

Note Title

9/9/2012

Verilog Coding

Module #4

A few

SEQUENTIAL STATEMENTS

For use in *always* and *initial* procedural blocks and also in *functions* and *tasks*

By
Gandhi Puvvada



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)*

page 44, 45

always

```

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

```

always @ (

) 

no ;

begin 

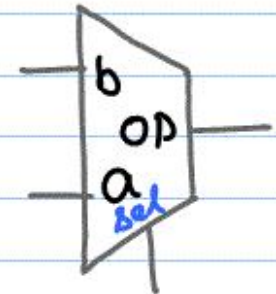
Sequential
Statements


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.

sequential with-in each



Concurrent

page 44, 45

always

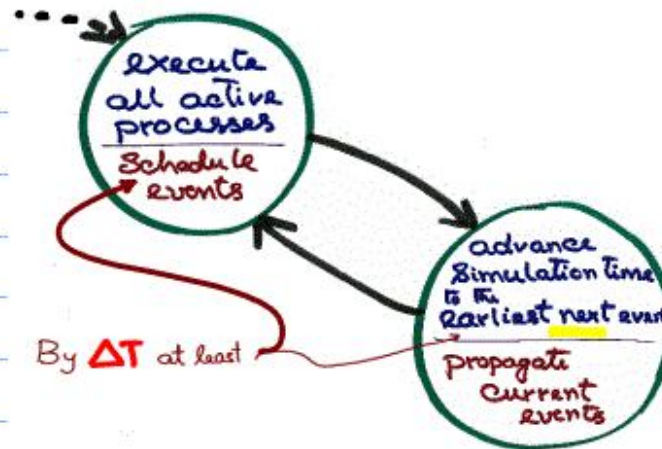
combination logic (a mux)

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

use
commas

```
always @(<event_expression>)
  begin
    → sequential statement(s)
  end
```

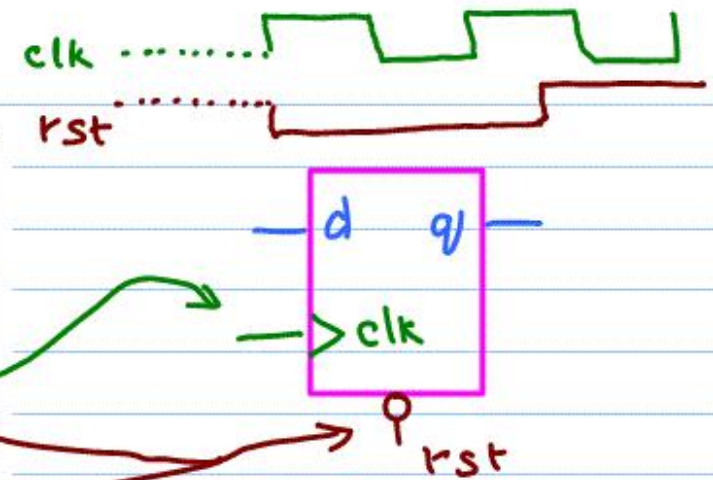
also called
sensitivity list



Sequential logic (a D-FF)

```

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



if (rst == 0)

notice "double equal" ==

if (!rst)

if (~rst)

equality check

operator

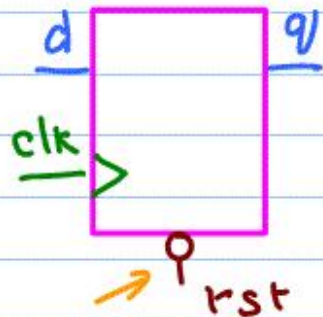
page 106

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

Verilog 2001 Syntax Commas

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

Suppose

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

Combinational Procedures

Rule 1: A RTL combinational procedure must have a complete sensitivity list, where all of the signals read in the process are included in the sensitivity list.

complete sensitivity list,

page 10

Verilog2001 Automatic Event List

Verilog2001 also adds an automatic event list for combinational logic only:-

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

* wild card

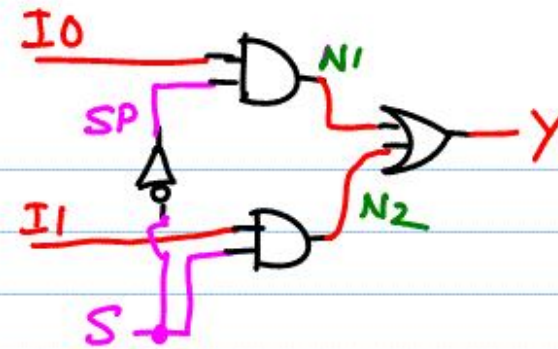
```
wire N1,N2,SP;
```

```
assign SP = ~ S;
```

```
assign N1 = I0 & SP;
```

```
assign N2 = I1 & S;
```

```
assign Y = N1 | N2;
```



```
wire N1,N2,SP;
```

```
always @ (I0, I1, S)
```

```
SP = ~ S;
```

```
N1 = I0 & SP;
```

```
N2 = I1 & S;
```

```
Y = N1 | N2;
```

Does
Order matter?

errors?

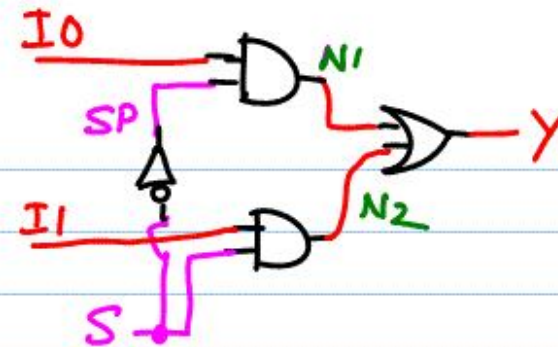
```
wire N1,N2,SP;
```

```
assign SP = ~ S;
```

```
assign N1 = I0 & SP;
```

```
assign N2 = I1 & S;
```

```
assign Y = N1 / N2;
```



```
reg wire N1,N2,SP;
```

```
always @(I0,I1,S)
```

```
begin
```

```
    SP = ~ S;
```

```
    N1 = I0 & SP;
```

```
    N2 = I1 & S;
```

```
    Y = N1 / N2;
```

```
end
```

Does
Order matter?

Yes

SOLUTION

errors?

page 84

initial

used in TESTBENCHES

no event list (i.e. no sensitivity list)

Sequential

