

Verilog

Module #1

HDL for simulation and synthesis

Basics of Timing Design

Event-Driven simulation

Zero-delay modeling and behavioral simulation, Delta-T delay

By
Gandhi Puvvada



HDL

HDL = Hardware Description Language

Two HDLs: Verilog and VHDL

-- Simulation

-- Synthesis

Verilog Standards

IEEE 1364 - 1995 ✓

ORIGINALLY CREATED
in GATEWAY DESIGN AUTOMATION
PRABHU GOEL
PHIL MOORBY

2001

2005 ← minor changes

Some very useful improvements

USC library has access to all IEEE standards

Xplore (IEEE electronic library)

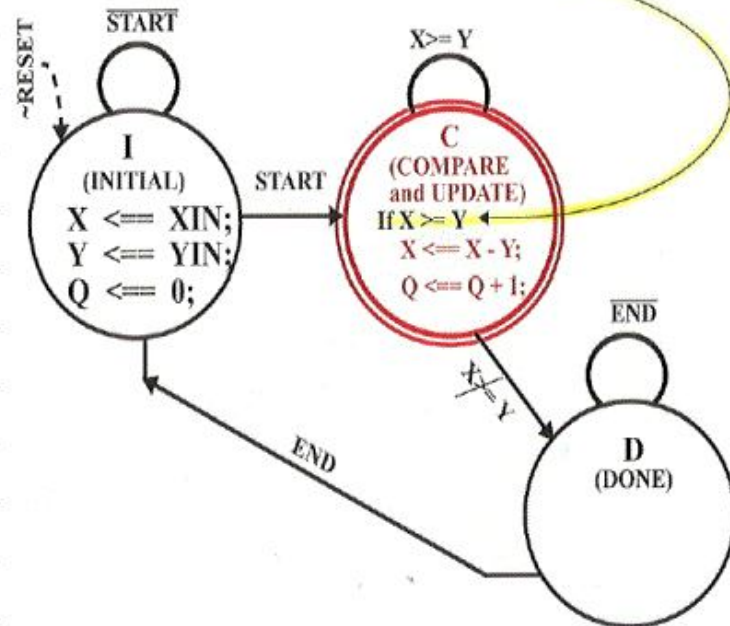
myUSC

IEEE 1800 - 2009
SYSTEM VERILOG

Engineer

Engineer is responsible for design and design improvement.

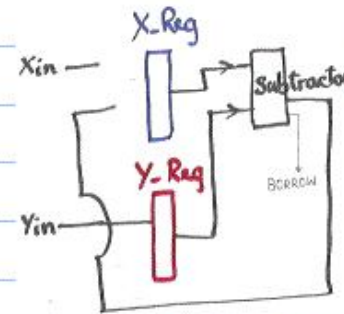
Divider -- Mealy machine design



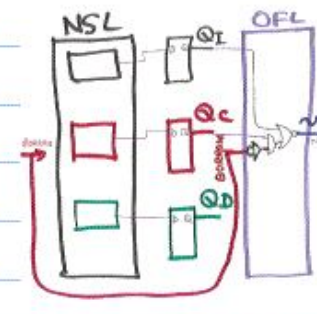
State diagram
with RTL

Technician

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



DPU



CU

Timing design

Setup Time check
Hold Time check

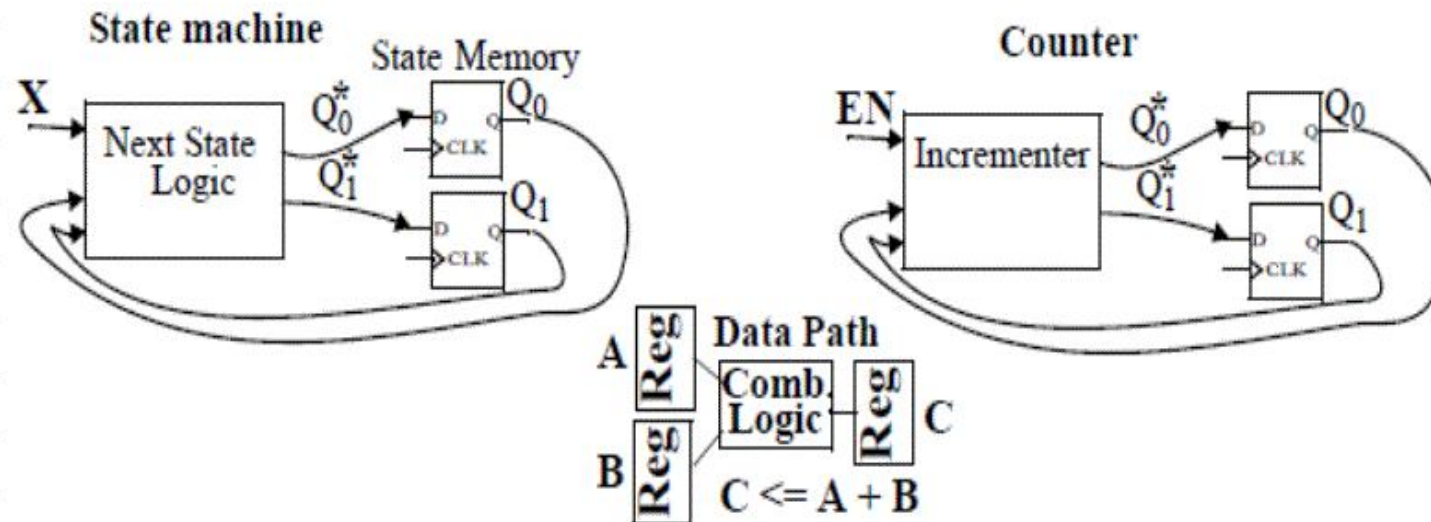
CAD Tools to synthesize

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

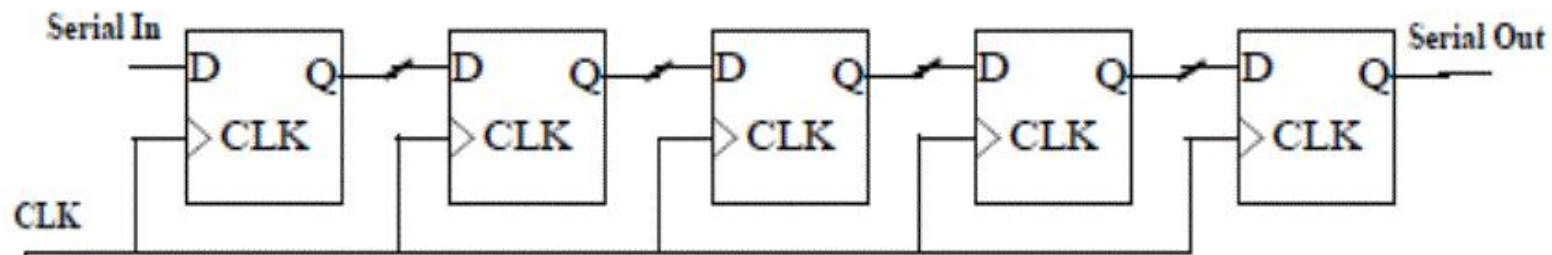
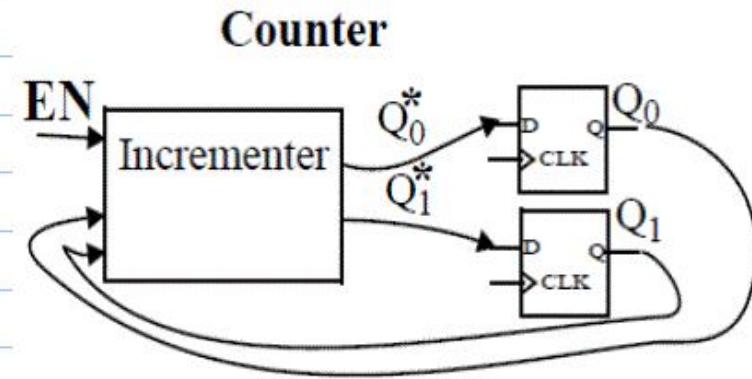
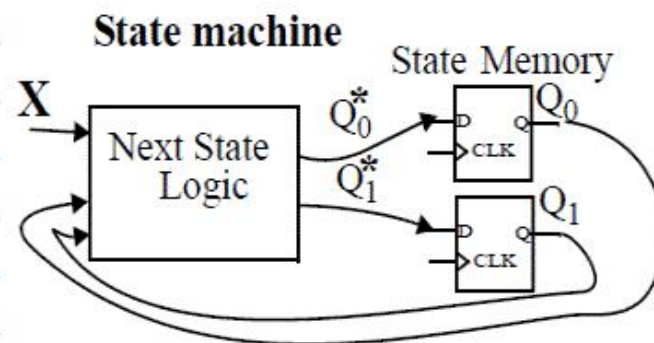
Pure Synchronous Design

