Lecture - 9

☆ Introduction to Procedural Assignment

* Behavioral style → Procedural assignment
* Two kinds of procedural blocks are:
  1) "initial" block
     - executed once at the beginning of simulation
     - used only in test benches, cannot be used in synthesis
  2) "always" block
     - a continuous loop that never terminates
* A procedural block defines:
  - A region of code containing sequential statements
  - The statements execute in the order they are written.

* Shortcuts in declaration

```
output [7:0] data;
reg    [7:0] data;      ⟶  output reg [7:0] data;

reg clock;
initial clock=0;        ⟶  reg clock=0;
```

* "always" Block
  - syntax:

always Block

```
always
  begin
    statement 1
    statement 2      } Executed Sequentially
    :
    statement n
  end
```

repeat

Scanned with CamScanner

* "initial" block

initial block

```
initial
  begin
    statement 1
    statement 2
    :
    statement n
  end
```

executed sequentially only ONCE

* Example:

```
module generating_clock;
  output reg clk;

  initial
    clk = 1'b0; // initialized to 0 at time 0

  always
    #5 clk = ~clk; // Toggle every 5 time units

endmodule
```

Note: 1) A module can contain any number of "always" or "initial" blocks

2) All "always" & "initial" block start at simulation time 0.

3) Absence of begin ..... end only executes a single statement in that block. (included)

4) Only reg type variable can be assigned within an initial / always block. (i.e. LHS of expression should be reg & reg type)

Reason: always block executes only on event expression so object must hold value even when no event exp. is present.

• Basic syntax of "always" block

```
always @(event_expression)
begin
    seq_st_1;
    seq_st_2;
    :
    seq_st_n;
end
```

* Types of Sequential Statements

* begin---.end

- It is the basic block which combines a no. of sequential statements into one composite statement.
- If no. of statements = 1, then begin---end is not required.

* If.... else

①
```
if (expression)
    begin---
    :
    end
```

②
```
if (expression)
    begin.... end
else
    begin---.end
```

③ if (expression_1)
begin .... end
else if (expression_2)
begin ---- end
else if (expression_3)
begin ---- end
else
begin ---- end

* case (expression)
expr1: seq_statement1; → can also be begin...end
expr2: seq_statement2;
:
exprn: seq_statement n;
default: default_statement;
endcase

* Two variations of case:
1) "casez"
All 'z' values in case alternatives or the case expression are treated as don't care.

2) "casex"
All 'z' & 'x' values in case alternatives or the case expression are treated as don't care.

* Example:

```
reg [3:0] state;
integer next_state;
casex (state)
    4'b1xxx : next_state=0;   // line 1
    4'bx1xx : next_state =1;  // line 2
    4'bxx1x : next_state =2;  // line 3
    4'bxxx1 : next_state=3;   // line 4
    default : next_state = 4; // line 5
endcase
```

- If state = 4'b1oxx then "line 1" will be executed
- If state = 4'b01zx then "line 2" will be executed