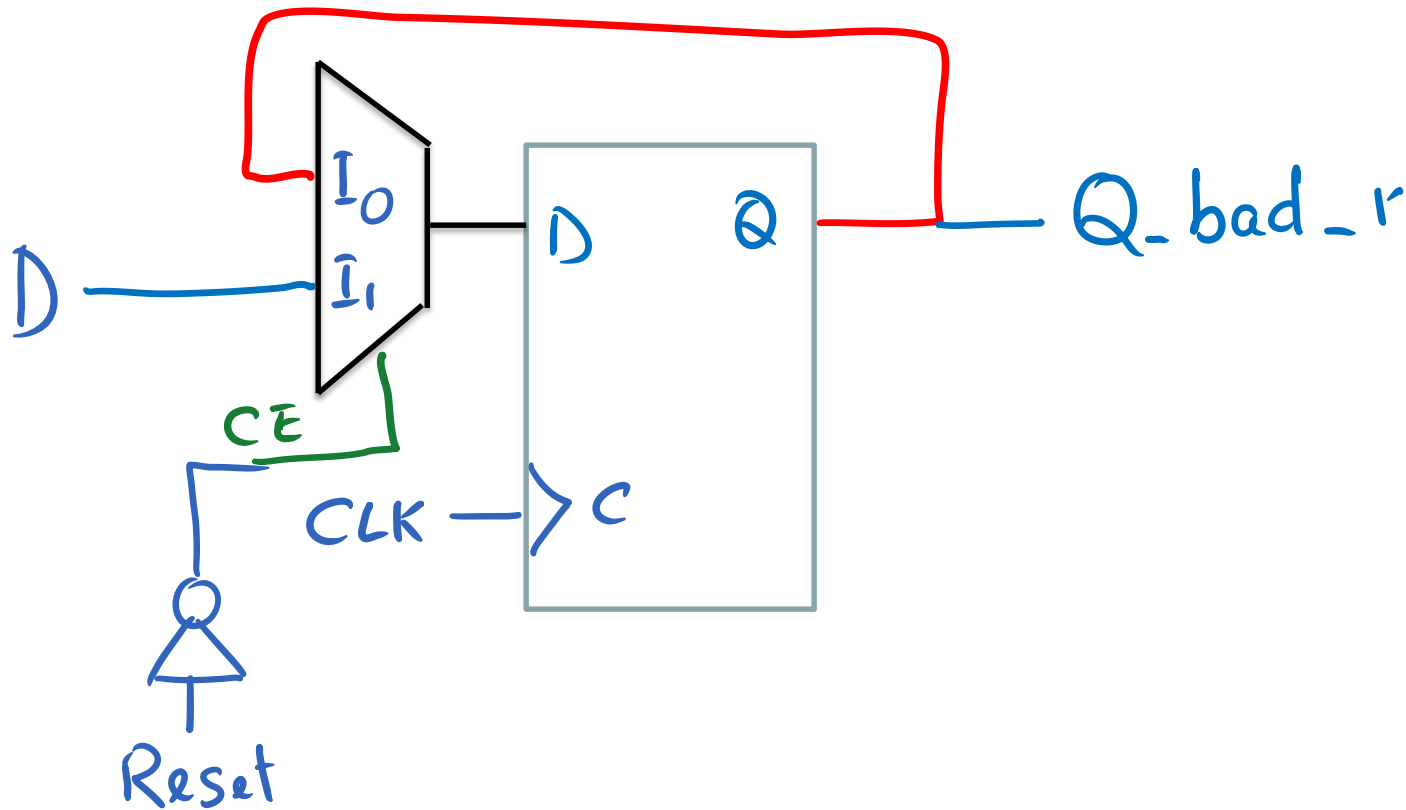
**; // nothing to do**

**else**

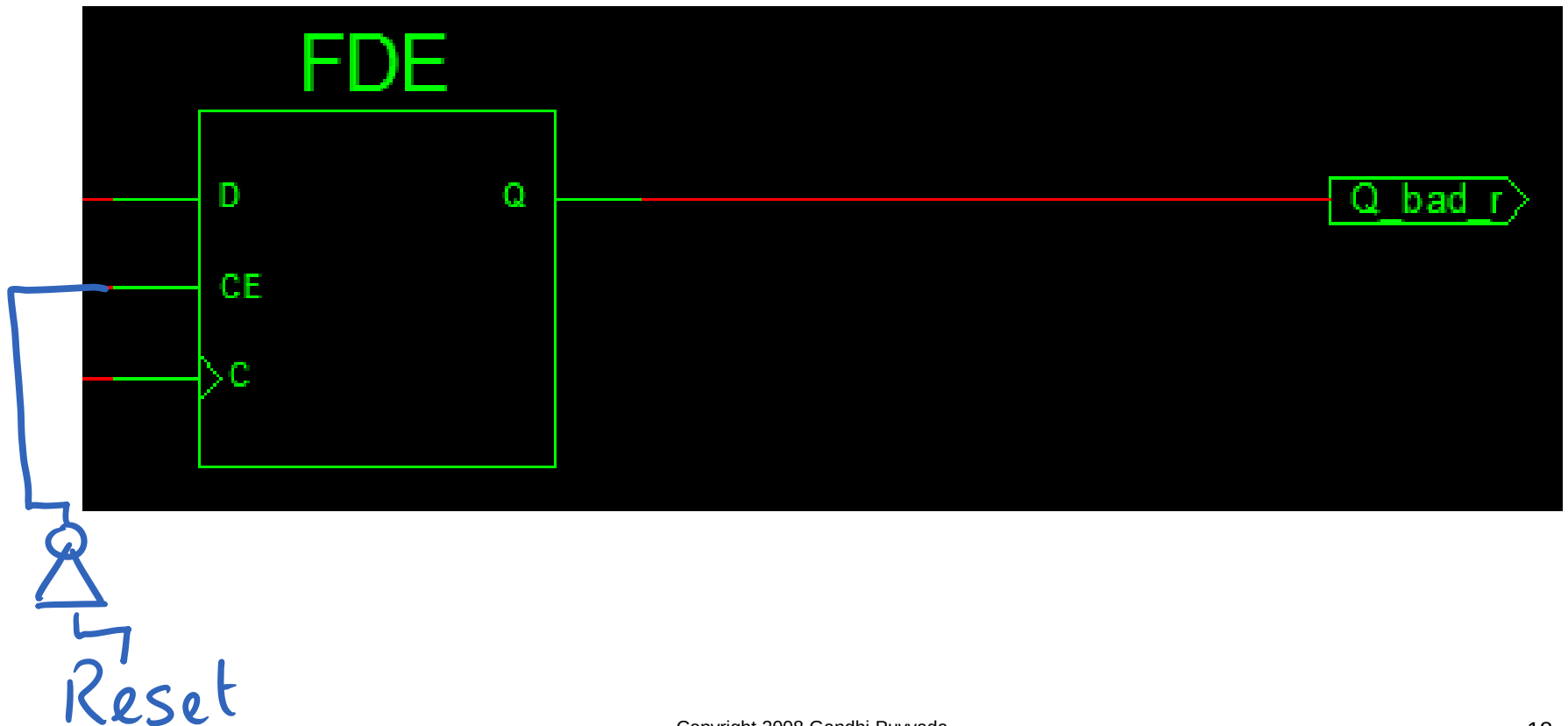
**Q\_bad\_r <= D;**

**end**

Q\_bad\_r = The BAD coding results in treating the RESET as a Data-enable control



Q\_bad\_r = The BAD coding results in treating the RESET as a Data-enable control



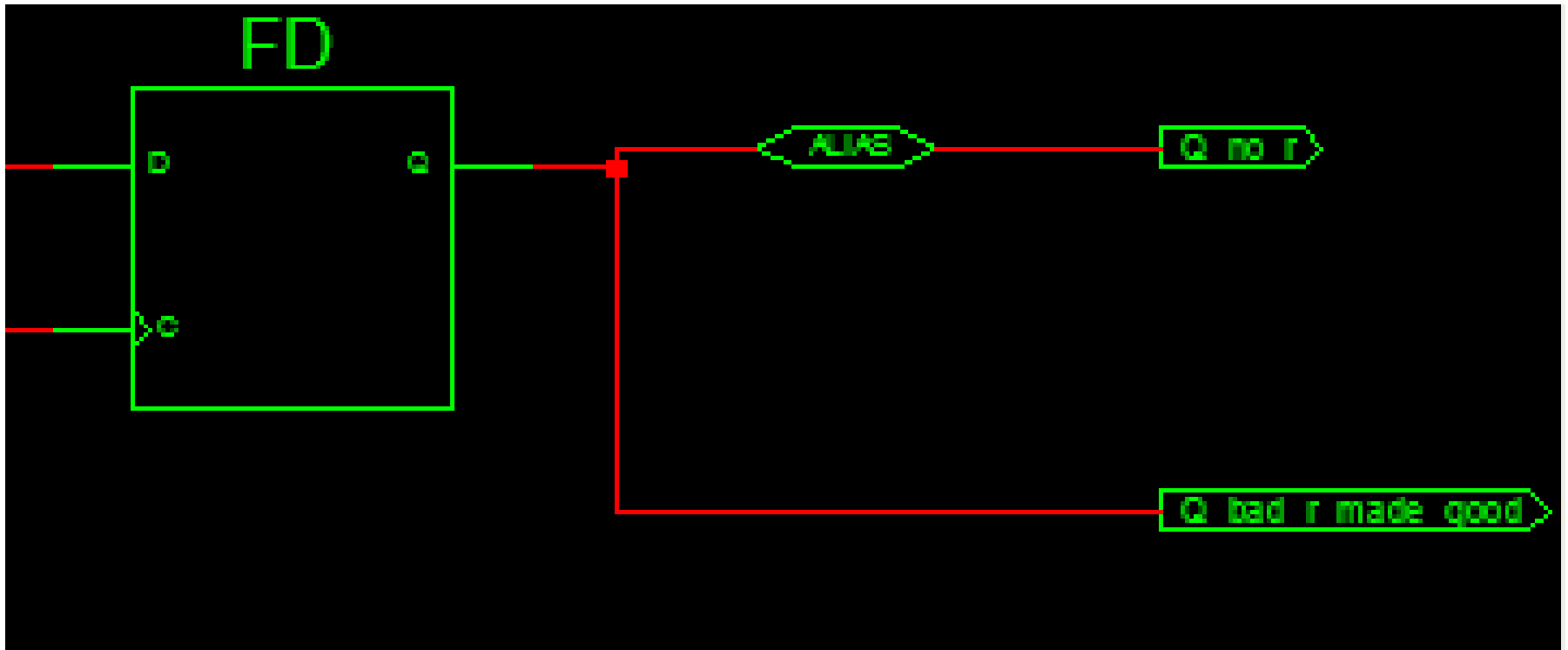
# Q\_bad\_r\_made\_good = BAD coding fixed!

```
always @(posedge Clk, posedge Reset)
begin
    : FF_BAD_R_MADE_GOOD

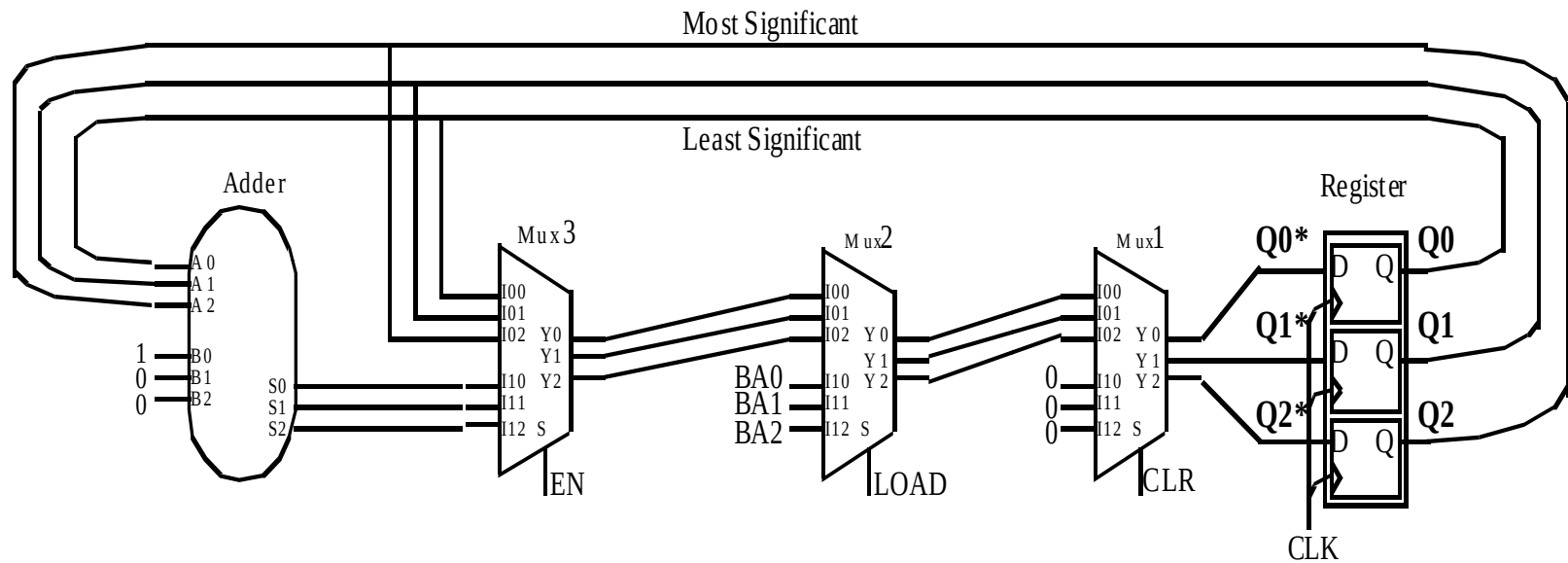
    if (Reset)
        Q_bad_r_made_good <= 1'bX ;

        // tell XST that you do not
        // care what happens during
        // the reset
    else
        Q_bad_r_made_good <= D;
end
```

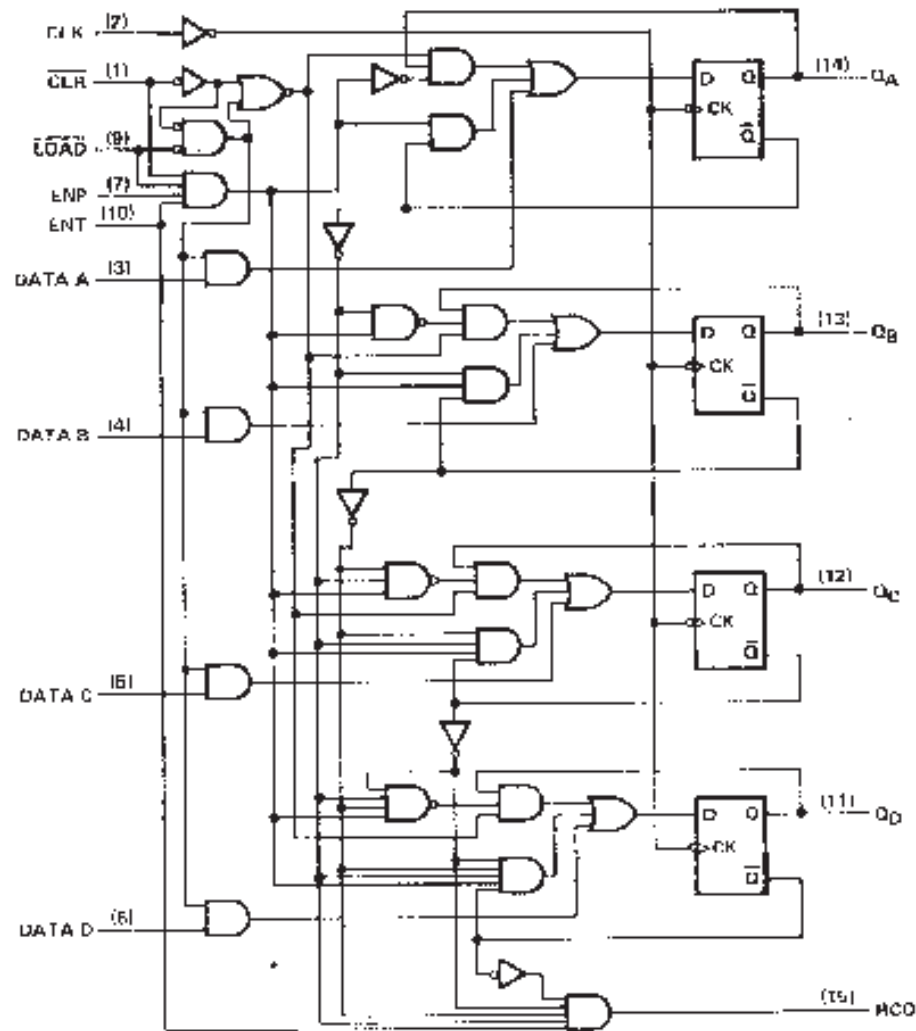
Q\_bad\_r\_made\_good =  
BAD coding fixed!



