Chapter 4: Behavioral Modeling

Prof. Soo-Ik Chae

Objectives

After completing this chapter, you will be able to:

- Describe the behavioral modeling structures
- Describe procedural constructs
- Understand the features of initial blocks
- Understand the features of always blocks
- Distinguish the differences between blocking and nonblocking assignments
- Understand the features of timing controls
- Understand the features of selection constructs
- Understand the features of loop constructs

Behavioral Modeling Structures

- Assignments:
  - continuous assignment (assign expression) (Dataflow modeling)
  - blocking assignment (=)
  - nonblocking assignment (<=)
  - procedural continuous assignment
    - assign … deassign
    - force … release
- Selection structures:
  - if … else
  - case (case, casex, casez)
- Iterative structures:
  - repeat
  - for
  - while
  - forever

Procedural Constructs

❖ Procedural Constructs

- initial statements are used to initialize variables and set values into variables or nets.
- always statements are used to model the continuous operations required in the hardware modules.
- Each always statement corresponds to a piece of logic circuit.
- initial and always statements:
  - Each represents a separate activity flow.
  - Each activity starts at simulation 0.
  - They cannot be nested.

Procedural Constructs

- All procedures in the Vrilog HDL are specified within one of the follwing four statements:
  - initial construct
  - always construct
  - Task
  - Function
- Tasks and functions are procedures that are enabled from one or more places in other procedures.

initial Statements (p. 143 in LRM)

- An initial block
  - is composed of all statements inside an initial statement.
  - executes exactly once during simulation.
  - is used to initialize signals, or monitor waveforms, etc.
  - starts to execute concurrently at simulation time 0 and finishes execution independently when multiple initial blocks exist.

```
reg  x, y, z;
initial
   begin      // complex statement
            x = 1`b0;  y = 1`b1;  z = 1`b0;
   #10      x = 1`b1;  y = 1`b1;  z = 1`b1;
   end
initial      x = 1`b0;  // single statement
```

initial Statements

- Combined variable declaration and initialization

```
reg  clk;            // regular declaration
initial clk = 0;

reg clk = 0;         // can be used only at module level
```

- Combined port/data declaration and initialization

```
module adder(x, y, c , sum, c_out);
input [3:0]          x, y;
input                c_in;
output reg [3:0]     sum = 0;
output reg           c_out = 0;
```

```
module adder(input [3:0]          x, y,
             input                c_in,
             output reg [3:0]     sum = 0,
             output reg           c_out = 0
);  // ANSI C style
```

always Statements (p. 144 in LRM)

- An always block
  - consists of all behavioral statements inside an always statement.
  - starts at simulation time 0.
  - executes continuously during simulation.
  - is used to model a block of activity being repeated continuously in a digital circuit.

```
reg  clock;                    // a clock generator
initial  clock = 1`b0;         // initial clock = 0

always  #5 clock = ~clock;  // period = 10
```

Q: What will be happened in the following statement?

```
always  begin
initial  clock = 1`b0;
   #5 clock =  ~clock;
end
```

Procedural Assignments

- The bit widths of both left-hand and right-hand sides need not be the same.
  - The right-hand side is truncated if it has more bits.
    - by keeping the least significant bits
  - The right-hand side is filled with zeros in the most significant bits when it has fewer bits.
- Two types of procedural assignments:
  - blocking: using the operator "="
  - nonblocking: using the operator "<="

Procedural Assignments

- Procedural assignments
  - must be placed inside initial or always blocks.
  - update values of variable data types (reg, integer, real, or time.) under the control of the procedural flow constructs that surround them.

    variable_lvalue = [timing_control] expression

    [timing_control] variable_lvalue = expression

    variable_lvalue <= [timing_control] expression

    [timing_control] variable_lvalue <= expression

  - variable_lvalue can be:
    - a reg, integer,
    - real, time, or
    - a memory word,
    - a bit select, a part select, a concatenation of any of the above.

Blocking Assignments

- Blocking assignments
  - are executed in the order they are specified.
  - use the "=" operator.

```
// an example illustrating blocking assignments
module blocking;
reg  x, y, z;
// blocking assignments
initial begin
   x = #5 1'b0;    // x will be assigned 0 at time 5
   y = #3 1'b1;    // y will be assigned 1 at time 8
   z = #6 1'b0;    // z will be assigned 0 at time 14
end
endmodule
```

Blocking Assignments

```
module twos_adder_behavioral(x, y, sub_or_add, sum, c_out);
// I/O port declarations
input  [3:0] x, y;            // declare as a 4-bit array
input  sub_or_add;            //  add if 0, subtract if 1
output reg  [3:0] sum;        // declare as a 4-bit array
output reg       c_out;
reg       [3:0]       t;      // outputs of xor gates
// specify the function of a two's complement adder
always @(x, y, sub_or_add) begin  // define two's complement adder function
    t = y ^ {4{sub_or_add}};  // What is wrong with:  t = y ^ c_in ?
    {sub_or_add, sum} = x + t + sub_or_add;
end
endmodule
```

A reg variable doe not correspond to a memory element after synthesizing the circuit.

