

Verilog HDL: Behavioral Modeling Procedural Assignments

Pravin Zode

- Structured Procedures
 - Initial
 - Always
- Blocking / Non-Blocking statements
- Wait statement
- Flow control
 - If..else
 - Case

Modeling Styles

Two different styles of description:

1. Data Flow

Continuous assignment

Using assignment statements

2. Behavioral

Procedural assignment

- Blocking
- Non-blocking

Using procedural statements similar to a program in high-level language

Introduction

- Procedural assignment models signal assignments based on an event (e.g., signal transition)
- Usage in **Sequential Logic**:
 - Used in flip-flops and finite state machines.
 - Assignments trigger on clock edges (e.g., always @posedge clk).
 - Can only drive variable data types (reg, integer, real, time).
- Usage in **Combinational Logic**:
 - Triggers assignments when any input changes.
 - Even though wires cannot be assigned, synthesizers recognize combinational logic.

Procedural Assignments

- Assignments execute sequentially in the order they are listed.
- Support for Programming Constructs :
 - if-else decisions.
 - case statements for multi-way branching.
 - loops (for, while, repeat, forever) for iterative execution
- Models sequential and combinational logic.
- Supports behavioral programming for testbenches

Procedural Blocks

- Procedural Assignments must be enclosed in procedural blocks (Sequential Statements)
- Verilog has two types of procedural blocks, *initial* and *always*
- Supports behavioral programming for testbenches

Initial Block

- All statements inside an initial block execute only once at simulation time 0
- After execution, the block does not run again
- If there are multiple initial blocks, all start concurrently at time 0
- Each block executes independently and finishes execution separately
- Multiple statements inside an initial block must be enclosed within begin ... End
- If there is only one statement, begin ... end is not required.

Example : Initial Block

```
module stimulus;

reg x,y, a,b, m;

initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial
    #50 $finish;

endmodule
```


always Block

- An always block executes repeatedly whenever its sensitivity condition is met
- Used for modeling sequential and combinational logic
- **Sensitivity List:** The always block is triggered by changes in signals listed in `@(...)`.
- **Example:** `always @(posedge clk)` triggers on rising edge of `clk`
- Multiple always Blocks: Each executes independently and concurrently.
- Useful for modeling multiple independent processes.

always @ event Expression

- The event expression specifies the event that is required to resume execution of the procedural block.
- The event can be any one of the following:
 - Change of a signal value.
 - Positive or negative edge occurring on signal (posedge or negedge) @ (posedge)
 - List of above-mentioned events, separated by “or” or comma @ (a or b or c), @(a,b,c)
 - Any variable @ (*)

always Block

Concurrent always Block : **Triggered by any input change**

```
1  always @(*) // Implicit sensitivity list
2  begin
3      | out = a & b; // AND gate
4  end
```

```
1  module xor3 (input a, b, c, output y);
2      reg y;
3      always @(a, b, c)
4      y = a ^ b ^ c;
5  endmodule
```

always Block

Majority Circuit

```
1  module maj3 (input a, b, c, output reg y);
2  always @(a, b, c)
3  begin
4  y = (a & b) | (b & c) | (a & c);
5  end
6  endmodule
```

Sequential always Block : **Triggered by clock edges**

```
1  always @(posedge clk)
2  begin
3  |    q <= d;  // D Flip-Flop
4  end
```

Example : Generating clock

```
1  module generating_clock;
2  output reg clk;
3  initial
4  |    clk = 1'b0; // initialized to 0 at time 0
5  always
6  |    #5 clk = ~clk; // Toggle after time 5 units
7  initial
8  #500 $finish;
9  endmodule
```

Wait statement

- The wait statement pauses execution until a specified condition becomes true
- Used in behavioral modeling for synchronization

```
wait (condition);
```

```
wait (ready == 1);  
data_out = data_in;
```

Example : Wait statement

```
1  ✓ module wait_example(input clk, input signal, output reg flag);  
2  ✓      always @(posedge clk) begin  
3          wait (signal == 1);  
4          flag = 1;  
5      end  
6  endmodule
```

- The process waits until signal is high before setting flag to 1

Assignment in Verilog

- Assignments in Verilog are categorized into:
 - Blocking (=) Assignments
 - Non-Blocking (<=) Assignments
- Blocking assignments execute sequentially
- Non-blocking assignments execute concurrently.