# Synchronous counter with CLR, LOAD, and EN controls

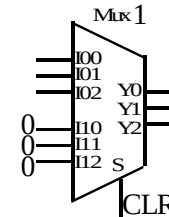
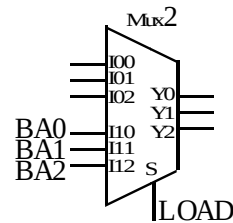
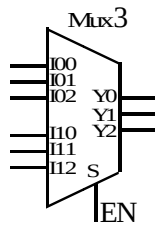
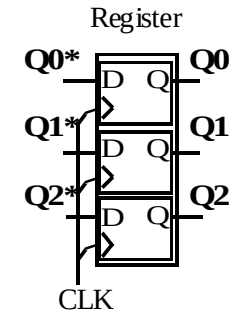
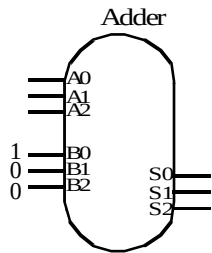


# 74LS163A Synchronous counter



Pin numbers shown are for D, J, H, and W packages

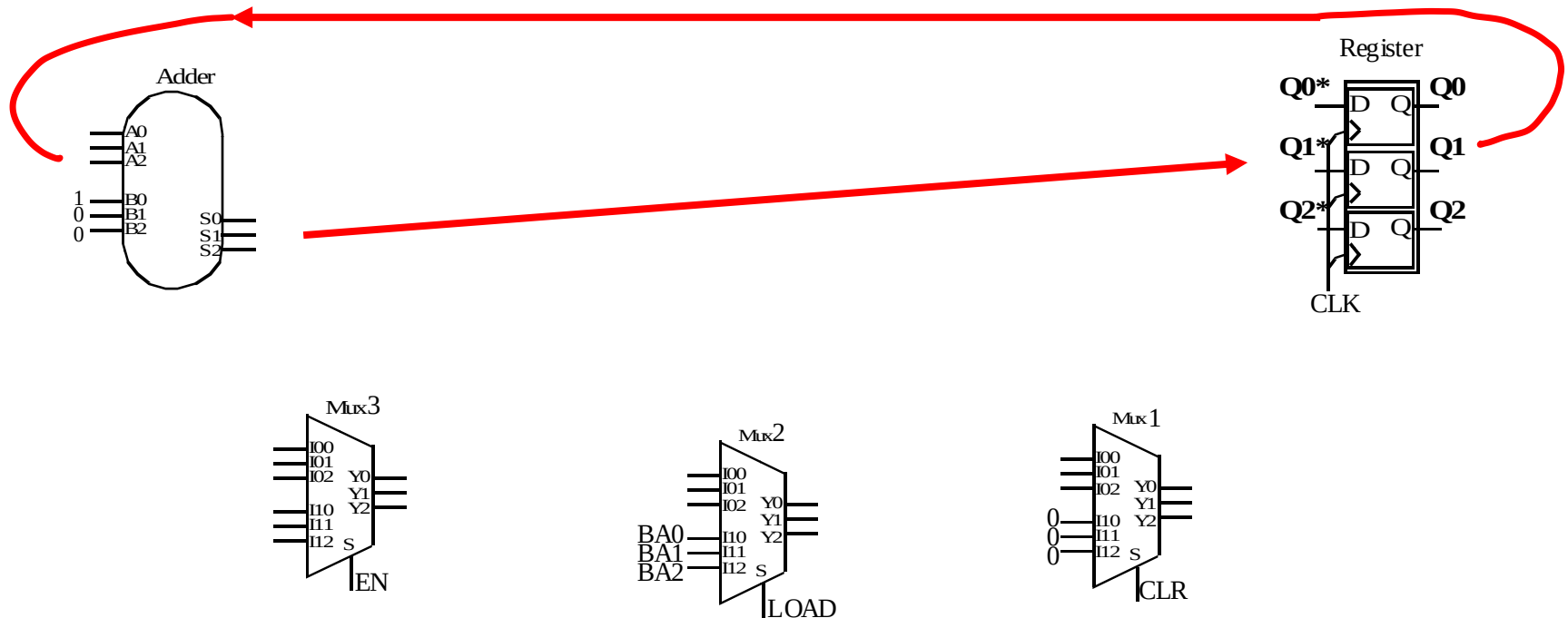
# Gradual development of counter



# Basic idea in datapath design in RTL: *intercept and inject*

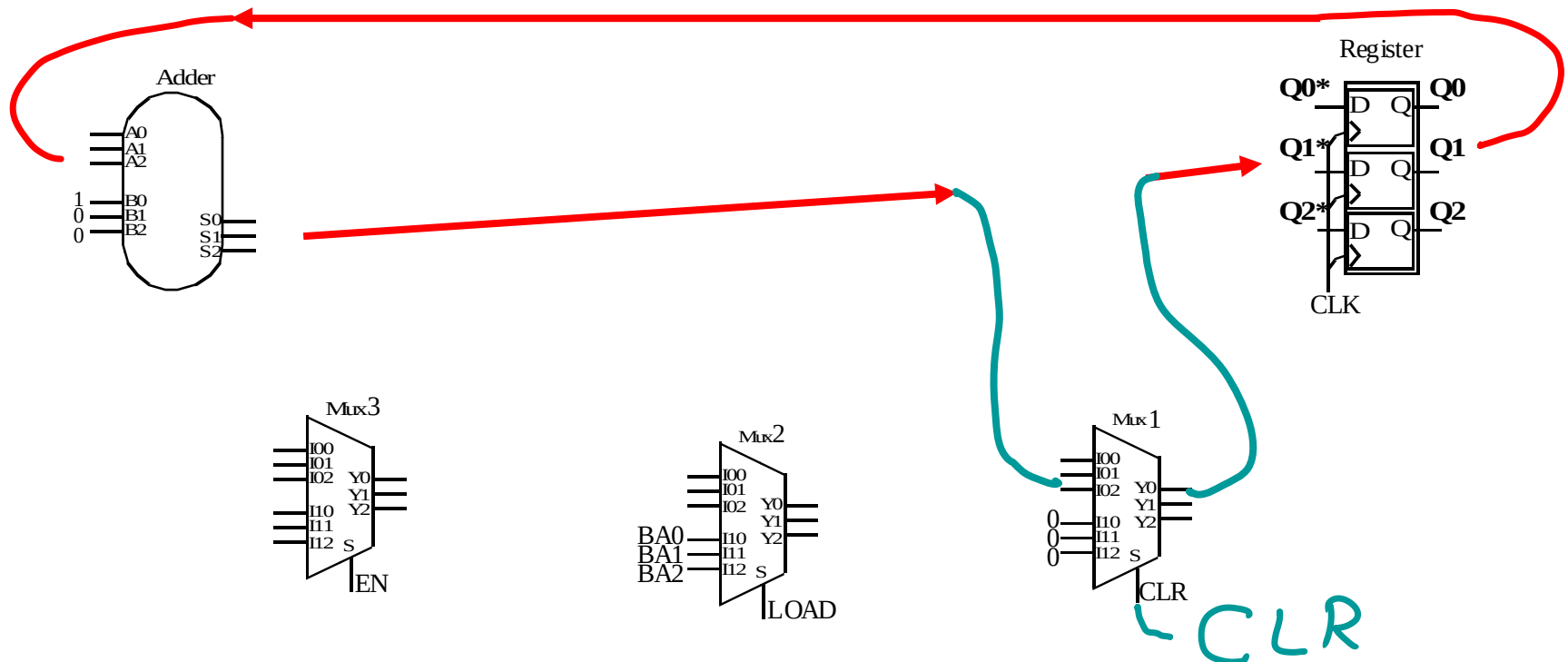
Free running counter

$i \leq i + 1;$



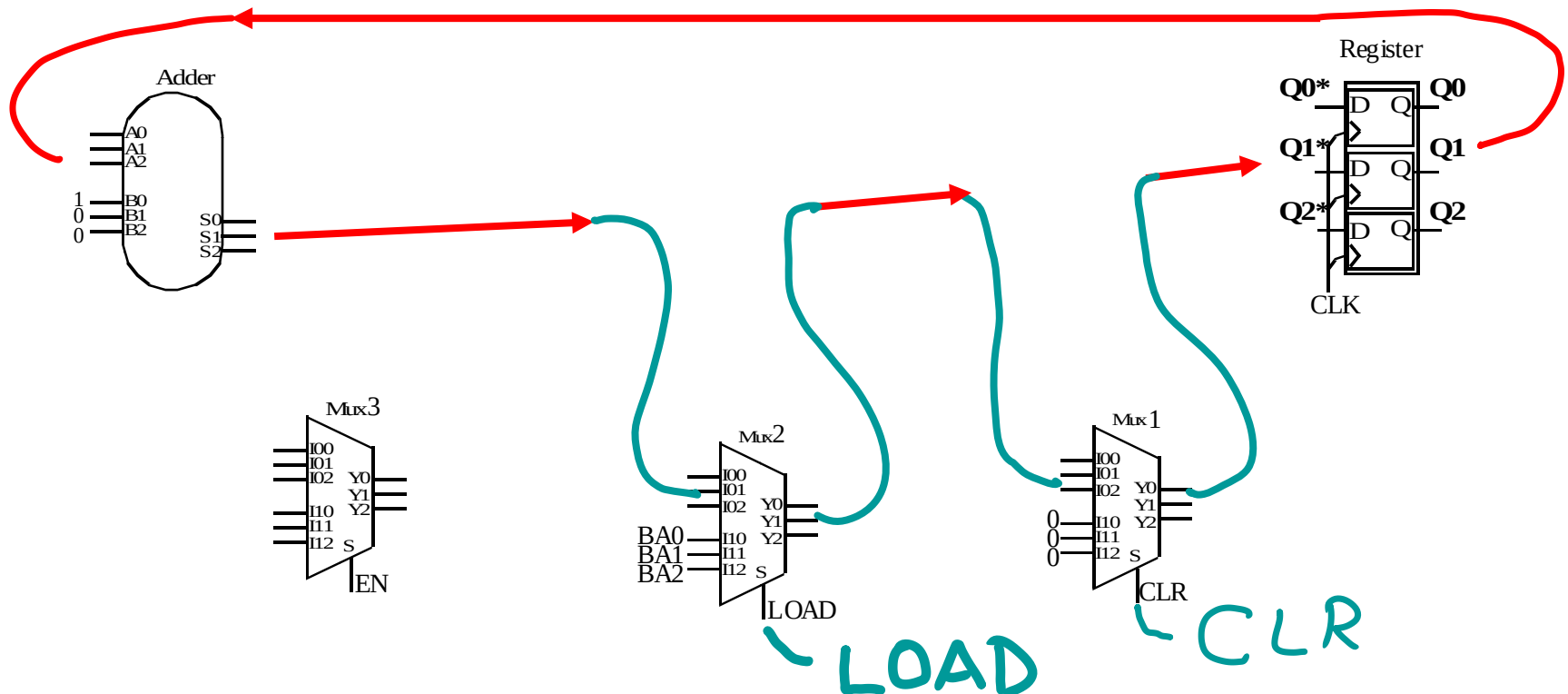
# Counter with CLR control

if (CLR)  $i \leq 3'b000$ ; else  $i \leq i + 1$ ;



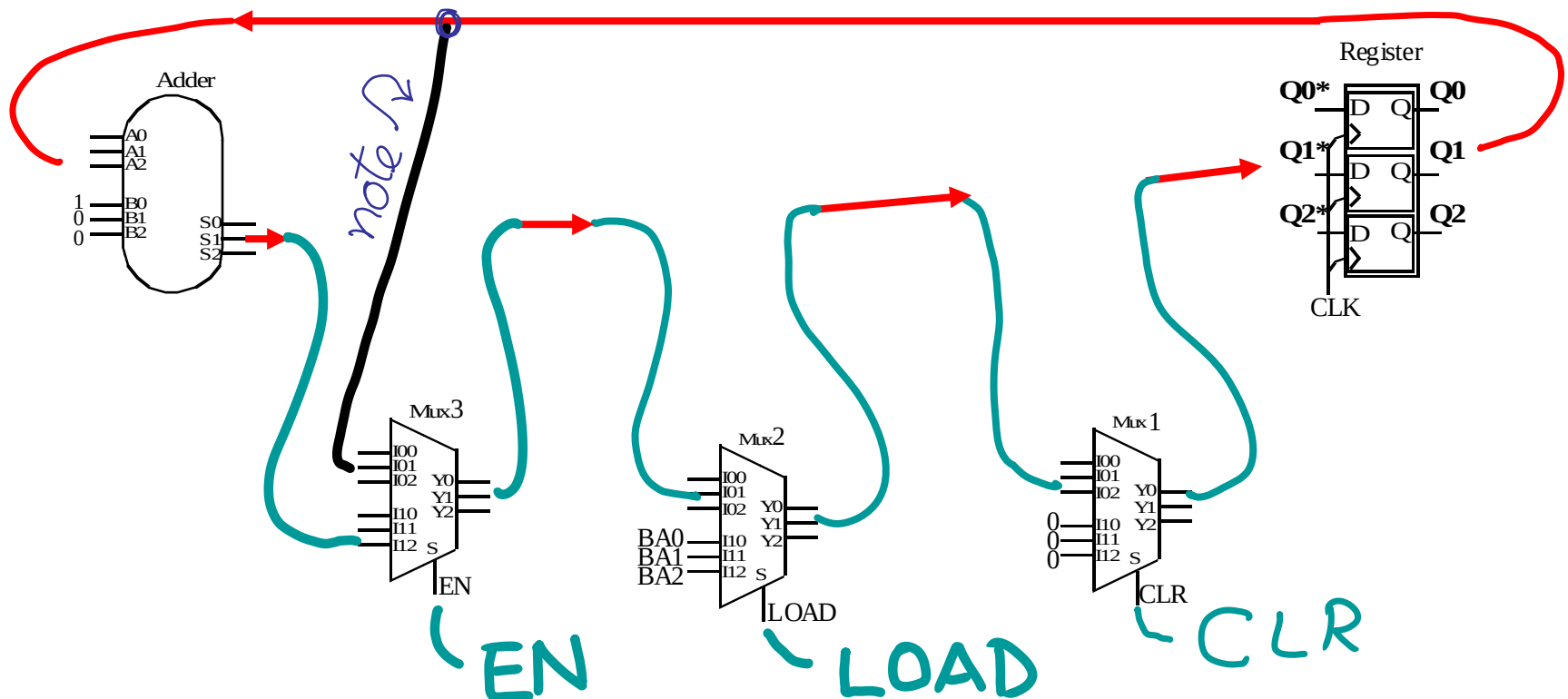
Counter with CLR and LOAD controls

if (CLR)  $i \leq 3'b000$ ;  
else if (LOAD)  $i \leq \{BA2, BA1, BA0\}$ ;  
else  $i \leq i + 1$ ;



# Counter with CLR, LOAD, and EN controls

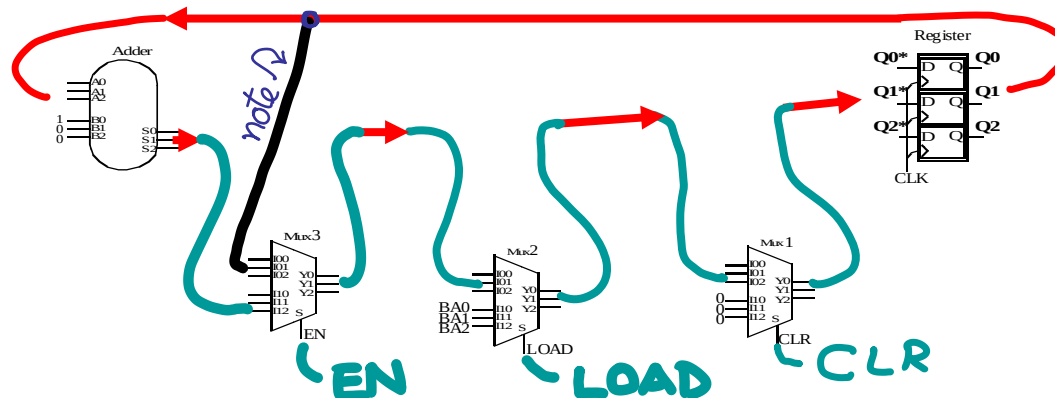
if (CLR)  $i \leq 3'b000$ ;  
else if (LOAD)  $i \leq \{BA2, BA1, BA0\}$ ;  
else if (EN)  $i \leq i + 1$ ;



