CSE-308: Digital System Design

Lecture 2

Description Styles in Verilog

- Two different styles of description:
  1. Data flow
     - Continuous assignment
  2. Behavioral
     - Procedural assignment
       - Blocking =
       - Non-blocking <=

assign

Data-flow Style: Continuous Assignment

- Identified by the keyword "assign".

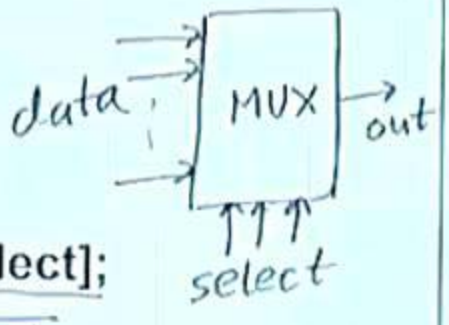
```
assign   a = b & c;
assign   f[2] = c[0];
```

- Forms a static binding between
  - The 'net' being assigned on the LHS,
  - The expression on the RHS.
- The assignment is continuously active.
- Almost exclusively used to model combinational logic.

- A Verilog module can contain any number of continuous assignment statements.
- For an "assign" statement,
  - The expression on RHS may contain both "register" or "net" type variables.
  - The LHS must be of "net" type, typically a "wire".
- Several examples of "assign" illustrated already.

assign b = <expr> ;

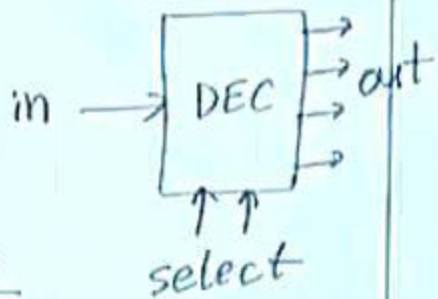
```
module   generate_mux (data, select, out);
   input  [0:7]  data;
   input  [0:2]  select;
   output  out;

   assign  out  =  data [ select];
endmodule
```




```
out = data[2];
```

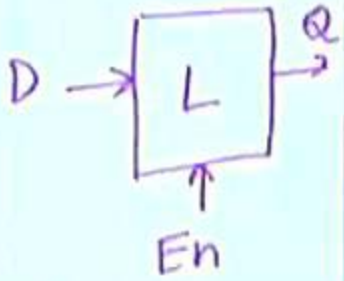
Non-constant index in expression on RHS generates a MUX

```
module generate_decoder (out, in, select);
   input  in;
   input [0:1]  select;
   output  [0:3]  out;

   assign  out [ select] = in;
endmodule
```



Non-constant index in expression on LHS generates a decoder

```
module  level_sensitive_latch  (D, Q, En);
   input  D, En;
   output  Q;

   assign  Q  =  en  ?  D  :  Q;
endmodule
```

