

Main Points in Verilog

0_Verilog_Main_Points_of_the_6_Lectures.pdf

1_Verilog_Introduction_mht.pdf

2_module_DataTypes_in_Verilog.pdf

3_behavioral_vs_structural_Verilog.pdf

4_Sequential_Statements_in_Verilog.pdf

5_blocking_non_blocking.pdf

6_RTL_coding_style.pdf

HDL

HDL = Hardware Description Language

Two HDLs: Verilog and VHDL

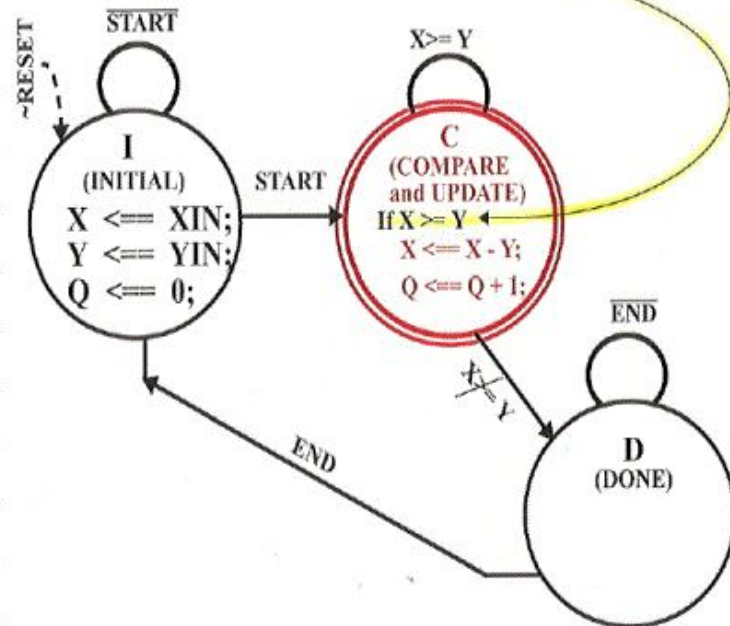
-- Simulation

-- Synthesis

Engineer

Engineer is responsible for design and design improvement.

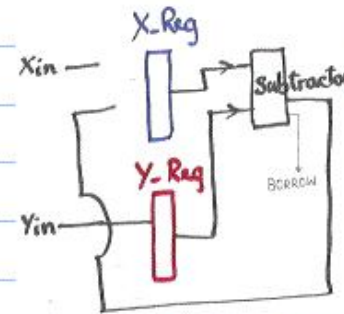
Divider -- Mealy machine design



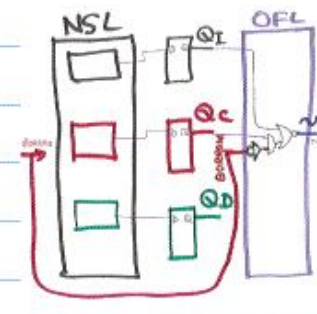
State diagram
with RTL

Technician

Technician is responsible for design detailing (DPU, CU, Timing design).



DPU



CU

Timing design

Setup Time check
Hold Time check

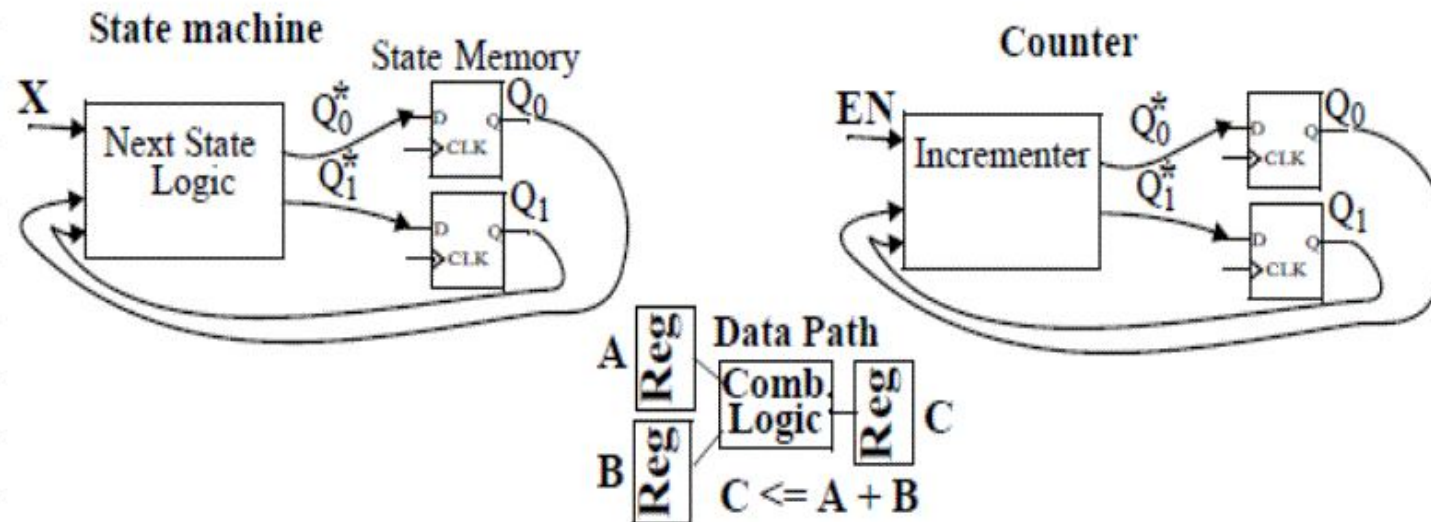
CAD Tools to synthesize

Now, CAD tools do what the skillful experienced technician was doing 30 years back. CAD tools can do much more tirelessly!

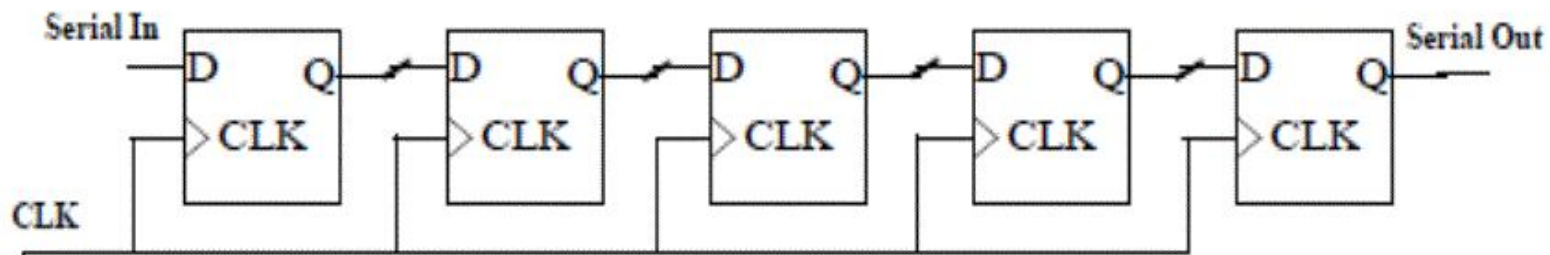
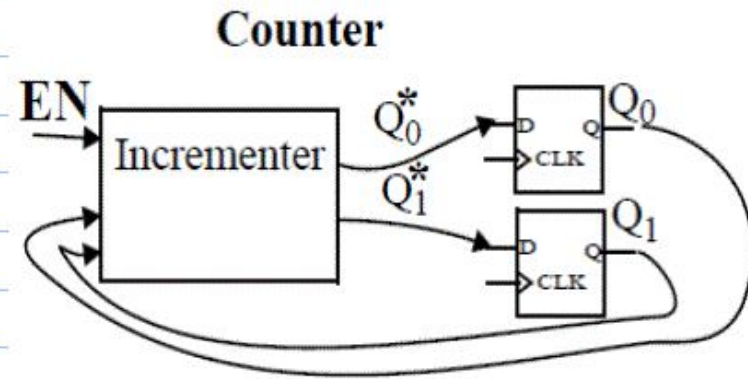
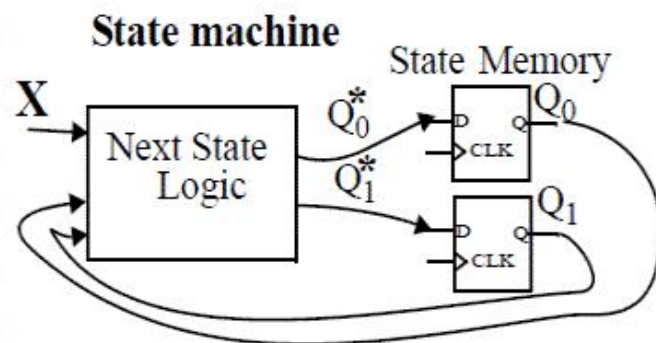
Pure Synchronous Design

Single Clock, all registers use the same edge of that single clock.

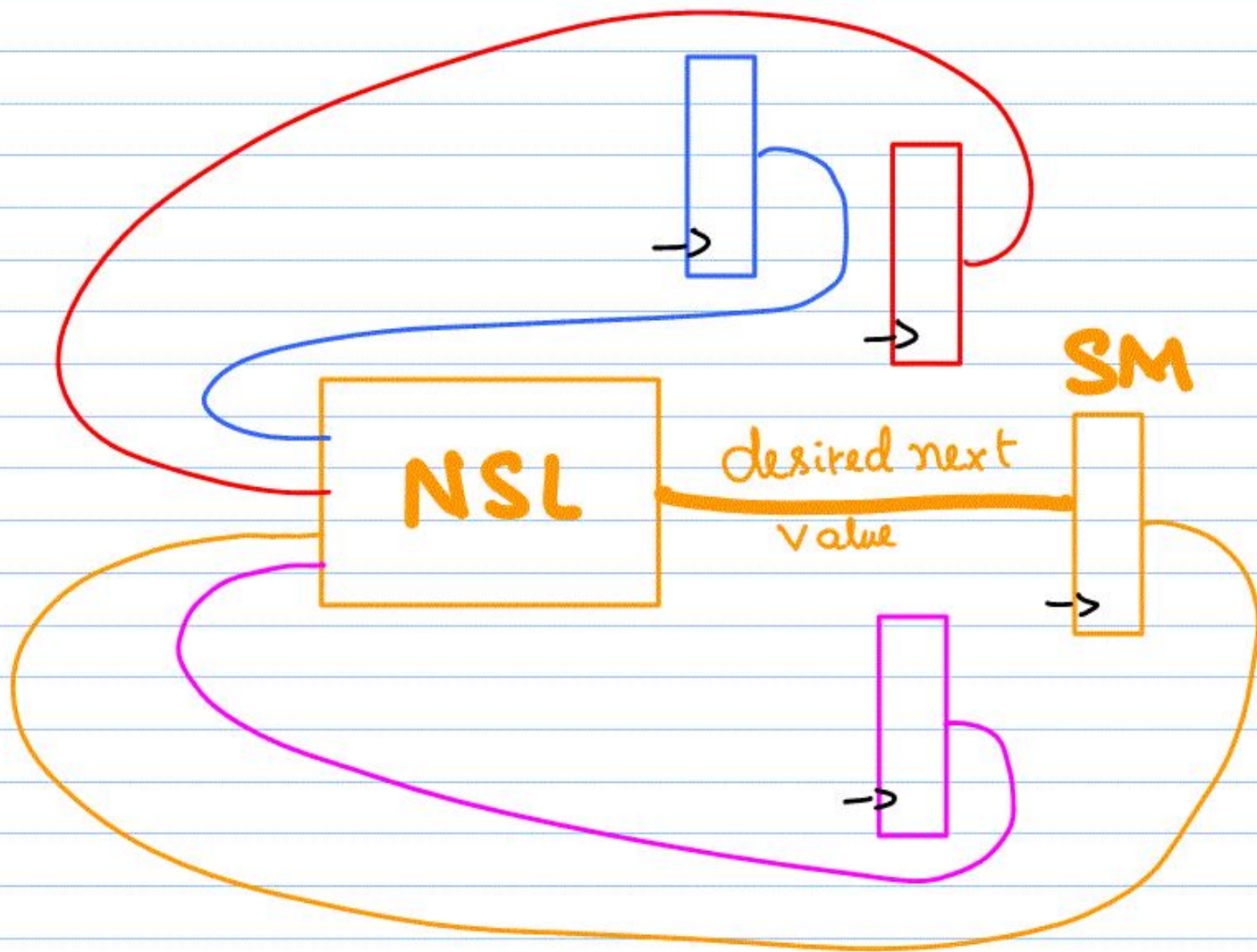
Design can be viewed as a bunch of registers with combinational logic in between them.



Note: Source registers themselves may be about to change!



