Behavioral Modeling

Dr.K.Sivasankaran
Associate Professor,
School of Electronics
Engineering

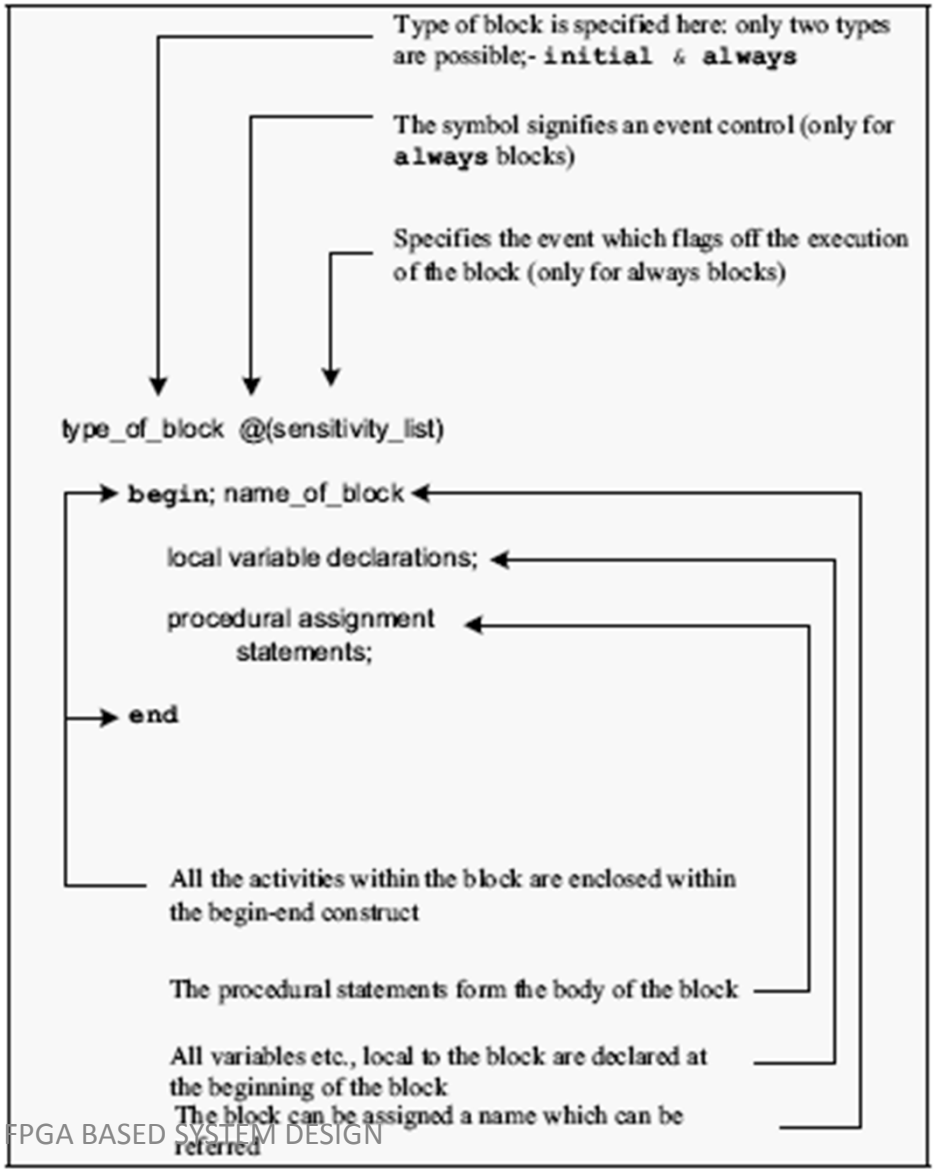
Behavioral Level Modeling

- This is the highest level of abstraction provided by Verilog HDL.
- A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details.
- Designing at this level is very similar to C programming.

Behavioral modeling

- There are two structured procedures in Verilog:
  - **initial**
  - **always**
- Concurrent execution is observed in between these procedures.
- Sequential / concurrent execution can be realized within these procedures.
- Only registers can be assigned in these procedures.
- The assignments in these procedures are called “procedural assignments”.

Typical Procedural Block



initial statement

- Starts execution at ‘0’ simulation time and executes only once during the entire simulation.
- Multiple statements in initial block can be grouped with (**begin** & **end**) or (**fork** & **join**) keywords.
- These blocks are not synthesizable.

initial statement

- initial blocks cannot be nested.
- Each initial block represent a separate and independent activity.
- initial blocks are used in generating test benches.

initial block structures

```
initial
xor_out = in1 ^ in2;
```

```
initial
begin
  and_out = a_in & b_in;
end
```

```
initial
begin
   enable = 1'b0;
   rst   = 1'b0;
   #100  rst = 1'b1;
   #20 enable = 1'b1;
end
```

```
initial
begin
   clk = 1'b0;
   reset   = 1'b0;
 initial
 begin
       #100  reset = 1'b1;
       #20 clk = 1'b1;
   end
   end
```

Multiple Initial Blocks

- A module can have as many **initial blocks as desired.**
- **All of them are** activated at the start of simulation.
- The time delays specified in one **initial** block are exclusive of those in any other block.

