Module – 4

Behavioral Modeling

- With the increasing complexity of digital design, it has become vitally important to make wise design decisions early in a project.
- Designers need to be able to evaluate the trade-offs of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware.
- Thus, architectural evaluation takes place at an algorithmic level where the designers do not necessarily think in terms of logic gates or data flow but in terms of the algorithm they wish to implement in hardware.
- They are more concerned about the behavior of the algorithm and its performance. Only after the high-level architecture and algorithm are finalized, designers start focusing on building the digital circuit to implement the algorithm.
- Verilog provides designers the ability to describe design functionality in an algorithmic manner. In other words, the designer describes the behavior of the circuit. Thus, behavioral modeling represents the circuit at a very high level of abstraction.

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 statements can appear only inside these structured procedure statements.
- Verilog is a concurrent programming language. Activity flows in Verilog run 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.

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 using the keywords begin and end. If there is only one behavioral statement, grouping is not necessary.
- For example,

```
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
endmodule
```

- In the above example, the three initial statements start to execute in parallel at time 0.
- If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, 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;

- 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 Port / Data Declaration and Initialization

- The combined port/data declaration can also be combined with an initialization as shown in the following program

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

Combined ANSI C Style Port Declaration and Initialization

- ANSI C style port declaration can also be combined with an initialization as shown in the following program

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

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.
- For example,

```
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 the above example, 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.
- Consider the following example,

```
module always_example();
reg clk, reset, q_in, data;
always @ (posedge clk)
begin
    if (reset)
            data = 0;
    else
            data = q_in;
end
endmodule
```

- In the above example of the always block, when the trigger event occurs, the code inside begin and end is executed; then once again the always block waits for next event triggering. This process of waiting and executing on event is repeated till simulation stops.

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.
- The syntax for the simplest form of procedural assignment is

  < lvalue > = < expression >
- The left-hand side of a procedural assignment < lvalue > can be one of the following:
  i. A reg, integer, real, or time register variable or a memory element
  ii. A bit select of these variables (e.g., addr [0])
  iii. A part select of these variables (e.g., addr [7:4])
  iv. 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 supported by Verilog can be used in behavioral expressions.
- There are two types of procedural assignment statements
  i. blocking
  ii. nonblocking

Blocking assignments

- Blocking assignment statements are executed in the order they are specified in a sequential block.
- The execution of next statement begins only after the completion of the present blocking assignments.
- A blocking assignment will not block the execution of the next statement in a parallel block.
- The blocking assignments are made using the operator =.
- For example,

```
initial
begin
        a = 1;
        b = #5 2;
        c = #2 3;
end
```

- In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 7.

Non-blocking assignments

- The nonblocking assignment allows assignment scheduling without blocking the procedural flow.
- The nonblocking assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other.
- Nonblocking assignments are made using the operator <=.
- Note that <= is same for less than or equal to operator, so whenever it appears in a expression it is considered to be comparison operator and not as non-blocking assignment.
- For example,

```
initial
begin
        a <= 1;
        b <= #5 2;
        c <= #2 3;
end
```

- In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 2 (because all the statements execution starts at time 0, as they are non-blocking assignments.

Timing Controls

- Timing controls provide a way to specify the simulation time at which procedural statements will execute.
- There are three methods of timing control
    i. Delay-based timing control
    ii. Event-based timing control
    iii. Level-sensitive timing control

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 #. The delay syntax is

    # <delay_value>

  delay_value can be specified by a number, identifier, or a min_typ_max_expression
- There are three types of delay control for procedural assignments
    a. regular delay control
    b. intra-assignment delay control
    c. 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.
- For example,

```
//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
        #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
```

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.
- For example,

```
    //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;
    end
```

- Regular delays defer the execution of the entire assignment. Intraassignment delays compute the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.
- Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.

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.
- For example,

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

- In the above example, 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.

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
    i. regular event control
    ii. named event control
    iii. event OR control

iv. 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; negedge is used for a negative transition.
- For example,

```
@ (clock)
        q = d;          // q = d is executed whenever signal clock changes value
@(posedge clock)
        q = d;          // q = d is executed during positive transition
@ (negedge clock)
        q = d;          // q = d is executed during negative transition
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 and the event is triggered by the symbol ->. The triggering of the event is recognized by the symbol @.
- For example,

```
// 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
```

```
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
- For example,

```
// 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.