Q: What will happen if we change blocking operators (=) into nonblocking operators (<=)?

Nonblocking Assignments (p. 118 in LRM)

- Nonblocking assignments
  - are executed without blocking the other statements.
  - use the <= operator.
  - are used to model several concurrent data transfers.

```
// an example illustrating nonblocking assignments
module nonblocking;
reg  x, y, z;
// nonblocking assignments
initial begin
   x <= #5 1'b0;  // x will be assigned 0 at time 5
   y <= #3 1'b1;  // y will be assigned 1 at time 3
   z <= #6 1'b0;  // z will be assigned 0 at time 6
end
endmodule
```

Nonblocking Assignments

```
// an example of right-shift register without reset.
module shift_reg_4b(clk, din, qout);
input  clk;
input  din;
output reg [3:0] qout;
// the body of a 4-bit shift register
always @(posedge clk)
    qout <= {din, qout[3:1]};  // Right shift
endmodule
```

Q: What will happen if we change nonblocking operator (<=) into blocking operator (=)?

Answer: same as before. Why ?

Nonblocking Assignments

```
// an example of right-shift register without reset.
module shift_reg_4b(clk, din, qout);
input  clk;
input  din;
output reg [3:0] qout;
// the body of a 4-bit shift register
always @(posedge clk)
    qout = {din, qout[3:1]};  // Right shift
endmodule
```

```
fork qout[3]=din; qout[2]=qout[3]; qout[1]=qout[2]; qout[0]=qout[1]; join
// vector blocking assignment: parallel assignment
```

Right-shift register with reset

```
// an example of right-shift register
module shift_reg_4b(clk, din, qout, dout, reset);
input  clk, reset;
input  din;
output dout;
output reg [3:0] qout;
// the body of a 4-bit shift register
assign dout = qout[0];
always @(negedge reset or posedge clk)
    begin
        if (reset=1b'0) qout <= 4'b0;
        else qout <= {din, qout[3:1]};  // Right shift
   end
endmodule
```

Race Conditions

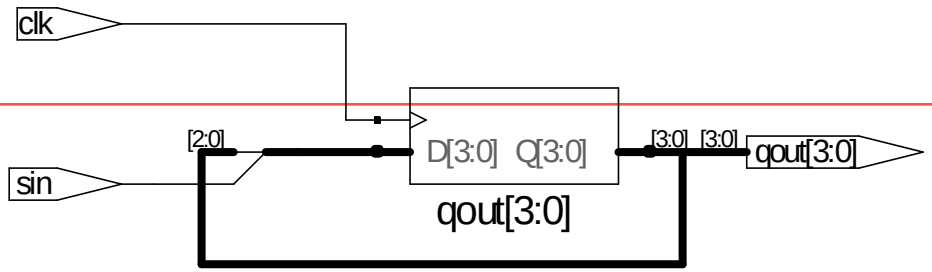
```
// using blocking assignment statements
always @(posedge clock)   // has race condition
        x = y;
always @(posedge clock)
        y = x;

// using nonblocking assignment statements
always @(posedge clock)   // has no race condition
        x <= y;
always @(posedge clock)
        y <= x;
```

Note that: In simulation stage, three steps are performed for nonblocking statements:

1. Read the values of all right-hand-side variables;
2. Evaluate the right-hand-side expressions and store in temporary variables;
3. Assign the values stored in the temporary variables to the left-hand-side variables.

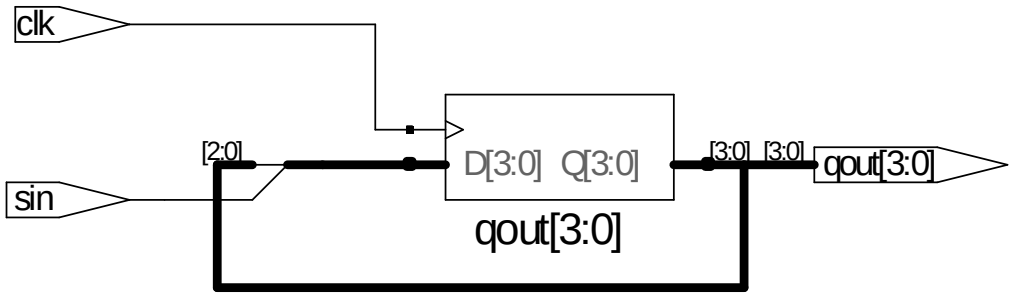
Blocking vs. Nonblocking Assignments

```
// shift register module example ---a correct implementation
module shift_reg_nonblocking(clk, sin, qout);
input  clk;
input  sin;  // serial data input
output reg [3:0] qout;
// The body of a 4-bit shift register
always @(posedge clk)
   begin                        // using nonblocking assignments
     qout[0] <= sin;
     qout[1] <= qout[0];        // better to use qout <= {qout[2:0], sin};
     qout[2] <= qout[1];
     qout[3] <= qout[2];
   end
endmodule
```



