MODULE-4

Behavioral Modeling: Structured procedures, initial and always, blocking and non-blocking statements, delay control, generate statement, event control, conditional statements, Multiway branching, loops, sequential and parallel blocks. Tasks and Functions: Differences between tasks and functions, declaration, invocation, automatic tasks and functions.

Behavioral Modeling

4.1 Structured Procedures

There are two structured procedure statements in Verilog: **always** and **initial**. These statements are the two most basic statements in behavioral modeling. All other behavioral statement can appear only inside these structured procedure statements.

Verilog is a concurrent programing language. Activity flows in Verilog nun in parallel rather than in sequence. Each **always** and **initial** statement represents a separate activity flow in Verilog. Each activity flow starts at simulation time 0. The statements **always** and **initial** cannot be nested.

4.1.1 Initial statement

All statements inside an initial statement constitute an **initial** block. An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple **initial** blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks. Multiple behavioral statements must be grouped, typically using the keywords **begin** and **end**. If there is only one behavioral statement, grouping is not necessary. Example 4.1 illustrates the use of the **initial** statement.

Example 4.1: initial statements

```
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

The execution sequence of the statements inside the initial blocks will be as follows:

Time	Statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

Combined Variable Declaration and Initialization

Variables can be initialized when they are declared. Example 4-2 shows such a declaration.

Example 4.2: Initial Value Assignment

//The clock variable is defined first

reg clock;

//The value of clock is set to 0

initial clock = 0;

// The clock variable can be initialized at the time of declaration itself.

reg clock = 0;

Combined Port/Data Declaration and Initialization

The combined port/data declaration can also be combined with an initialization. Example 4-3 shows such a declaration.

Example 4.3: Combined Port/Data Declaration and Variable Initialization

module adder (sum, co, a, b, ci);

```
output reg [7:0] sum = 0; //Initialize 8 bit output sum
output reg co = 0; //Initialize 1 bit output co
input  [7:0]  a,  b;
input ci;
--
endmodule
```

Combined ANSI C Style Port Declaration and Initialization

ANSI C style port declaration can also be combined with an initialization. Example 4-4 shows such a declaration.

Example 4.4: Combined ANSI C Port Declaration and Variable Initialization

```
module adder (output reg [7:0] sum = 0, reg co = 0, input [7:0] a, b, input ci);
--
endmodule
```

4.1.2 always statement

All behavioral statements inside an always statement constitute an always block. The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit. An example is a clock generator module that toggles the clock signal every half cycle. In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on. Example 4-5 illustrates one method to model a clock generator in Verilog.

Example 4.5: always Statement

```
module clock_gen (output reg clock);
//Initialize clock at time zero initial
clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always
#10 clock =~clock;
initial
#1000 $finish;
endmodule
```

In Example 4-5, the always statement starts at time 0 and executes the statement clock =

~clock every 10 time units. If there is no $stop or $finish statement to halt the simulation, the clock generator will run forever.

4.2 Procedural Assignments

Procedural assignments update values of **reg, integer, real,** or **time** variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value. These are unlike continuous assignments, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net. The syntax for the simplest form of procedural assignment is shown below:

assignment ::= variable_lvalue = [delay_or_event_control]
expression

The left-hand side of a procedural assignment <lvalue> can
be one of the following:

- A reg, integer, real, or time register variable or a memory element
- A bit select of these variables (e.g., addr[0])
- A part select of these variables (e.g., addr[31:16])
- A concatenation of any of the above

The right-hand side can be any expression that evaluates to a value. In behavioral modeling, all operators can be used in behavioral expressions.

There are two types of procedural assignment statements:
blocking and nonblocking.

4.2.1 Blocking Assignments

Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block. The = operator is used to specify blocking assignments.

Example 4.6: Blocking Statements

