

## Lecture - 9

### \* Introduction to Procedural Assignment

\* Behavioral style  $\rightarrow$  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.

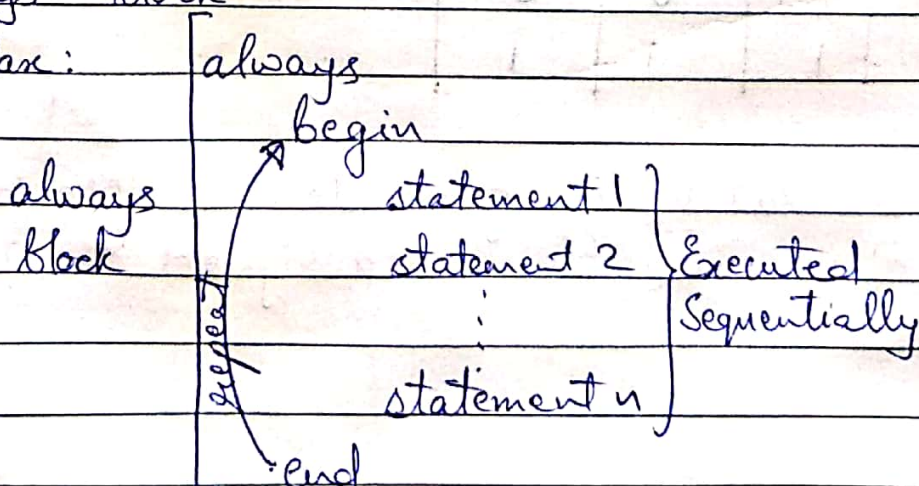
\* Shortcuts in declaration

$\left. \begin{array}{l} \text{output } [7:0] \text{ data;} \\ \text{reg } [7:0] \text{ data;} \end{array} \right\} \rightarrow \text{output reg } [7:0] \text{ data;} ;$

$\left. \begin{array}{l} \text{reg clock;} \\ \text{initial clock} = 0; \end{array} \right\} \rightarrow \text{reg clock} = 0 ;$

\* "always" block

• syntax:





### \* "initial" block

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

initial block

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. (includes)

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-statementn;  
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.

x Example:

```
reg [3:0] state;  
integer next_state;  
case (state)
```

```
4'b1xxx: next_state = 0; // line 1
```

```
4'b0xxx: 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'b10xx then "line 1" will be executed

• If state = 4'b01zx then "line 2" will be executed