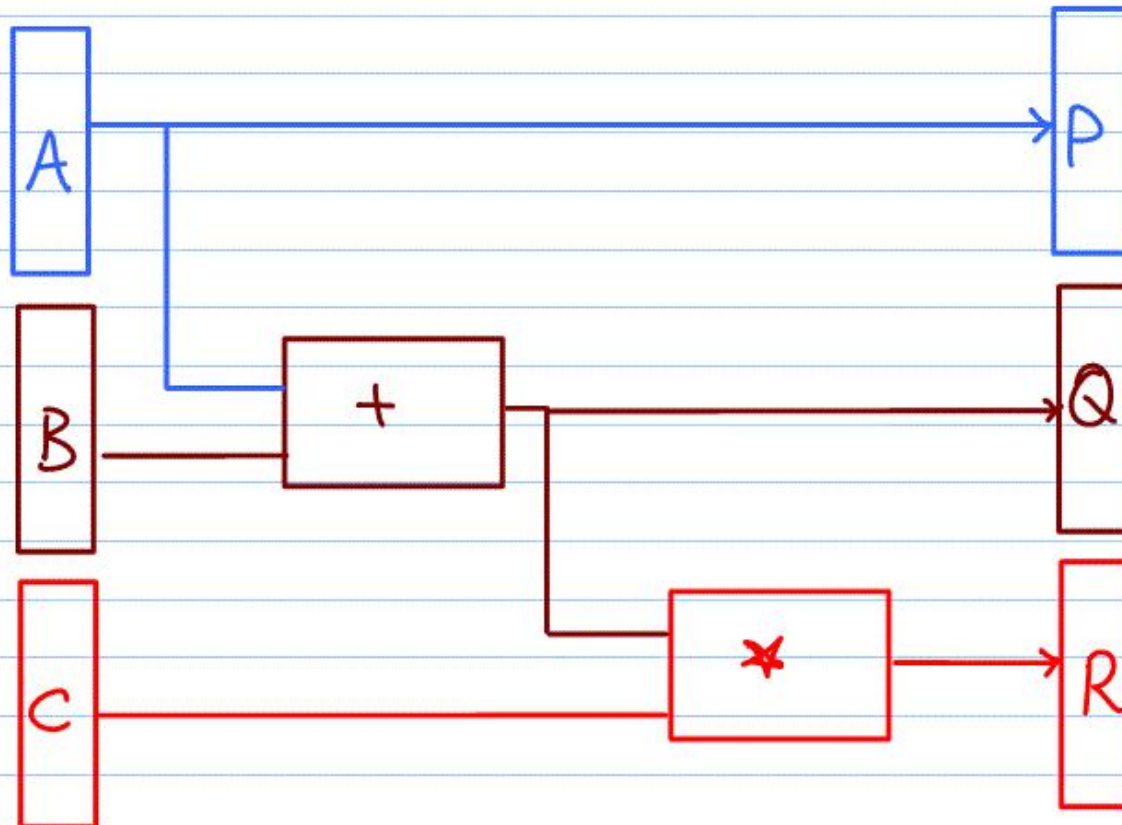
Relation between t_{ffpd} t_h



Timing Design

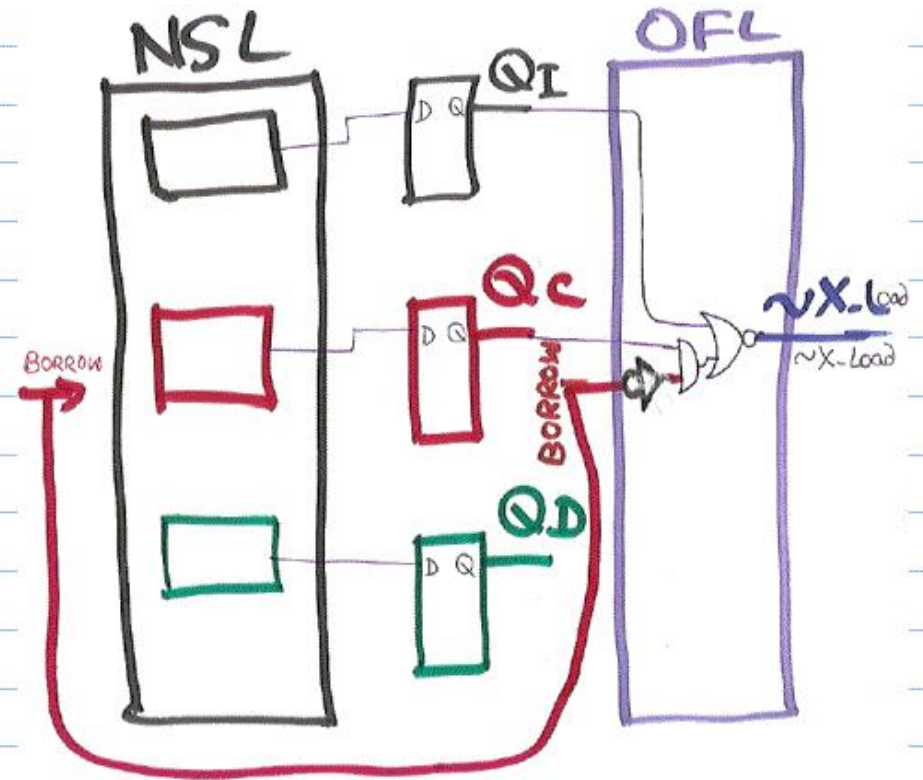
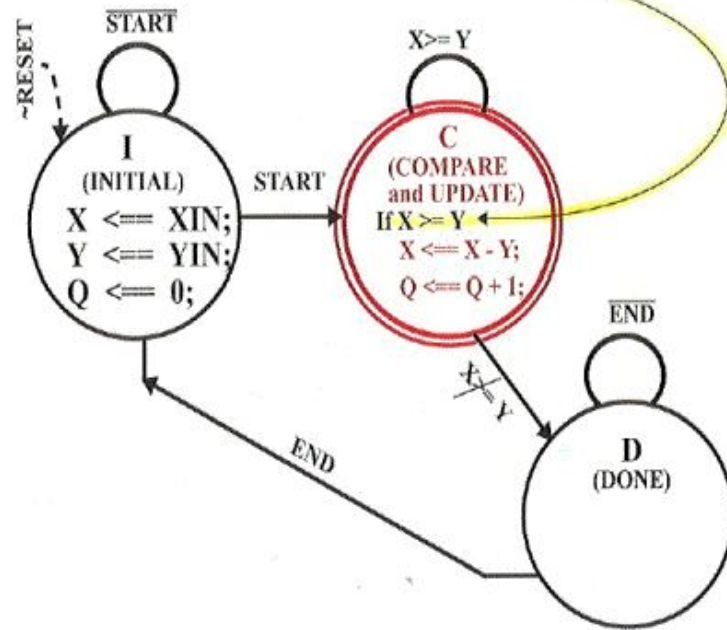
$$\text{Setup time Margin} = t_{\text{clk}} - t_{\text{ffPD(max)}} - t_{\text{comb(max)}} - t_{\text{setup(min)}}$$

$$\text{Hold time Margin} = t_{\text{ffPD(min)}} + t_{\text{comb(min)}} - t_{\text{hold(min)}}$$



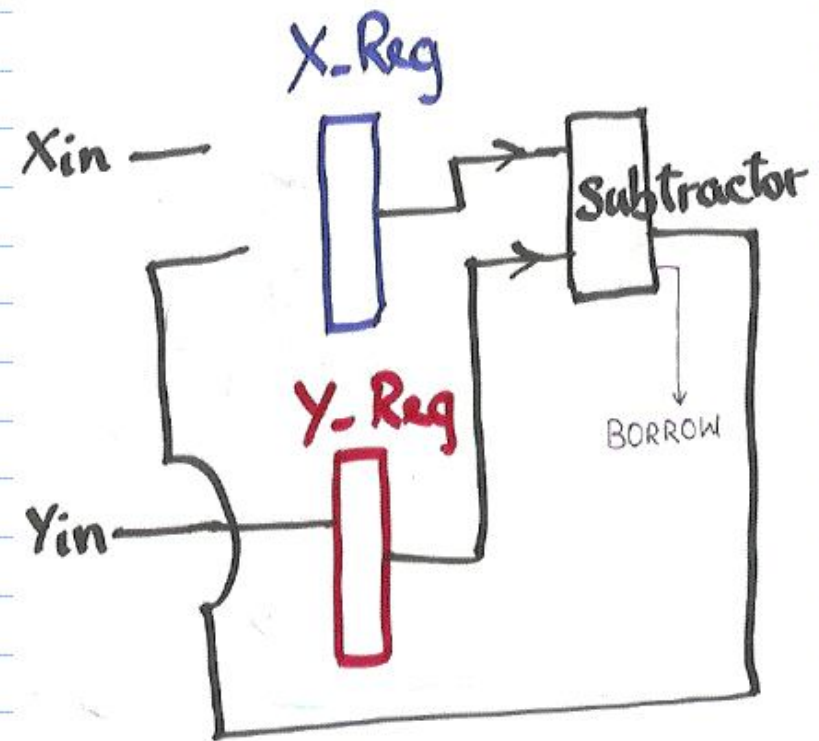
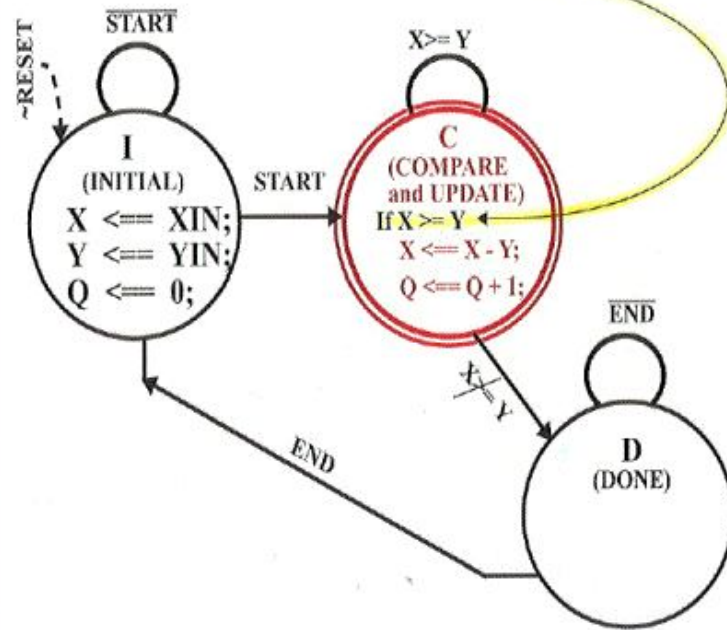
Synthesis

Divider -- Mealy machine design



Synthesis

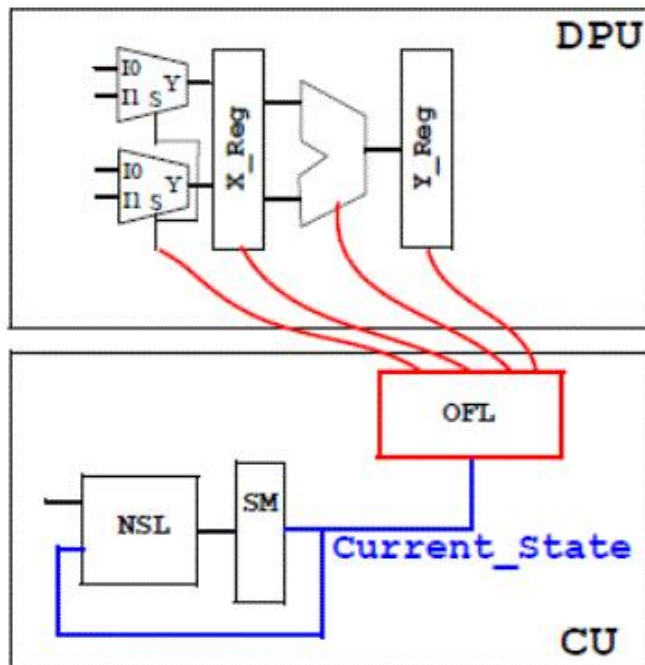
Divider -- Mealy machine design



Synthesis

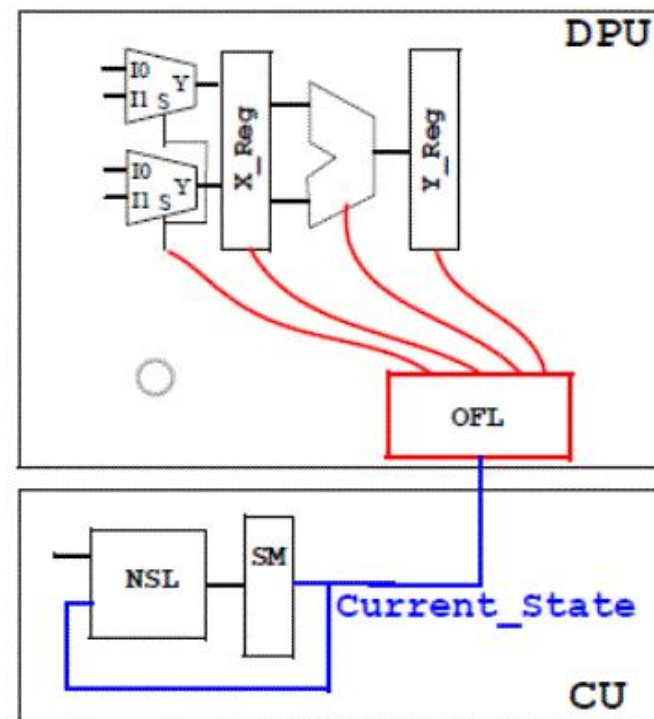
Traditional division between DPU and CU

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.



```
case (state)
  INITIAL :
    begin
      // state transitions in the control unit
      if (Start)
        state <= COMPUTE;
      // RTL operations in the Data Path
      x <= Xin;
      y <= Yin;
      Quotient <= 0;
    end
  COMPUTE :
    begin
```

Engineer

Engineer also needs simulation tools to verify his large complex designs.

Cycle-based Simulation

Even-driven simulation

Does the engineer follow pure synchronous design using a single-clock?
Then cycle-based simulation is good enough and is also fast.

Or does he have multiple clocks in his design and does he use gated-clocking, latch-based design using the two phases of the clock, etc.?
Then event-driven simulation is needed.

Event-driven simulation, though a little slower will cover all situations (pure synchronous designs as well as designs deviating from pure synchronous design methodology). Clock-gating became common in current design to save power. Also you want to perform timing simulation of the synthesized (after place and route) design to gain confidence of the synthesis process.