```
reg x, y, z;
reg [15:0] reg_a, reg_b; integer count;
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
```

In Example 4-6, the statement y = 1 is executed only after x = 0 is executed. The behavior in a particular block is sequential in a begin-end block if blocking statements are used, because the statements can execute only in sequence. The statement count = count + 1 is executed last. The simulation times at which the statements are executed are as follows:

- All statements x = 0 through reg_b = reg_a are executed at time 0
- Statement reg_a[2] = 0 at time = 15
- Statement reg_b[15:13] = {x, y, z} at time = 25
- Statement count = count + 1 at time = 25
- Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

4.2.2 Nonblocking Assignments

Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A <= operator is used to specify nonblocking assignments. The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment. To illustrate the behavior of nonblocking statements and its difference from blocking statements, let us consider Example 4-7, where we convert some blocking assignments to nonblocking assignments.

Example 4.7: Nonblocking Assignments

```
reg x, y, z;
reg  [15:0]  reg_a,  reg_b;
integer count;
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
reg_a[2] <= #15 1'b1; //Bit select assignment with delay
reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation to part select of a vector
```

```
count <= count + 1; //Assignment to an integer (increment)
end
```

In this example, the statements x = 0 through reg_b = reg_a are executed sequentially at time 0. Then the three nonblocking assignments are processed at the same simulation time.

1. reg_a[2] = 0 is scheduled to execute after 15 units (i.e., time = 15)
2. reg_b[15:13] = {x, y, z} is scheduled to execute after 10 time units (i.e., time = 10)
3. count = count + 1 is scheduled to be executed without any delay (i.e., time = 0)

Thus, the simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution. Typically, nonblocking assignment statements are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.

4.3 Timing Controls

Various behavioral timing control constructs are available in Verilog. In Verilog, if there are no timing control statements, the simulation time does not advance. Timing controls provide a way to specify the simulation time at which procedural statements will execute.

There are three methods of timing control: delay-based timing control, event-based timing control, and level- sensitive timing control.

4.3.1 Delay-Based Timing Control

Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed. Delays are specified by the symbol #. Syntax for the delay-based timing control statement is shown below.

```
delay3 ::= # delay_value | # ( delay_value [ , delay_value [ , delay_value ] ] )
delay2 ::= # delay_value | # ( delay_value [ , delay_value ] )
delay_value ::=
unsigned_number
| parameter_identifier
| specparam_identifier
| mintypmax_expression
```

Delay-based timing control can be specified by a number, identifier, or a mintypmax_expression. There are three types of delay control for procedural assignments: regular delay control, intra-assignment delay control, and zero delay control.

Regular delay control

Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment. Usage of regular delay control is shown in Example 4.10.

Example 4.10: Regular Delay Control

```
//define parameters
parameter latency = 20; parameter delta = 2;
//define register variables
reg x, y, z, p, q;
initial
begin
x = 0; // no delay control
#10 y = 1; // delay control with a number. Delay execution of y = 1 by 10 units
#latency z = 0; // Delay control with identifier. Delay of 20 units
#(latency + delta) p = 1; // Delay control with expression
#y x = x + 1; // Delay control with identifier. Take value of y.
#(4:5:6) q = 0; // Minimum, typical and maximum delay values.
end
```

In Example 4.10, the execution of a procedural assignment is delayed by the number specified by the delay control. For begin-end groups, delay is always relative to time when the statement is encountered. Thus, y =1 is executed 10 units after it is encountered in the activity flow.

Intra-assignment delay control

Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator. Such delay specification alters the flow of activity in a different manner. Example 4.11 shows the contrast between intra-assignment delays and regular delays.

Example 4.11: Intra-assignment Delays

```
//define register variables
reg x, y, z;
//intra assignment delays
initial
begin
x = 0; z = 0;
```

