Lecture: Verilog#1
File: EE101_Verilog_1.pdf

EE101

By Gandhi Puvvada

Concurrency in Verilog HDL
The three concurrent constructs
Inferring connections by data-flow graph

Operators in Verilog

Structural vs. behavioral coding

For detailed coverage of Verilog essentials, please refer to the 6 lectures used in EE354L and EE457.

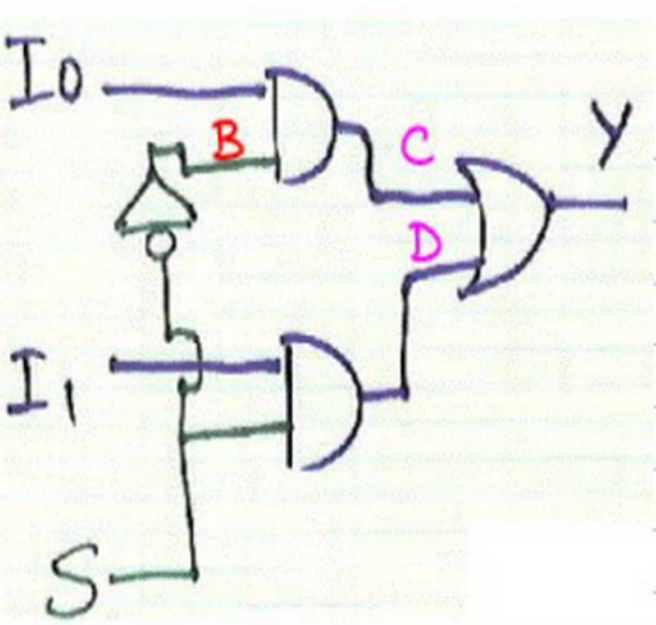
Hardware is Concurrent.

An **HDL** should be able to be able to **represent** and **preserve** this **concurrent** nature of Hardware.

The three concurrent constructs in Verilog are:

1. **assign (continuous) statements**
2. **always procedural blocks**
3. **instantiations of lower level modules**

assign statement example



Operators

~	NOT	
&	AND	
|	OR	

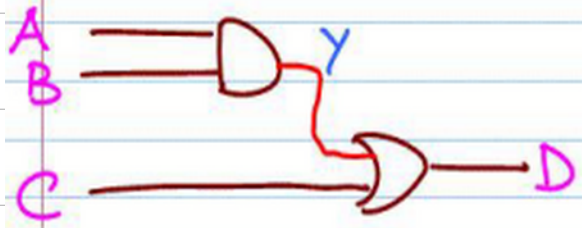
```
assign  B = ~S;
assign  C = I0 & B;
assign  D = I1 & S;
assign  Y = C | D;
```

4

Concurrent Statements

Order does not matter.

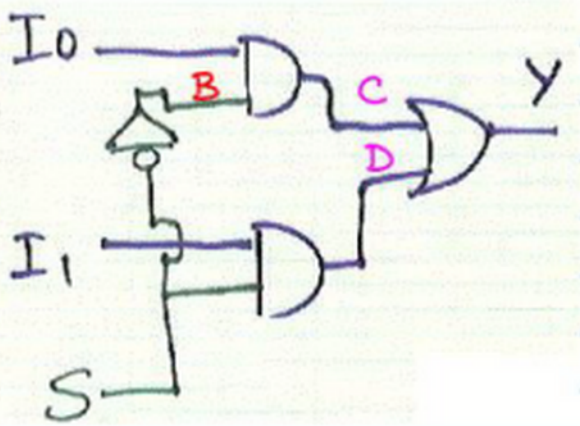
Data flow model




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

Order does not matter as the statements are CONCURRENT.