Hardware

is

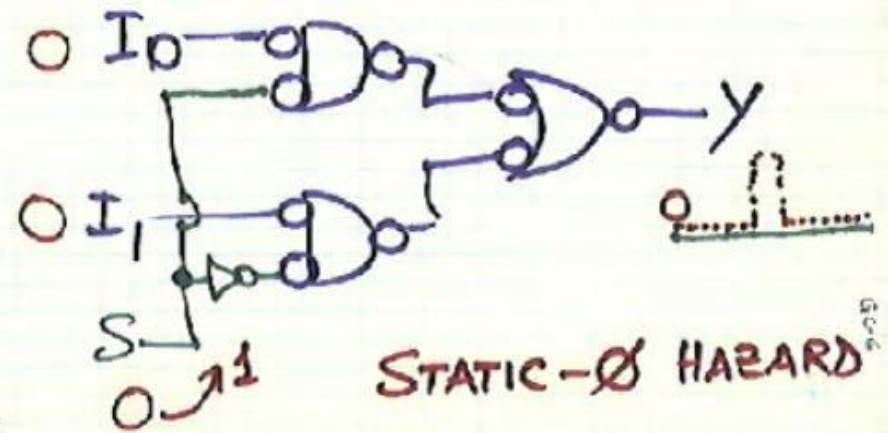
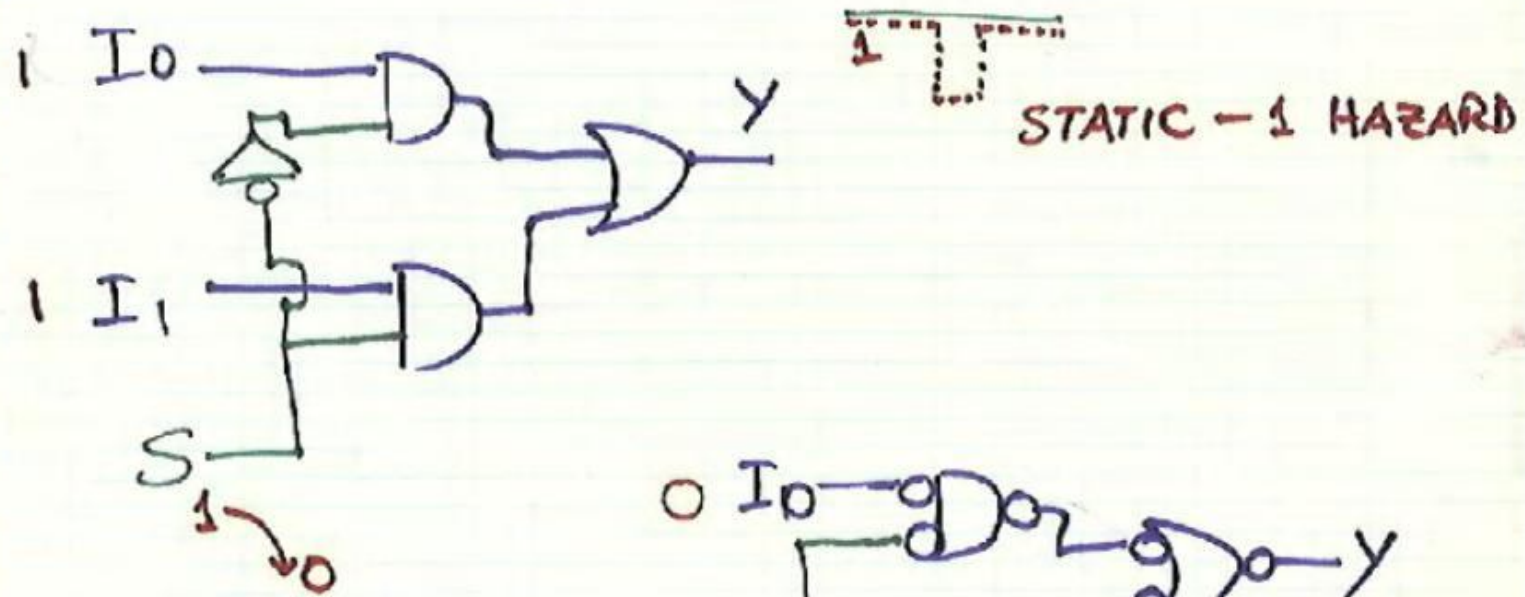
Concurrent.

An **HDL** should be able to
be able to

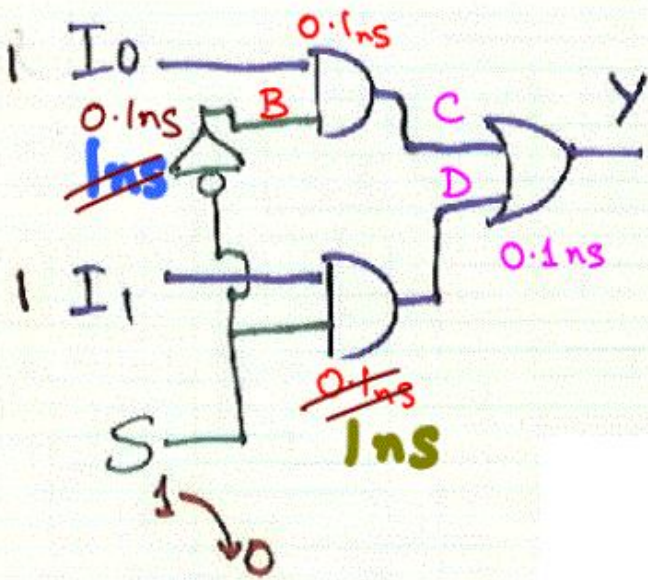
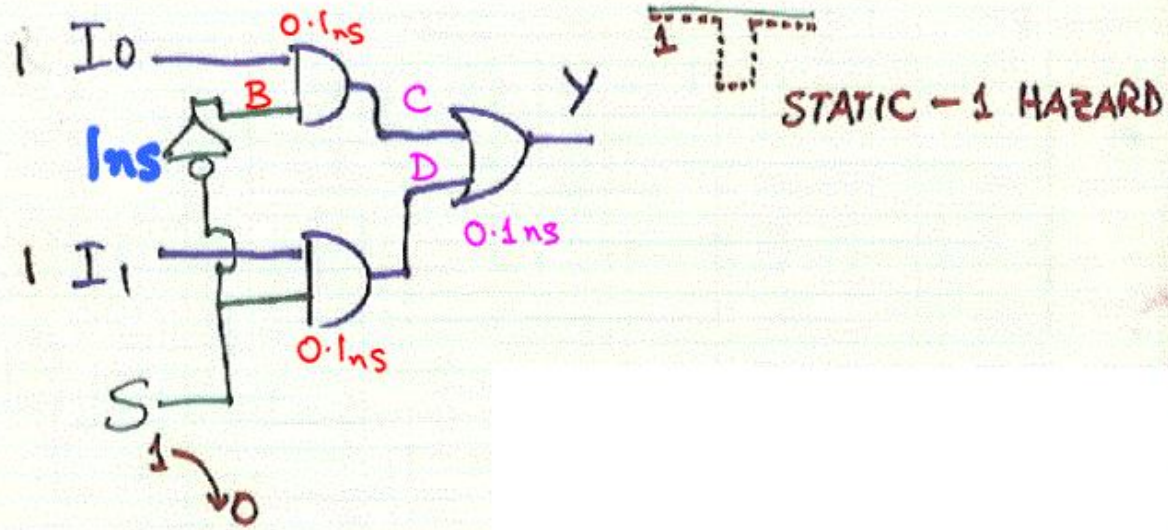
represent and **preserve**

this **Concurrent**
nature of Hardware.

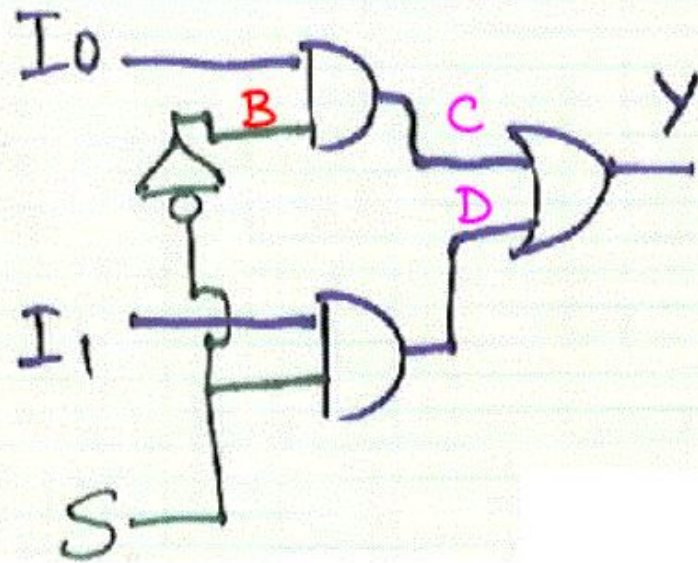
Glitches in Combinational logic



7a.



Do you get a glitch in both cases?



Operators

$\sim$ NOT $\neg$

$\&$ AND $\Rightarrow$

$|$ OR $\Rightarrow$

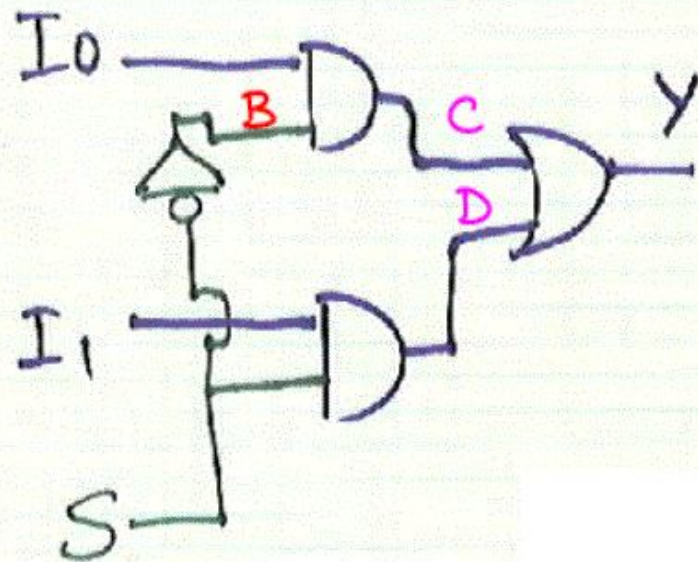
$\wedge$ XOR $\Rightarrow$

$\wedge \sim$ XNOR $\Rightarrow$

or
 $\sim \wedge$ XNOR $\Rightarrow$

$\frac{A}{B} \Rightarrow$ NAND

assign $c = \sim (A \& B);$



Operators

$\sim$	NOT	$\neg$
$\&$	AND	$\wedge$
$ $	OR	$\vee$

assign $B = \sim S;$

assign $C = I_0 \& B;$

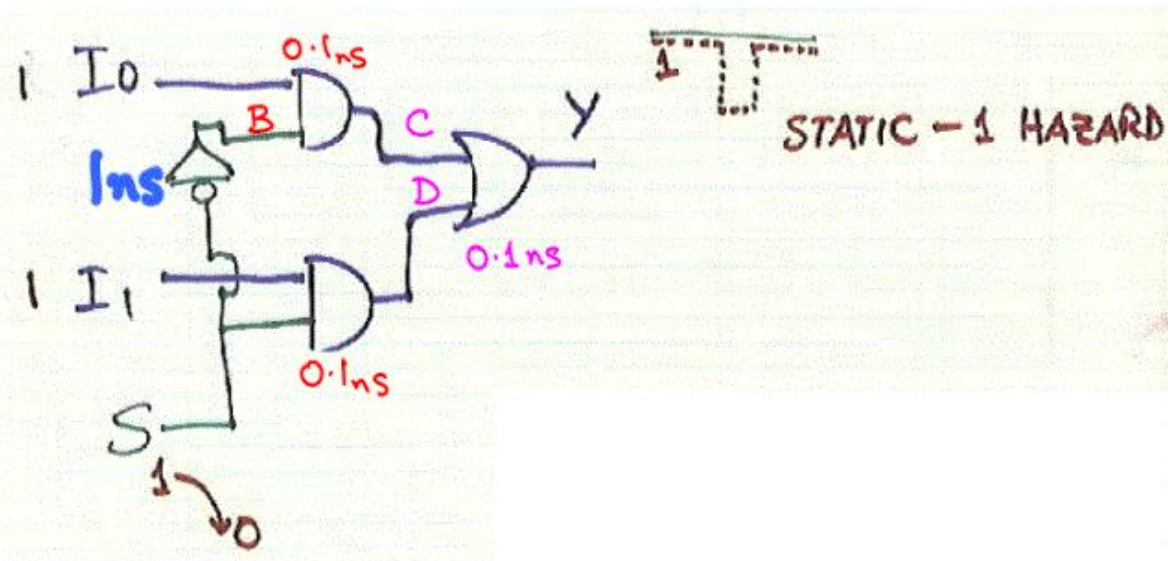
assign $D = I_1 \& S;$

assign $Y = C | D;$

4

Concurrent
Statements

Order does not
matter.



```
assign #1 B = ~S;
```

```
assign #0.1 C = I0 & B;
```

```
assign #0.1 D = I1 & S;
```

```
assign #0.1 Y = C | D;
```

Timescale 1ns/100ps

Units

Precision

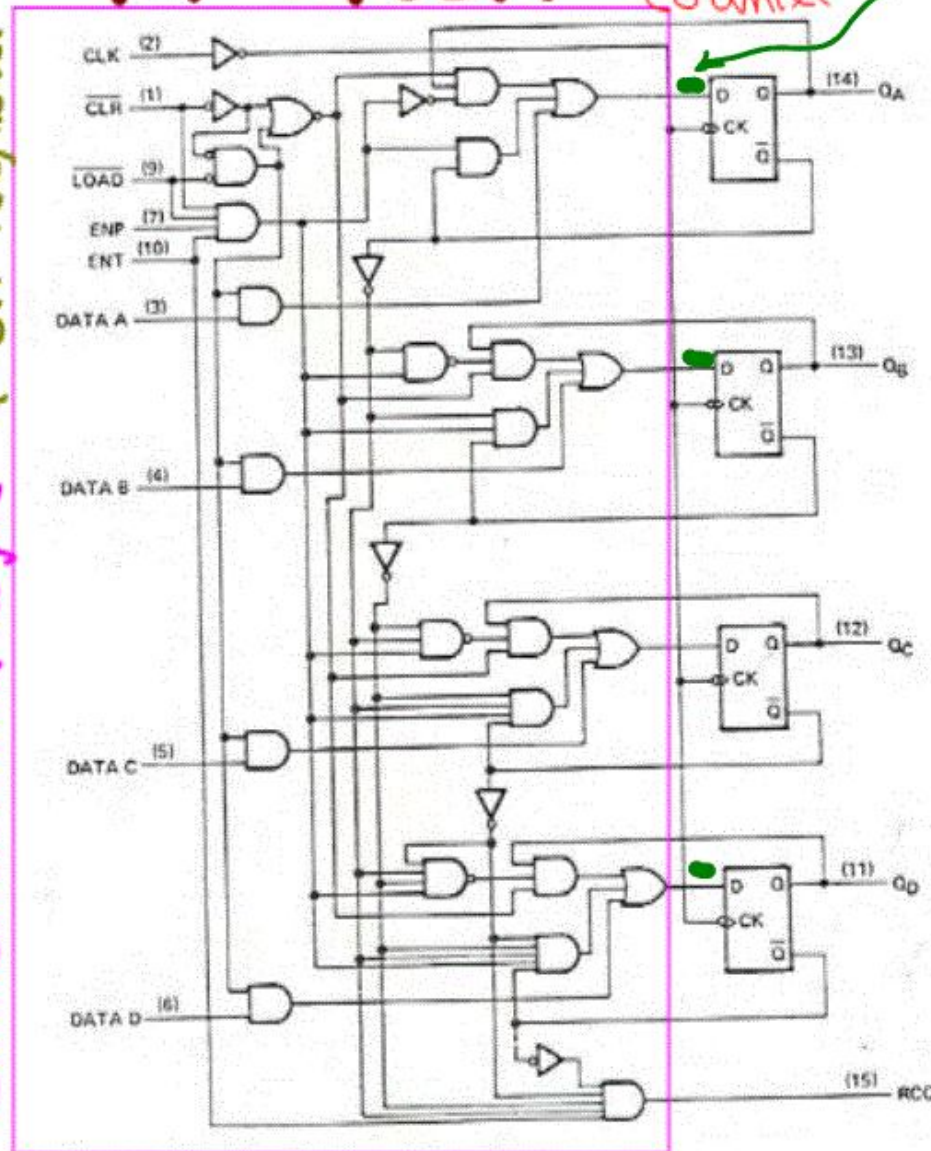
upstream of the
4-bit Register

Combinational logic

74LS163A

4-bit
counter

glitches?



