Verilog HDL:
Block Statements

Pravin Zode

Outline

- Initial begin end block
- Fork-join block
- Named block
- Generate block

Block statement

- Block statement is a way of grouping multiple procedural statements together
- They make the group of statements act syntactically like a single statement
- Useful inside procedural blocks such as initial and always
- Two types of grouped blocks are allowed in Verilog:
  - Sequential Block (begin … end) → (Sequential)
  - Parallel Block (fork … join) → (Parallel)
- Helps in organizing code, especially when multiple statements must execute in a controlled manner.

Sequential Block

- Uses begin … end Statements inside are executed sequentially (one after the other)
- Execution blocks until all statements are finished
- Can contain blocking (=) or non-blocking (<=) assignments

```
initial begin
    a = 1;     // executes first
    b = 2;     // executes after 'a'
    c = a + b; // executes last
end
```

Example : Sequential Block

```
1   module stimulus;
2
3   reg x,y, a,b, m;
4
5   initial
6       m = 1'b0; //single statement; does not need to be grouped
7
8   initial
9   begin
10      #5 a = 1'b1; //multiple statements; need to be grouped
11      #25 b = 1'b0;
12  end
13
14  initial
15  begin
16      #10 x = 1'b0;
17      #25 y = 1'b1;
18  end
19
20  initial
21      #50 $finish;
22
23  endmodule
```

Fork Join

- Used to create parallel execution of multiple blocks
- Part of the initial or always block.
- All blocks inside fork...join run simultaneously
- Control resumes only after all parallel threads complete
- Commonly used in testbenches, simulation timing, or parallel task modeling

Fork Join

- Code inside fork...join runs in parallel
- Execution resumes after all parallel blocks finish.

```
initial begin
  fork
    // Parallel block 1
    // Parallel block 2
    // ...
  join
end
```

Example: Fork Join

```
1   module fork_join_demo;
2
3     initial begin
4       $display("Simulation starts at time = %0t", $time);
5
6       fork
7         begin
8           #5 $display(">> Task A completed at time = %0t", $time);
9         end
10
11        begin
12          #10 $display(">> Task B completed at time = %0t", $time);
13        end
14
15        begin
16          #3 $display(">> Task C completed at time = %0t", $time);
17        end
18      join
19
20      $display("All parallel tasks done. Resuming after fork-join at time = %0t", $time);
21    end
22
23  endmodule
```

Example Output : Fork Join

```
1   module fork_join_demo;
2
3     initial begin
4       $display("Simulation starts at time = %0t", $time);
5
6       fork
7         begin
8           #5 $display(">> Task A completed at time = %0t", $time);
9         end
10
11        begin
12          #10 $display(">> Task B completed at time = %0t", $time);
13        end
14
15        begin
16          #3 $
17        end
18      join
19
20      $display
21    end
22
23  endmodule
```

```
Simulation starts at time = 0
>> Task C completed at time = 3
>> Task A completed at time = 5
>> Task B completed at time = 10
All parallel tasks done. Resuming after fork-join at time = 10
```

Nested Blocks

- Sequential and parallel blocks can be nested

```
2   module sequential;
3
4   reg x, y;
5   reg [1:0] z, w;
6
7   initial
8           $monitor($time, " x = %b, y = %b, z = %b, w = %b\n", x, y, z, w);
9
10  //Nested blocks
11  initial
12  begin
13          x = 1'b0;
14          fork
15                  #5 y = 1'b1;
16                  #10 z = {x, y};
17          join
18          #20 w = {y, x};
19  end
20
21  endmodule
```

Named Blocks

Blocks can be given names

- Local variables can be declared inside named block
- Named blocks are part of design hierarchy
- Variables in a named block can be accessed by using hierarchical name referencing
- Named blocks can be disabled. i.e. their execution can be stopped.

Example: Named Block

```
//Named blocks
module top;

initial
begin: block1 //sequential block named block1
integer i; //integer i is static and local to block1
           // can be accessed by hierarchical name, top.block1.i
…
…
end

initial
fork: block2 //parallel block named block2
reg i; // register i is static and local to block2
                 // can be accessed by hierarchical name, top.block2.i
…
…
join
```

Disabling Named Block

- Keyword disable provides a way to terminate the execution of named block
- It is similar to BREAK statement in C.
- It is used to get out of loops, handle error conditions or control execution of piece of code

Example: Disabling Named Block

```
1   module find_true_bit;
2   //Illustration: Find the first bit with a value 1 in flag (vector variable)
3   reg [15:0] flag;
4   integer i; //integer to keep count
5   initial
6   begin
7     flag = 16'b 0010_0000_0000_0000;
8     i = 0;
9     begin: block1 //The main block inside while is named block1
10          while(i < 16)
11                  begin
12       if (flag[i])
13       begin
14          $display("Encountered a TRUE bit at element number %d", i);
15          disable block1; //disable block1 because you found true bit.
16       end
17       i = i + 1;
18                  end
19    end
20  end
21  endmodule
```