Nonblocking Assignments

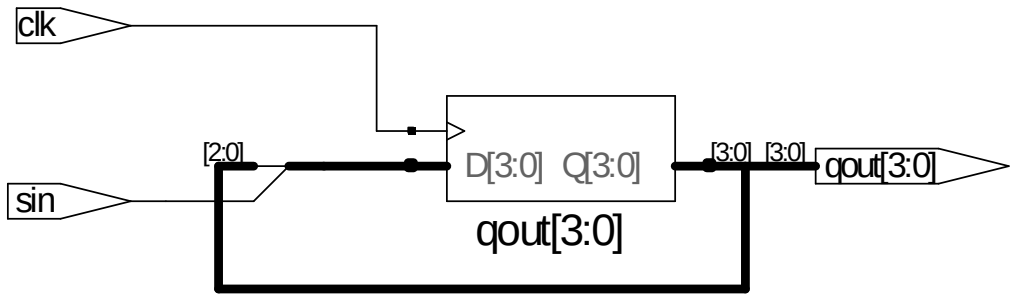
```
// shift register module example --- an correct implementation
module shift_reg_blocking(clk, sin, qout);
input  clk;
input  sin;  // serial data input
output reg [3:0] qout;
// The body of a 4-bit shift register
always @(posedge clk)
   qout <= {qout[2:0], sin}; // using blocking assignments
endmodule
```



Blocking Assignments

```
// shift register module example --- an correct implementation
module shift_reg_blocking(clk, sin, qout);
input  clk;
input  sin;  // serial data input
output reg [3:0] qout;
// The body of a 4-bit shift register
always @(posedge clk)
   begin      // using blocking assignments
      qout[3] = qout[2];
      qout[2] = qout[1];
      qout[1] = qout[0];
      qout[0] = sin;
end
endmodule
```

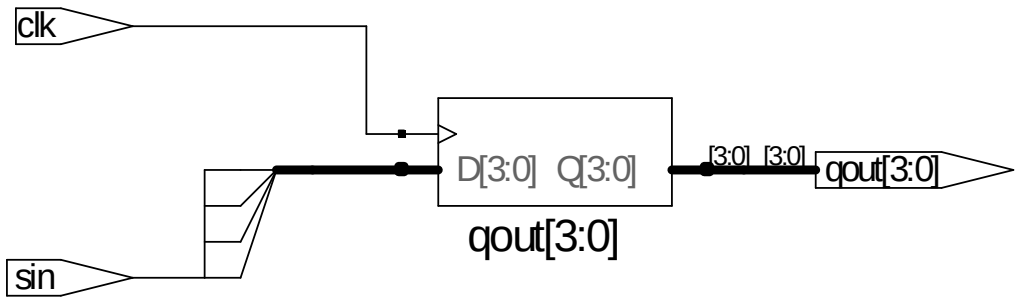
// four bit-select blocking assignments are sequentially executed.



Blocking Assignments

```
// shift register module example --- an incorrect implementation
module shift_reg_blocking(clk, sin, qout);
input  clk;
input  sin;  // serial data input
output reg [3:0] qout;
// The body of a 4-bit shift register
always @(posedge clk)
  begin      // using blocking assignments
    qout[0] = sin;
    qout[1] = qout[0];
    qout[2] = qout[1];
    qout[3] = qout[2];
  end
endmodule
```

// four bit-select blocking assignments are sequentially executed.

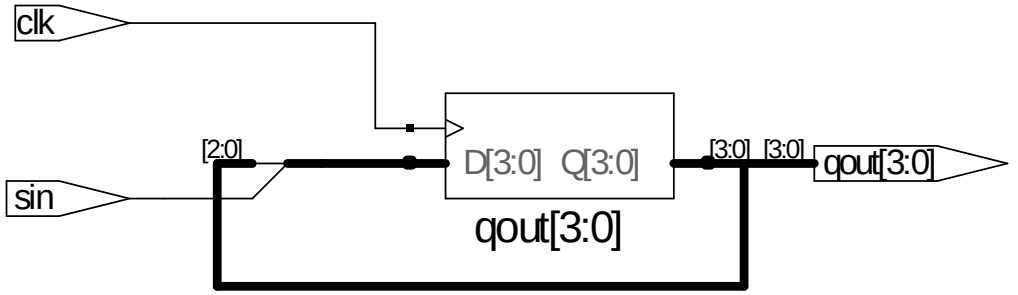


Reverse order

Blocking Assignments

```
// shift register module example --- an correct implementation
module shift_reg_blocking(clk, sin, qout);
input  clk;
input  sin;  // serial data input
output reg [3:0] qout;
// The body of a 4-bit shift register
always @(posedge clk)
  qout = {qout[2:0], sin}; // using blocking assignments
endmodule
```

// a vector blocking assignment is currently executed bit by bit.