Example

```
module nil1;
initial
reg a, b;
begin
a = 1'b0;
b = 1'b0;
$display ($time,"display: a = %b, b = %b", a, b);
#2 a = 1'b1;
#3 b = 1'b1;
#1 a = 1'b0;
end
initial
#100$stop;
initial $monitor ($time, “monitor: a =
%b, b = %b", a, b);
initial
begin
#2 b = 1'b1;
end
endmodule
```

always statement

- Starts execution at ‘0’ simulation time and is active all through out the entire simulation.
- Multiple statements inside always block can be grouped with (**begin** & **end**) or (**fork** & **join**) keywords.
- Execution of always blocks is controlled by using the timing control.
- always blocks cannot be nested.

always statement

- An always block without any sensitivity control will create an infinite loop and execute forever.
- Each always block represent a separate and independent activity.
- These blocks can synthesize to different hardware depending on their usage.
- always block with timing control are synthesizable.

always block structures

```
always
  xor_out = in1 ^ in2;
```

```
always @(a_in or b_in)
begin
   and_out = a_in & b_in;
end
```

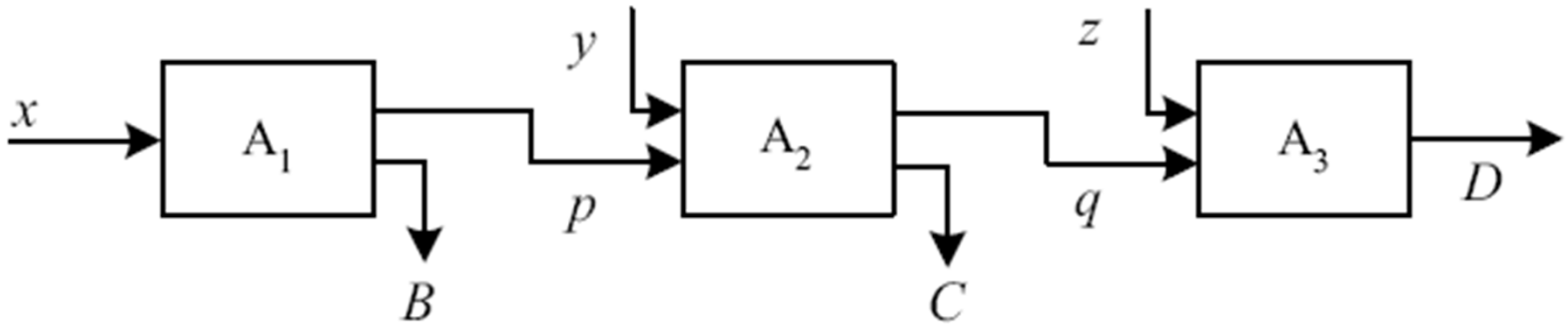
```
always @(posedge reset)
  begin
   if (reset == 1'b1)
   q_out =  1'b0;
   else
   q_out =  d_in;
  end
```

```
always
begin
    cnt = 1'b0;
    reset   = 1'b0;
    always @(posedge clk)
    begin
        #100  cnt = 1'b1;
        #20 enable = 1'b1;
    end
end
```