```
y = #5 x + z; //Take value of x and z at the time=0, evaluate x + z and then wait 5 time units
//to assign value to y.
end
//Equivalent method with temporary variables and regular delay control
initial
begin
x = 0; z = 0;
temp_xz = x + z;
#5 y = temp_xz; //Take value of x + z at the current time and store it in a temporary variable.
end
```

Zero delay control

Procedural statements in different always-initial blocks may be evaluated at the same simulation time. The order of execution of these statements in different always-initial blocks is nondeterministic. Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed. This is used to eliminate race conditions. However, if there are multiple zero delay statements, the order between them is nondeterministic. Example 4.12 illustrates zero delay control.

Example 4.12: Zero Delay Control

```
Initial
begin
x = 0;
y = 0;
end
initial
begin
#0 x = 1; //zero delay control
#0 y = 1;
end
```

In Example 4.12, four statements: x = 0, y = 0, x = 1, y = 1, are to be executed at simulation time 0. However, since x = 1 and y = 1 have #0, they will be executed last. Thus, at the end of time 0, x will have value 1 and y will have value 1. The order in which x = 1 and y = 1 are executed is not deterministic. Using #0 is not a recommended practice.

4.3.2 Event-Based Timing Control

An event is the change in the value on a register or a net. Events can be utilized to trigger execution of a statement or a block of statements. There are four types of event-based timing control: regular event control, named event control, event OR control, and level sensitive timing control.

Regular event control

The @ symbol is used to specify an event control. Statements can be executed on changes in signal value or at a positive or negative transition of the signal value. The keyword posedge is used for a positive transition, as shown in Example 4.13.

Example 4.13: Regular Event Control

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does a positive transition //
( 0 to 1,x or z, x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does a negative transition //
( 1 to 0,x or z, x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned to q at the positive edge //
of clock
```

Named event control

Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event. The event does not hold any data. A named event is declared by the keyword event. An event is triggered by the symbol ->. The triggering of the event is recognized by the symbol @.

Example 4.14: Named Event Control

```
//This is an example of a data buffer storing data after the last packet of data has arrived.
event received_data; //Define an event called received_data
always @(posedge clock) //check at each positive clock edge
begin
if(last_data_packet) //If this is the last data packet
->received_data; //trigger the event received_data
end
always @(received_data) //Await triggering of event received_data. When event is triggered, //
store all four packets of received data in data buffer use concatenation operator { }
data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

Event OR Control

Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements. This is expressed as an OR of events or signals. The list of events or signals expressed as an OR is also known as a sensitivity list. The keyword or is used to specify multiple triggers, as shown in Example 4.15.

Example 4.15 Event OR Control (Sensitivity List)

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
//Wait for reset or clock or d to change
begin
if (reset) //if reset signal is high, set q to 0.
q = 1'b0;
else if(clock) //if clock is high, latch input
q = d;
end
```

Sensitivity lists can also be specified using the "," (comma) operator instead of the or operator. Example 4.16 shows how the above example can be rewritten using the comma operator. Comma operators can also be applied to sensitivity lists that have edge-sensitive triggers.

Example 4.16: Sensitivity List with Comma Operator

```
//A level-sensitive latch with asynchronous reset
always @( reset, clock, d)
//Wait for reset or clock or d to change
begin
if (reset) //if reset signal is high, set q to 0.
q = 1'b0;
else if(clock) //if clock is high, latch input
q = d;
end
//A positive edge triggered D flipflop with asynchronous falling reset can be modeled as
//shown below
always @(posedge clk, negedge reset) //Note use of comma operator
if(!reset)
q <=0;
```

```
else
q <=d;
```

When the number of input variables to a combination logic block are very large, sensitivity lists can become very cumbersome to write. Moreover, if an input variable is missed from the sensitivity list, the block will not behave like a combinational logic block. To solve this problem, Verilog HDL contains two special symbols: @* and @(*). Both symbols exhibit identical behavior. These special symbols are sensitive to a change on any signal that may be read by the statement group that follows this symbol. Example 4.17 shows example of this special symbol for combinational logic sensitivity lists.