Blocking Assignment (=)

- Execute in order (step-by-step execution)
- Used in combinational logic
- Can cause race conditions in sequential circuits.

```
1  always @(posedge clk)
2  begin
3      a = b;
4      c = a; // Uses the new value of 'a'
5  end
```

c gets the updated value of a from the first assignment

Blocking Assignment (=)

```
1  module block;
2  reg [7:0] x1,x2,x3,y1,y2;
3  initial begin
4  x1=8'h10;
5  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
6  x2=8'h20;
7  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
8  x3=8'h30;
9  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
10 end
11
12 initial begin
13 y1=8'h40;
14 $display("[%0t] y1=0x%0h y2=0x%0h",$time, y1,y2);
15 y1=8'h50;
16 $display("[%0t] y1=0x%0h y2=0x%0h",$time, y1,y2);
17 end
18 endmodule
```

Output

[0]	x1=0x10	x2=0xxx	x3=0xxx
[0]	x1=0x10	x2=0x20	x3=0xxx
[0]	x1=0x10	x2=0x20	x3=0x30
[0]	y1=0x40	y2=0xxx	
[0]	y1=0x50	y2=0xxx	

Example: Delayed Blocking Assignment (=)

```
1  module block;
2  reg [7:0] x1,x2,x3,y1,y2;
3  initial begin
4  x1=8'h10;
5  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
6  #20 x2=8'h20;
7  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
8  #30 x3=8'h30;
9  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
10 end
11
12 initial begin
13 #15 y1=8'h40;
14 $display("[%0t] y1=0x%0h y2=0x%0h",$time, y1,y2);
15 #25 y1=8'h50;
16 $display("[%0t] y1=0x%0h y2=0x%0h",$time, y1,y2);
17 end
18 endmodule
```

Output

```
[0] x1=0x10 x2=0xxx x3=0xxx
[15] y1=0x40 y2=0xxx
[20] x1=0x10 x2=0x20 x3=0xxx
[40] y1=0x50 y2=0xxx
[50] x1=0x10 x2=0x20 x3=0x30
```

Non-Blocking Assignment (<=)

- Execute concurrently (all updates happen at the end of the time step)
- Used in sequential logic
- Prevents race conditions in flip-flops and registers.

```
1  always @(posedge clk)
2  √ begin
3      a <= b;
4      c <= a; // Uses the old value of 'a'
5  end
```

c does not get the updated a immediately.

Non-Blocking Assignment (<=)

```
1  module block;
2  reg [7:0] x1,x2,x3,y1,y2;
3  initial begin
4  x1<=8'h10;
5  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
6  x2<=8'h20;
7  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
8  x3<=8'h30;
9  $display("[%0t] x1=0x%0h x2=0x%0h x3=0x%0h",$time, x1,x2,x3);
10 end
11
12 initial begin
13 y1<=8'h40;
14 $display("[%0t] y1=0x%0h y2=0x%0h",$time, y1,y2);
15 y1<=8'h50;
16 $display("[%0t] y1=0x%0h y2=0x%0h",$time, y1,y2);
17 end
18 endmodule
```

```
[0] x1=0xxx x2=0xxx x3=0xxx
[0] x1=0xxx x2=0xxx x3=0xxx
[0] x1=0xxx x2=0xxx x3=0xxx
[0] y1=0xxx y2=0xxx
[0] y1=0xxx y2=0xxx
```

All outputs are unknown here

Nothing is assigned at 0ns all statements are executed

So blocking assignments are preferred if no delays are present

Example

```
1  module add_1bit (input a, b, ci, output s, co );
2      reg s, co;
3      always @(a, b, ci) begin
4          s = #5 a ^ b ^ ci;
5          co = #3 (a & b) | (b & ci) | (a & ci);
6      end
7  endmodule
```

The carry-out (co) is computed before s, but s depends on co. incorrect behavior in simulations where the ordering matters