Note that: When using qout = {qout[2:0], sin} instead of qout <= {qout[2:0], sin}, the result will not be different.

Blocking vs. Nonblocking Assignments

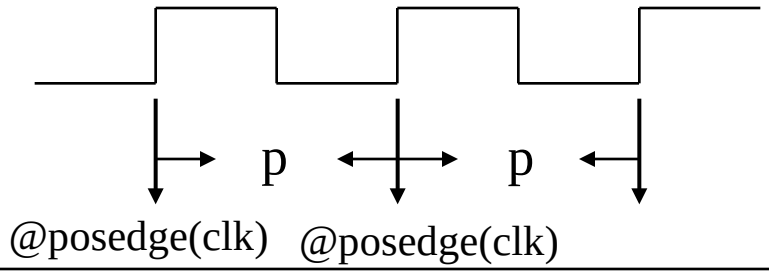
- Consider the difference between the following two always blocks. Assume that the value of count is 1 and finish is 0 before entering the always block at time n.

```
always @(posedge clk) begin: block_a
    count = count – 1;
    if (count == 0) finish = 1;
end
```

Result:
@[n,n+p): count=0, finish = 1.
@[n+p,n+2p): count=-1, finish=1.

```
always @(posedge clk) begin: block_b
    count <= count – 1;
    if (count == 0) finish <= 1;
end
```

Result:
@[n,n+p): count=0, finish = 0.
@[n+p,n+2p): count=-1, finish=1.



Compare with Figure 4.3 (p. 116 in Lin)

Coding Style for Blocking / Nonblocking Assignments

- Coding style: In the always block
  - Use nonblocking operators (<=) when it is a piece of sequential logic;
    - Otherwise, the result of RTL behavioral may be inconsistent with that of gate-level.
  - Use blocking operators (=) when it is a piece of combinational logic.

Procedural continuous assignments

- The assign statement shall override all procedural assignments to a variable.
- The deassign statement shall end a procedural assignment to a variable.
- Asynchronous clear/preset D-type edge-triggered flip-flop

```
module dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
req q;

always @ (clear or preset)
        if (!clear)
                assign q = 0;
        else if (!preset)
                assign  q = 1;
        else
                deassign q;

always @ (posedge clock)
        q = d;
endmodule;
```

Procedural continuous assignments

- The force/release statements have a similar effect to the assign/deassign pair, but a force can be applied to nets as well as variables.
- The LHS of the assignment cannot be a memory word (array reference) or a bit-select or a part-select of a vector variable
- force until release

```
module test;
reg a, b, c, d;
wire e;
and and1(e, a, b, c);
initial begin
    $monitor ("%d d=%b,e=%b",$stime, d, e);
    a=1; b=0; c=1;
    #10;
    force d = (a | b | c);
    force e = (a | b | c);
    #10;
    release d;
    release e;
    #10;
    $finish;
end;
endmodule;
```

Timing Controls

- Timing controls specify the simulation time at which procedural statements will be executed.
- In Verilog HDL, if there are no timing control statements, the simulation time will not advance.
- Timing Controls
  - Delay timing control
    - Delay control (#)
    - Intra assignment delay control (#)
  - Event timing control
    - Edge-triggered event control
      - Named event control (event declaration, event triggering(->))
      - Event control (@)
      - Event or operator (,)
    - Level-sensitive event control (wait)

Delay Control

```
delay_control ::= (From A.6.5)
        # delay_value
      | # ( mintypmax_expression )
event_control ::=
        @ hierarchical_event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
procedural_timing_control ::=
        delay_control
      | event_control
procedural_timing_control_statement ::=
      | procedural_timing_control statement_or_null
```

Delay Control

- Delay control
  - A non-zero delay is specified to the left of a procedural assignment.
  - It defers the execution of the entire statement.

```
reg  x, y;
integer count;
// The "<=" operators in the following statements can be replaced with "="
// without affecting the results.
   #25        y <= ~x;              // execute at time 25
   #15  count <= count  + 1;        // execute at time 40
```

Intra-Assignment Delay Control

- Intra-assignment delay control
  - A non-zero delay is specified to the right of the assignment operator.
  - It defers the assignment to the left-hand-side variable.

```
    y = #25  ~x;          // evaluate at time 0 but assign to y at time 25
count = #15 count  + 1;   // evaluate at time 25 but assign to count at time 40
```

```
    y <= #25  ~x;         // evaluate at time 0 but assign to y at time 25
count <= #15 count  + 1;  // evaluate at time 0 but assign to count at time 15
```