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

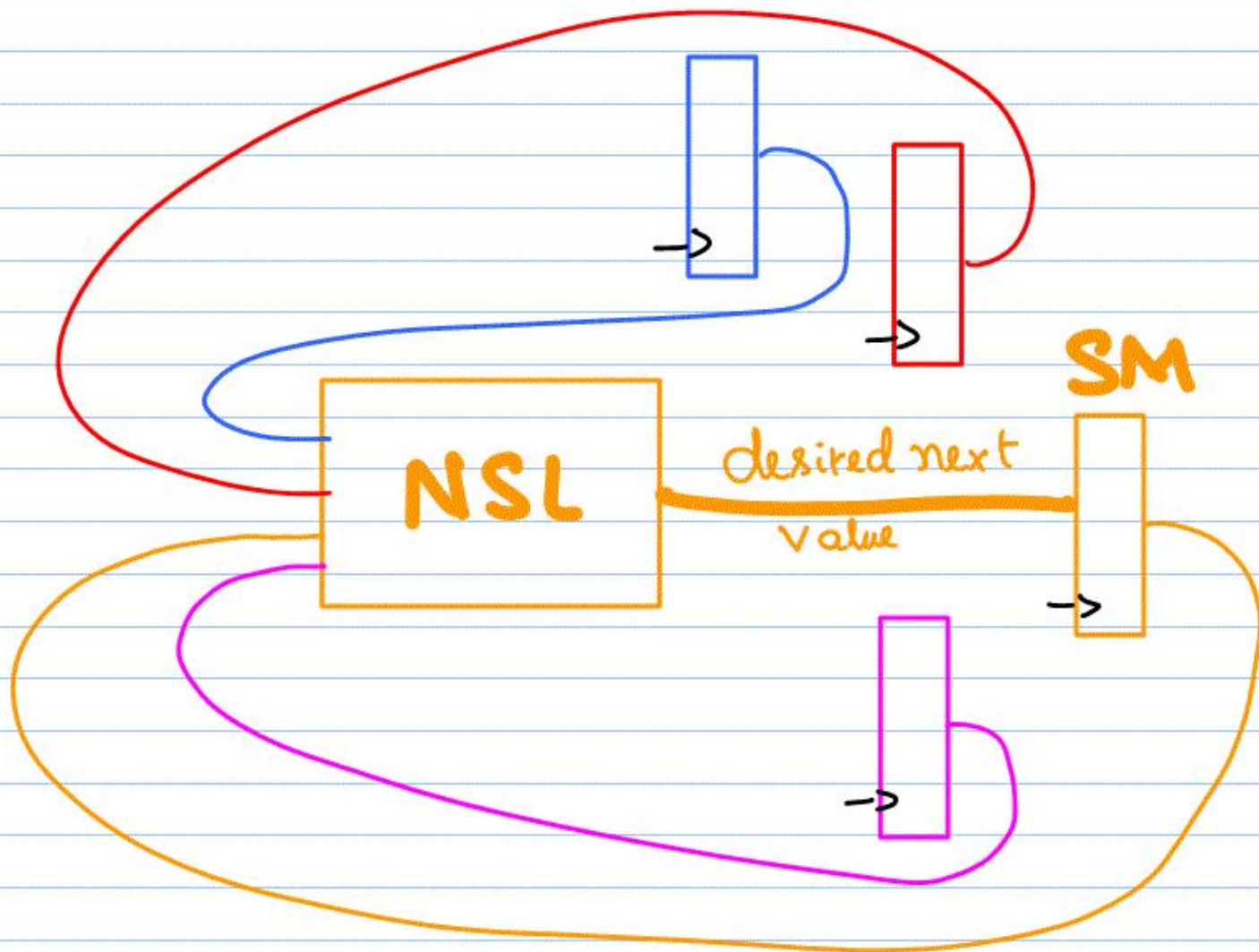
Design can be viewed as a bunch of registers with combinational logic in between them.



Note: Source registers themselves may be about to change!



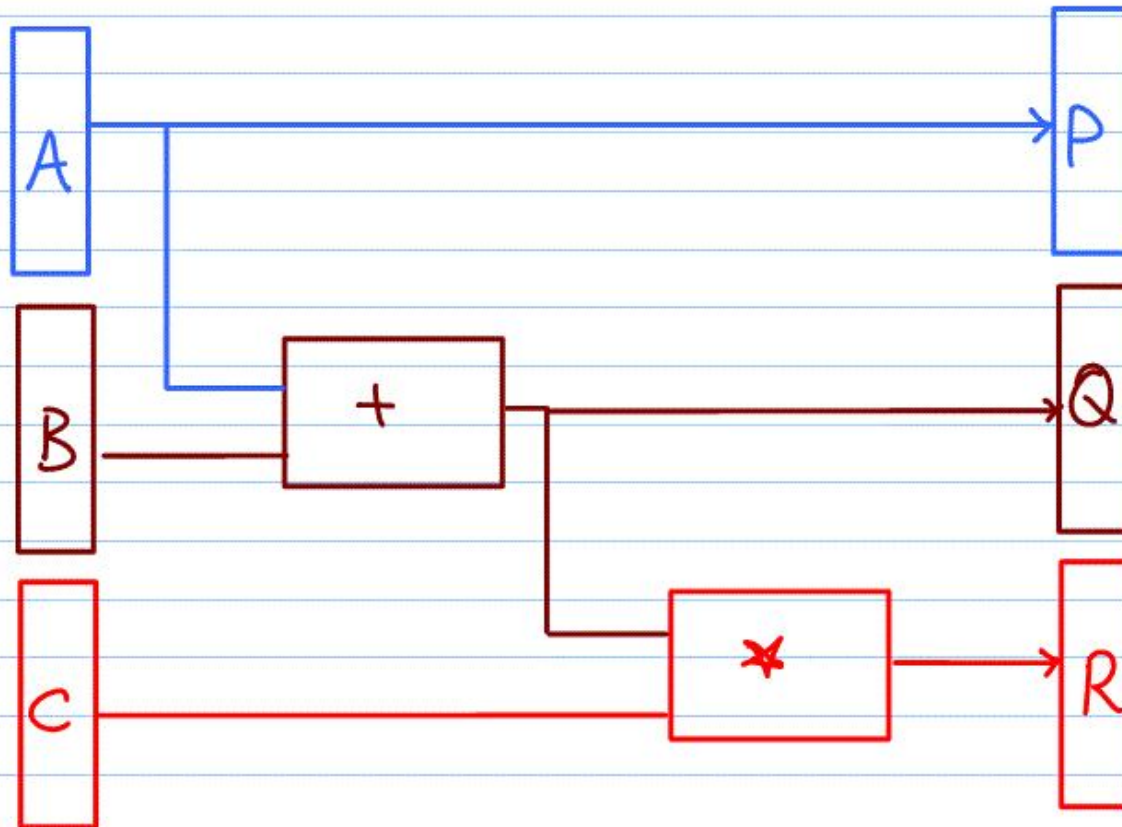
Relation between t_{ffpd} t_h



Timing Design

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

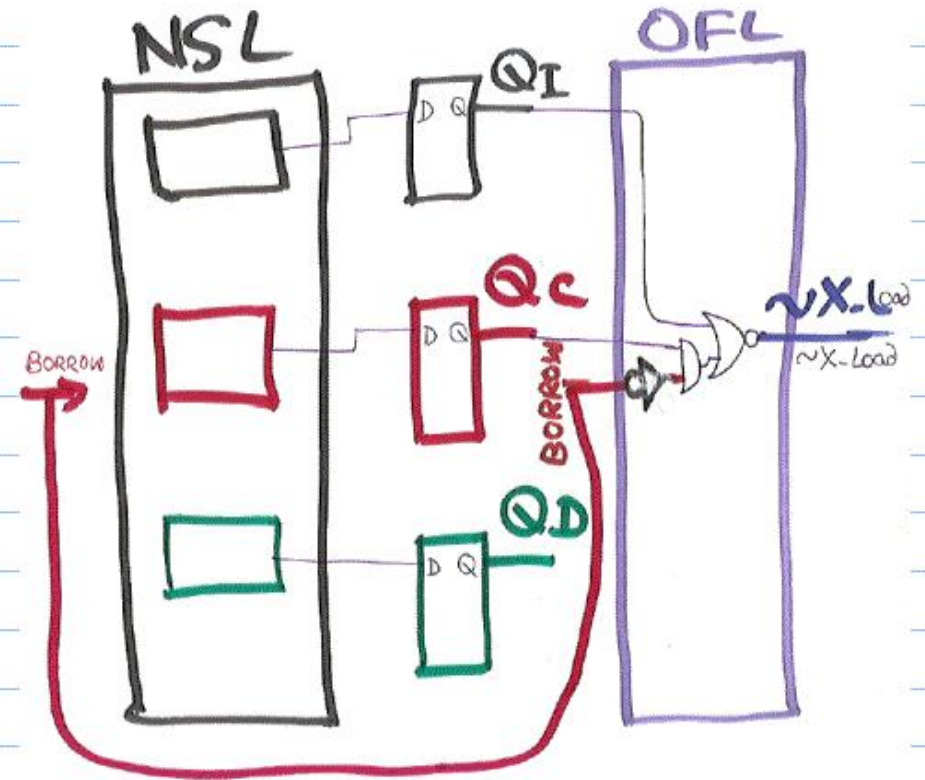
$$\text{Hold time Margin} = t_{\text{ffPD(min)}} + t_{\text{comb(min)}} - t_{\text{hold(min)}}$$



```
graph TD
    I((I  
(INITIAL)  
X <= XIN;  
Y <= YIN;  
Q <= 0;))
    C(((C  
(COMPARE  
and UPDATE)  
If X >= Y  
X <= X - Y;  
Q <= Q + 1;)))
    D((D  
(DONE)))

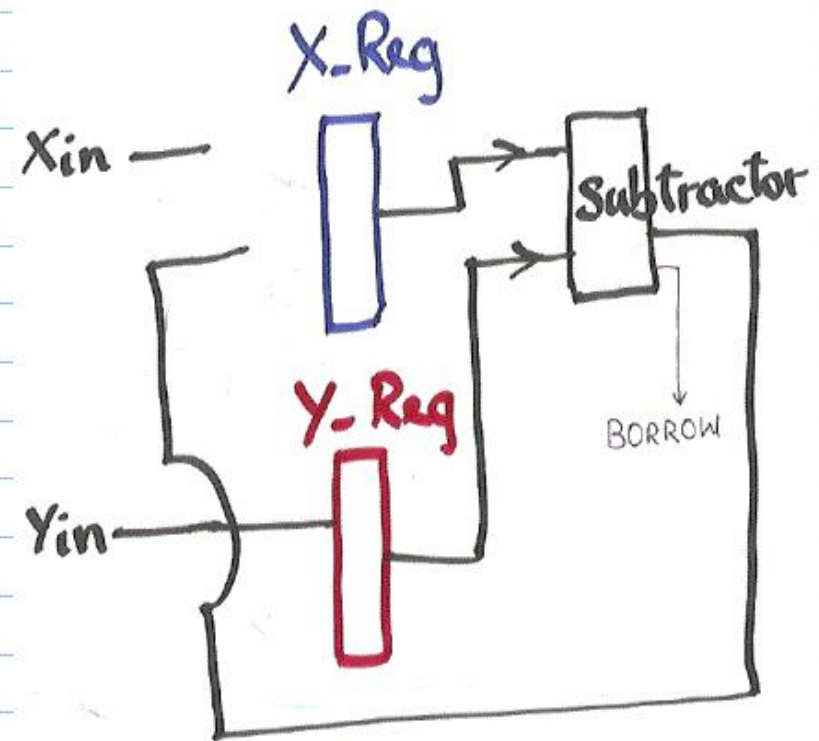
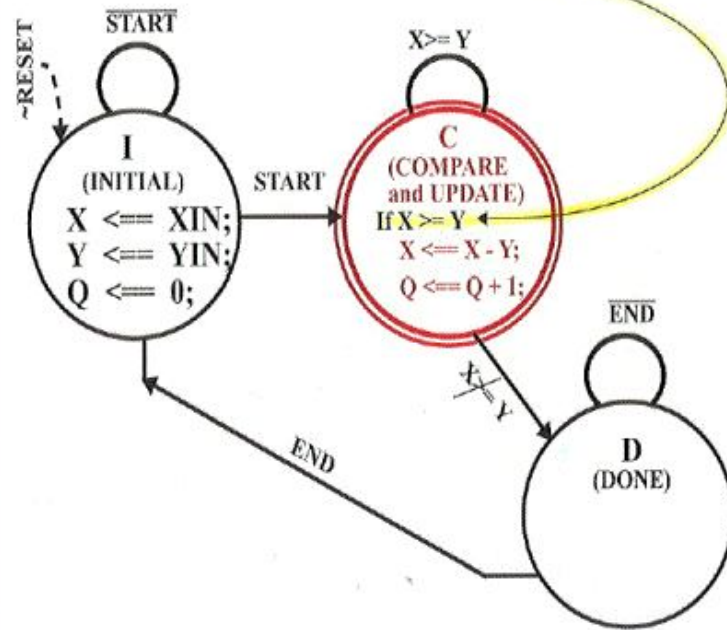
    I -- START --> C
    C -- "X >= Y" --> C
    C -- "X < Y" --> I
    C -- END --> D
```