If you are
given **delays** of
all **20+** gates,
and input stimulus
can you find
if and when
glitches are
produced?

EVENT DRIVEN Simulation of the CONCURRENT Statements

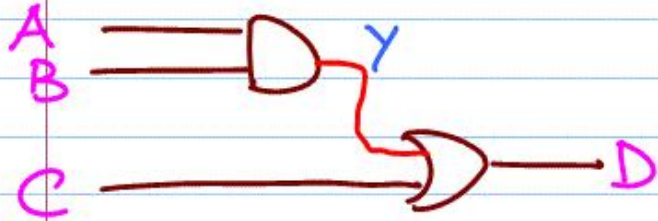
74LS163

20+ Assign Statements

Order of writing these

20+ statements does not matter.
They are all concurrent!

Data flow model

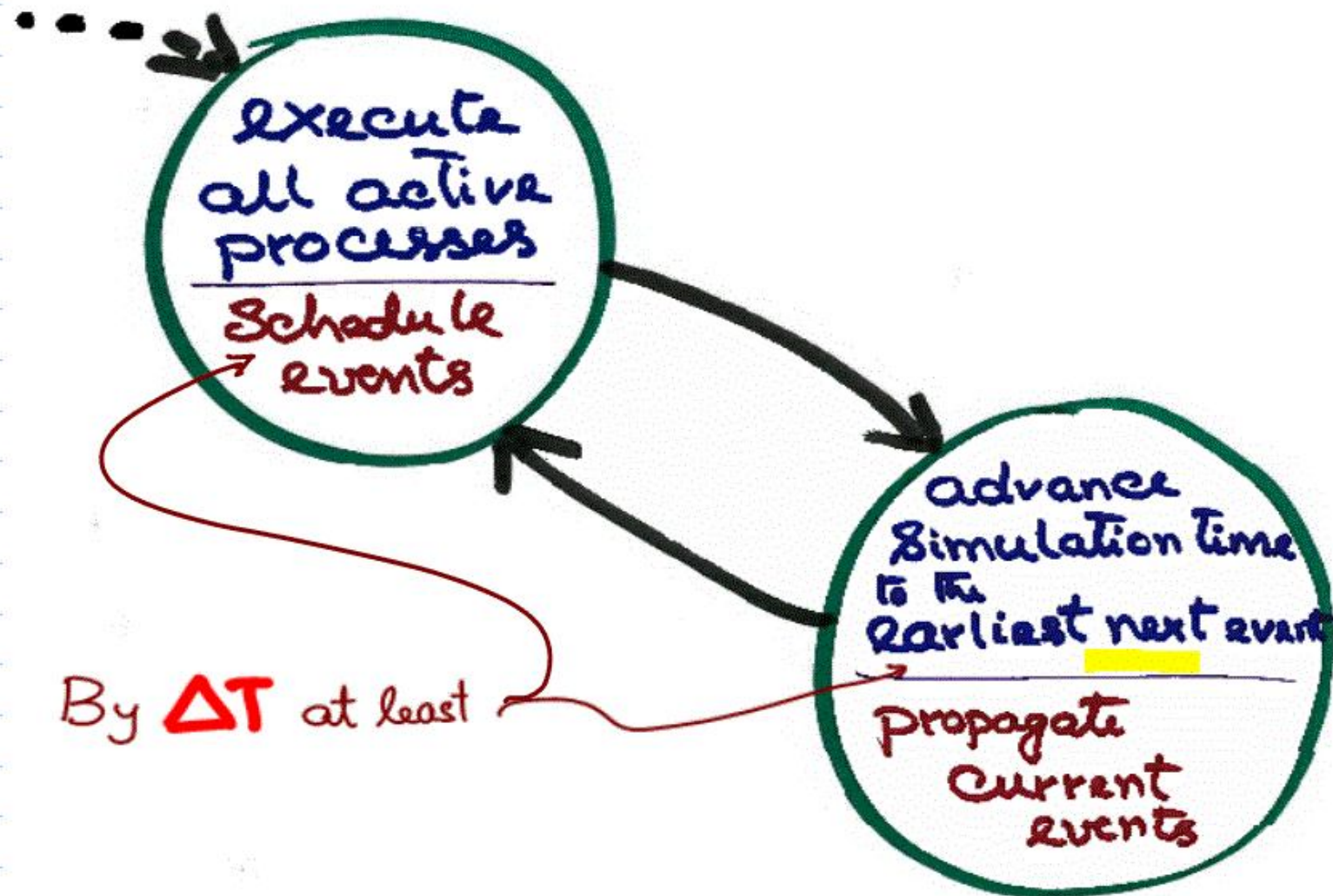


```
assign Y = A & B;  
assign D = Y | C;
```

```
assign D = Y | C;  
assign Y = A & B;
```

Order does not
matter as the
statements

are **CONCURRENT**.



Event-driven simulation

Freezing simulation time and executing the active concurrent items, one after another,
in the event-driven simulation

is similar to



drawing curtain and rearranging the stage (if only we can freeze the audience)

Theater Curtain. Stage Curtain. stock video



Scene 1

Theater Curtain. Stage Curtain. stock video



Curtain drawn =
Freeze simulation time and
evaluate all active processes

Theater Curtain. Stage Curtain. stock video

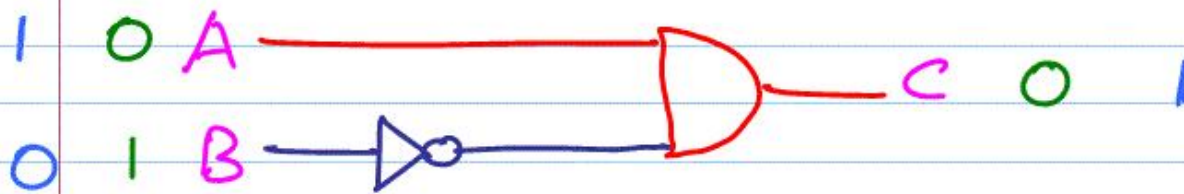


Scene 2

Behavioral/functional simulation before synthesis

ZERO-DELAY MODELING

ΔT delay



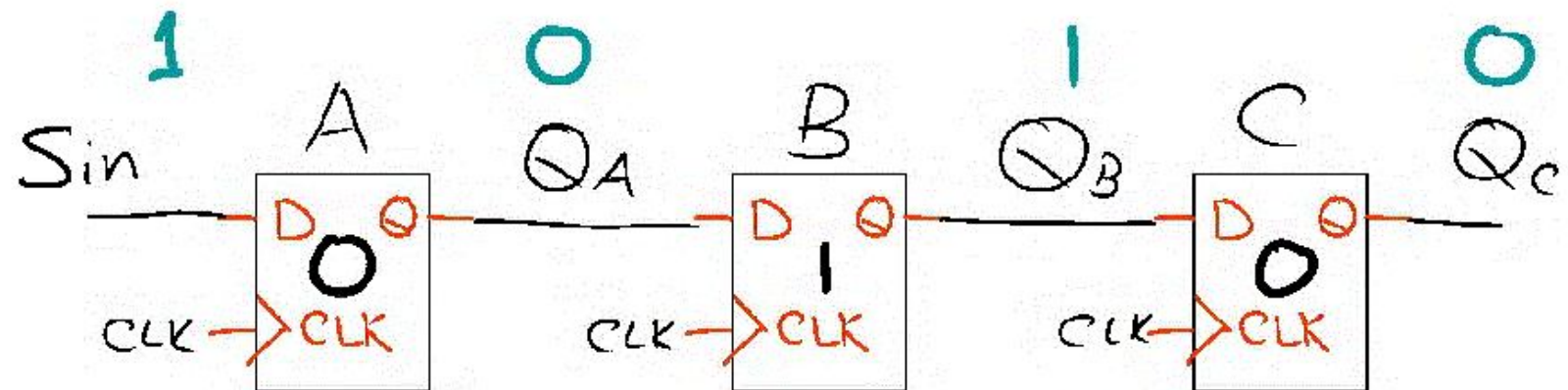
initially

at 20ns

At 20 ns _____ will be evaluated.

At $20 + \Delta T$ _____ will be evaluated.

RIGHT-SHIFT register



Current content =

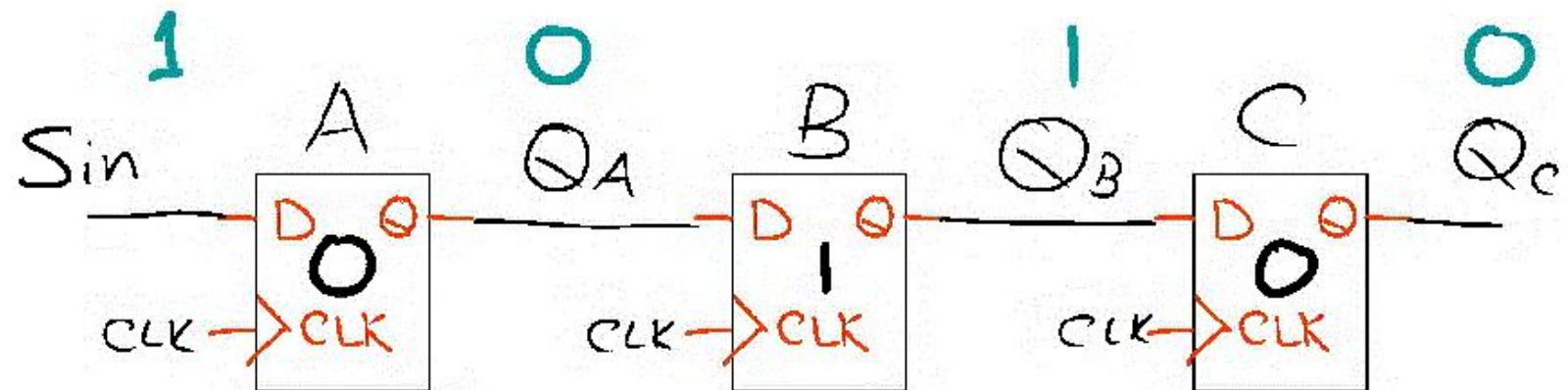
A	B	C
0	1	0

New content =

A	B	C
1	0	1

focus on B

RIGHT-SHIFT register



Looks like a **RACE**
Condition!

RACE

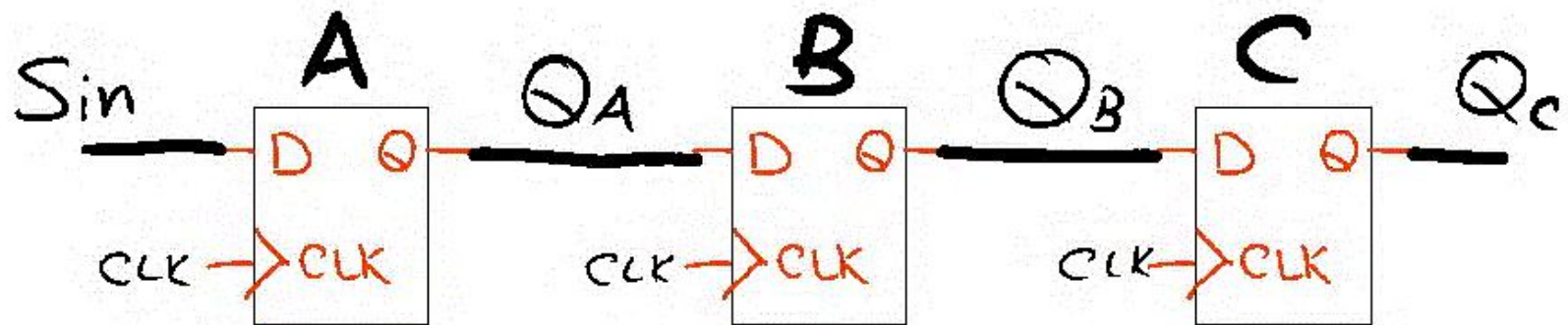
Condition $\equiv$

a condition that produces
non-deterministic values

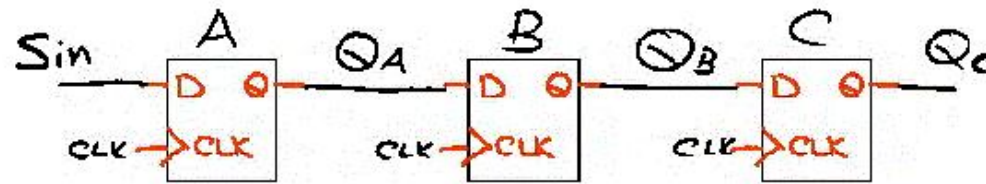
Circuit that produces
non-deterministic values is

no good !

How do we **avoid**
race condition?



Shift one bit at a Time?

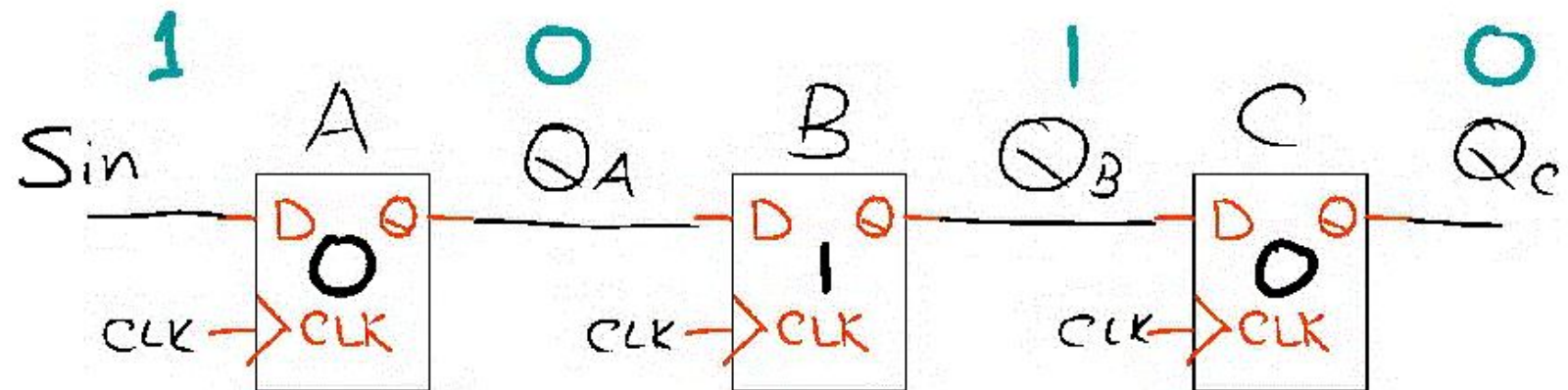


Shift one bit at a Time?