Example 4.17: Use of @* Operator

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e or f or g or h or p or m)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
//Instead of the above method, use @(*) symbol. Alternately, the @* symbol can be used
//All input variables are automatically included in the sensitivity list.
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m; end
```

4.3.3 Level-Sensitive Timing Control

Event control discussed earlier waited for the change of a signal value or the triggering of an event. The symbol @ provided edge-sensitive control. Verilog also allows level sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed. The keyword wait is used for level sensitive constructs.

```
always
wait (count_enable) #20 count = count + 1;
```

In the above example, the value of count_enable is monitored continuously. If count_enable is 0, the statement is not entered. If it is logical 1, the statement count = count + 1 is executed after 20 time units. If count_enable stays at 1, count will be incremented every 20 time units.

4.4 Conditional Statements

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed. Keywords if and else are used for conditional statements. There are three types of conditional statements. Usage of conditional statements is shown below.

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;
//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if (<expression>) true_statement ; else false_statement ;
//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

The <expression> is evaluated. If it is true (1 or a non-zero value), the true_statement is executed. However, if it is false (zero) or ambiguous (x), the false_statement is executed. The <expression> can contain any operators. Each true_statement or false_statement can be a single statement or a block of multiple statements. A block must be grouped, typically by using keywords begin and end. A single statement need not be grouped.

Example 4.18: Conditional Statement Examples

```
//Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;
//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
data_queue = data;
number_queued = number_queued + 1;
end
else
```

```
$display("Queue Full. Try again");
//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
y = x + z;
else if(alu_control == 1)
y = x - z;
else if(alu_control == 2)
y = x * z;
else
$display("Invalid ALU control signal");
```

4.5 Multiway Branching

In Conditional Statements, there were many alternatives, from which one was chosen. The nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the **case** statement.

4.5.1 case Statement

The keywords **case, endcase,** and **default** are used in the case statement.

```
case (expression)
alternative1: statement1;
alternative2: statement2;
alternative3: statement3;
...
...
default: default_statement;
endcase
```

Each of statement1, statement2 ….. , default_statement can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords **begin** and **end**. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default_statement is executed. The default_statement is optional. Placing of multiple default statements in one case statement is not allowed. The case statements can be nested. The following Verilog code implements the type 3 conditional statement in Example 4.18.

```
//Execute statements based on the ALU control signal
reg [1:0] alu_control;
...
...
case    (alu_control)
2'd0 : y = x + z; 2'd1
: y = x - z;
2'd2 : y = x * z;
default  :  $display("Invalid  ALU  control  signal");
endcase
```

Example 4.19: 4-to-1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input  i0,  i1,  i2,  i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals d0
: out = i0;
2'd1  :  out  =  i1;
2'd2  :  out  =  i2;
2'd3 : out = i3;
default:  $display("Invalid   control   signals");
endcase
endmodule
```

The case statement compares 0, 1, x, and z values in the expression and the alternative bit for bit. If the expression and the alternative are of unequal bit width, they are zero filled to match the bit width of the widest of the expression and the alternative.

4.5.2 casex, casez Keywords

There are two variations of the case statement, denoted by keywords, **casex** and **casez**.

- **casez** treats all z values in the case alternatives or the case expression as don't cares. All bit positions with z can also represented by ? in that position.

- **casex** treats all x and z values in the case item or the case expression as don't cares.

The use of casex and casez allows comparison of only non-x or -z positions in the case expression and the case alternatives. Example 4.21 illustrates the decoding of state bits in a finite state machine using a **casex** statement. The use of **casez** is similar. Only one bit is considered to determine the next state and the other bits are ignored.

Example 4.21: casex Use

```
reg [3:0] encoding;
integer state;
casex (encoding) //logic value x represents a don't care bit.
4'b1xxx : next_state = 3;
4'bx1xx : next_state = 2;
4'bxx1x : next_state = 1;
4'bxxx1 : next_state = 0;
default : next_state = 0;
endcase
```

Thus, an input encoding = 4'b10xz would cause next_state = 3 to be executed.