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



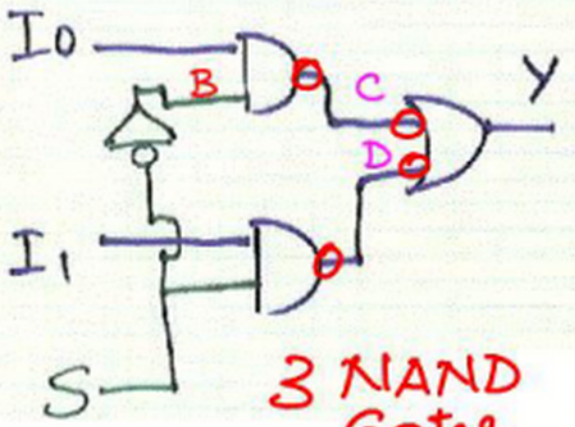
Data flow modeling

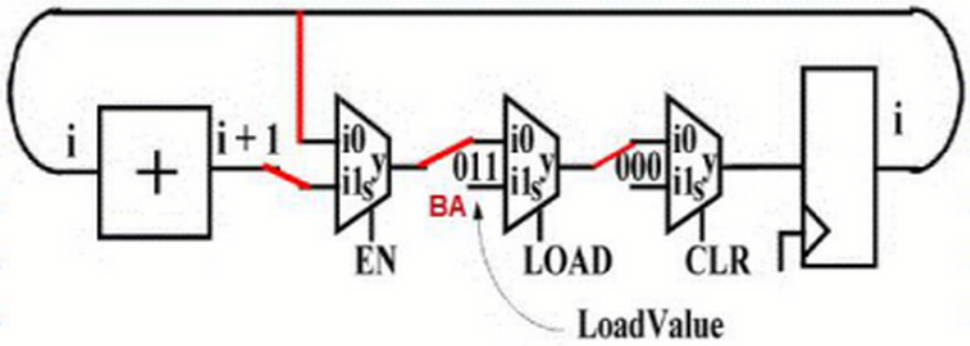
```
assign  B = ~S;
assign  C = I0 & B;
assign  D = I1 & S;
assign  Y = C | D;
```

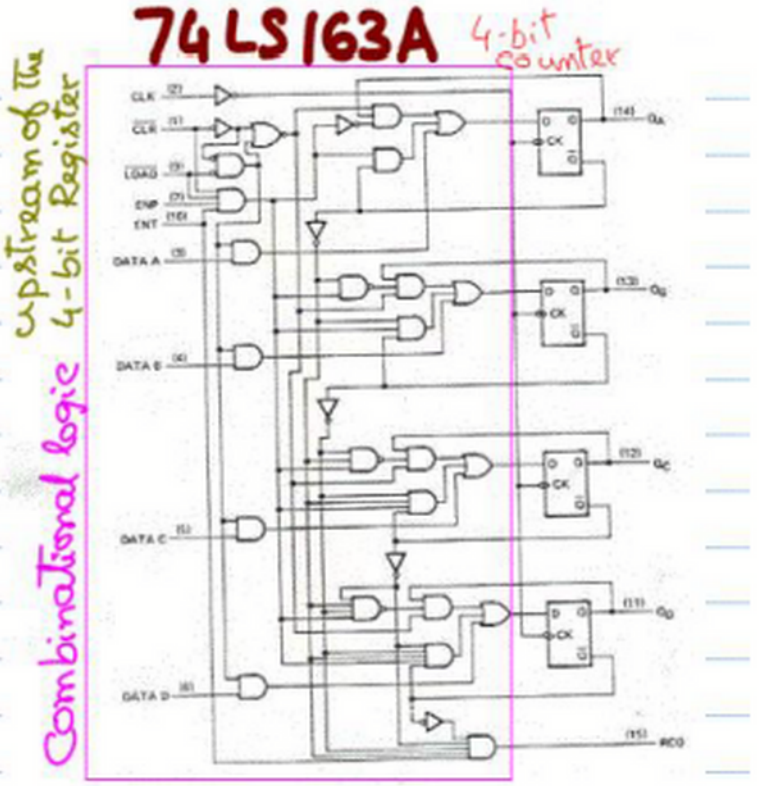
behavioral modeling

```
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
```

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.

Some basic Verilog syntax

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 comment
   is like in "C" */

   /* - - - .
   - - - -
   - - - -
   - - - */

3. Compiler directives ⇒ example `define `timescale

4. System calls: $display $fopen

`Compiler Directives

Directives are preceded by the accent grave character (`) and are **not** terminated by a semicolon.