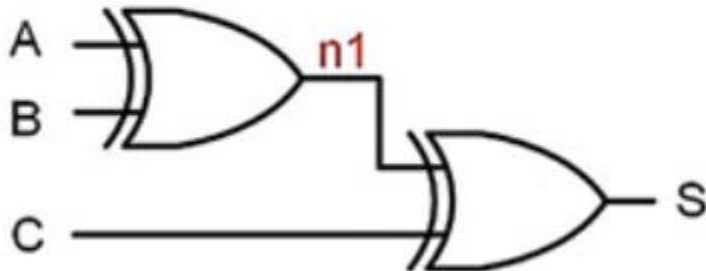
```
1  ✓ module add_1bit (input a, b, ci, output s, co );
2      reg s, co;
3  ✓  always @(a, b, ci) begin
4          s <= #5 a ^ b ^ ci;
5          co <= #3 (a & b) | (b & ci) | (a & ci);
6      end
7  endmodule
```

both updates are scheduled at the same time, this prevents race conditions

Example: Blocking Assignments

```
module BlockingEx1 (output reg S,  
                    input  wire A, B, C);  
  
    reg n1;  
  
    always @ (A, B, C)  
    begin  
        n1 = A ^ B;    // statement 1  
        S = n1 ^ C;    // statement 2  
    end  
  
endmodule
```

Resulting Circuit

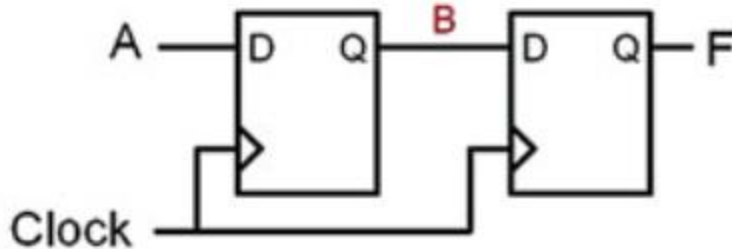


Both statement 1 and statement 2 are treated as separate circuits that execute concurrently when using blocking assignments.

Example: Non-Blocking Assignments

```
module NonBlockingEx1 (output reg F,  
                      input  wire A,  
                      input  wire Clock);  
  
    reg B;  
  
    always @ (posedge Clock)  
    begin  
        B <= A;    // statement 1  
        F <= B;    // statement 2  
    end  
  
endmodule
```

Resulting Circuit



Notice that the value of B in statement 2 is not immediately updated with the assignment made in statement 1 due to the nature of non-blocking assignments.

Example: Identical Behavior

```
module BlockingEx2
(output reg Y, Z,
 input wire A, B, C);

always @ (A, B, C)
begin
    Y = A & B;    // statement 1
    Z = B | C;    // statement 2
end

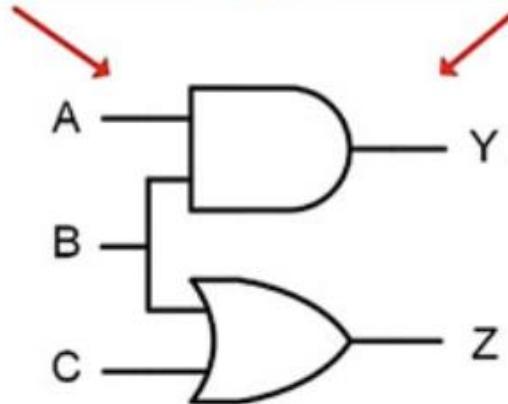
endmodule
```

```
module NonBlockingEx2
(output reg Y, Z,
 input wire A, B, C);

always @ (A, B, C)
begin
    Y <= A & B;    // statement 1
    Z <= B | C;    // statement 2
end

endmodule
```

Resulting Circuit



Both modeling approaches yield the same result because there are no signal interdependences between the statements.

Best Practices for Assignments

- Use = for combinational logic inside always @(*)
- Use <= for sequential logic inside always @(posedge clk)
- Avoid mixing = and <= in the same block
- Follow proper coding guidelines to prevent timing mismatches

Feature	Blocking (=)	Non-Blocking (<=)
Execution Order	Sequential	Concurrent
Usage	Combinational Logic	Sequential Logic
Timing	Immediate Update	Update at End of Cycle
Data Dependency	Affects Next Statement Immediately	Old Value Used

Flow Control

- Conditional statements alter the flow within a behavior based on certain conditions
- The choice among alternative statements depends on the Boolean value of an expression
- Alternative statements can be:
 - A single statement
 - A block of statements enclosed by begin ... End
- Keywords: if, else, case