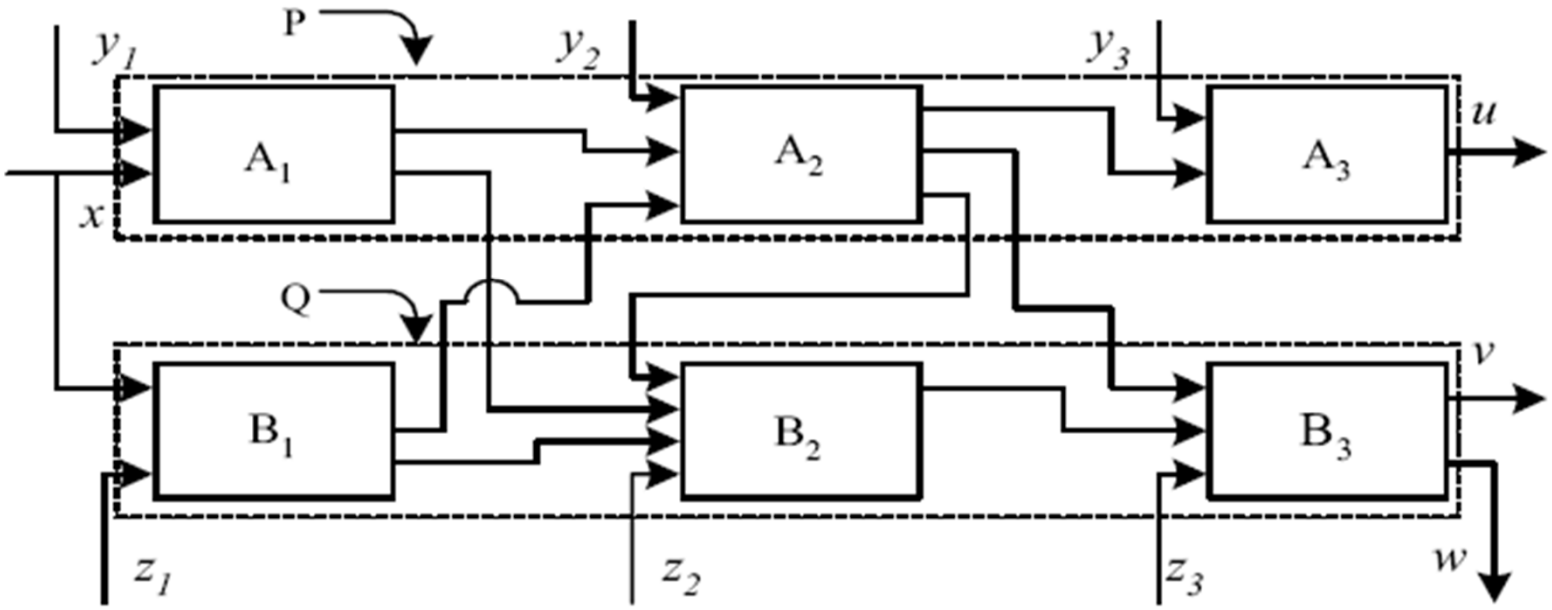
Multiple always Blocks

- All the activities within an always block are scheduled for sequential execution.
- The activities can be of a combinational nature, a clocked sequential nature, or a combination of these two.
- Basically, any circuit block whose end-to-end operation can be described as a continuous sequence can be described within an **always** block.

A module where execution proceeds through three blocks sequentially



A module where execution proceeds concurrently through two groups of blocks



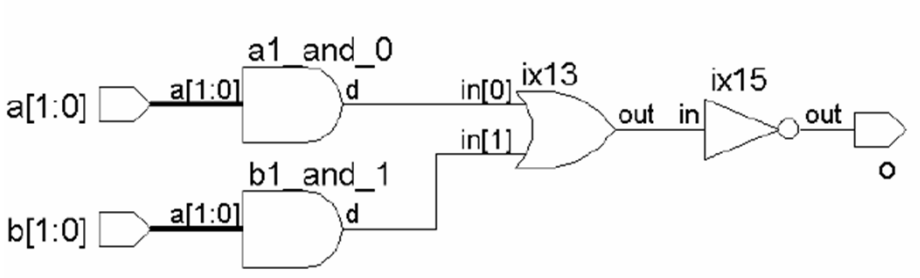
Designs at Behavioral Level

- All simple algebraic as well as logical expressions can be described at the behavioral level.

- One can also mix them with blocks at the gate level as well as the data flow level to form composite as well as more involved modules.

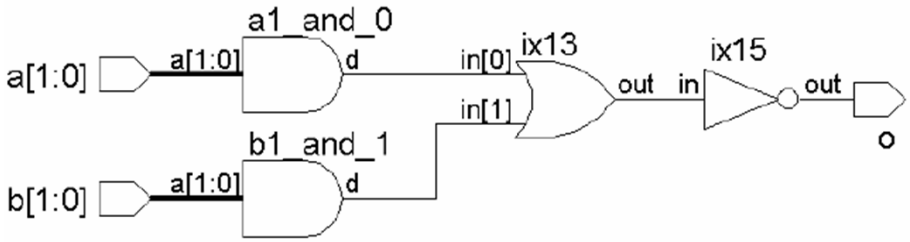
Example-1 (with begin & end Statement)

```
module aoibeh(o,a,b);
output o;
input[1:0]a,b;
reg o,a1,b1,o1;
always@(a[1] or a[0]or b[1]or b[0])
begin
a1=&a;
b1=&b;
o1=a1|b1;
o=~o1;
end
endmodule
```