4.6 Loops

There are four types of looping statements in Verilog: while, for, repeat, and forever. The syntax of these loops is very similar to the syntax of loops in the C programming language. All looping statements can appear only inside an initial or always block. Loops may contain delay expressions.

4.6.1 While Loop

The keyword **while** is used to specify this loop. The **while** loop executes until the while expression is not true. If the loop is entered when the while-expression is not true, the loop is not executed at all. Each expression can contain the operators. Any logical expression can be specified with these operators. If multiple statements are to be executed in the loop, they must be grouped typically using keywords **begin** and **end**. Example 4.22 illustrates the use of the while loop.

Example 4.22: While Loop

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128. Display count variable.
integer count;
initial
begin
```

```
        count = 0;
        while (count < 128) //Execute loop till count is 127. exit at count 128.
        begin
        $display("Count = %d", count);
        count = count + 1;
end
end
//Illustration 2: Find the first bit with a value 1 in flag (vector variable)
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;
initial
begin
flag = 16'b 0010_0000_0000_0000;
i = 0;
continue = 'TRUE;
while((i < 16) && continue ) //Multiple conditions using operators.
begin
if (flag[i])
begin
$display("Encountered a TRUE bit at element number %d", i);
continue = 'FALSE;
end
i = i + 1;
end
end
```

4.6.2 for Loop

The keyword **for** is used to specify this loop. The **for** loop contains three parts:

- An initial condition
- A check to see if the terminating condition is true
- A procedural assignment to change value of the control variable

The counter described in Example 4.22 can be coded as a for loop (Example 4.23). The initialization condition and the incrementing procedural assignment are included in the **for** loop and do not need to be specified separately. Thus, the **for** loop provides a more compact loop structure than the while loop. Note, however, that the while loop is more general-purpose than the **for** loop. The **for** loop cannot be used in place of the **while** loop in all situations.

Example 4.23: For Loop

```
integer count; initial
for ( count=0; count < 128; count = count + 1)
$display("Count = %d", count);
```

for loops can also be used to initialize an array or memory, as shown below.

```
//Initialize array elements 'define MAX_STATES 32
integer state [0: 'MAX_STATES-1]; //Integer array state with elements 0:31
integer i;
initial
begin
for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0
state[i] = 0;
for(i = 1; i < 32; i = i + 2) //initialize all odd locations with 1
state[i] = 1;
end
```

for loops are generally used when there is a fixed beginning and end to the loop. If the loop is simply looping on a certain condition, it is better to use the while loop.

4.6.3 Repeat Loop

The keyword **repeat** is used for this loop. The **repeat** construct executes the loop a fixed number of times. A **repeat** construct cannot be used to loop on a general logical expression. A while loop is used for that purpose. A **repeat** construct must contain a number, which can be a constant, a variable or a signal value. However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.

The counter in Example 4.22 can be expressed with the repeat loop, as shown in Illustration 1 in Example 4.24. Illustration 2 shows how to model a data buffer that latches data at the positive edge of clock for the next eight cycles after it receives a data start signal.

Example 4.24: Repeat Loop

```
//Illustration 1: increment and display count from 0 to 127
integer count;
initial
begin
count = 0;
repeat(128)
begin
$display("Count = %d", count);
count = count + 1;
//Illustration 2: Data buffer module example.
module data_buffer(data_start, data, clock);
parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;
always @(posedge clock)
begin
if(data_start) //data start signal is true
begin
i = 0;
repeat(cycles) //Store data at the posedge of next 8 clock cycles
begin
@(posedge clock)
buffer[i] = data; //waits till next posedge to latch data
i = i + 1;
endmodule
```

4.6.4 Forever loop

The keyword **forever** is used to express this loop. The loop does not contain any expression and executes forever until the $finish task is encountered. The loop is equivalent to a while loop with an expression that always evaluates to true, e.g., while (1). A forever loop can be exited by use of the disable statement.

A forever loop is typically used in conjunction with timing control constructs. If timing control constructs are not used, the Verilog simulator would execute this statement infinitely without advancing simulation time and the rest of the design would never be executed. Example 4.25 explains the use of the forever statement.

Example 4.25: Forever Loop