Delay Control

```
module
reg [1:0] a, b;

initial begin
    a = 'b1;
    b = 'b0;
end

always begin
    #50 a <= ~a;
end

always begin
    #100 b<= ~b;
end

endmodule
```

begin-end block: sequential block

Delay Control

```
        #25;            //wait 25 time units
        x= a + 6;       // execute immediately
```

```
        #25 x = a +6; // wait 25 time units and execute
```

```
reg x, y, z;
integer count;

initial  begin
            y<= ~x;                 // execute at time 0
#10         z = z +1;               // execute at time 10
            count <= count + 1; // execute at time 10
end
```

Event Timing Control

- Event Timing Control
  - An event is the change in the value on a variable or a net.
  - The execution of a procedural statement can be synchronized with an event.
- Two types of event control
  - Edge-triggered event control
    - Named event control
    - Event or operator
  - Level-sensitive event control

Edge-Triggered Event Control

- Edge-triggered event control
  - The symbol @ is used to specify such event control.
    - @(posedge clock): at the positive edge
    - @(negedge clock): at the negative edge

```
always @(posedge clock)  begin
    reg1 <= #25  in_1;                          // intra-assignment delay control
    reg2 <= @(negedge clock) in_2 ^ in_3;  // edge-triggered event control
    reg3 <= in_1;                               // no delay control
end
```

Detecting posedge and negedge

Table 9-1—Detecting posedge and negedge

From	To			
	0	1	x	z
0	No edge	posedge	posedge	posedge
1	negedge	No edge	negedge	negedge
x	negedge	posedge	No edge	No edge
z	negedge	posedge	No edge	No edge

```
@r rega = regb; // controlled by any value change in the reg r
@(posedge clock) rega = regb; // controlled by posedge on clock
forever @(negedge clock) rega = regb; // controlled by negative edge
```

Named Event Control

- A named event triggering scenario consists of three steps
  1. (Declaration) is declared with the keyword event.
     - does not hold any data.
  2. (Triggering) triggered by the symbol ->.
  3. (Recognition) recognized by the symbol @.

```
event  received_data;  // declare an event

// trigger event received_data
always @(posedge clock)  if (last_byte) -> received_data;

always @(received_data) begin
   …..
end  // execute event-dependent operations
```

Event or Control

- Event or control
  - uses the keyword or to specify multiple triggers.
  - can be replaced by the ",".
  - can use @* or @(*) to mean a change on any signal.

```
always @(posedge clock or negative reset_n)  // event or control
begin
    if (!reset_n) q <= 1`b0;    // asynchronous reset.
    else          q <= d;
end
```

Level-Sensitive Event Control

- Level-sensitive event control
  - uses the keyword wait.
  - If the condition is false, the procedural statements following the wait statement shall remain blocked until that condition becomes true.

```
always
    wait  (count_enable) count = count –1 ;

always
    wait  (!enable) #10 a = b ;
    #10 c  = d;
```

Intra-assignment timing control

```
blocking_assignment ::= (From A.6.2)
      variable_lvalue = [ delay_or_event_control ] expression
nonblocking_assignment ::=
      variable_lvalue <= [ delay_or_event_control ] expression
delay_control ::= (From A.6.5)
       # delay_value
      | # ( mintypmax_expression )
delay_or_event_control ::=
       delay_control
      | event_control
      | repeat ( expression ) event_control
event_control ::=
       @ hierarchical_event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
event_expression ::=
       expression
      | posedge expression
      | negedge expression
      | event_expression or event_expression
      | event_expression , event_expression
```

Syntax 9-12—Syntax for intra-assignment delay and event control

Intra-assignment timing control

Table 9-2—Intra-assignment timing control equivalence

Intra-assignment timing control	
With intra-assignment construct	**Without intra-assignment construct**

`a = #5 b;`	`begin`
	`  temp = b;`
	`  #5 a = temp;`
	`end`

`a = @(posedge clk) b;`	`begin`
	`  temp = b;`
	`  @(posedge clk) a = temp;`
	`end`

`a = repeat(3)`
`  @(posedge clk) b;`
	`begin`
	`  temp = b;`
	`  @(posedge clk);`
	`  @(posedge clk);`
	`  @(posedge clk) a = temp;`
	`end`

Intra-assignment timing control

```
fork                    // a race condition
    #5 a =b;
    #5 b=a;
join
```

```
fork                    // a data swap
    a = # 5 b;
    b = #5  a;
join
```

```
fork                    // a data shift
    a = @(posedge clk) b;
    b = @(posedge clk) c;
join
```