`timescale

page 21

page 18

```
`define width 7
reg [`width:0] vec8;
reg [`width:0] mema [1:1024];
```

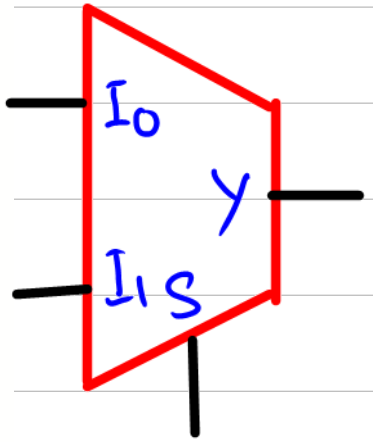
```
`timescale 1ns/100ps
```

```
`timescale unit / precision
```

```
1  module mux_1_bit_wide (I0, I1, S, Y);
2
3  input I0, I1, S;
4  output Y;
5  reg Y;
6
7  always @(*)
8    begin
9      if (S)
10         Y = I1;
11     else
12         Y = I0;
13   end
14
15 endmodule //mux_1_bit_wide
```

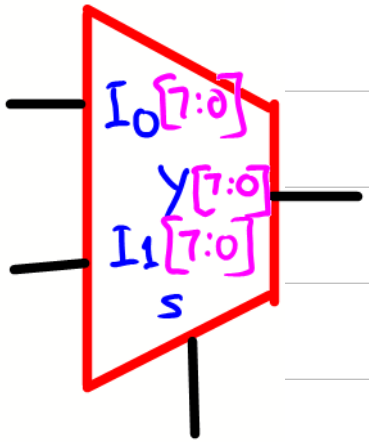
all inputs

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



1-bit wide 2-to-1 mux

```
1  module mux_8_bit_wide (I0, I1, S, Y);
2
3  input [7:0] I0, I1;
4  input S;
5  output [7:0] Y;
6  reg [7:0] Y;
7
8  always @(*)
9    begin
10     if (S)
11         Y = I1;
12     else
13         Y = I0;
14   end
15
16 endmodule //mux_8_bit_wide
```

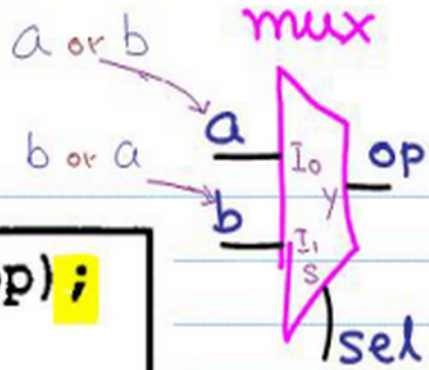


8-bit wide 2-to-1 mux

case statement for 4-to-1 mux

```
1  module mux_1_bit_wide_4_to_1 (I0, I1, I2, I3, S, Y);
2
3  input I0, I1, I2, I3;
4  input [1:0]S;
5  output Y;
6  reg Y;
7
8  always @(*)
9    begin
10     case (S)
11         2'b00: Y = I0;
12         2'b01: Y = I1;
13         2'b10: Y = I2;
14         2'b11: Y = I3;
15     endcase
16   end
17
18 endmodule //mux_1_bit_wide_4_to_1
```

Esperan pages 94, 95, ~~96~~ module




```
module mux (a, b, sel, op);
   input [7:0] a, b;
   input sel;
   output [7:0] op;
   reg [7:0] op;
   wire sel; wire [7:0] a,b;
   always @(a, or b, or sel)
     op = sel ? a : b;

endmodule
```

One of the two common data types
wire and reg

wire is the default data type.

unnecessary to declare

Verilog 2001 preferred notation


```
<condition>?<true_expr>:<false_expr>;
```

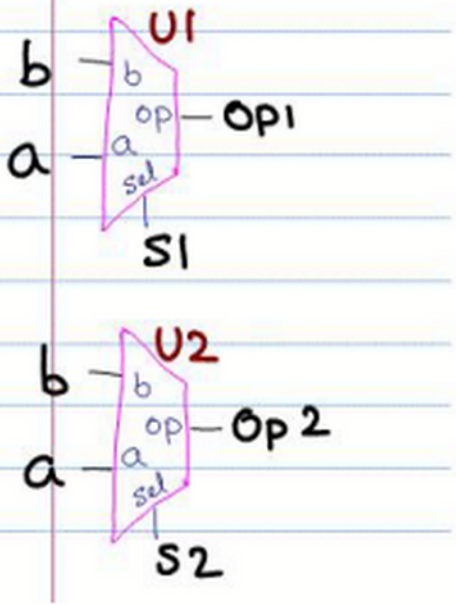
page 95

Module Instantiation Named and Positional Port Connection

```
mux U1(.a(a), .b(b), .sel(s1), .op(op1));
```