It works, but....

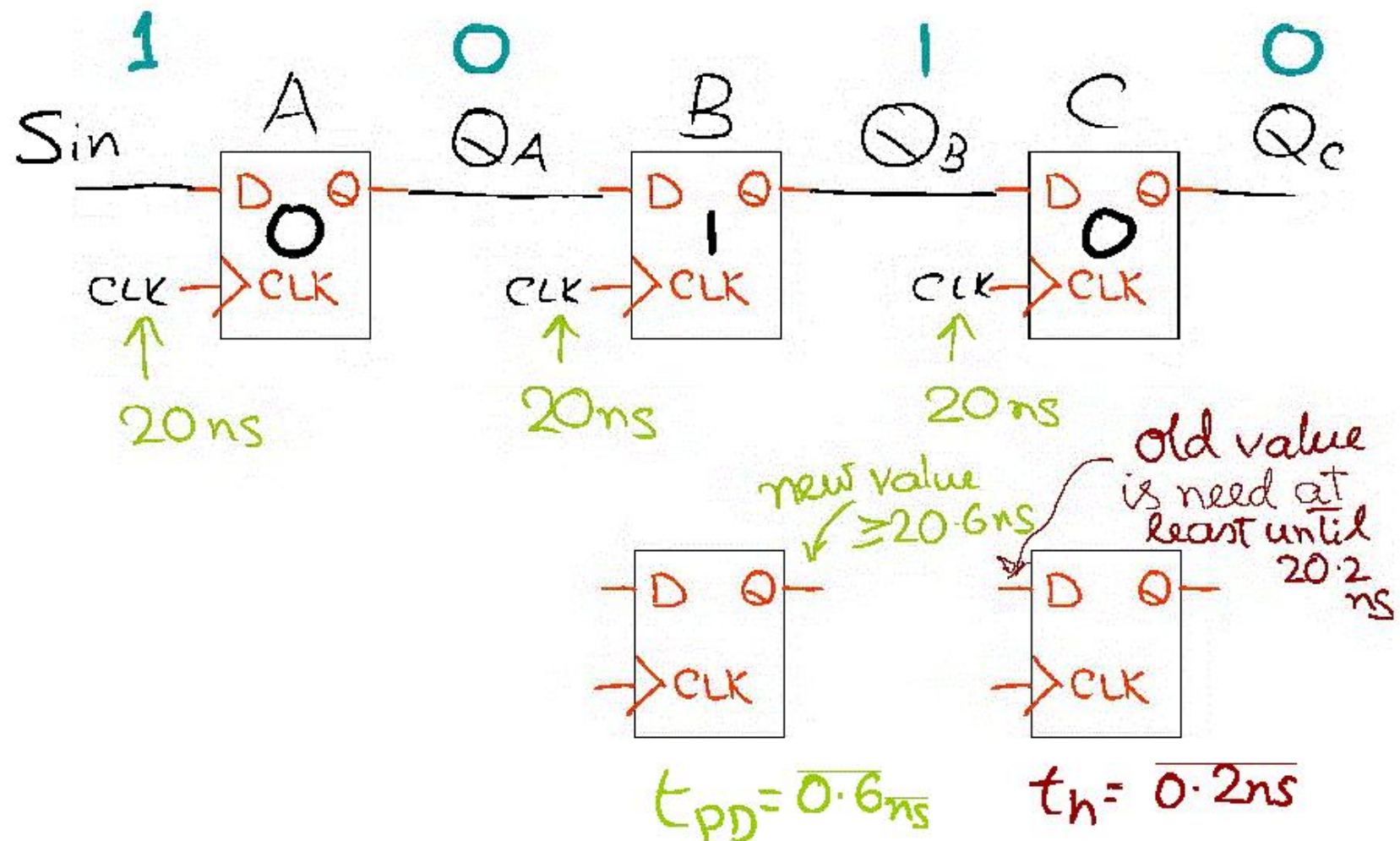
At this rate, it takes 32 clocks
to right-shift by 1 bit
in a 32-bit Shift Register.

RIGHT-SHIFT register

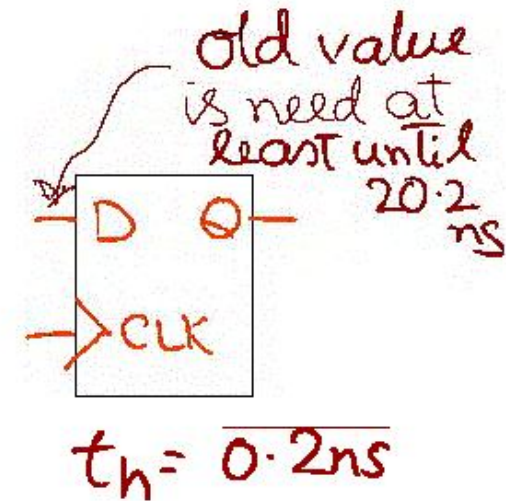
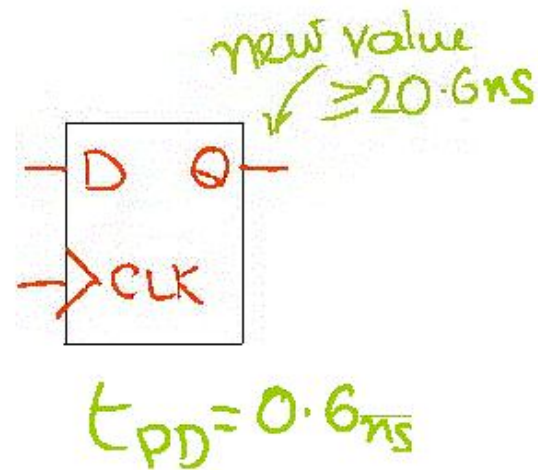


The **RACE** condition is **prevented**
in hardware design by **proper**
Timing design!

RIGHT-SHIFT register



TIMING Design avoids race



$t_{PD} \geq t_h$

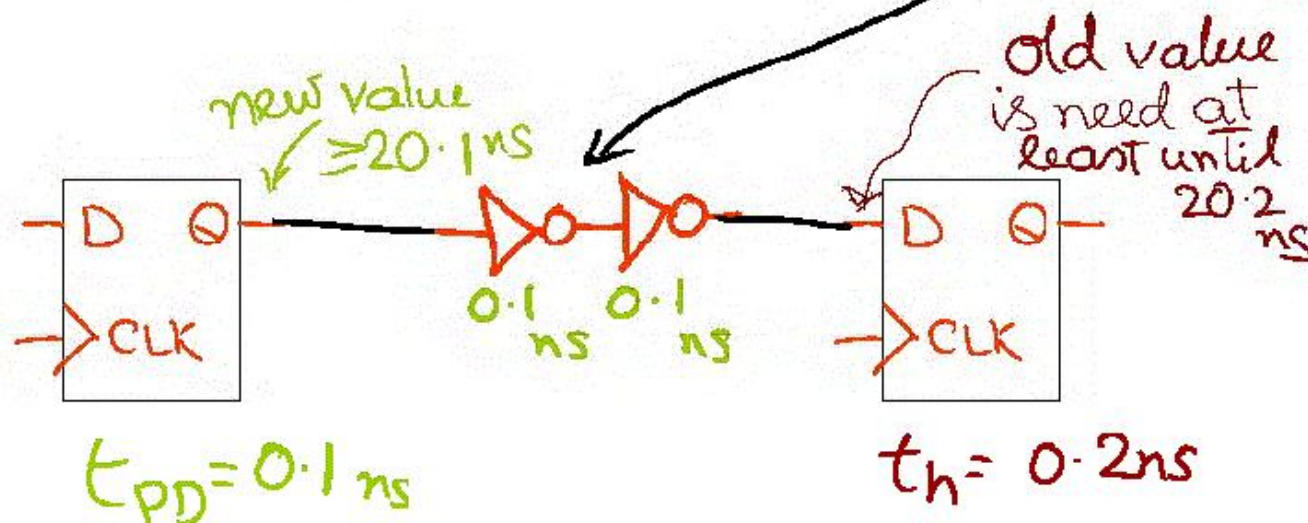
Golden rule!

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 Analyzer (STA)
Static Timing Analysis (STA)
Static Timing Designer

Copyright 2008 Gandhi Puvvada

all same

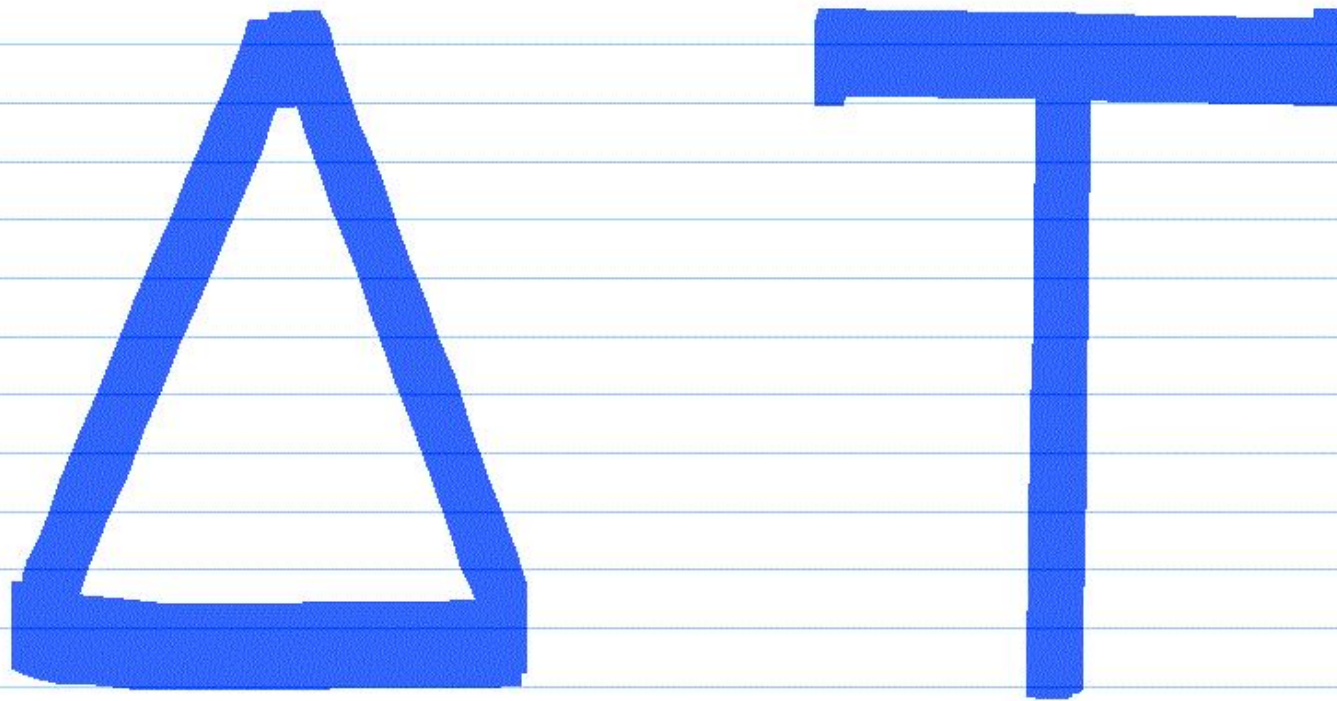
Static Timing Designer

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!

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

$$20 + \Delta T > 20.0 \leftarrow$$



The **delta-T** concept in HDL (Verilog or VHDL) is very fundamental and important.

non-blocking assignments

L.H.S



```

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

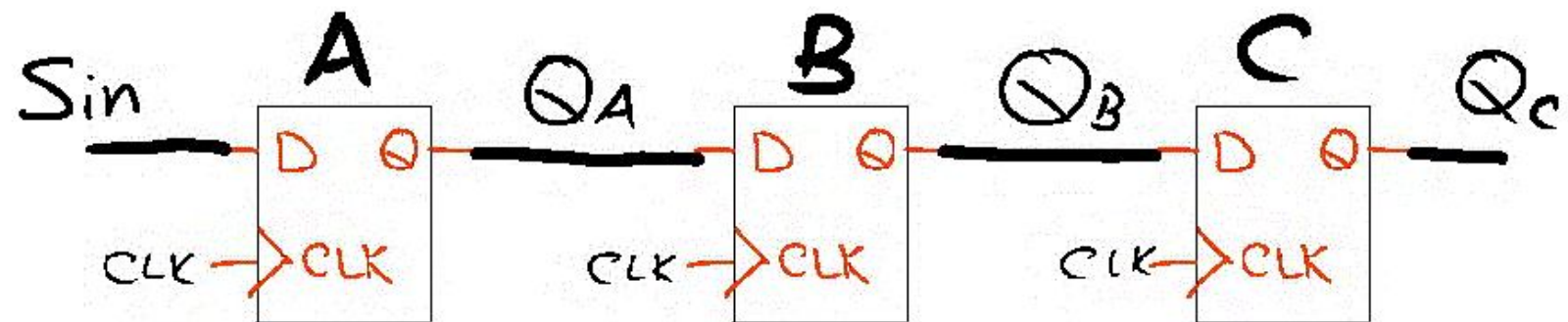
```

non-blocking assignment

L.H.S is updated only after ΔT !

important 

RIGHT-SHIFT register



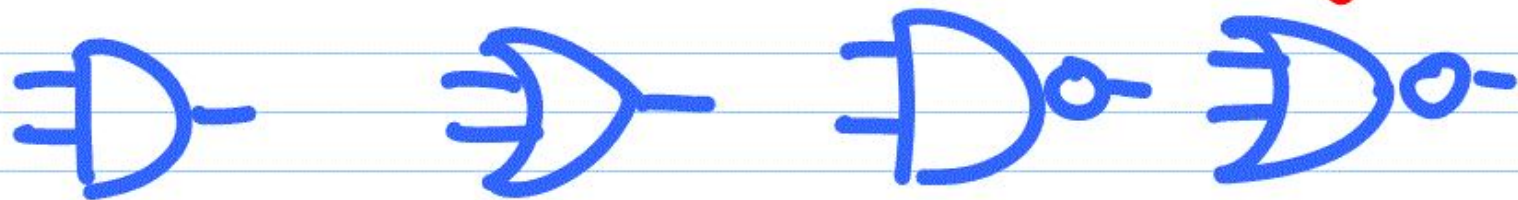
*Scheduled to
change after ΔT*

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

*Note: The order of
these statements does
not matter.*

*This is the current
(unupdated) Q_A .*

Structural Modeling



Start with small components.