```

initial begin
    ip = 8'h00; //initialize
    #10 ip = 8'hf0; // pattern one
    #10 ip = 8'hel; // pattern two
    #10 ip = 8'hd2; // pattern three
    #10 ip = 8'hc3; // last pattern
end

```

10ns

page 21

\timescale 1ns / 10ps

\timescale 1ns / 100ps

\timescale unit / precision

page 82

if ... else if ... else

```
if (condition)
    procedural_statement(s)
else if (condition)
    procedural_statement(s)
else
    procedural_statement(s)
```

statement(s)

use

begin

end

"else if" is optional

```
if (condition)
    procedural_statement(s)
else if (condition)
    procedural_statement(s)
else
    procedural_statement(s)
```

just "if"

```
if (condition)
    procedural_statement(s)
else if (condition)
    procedural_statement(s)
else
    procedural_statement(s)
```

if (condition)

if (sel)

if (sel == 1)

if (sel == 1'b1)

if (!rst)

if (rst == 0)

if (rst == 1'b0)

page 104

Logical Operators

Reduce both operands to a single bit, and then performs a single bit operation.

!	not
&&	and
	or

if (~rst)

Bit-Wise Operators

~	not
---	-----

page 105

Relational Operators

Comparison operators:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

page 106

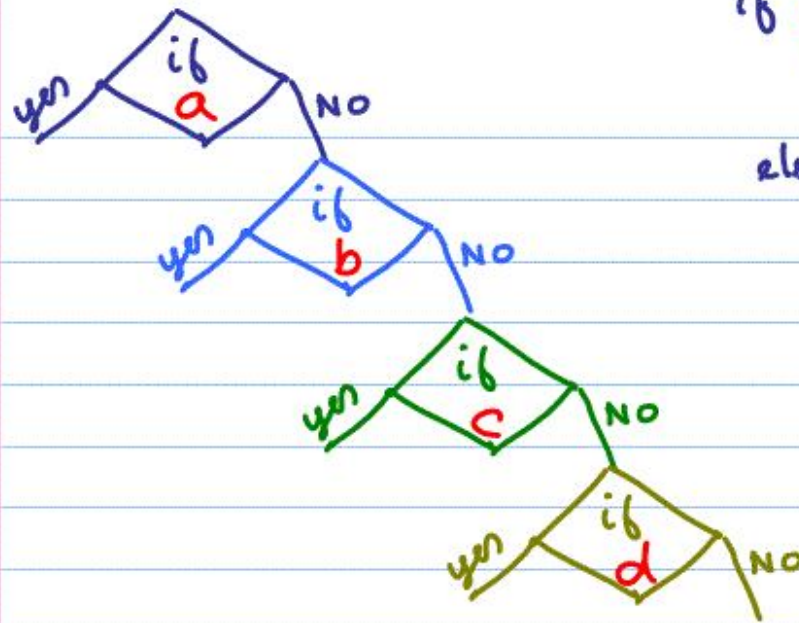
Equality Operators

There are two forms of equality, logical and case.

The difference is in the handling of the x and z values:

==	logical equality
!=	logical inequality
===	case equality
!==	case inequality

only in Test bench

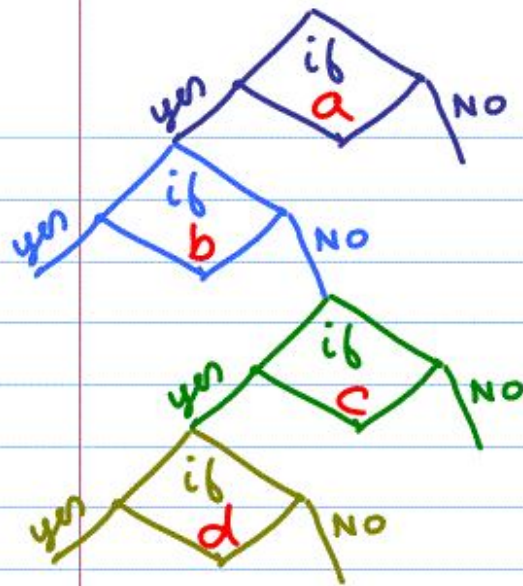


```
if a
begin
  // yes statements
end
else if b
```

```
begin
  // yes statements
end
else if c
```

```
begin
  // yes statements
end
```

```
else if d
begin
  // yes statements
end
else
```



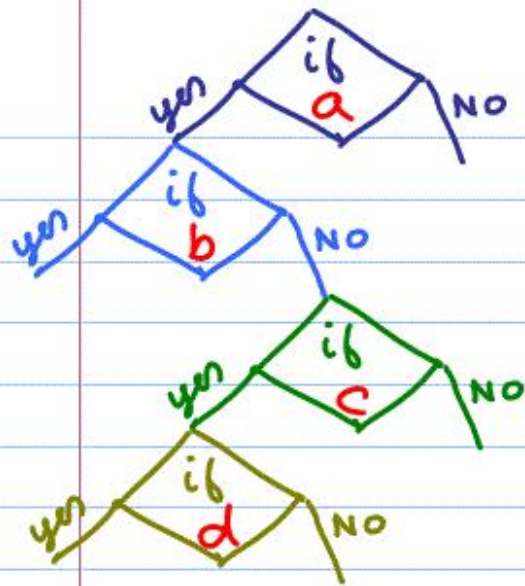
```
if a
begin
  // yes statements
end
else
```

```
if b
begin
  // yes statements
end
else
```

```
if c
begin
  // yes statements
end
else
```

```
if d
begin
  // yes statements
end
else
```

SOLUTION



```
if a
begin
// yes statements
end
else
```

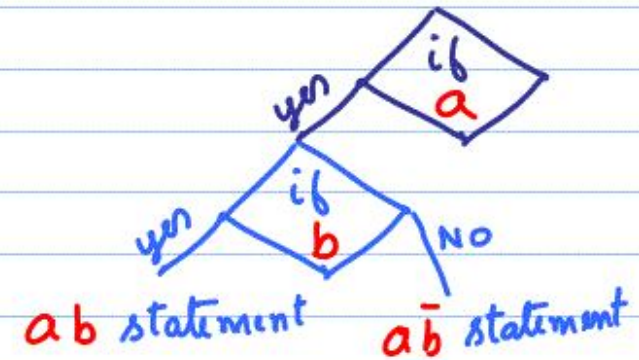
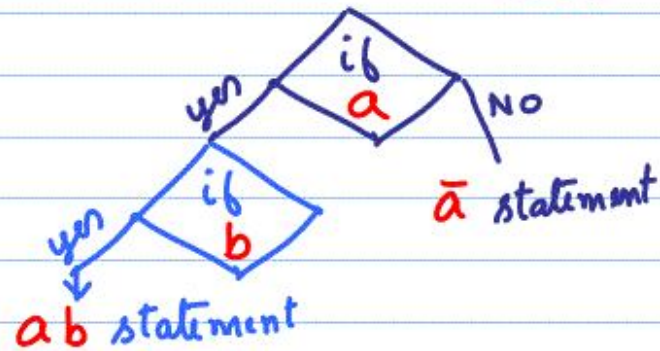
```
if b
begin
// yes statements
end
else
begin
end
```

```
if c
begin
// yes statements
end
else
```

```
if d
begin
// yes statements
end
else
```

begin
end

Use begin...end as often as you need to
make it clear, which "else" applies to which "if"
If you are not careful, you may want to code the
 $\bar{a}$ statement below, and you may end up coding the
 $a\bar{b}$ statement instead!