The flowchart illustrates the subtraction algorithm using the borrow flag. It consists of three states: I (INITIAL), C (COMPARE and UPDATE), and D (DONE). State I initializes the registers X, Y, and Q. State C performs the comparison and update of the borrow flag Q. If X is greater than or equal to Y, the subtraction is performed, and Q is incremented. If X is less than Y, the borrow flag is cleared, and the process returns to state I. The process ends at state D.



Synthesis

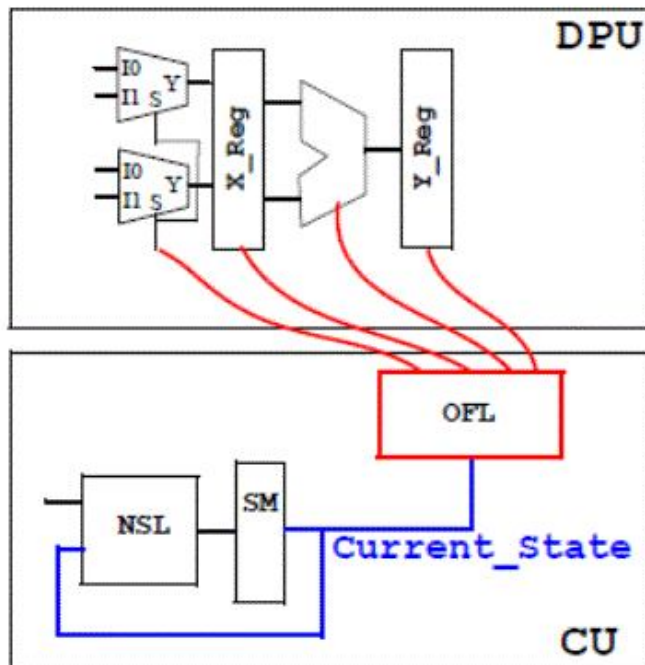
Divider -- Mealy machine design



Synthesis

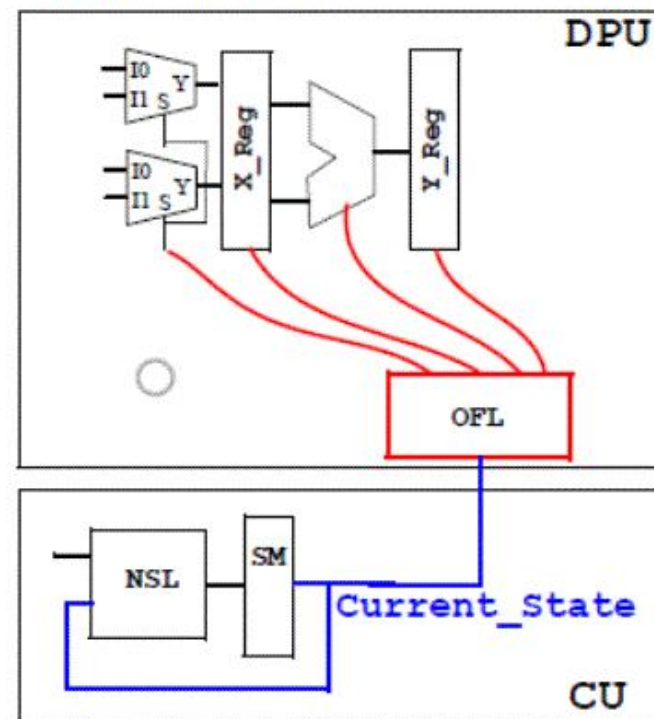
Traditional division between DPU and CU

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.

Ordinary programming languages are sequential in execution.

But Hardware is concurrent.

So HDL (Verilog or VHDL) is concurrent.

==> Event Driven simulation

Hardware

is

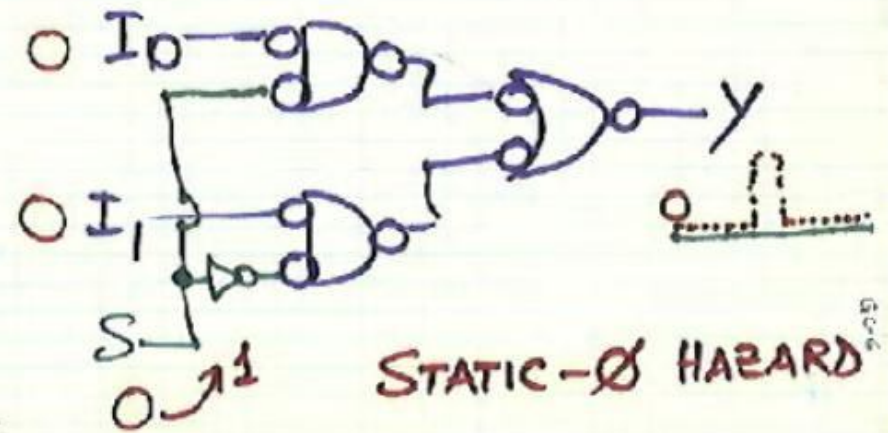
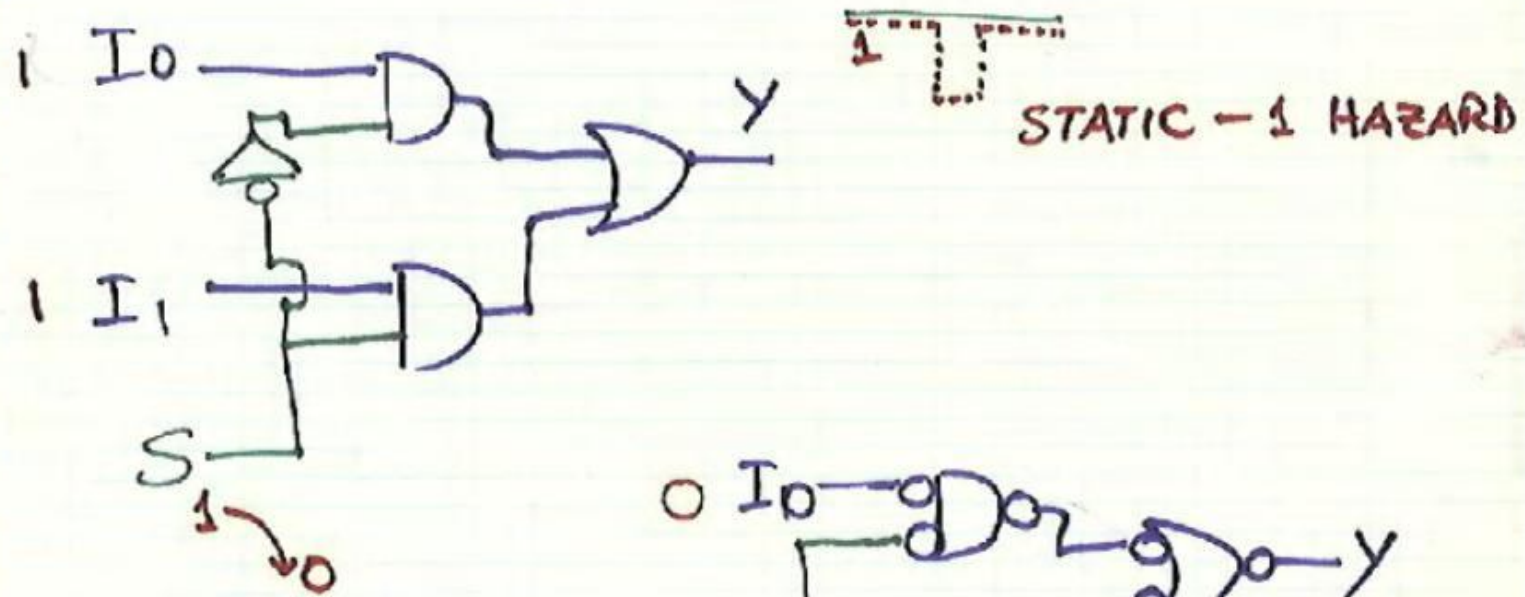
Concurrent.