Intra-assignment timing control

```
a <= repeat(5) @(posedge clk) data;
```

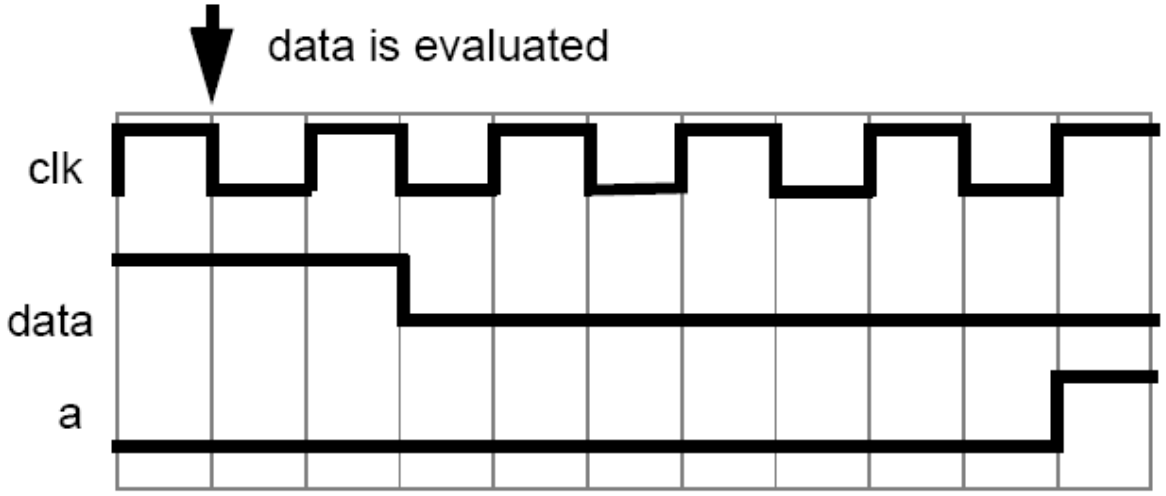


Figure 9-1—Repeat event control utilizing a clock edge

Block statements

- Sequential block: begin-end block

```
begin : block_1
    areg = breg;
    creg = areg;      // creg stores the value of breg
end
```

- Parallel block: fork-join block

```
fork : block_2
    # 50   r = 'h35;
    #100  r = 'hE2;
    #150  r = 'h00;
    #200  r = 'hF7;
    #250  -> end_wave;
join
```

Start and finish time of Block statements

- Sequential block: begin-end block
  - Start time: when the first statement is executed
  - Finish time: when the last statement has been executed

- Parallel block: fork-join block
  - Start time: the same for all statements
  - Finish time: when the last time-ordered statement has been executed

Block statements

```
parameter d= 50;
reg [7:0] r;

begin :block_3
    #d  r = 'h35;
    #d  r = 'hE2;
    #d  r = 'h00;
    #d  r = 'hF7;
    #d  -> end_wave;
end
```

```
fork : block_2
    # 50   r = 'h35;
    #100  r = 'hE2;
    #150  r = 'h00;
    #200  r = 'hF7;
    #250  -> end_wave;
join
```

Block statements

```
fork : block_4
    #250  -> end_wave;
    #200  r = ‘hF7;
    #150  r = ‘h00;
    #100  r = ‘hE2;
    # 50   r = ‘h35;
join
```

```
fork : block_2
    # 50   r = ‘h35;
    #100  r = ‘hE2;
    #150  r = ‘h00;
    #200  r = ‘hF7;
    #250  -> end_wave;
join
```

Block statements

```
begin
    fork
        @Aevent;
        @Bevent;
    join
    areg = breg;
end
```

```
begin
    begin
        @Aevent;
        @Bevent;
    end
    areg = breg;
end
```

```
fork
    @enable_a
        begin
            #ta wa = 0;
            #ta wa = 1;
            #ta wa = 0;
        end
    @enable_b
        begin
            #tb wb = 0;
            #tb wb = 1;
            #tb wb = 0;
        end
join
```

Disabling of named blocks and tasks

```
begin : bloak_a
    rega=regb;
    disable block_a;
    regc=rega;        // never executed
end
```

```
begin : break
   for (i=0; i<n; i= i+1) begin : continue
        @clk
             if (a==0)              // “continue” loop
                  disable continue;
             statements;
        @clk
             if (a==b)              //”break” from loop
                  disable break;
             statements;
    end
end
```

Scheduling and execution of events

- A design : consists of connected threads of execution or processes.
- Processes : are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs.
  Processes include primitives, modules, initial and always procedural blocks, continuous assignments, tasks, procedural assignment statements.
- Every change in value of a net or variable in the circuit being simulated, as well as the named event, is considered an update event.
- Processes are sensitive to update events.