.<port name>(<variable_name>)

• pin (system_label)




```
module muxes (a, b, s1, s2, op1, op2);
  input [7:0] a, b;
  input s1, s2;
  output [7:0] op1, op2;

  mux U1 (.a(a), .b(b),
          .sel(s1), .op(op1));
  mux U2 (a, b, s2, op2);

endmodule
```

named association is the preferred practice.

Data types: Language designer's notion of NET data types and REGISTER data types

NET

wire — *constantly driven*

Example: LHS of an assign (continuous) statement

REGISTER

reg — *occasionally driven*

Example: LHS of a sequential statement in an always procedural block.

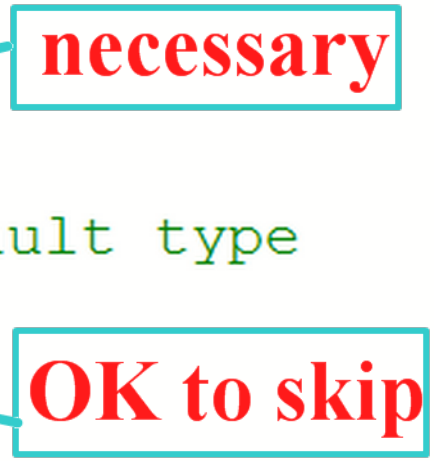
Note: RHS of any statement can contain both NET type and REGISTER type data items.

reg

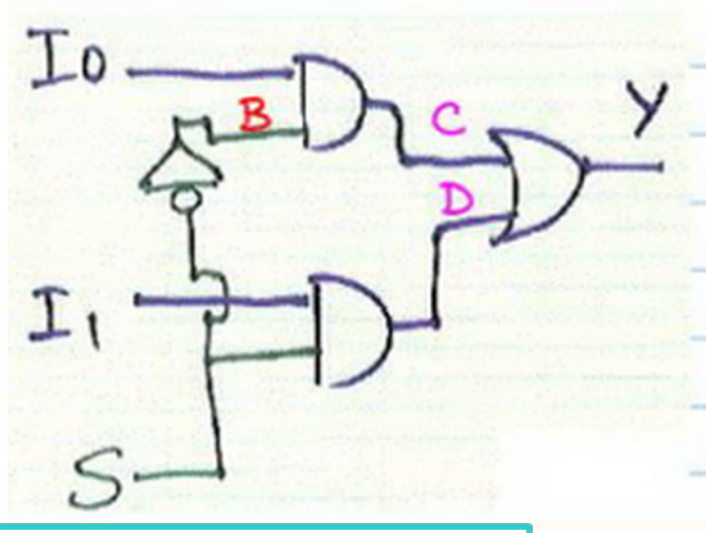
does not mean

a physical register

```
1  module mux_1_bit_wide_gate_level_assign (I0, I1, S, Y);
2
3  input I0, I1, S;
4  output Y;
5  wire B, C, D;
6  wire Y; // wire is the default type
7
8  assign B = ~S;
9  assign C = I0 & B;
10 assign D = I1 & S;
11 assign Y = C | D;
12
13 endmodule //mux_1_bit_wide_gate_level_assign
```



```
1  module mux_1_bit_wide_gate_level_always (I0, I1, S, Y);
2
3  input I0, I1, S;
4  output Y;
5  reg B, C, D;
6  reg Y;
7
8  always @(*)
9    begin
10     B = ~S;
11     C = I0 & B;
12     D = I1 & S;
13     Y = C | D;
14   end
15
16 endmodule //mux_1_bit_wide_gate_level_always
```



B, C, D, and Y are not physical registers, but unfortunately, they have to be declared as reg

Order matters in an always block!

Inside an "always" block, execution is sequential!

We read (and understand) combinational logic
by going from inputs to outputs in a methodic way (in a sequential fashion) though all pieces of the logic are concurrent.