Build MSI functions:

HA, FA, 4-bit adder, FF, Register

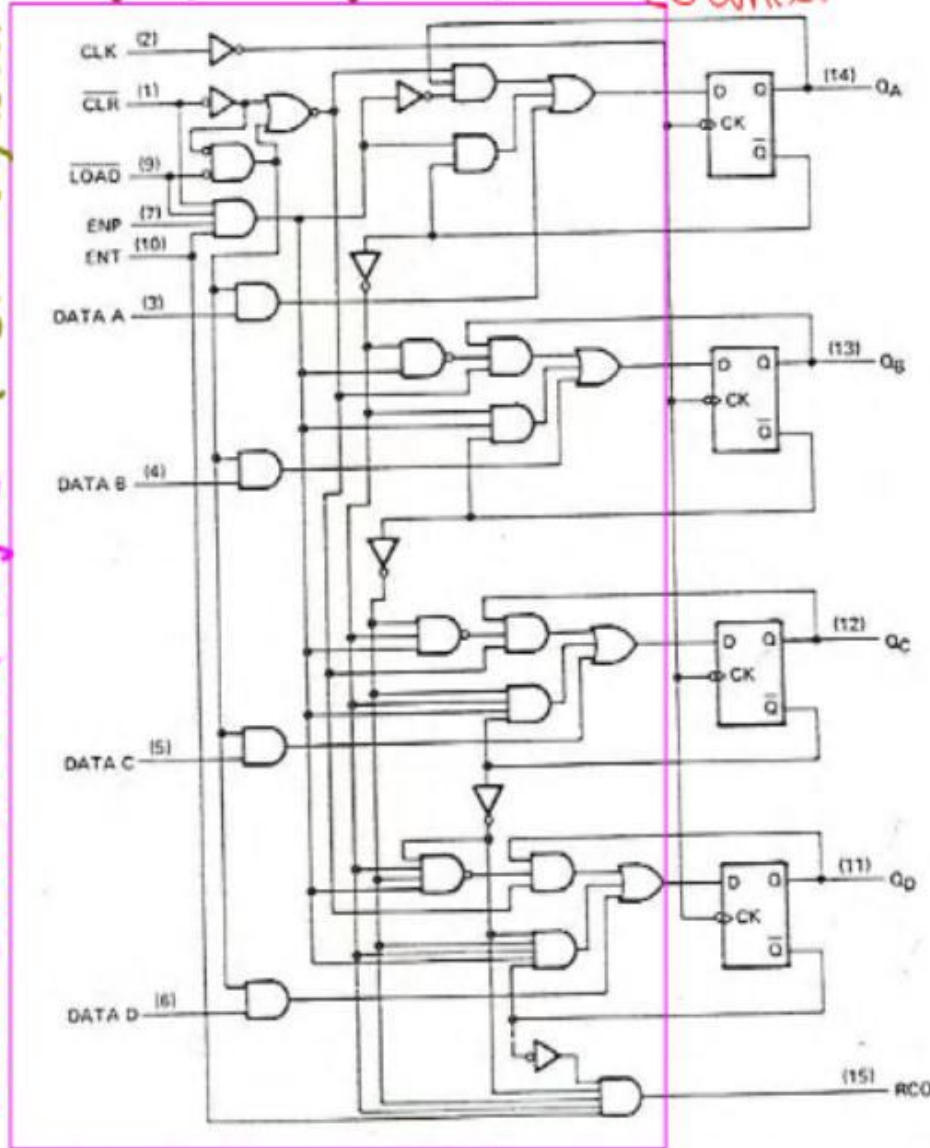
Build subsystems: ALU, Register file

Build the system: CPU

upstream of the
4-bit Register

Combinational logic

74LS163A 4-bit counter



Entirely
Structural
Coding
soon becomes
totally
unreadable
and
unmaintainable!

So we limit the usage of

- ① instantiations
- ② assign (continuous) statements

and start using

- ③ always blocks

to describe

our design behaviorally.

Concurrent constructs in Verilog

We do not want to give up on

the advantages of

Sequential language

in behaviorally describing a

function of a fairly complex piece

of logic, just because we are using **HDL**

which is supposed to be **concurrent!**

We want to use our favorite
if statement,

Case statement,

loop construct, etc.

of the sequential language in our HDL coding.

Sure, you can use sequential language constructs
in **always procedural blocks**
in Verilog!

Concurrent constructs in Verilog

- ① instantiations
- ② assign (continuous statements)
- ③ always blocks

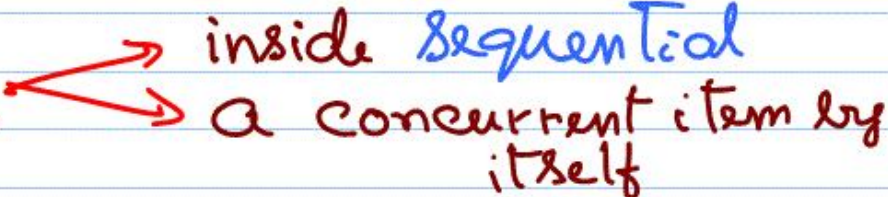
Say, in a HDL design, there are

14 instantiations

8 assign statements

6 always blocks

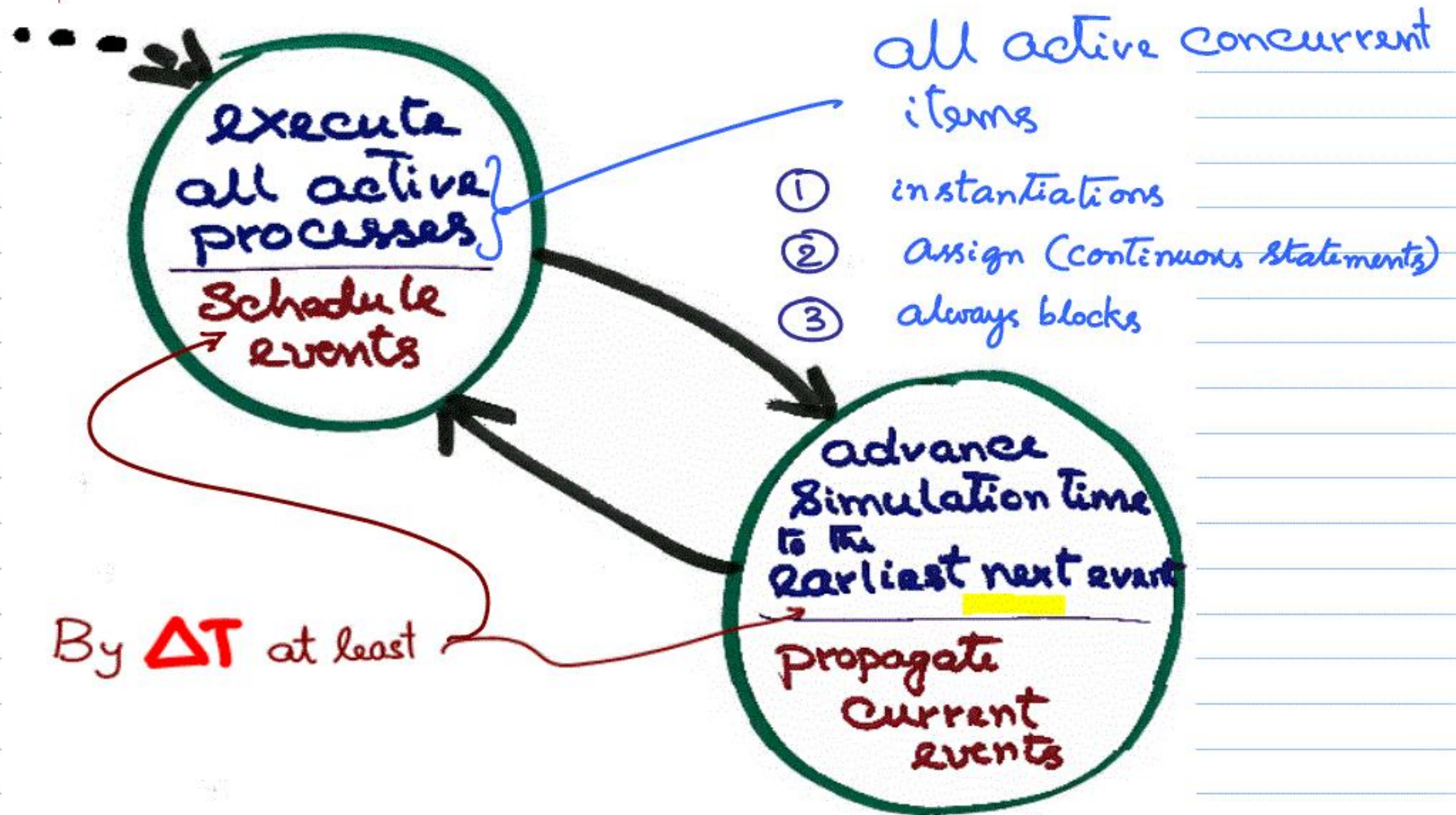
28 total concurrent items

always block  inside sequential
a concurrent item by itself

sequential with-in each



Concurrent



Sensitivity list: If **inputs** change, we want to **re-evaluate** its outputs



Assign $C = A \& B;$

implicit sensitivity list items

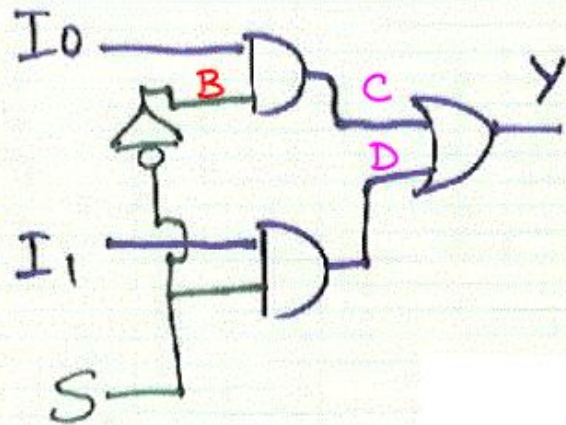
always @ (A, B)

begin

$C = A \& B;$

end

explicit sensitivity list items



Data flow modeling

```

assign B = ~S;
assign C = I0 & B;
assign D = I1 & S;
assign Y = C | D;

```

behavioral modeling

```

always @ (I0, I1, S)

```

```

begin

```

```

    if (S)

```

```

        Y = I1;

```

```

    else

```

```

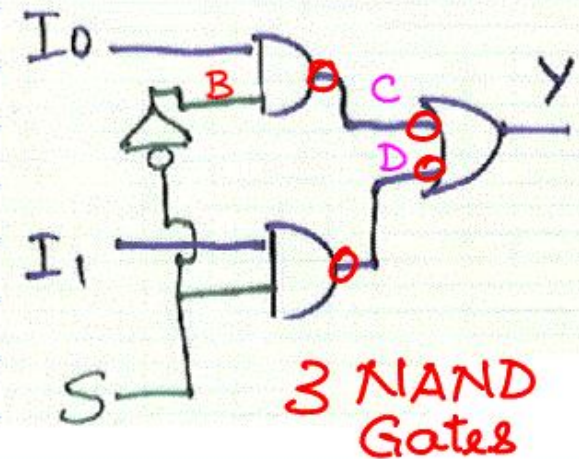
        Y = I0;

```

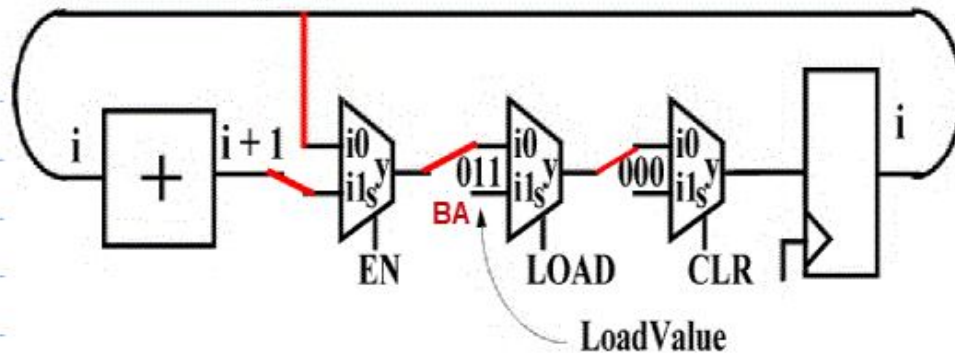
```

end

```



← preferred

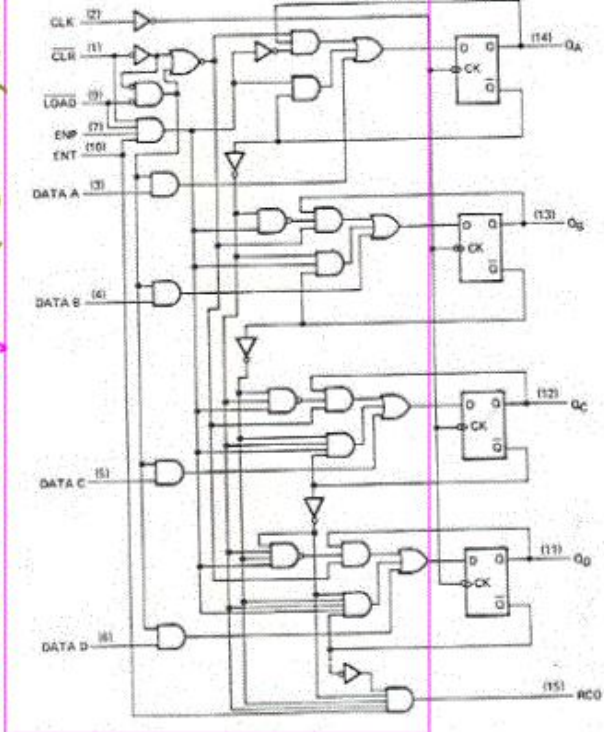


```
always @ (posedge clk)
```

```
begin : COUNT_BLOCK
    if (clr)
        Q <= 3'b000;
    else if (load)
        Q <= BA;
    else if (en)
        Q <= Q + 1;
end
```

upstream of the
4-bit Register

74LS163A 4-bit counter



We code a counter with clear, load and enable controls

-- not at gate level

-- not at MSI level

but **behaviorally** using if .. else statement in a procedural "always" block.

Cycle-based Simulation

or

Even-driven simulation

For initial behavioral simulation with zero delays, cycle-based simulation is much faster but such simulation is valid only if the designer followed pure synchronous design methodology using a single-clock.

