Event Control

- Fixed delay operator (#) is used only when the required delay is known in advance.
- If we do not know when the signal is going to change, the event control operator @ is used.
- @ may be described as *wait for event*.
- @ can also be used with *event data type*.
- There are three keywords available in verilog for working with event control.
  - ✓ *posedge*
  - ✓ *negedge*
  - ✓ *or*
  - *posedge* and *negedge* used with one variable (edge sensitive *clock*).
  - *or* is used with multiple variables.

Example D-flip flop using event control

```
module dff (clock, d, q);
   input clock;
   input d;
   output q;
   reg q;

 always @( posedge clock)
       q = d;

 endmodule
```

Example using event data type and event control

```
module show_event;

  event a_event, b_event;

  always @ a_event
     begin
        $display($time, “ns event a is triggered”);
        -> b_event;
   end

always @ b_event
    $display(“$time, “ns event b is triggered);

 initial
   begin
     #2 -> a_event
     #5 -> b_event;
     #1 -> a_event;
   end

endmodule
```

Note: “ -> “ is a signal event operator

- Generally the symbol @ is used as edge sensitive with respect to clock signal.
- Verilog also allows level sensitive timing control, i.e., the ability to wait for a certain condition to be true before a statement or block of statements is executed.
- The keyword ***wait*** is used for level sensitive constructs.

Example

```
always
  wait (clock)
        #1000 count = count + 1;
```

Control and Loop statements

- The *if* statement

Syntax

```
if(condition)
    statement
else
    statement
```

If more than one statement appears within the body of if then you need to use a ***begin .. end*** or ***fork .. join*** block.

Example

```
module dut (a,b,c,y);
    input a,b,c;
    output y;
    reg y;
always @(a or b or c)
  if (c)
     y = a;
  else
     y = b;
endmodule
```

Caution: Omitting else will result in unintentional latch during synthesis !

- The **case** statement

Three forms of **case** statements

case
casex
casez

Syntax

```
case (expression)
        case index 1 = expression1;
        case index 2 = expression2;
           ..........................
           ..........................
        default  : expression;
endcase
```

Example: ***Up-down Counter using case***

```
module counter (clock, reset, load, up, load_data, count);
        input clock, reset, load, up;
        input[3:0] load_data;
        output[3:0] count;
        reg[3:0] count ;
always @ (posedge clock)
        case ({reset, load, up })    //concatenate control signals
                3'b100: count = 4'b0;
                3'b010: count = load_data;
                3'b001: count = count+1;
                3'b000 : count = count -1;
                default : count = 4'bx;
         endcase
 endmodule
```

Example: ***Up-down Counter using casez***

```
module counter (clock, reset, load, up, load_data, count);
    input clock, reset, load, up;
    input[3:0] load_data;
    output[3:0] count;
    reg[3:0] count;
always @ (posedge clock)
      casez ({reset, load, up })    //concatenate control signals
            3'b1zz: count = 4'b0;
            3'b01?: count = load_data;
            3'b001: count = count+1;
            3'b000 : count = count -1;
            default : count = 4'bx;
       endcase
 endmodule
```

- ***Difference between casez, casex, and case***

  casez treats – z and ? as don’t cares – matches z and ? to either 1 or 0.

  casex treats – z and x as don’t cares – matches z and x to either 1 or 0.

  case treats – z and x as it is.

- case equality (===) and case inequality (!==) operator

  Suppose, vector1 = 4’b1x01, vector2 = 4’b1101

  if(vector1 === vector2) – results a logic LOW

  if(vector1 == vector2) – results a x

Summary of case values and match per Case type.

Value	case	casez	casex
0	0	0	0
1	1	1	1
x	x	x	0 1 x z
z	z	0 1 x z	z
?	unused	0 1 x z	unused
default	0 1 x z	0 1 x z	0 1 x z

- The ***while*** loop
  - The ***while*** loop is executed as long as its condition is true.

Syntax:

while (expression)

statement

Note: If more than one statement appears within the body of while loop then you need to use a ***begin .. end*** or ***fork .. Join*** block.

for loop

Syntax

```
for( index = initial value;  index <= final_value ; index = index + 1)
        statement
```

Note: If more than one statement appears within the body of for loop then you need to use a *begin .. end* or *fork .. Join* block.

Example

```
module hello_disp ;
integer i ;
initial
  for (i = 0; i < 4 ; i = i + 1)
        $display("Hello Verilog ");
endmodule
```

The *repeat* loop

Syntax

repeat (Numerical expression)

statement ;

Example

```
repeat (5)
    $display ( “Hello Verilog”);
```

forever loop

The forever loop executes continuously until the end of a simulation is requested by a $finish.

The forever loop must be used with a timing control to limit its execution, otherwise its statement would be executed continuously at the expense of the rest of the design.