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

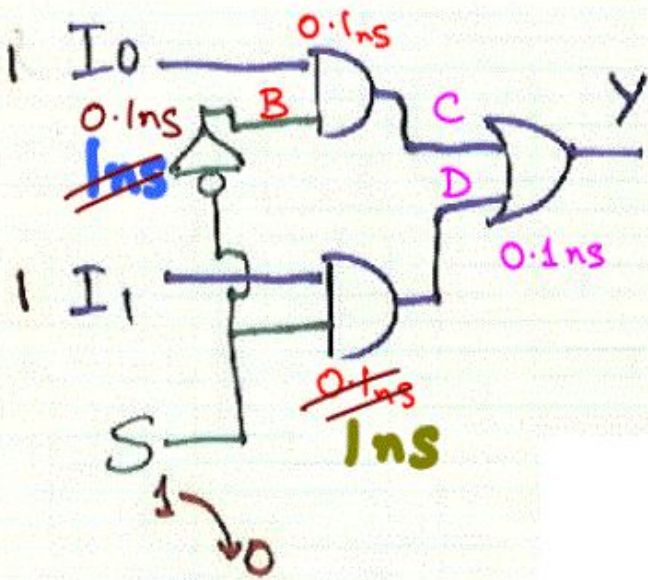
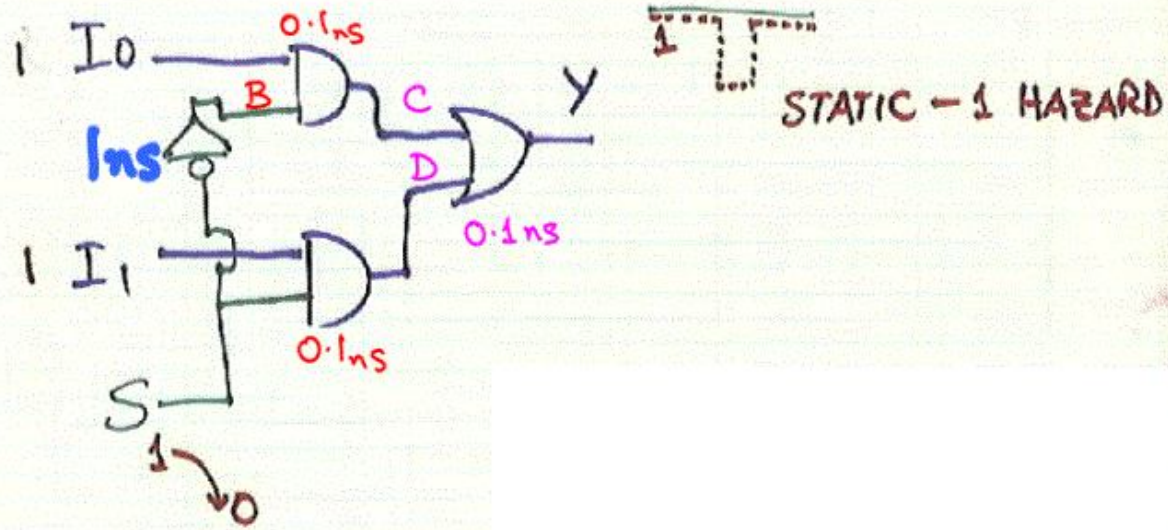
represent and **preserve**

this **Concurrent**
nature of Hardware.

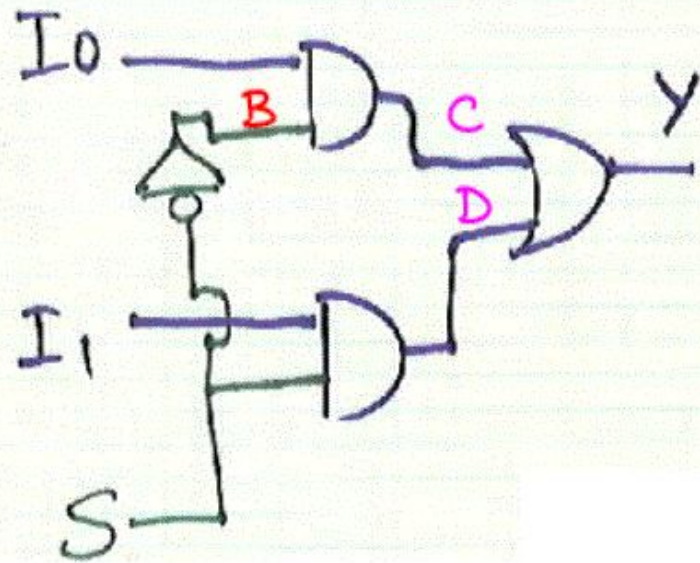
Glitches in Combinational logic



7a.



Do you get a glitch in both cases?