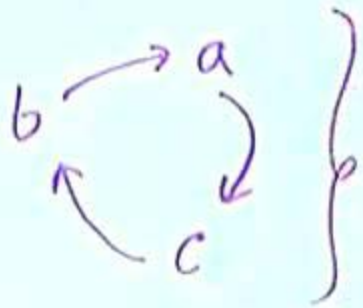


Using "assign" to describe sequential logic

assign a = b ·---

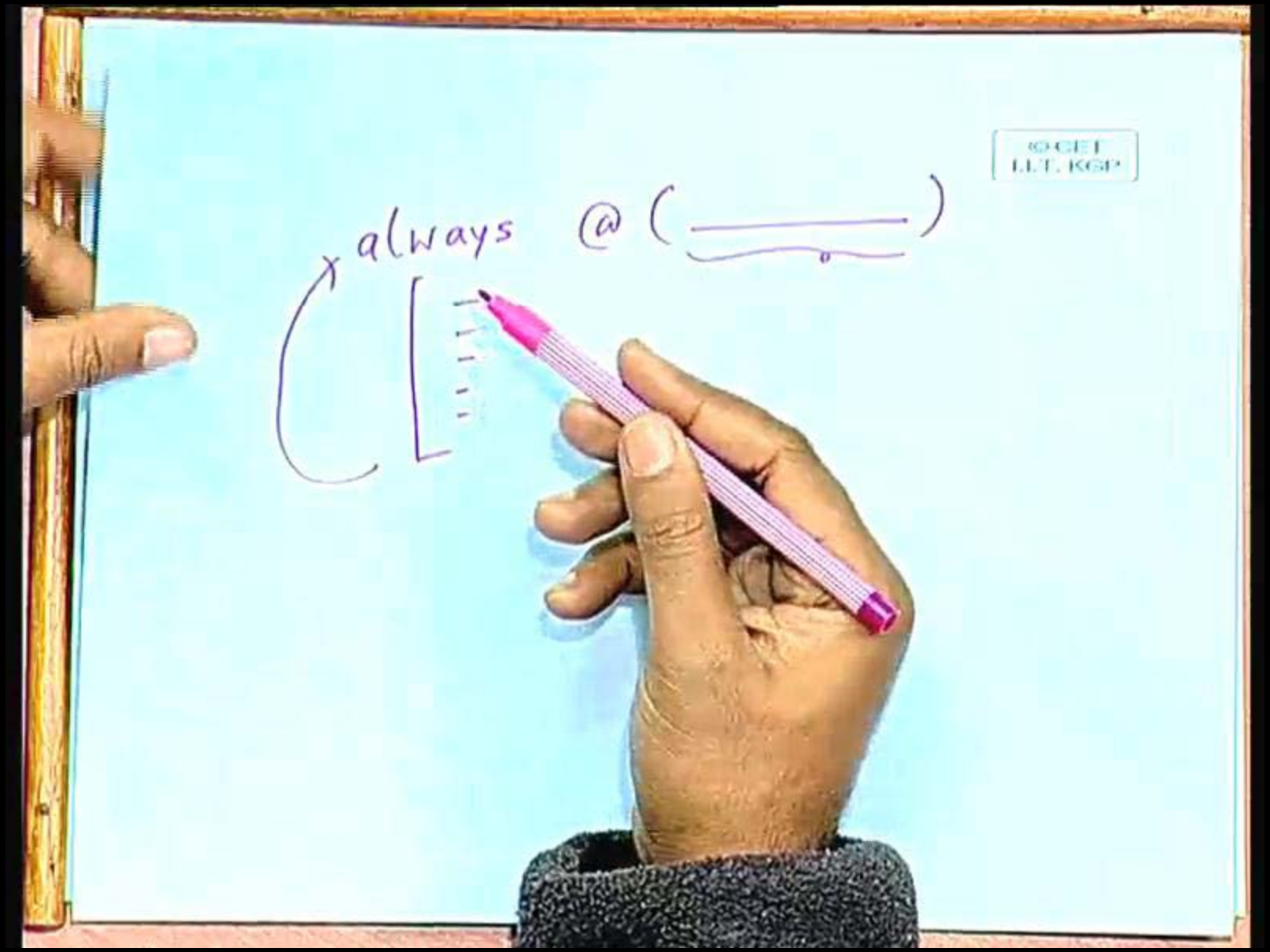
assign b = c ·--·

assign c = a ·—·—



Behavioral Style: Procedural Assignment

- The procedural block defines
  - A region of code containing *sequential* statements.
  - The statements execute in the order they are written.
- Two types of procedural blocks in Verilog
  - The "always" block
    - A continuous loop that never terminates.
  - The "initial" block
    - Executed once at the beginning of simulation (used in Test-benches).


©GET
LET KGP
always @ (

Behavioral Style: Procedural Assignment

- The procedural block defines
  - A region of code containing *sequential* statements.
  - The statements execute in the order they are written.
- Two types of procedural blocks in Verilog
  - The "always" block
    - A continuous loop that never terminates.
  - The "initial" block
    - Executed once at the beginning of simulation (used in Test-benches).

- A module can contain any number of "always" blocks, all of which execute concurrently.
- Basic syntax of "always" block:

```
always  @ (event_expression)
begin
    statement;
    statement;
end
```