```
if a
  if b
    //ab statement
  else
    //a statement
```

```
if a
  if b
    //ab statement
  else
    //a statement
```

Both the above else statements, irrespective of indenting and coloring, should behave the same way!

Note:

```

if a
  if b
    //ab statement
  else
    //a $\bar{b}$  statement

```

```


if a
  if b
    //ab statement
  else
    //a $\bar{b}$  statement


```

```

if a
  begin
    if b
      //ab statement
    end
  else
    //a $\bar{b}$  statement

```

Both the above else statements, irrespective of indenting and coloring, should behave the same way! **Yes both** $\Rightarrow$ //a $\bar{b}$ statement

Note: "else" is associated with the nearest "if".

SOLUTION

Extract from page 128 (pdf page 151 of 879) of

IEEE Std 1364-2001
(Revision of
IEEE Std 1364-1995)

Because the else part of an if-else is optional, there can be confusion when an else is omitted from a nested if sequence. This is resolved by always associating the else with the closest previous if that lacks an else. In the example below, the else goes with the inner if, as shown by indentation.

```
if (index > 0)
    if (rega > regb)
        result = rega;
    else      // else applies to preceding if
        result = regb;
```

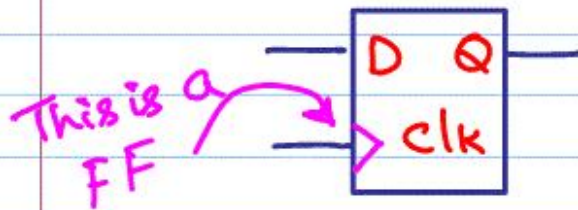
If that association is not desired, a begin-end block statement shall be used to force the proper association, as shown below.

```
if (index > 0) begin
    if (rega > regb)
        result = rega;
end
else result = regb;
```

The following process
infers a transparent latch:

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

no "else" $\Rightarrow$ Latch



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

```
  if (ctrl)
```

```
    b <= a;
```

```
  else
```

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

?

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

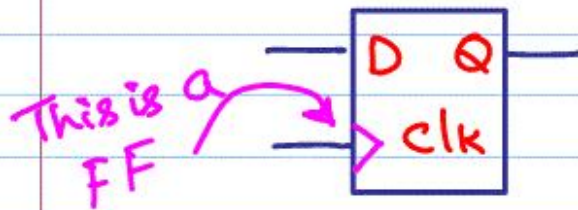
```
  b <= a;
```

?

The following process
infers a transparent latch:

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

no "else" $\Rightarrow$ Latch



SOLUTION

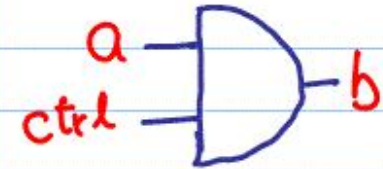
```
always @ (ctrl, a)
```

```
  if (ctrl)
```

```
    b <= a;
```

```
  else
```

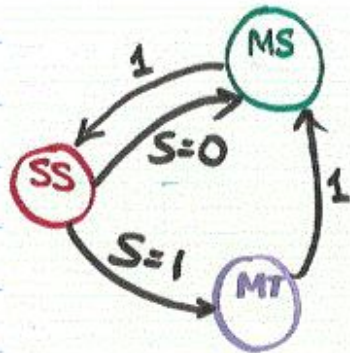
```
    b <= 1'b0;
```



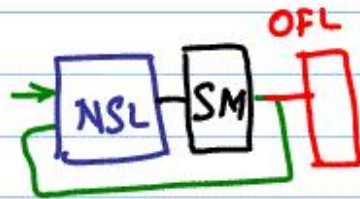
```
always @ (posedge ctrl)
```

```
  b <= a;
```





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

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.

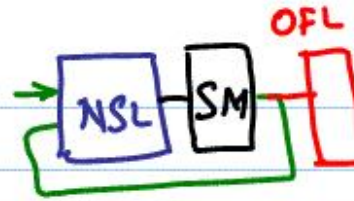
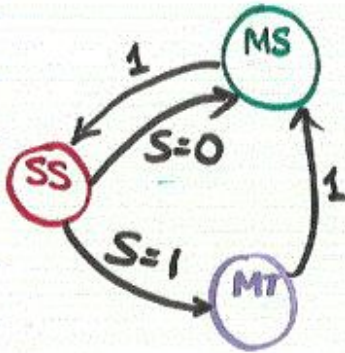
Please remember this basic rule and the intent behind it.

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)

OFL



// state machine output function logic