```
//Example 1: Clock generation
//Use forever loop instead of always block
reg clock;
initial
begin
clock = 1'b0;
forever #10 clock = ~clock; //Clock with period of 20 units
end
//Example 2: Synchronize two register values at every positive edge of clock
reg clock;
reg x, y; initial
forever @(posedge clock) x = y;
```

4.7 Sequential and Parallel Blocks

Block statements are used to group multiple statements to act together as one. In this section we discuss the block types: sequential blocks and parallel blocks. We also discuss three special features of blocks: named blocks, disabling named blocks, and nested blocks.

4.7.1 Block Types

There are two types of blocks: sequential blocks and parallel blocks.

Sequential blocks

The keywords **begin** and **end** are used to group statements into sequential blocks. Sequential blocks have the following characteristics:

- The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution.
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

Two examples of sequential blocks are given in Example 4.26. Statements in the sequential block execute in order. In Illustration 1, the final values are x = 0, y= 1, z = 1, w = 2 at simulation time 0. In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.

Example 4.26: Sequential Blocks

```
//Illustration 1: Sequential block without delay
reg x, y;
reg [1:0] z, w;
initial
begin
        x = 1'b0;
        y = 1'b1;
        z = {x, y};
        w = {y, x};
end
//Illustration 2: Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;
initial
begin
        x = 1'b0; //completes at simulation time 0
        #5 y = 1'b1; //completes at simulation time 5
        #10 z = {x, y}; //completes at simulation time 15
        #20 w = {y, x}; //completes at simulation time 35
end
```

Parallel blocks

Parallel blocks are specified by the keywords **fork** and **join**. Parallel blocks have the following characteristics:

- Statements in a parallel block are executed concurrently.
- Ordering of statements is controlled by the delay or event control assigned to each statement.
- If delay or event control is specified, it is relative to the time the block was entered.

The fundamental difference between sequential and parallel blocks are that, all statements in a parallel block start at the time when the block was entered. Thus, the order in which the statements are written in the block is not important.

Let us consider the sequential block with delay in Example 4.26 and convert it to a parallel block. The converted Verilog code is shown in Example 4.27. The result of simulation remains the same except that all statements start in parallel at time 0. Hence, the block finishes at time 20 instead of time 35.

Example 4.27 Parallel Blocks

```
//Example 1: Parallel blocks with delay.
reg x, y;
reg [1:0] z, w;
initial
fork
        x = 1'b0; //completes at simulation time 0
        #5 y = 1'b1; //completes at simulation time 5
        #10 z = {x, y}; //completes at simulation time 10
        #20 w = {y, x}; //completes at simulation time 20
join
```

Parallel blocks provide a mechanism to execute statements in parallel. However, it is important to be careful with parallel blocks because of implicit race conditions that might arise if two statements that affect the same variable complete at the same time. Shown below is the parallel version of Illustration 1 from Example 4-26. Race conditions have been deliberately introduced in this example. All statements start at simulation time 0.

The keyword fork can be viewed as splitting a single flow into independent flows. The keyword join can be seen as joining the independent flows back into a single flow. Independent flows operate concurrently.

4.7.2 Special Features of Blocks

The three special features available with block statements are: nested blocks, named blocks, and disabling of named blocks.

Nested blocks

Blocks can be nested. Sequential and parallel blocks can be mixed, as shown in Example 4.28.

Example 4.28: Nested Blocks

```
//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. 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 4-29 shows naming of blocks and hierarchical naming of blocks.