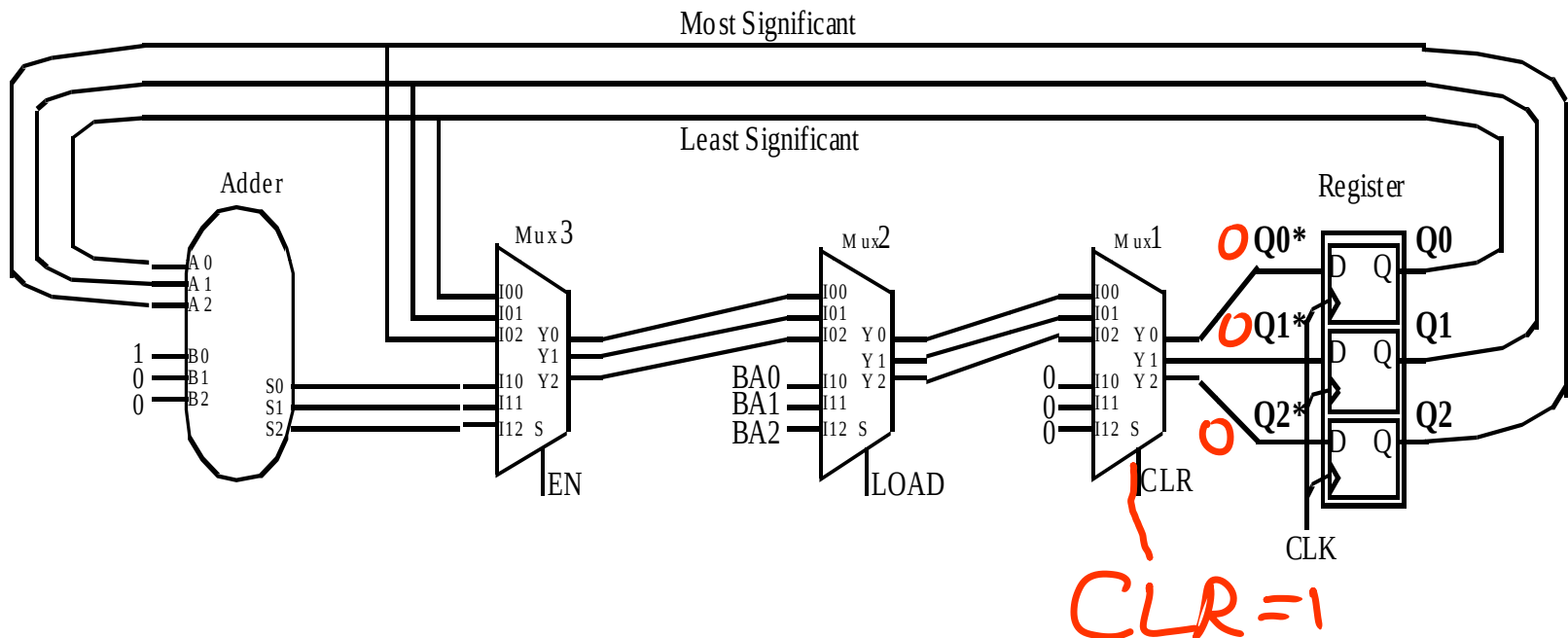
# Counter with CLR, LOAD, and EN controls

```
if (CLR) i <= 3'b000;  
else if (LOAD) i <= {BA2, BA1, BA0};  
else if (EN) i <= i + 1;  
else i <= i;
```

implicit. No harm but shows that you are writing RTL for the first time!

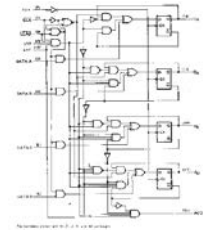


Is this **CLR control** a **synchronous** clear control or an **asynchronous** clear control?

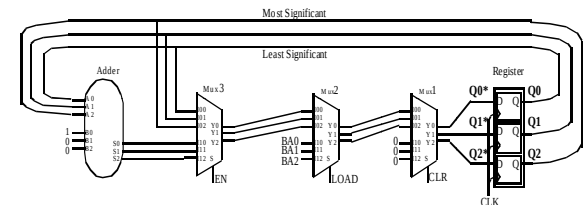


# How do we describe this counter in Verilog HDL?

- Structurally emulating the gate-level logic schematic of 74LS163A?

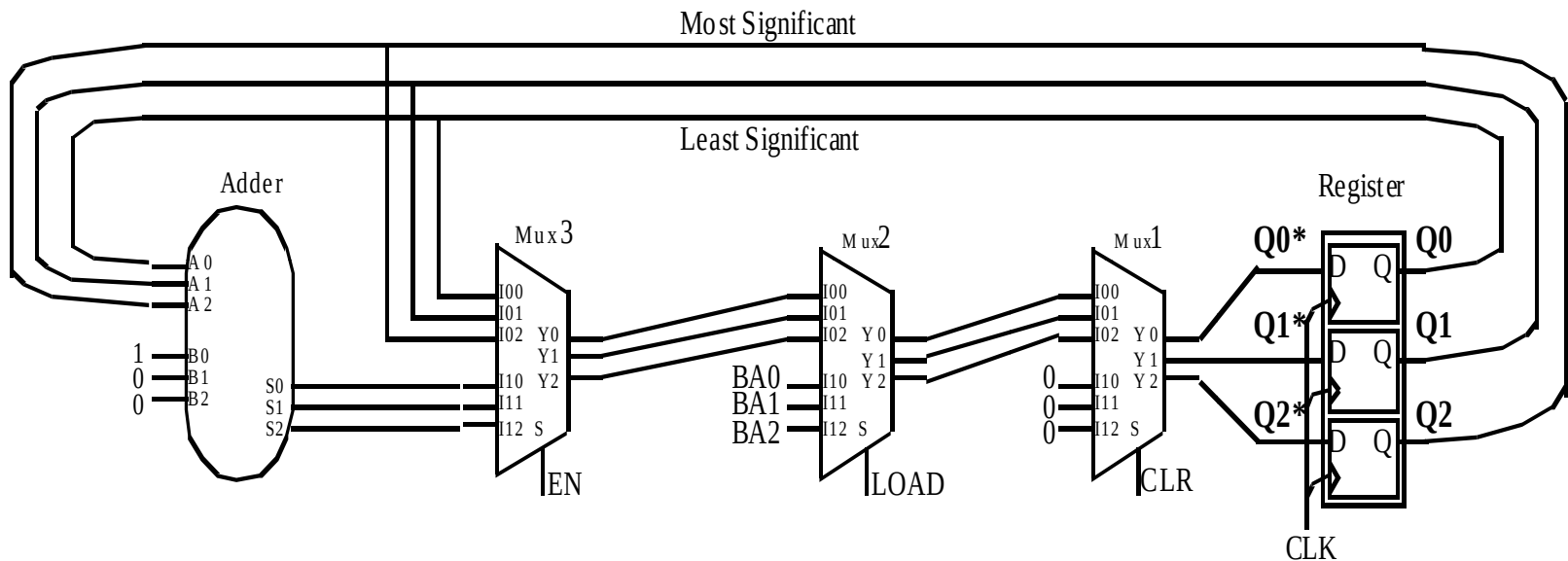


- Behaviorally based on the MSI level function design diagram?



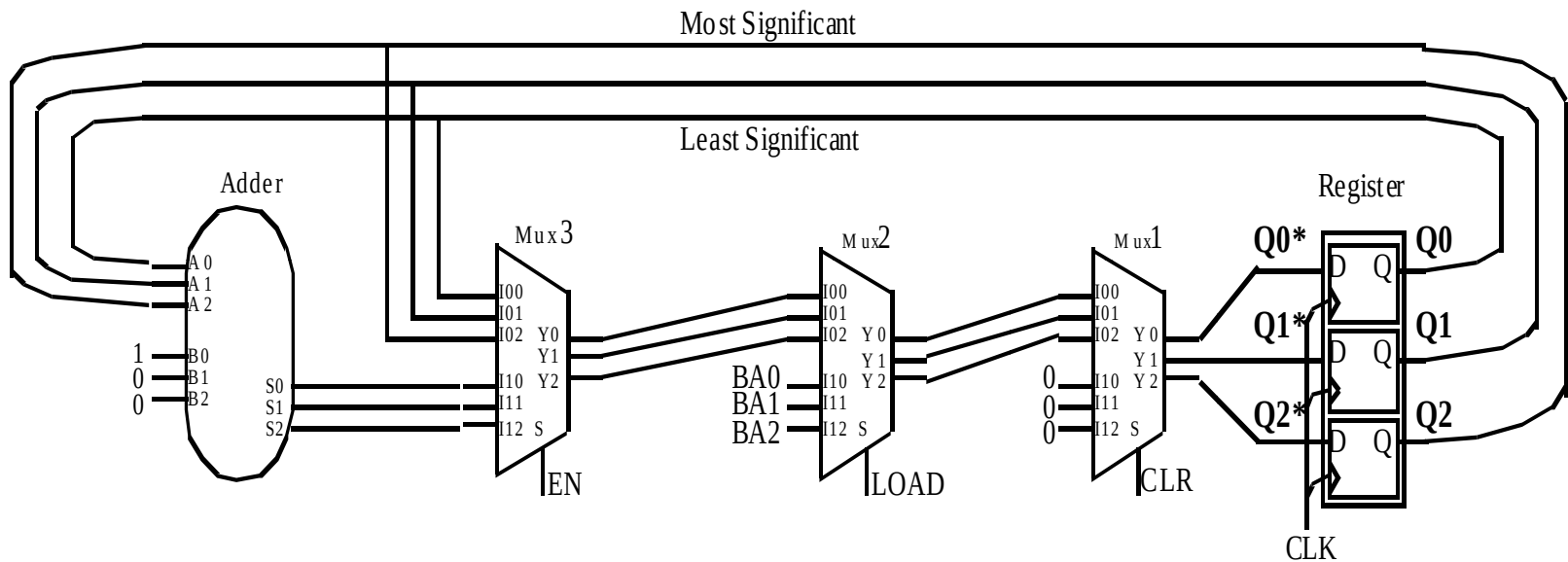
# Ok, we all agree -- behaviorally

- So, shall we describe three muxes and one incrementer and a register ?

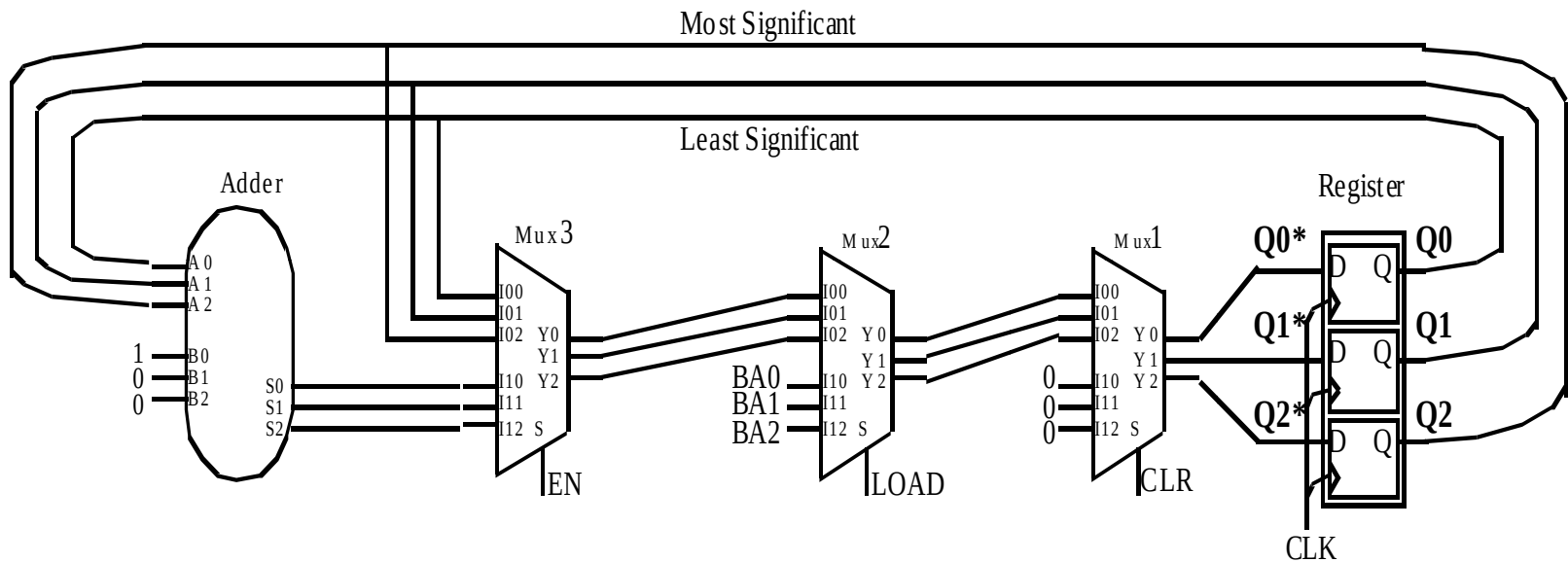


No!

5 separate items and their  
interconnection is **not quite readable!**



# An **if** statement in a clocked always block!



# Closest code `Q_sync_r_de = FF` with synchronous reset with Data Enable

```
always @(posedge clk)

begin      : FF_SYNC_R_DE

    if (Reset)
        Q_sync_r_de  <= 1'b0;
    else if (de)
        Q_sync_r_de  <= D;
end
```

# Coding of a counter with **synchronous** CLR, and synchronous LOAD, and EN

```
always @(posedge CLK, posedge CLR)  
begin    : COUNTER  
    if (CLR)  
        Count    <= 3'b000;  
    else if (LOAD)  
        Count    <= Load_input;  
    else if (EN)  
        Count    <= Count +1;  
end
```

# Coding of a counter with **asynchronous** CLR, and synchronous LOAD, and EN

```
always @(posedge CLK, posedge CLR)
begin    : COUNTER
    if (CLR)
        Count    <= 3'b000;
    else if (LOAD)
        Count    <= Load_input;
    else if (EN)
        Count    <= Count +1;
end
```

# **blocking and non-blocking assignments in procedural blocks**

*very important topic!*

*very confusing topic!*

# blocking assignments

```
...;  
a = b;  
...;  
...;
```

blocking  
assignment  
operator  
=

# blocking assignments

L.H.S

...;  
**a** = b;

blocking  
assignment

...;  
...;

Blocks these later  
Statements from executing  
until LHS is updated.

# blocking assignments

L.H.S



```
...;  
a = b;  
...;  
...;
```

blocking  
assignment

L.H.S is updated

immediately.

important

# blocking assignments

L.H.S

...;  
**a** = b;  
...

blocking  
assignment

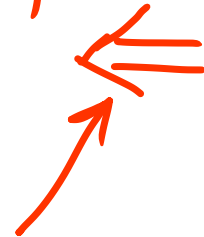
**c** = **a** + 1;  
...

This **a** is the  
new (updated) **a**.

# non-blocking assignments

```
...;  
a ≤ b;  
...;  
...;
```

non-blocking  
assignment  
operator



note the arrow.

# non-blocking assignments

L.H.S

...;  
 **$a \Leftarrow b$** ;

non-blocking  
assignment

...;  
...;

Does not  
block these later

statements from executing  
until LHS is updated.

# non-blocking assignments

L.H.S

  
...;  
**a <=> b;**  
...;  
...;

non-blocking  
assignment

L.H.S is updated  
only after  $\Delta T$  !

important 

# non-blocking assignments

L.H.S

...;  
**a** **≤** b;  
**c** **≤** **a** + 1;  
...;

non-blocking  
assignment

This **a** is the  
old (unupdated) **a**.

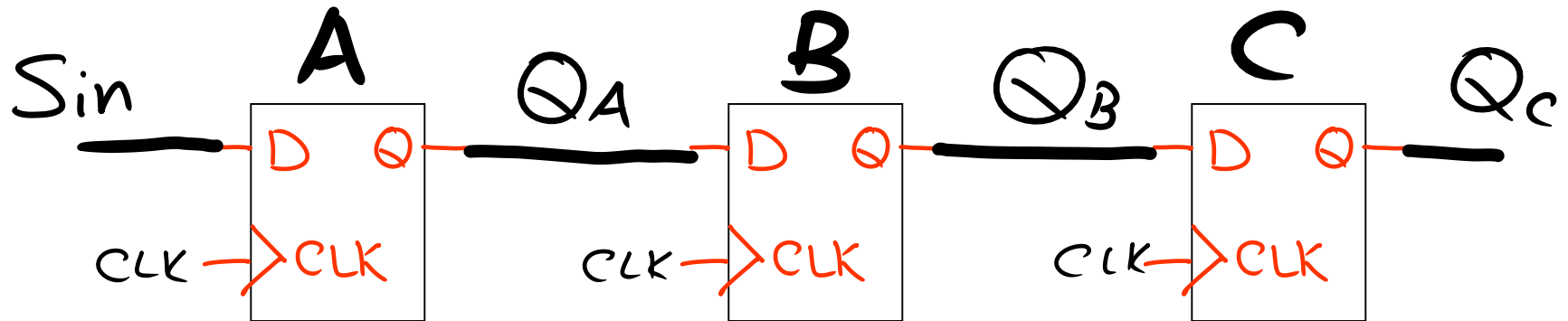
**blocking =  
and  
non-blocking ⇐  
assignments  
in procedural blocks**

*So where do we use  
what?*

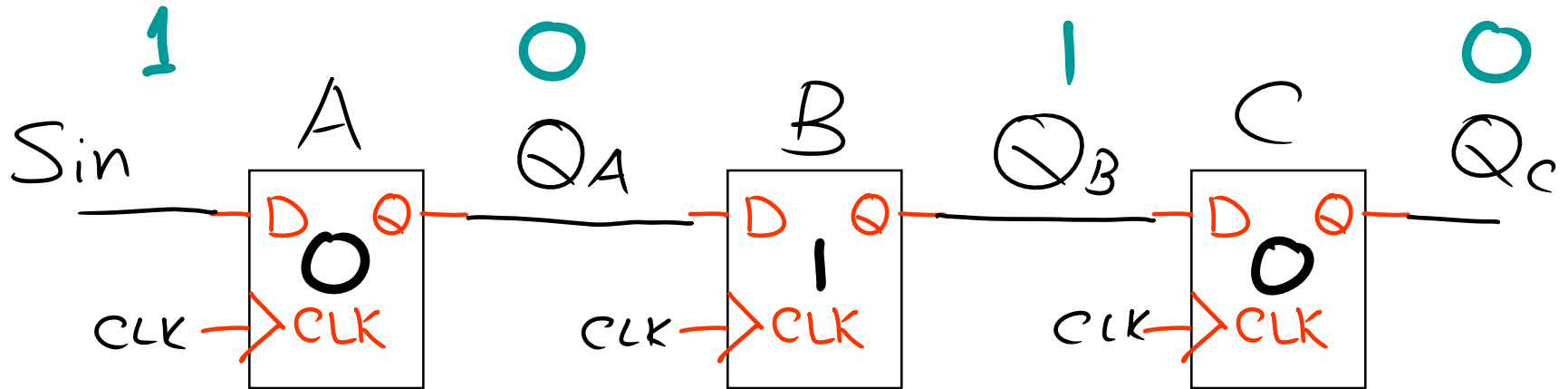
We will explain  
using a fitting example.