Generate Block

- Used in synthesizable Verilog to instantiate modules or logic repetitively or conditionally.
- Mainly used in RTL coding for Arrays of logic, Parameterized designs, Clean and scalable design structure
- Generate block structure
  - for-generate (loop)
  - if-generate (conditional)
  - case-generate (case-based structure)

Generate Block

- Generate loop permits one or more following to be instantiated multiple times using a for loop
  - Variable declarations
  - Modules
  - User defined primitives, gate primitives
  - Continuous assignments
  - `Initial` and `always` blocks
- Data types supported inside generate scope
    - Net, reg
    - Integer, real, time, realtime
    - event

Restrictions in Generate Statements

Following module declarations and module items are not permitted in generate statement

- Parameters, local parameters
- Input, output, inout declarations
- Specify blocks

For Generate Block

```
genvar i;
generate
  for (i = 0; i < 4; i = i + 1) begin : gen_loop
    my_module u (.a(in[i]), .b(out[i]));
  end
endgenerate
```

- Instantiates 4 copies of my_module with indexed connections

For Generate Block with gate level primitives

```
1   module bitwise_xor_gate #(parameter N = 8) (
2       input  wire [N-1:0] a,
3       input  wire [N-1:0] b,
4       output wire [N-1:0] y
5   );
6
7   genvar i;
8   generate
9       for (i = 0; i < N; i = i + 1) begin : xor_loop
10          xor (y[i], a[i], b[i]);
11      end
12  endgenerate
13
14  endmodule
```

For Generate Block with assign

```
1  module bitwise_xor #(parameter N = 8) (
2      input  wire [N-1:0] a,
3      input  wire [N-1:0] b,
4      output wire [N-1:0] y
5  );
6
7  genvar i;
8  generate
9      for (i = 0; i < N; i = i + 1) begin : xor_gen
10         assign y[i] = a[i] ^ b[i];
11     end
12 endgenerate
13
14 endmodule
```

For Generate Block with always

```
1  module bitwise_xor #(parameter N = 8) (
2      input  wire [N-1:0] a,
3      input  wire [N-1:0] b,
4      output reg  [N-1:0] y
5  );
6
7  genvar i;
8  generate
9      for (i = 0; i < N; i = i + 1) begin : xor_gen
10         always @(*) begin
11             y[i] = a[i] ^ b[i];
12         end
13     end
14 endgenerate
15
16 endmodule
```

Example : Full Adder

```verilog
module full_adder (
    input  a,
    input  b,
    input  cin,
    output sum,
    output cout
);

wire axorb, aandb, aandcin, bandcin;

// sum = a ^ b ^ cin
xor (axorb, a, b);
xor (sum, axorb, cin);

// cout = (a & b) | (a & cin) | (b & cin)
and (aandb, a, b);
and (aandcin, a, cin);
and (bandcin, b, cin);
or  (cout, aandb, aandcin, bandcin);
```

```verilog
module ripple_carry_adder #(parameter N = 4)(
    input  [N-1:0] a,
    input  [N-1:0] b,
    input          cin,
    output [N-1:0] sum,
    output         cout
);
wire [N:0] carry;
assign carry[0] = cin;
genvar i;
generate
    for (i = 0; i < N; i = i + 1) begin : rca_stage
        full_adder fa (
            .a(a[i]),
            .b(b[i]),
            .cin(carry[i]),
            .sum(sum[i]),
            .cout(carry[i+1])
        );
    end
endgenerate
assign cout = carry[N];
endmodule
```

IF Generate Block (Conditional Instantiation)

```
generate
  if (MODE == 1) begin : gen_mode1
    // logic for mode 1
  end else begin : gen_mode0
    // logic for mode 0
  end
endgenerate
```

- Very useful for selecting implementations based on parameters

IF Generate Block (Conditional Instantiation)

```
1   module simple_if_generate #(
2       parameter USE_AND = 1  // Set to 1 for AND, 0 for OR
3   )(
4       input wire a,
5       input wire b,
6       output wire y
7   );
8
9   // Conditional logic using if-generate
10  generate
11      if (USE_AND) begin : and_block
12          assign y = a & b;
13      end else begin : or_block
14          assign y = a | b;
15      end
16  endgenerate
17
18  endmodule
```

Case-Generate (Select Instantiation)

```
generate
  case (DATA_WIDTH)
    8: begin : gen8
      // logic for 8-bit
    end
    16: begin : gen16
      // logic for 16-bit
    end
    default: begin : gen_default
      // default logic
    end
  endcase
endgenerate
```

- Useful in flexible bus-width or multi-mode designs

Case-Generate (Select Instantiation)

```
1   module generate_case_example #(
2       parameter MODE = 0  // Selects the operation: 0 = AND, 1 = OR, 2 = XOR)
3       (input wire a,
4       input wire b,
5       output wire y
6   );
7   generate
8       case (MODE)
9           0: begin : and_block
10              assign y = a & b;
11          end
12          1: begin : or_block
13              assign y = a | b;
14          end
15          2: begin : xor_block
16              assign y = a ^ b;
17          end
18          default: begin : default_block
19              assign y = 1'b0; // Default output
20          end
21      endcase
22  endgenerate
23  endmodule
```

Thank you !

Happy Learning