```
assign msg = (state == `MS) ;
```

```
assign mtg = (state == `MT) ;
```

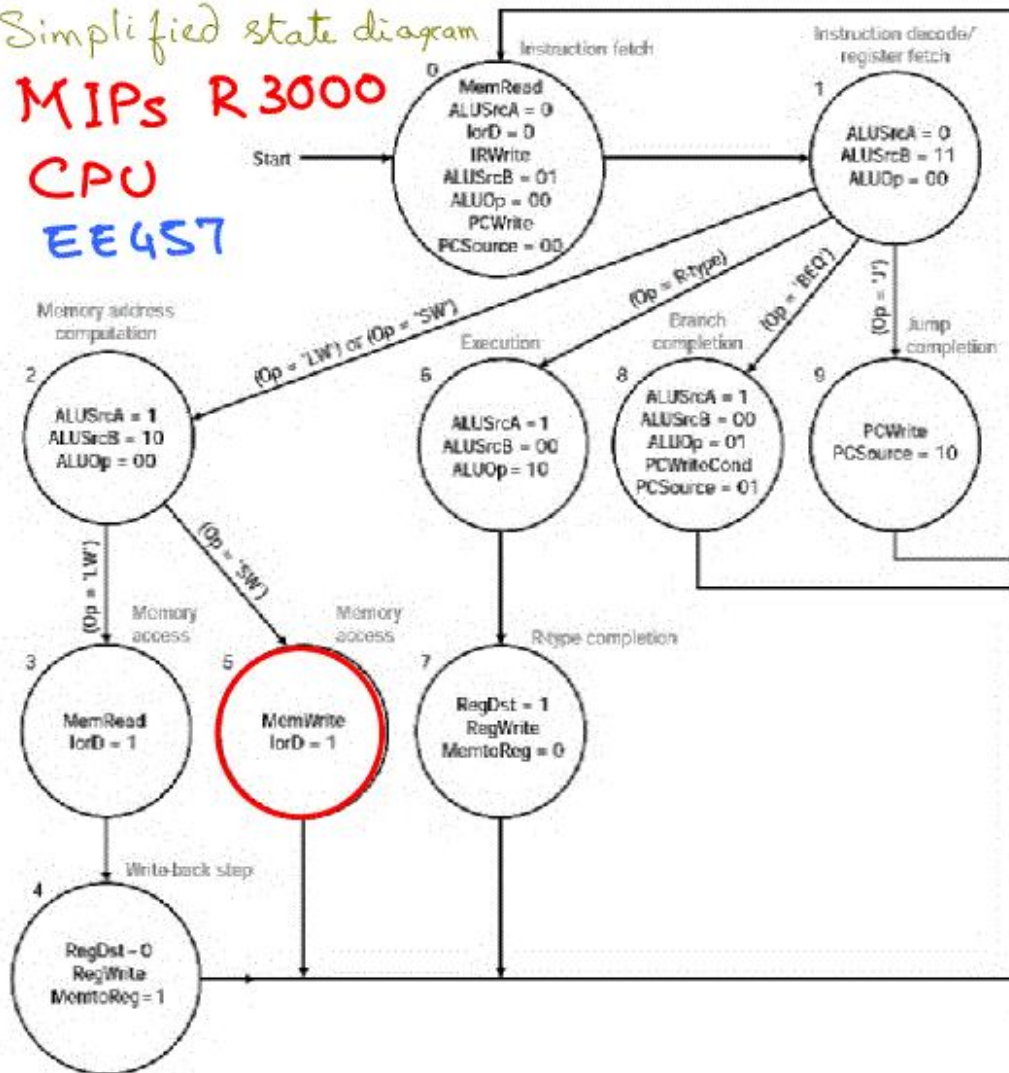
```
assign ssg = (state == `SS) ;
```

Simplified state diagram

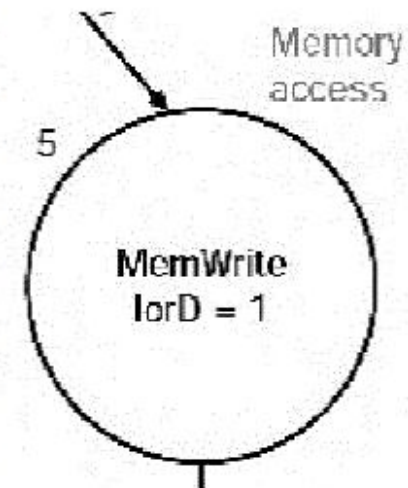
MIPS R3000

CPU

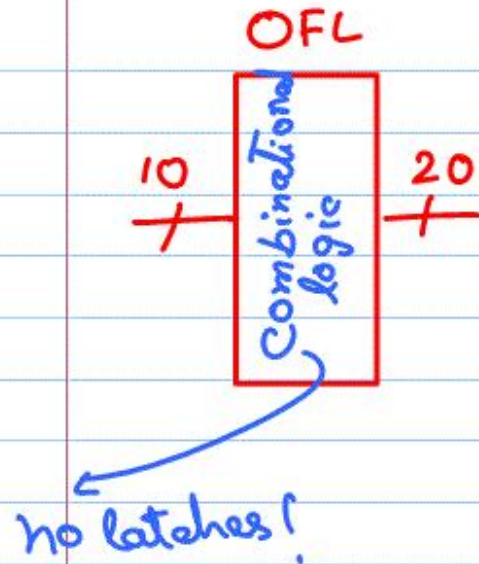
EE457



Say 10 states
and total 20 outputs



MemWrite goes active
only in state 5.
Other wise it is inactive!



//OFL

/* method 1: In each of the case branches, list all the 20 outputs.

10 * 20 = 200 lines of Code! */

/* method 2: Inactivate all the outputs in the first 20 lines of the always block (default assignments) and Over-ride these selectively in the case branches. */

page 54

case

Syntax

```
case (case_expr)
  item_expr : statement(s);
  item_expr : statement(s);
  ...
  default : statement(s);
endcase
```

begin

end

priority

one word (note there is no endif)

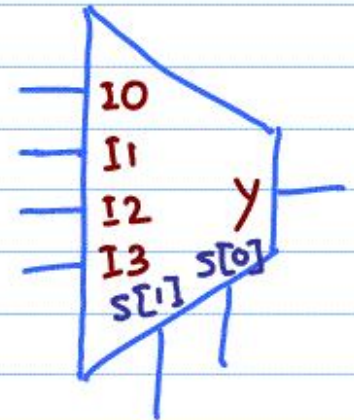
The remaining case-expression values

4 to 1 mux coding using a case statement