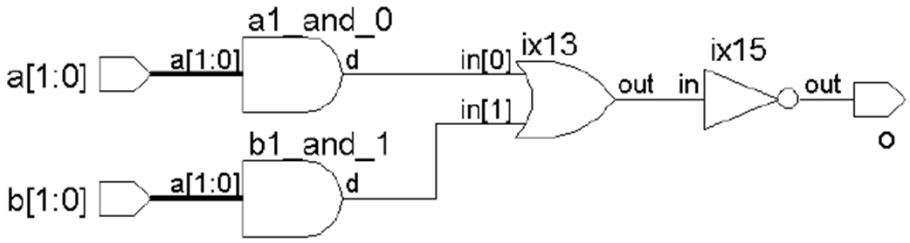
Example-2 (without begin & end Statement)

```
module aoibeh1(o,a,b);
output o;
input[1:0]a,b;
reg o;
always@(a[1]ora[0]or b[1]orb[0])
    o=~((&a)|(&b));
endmodule
```



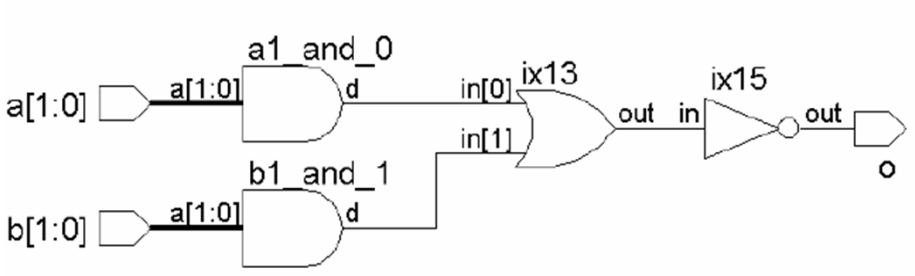
Example-3(Combination of primitive instantiation & Procedural assignment)

```
module aoibeh2(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
and g1(a1,a[1],a[0]),g2(b1,b[1],b[0]);
always@(a1 or b1)
o=~(a1|b1);
endmodule
```



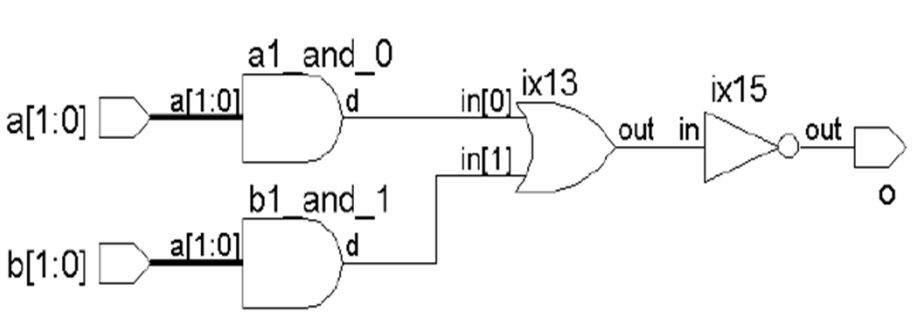
Example-4 (Combination of Continuous Assignment & Procedural assignment)

```
module aoibeh3(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
assign a1=&a,b1=&b;
always@(a1 or b1)o=~(a1|b1);
endmodule
```



Example-5 (Combination of Continuous Assignment ,Gate Instantiation & Procedural assignment)

```
module aoibeh4(o,a,b);
output o;
input[1:0]a,b;
wire a1,b1;
reg o;
assign a1=&a;
and g2(b1,b[1],b[0]);
always@(a1 or b1)
o=~(a1|b1);
endmodule
```



Block statements

- Provides a means of grouping two or more procedural statements together.

- Types of blocks:
  - *sequential block / **begin-end** block*
  - *parallel block / **fork-join** block*

- These blocks can be nested.

- These blocks can be mixed.

begin-end Construct

Format:

```
begin
  <statements>
end
```

- Statements are executed sequentially one after the other.
- If delay is specified, it is relative to the time when the previous statement in the block is executed.
- Control leaves the block after executing the last statement.

begin-end Construct

- If a procedural block has only one assignment to be carried out, it can be specified as below:

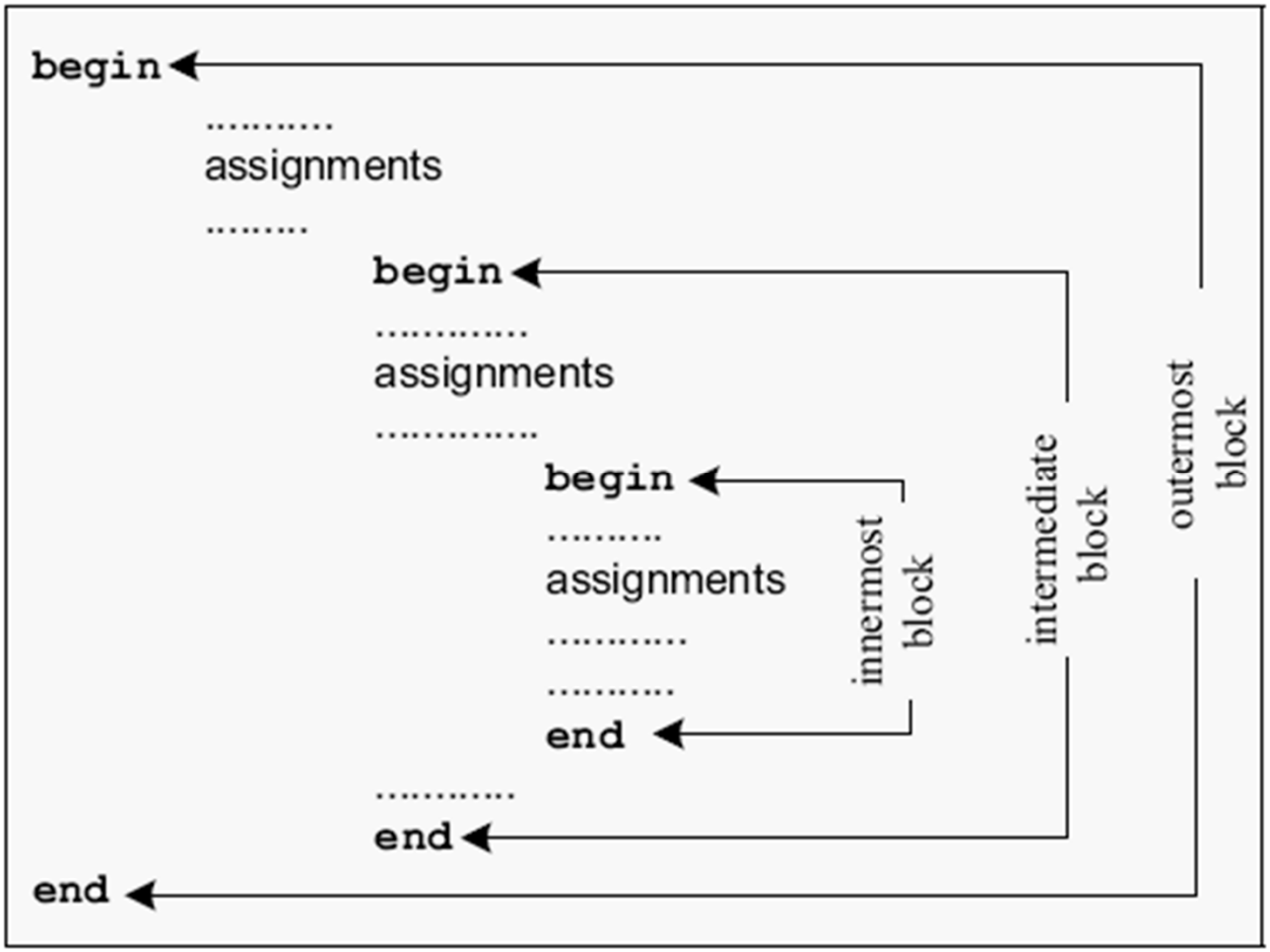
initial #2 a=0;

- More often more than one procedural assignment is to be carried out in an **initial** block**.**
- All such assignments are grouped together between "**begin**" **and "end"** declarations.

The following are to be noted here:

- Every **begin declaration must have its associated end declaration.**
- **begin – end constructs can be nested as many times as desired.**
- For clarity in description and to avoid mistakes, nested **begin – end blocks**
- are separated suitably

Nesting of begin-end block



Parallel block

Format:

```
fork
  <statements>
join
```

- Statements are executed concurrently.
- If delay is specified, it is relative to the time, when the block has started its execution.
- Control leaves the block after executing the last time ordered statement.

fork-join Construct

- The **fork–join block is an alternate one where all the** assignments are carried out concurrently

```
module fk_jn_a;
integer a;
initial
begin
a=0;
#1 a=1;
#2 a=2;
#3 a=3;
#4 $stop;
end
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule
//Simulation results
# a=0, t=0
# a=1, t=1
# a=2, t=3
# a=3, t=6
```

```
module fk_jn_b;
integer a;
initial
fork
a=0;
#1 a=1;
#2 a=2;
#3 a=3;
#4 $stop;
join
initial $monitor ("a=%0d,
t=%0d",a,$time);
endmodule
//Simulation results
# a=0, t=0
# a=1, t=1
# a=2, t=2
# a=3, t=3
```

Nested blocks

- Blocks can be nested.
- Sequential and parallel blocks can be mixed

```
//Nested blocks
initial
begin
x = 1'b0;
fork
#5 y = 1'b1;
 #10 z = {x, y};
join
#20 w = {y, x};
end
```

Named blocks

- Blocks can be given names.
- Local variables can be declared for the named block.
- Named blocks are a part of the design hierarchy.
- Named blocks can be disabled, i.e., their execution can be stopped.

```
//Named blocks module top;
 initial begin: block1 //sequential block named block1
integer i; //integer i is static and local to block1
... ...
 end
initial fork: block2 //parallel block named block2
reg i; // register i is static and local to block2
.. ... join
```

Disabling named blocks

- The keyword **disable** provides a way to terminate the execution of a named block.
- **disable** can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on a control signal.
- Disabling a block causes the execution control to be passed to the statement immediately succeeding the block.