The "always" procedural block allows this.

Inside the always block execution is sequential.
So we need to produce a variable before consuming it.

Since it is difficult for humans to form the data-flow graph (schematic) in their mind to understand the code, it is recommended to reduce the number of concurrent items in the code by using always blocks to gather all related pieces into one always block!
So, avoid assign statements for most of your Verilog coding!

page 44, 45

always

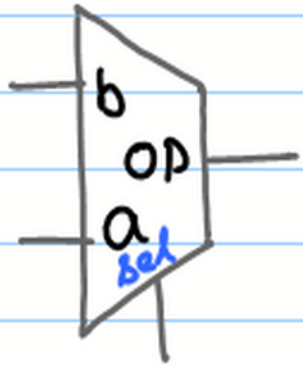
```
always @(sel, a, b)
  if (sel)
    op = a;
  else
    op = b;
```

always @ ( ) no ;

begin

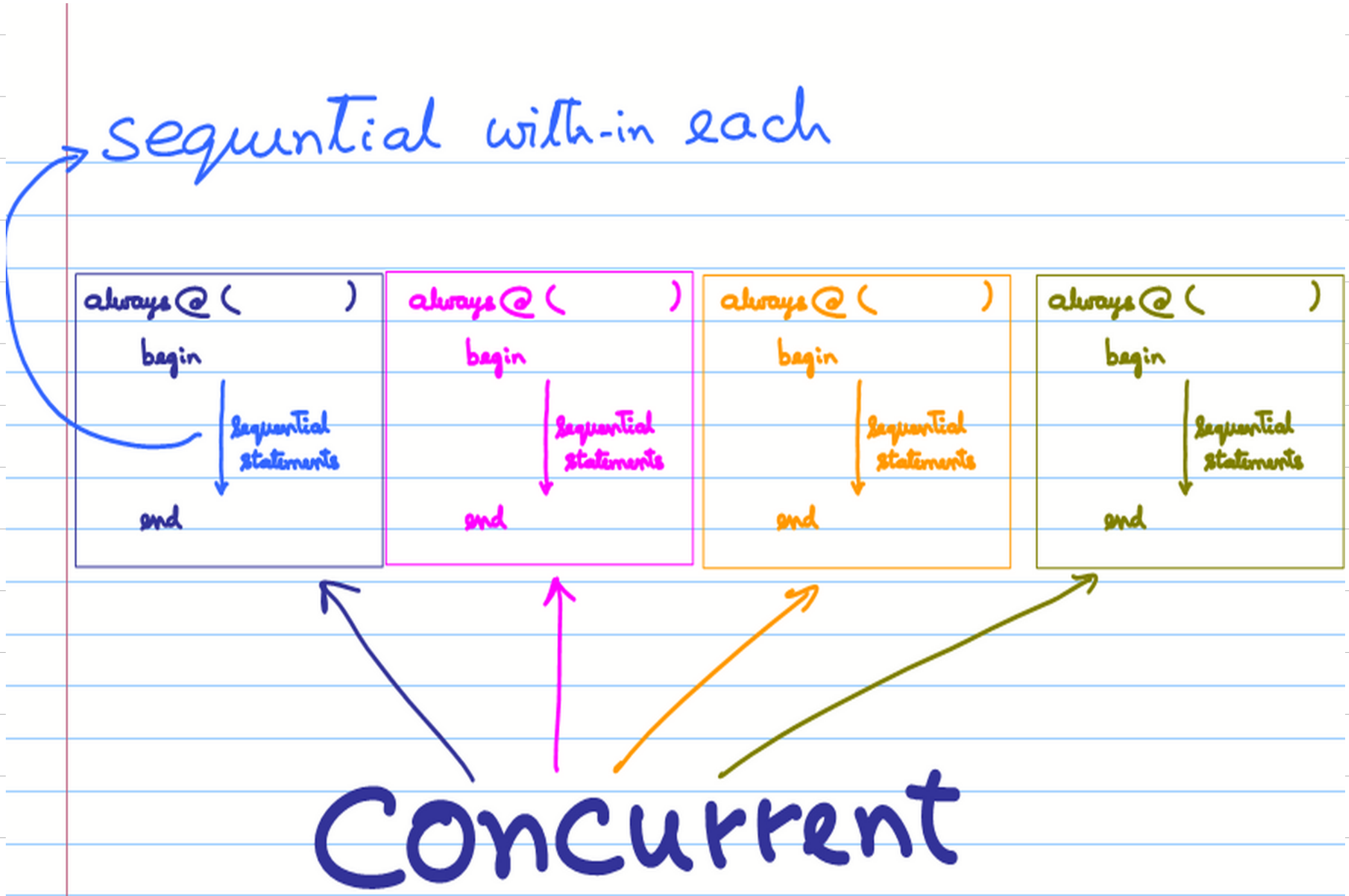
Sequential statements

end



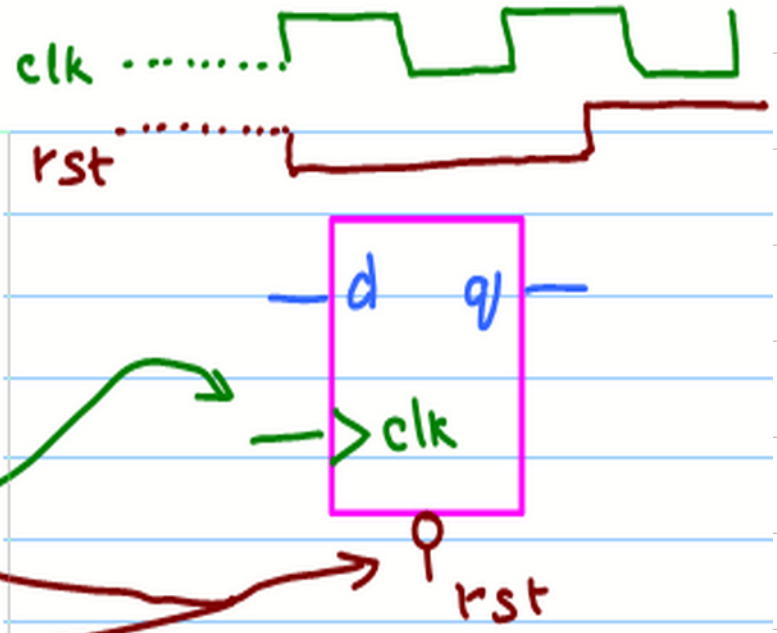
page 51

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
always @ ( )
begin
sequential statements
end
always @ ( )
begin
sequential statements
end
always @ ( )
begin
sequential statements
end
always @ ( )
begin
sequential statements
end
Concurrent