```
module mux_case (I0, I1, I2, I3, S, Y)
input I0, I1, I2, I3;
input [1:0] S;
output Y;

always @(
    case (S)

endcase
endmodule
```



```
module mux_case (I0, I1, I2, I3, S, Y)
input I0, I1, I2, I3;
input [1:0] S;
output Y;

reg Y;

always @(I0, I1, I2, I3, S)
    case (S)
        2'b00: Y = I0;
        2'b01: Y = I1;
        2'b10: Y = I2;
        2'b11: Y = I3;
    endcase
endmodule
```

SOLUTION

Full Case

Case branches do not need to cover all values for the case expression, but this implies a latch for synthesis.

We do not want a LATCH in a combinational logic!

consider a 3 to 1 mux.

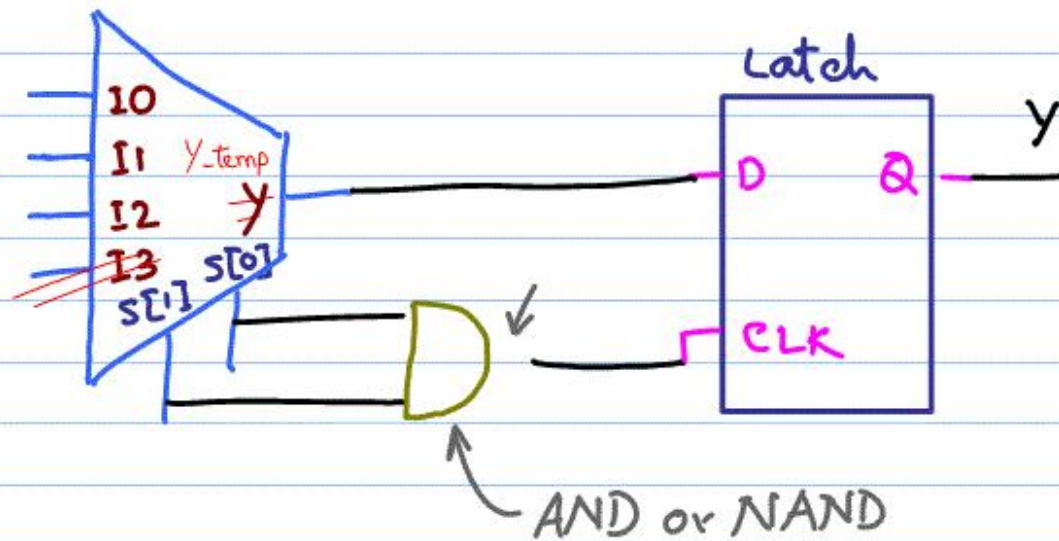
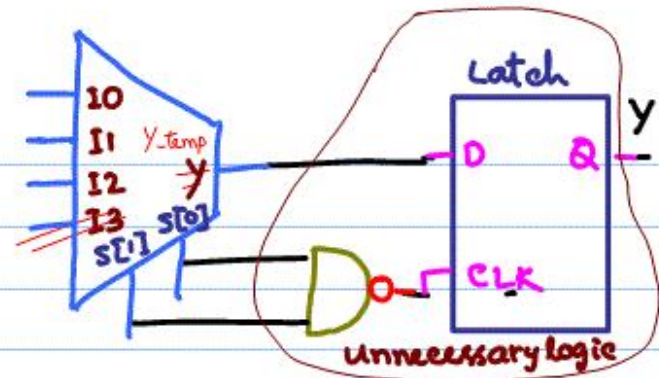
```
always @(I0, I1, I2, I3, S)
  case (S)
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    2'b11: Y = I3;
  endcase
```

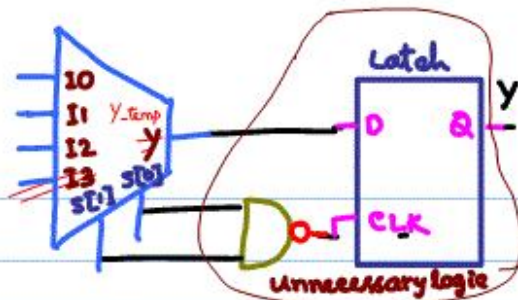
This combination never occurs on S, we were told.

```

always @(I0, I1, I2, I3, S)
  case (S)
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    2'b11: Y = I3;
  endcase

```





FIX the problem:
Add a **default**

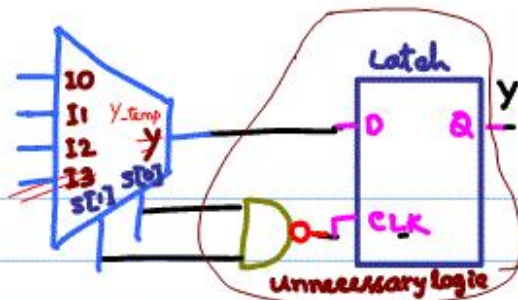
consider a 3 to 1 mux.

```
always @(I0, I1, I2, I3, S)
  case (S)
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    2'b11: Y = I3;
  endcase
```

```
always @(I0, I1, I2, S)
  case (S)
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    default: Y = 1'bX;
  endcase
```

default: Y = 1'bX;

This combination never occurs on S, we were told.



FIX the problem:
Add a special comment

//synopsys full_case ← preferred

//rtl_synthesis full_case ← any one

//pragma full_case

consider a 3 to 1 mux.

```
always @(I0, I1, I2, I3, S)
  case (S) //synopsys full_case
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    2'b11: Y = I3;
  endcase
```

This combination never occurs on S, we were told.

Which is better? And why?

```
always @(I0, I1, I2, S)
  case (S)
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    default: Y = 1'bX;
  endcase
```

```
always @(I0, I1, I2, S)
  case (S) // synopsys full_case
    2'b00: Y = I0;
    2'b01: Y = I1;
    2'b10: Y = I2;
    [redacted]
  endcase
```

From the "full_case parallel_case", the Evil Twins of Verilog Synthesis

The first method is better as the information is conveyed to both the simulation tool and the synthesis tool. In the case of the 2nd method, you let the simulator behave on the basis of the presence of the latch and told the synthesis tool to remove the latch. If your simulation appeared to have worked because of the presence of the latch, then you caused **"simulation synthesis mismatch"** and this kind of problems are difficult to trace.

The discussion on

`//synopsys full_case`

and

`//synopsys paralle_case`

is not very important at this point.

You can read the last few pages on **for** page 71

after you finish reading about **blocking** and **non-blocking** assignments.

This page is based on → "full_case parallel_case", the Evil Twins of Verilog Synthesis Recommendation from → Clifford E. Cummings Sunburst Design, Inc.

It is undesirable to use `//synopsys parallel_case` to override the implied priority of the case statement. Use other explicit coding.

Your simulator ignores any special comment. Your design may work in simulation but not after synthesis!

So try to avoid using the following: (Clifford's recommendation)

`//synopsys full_case parallel_case`

`//rtl_synthesis full_case parallel_case`

`//pragma full_case parallel_case`

`//synopsys full_case //synopsys parallel_case`

`//rtl_synthesis full_case //rtl_synthesis parallel_case`

`//pragma full_case //pragma parallel_case`

No LATCHES in combinational logic!

If you are describing a combinational logic, see that there is never a situation, where you did not assign any value for an output. Otherwise you get a **latch**!

GOLDEN RULE!

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

The **for** loop in a synthesizable code executes all its iterations without any advancement of simulation time.

The **for** loop will be unrolled by the synthesis tool.

The **for** loop is used for describing combinational logic.

integer i;

Loop iteration control i is declared as an integer.

for parity generator