Operators

$\sim$ NOT $\neg$

$\&$ AND $\Rightarrow$

$|$ OR $\vee$

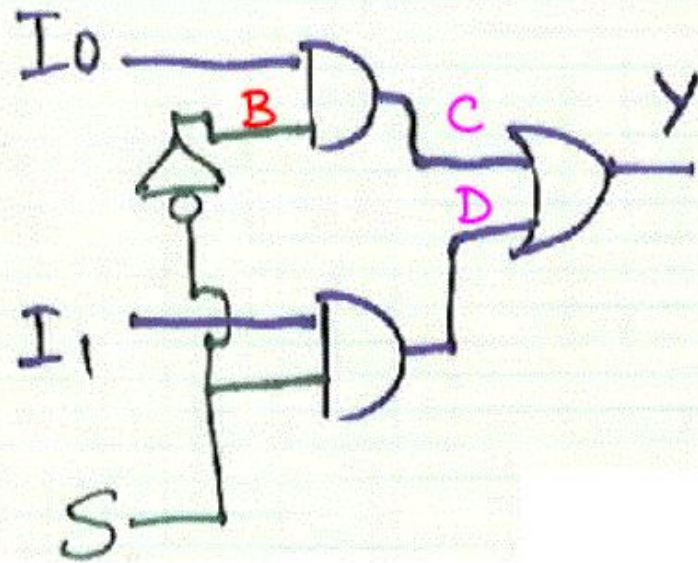
$\wedge$ XOR $\Rightarrow$

$\wedge \sim$ XNOR $\Rightarrow$

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

$\frac{A}{B}$ NAND

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



Operators

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

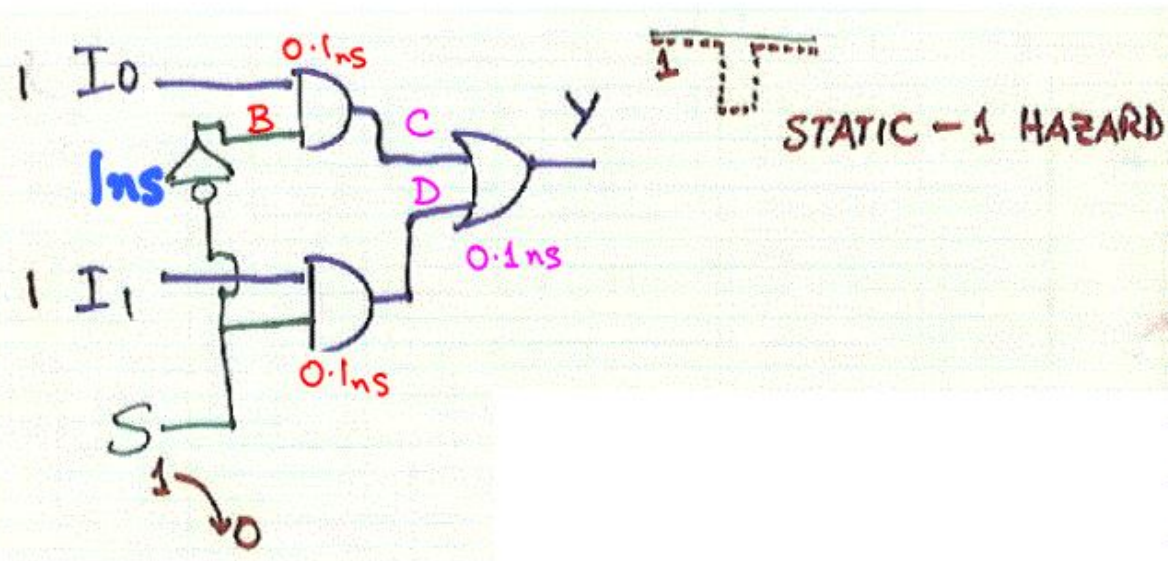
```

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

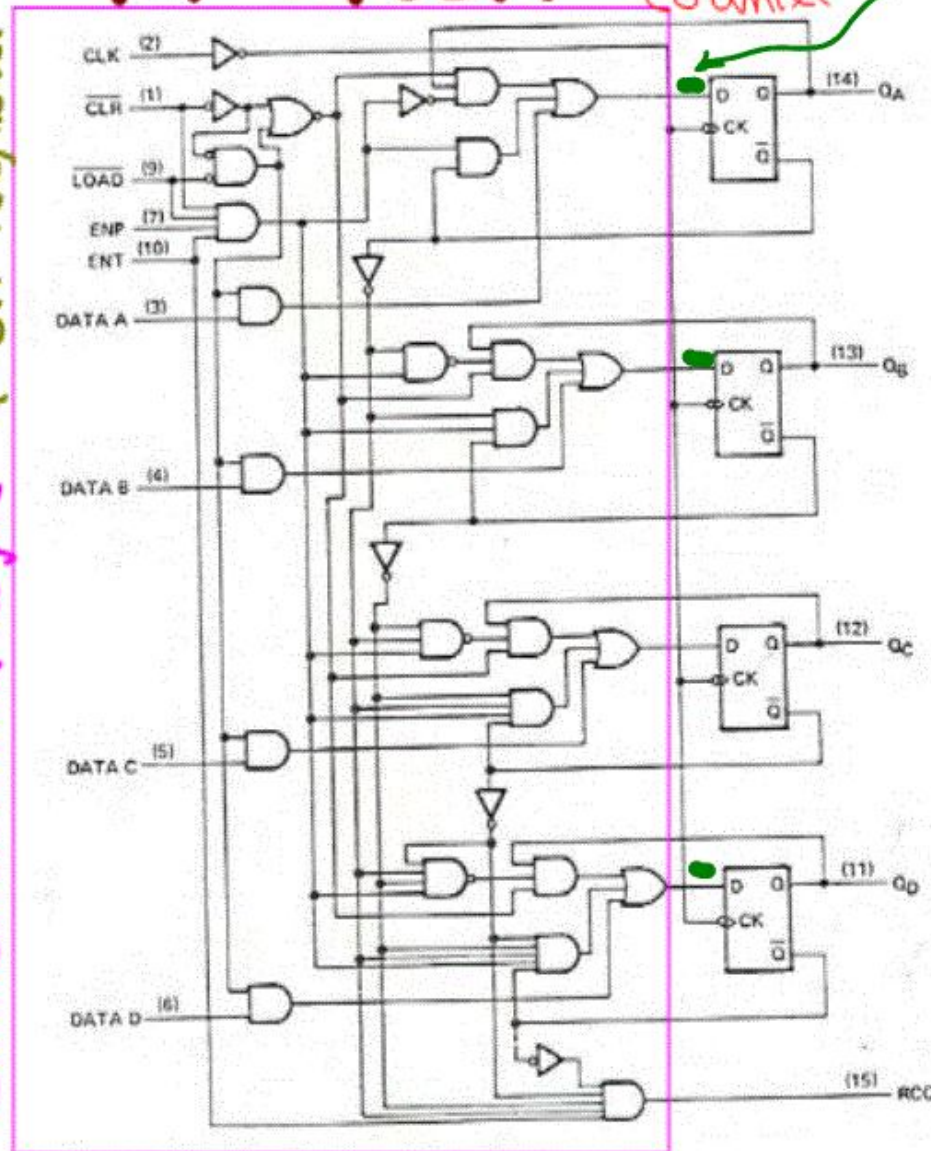
upstream of the
4-bit Register

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

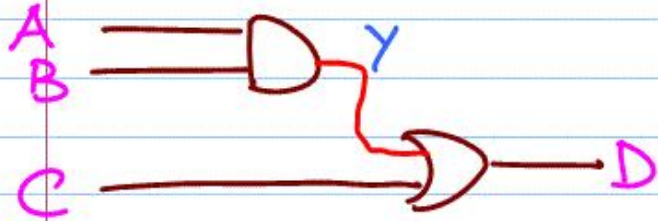
74LS163

20+ Assign Statements

Order of writing these

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

Data flow model

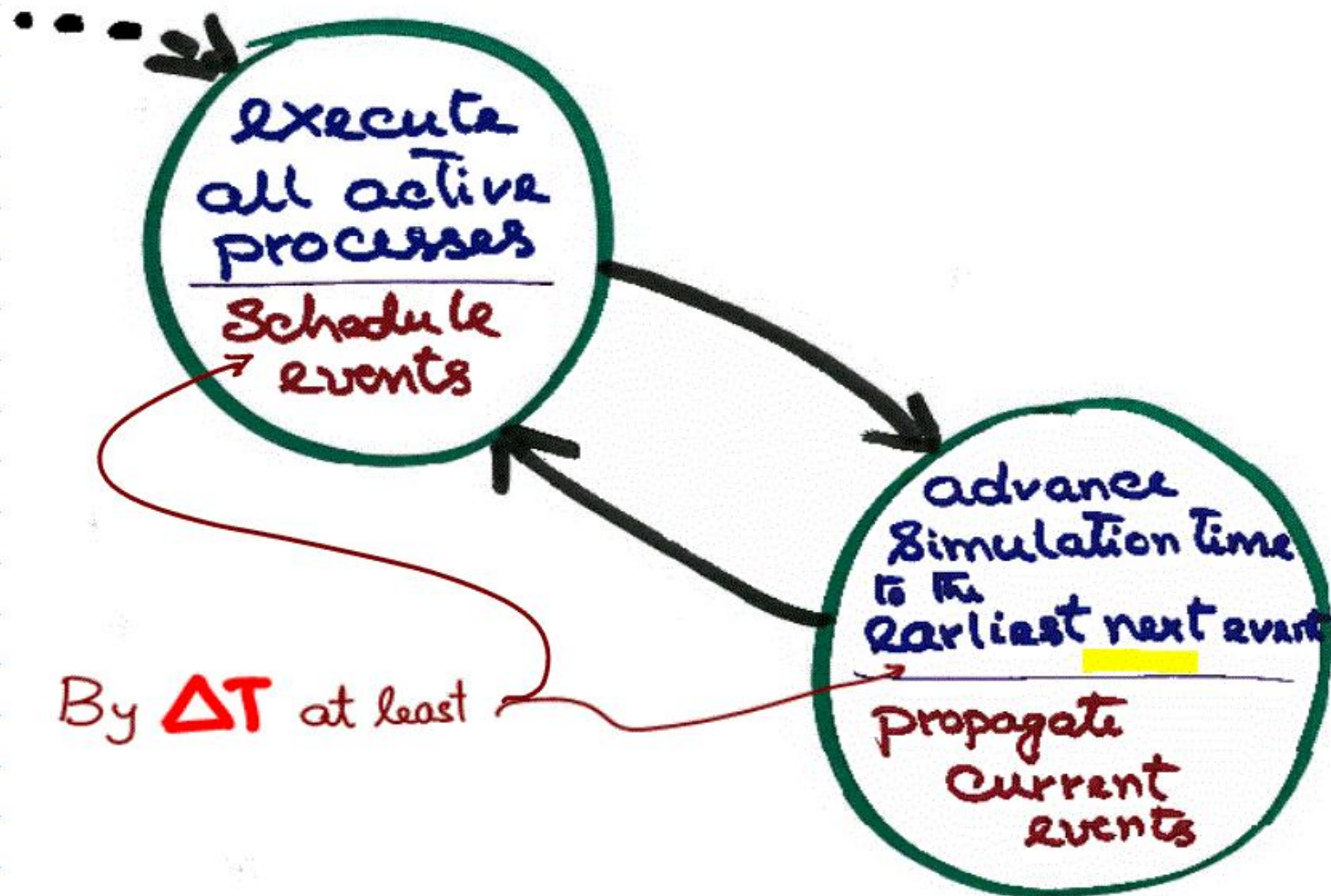


```
assign Y = A & B;  
assign D = Y | C;
```

```
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

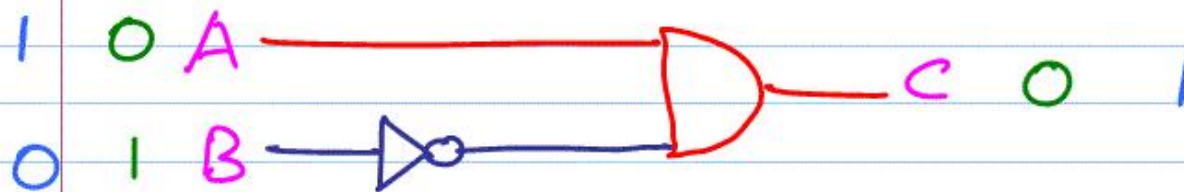


Scene 2

Behavioral/functional simulation before synthesis

ZERO-DELAY MODELING

ΔT delay



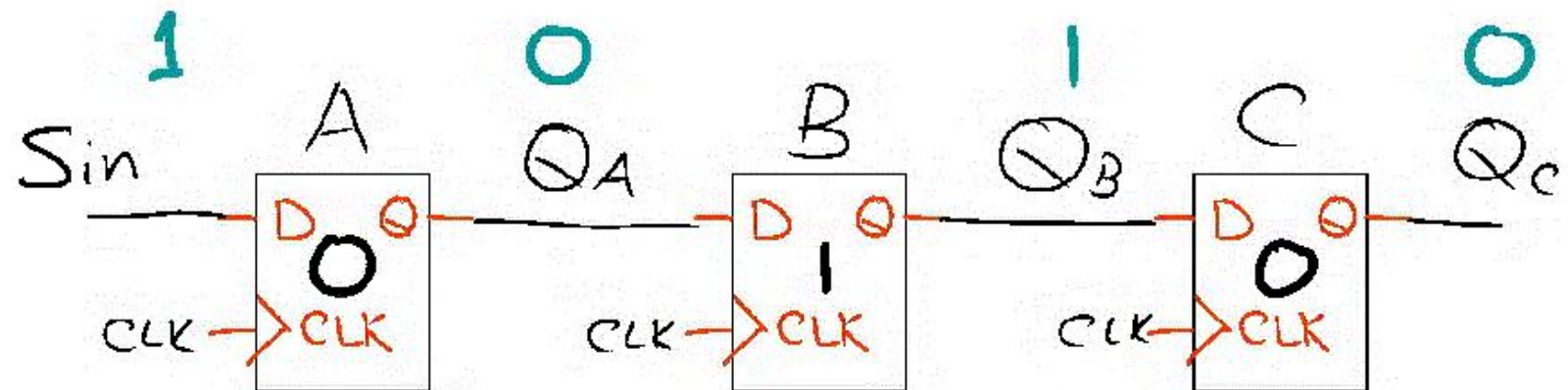
initially

at 20ns

At 20 ns _____ will be evaluated.

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

RIGHT-SHIFT register



Current content =

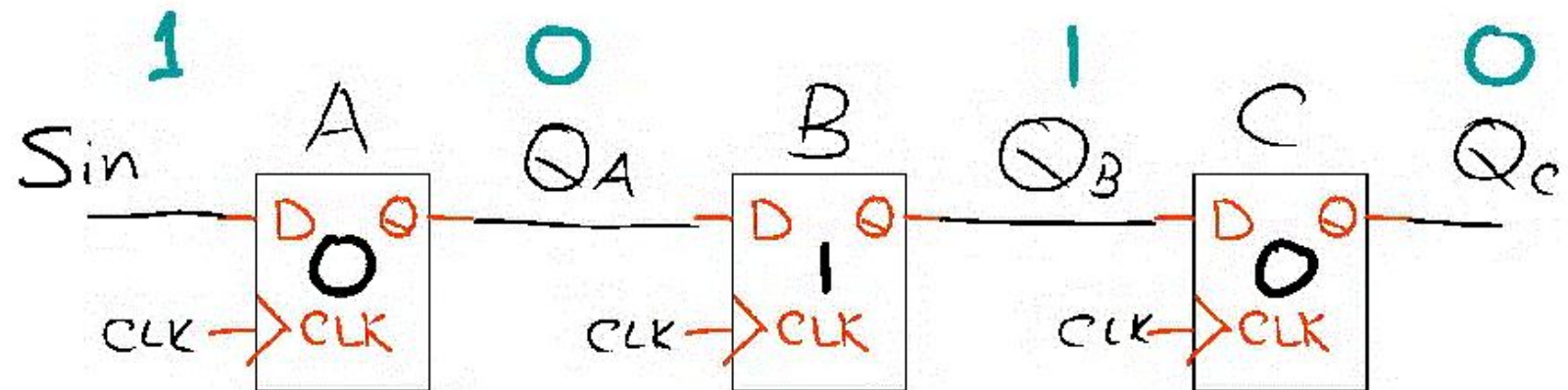
A	B	C
0	1	0

New content =

A	B	C
1	0	1

focus on B

RIGHT-SHIFT register



Looks like a **RACE**
Condition!