Event-driven simulation, though a little slower will cover all situations (pure synchronous designs as well as designs deviating from pure synchronous design methodology). Designs involving clock-gating, multiple phases of clock, require event-driven simulation even for the initial behavioral simulation with zero delays. Also we want to perform timing simulation of the synthesized (after place and route) design to gain confidence of the synthesis process. If the STA (Static Timing Analyzer) has done a good job, then glitches in the combinational circuit output should have died down by the time the next clock edge arrives. Event-driven simulation can show glitches and when they died

Main Points in Verilog

0_Verilog_Main_Points_of_the_6_Lectures.pdf

1_Verilog_Introduction_mht.pdf

2_module_DataTypes_in_Verilog.pdf

3_behavioral_vs_structural_Verilog.pdf

4_Sequential_Statements_in_Verilog.pdf

5_blocking_non_blocking.pdf

6_RTL_coding_style.pdf

1. Verilog is Case-sensitive

Keywords are all in lower-case:

```
example:
module
endmodule
if else
case for
```

2. Comments:

// single-line comment starts with

// double slash!

/* multi-line

is like in "C" comment */

```
/*
- - -
- - -
- - -
- - - */
```

3. Compiler directives $\Rightarrow$ example 'define 'timescale

4. System calls: \$display \$fopen

page 95

```

module muxn (a, b, sel, op);
  parameter WIDTH = 2;
  input [WIDTH-1:0] a, b;
  input sel;
  output [WIDTH-1:0] op;

  assign op = sel ? a : b;

endmodule

```

Op defaults to wire type.

page 94

```

module mux (a, b, sel, op);
  input [7:0] a, b;
  input sel;
  output [7:0] op;
  reg [7:0] op;

  always @(a or b or sel)
    op = sel ? a : b;

endmodule

```

MSB

reg [7:0] op;

vector
data item
[MSB:LSB]

```

always @(a or b or sel)
  op = sel ? a : b;

```

IMPORTANT NOTE

Object declared as a reg data type need not map to a physical register!

An array of NETs or REGISTERS

Useful in modeling **memories**.

`reg [7:0] mem [0:1023];`

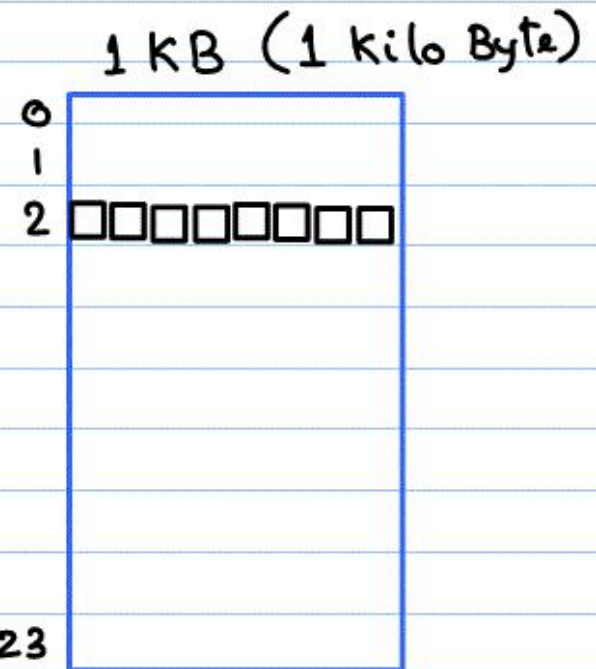
Width of
a location

Number of
locations

`reg [15:0] a, b, c;`

Three 16-bit registers named
a, b, c

need not be physical registers!



Memory Array declaration in Verilog

Main Points in Verilog

0_Verilog_Main_Points_of_the_6_Lectures.pdf

1_Verilog_Introduction_mht.pdf

2_module_DataTypes_in_Verilog.pdf

3_behavioral_vs_structural_Verilog.pdf

4_Sequential_Statements_in_Verilog.pdf

5_blocking_non_blocking.pdf

6_RTL_coding_style.pdf

sequential with-in each



Concurrent

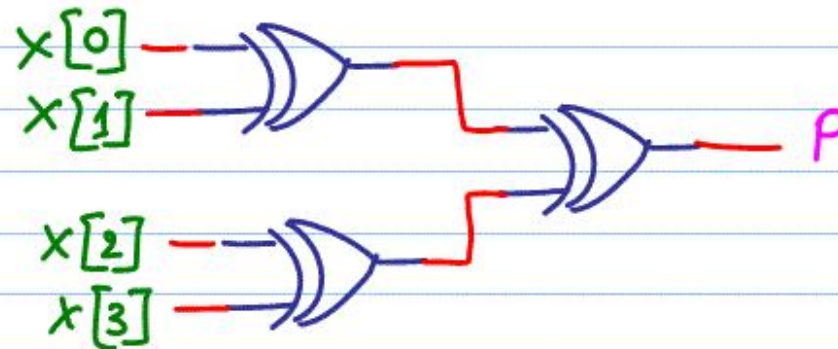
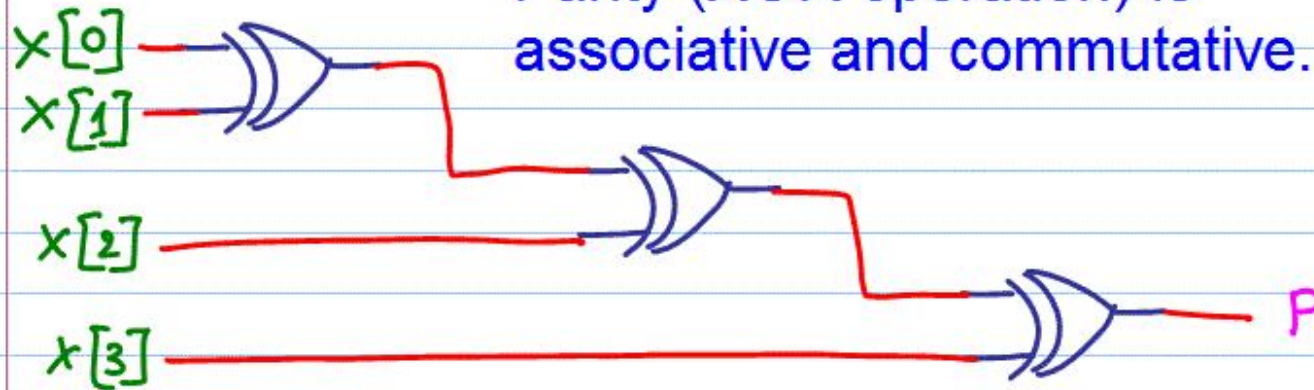
Use sequential constructs (case, if, for, etc.) inside an always block to describe a lot in one block.

Use as few concurrent items as possible.

parity generator

Given 4-bit x
produce odd
parity

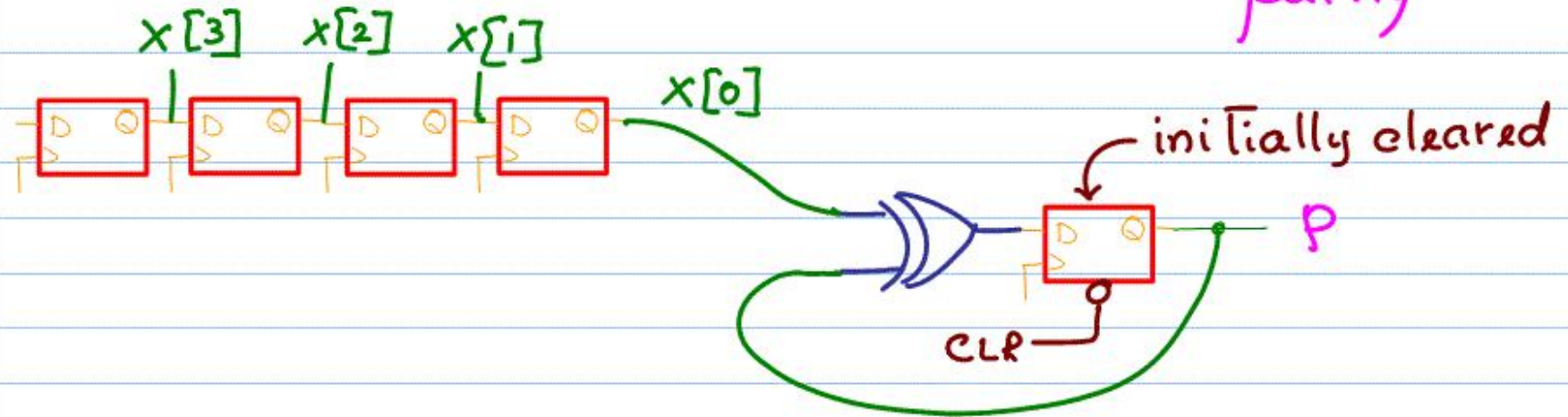
Parity (XOR operation) is
associative and commutative.



For 4-bit you
can design
at gate-level.
But if it is 64-bit
you want a better
method

parity generator

Given 4-bit x
produce odd
parity



SERIAL PARITY GENERATOR

parity generator

always @(x)

begin

$P = 1'b0;$

for ($i = 0; i \leq 3; i = i + 1$)

begin

$P = P \wedge x[i];$

end

end

Given 4-bit x
produce odd
parity

integer $i;$

blocking assignments

Time is FROZEN, the "for" loop is unrolled, and all iterations are executed in one stretch.

"for" loop is not meant for hardware iteration!

Main Points in Verilog

0_Verilog_Main_Points_of_the_6_Lectures.pdf

1_Verilog_Introduction_mht.pdf

2_module_DataTypes_in_Verilog.pdf

3_behavioral_vs_structural_Verilog.pdf

4_Sequential_Statements_in_Verilog.pdf

5_blocking_non_blocking.pdf

6_RTL_coding_style.pdf

always *page 44*

if ... else if ... else
page 82

initial *page 84*

case, casex, casez
page 54

begin...end
page 51

for
page 71

*Casex, Casez
not important
(So I removed those slides)*

Please download the Esperan Verilog Guide and refer to it for syntactical details.

page 44, 45

always

```

always @(sel, a, b)
  if (sel)
    op = a;
  else
    op = b;

```

always @ (

) 

no ;

begin 

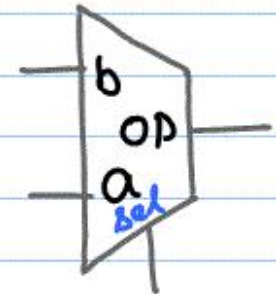
Sequential
Statements


end 

begin

end

page 51



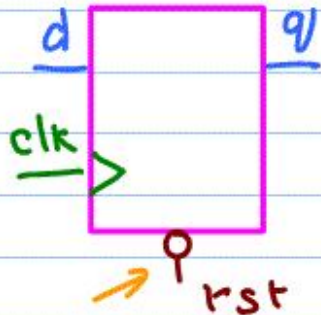
necessary if you have two or more (more than one) sequential statements, but better to type them even for a single statement.

page 113, 97
page 12, 13

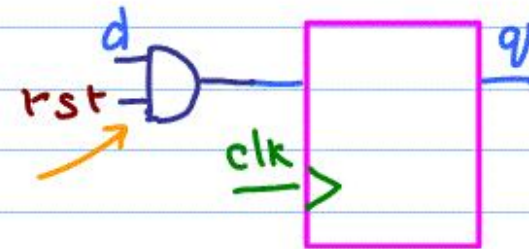
```
always @(posedge clk or negedge rst)
  if (rst == 0)
    q <= 0;
  else
    q <= d;
```

remove

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



↑ asynchronous reset

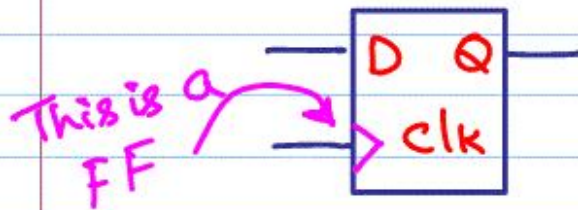


↑ synchronous reset

The following process
infers a transparent latch:

```
always @(ctrl, or a)
  if (ctrl)
    b = a;
```

no "else" $\Rightarrow$ Latch



SOLUTION

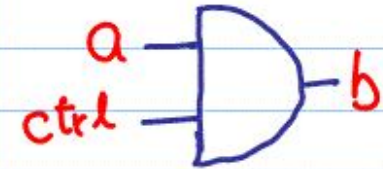
```
always @ (ctrl, a)
```

```
  if (ctrl)
```

```
    b <= a;
```

```
  else
```

```
    b <= 1'b0;
```

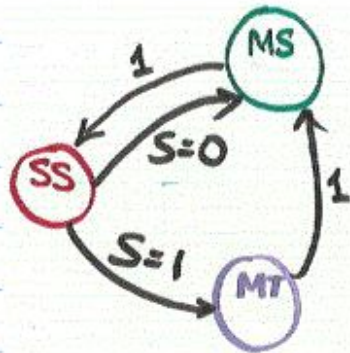


```
always @ (posedge ctrl)
```

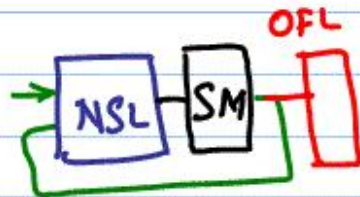
```
  b <= a;
```



We do not use LATCHES in most od RTL designs!



OFL



Default assignments avoid latches
always @ (state)

begin

```

msg = 1'b0;
mtg = 1'b0;
ssq = 1'b0;
if (state == 'MS)
    msg = 1'b1;
if (state == 'MT)
    mtg = 1'b1;
if (state == 'SS)
    ssq = 1'b1;
end
  
```

Write default assignments at the beginning and later selectively override them. <== Very useful technique

In a procedural block

(meaning in the sequential coding area between "begin .. end" in constructs such as always, initial, task and function),

later assignments override earlier assignments.

Please remember this basic rule and the intent behind it.