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

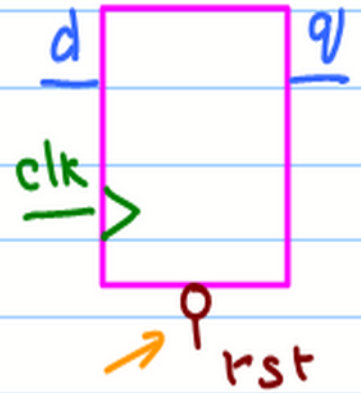
page 113, 97

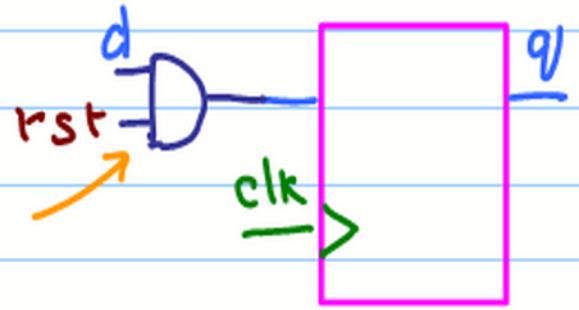
page 12, 13

remove

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

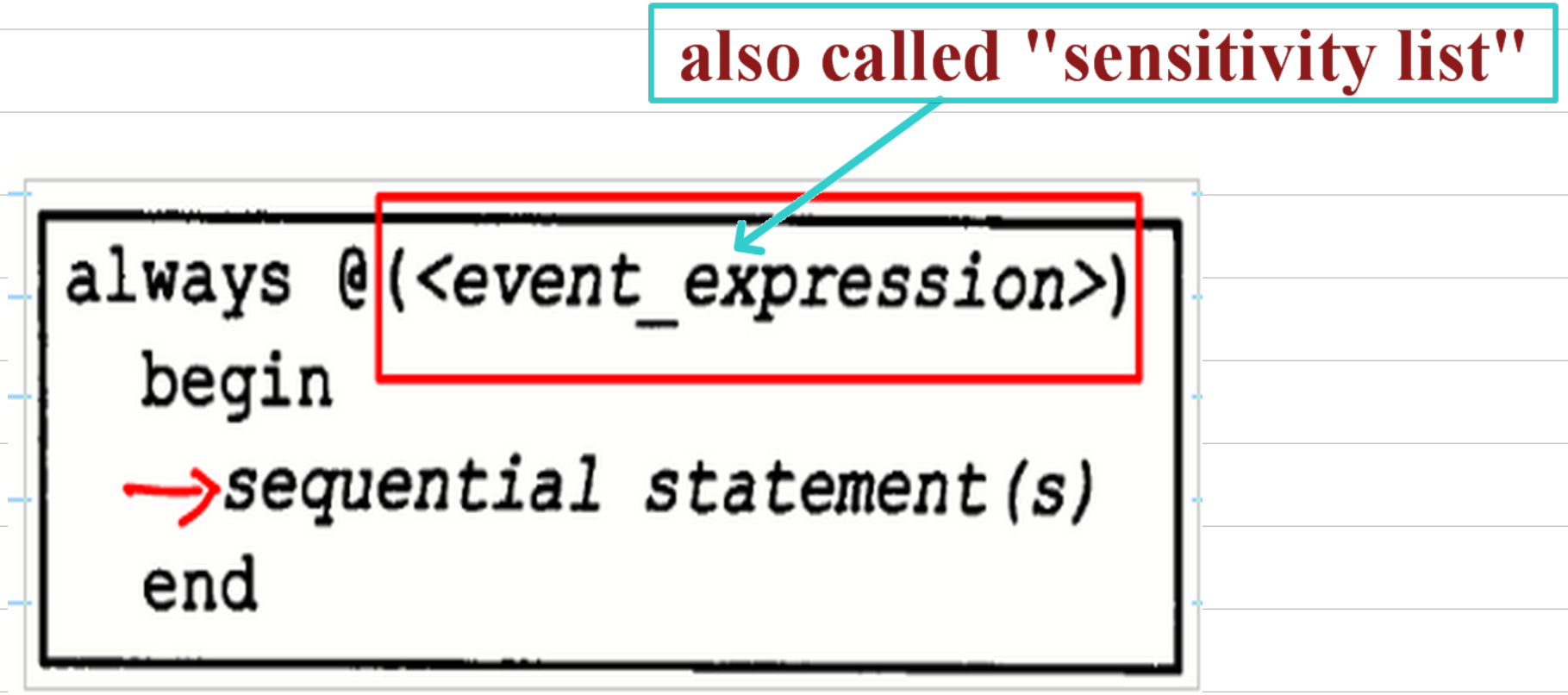
```
always @(posedge clk)
  if (rst == 0)
    q <= 0;
  else
    q <= d;
```





asynchronous reset

synchronous reset



For an "always" block coding combinational logic, the sensitivity list must be complete.

But notice that the "D" input to the Flip-flop is not in the sensitivity list of the always block coding the D-FF.

```
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

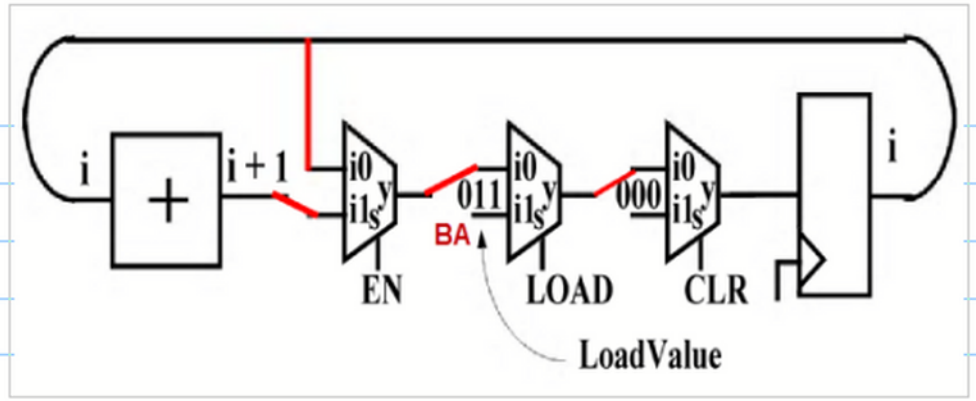
Sensitivity list : If inputs change, we want to re-evaluate its outputs