Consider a **RIGHT-SHIFT** register.

# RIGHT-SHIFT register



# RIGHT-SHIFT register



Current content = 

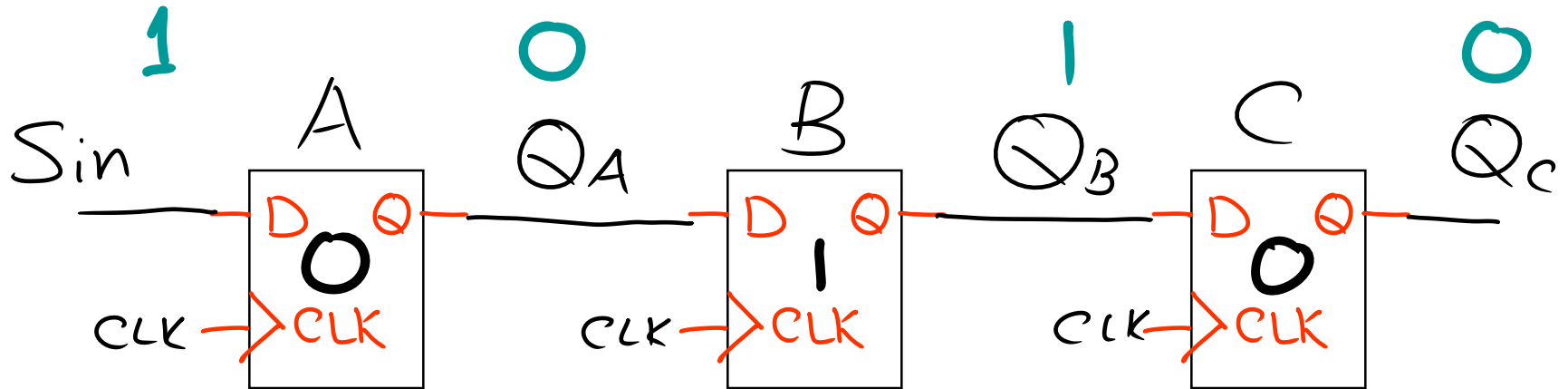
|   |   |   |
|---|---|---|
| A | B | C |
| 0 | 1 | 0 |

New Content = 

|   |   |   |
|---|---|---|
| 1 | 0 | 1 |
|---|---|---|

focus on B

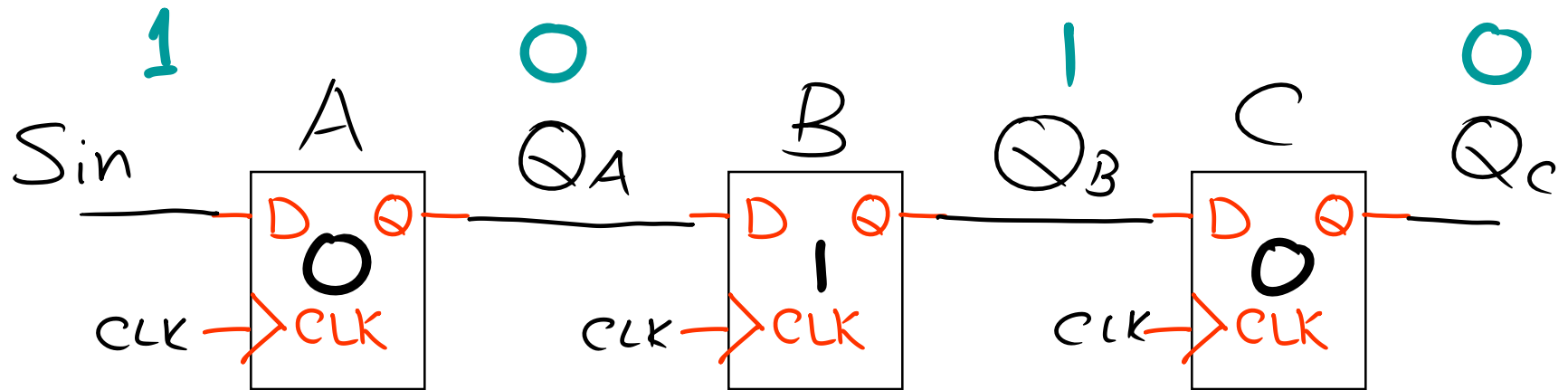
# RIGHT-SHIFT register



B's old value 1 goes into C'

B itself is getting a new value 0.

# RIGHT-SHIFT register



Looks like a RACE Condition!

# RACE

Condition  $\equiv$

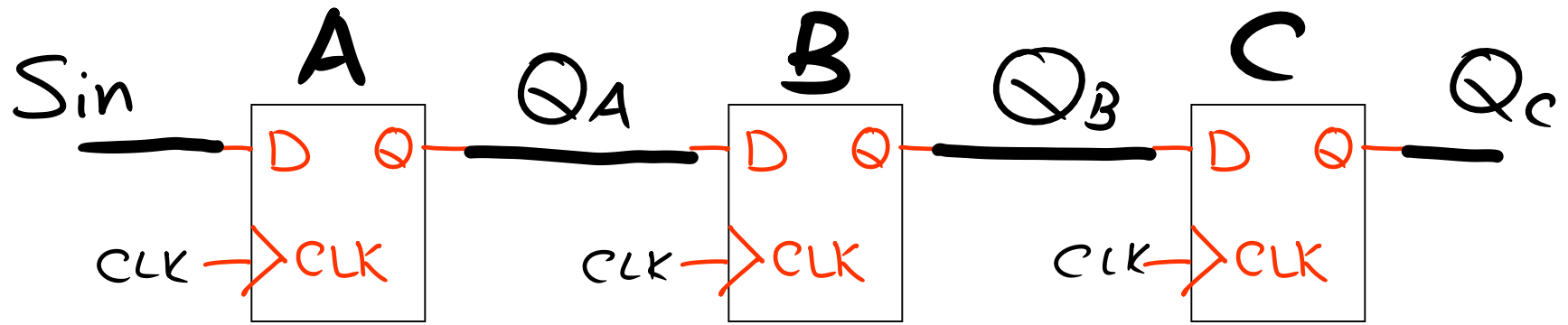
a condition that produces  
non-deterministic values

Circuit that produces  
non-deterministic values is

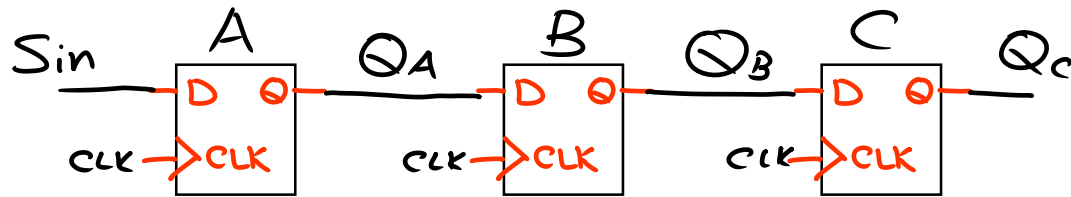
no good !

How do we avoid  
race condition?

How do we avoid  
race condition?



Shift one bit at a Time?

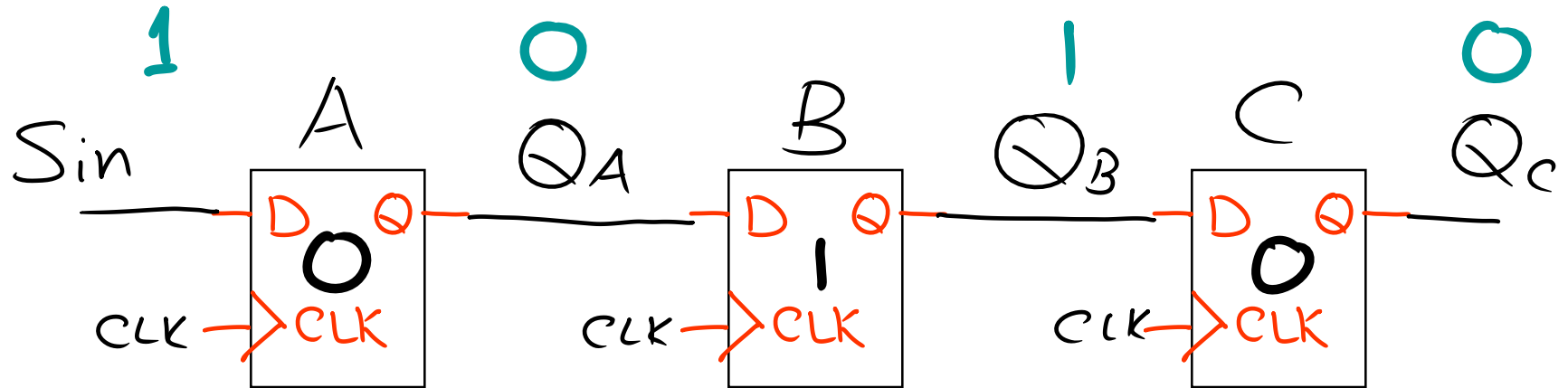


Shift one bit at a Time?

It works, but....

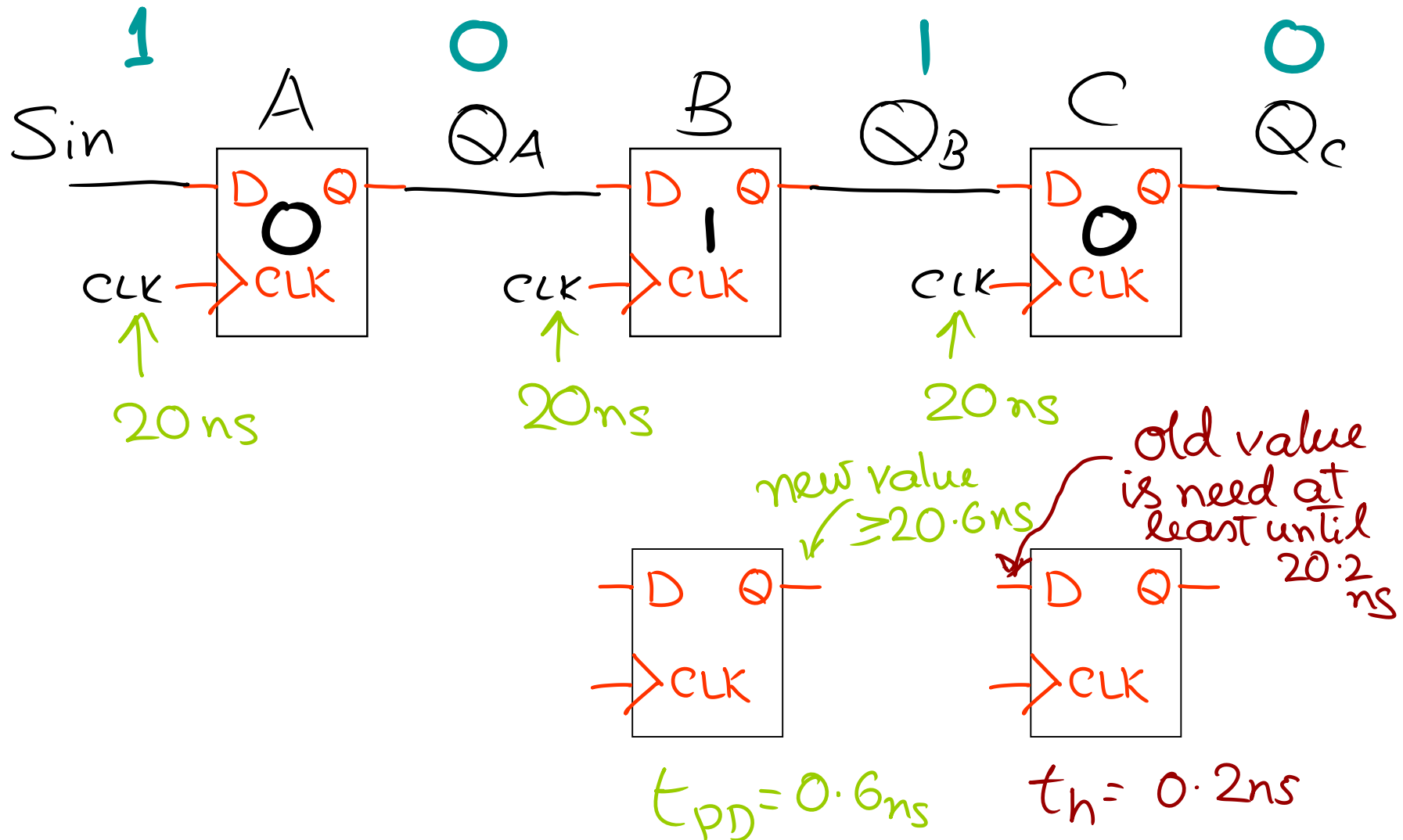
At this rate, it takes 32 clocks  
to right-shift by 1 bit  
in a 32-bit Shift Register.

# RIGHT-SHIFT register

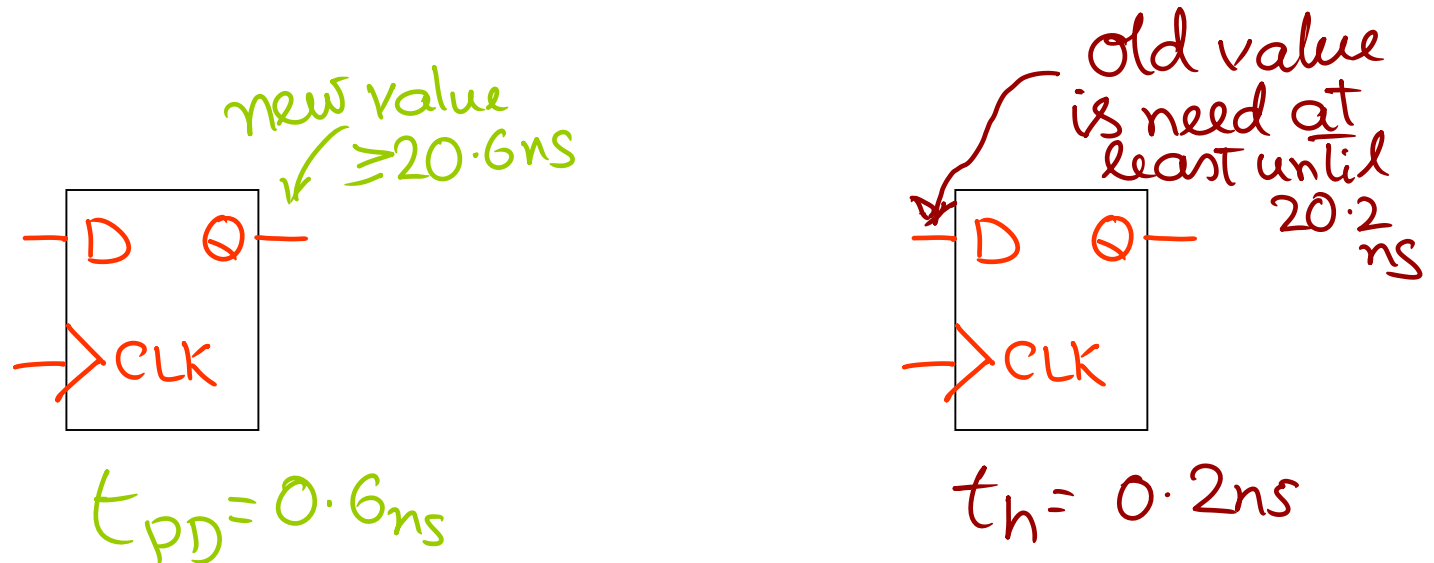


The **RACE** condition is **prevented**  
in hardware design by **proper**  
**Timing design !**

# RIGHT-SHIFT register



# TIMING Design avoids race



$t_{PD} \geq t_h$

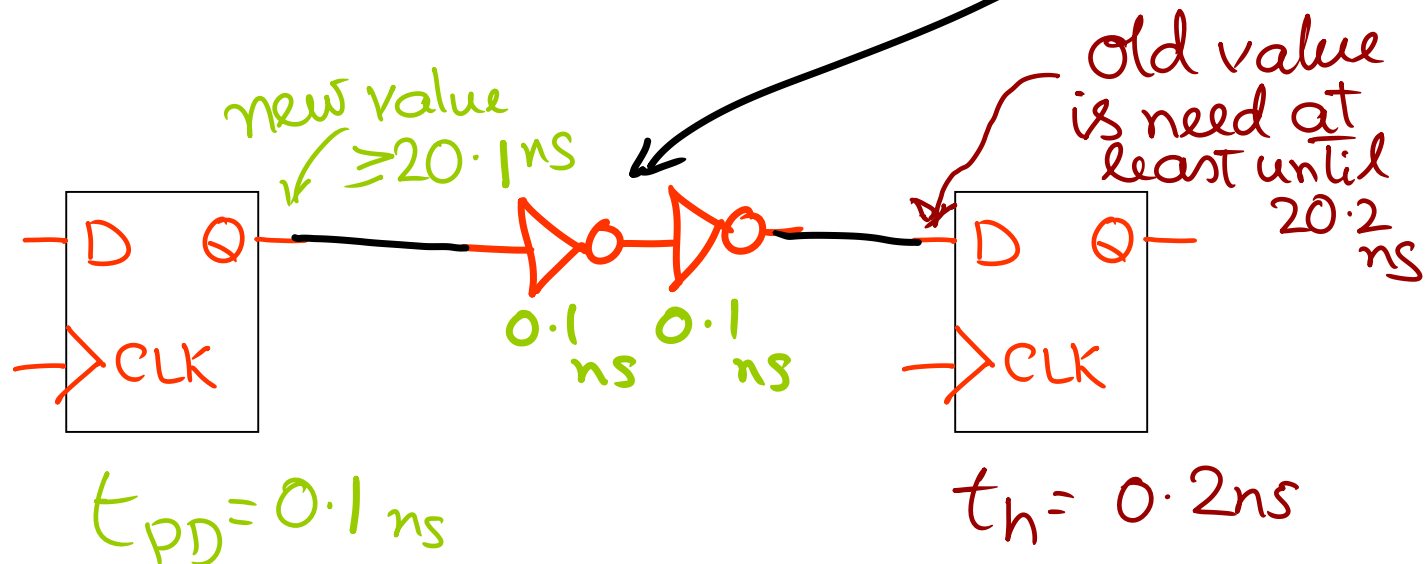
Golden rule!