Example

```
reg clock;
initial begin
   clock = 1'b0;
   forever #5 clock = ~clock; // the clock flips every 5 time units.
end
initial #2000 $finish;
```

Tasks

- A task is defined in a module and is invoked when its name is called in procedural code.
- A task has access to all data objects.
- Tasks can have delays, wait statements and event controls.
- Tasks may contain one or more statements.
- If it contains more than one statement, *begin .. end* or *fork ..join* block must be used

Syntax

```
task task_name;
        declarations;
       statement;
endtask
```

```
//adder using task
module adder_task(a, b, c, sum, carry);

  // IO Ports
  input  wire [3:0] a, b;
  input  wire       c;
  output reg  [3:0] sum;
  output reg        carry;

  always @(a, b, c)
    adder(a, b, c, sum, carry);

  //task
  task adder
    input [3:0]  x, y;
    input        z;
    output [3:0] s;
    output       cout;
    begin
      s =  x ^ y ^ z;
      cout = (x&y) | (y&z) | (x&z);
    end
  endtask

endmodule
```

Functions

- Functions must return a value.
- Functions take zero time.
- Functions can not contain delay or event control.
- Function returns a value by assigning a value back to a pseudo-variable represented by the function name.
- When a function is declared it is necessary to declare the type and size of the return type (Default is 1-bit).

Syntax

```
function return_type function_name;
    declarations
    statement
endfunction
```

- Note: Name of function should appear at least once on the left hand side of the expression.
- If more than one statement present within the body of a function then they must appear within begin..end or fork..join block.

Example *Counting number of 1's.*

```
module countbits;
    initial
        $display(" The number of 1's  = %d", count_bits(88));

    function integer count_bits;
        input [15:0] a;
        begin
            count_bits = 0;
            while (a) begin
                if( a[0] )
                    count_bits = count_bits + 1;
                    a = a >> 1;
                end
        end
    endfunction
endmodule
```

Ex: parity_using_function

```
module parity_using_function (
  data_in     , //  8 bit data in
  parity_out   //  1 bit parity out
);
output  parity_out ;
input [7:0] data_in ;
wire parity_out ;

function parity;
  input [7:0] data;
  integer i;
  begin
    parity = 0;
    for (i = 0; i < 7; i = i + 1) begin
      parity = parity ^ data[i];
    end
  end
endfunction

always @ (data_in)
   begin
     parity_out = parity(data_in);
   end
   endmodule
```

Example: Mux with function and continuous assignment

```
module muxfunct(out, in1, in2, in3, in4, sel);
output[7:0] out;
input[7:0] in1, in2, in3, in4;
input[1:0] sel;

assign out = mux(sel, in1, in2, in3, in4);

function [7:0] mux ;
    input [1:0] sel ;
    input [7:0] in1, in2, in3, in4;
    case(sel)
      2'b00 : mux = in1;
      2'b01 : mux = in2;
      2'b10 : mux = in3;
      2'b11 : mux = in4;
      default : mux = 7'bz;
    endcase
 endfunction
endmodule
```

Example: Adder using function

```
//adder using functions
module adder_function(a, b, c, sum, carry);

  // IO Ports
  input  wire [3:0] a, b;
  input  wire       c;
  output wire [3:0] sum;
  output wire       carry;

  assign sum   = sum_ftn(a, b, c);
  assign carry = carry_ftn(a, b, c);

  //functions return only one value
  function sum_ftn
    input x ,y,z;
    begin
      sum_ftn = x ^ y ^ z;
    end

  function sum_ftn
    input x ,y,z;
      begin
        carry_ftn = (x&y) | (y&z) | (x&z);
    end

endmodule
```

Parameterized modules

- Parameters are often used to describe the word size of the module, the number of words in a memory, or delays.
- Parameters are used as constants
- The only legal way to change the value of a constant is with the defparam statement.
- Parameters and defparam statements are evaluated only during compilation
- Parameters can be changed during the simulation time.
- Default width of parameter type variable is 32-bits.
- Width can be changed by specifying a range.