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.
- For example,

```
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 executed. 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.

Conditional Statements

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

```
// 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.
- For example,

```
// 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");
```

Multiway Branching

- In type 3 conditional statement, 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.

case Statement

- The syntax of the case statement is formed by the keywords case, endcase, and default.

```
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.
- For example,

```
// 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
```

- 4-to-1 multiplexer can be modelled using case statements.

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @ (s1 or s0 or i0 or i1 or i2 or i3)
begin
    case ({s1, s0}) // Switch based on concatenation of control signals
    2'd0 : out = i0;
    2'd1 : out = i1;
    2'd2 : out = i2;
    2'd3 : out = i3;
    default: $display ("Invalid control signals");
    endcase
end
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.
- For example, if we define a 1-to-4 demultiplexer for which all the outputs should be specified and it can be done through begin and end statements.

```
module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);
// Port declarations from the I/O diagram
output out0, out1, out2, out3;
```

```
reg out0, out1, out2, out3;
input in;
input s1, s0;
always @(s1 or s0 or in)
begin
        case ({s1, s0})          // Switch based on control signals
        2'b00 : begin out0 = in; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz; end
        2'b01 : begin out0 = 1'bz; out1 = in; out2 = 1'bz; out3 = 1'bz; end
        2'b10 : begin out0 = 1'bz; out1 = 1'bz; out2 = in; out3 = 1'bz; end
        2'b11 : begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = in; end
        default: $display ("Unspecified control signals");
        endcase
end
endmodule
```

casex, casez Keywords

- There are two variations of the case statement. They are 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.
- For example,

```
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.

Loops

- There are four types of looping statements in Verilog: while, for, repeat, and forever.
- All looping statements must appear only inside an initial or always block. Loops may contain delay expressions.

While Loop

- The keyword **while** is used to specify this loop. The while loop is executed until the while-expression is true.
- If 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.
- For example,

```
// Increment count from 0 to 127. Exit at count 128. Display the count variable.
    integer count;
    initial
    begin
        count = 0;
        while (count < 128)          // Execute loop till count is < 127.
        begin
            $display("Count = %d", count);
            count = count + 1;
        end
    end
```

For Loop

- The keyword **for** is used to specify this loop. The for loop contains three parts:
    i. An initial condition

ii. A check to see if the terminating condition is true

iii. A procedural assignment to change value of the control variable

- 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. 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.
- For example,

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

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.
- For example,

```
// 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;
                end
```

```
end
```

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 terminated by disable statement.
- A forever loop is typically used in generation of periodic signals
- For example,

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

Sequential and Parallel Blocks

- Block statements are used to group multiple statements to act together as one. Keywords begin and end is used to group multiple statements.
- There are two types of blocks
    i. Sequential blocks
    ii. Parallel blocks

Sequential blocks

- The keywords begin and end is used to group statements into sequential blocks.

- 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 (except for nonblocking assignments with intra-assignment timing control).
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.
- For example in the 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.

```
// 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 provide a mechanism to execute statements in parallel and are specified by keywords fork and join.

- 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.
- 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.

```
// 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
```

- 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.
- All the independent flows operate concurrently.

Nested blocks

- Blocks can be nested. Sequential and parallel blocks can be mixed, as shown below

```
// 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.

```
// 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
```

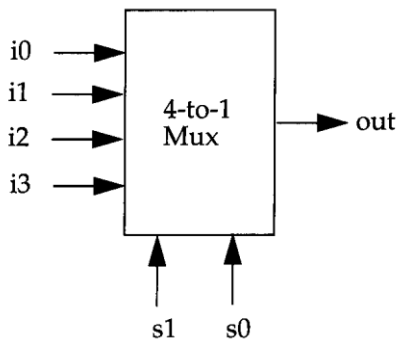
Disabling Named Blocks

- 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.

```
// Illustration: Find the first bit with a value 1 in flag (vector variable)
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
```

Verilog HDL Code to design 4-to-1 Multiplexer



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Fig.4.1: 4:1 Multiplexer

- Behavioral description of 4-to-1 multiplexer with the case statement is given below

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
        output out;
        input i0, i1, i2, i3;
        input s1, s0;
```

```
// output declared as register
        reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
        begin
                case ({s1, s0})
                2'b00: out = i0;
                2'b01: out = i1;
                2'b10: out = i2;
                2'b11: out = i3;
                default: out = 1'bx;
                endcase
        end
endmodule
```

Verilog HDL Code to design 4-bit Counter

- Behavioral description of Ripple Counter is given below

```
module counter(clock, reset, q);
// I/O ports
        input clock, clear;
        output [3:0] q;
// output defined as register
        reg [3:0] q;
always @( reset or negedge clock)
begin
        if (reset)
                q <= 4'd0;
        else
                q <= q + 1;
end
endmodule
```