# TIMING Design avoids race

what if  $t_{PD} = 0.1\text{ns}$ , and consequently

$$t_{PD} \geq t_h$$

Then simply apply BRUTE FORCE DELAY



**Static Timing Designer**  
(in synthesis and implementation tools)  
**performs**  
**millions of these timing checks**  
**to ensure that there are no RACE**  
**conditions!**

# **Static Timing Designer ....**

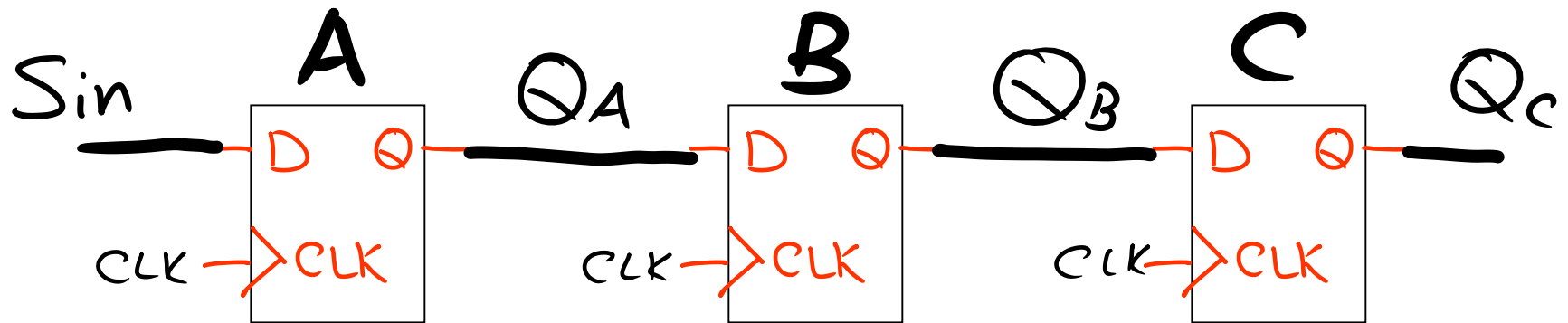
**But then, during initial behavioral simulation, we do not state any delays in our code! So, our ZERO delay coding will fail?!?**

Yes, that is where the  
**non-blocking** assignments  
with **delta-delay**  
save the day!

$$20 \cdot 6 \text{ ns} > 20 \cdot 2 \text{ ns}$$

$$20 + \Delta T > 20 \cdot 0$$

# RIGHT-SHIFT register



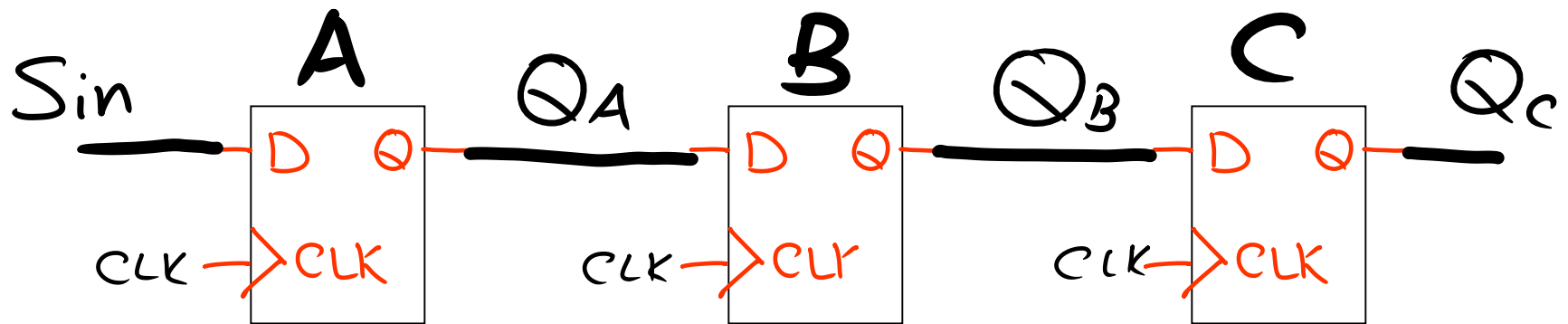
```
QC <= QB;  
QB <= QA;  
QA <= Sin;
```

```
QA <= Sin;  
QB <= QA;  
QC <= QB;
```

```
QC = QB;  
QB = QA;  
QA = Sin;
```

```
QA = Sin;  
QB = QA;  
QC = QB;
```

# RIGHT-SHIFT register



**QC** <= **QB**;  
**QB** <= **QA**;  
**QA** <= **Sin**;

*will update after  $\Delta T$*

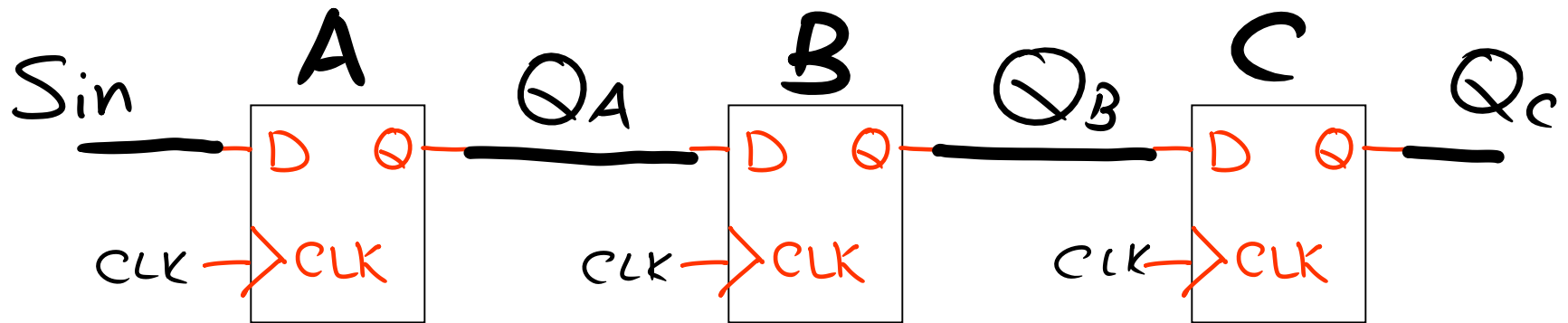
**QA** <= **Sin**;  
**QB** <= **QA**;  
**QC** <= **QB**;

**QC** = **QB**;  
**QB** = **QA**;  
**QA** = **Sin**;

*will update immediately*

**QA** = **Sin**;  
**QB** = **QA**;  
**QC** = **QB**;

# RIGHT-SHIFT register



3 FFs

```
QC <= QB;
QB <= QA;
QA <= Sin;
```



3 FFs

```
QA <= Sin;
QB <= QA;
QC <= QB;
```



```
QC = QB;
QB = QA;
QA = Sin;
```



Creates  
race  
elsewhere

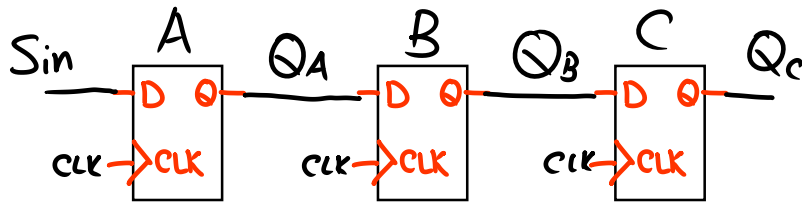
Results in  
3 FFs

```
QA = Sin;
QB = QA;
QC = QB;
```



Results in 1 FF

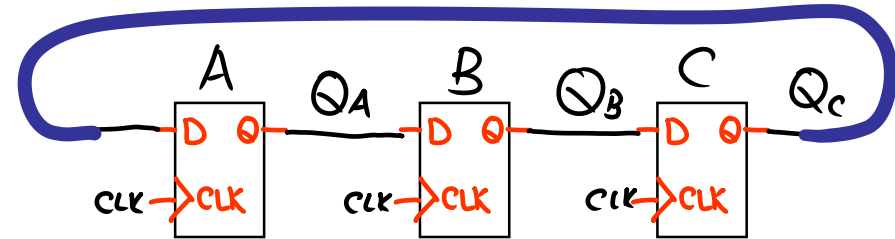
## RIGHT-SHIFT register



```
QC = QB;  
QB = QA;  
QA = Sin;
```

✓?  
Creates  
race  
Results in  
3FFs elsewhere

## CIRCULAR-SHIFT register



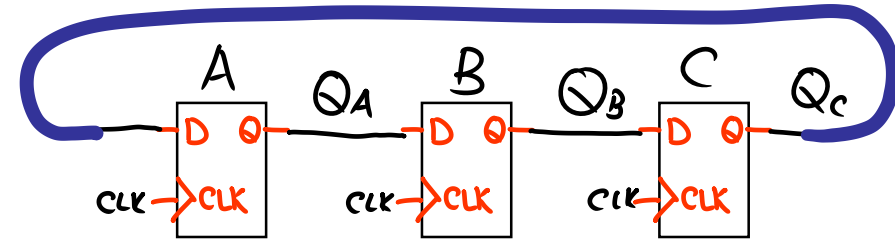
```
QC = QB;  
QB = QA;  
QA = QC;
```

```
QC <= QB;  
QB <= QA;  
QA <= QC;
```

Does any of the two work?

# CIRCULAR-SHIFT register

*Q<sub>C</sub> is effectively a temporary variable*



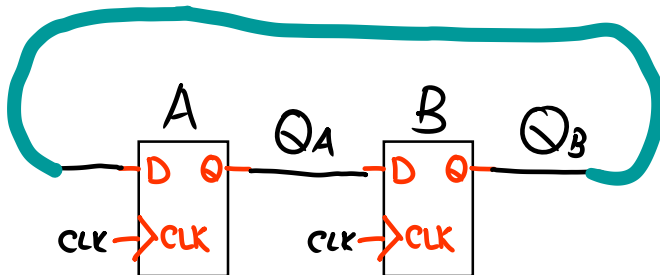
*new Q<sub>B</sub>*  
~~QC = QB;~~  
~~QB = QA;~~  
~~QA = QC;~~

*old Q<sub>B</sub> as saved in Q<sub>C</sub>*

~~QC = QB;~~  
~~QB = QA;~~  
~~QA = QC;~~



QC <= QB;  
QB <= QA;  
QA <= QC;



Does any of the two work?

# Conclusion on non-blocking assignment usage: Use it for assigning to physical registers

In any real system, there are hundreds of registers updating on the same clock tick.

Each register (through its NSL) should decide what to do (what value to acquire) based on the current state (before the next clock edge) of other registers (including itself).

Hence, use **non-blocking assignments** to assign a value to a register so that the new value appears after **delta-T** and does not cause any race condition.

Then, when do we use  
blocking assignments?

For intermediate variables, (and in  
combinational logic)

# intermediate variables in multi-level combinational logic

---

**Example problem:**  
**Given four numbers P, Q, X, Y**  
**find the smallest value and**  
**output it as R (the result).**

One can use the **conditional select operator** in the continuous assign statement or in a procedural assignment statement, but the if statement (in a procedural block) is much more readable.

# intermediate variables in multi-level combinational logic

---

## Without intermediate variables

//conditional select operator

sel ? True : false

assign R\_assign1 =

( ((P < Q) ? P : Q) < ((X < Y) ? X : Y) ) ?

((P < Q) ? P : Q) : ((X < Y) ? X : Y);

# intermediate variables in multi-level combinational logic

---

## With wire type intermediate variables (Data Flow modeling)

//conditional select operator

sel ? True : false

assign small\_PQ\_wire = (P < Q) ? P : Q ;

assign small\_XY\_wire = (X < Y) ? X : Y ;

assign R\_assign2 = (small\_PQ\_wire < small\_XY\_wire) ?  
small\_PQ\_wire : small\_XY\_wire ;

# intermediate variables in multi-level combinational logic

## With reg type intermediate variables (behavioral modeling)

```
always @*
```

```
begin : SMALL_ALWAYS
```

```
// local reg variables
```

```
reg [1:0] small_PQ_reg;
```

```
reg [1:0] small_XY_reg;
```

body of the block

```
end
```

```
if (P < Q)
```

```
    small_PQ_reg = P;
```

```
else
```

```
    small_PQ_reg = Q;
```

```
if (X < Y)
```

```
    small_XY_reg = X;
```

```
else
```

```
    small_XY_reg = Y;
```

```
if (small_PQ_reg < small_XY_reg)
```

```
    R_always = small_PQ_reg;
```

```
else
```

```
    R_always = small_XY_reg;
```

blocking  
assignments  
only

??

**Final** (final, not intermediate)  
**assignment** in combinational logic can be  
*either blocking or non-blocking.*  
As a thumb rule, we can use blocking  
assignments uniformly for describing  
combinational logic.

```
if (P < Q)
    small_PQ_reg = P;
else
    small_PQ_reg = Q ;
if (X < Y)
    small_XY_reg = X;
else
    small_XY_reg = Y ;
if (small_PQ_reg < small_XY_reg)
    R_always = small_PQ_reg;
else
    R_always = small_XY_reg ;
```

The diagram illustrates the classification of assignments in the provided code. Red arrows point to the following lines, which are identified as blocking assignments: `small_PQ_reg = P;`, `small_PQ_reg = Q ;`, `small_XY_reg = X;`, and `small_XY_reg = Y ;`. A blue arrow points to the line `R_always = small_XY_reg ;` in the `else` branch of the final `if` statement, which is identified as a non-blocking assignment.