Sequential statements

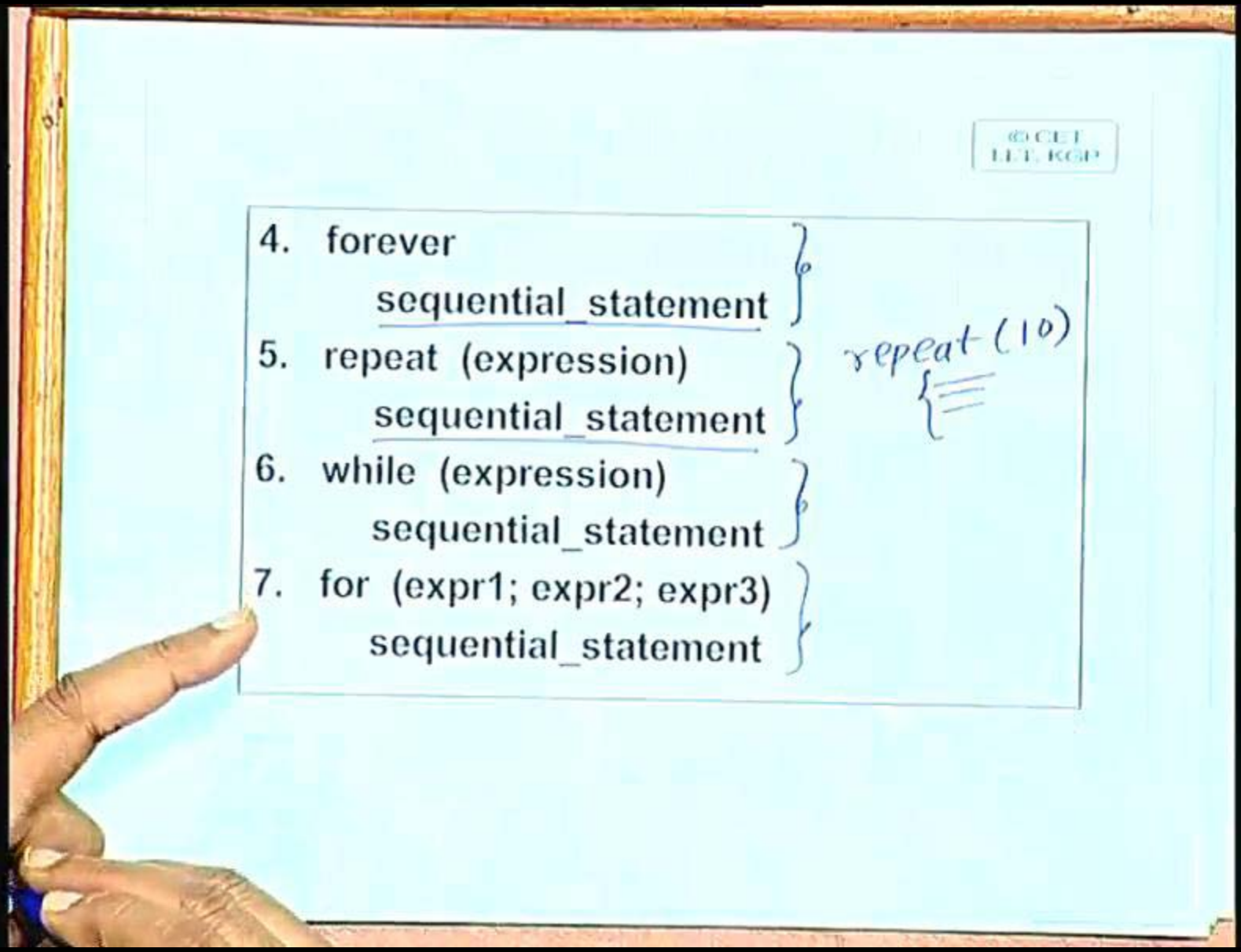
- The @(event_expression) is required for both combinational and sequential logic descriptions.

- Only "reg" type variables can be assigned within an "always" block. Why??
  - The sequential "always" block executes only when the event expression triggers.
  - At other times the block is doing nothing.
  - An object being assigned to must therefore remember the last value assigned (not continuously driven).
  - So, only "reg" type variables can be assigned within the "always" block.
  - Of course, any kind of variable may appear in the event expression (reg, wire, etc.).

Sequential Statements in Verilog

1. begin
   sequential_statements
   end
2. if (expression)
   sequential_statement
   [else
   sequential_statement]
3. case (expression)
   expr: sequential_statement
   .....
   default: sequential_statement
   endcase

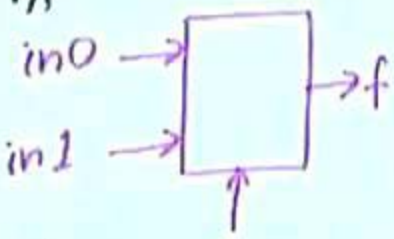
begin...end not required if there is only 1 stmt.

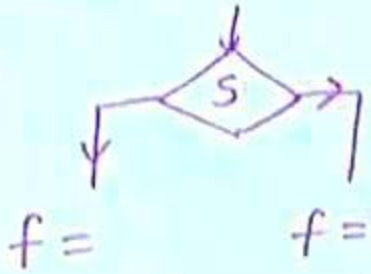

4. forever
sequential_statement
5. repeat (expression)
sequential_statement
6. while (expression)
sequential_statement
7. for (expr1; expr2; expr3)
sequential_statement
repeat (10)

8. # (time_value)
   - Makes a block suspend for "time_value" time units.
9. @ (event_expression)
   - Makes a block suspend until event_expression triggers.