Scheduling and execution of events

- When an update event is executed, all the processes that are sensitive to that event are evaluated **in an arbitrary order**.
- The evaluation of a process is also an event, known as an evaluation event.
- The term simulation time is used to refer to the time value maintained by the simulator to the model the actual time it would take for the circuit being simulated.
- Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an event queue, ordered by the simulation time. Putting an event on the queue is called scheduling time.

Simulation Terminology

- Simulation time
  - Time value used by simulator to model actual time.
- Simulation cycle
  - Complete processing of all currently active events
- Can be multiple simulation cycles per simulation time

- Explicit zero delay (#0)
  - Forces process to be inactive event instead of active
  - Incorrectly used to avoid race conditions
  - #0 doesn't synthesize!
  - Don't use it

Stratified event queue

- The Verilog event queue is logically segmented into five different regions. Events are added to any of the five regions, but are only removed from the active region.
  1. Active events occur at the current simulation time and can be processed in any order.
  2. Inactive events occur at the current simulation time, but shall be processed after all the active events are processed.
  3. Nonblocking assignment (NBA) update events have been evaluated during some previous simulation time, but shall be assigned at this simulation time after all active and inactive events are processed.
  4. Monitor events shall be processed after all active, inactive, and NBA update events are processed.
  5. Future events occurs at some future simulation time. Future events are divided into future active events and future NBA update events.

Stratified event queue

- Active events region: the events in this region result from
  - evaluating the RHS of nonblocking assignments
  - evaluating the inputs of a primitive and changing the output
  - executing a procedural (blocking) assignment to a register variable
  - evaluating the RHS of a continuous assignment and updating the LHS
  - evaluating the RHS of a procedural continuous assignment and updating the LHS
  - evaluating and executing $display and $write system tasks.
- The processing of all active events is called a simulation cycle.
- An explicit zero delay control (#0) requires that the process be suspended and added as an inactive event for the current time so that the process is resumed in the next simulation time in the current time.
- The $monitor and $strobe system tasks create monitor events for their arguments.

Verilog simulation reference model

```
while  (there are events)
    if (no active events)
        if (there are inactive events) {
                activate all inactive events;
        } else if (there are nonblobking assign update events) {
                activate all nonblocking assign update events;
        } else if (there are monitor events) {
                activate all monitor events;
        } else {
                advance T to the next event time;
                activate all inactive events for time T;
        }
    }
    E = any active events;
    if (E is an update event) {
        update the modified object;
        add evaluation events for the sensitive processes to event queue;
    } else  { /* shall be an evaluation event */
        evaluate the process;
        add all update events to the event queue;
    }
}
```

Inactive events

- “a = b;” : immediately evaluated and updated:
- “a = #0 b;”: evaluated and an inactive update event is scheduled

- “a <= b;” : evaluated and an NBA update event is scheduled
- “a <= #0 b;”: evaluated and an NBA update event is scheduled

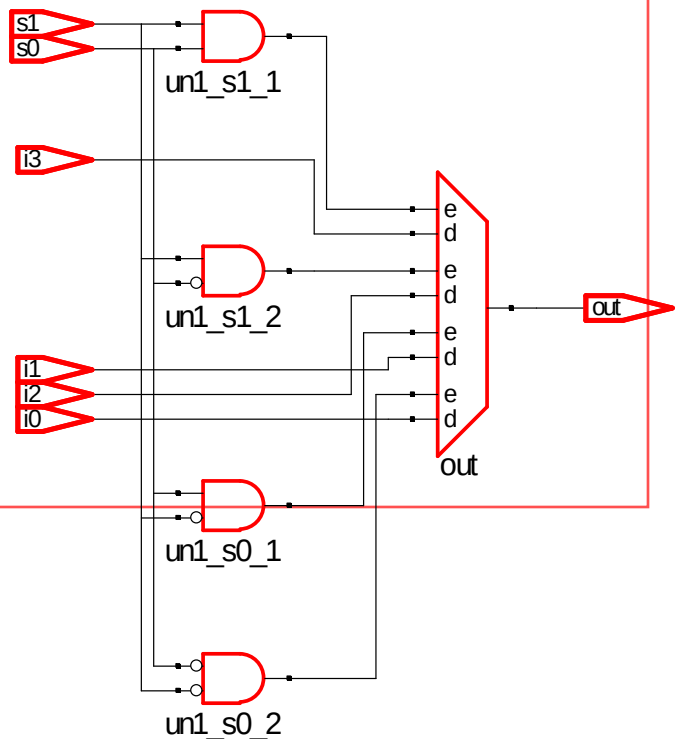
- “#0 a = b;”: an inactive evaluation event is scheduled
- “#0 a <= b;”: an inactive evaluation event is scheduled

Selection Constructs

```
if  (<expression>)  true_statement ;

if  (<expression>) true_statement; else false_statement;

if  (<expression1>) true_statement1;
else if (<expression2>) true_statement2;
else false_statement;
```