Disabling named blocks -Example

```
reg [15:0] flag;
integer i; //integer to keep count
initial
begin
flag = 16'b 0010_0000_0000_0000;
 i = 0; begin: block1 //The main block inside while is named block1
while(i < 16)
begin
if (flag[i])
begin
$display("Encountered a TRUE bit at element number %d", i);
disable block1; //disable block1 because you found true bit.
end
i = i + 1;
end
end
end
```

Continuous Vs. procedural assignments

Continuous assignment	Procedural assignment
Occurs within a module.	Occurs inside an always or an initial statement.
Executes concurrently with other statements.	Execution is w.r.t. other statements surrounding it.
Drives nets.	Drives registers.
Uses “=” assignment operator.	Uses “=” or “<=” assignment operator.
Uses assign keyword.	No assign keyword (exception).

Procedural assignments

- These are for updating **reg**, **integer**, **real**, **time** and their bit / part selects.

- The values of the variables can get changed only by another procedural assignment statement.

- Procedural assignments are of two types
  - blocking procedural assignment
  - non-blocking procedural assignment

Blocking Assignment

- Most Commonly used type
- The target of assignment gets updated before the next sequential statement in procedural block is executed.
- A statement using blocking assignment blocks the execution of the statements following it, until it gets completed.
- Recommended style for modeling combinational logic.(data dependency)

Blocking Assignment-Example

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
 integer count; //All behavioral statements must be inside an initial or always block
initial
begin
x = 0;
y = 1;
 z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0;
reg_b = reg_a; //initialize vectors
#15 reg_a[2] = 1'b1; //Bit select assignment with delay
#10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to  part select of a vector
 count = count + 1; //Assignment to an integer (increment)
end
```

Non-Blocking Assignment

- The assignments to the target gets scheduled for the end of the simulation cycle.
  - Normally occurs at the end of the sequential block (**begin** ….. **end**)
  - Statements subsequent to the instruction under the consideration are not blocked by the assignment.
- Recommended style for modeling sequential logic

  - Can be used to assign several '**reg**' type variables synchronously, under the control of a common block

Nonblocking Assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial
begin
 x = 0;
 y = 1;
z = 1;
count = 0;
reg_a = 16'b0;
reg_b = reg_a;
reg_a[2] <= #15 1'b1;
reg_b[15:13] <= #10 {x, y, z};
 count <= count + 1;
 end
```

```
always @(posedge clock)
begin
 reg1 <= #1 in1;
 reg2 <= @(negedge clock) in2 ^ in3;
reg3 <= #1 reg1;
 end
```

Rules to be followed

- Verilog synthesizer ignores the delays specified in a procedural assignment statement.
  - May lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of both a blocking and a non-blocking assignment.

Following is not permissible

```
value = value +1;
value <=init
```

Parameter

A *parameter is a constant with a name.*

- No size is allowed to be specified for a parameter.

– The size gets decided from the constant itself (32-bits if nothing is specified).

- Examples:

parameter HI = 25, LO = 5;

Parameterized design: an N-bit counter

```
module counter (clear, clock, count);
parameter N = 7;
input clear, clock;
output [0:N] count;
reg [0:N] count;
always @ (negedge clock)
if (clear)
count <= 0;
else
count <= count + 1;
endmodule
```

Using more than one clocks in a module

```
module multiple_clk (clk1, clk2, a, b, c, f1, f2);
input clk1, clk2, a, b, c;
output f1, f2;
reg f1, f2;
always @ (posedge clk1)
f1 <= a & b;
always @ (negedge clk2)
f2 <= b ^ c;
endmodule
```

Using multiple edges of the same clock for High Speed Circuit Design

```
module multi_phase_clk (a, b, f, clk);
input a, b, clk;
output f;
reg f, t;
always @ (posedge clk)
f <= t & b;
always @ (negedge clk)
t <= a | b;
endmodule
```

A Ring Counter - Example

```
module ring_counter (clk, init, count);
input clk, init; output [7:0] count;
reg [7:0] count;
always @ (posedge clk)
begin
if (init)
count = 8'b10000000;
else
begin
count = count << 1;
count[0] = count[7];
end
end
endmodule
```

A Ring Counter Example (Modified-1)

```
module ring_counter_modi1 (clk, init, count);
input clk, init; output [7:0] count;
reg [7:0] count;
always @ (posedge clk)
begin
if (init)
count = 8'b10000000;
else
 begin
count <= count << 1;
count[0] <= count[7];
end
end
endmodule
```