RACE

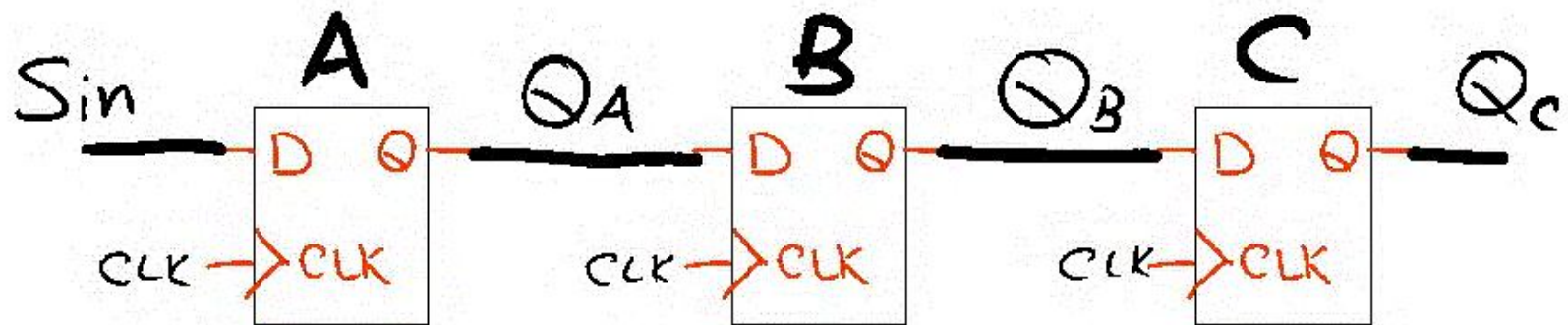
Condition $\equiv$

a condition that produces
non-deterministic values

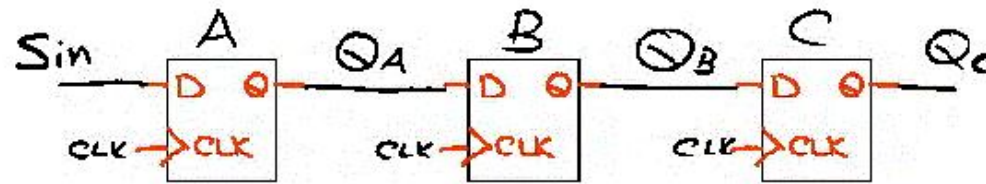
Circuit that produces
non-deterministic values is

no good!

How do we **avoid**
race condition?



Shift one bit at a Time?

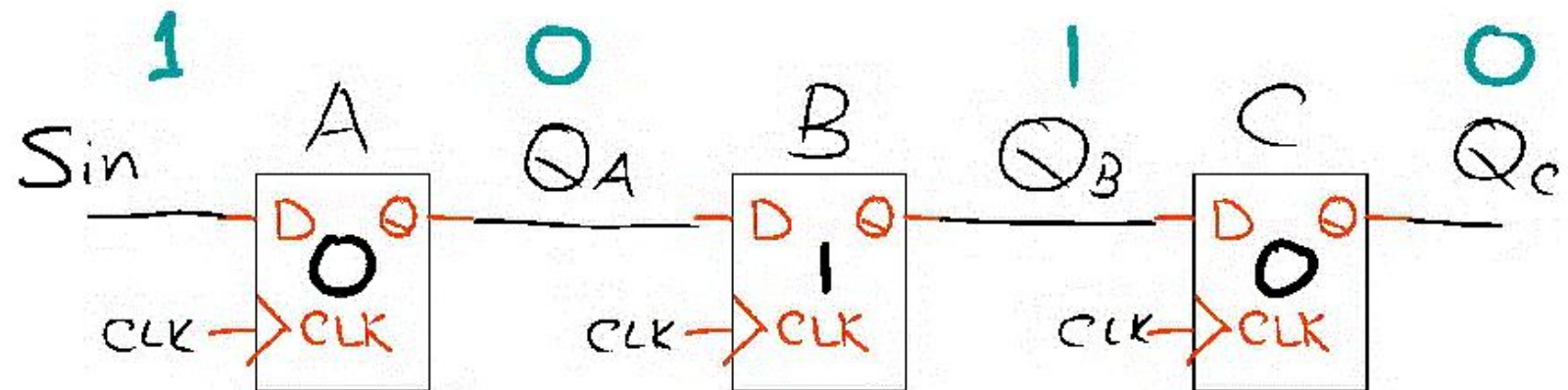


Shift one bit at a Time?

It works, but....

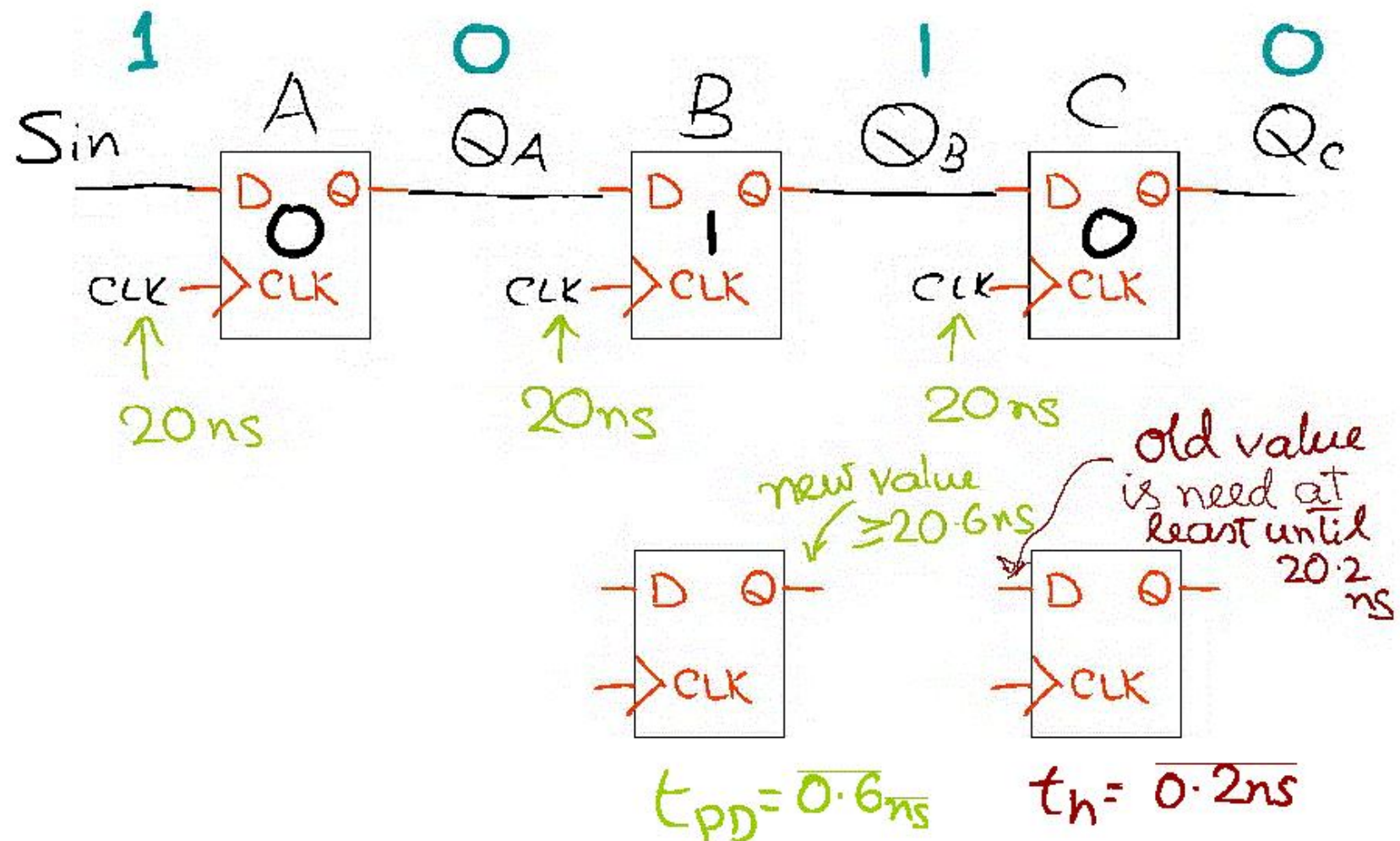
At this rate, it takes 32 clocks
to right-shift by 1 bit
in a 32-bit Shift Register.

RIGHT-SHIFT register

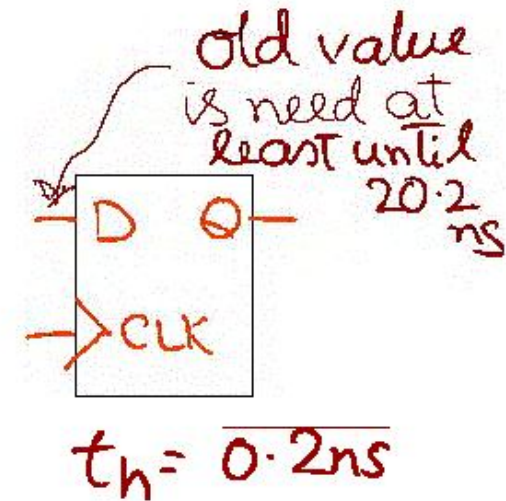
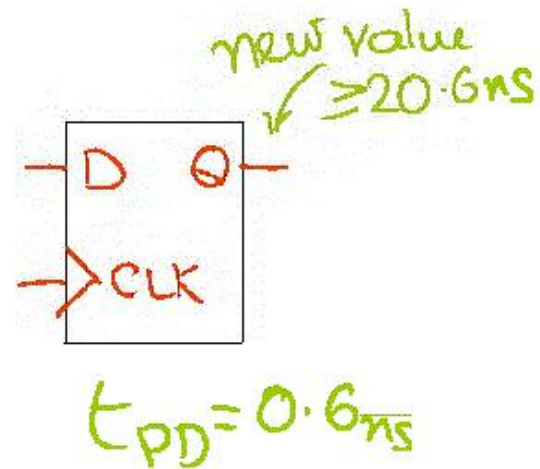


The **RACE** condition is **prevented**
in hardware design by **proper**
Timing design !