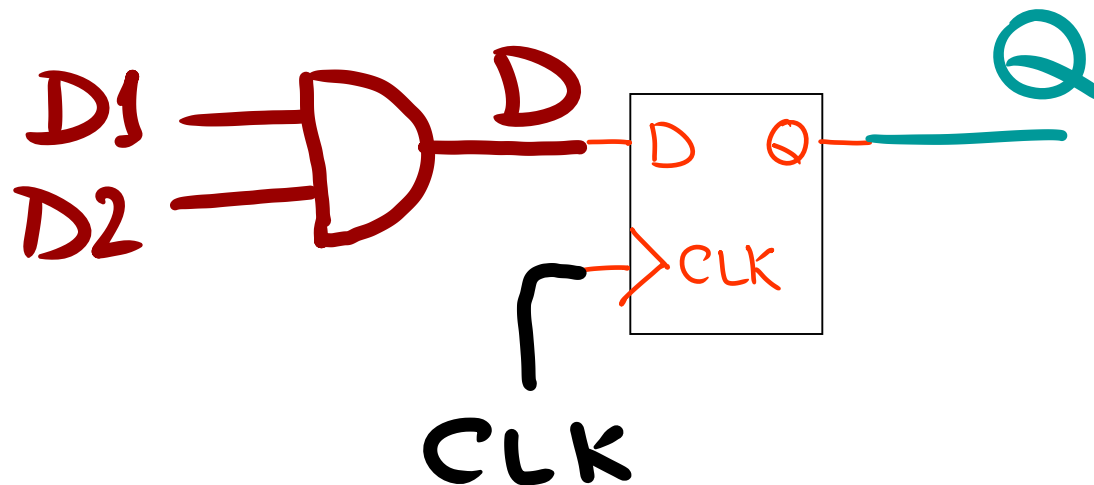
blocking  
assignments  
only

final assignments

# blocking and non-blocking assignments in clocked always blocks

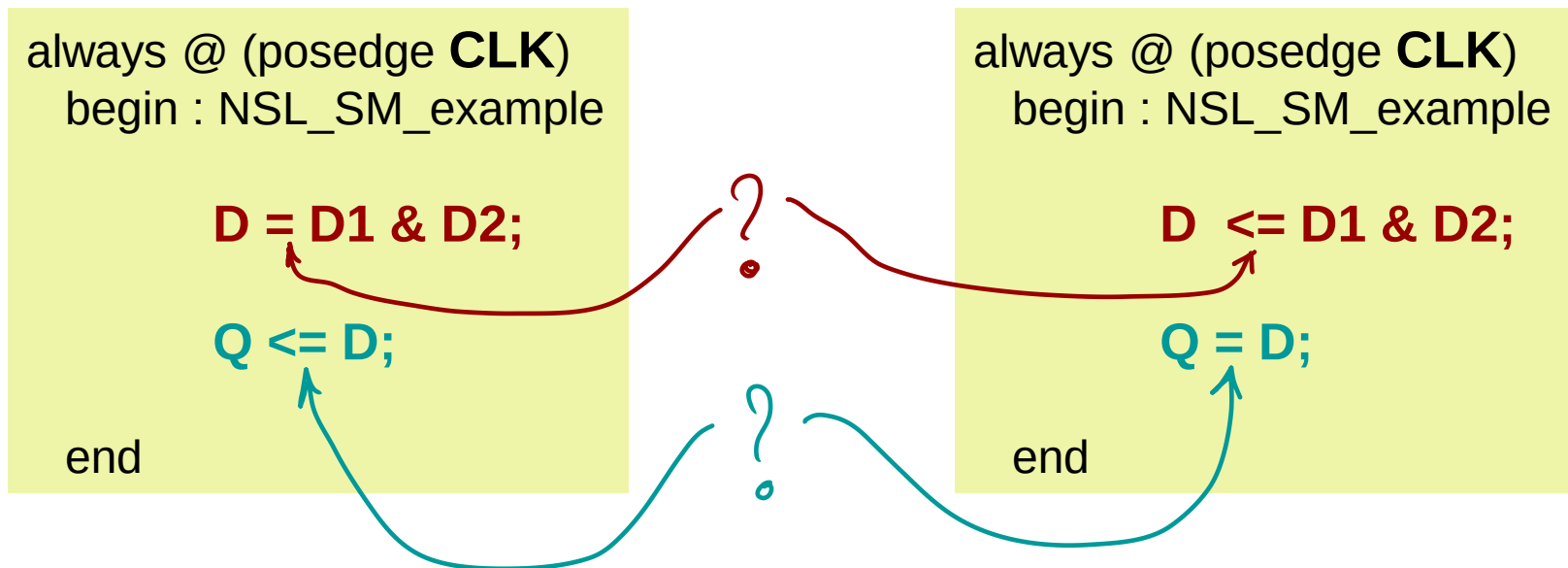
- For **physical registers**, use **non-blocking** assignments only to prevent race condition and to prevent discrepancy between simulation and synthesis
- In the processes of producing values to be registered, if you use **intermediate variables**, then use **blocking** assignments to produce the intermediate variables.

# blocking and non-blocking assignments in clocked always blocks



let us code the above in one  
always block.

# blocking and non-blocking assignments in clocked always blocks



# blocking and non-blocking assignments in clocked always blocks

Good code. D is an intermediate variable and does not cause register inference.

Some describe the D as a collapsible or removable variable. Only one FF for Q is inferred as desired in the diagram two pages before. Both synthesis and simulation tools interpret the code the same way.

```
always @ (posedge CLK)
begin : NSL_SM_example
```

**D = D1 & D2;** ✓

**Q <= D;** ✓

end

**Some think that  
both are OK.**

Synthesis tools will infer two FFs, one for D and one for Q

Simulation tools vary in their treatment of this bad code. Some may convey the new value for Q to other always blocks and some may convey the old. Simulation/synthesis discrepancy is very likely to happen.

```
always @ (posedge CLK)
begin : NSL_SM_example
```

Will change after Delta-T → **D <= D1 & D2;** ✗

**Q = D;** ? **not ok**  
old D

end

Other clocked always blocks may have Q on the RHS. You do not want to cause RACE condition nor simulation synthesis discrepancy.

# Golden rule:

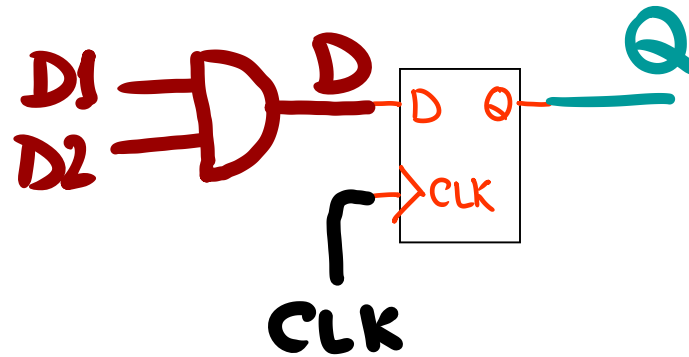
Every non-blocking assignment in a clocked processes will result in a physical register!

```
always @ (posedge CLK)
begin : NSL_SM_example
```

```
    D = D1 & D2;
```

```
    Q <= D;
```

```
end
```

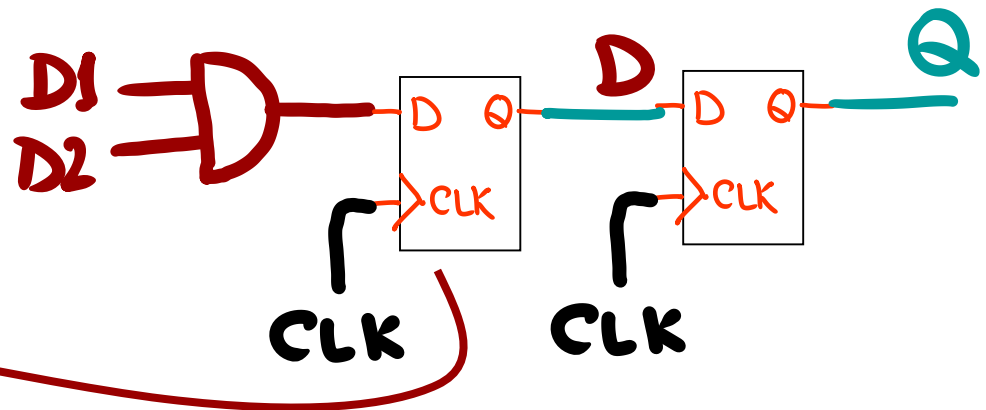


```
always @ (posedge CLK)
begin : NSL_SM_example
```

```
    D <= D1 & D2;
```

```
    Q <= D;
```

```
end
```



# intermediate variables?

Then, do not try to read them before you assign to them!

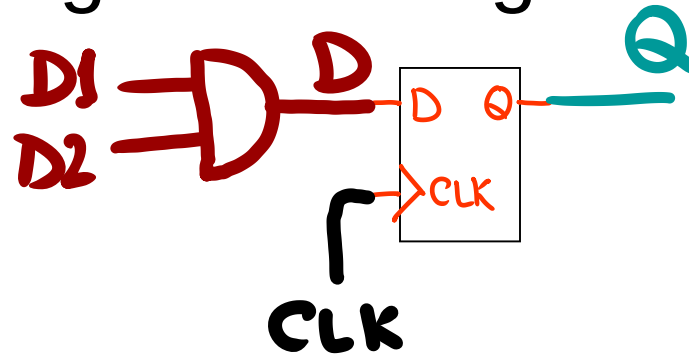
Otherwise you get extra registers!

```
always @ (posedge CLK)
begin : NSL_SM_example

    D = D1 & D2;

    Q <= D;

end
```

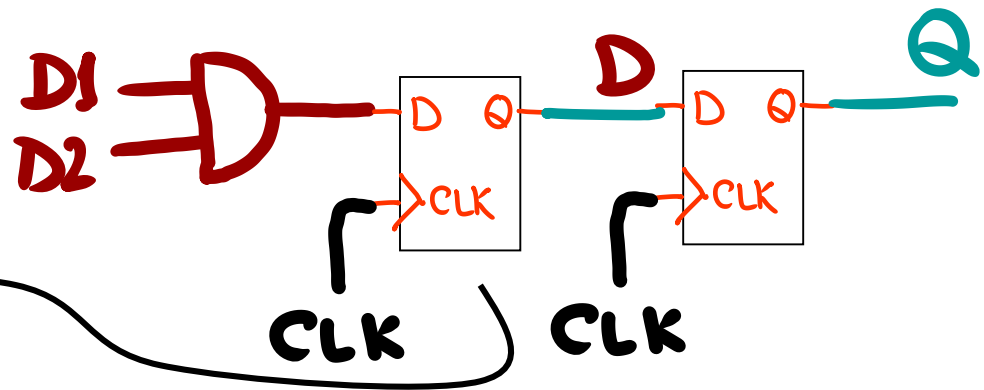


```
always @ (posedge CLK)
begin : NSL_SM_example

    Q <= D; ← old D

    D = D1 & D2;

end
```



intermediate variables in a clocked block?  
Then, treat them as local and do not **refer to**  
them from outside!

```
always @ (posedge CLK)
begin : NSL_SM_example

    D = D1 & D2;

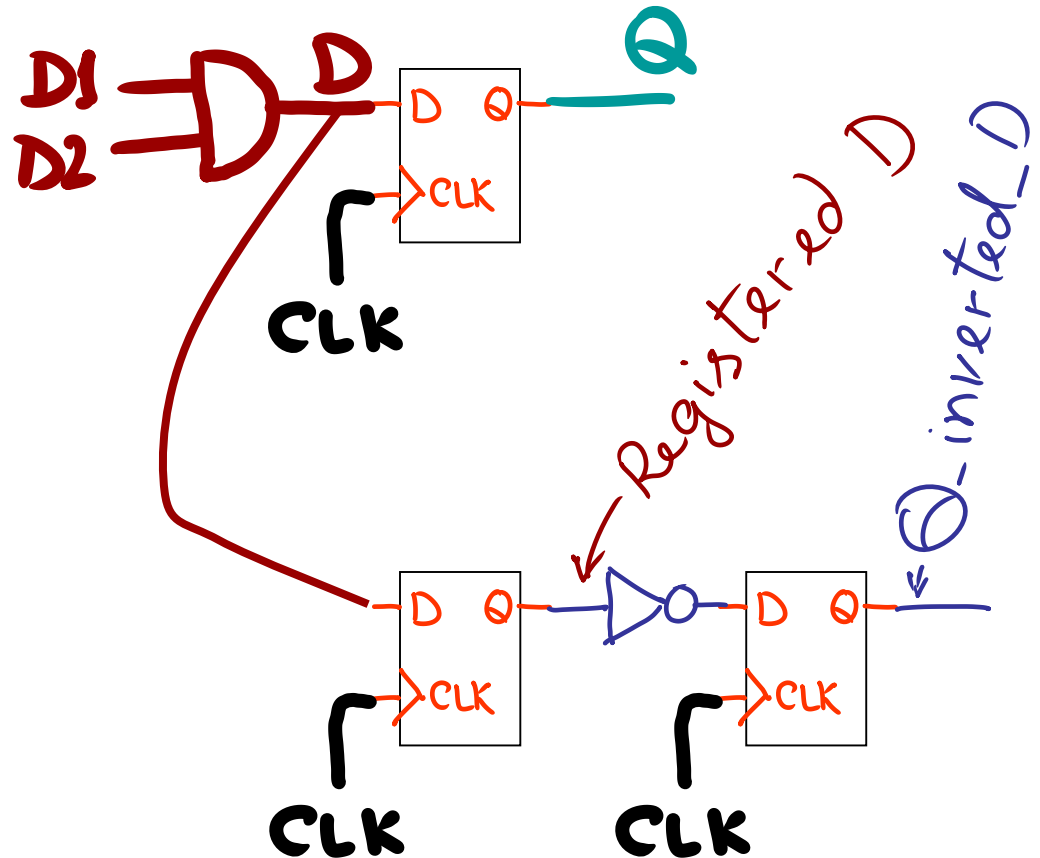
    Q <= D;

end
```

```
always @ (posedge CLK)
begin : NSL_SM_example

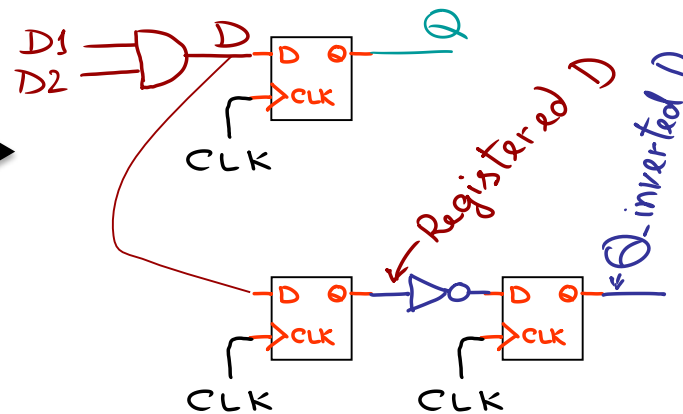
    Q_inverted_D = ~ D;

end
```



Undesirable use of D here, which is an intermediate variable in another always block.

If you really wanted this hardware, then code it as..



```
always @ (posedge CLK)
begin : NSL_SM_example

    D = D1 & D2;
    Q <= D;

end
```

```
always @ (posedge CLK)
begin : NSL_SM_example

    D = D1 & D2;
    DR <= D;
    Q <= D;

end
```

```
always @ (posedge CLK)
begin : NSL_SM_example

    Q_inverted_D = ~ D;

end
```

```
always @ (posedge CLK)
begin : NSL_SM_example

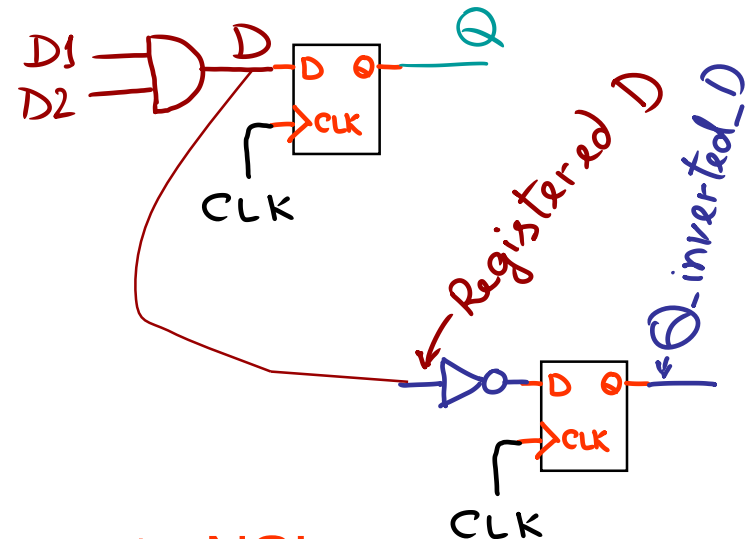
    Q_inverted_D = ~ DR;

end
```

If you wanted this hardware, then code **all** NSL and both registers **in one always block** or code the **NSL separately** in a combinational block.

```
always @ (posedge CLK)
begin : NSL_SM_example

    D = D1 & D2;
    Q <= D;
    Q_inverted_D <= ~ D;
end
```

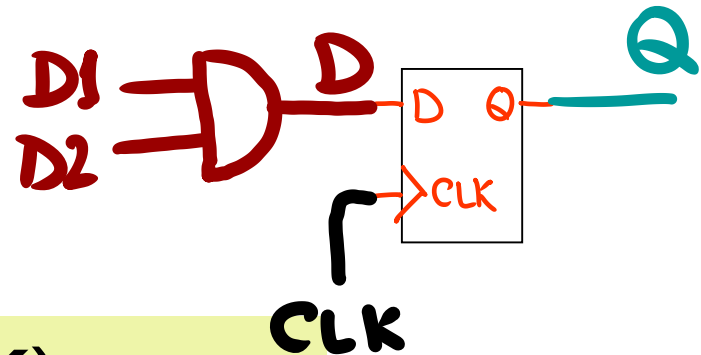


**assign D = D1 & D2;** ← **Separate NSL**

```
always @ (posedge CLK)
begin : SM_example
    Q <= D;
    Q_inverted_D <= ~ D;
end
```

If D is an intermediate variable, and you if do not want anyone to access it by mistake, then restrict its visibility!