A good exercise on the application of this property is Q#2 of

http://www-classes.usc.edu/engr/ee-s/457/ee457_Fall2010

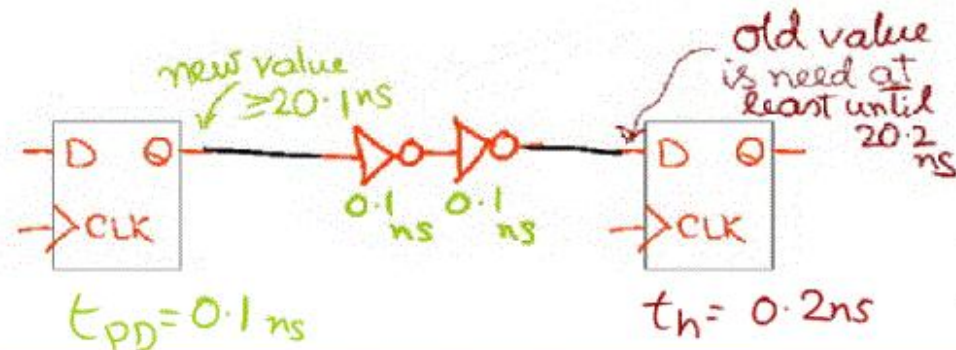
[exams/ee457_Quiz_f12010.pdf](http://www-classes.usc.edu/engr/ee-s/457/ee457_Fall2010/exams/ee457_Quiz_f12010.pdf)

Non-blocking assignments for real registers on the L.H.S in a clocked always block.

$i \leftarrow i + 1;$ // Hardware counter

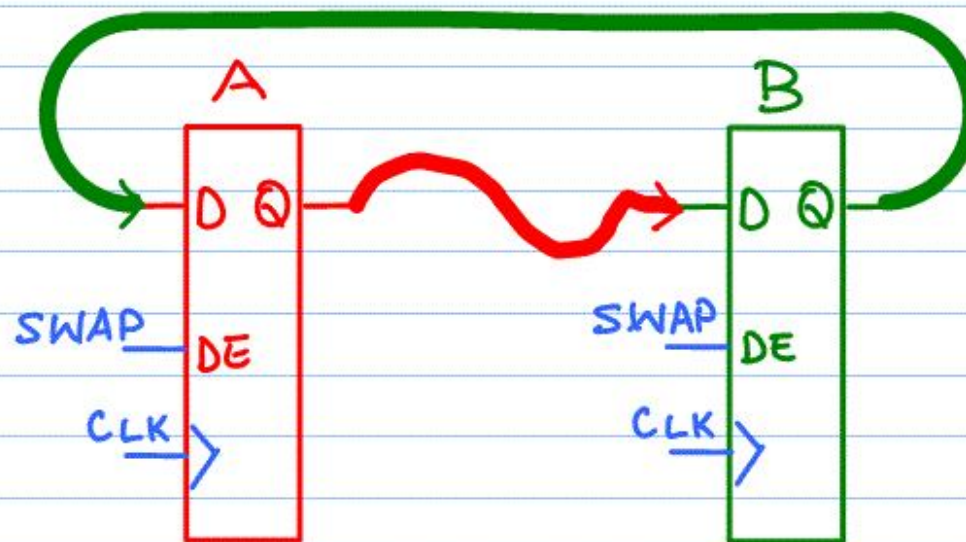
$State \leftarrow f(State, inputs);$ // State machine

$SR \leftarrow \{1'b1, SR[7:1]\}$ // Right Shift register



$$20 + \Delta T > 20.0$$

To swap the contents of two registers, A and B,
do you use Blocking or Non-blocking assignments or
do you have a choice?



**Of course Non-blocking!
No choice at all!**

```
SWAP_STATE:
begin
  if (SWAP)
    begin
      → A <= B;
      → B <= A;
    end
end
```

```
SWAP_STATE:
begin
  if (SWAP)
    begin
      A = B;
      B = A;
    end
end
```

Main Points in Verilog

0_Verilog_Main_Points_of_the_6_Lectures.pdf

1_Verilog_Introduction_mht.pdf

2_module_DataTypes_in_Verilog.pdf

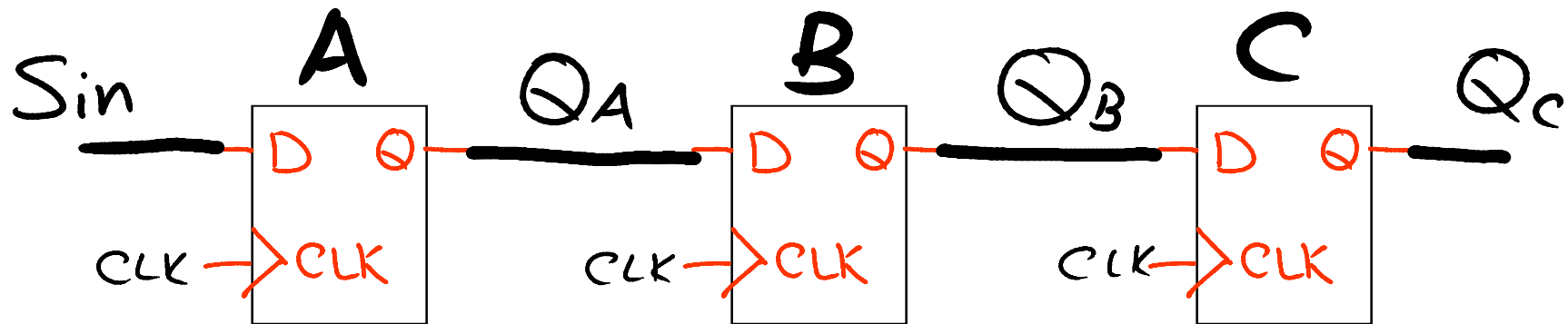
3_behavioral_vs_structural_Verilog.pdf

4_Sequential_Statements_in_Verilog.pdf

5_blocking_non_blocking.pdf

6_RTL_coding_style.pdf

RIGHT-SHIFT register



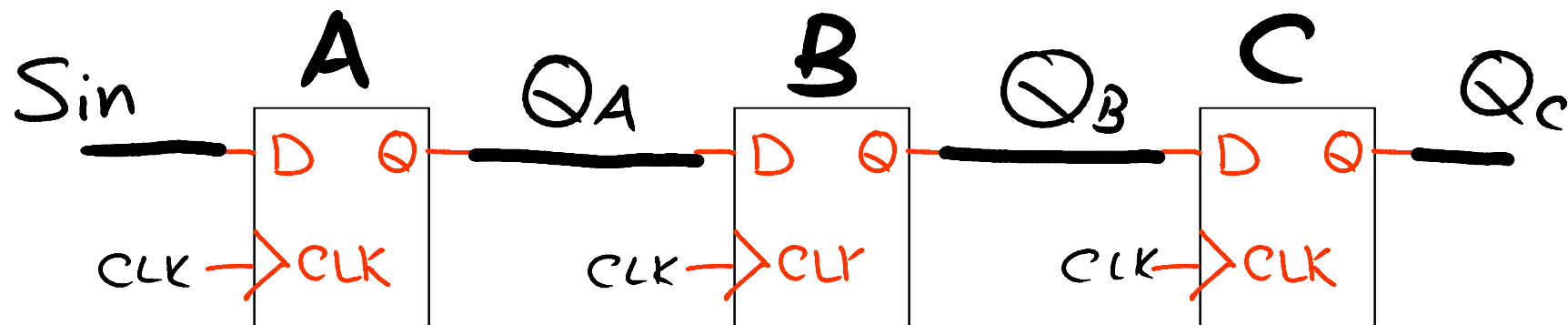
```
QC <= QB;  
QB <= QA;  
QA <= Sin;
```

```
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 ΔT

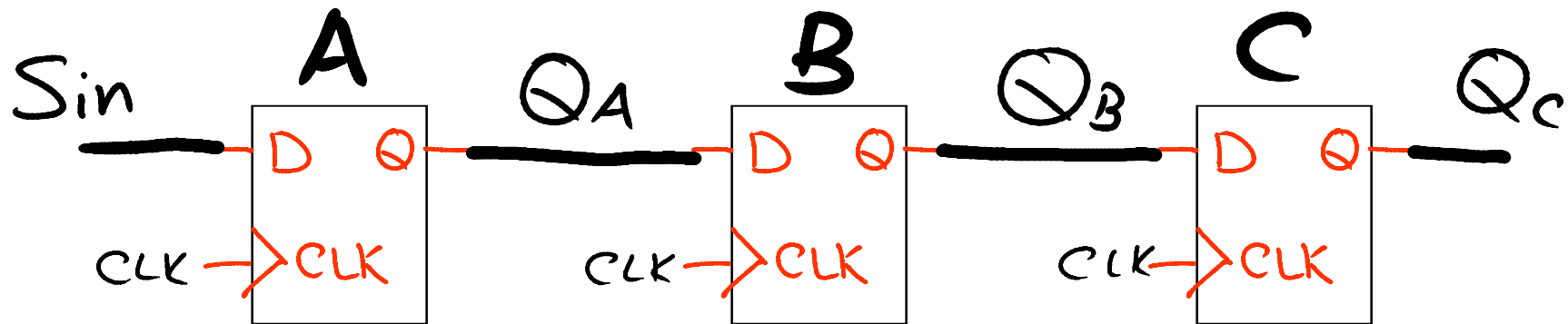
QA <= Sin;
QB <= QA;
QC <= QB;

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

Counter coding

need for a mix 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.

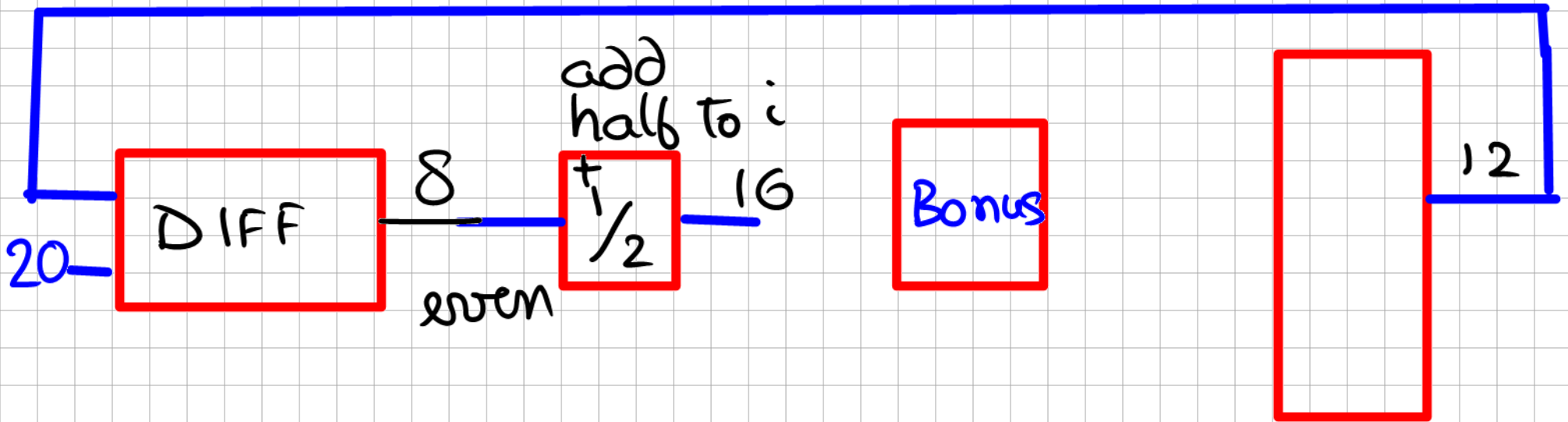
We said, "or a little beyond the mid point M".

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 the current count of 14, the jump takes you only to 17.

Special Counter

→ 0 → 20



```

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
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

```

Main Points in Verilog

0_Verilog_Main_Points_of_the_6_Lectures.pdf

1_Verilog_Introduction_mht.pdf

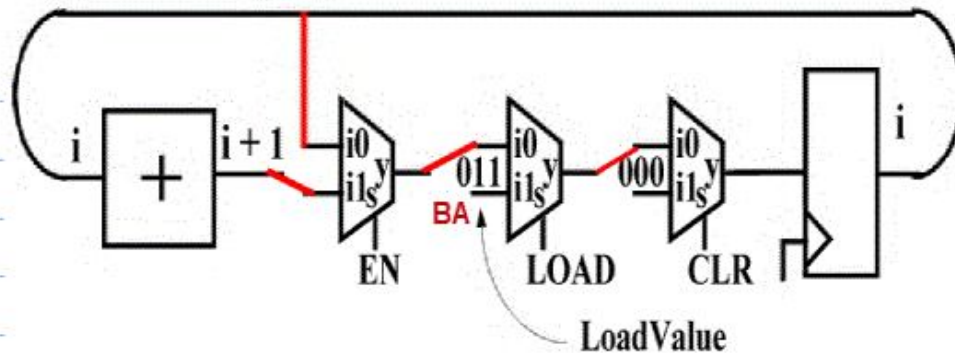
2_module_DataTypes_in_Verilog.pdf

3_behavioral_vs_structural_Verilog.pdf

4_Sequential_Statements_in_Verilog.pdf

5_blocking_non_blocking.pdf

6_RTL_coding_style.pdf

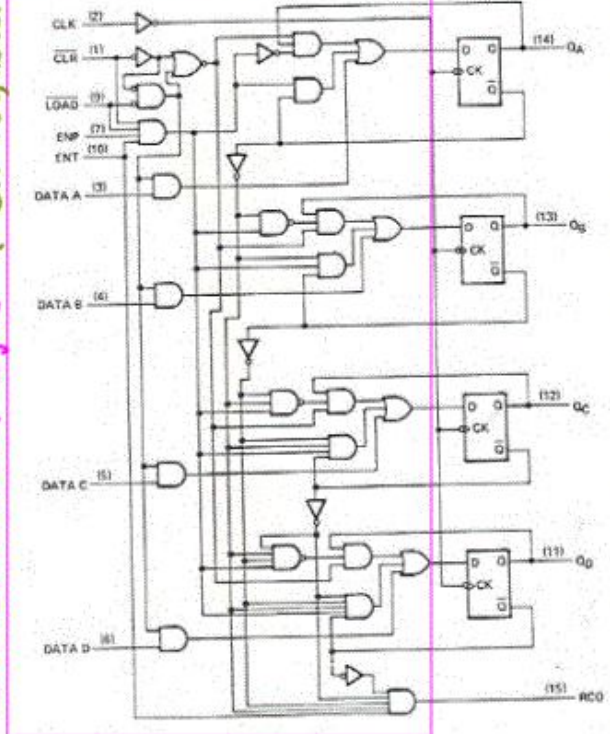


```
always @ (posedge clk)
```

```
begin : COUNT_BLOCK
    if (clr)
        Q <= 3'b000;
    else if (load)
        Q <= BA;
    else if (en)
        Q <= Q + 1;
end
```

upstream of the
4-bit Register
Combinational Logic

74LS163A 4-bit counter



We code a counter with clear, load and enable controls

-- not at gate level

-- not at MSI level

but **behaviorally** using if .. else statement in a procedural "always" block.