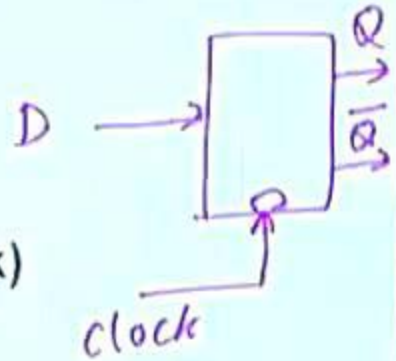
#20 

```
// A combinational logic example

module mux21 (in1, in0, s, f);
   input in1, in0, s;
   output f;
   reg f;

   always @ (in1 or in0 or s)
      if (s)
         f = in1;
      else
         f = in0;
endmodule
```



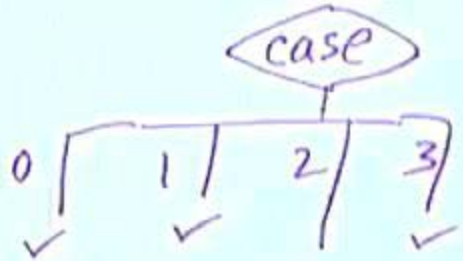


```
// A sequential logic example

module dff_negedge (D, clock, Q, Qbar);
   input D, clock;
   output Q, Qbar;
   reg Q, Qbar;

   always @ (negedge clock)
      begin
         Q = D;
         Qbar = ~D;
      end
endmodule
```



// Another sequential logic example


```
module incomp_state_spec (curr_state, flag);
   input  [0:1] curr_state;
   output [0:1] flag;
   reg    [0:1] flag;

   always @ (curr_state)
      case (curr_state)
         0, 1 : flag = 2;
         3    : flag = 0;
      endcase
endmodule
```

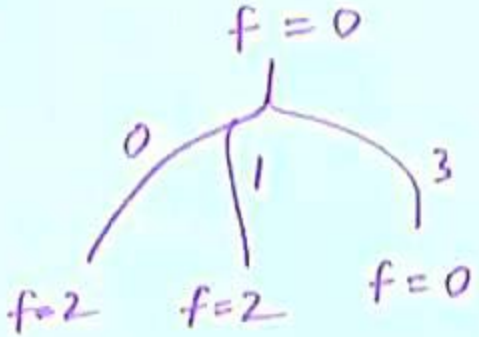


The variable 'flag' is not assigned a value in all the branches of case.

→ Latch is *inferred*

// A small change made

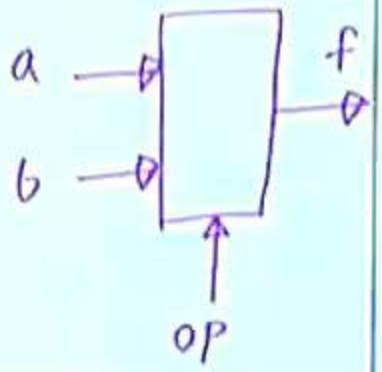

```
module incomp_state_spec (curr_state, flag);
   input  [0:1] curr_state;
   output [0:1] flag;
   reg    [0:1] flag;

   always @ (curr_state)
   begin
      flag = 0;
      case (curr_state)
         0, 1 : flag = 2; break
         3    : flag = 0; break
      endcase
   end
endmodule
```



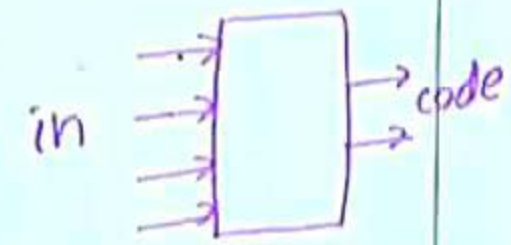
'flag' defined for all values of curr_state.

➔ Latch is *avoided*

```
module  ALU_4bit  (f, a, b, op);

   input   [1:0]  op;        input [3:0]  a, b;
   output [3:0]  f;          reg    [3:0]  f;

   parameter  ADD=2'b00, SUB=2'b01,
                    MUL=2'b10, DIV=2'b11;

   always  @ (a or b or op)
      case (op)
         ADD : f = a + b;
         SUB : f = a – b;
         MUL : f = a * b;
         DIV   : f = a / b;
      endcase
endmodule
```



```
module priority_encoder (in, code);
   input [0:3] in;
   output [0:1] code;
   reg [0:1] code;
   always @ (in)
      case (1'b1)
         input[0] : code = 2'b00;
         input[1] : code = 2'b01;
         input[2] : code = 2'd10;
         input[3] : code = 2'b11;
      endcase
endmodule
```



0
0
0
0