Example : if....else statement

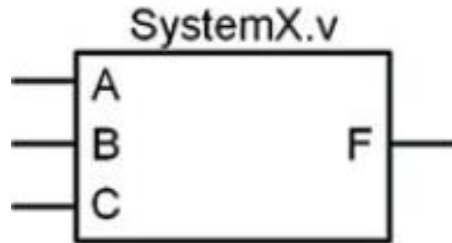
Multiplexer

```
1  module mux2_1 (input i0, i1, s, output reg y);
2  always @(i0, i1, s) begin
3  if (s==1'b0)
4  y = i0;
5  else
6  y = i1;
7  end
8  endmodule
```

Adder_Subtractor

```
1  module add_sub_4bit (input [3:0] a, b, input ci, m,
2  output reg [3:0] s, output reg co );
3  always @(a, b, ci, m)
4  |   if ( m )
5  |       { co, s } = a + b + ci;
6  |   else
7  |       { co, s } = a - b - ci;
8  endmodule
```

Example : if....else statement



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



```
module SystemX
(output reg F,
 input wire A, B, C);

always @ (A, B, C)
begin
    if      (A==1'b0 && B==1'b0 && C==1'b0)
        F = 1'b1;
    else if (A==1'b0 && B==1'b1 && C==1'b0)
        F = 1'b1;
    else if (A==1'b1 && B==1'b1 && C==1'b0)
        F = 1'b1;
    else
        F = 1'b0;
end

endmodule
```

when modeling combinational logic using procedural assignment , all inputs to the circuit must be listed in the sensitivity list and blocking assignments are used

Example : DFF if...else statement

```
1  // D flip-flop with synchronous set and reset
2  module dff (q, qbar, d, set, reset, clk);
3  input d, set, reset, clk;
4  output reg q; output qbar;
5  assign qbar = ~q;
6  always @ (posedge clk)
7  begin
8      if (reset == 0) q <= 0;
9  else if (set == 0) q <= 1;
10     else q <= d;
11 end
12 endmodule
```

Example : DFF if....else statement

```
1  // D flip-flop with asynchronous set and reset
2  module dff (q, qbar, d, set, reset, clk);
3  input d, set, reset, clk;
4  output reg q; output qbar;
5  assign qbar = ~q;
6  always @ (posedge clk or negedge set or negedge reset)
7  begin
8  ✓ if (reset == 0)
9  |     q <= 0;
10 ✓ else if (set == 0)
11 |     q <= 1;
12 ✓ else
13 |     q <= d;
14 end
15 endmodule
```

Example : Counter

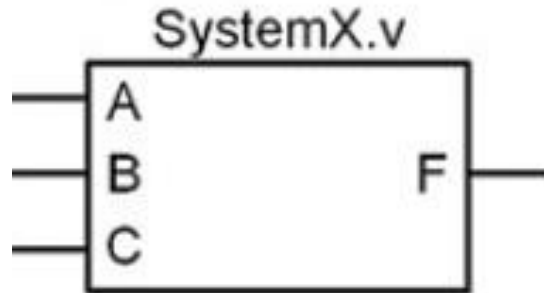
```
module counter (clear, clock, count);  
    parameter N = 7;  
    input clear, clock;  
    output reg [0:N] count;  
  
    always @(negedge clock)  
        if (clear)  
            count <= 0;  
        else  
            count <= count + 1;  
endmodule
```

```
module test_counter;  
    reg clk, clr;  
    wire [7:0] out;  
  
    counter CNT (clr, clk, out);  
  
    initial clk = 1'b0;  
  
    always #5 clk = ~clk;  
  
    initial  
        begin  
            clr = 1'b1;  
            #15 clr = 1'b0;  
            #200 clr = 1'b1;  
            #10 $finish;  
        end  
    initial  
        begin  
            $dumpfile ("counter.vcd");  
            $dumpvars (0, test_counter);  
            $monitor ($time, " Count: %d", out);  
        end  
endmodule
```


Case statement

- The case statement is used for selecting between multiple values efficiently.
- It can only be used inside a procedural block (**always** or **initial**).
- The statement starts with the **case** keyword, followed by an expression in parentheses.
- Each case branch consists of an input condition followed by an assignment.
- Multiple input conditions with the same assignment can be comma-separated.
- The **default** keyword is used for cases not explicitly listed.
- The statement is closed with the **endcase** keyword.