RIGHT-SHIFT register



TIMING Design avoids race



$t_{PD} \geq t_h$

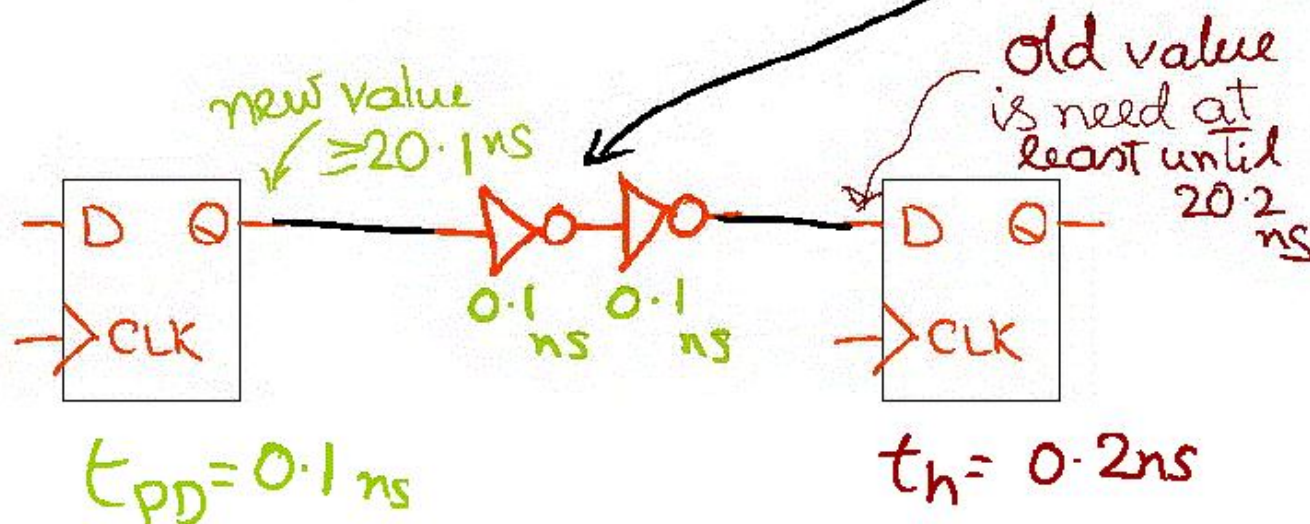
Golden rule!

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

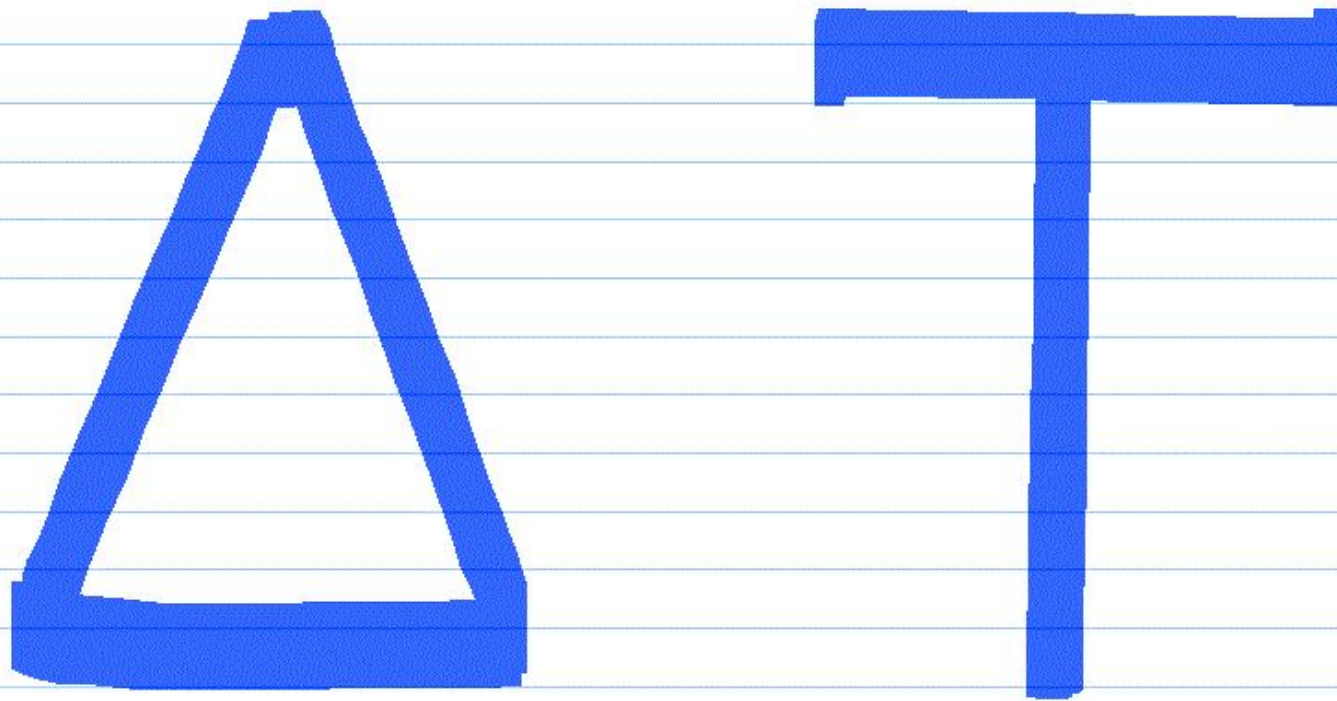
Static Timing Designer

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;
...;
...;

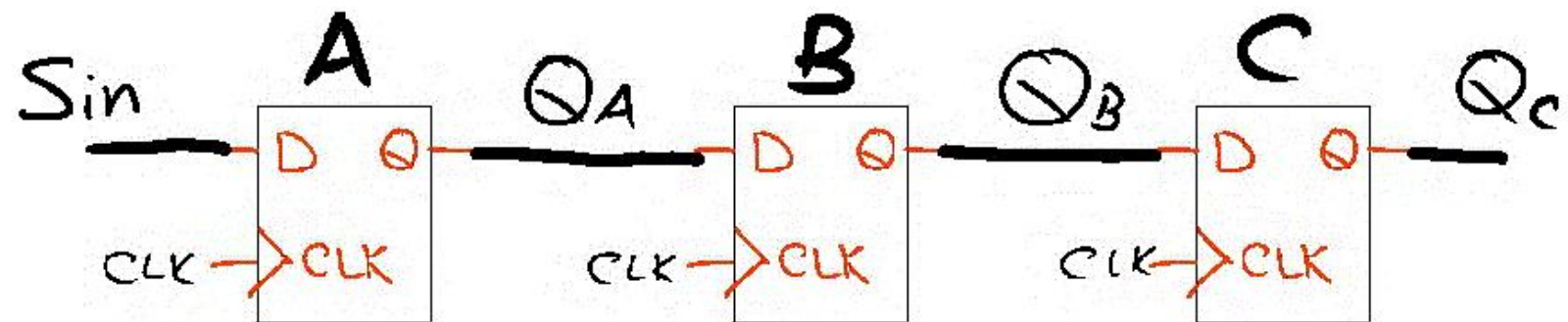
```

non-blocking assignment

L.H.S is updated only after ΔT !

important 

RIGHT-SHIFT register



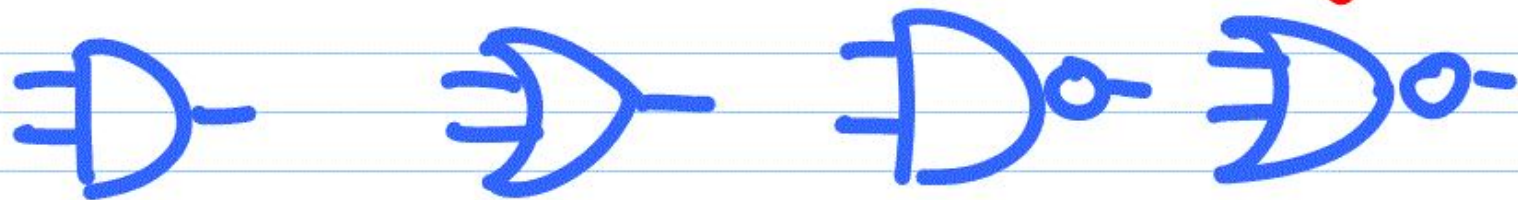
*Scheduled to
change after ΔT*