Example 4.29: Named Blocks

```
//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 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. For C programmers, this is very similar to the break statement used to exit a loop.

Example 4.30: Disabling Named Block

```
// Illustration: Find the first bit with a value 1 in flag (vector variable)
reg (150) 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
$isplay ("Encountered a TRUE bit at element number %d", i);
disable blockl; //disable block1 because you found true bit
end
I = I + 1;
end
end
end
```

4.8 Example

4.8.1 4:1 multiplexer using behavioral case statement

```
module mux4_to_1 (out, in, 11, 12, i3, sl, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out; //output declared as register
```

```
//recompute the signal out if any input signal changes. All input signals that cause a
//recomputation of occur must go into the always @(...) sensitivity list.
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
case ({s1, s0))
2'b00; out = 10;
2 b01: out = il;
2'b10: out = 12;
2'b11: out = 13;
default: out = l'bx;
endcase
end
endmodule
```

4.8.1 4-bit counter using behavioral statement

```
module counter (Q, clock, clear);
output 3:0] Q;
input clock, clear;
reg (3:0) Q:
always ( posedge clear or negedge clock)
begin
if (clear)
Q <= 4'd0; // Nonblocking assignment are recommended for creating sequential logic such as
//flip-flops
else
Q <= Q + 1; // Modulo 16 is not necessary because Q is 4-bit value and wraps around.
end
endmodule
```

4.9 Tasks & Functions

Verilog provides tasks and functions to break up large behavioral designs into smaller pieces. Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design. Tasks are similar to SUBROUTINE and functions are similar to FUNCTION.

4.9.1 Differences between Tasks and Functions

Tasks and functions serve different purposes in Verilog. The differences are listed below:

Table 4.1: Tasks and Functions

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one input argument. They can have more than one input.	Tasks may have zero or more arguments of type input, output, or inout.
Functions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

Both tasks and functions must be defined in a module and are local to the module. Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments. Functions are used when common Verilog code is purely combinational, executes in zero simulation time, and provides exactly one output. Functions are typically used for conversions and commonly used calculations.

Tasks can have **input**, **output**, and **inout** arguments: functions can have **input** arguments. In addition, they can have local variables, registers, time variables, integers, seal, or events. Tasks or functions cannot have wires. Tasks and functions contain behavioral statements only. Tasks and functions do not contain **always** or **initial** statements but are called from **always** blocks, **initial** blocks, or other tasks and functions.

4.10 Tasks

Tasks are declared with the keywords **task** and **endtask**. Tasks must be used if any one of the following conditions is true for the procedure:

- There are delay, timing, or event control constructs in the procedure.
- The procedure has zero or more than one output arguments.
- The procedure has no input arguments.

4.10.1 Task declaration and Invocation

Task declaration and invocation syntax are as follows:

```
task declaration : :=
        task [ automatic ] task identifier ;
        { task_item_declaration)
        statement
        endtask
| task [ automatic ] task_identifier ( task port_list );
        { block_item_declaration)
        statement
        endtask
```

4.10.2 Task Example: Input and output arguments in task

```
//Define a module called operation that contains the task bitwise_oper
module operation
............
parameter delay = 10;
reg (15:01 A, B,
reg (15:0) AB_AND, AB_OR, AB_XOR;
always (A or B) // whenever A or B changes in value
begin
//invoke the task bitwise_oper. Provide 2 input arguments A, B. Expect 3 output arguments
//AB AND, AB_OR, AB XOR. The arguments must be specified in the same order as they
//appear in the task declaration
       bitwise_oper (AB_AND, AB_OR, AB_XOR, A, B);
end
............
//define task bitwise_oper
task bitwise_oper;
output (15:0) ab_and, ab_or, ab_xor; //outputs from the task
input (15:0] a, bi; //inputs to the task
begin
# delay ab_and = a & b;
ab_or =  a | b;
ab_xor = a ^ b;
end
```