```
assign C = A & B;
```

implicit sensitivity list items

```
always @ (A, B)
    begin
        C = A & B;
    end
```

explicit sensitivity list items




```
always @(posedge clk)

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

Synchronous counter with a synchronous clear.
So the sensitivity list has just

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

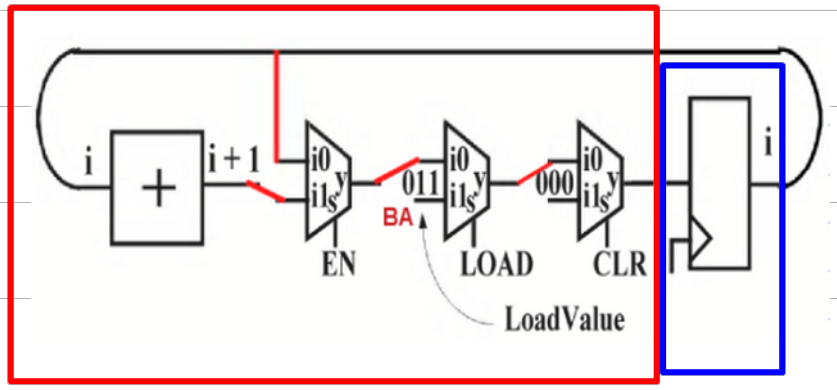
We should not include, clr, load, en in the sensitivity list.

Only clock and any asynchronous reset or preset go into the sensitivity list for a clocked always block

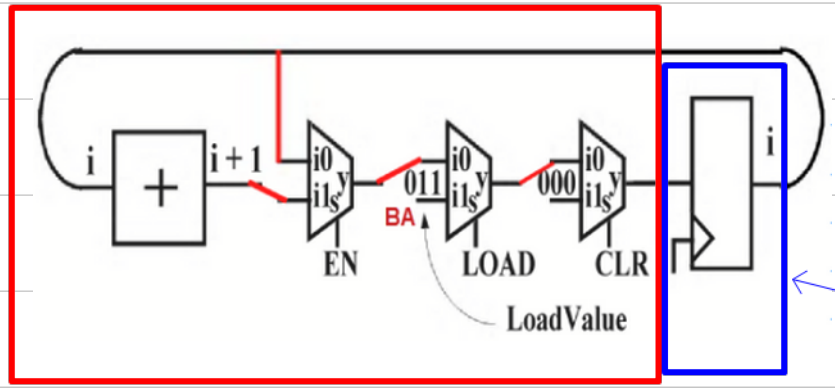
Combinational logic can be coded in two ways:
-- using assign statements (one or more)
-- using always blocks (one or more)

Flip-flops or registers need to be coded in only one way, using a clocked always block:
example:
always @(posedge clk)
always @(posedge clk, negedge rst_bar)

However, if your design has both combinational logic and registers, try to code them all in one clocked always block as that improves readability!!! <= Very important!



Technician's way of coding: They separate combinational logic and registers. It reduces readability :(




```
always @(posedge clk)
  begin
     Q <= Q_next;
  end
```

```
always @(*)
  begin
     if (clr)
          Q_next = 0;
     else if (load)
          Q_next = BA;
          // BA = Branch address
          // = some load value
     else if (en)
          Q_next = Q_next + 1;
     else
          Q_next = Q;
  end
```

Very rarely we code a combinationl logic only.
Most desigs have both combinational logic and registers.
So do you want to separate the combinational logic and registers? No that is technician's way of coding.

The important point is that the "upstream combinational logic" (upstream to the register being coded in the clocked always block) can be coded in the same clocked always block, provided it is coded with blocking assignments (=) upstream of the final register assignment using non-blocking assignment (<=).

You will be taught this aspect in a few examples later.