```
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

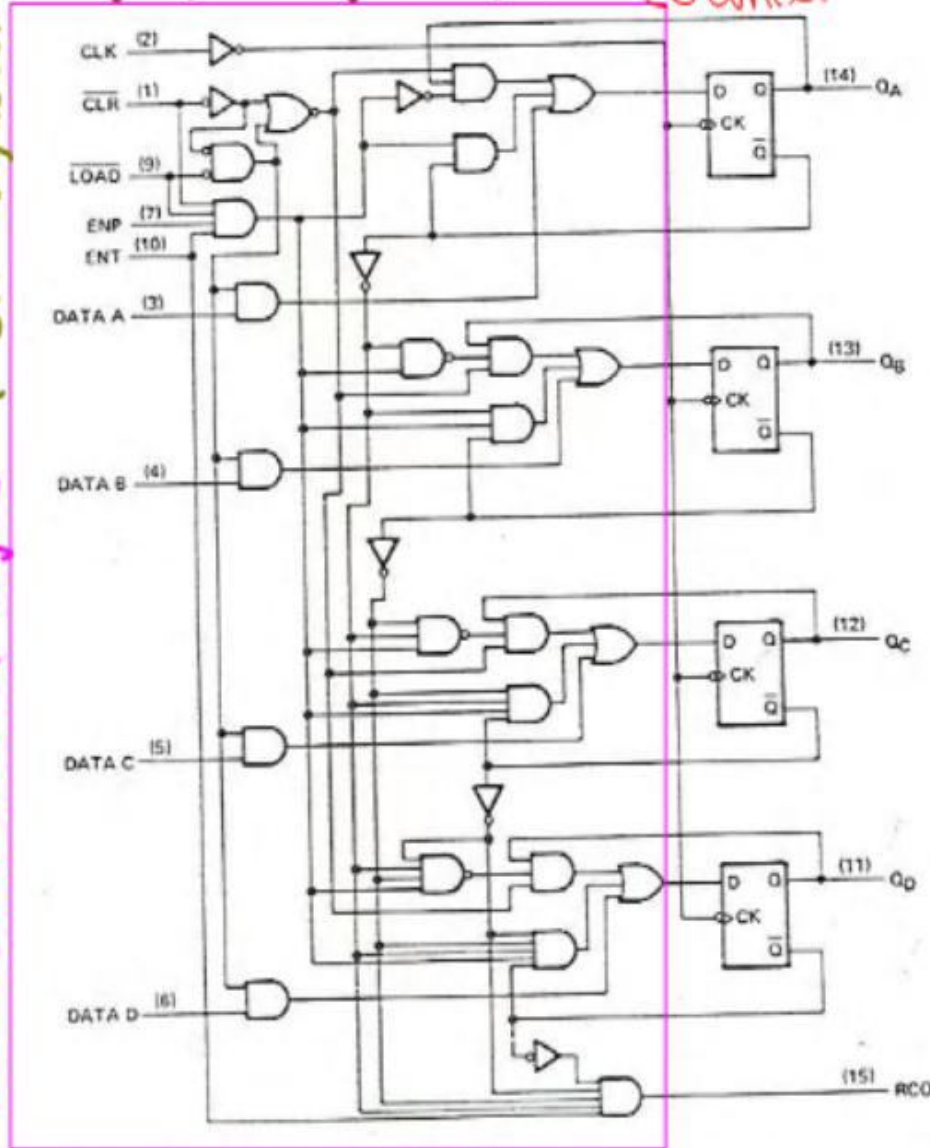
Build subsystems: ALU, Register file

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

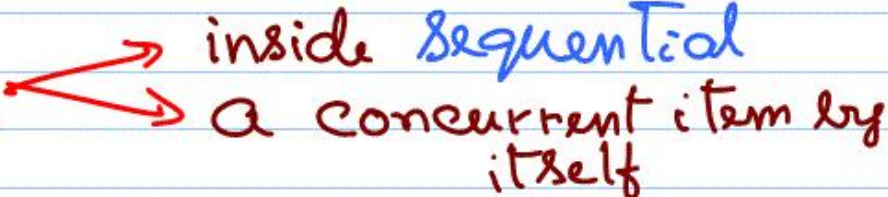
Say, in a HDL design, there are

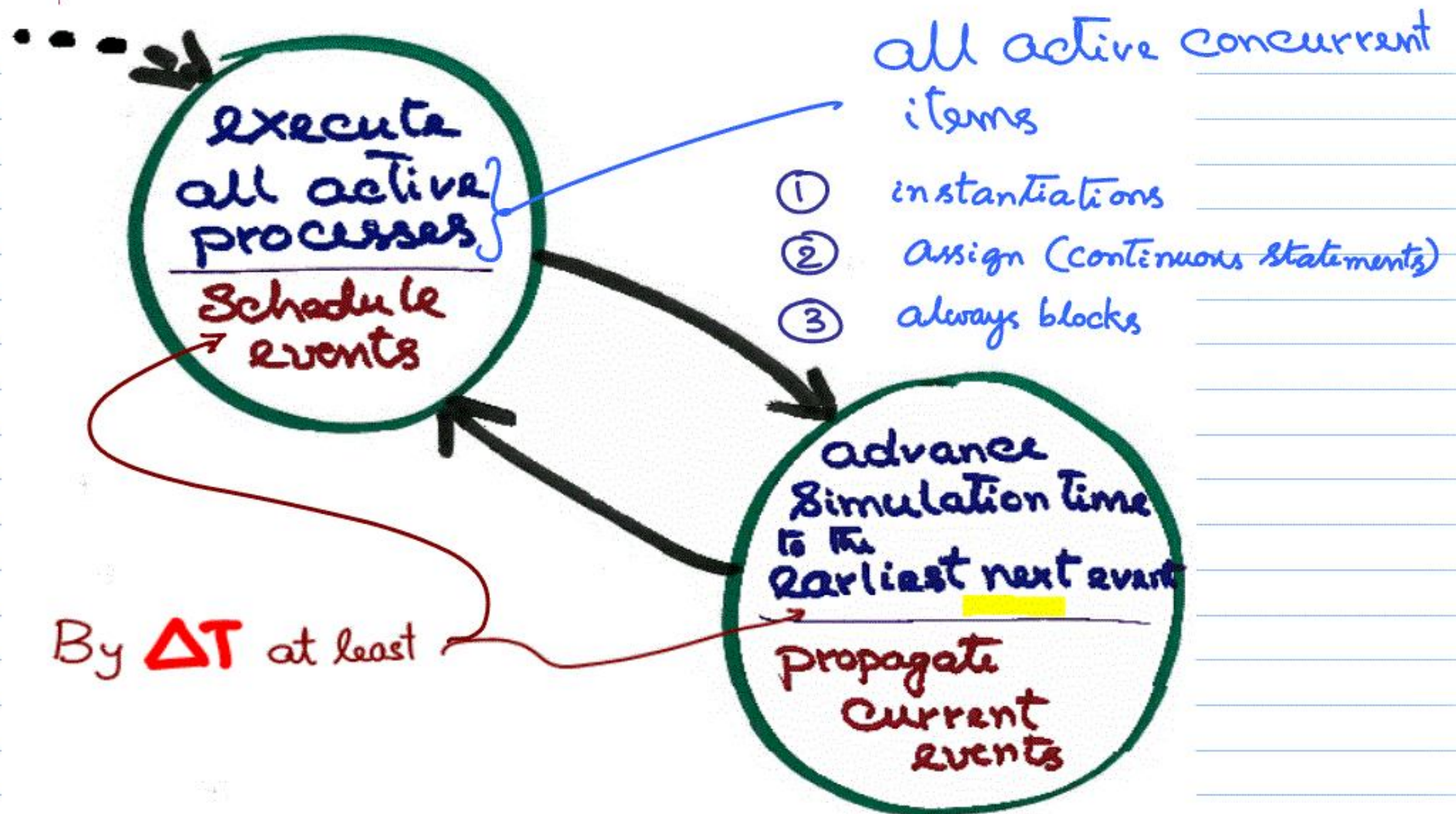
14 instantiations

8 assign statements

6 always blocks

28 total concurrent items

always block  inside sequential
a concurrent item by itself



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



Assign $C = A \& B;$

implicit sensitivity list items

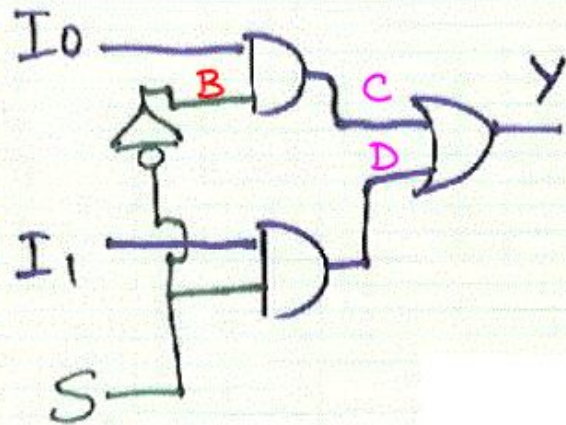
always @ (A, B)

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

```

```

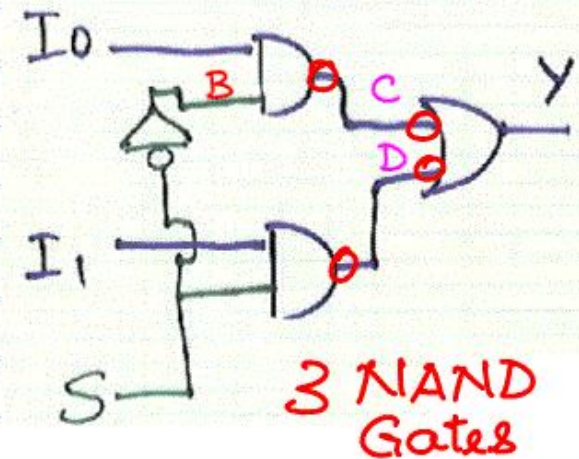
        Y = I0;

```

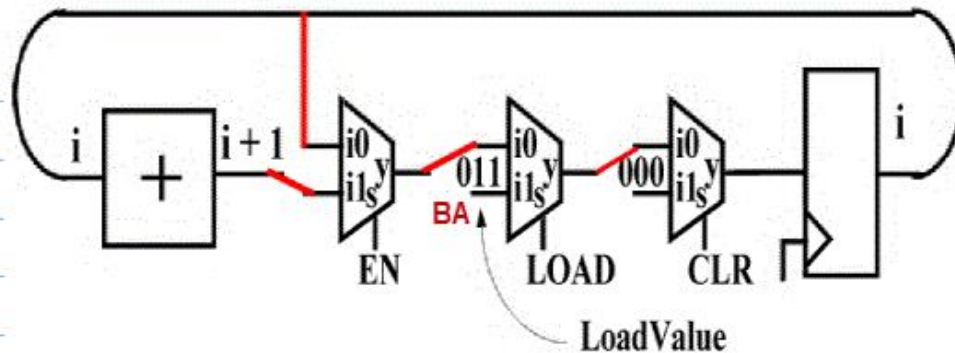
```

end

```



← preferred

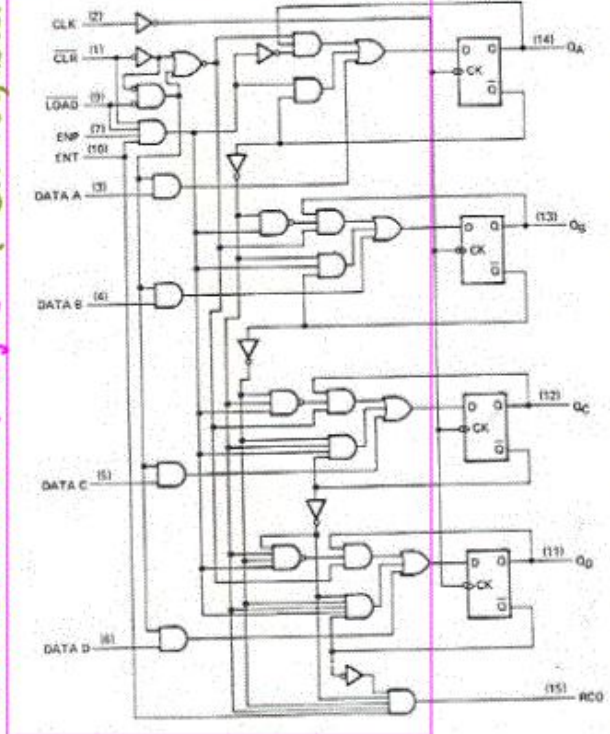


```
always @ (posedge clk)
```

```
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
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.

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