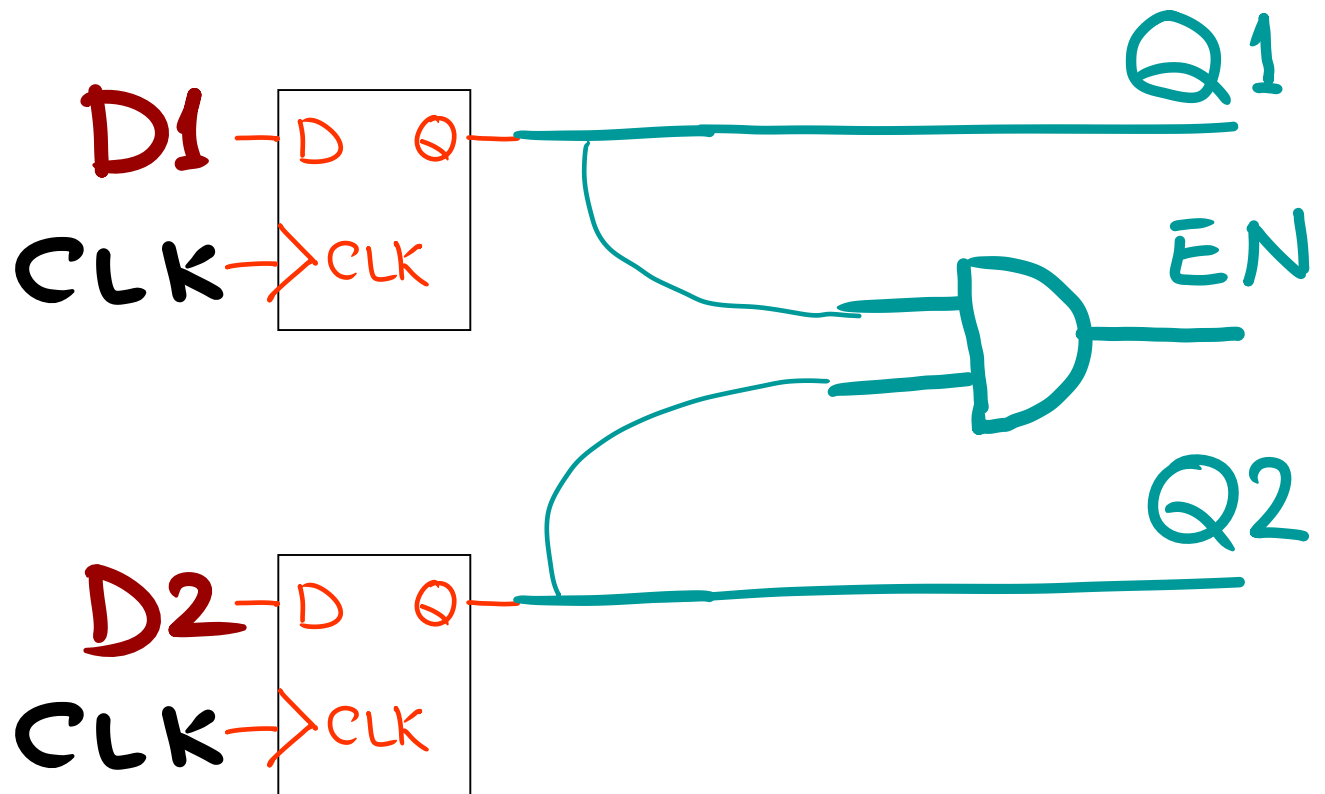
named begin ... end block can  
declare local variables



```
always @ (posedge CLK)
begin : NSL_SM_example
    reg D; // local declaration
    D = D1 & D2;
    Q <= D;
end
```

Preferred practice  
(follows VHDL variable  
declaration inside a process)

# Can you code all this in one always clocked block?



```
always @ (posedge CLK)
begin : SM_OFL_example
```

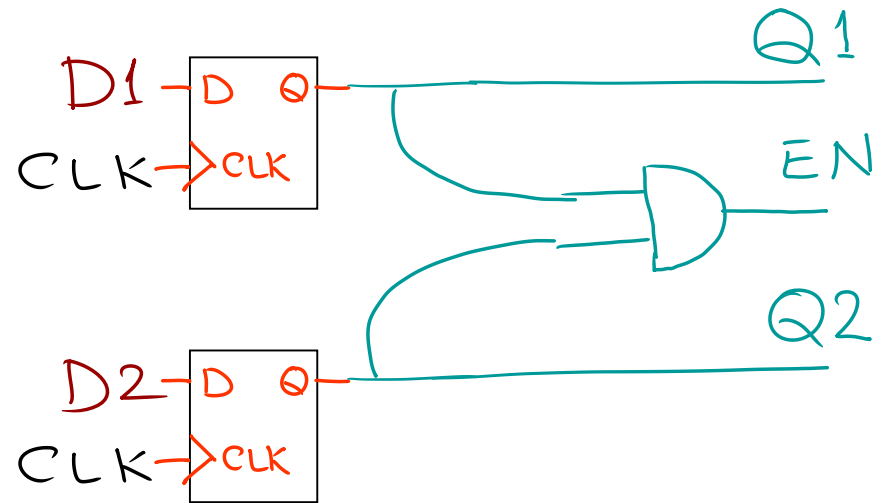
?

```
Q1 <= D1;
```

```
Q2 <= D2;
```

```
EN <= Q1 & Q2;
```

```
end
```



For each of the above statements,  
do you want to use **blocking** or  
**non-blocking** assignments?  
Or, is there something more fundamental  
that we are missing?

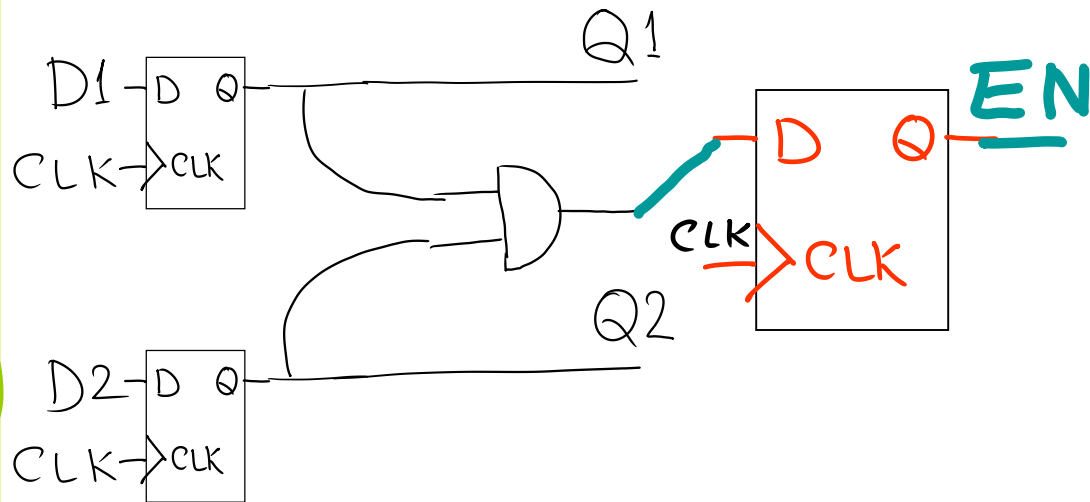
```
always @ (posedge CLK)
begin : SM_OFL_example
```

```
    Q1 <= D1;
```

```
    Q2 <= D2;
```

```
    EN <= Q1 & Q2;
```

```
end
```



**Any object (such as EN here) assigned in the clocked always block, *irrespective of whether it is assigned using a blocking assignment operator or non-blocking assignment operator, will result in a physical register*, if it is referred from outside of that block.**

If it is **not** referred (referred = read, placed on the RHS) from outside or inside, the synthesis or the implementation tool will treat it as a “waste” and will remove it, as it is not consumed either inside or outside the block where it is produced.

# What hardware is inferred by this?

```
always @ (posedge CLK)  
begin : SM_OFL_example  
  
    Q1 <= D1;  
  
    Q2 <= D2;  
  
    EN = Q1 & Q2;  
  
    if (EN) Q3 <= D3;  
  
end
```

```
always @ (posedge CLK)  
begin : SM_OFL_example
```

```
    Q1 <= D1;
```

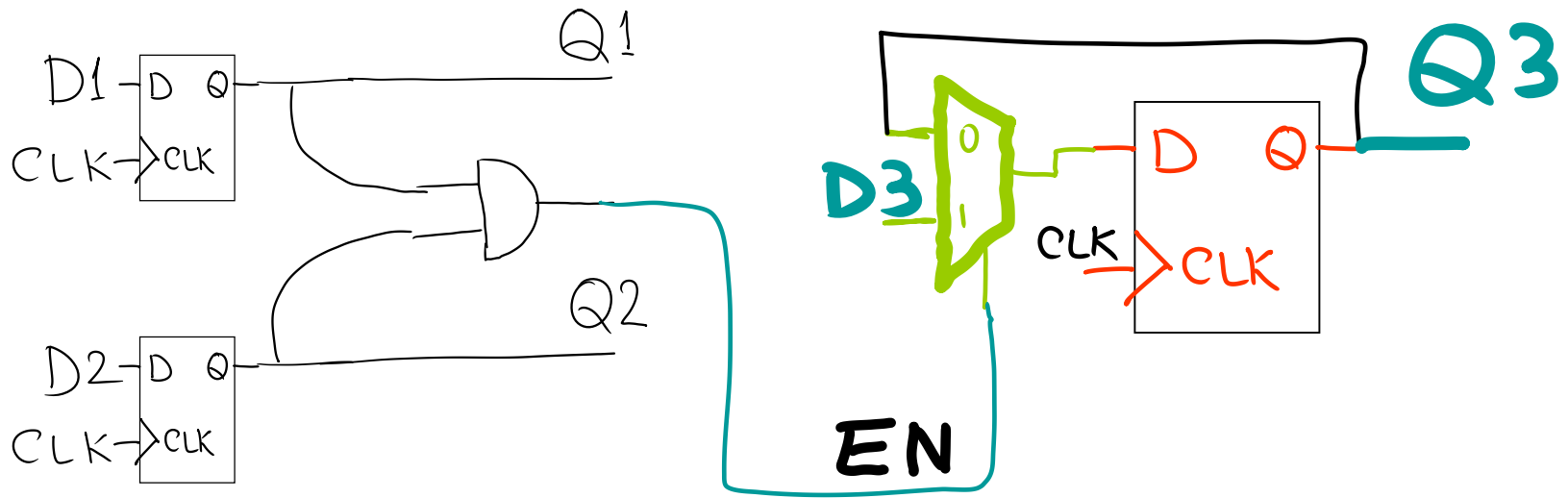
```
    Q2 <= D2;
```

```
    EN = Q1 & Q2;
```

```
    if (EN) Q3 <= D3;
```

```
end
```

hardware inferred

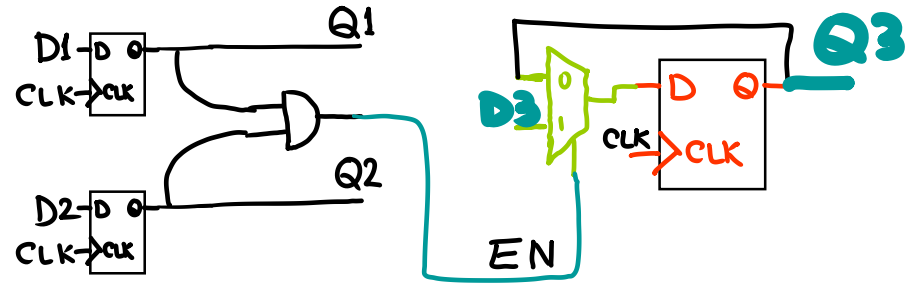


# which of the two codes infers the same hardware?

```
always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;

end

always @ (posedge CLK)
begin : SM_OFL_example_1B
    EN = Q1 & Q2;
    if (EN) Q3 <= D3;
end
```



```
always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;
    EN = Q1 & Q2;

end

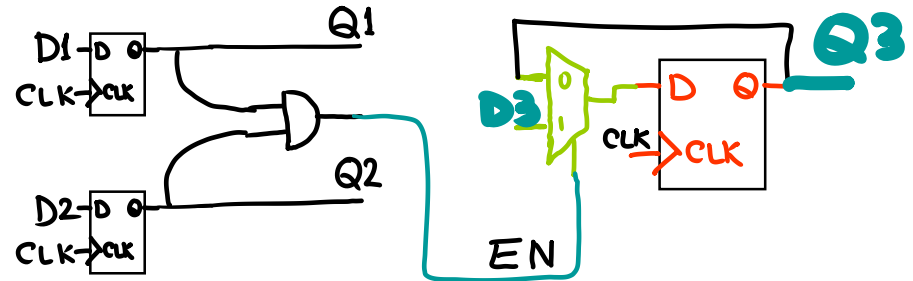
always @ (posedge CLK)
begin : SM_OFL_example_1B
    if (EN) Q3 <= D3;
end
```

# which of the two codes infers the same hardware

```
always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;

end

always @ (posedge CLK)
begin : SM_OFL_example_1B
    EN = Q1 & Q2;
    if (EN) Q3 <= D3;
end
```

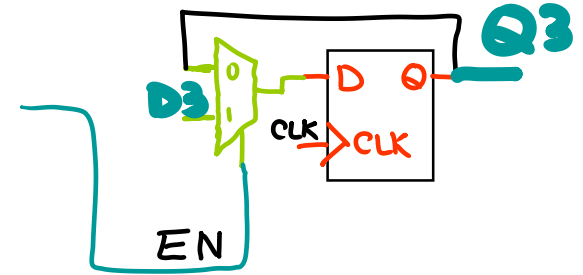
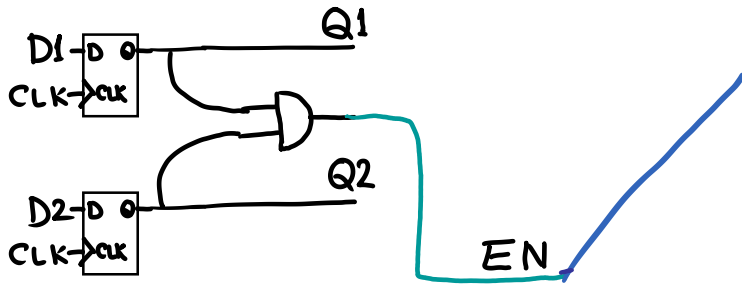


```
always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;
    EN = Q1 & Q2;
end

always @ (posedge CLK)
begin : SM_OFL_example_1B
    if (EN) Q3 <= D3;
end
```

# solution

# Then what hardware is inferred by this code?

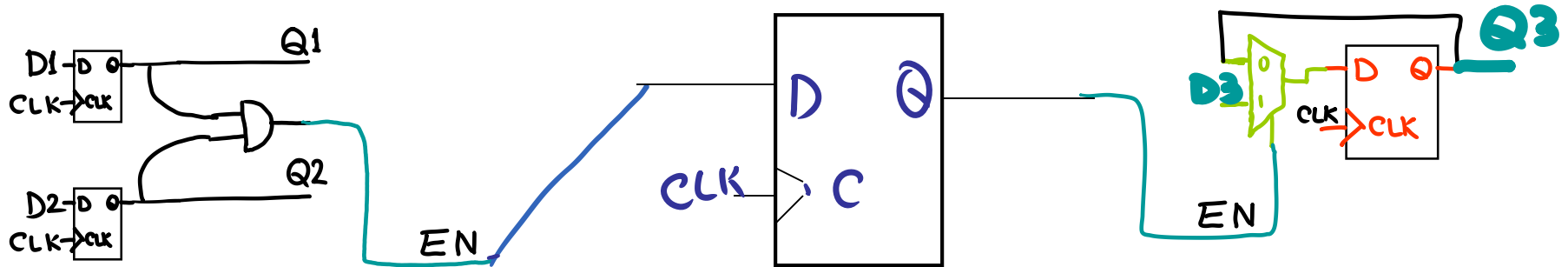


```
always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;
    EN = Q1 & Q2;
end
always @ (posedge CLK)
begin : SM_OFL_example_1B

    if (EN)  Q3 <= D3;

end
```

# Then what hardware is inferred by this code?



Synthesis tools definitely infer the above middle Flip-Flop.