```
integer i;
```

```
always @ (x)
```

```
begin
```

```
  P = 1'b0;
```

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

```
  begin
```

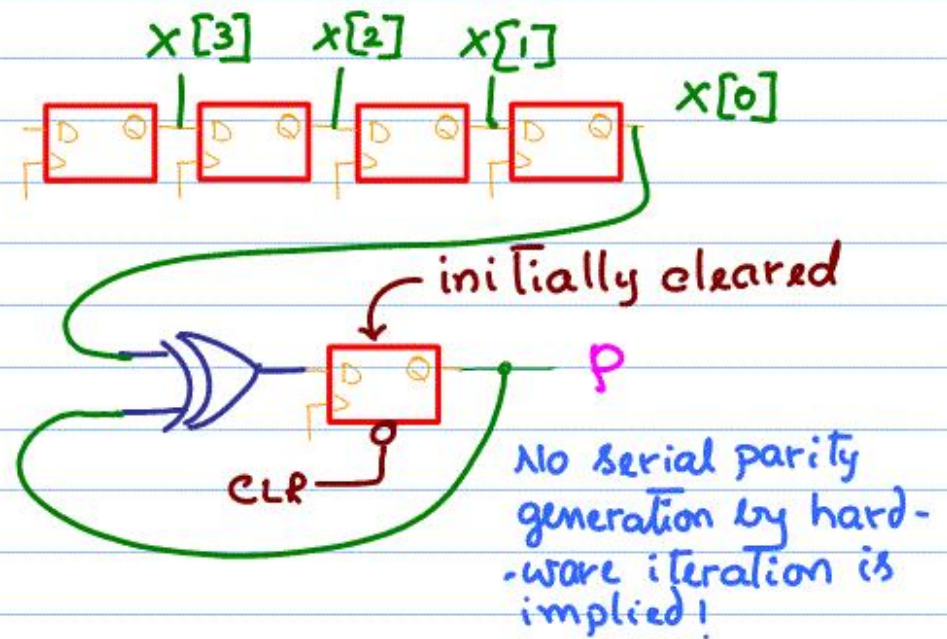
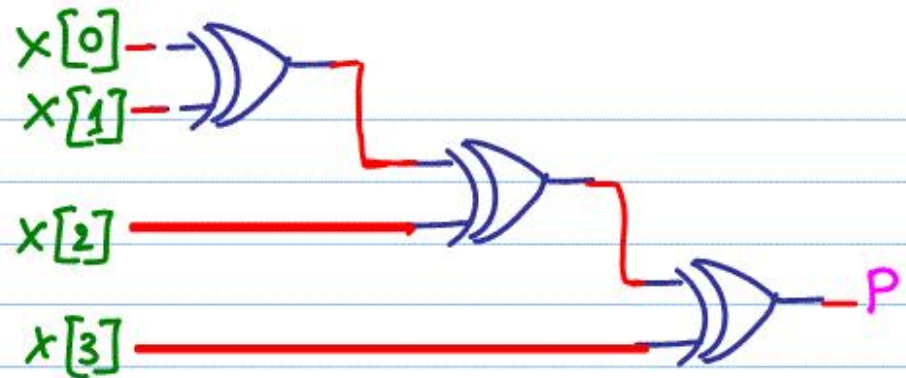
```
    P = P ^ x[i];
```