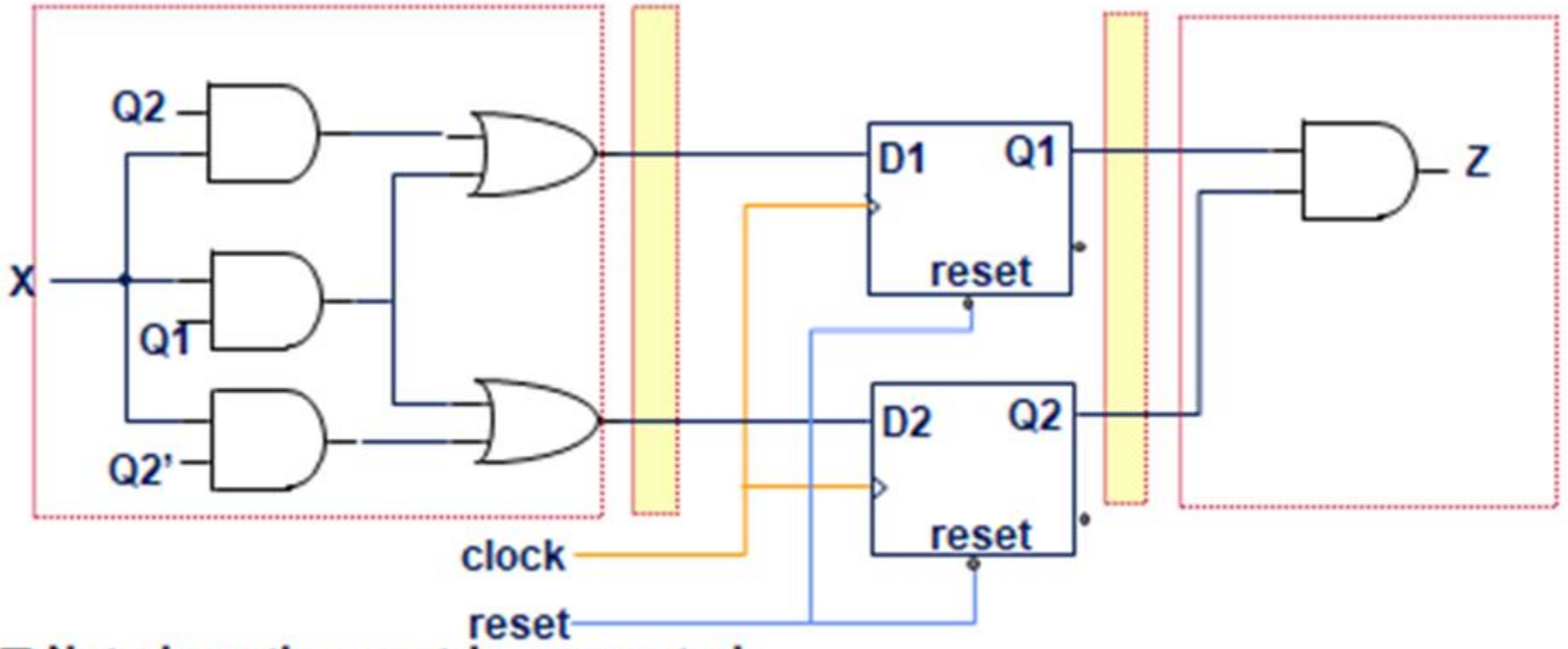
Example using parameter statement

```
/* Mux with parameterized width and number of inputs */
    module mux(z,sel,D);
        parameter N = 8;
        parameter M = 4;
        parameter S = 2;
        parameter w = M *N;
        `define DTOTAL  w-1 : 0
        ‘define DWIDTH  N-1 : 0
        ‘define SELW       S-1 :0
        ‘define WORDS   M-1:0
        input   [‘DTOTAL]   D;
        input   [‘SELW]  sel;
        output [‘DWIDTH] z;
```

```
module my_module ;
integer i;
reg[‘DWIDTH] tmp, z;

always @ ( sel or D) begin
for( i=0; i < N;  i=i+1)
     tmp[i] = D [ N*sel + i ];
     z = tmp ;
end
endmodule
```

Moore Machine — 101 Detector



- Note how the reset is connected
  - Reset will make both of the FFs zero, thus putting them into state A.
  - Most FFs have both reset and "preset" inputs (preset sets the FF to one).
  - The reset connections (to FF reset and preset) are determined by the state assignment of the reset state.

```
// sequence detector Mealy Machine - 101
module seq_detector(in, clk, rst, out);
  input in, clk, rst;
  output reg out = 1'b0;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;

  reg [1:0] ps;
  reg [1:0] ns;
 // Clocked Present State Logic
  always @(posedge clk) begin
    if (rst)
      ps <= S0;
    else
      ps <=ns;
  end
// Combinational next state and output Logic
  always @(in, ps) begin
    case(ps)
      S0: if(in) begin
            ns = S1;
            out = 1'b0;
          end
          else begin
            ns = S0;
            out = 1'b0;
          end
```

```
      S1: if(in) begin
            ns = S1;
            out = 1'b0;
          end
          else begin
            ns = S2;
            out = 1'b0;
          end

      S2: if(in) begin
            ns = S1;
            out = 1'b1;
          end
          else begin
            ns = S0;
            out = 1'b0;
          end
    endcase
  end
endmodule
```

```
module test_seq_detector;
  reg in, clk, rst;
  wire out;

  seq_detector dut(.*);

  initial begin
    in = 1'b0;
    clk = 1'b0;
    rst = 1'b1;
  end

  always #5 clk = ~clk;

  initial begin
    #6 rst = 1'b0;
    #6 in = 1'b1;
    #6 in = 1'b0;
    #8 in = 1'b1;
    #10 in = 1'b0;
  end

  initial begin
    $dumpfile("testbench.vcd");
    $dumpvars;
    #200 $finish;
  end
endmodule
```