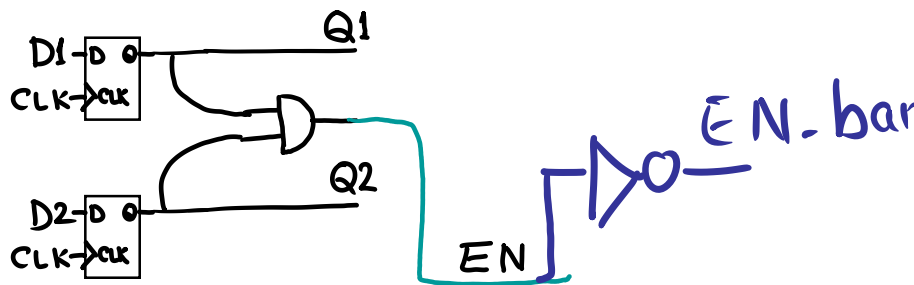
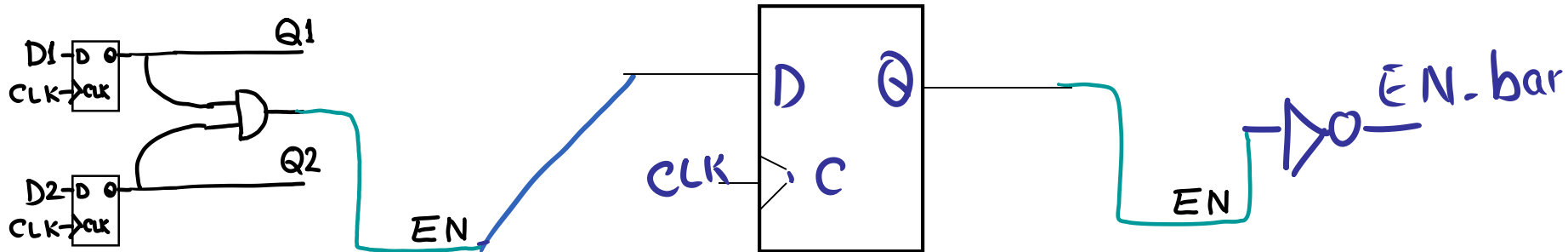
But some simulations tools inferred a FF, and some did not. And this may cause simulation/synthesis discrepancy ☹

In any case it is a bad coding practice and we should never do this.

## solution

```
always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;
    EN = Q1 & Q2;
end
always @ (posedge CLK)
begin : SM_OFL_example_1B
    if (EN) Q3 <= D3;
end
```

# What if the 2<sup>nd</sup> always block is combinational?



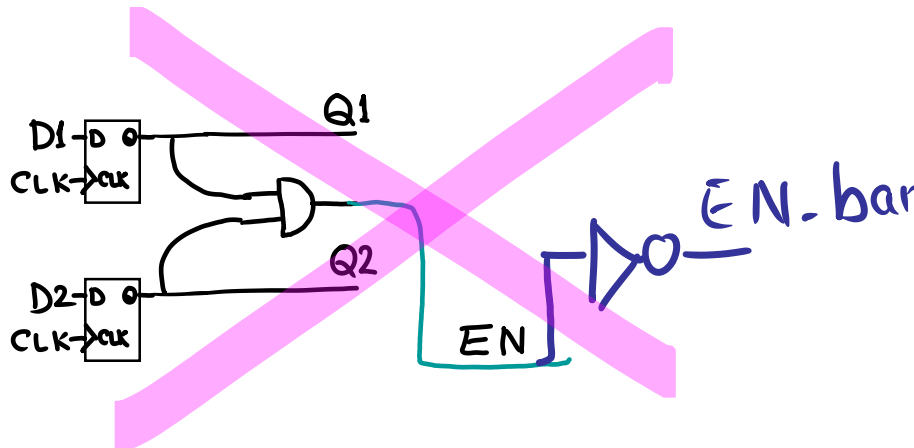
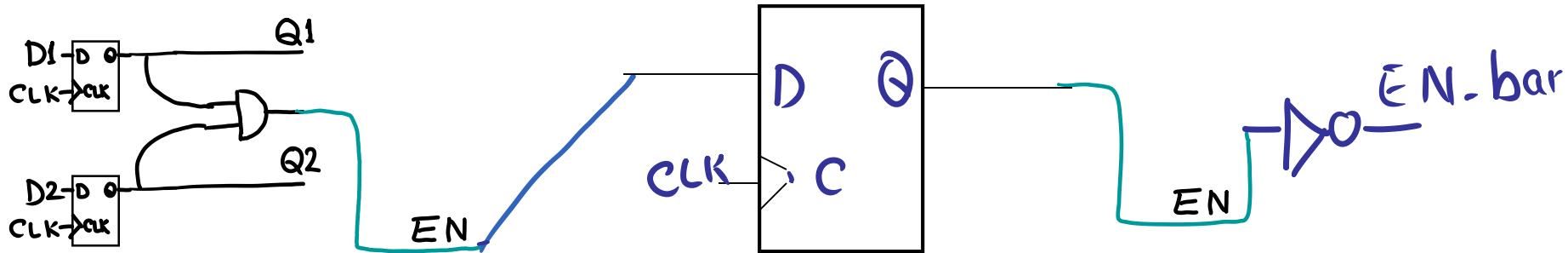
```

always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;
    EN = Q1 & Q2;
end
always @ (EN)
begin : SM_OFL_example_1B

    EN_bar <= ~ EN;

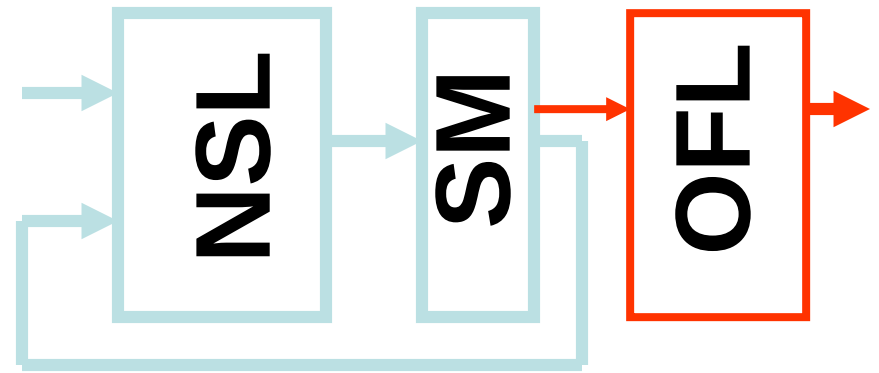
end
    
```

# What if the 2<sup>nd</sup> always block is combinational?



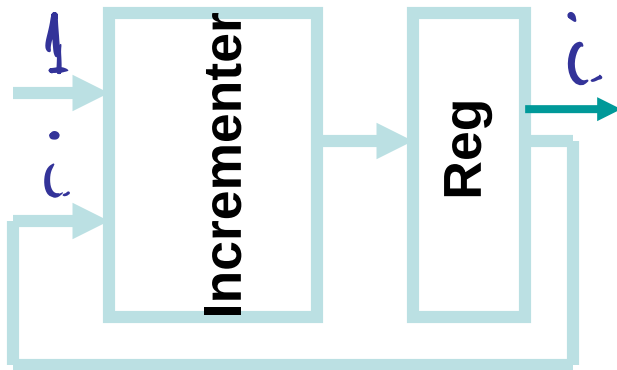
## solution

```
always @ (posedge CLK)
begin : SM_OFL_example_1A
    Q1 <= D1;
    Q2 <= D2;
    EN = Q1 & Q2;
end
always @ (EN)
begin : SM_OFL_example_1B
    EN_bar <= ~ EN;
end
```

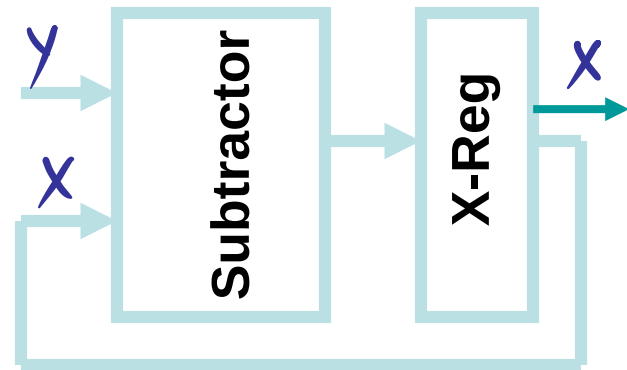


Generalization:

Any **upstream** logic to a register can be called a NSL!

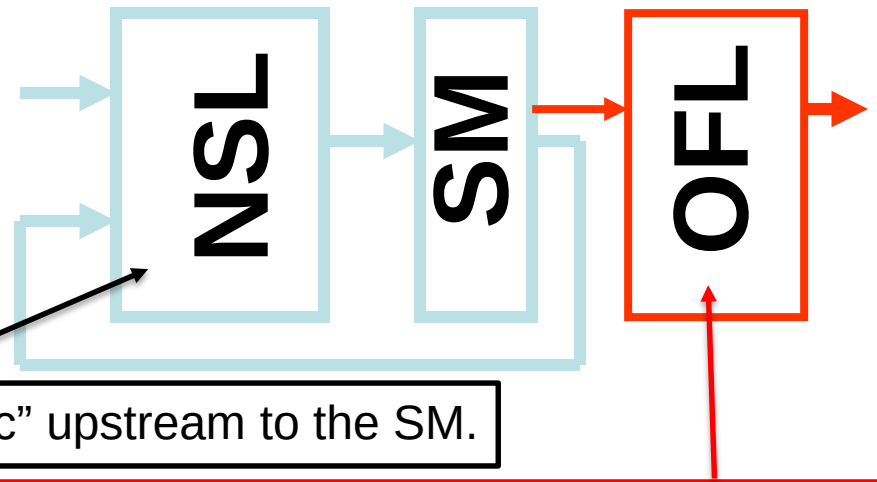


counter



Data path

# NSL, SM, OFL



NSL is the “Upstream Combinational Logic” upstream to the SM.

OFL is the “Downstream Combinational Logic” downstream to the SM.

SM goes in a clocked always block.

Its NSL (the SM’s NSL) can be coded along with the SM in the same clocked always block.

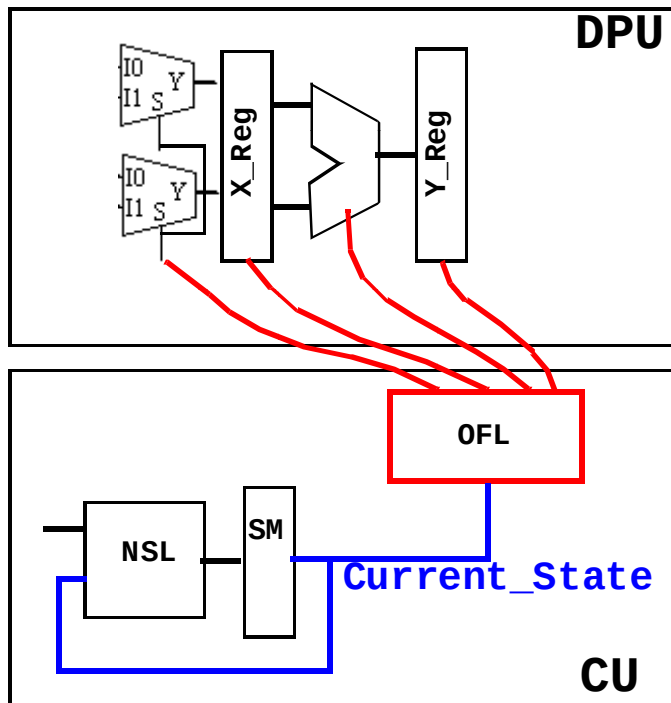
However, the OFL outputs are purely combinational and should not be coded along with the SM in the clocked always block. OFL should be coded separately in a combinational block (or produced using assign statements).

Or if the OFL is feeding into another NSL of another SM, then it can be combined in the coding of that SM under that clocked always block.

# Division between DPU and CU

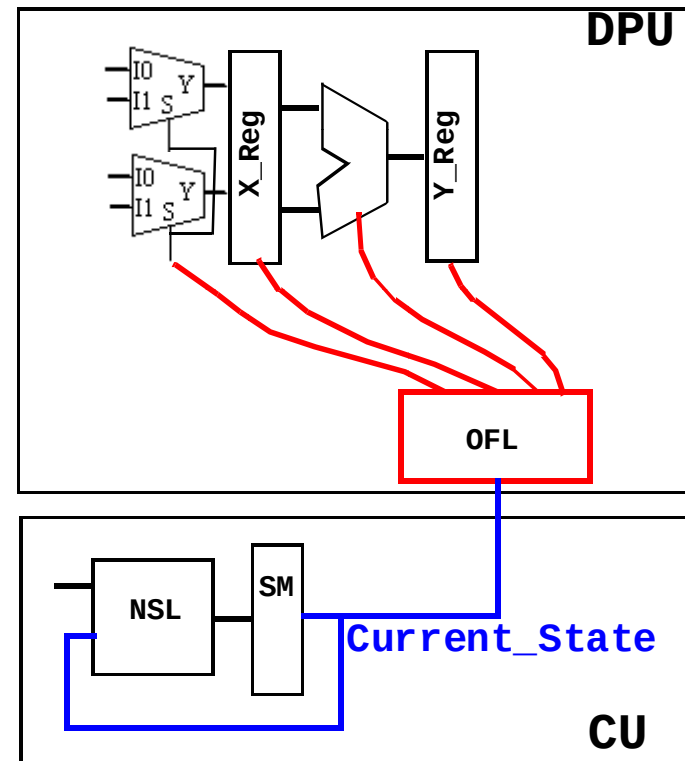
## Traditional division between DPU and CU

OFL (combinational logic)  
is in the CU.



## Division between DPU and CU for HDL coding

OFL (combinational logic)  
is moved to DPU. It is  
NOT coded explicitly.  
The OFL is implicit in the  
DPU's RTL in the CASE  
statement.



# Counter coding

Simple, all non-blocking assignments

```
always @(posedge CLK, posedge CLR)
begin    : COUNTER
    if (CLR)
        Count    <= 3'b000;
    else if (LOAD)
        Count    <= Load_input;
    else if (EN)
        Count    <= Count +1;
end
```

# Counter coding

## to illustrate need for proper mixing of blocking and non-blocking assignments

This is a made-up problem. It is a 5-bit truncated UP counter going from 0 to 20 and back to zero.

When the JUMP is asserted, the counter should jump to the mid-point M or a little beyond the midpoint M as detailed below. The midpoint M is the middle point between the current count and 20.

Example: Say, the current count is 12. Distance from 20 is 8. Half is 4. So the mid point M is at 16 ( $12+4 = 16$ ).

But what if the difference is an ODD number? Then you need to jump by either the "floor" or the "ceiling" of the half of the difference depending on whether the difference is less than ten or more than ten respectively.

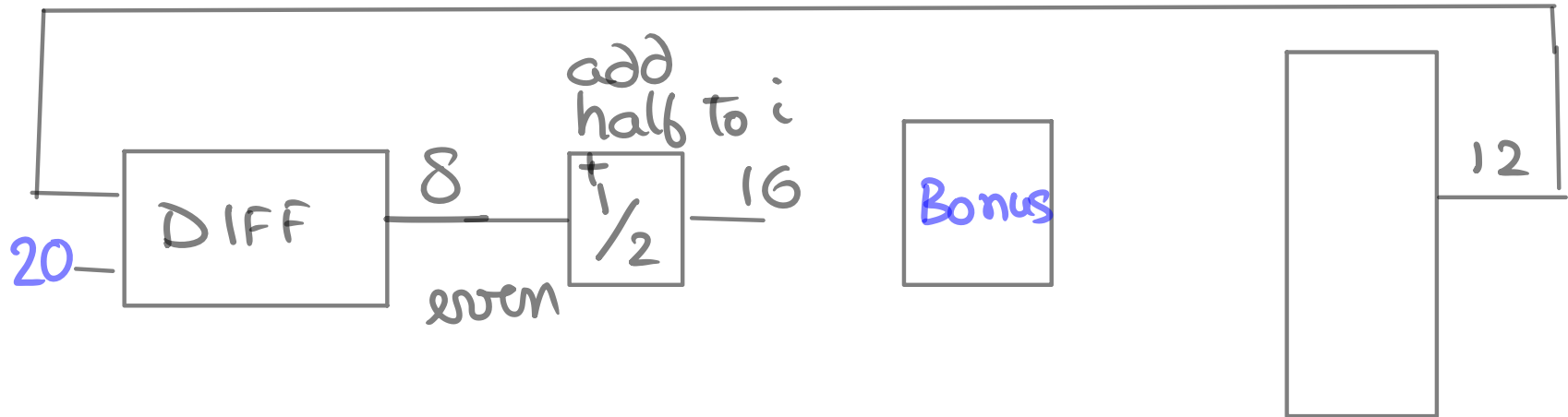
We said, "or a little beyond the mid point M".

After calculating the mid point M, a bonus to the jump by a constant of 4 is allowed if this bonus addition does not cause wrapping around (due to going beyond 20).

Examples: So, for the current count of 12, the jump takes you to 20, but, for a current count of 14, the jump takes you only to 17.

→ 0 → 20

## Special Counter



Part of the counter coding (in a clocked always block) illustrating proper mix of blocking and non-blocking assignments is presented on the next page.

```

if (jump)
begin
    Q_next = Q; // notice blocking assignment
    diff = 20 - Q_next; // (20 - Q) also works
    if (diff[0] == 0) // if the difference is even
        begin
            Q_next = Q_next + { 1'b0 , diff[4:1] };
        end
    else // if odd
        begin
            if (diff < 10)
                begin
                    Q_next = Q_next + { 1'b0 , diff[4:1] };
                end
            else
                begin
                    Q_next = Q_next + { 1'b0 , diff[4:1] } + 1'b1;
                end
        end
    if ( (20 - Q_next) >= 4)
        begin
            Q_next = Q_next + 4; // bonus 4
        end
    Q <= Q_next; // notice the non-blocking assignment
end

```