A Ring Counter Example (Modified-2)

```
module ring_counter_modi1 (clk, init, count);
input clk, init; output [7:0] count;
reg [7:0] count;
always @ (posedge clk)
begin
if (init)
count = 8'b10000000;
else
count = { count[6:0], count[7]};
end
endmodule
```

Race Condition

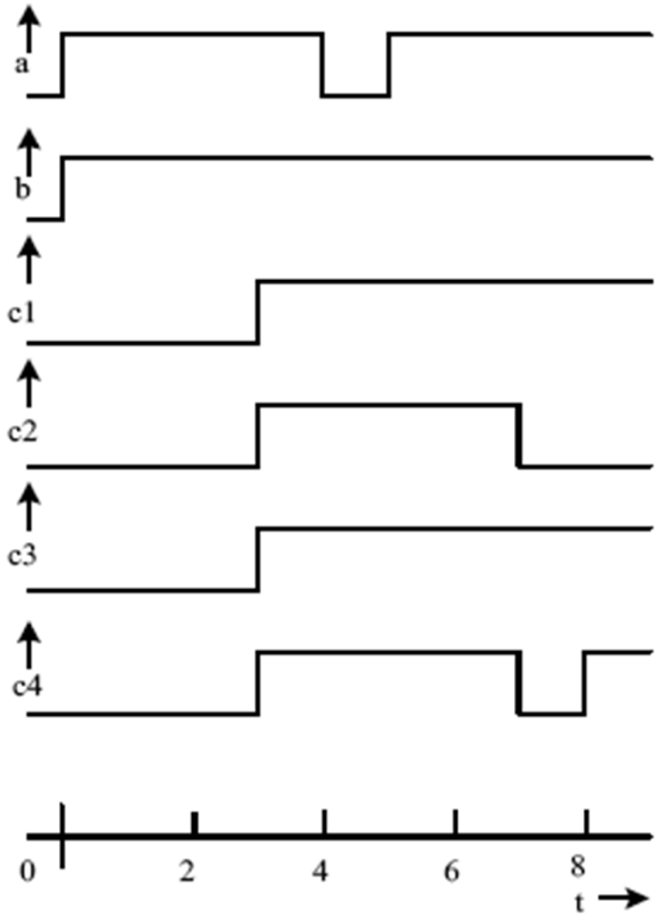
```
always @ (a or b)
a=b;
always @ (a or b)
b=a;
```

```
always @ (a or b)
 a=b;
always @ (x)
 b=a;
```

```
always @ (a or b)
a<=b;
always @ (a or b)
b<=a;
```

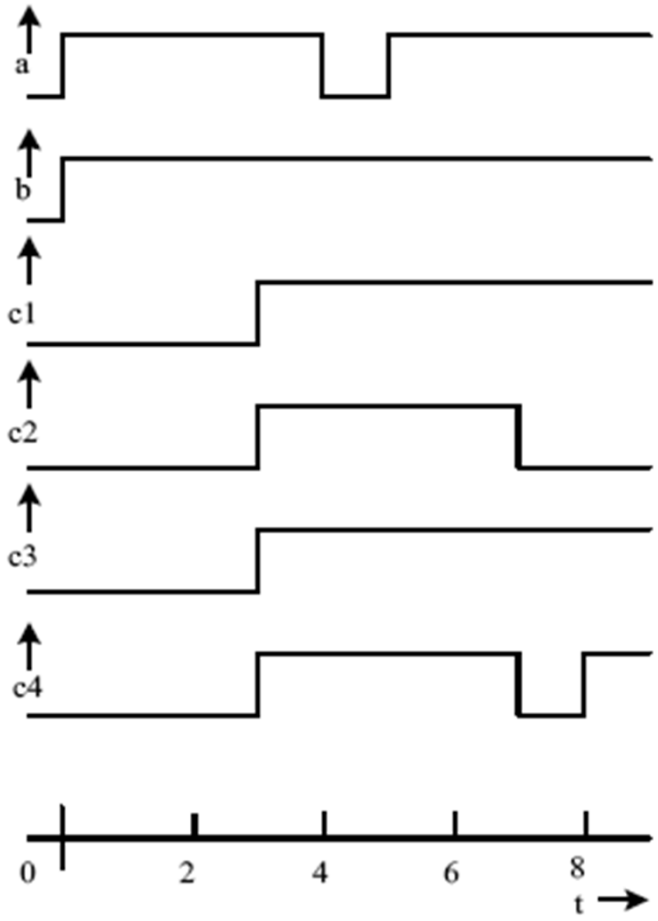
Blocking Assignment-Inter assignment delay

```
module nil1 (c1, a, b);
output c1;
input a, b;
reg c1;
always @(a or b)
#3 c1 = a & b;
endmodule
```