```
  end
```

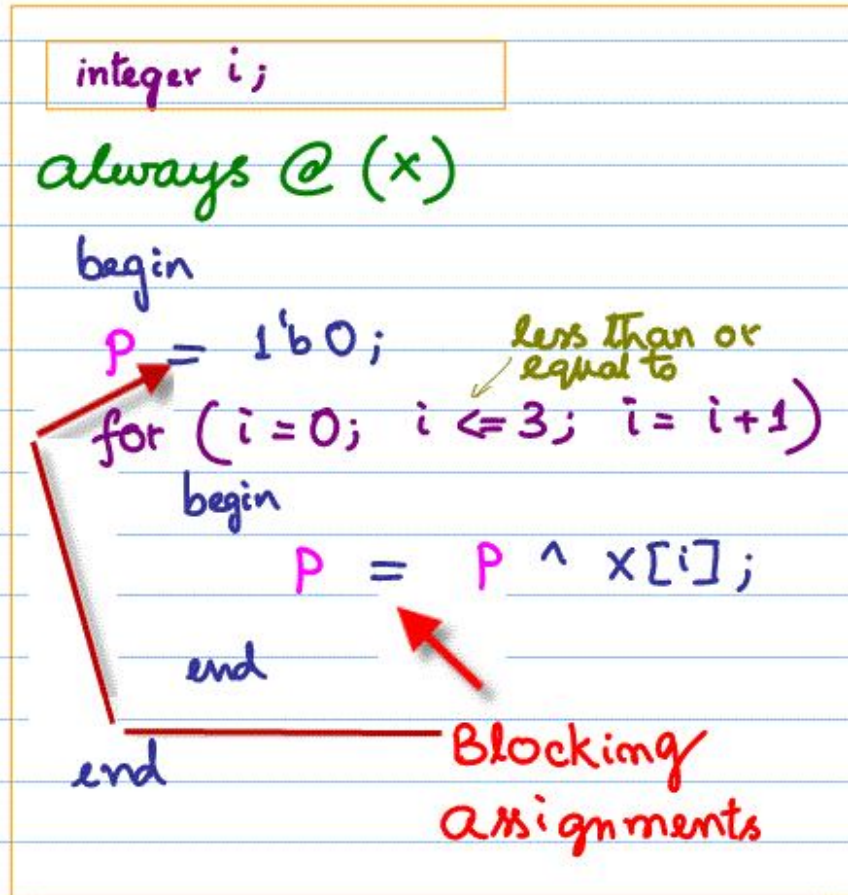
```
end
```

Blocking
assignments

less than or
equal to



Blocking or Non-blocking assignments



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

Blocking
assignment

No ΔT . Immediate
assignment

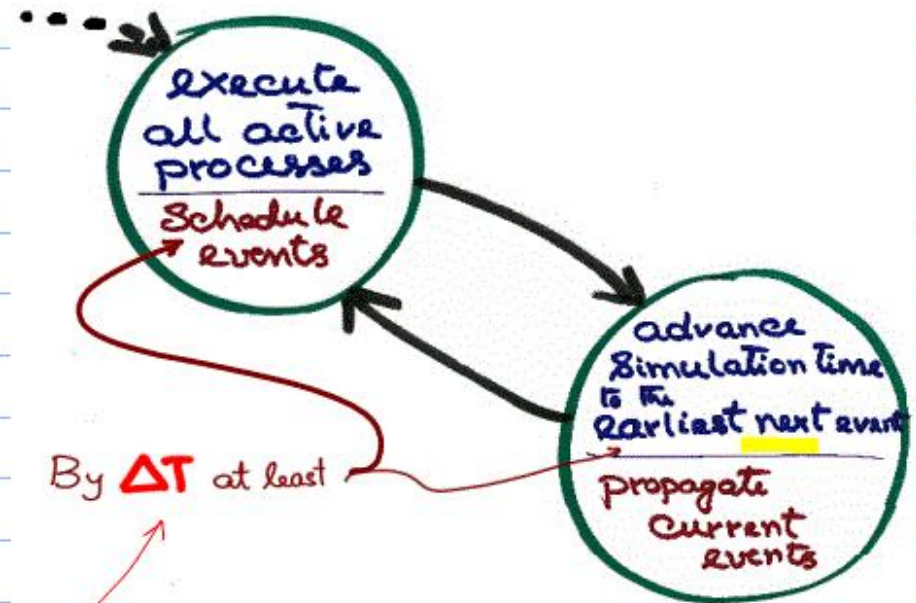
$$P \Leftarrow P \wedge x[i];$$

Non blocking
assignment

ΔT . Left side
item is scheduled to
receive its new value
in future after ΔT .

Non blocking assignments

So where do I use $\uparrow$



Whenever I need the ΔT

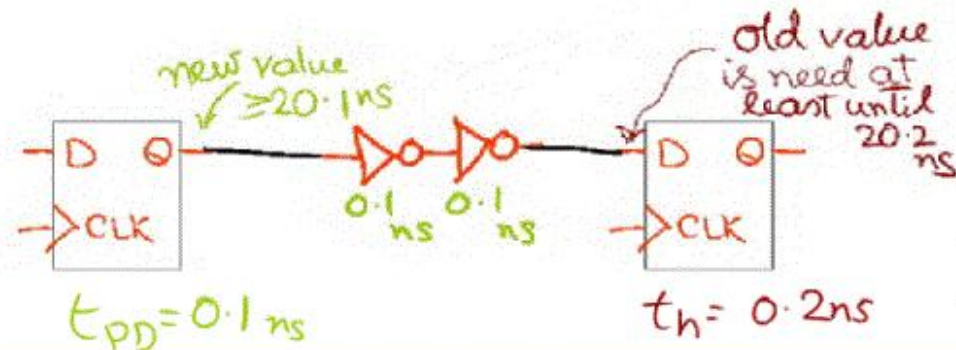
example: Hardware counter, FFs in State Memory, ...

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

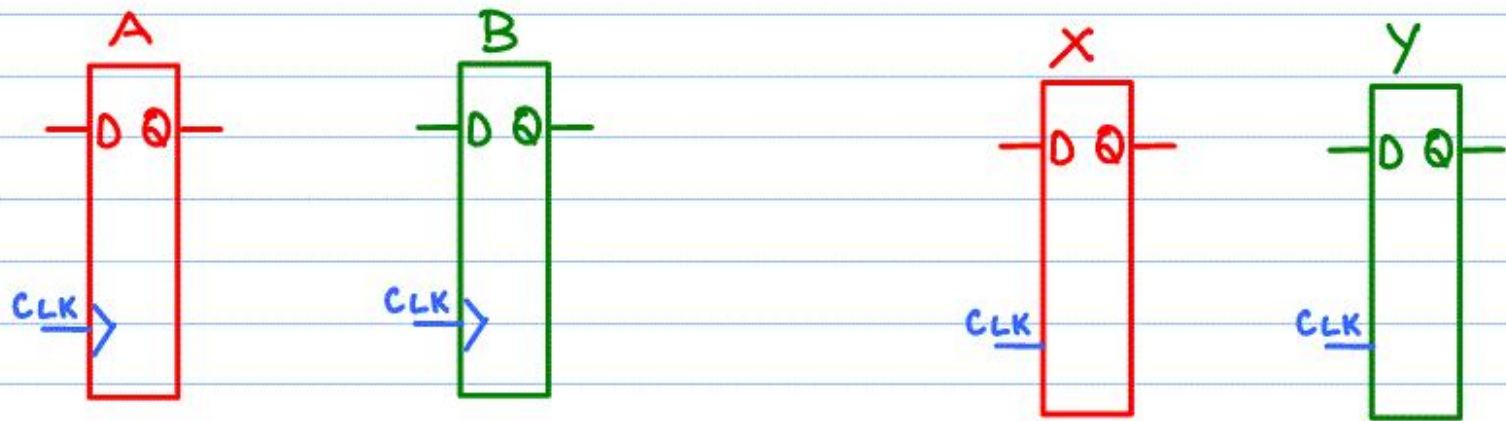


$$20 + \Delta T > 20.0$$

Can we **swap** the contents of **two registers**, A and B, simultaneously (on the same clock edge) without any need for temporary registers?

Software people use temporary variables.

Is swaping the contents of **two latches**, X and Y same as swapping the contents of **two registers** A and B?

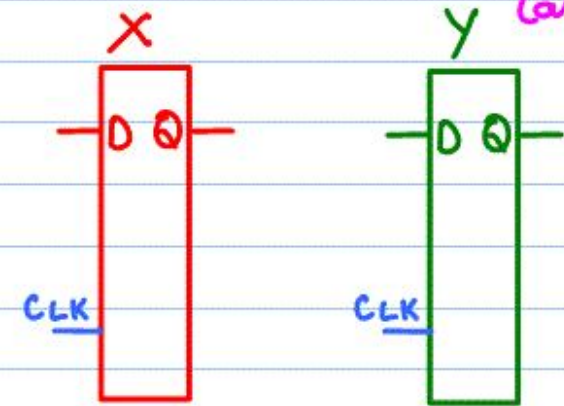
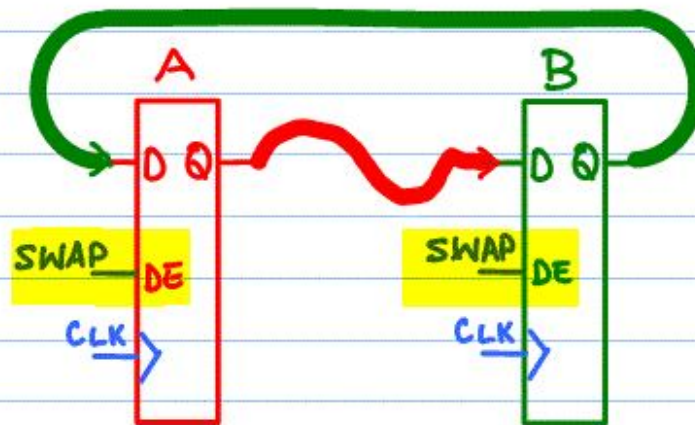


Can we **swap** the contents of **two registers**, A and B, simultaneously (on the same clock edge) without any need for temporary registers? **Yes**

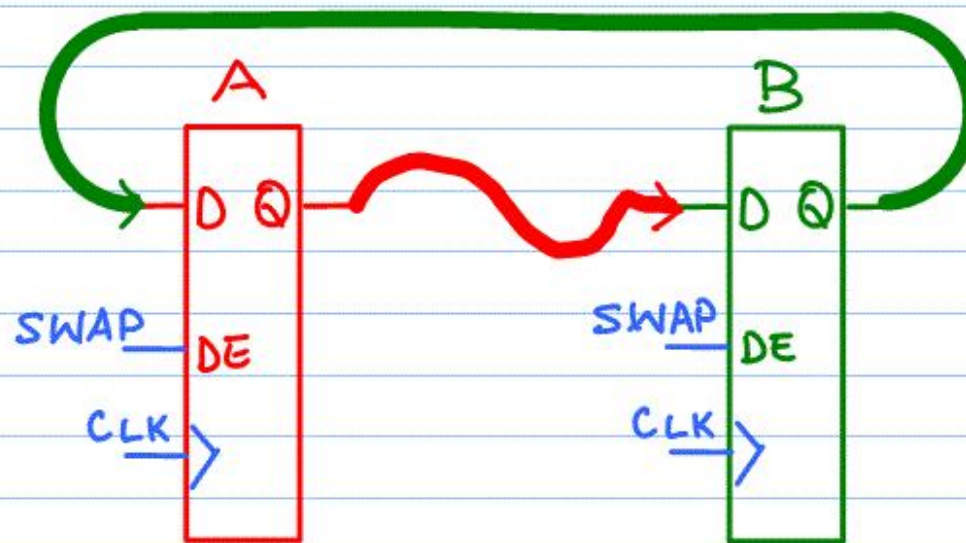
Software people use temporary variables.

Is swaping the contents of **two latches**, X and Y same as swapping the contents of **two registers** A and B?

No *You need temporary latch/reg.*



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