```
endtask
……………..
endmodule
```

4.10.3 Automatic (Re-entrant) Tasks

Tasks are normally static in nature. All declared items are statically allocated and they are shared across all uses of the task executing concurrently. Therefore, if a task is called concurrently from two places in the code, these task calls will operate on the same task variables. It is highly likely that the results of such an operation will be incorrect.

To avoid this problem, a keyword **automatic** is added in front of the **task** keyword to make the tasks re-entrant. Such tasks are called *automatic tasks*. All items declared inside automatic tasks are allocated dynamically for each invocation. Each task call operates in an independent space. Thus, the task calls operate on independent copies of the task variables. This results in correct operation. Example 4.31 shows how an automatic task is defined and used.

Example 4.31: Re-entrant (Automatic) Tasks

```
// Module that contains an automatic (re-entrant) task. Only a small portion of the module
// that contains the task definition is shown in this example. There are two clocks. clk2 runs
// at twice the frequency of clk and is synchronous with clk.
model top;
reg (15:0] cd_xor, ef_xor; //variables in module top
reg (15:0) c, d, e, f; //variables in module top
………………..
task automatic bitwise_xor
output [15:0] ab_xor; //output from the task
input (15:0) a, b; //inputs to the task
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask
………………
………………
```

```
//These two always blocks will call the bitwise_xor task concurrently at each positive edge
//of clk. However, since the task is re-entrant, these concurrent calls will work correctly
always @ (posedge clk)
        bitwine_xor (ef_xor, e, f);
always @ (posedge clk2) // twice the frequency as the previous block
        bitwise_xor (cd_xor, c, d);
endmodule
```

4.11 Functions

Functions are declared with the keywords **function** and **endfunction**. Functions are used if all of the following condition are true for the procedure:

- There are no delay, timing, or event control constructs in the procedure.
- The procedure returns a single value.
- There is at least one input argument.
- There are no output or inout arguments
- There are no nonblocking assignments.

4.11.1 Function Declaration and Invocation

The syntax for functions follows:

function declaration : :=

function [**automatic**] [signed] [range_or_type]

function_identifier;

function_item_declaration { function_item_declaration }

function_statement

endfunction

| **function** [**automatic**] [**signed**] [range for type]

function_identifier (function_port_list);

block_item_declaration { block_item_declaration }

function_statement

endfunction

4.11.2 Function example: Parity calculation

```
// Define a module that contains the function calc_parity
module parity
reg [31:0] addr:
reg parity;
```

```
// compute new parity whenever address value changes
always @ (addr)
        parity = calc_parity (addr); //First invocation of calc_parity
        $display ( "Parity calculated = %b", calc_parity (addr); /Second invocation of calc_parity
end
..............
// Define the parity calculation function
function calc_parity;
input [31:0] address;
begin
// set the output value appropriately. Use the implicit internal register calc_parity.
calc_parity = ^address; //Return the xor of all address bits
end
endfunction
............
...........
Endmodule
```

4.11.3 Automatic (Recursive) Functions

Functions are normally used non-recursively. If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space.

However, the keyword **automatic** can be used to declare a recursive (automatic) function where all function declarations are allocated dynamically for each recursive call. Each call to an automatic function operates in an independent variable space. Automatic function items cannot be accessed by hierarchical references. Automatic functions can be invoked through the use of their hierarchical name. Example 4.31 shows how an automatic function is defined to compute a factorial.

Example 4.31: Recursive (Automatic) Function

```
//Define a factorial with recursive function
module top;
// Define the function
function automatic integer factorial;
input (31:0) oper;
```

```
integer i;
begin
if (operand > 2)
factorial = factorial (oper -1) * oper; //recursive call
else
factorial =1;
end
endfunction
//Call the function
integer result;
initial
begin
        result = factorial (4); // Call the factorial of 7
        $display ("Factorial of 4 is % 0d", result); //Displays 24
end
............
endmodule
```