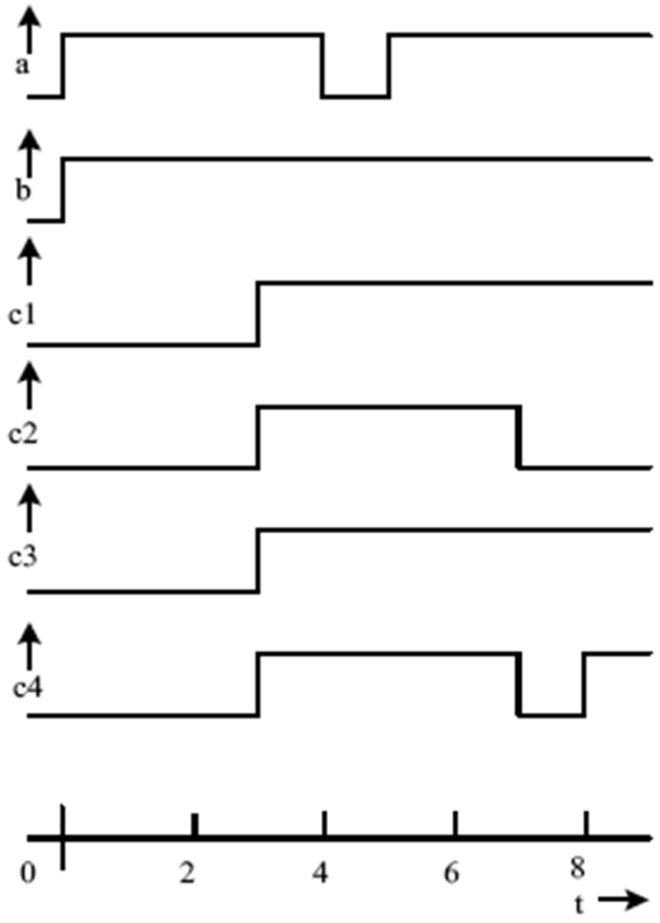
Blocking Assignment-Intra assignment delay

```
module nil2 (c2, a, b);
output c2;
input a, b;
reg c2;
always @(a or b)
c2 = #3 a&b;
endmodule
```



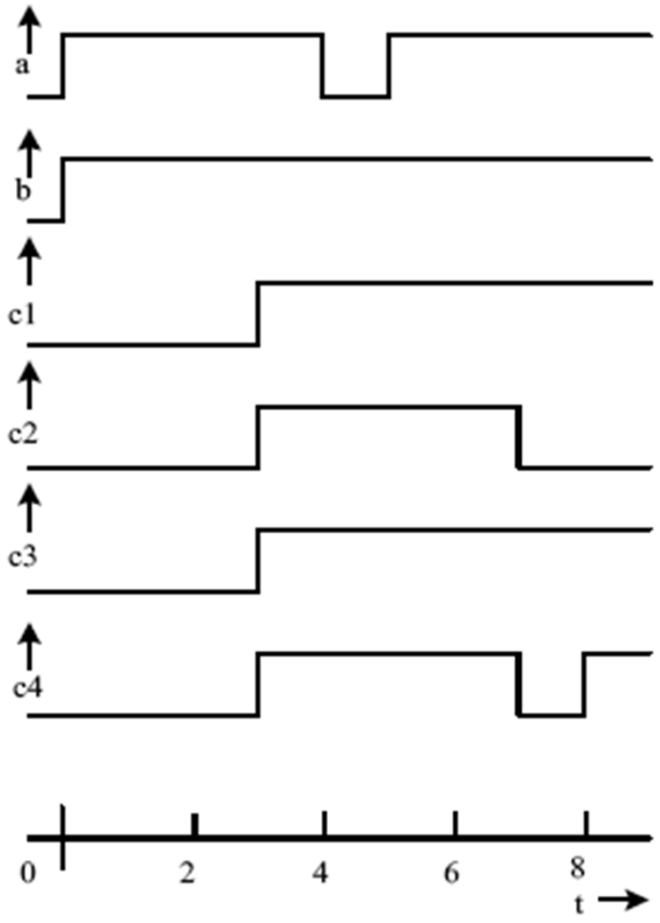
Non-Blocking Assignment-Inter assignment delay

```
module nil3 (c3, a, b);
output c3;
input a, b;
reg c3;
always @(a or b)
#3 c3 <= a&b;
endmodule
```



Non-Blocking Assignment-Intra assignment delay

```
module nil4 (c4, a, b);
output c4;
input a, b;
reg c4;
always @(a or b)
c4 <= #3 a&b;
endmodule
```



Reference

1. Samir Palnitkar,"Verilog HDL: A Guide to Digital Design and Synthesis" Prentice Hall, Second Edition,2003
2. T.R.Padmanabhan and B.Balatripura Sundari, "Design Through Verilog HDL" Wiley Student Edition