```

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

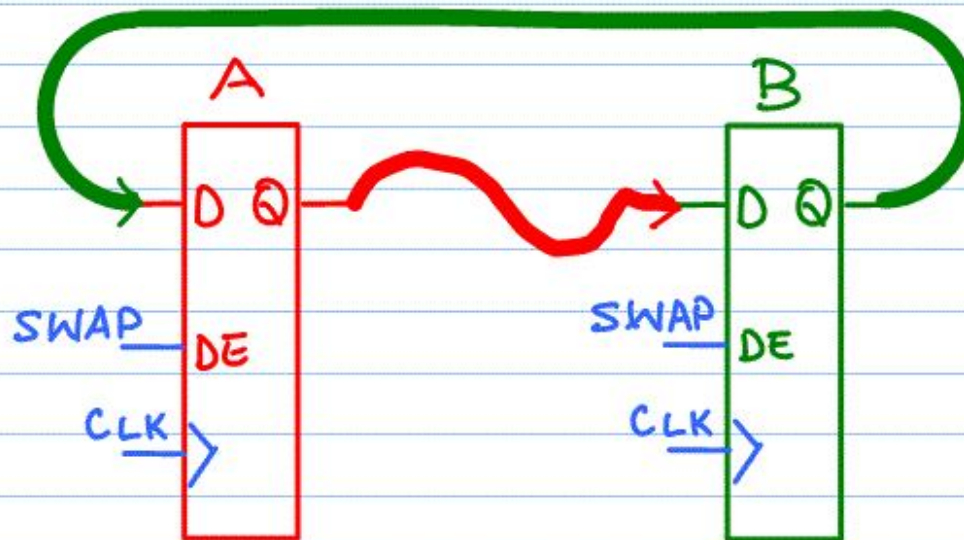
```

```

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

```

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



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

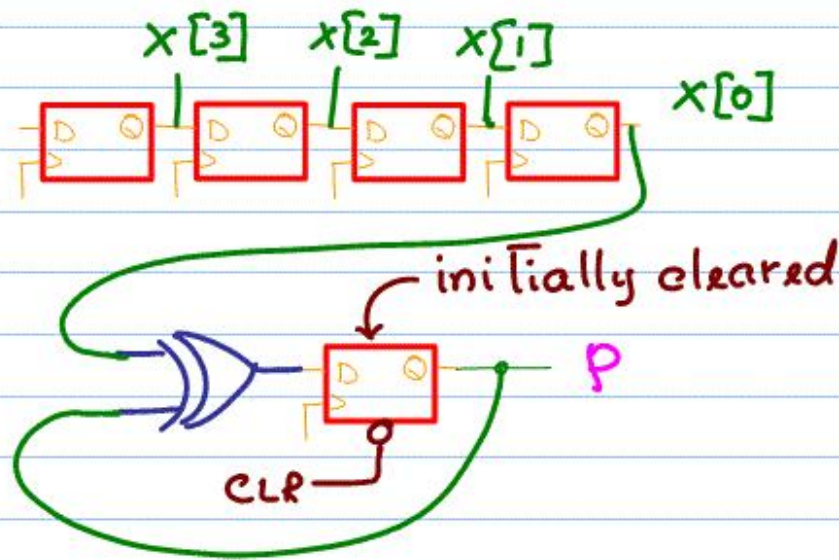
```
SWAP_STATE:
begin
  if (SWAP)
    begin
      A = B;
      B = A;
    end
end
```

Of course Non-blocking!
No choice at all!

Non-blocking assignments to all physical registers in a clocked always block to avoid any race condition in simulation.

Please remember this basic rule and the intent behind it.

Note: **for** loop can not be used for hardware iterations.



In order to design the above Serial Parity Generator you need a control unit (CU) besides the above data path unit (DPU).