Case statement



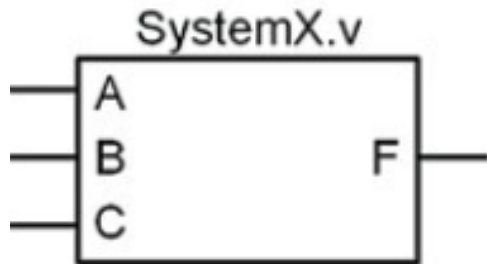
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



```
module SystemX
  (output reg F,
   input wire A, B, C);
  always @ (A, B, C)
  begin
    case ( {A,B,C} )
      3'b000 : F = 1'b1;
      3'b001 : F = 1'b0;
      3'b010 : F = 1'b1;
      3'b011 : F = 1'b0;
      3'b100 : F = 1'b0;
      3'b101 : F = 1'b0;
      3'b110 : F = 1'b1;
      3'b111 : F = 1'b0;
      default : F = 1'bX;
    endcase
  end
endmodule
```

Case statement

Alternate approach



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



```
case ( {A,B,C} )  
  3'b000 : F = 1'b1;  
  3'b010 : F = 1'b1;  
  3'b110 : F = 1'b1;  
  default : F = 1'b0;  
endcase
```

```
case ( {A,B,C} )  
  3'b000, 3'b010, 3'b111 : F = 1'b1;  
  default                : F = 1'b0;  
endcase
```

Example : case statement

```
1  module mux_4to1(input [1:0] sel, input [7:0] in0, in1, in2, in3, output reg [7:0] out);
2      always @(*) begin
3          case (sel)
4              2'b00: out = in0;
5              2'b01: out = in1;
6              2'b10: out = in2;
7              2'b11: out = in3;
8              default: out = 8'b00000000;
9          endcase
10     end
11 endmodule
```

Wildcard in case statement

- case, casez, and casex are used for multi-way branching
- Syntax is similar, but behavior differs with respect to x and z bits
- Useful in FSMs and combinational logic

```
case (sel)
  2'b00: out = 0;
  2'b01: out = 1;
  2'b1x: out = 2; // Won't match if sel contains 'x'
```

Don't card in regular case statement

Example: casex

```
1  module priority (W, Y, z);
2      input [3:0] W;
3      output reg [1:0] Y;
4      output reg z;
5      always @(W)
6      begin
7          z = 1;
8          casex (W)
9              4'b1xxx: Y = 3;
10             4'b01xx: Y = 2;
11             4'b001x: Y = 1;
12             4'b0001: Y = 0;
13         default:
14             begin
15                 z = 0;
16                 Y = 2'bx;
17             end
18         endcase
19     end
20 endmodule
```

Example : casez

- Variation of case that treats z and ? as wildcards in case items
- Allows flexible pattern matching using "don't care" bits.

```
1  module casez_example(input [3:0] sel, output reg [7:0] out);
2
3  always @(*) begin
4      casez (sel)
5          4'b1???: out = 8'hF0; // Matches any input starting with 1 (1000-1111)
6          4'b01??: out = 8'h0F; // Matches 0100-0111
7          4'b0001: out = 8'hAA; // Matches exactly 0001
8          default: out = 8'h00; // Fallback for anything else
9      endcase
10 end
11
12 endmodule
```

Comparison : Case statements

Feature	case	casez	casex
Treats x as wildcard	✗	✗	✓
Treats z as wildcard	✗	✓	✓
Wildcard in expression	✗	✗	✓
Safety	✓ (Safe)	⚠ (Moderate – use with care)	✗ (Risky – avoid if possible)
When to Use	Default, strict matching	Decoders, FSMs with "don't cares"	Rarely, legacy or synthesized code

Summary

- Sequential blocks model time-dependent behavior
- initial is used once for simulation setup or testing
- if-else is best for conditional decision-making
- case, casez, casex are used for multi-way branching
- Always place these inside always blocks for procedural logic
- Use non-blocking (\leq) assignments for sequential circuits



Thank you !

Happy Learning