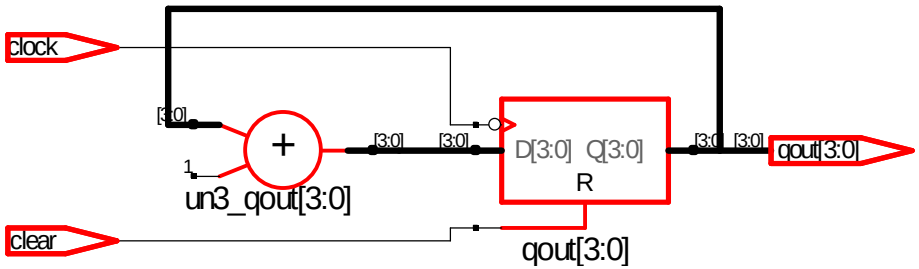
Selection Constructs

```
module mux4_to_1_ifelse (i0, i1, i2, i3, s1, s0, out);
// port declarations
input    i0, i1, i2, i3;
input    s1, s0;
output  reg out;
// using conditional operator if  else statement
always @(*)  // triggered for all signals used in the
                    // if else statement
    if (s1) begin
        if (s0)  out = i3; else out = i2; end
    else begin
        if (s0)  out = i1; else out = i0; end
endmodule
```



A Simple 4-bit Counter

```
module counter (clock, clear, qout);
input   clock, clear;
output reg [3:0] qout;
// the body of the 4-bit counter.
always @(negedge clock or posedge clear)
begin
     if (clear)
          qout <= 4'd0;
     else
          qout <= (qout + 1) ;   // qout = (qout + 1) % 16;
end
endmodule
```

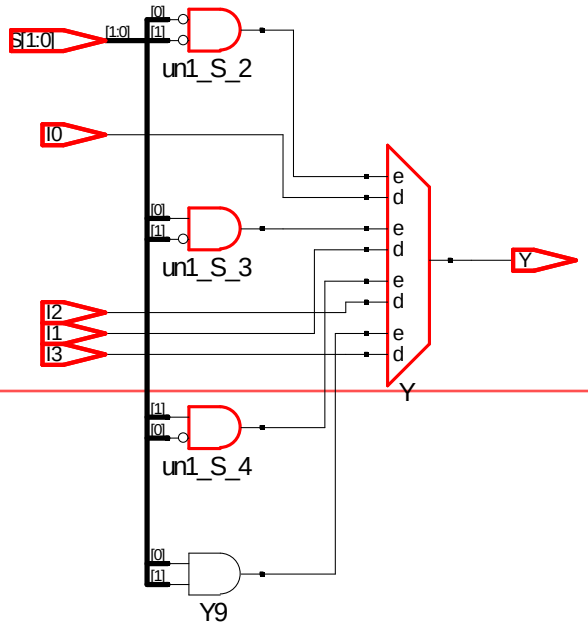


Selection Constructs

- case statement: a multiway selection.
  - compares the expression to the alternatives in the order they are written.
  - compares 0, 1, x, and z values in the expression and the items (alternatives) bit for bit.
  - executes the default statement if no matches are made.
  - fills zeros to match the unequal bit widths between the expression and an item (alternative).
  - is acted like a multiplexer.
- The default statement is optional and at most one default statement can be placed inside one case statement.
- A block statement must be grouped by begin and end.

A 4-to-1 MUX Example

```
// a 4-to-1 multiplexer using case statement
module mux_4x1_case (I0, I1, I2, I3, S, Y);
input I0, I1, I2, I3;
input [1:0] S;    // declare S as a two-bit selection signal.
output reg  Y;
always @(I0 or I1 or I2 or I3 or S) // It can use always @(*).
   case (S)
      2'b00: Y = I0;
      2'b01: Y = I1;
      2'b10: Y = I2;
      2'b11: Y = I3;
   endcase
endmodule
```



A 4-to-1 MUX Example

```
// a 4-to-1 multiplexer using case and default statements.
module mux4_to_1_case_default (i0, i1, i2, i3, s1, s0, out);
input   i0, i1, i2, i3, s1, s0;
output reg out;       //output declared as register
always @(s1 or s0 or i0 or i1 or i2 or i3)
   case ({s1, s0})   // concatenate s1 and s0 as a two-bit selection signals
      2'b00: out = i0;
      2'b01: out = i1;
      2'b10: out = i2;
      2'b11: out = i3;
      default: out = 1'bx;  // using default to include all other possible cases.
   endcase
endmodule
```

Selection Constructs

- casex and casez statements
  - are used to perform a multiway selection like that of case statement.
  - compare only non-x or z positions in the case expression and the case items (alternatives).
- casez treats all z values as don't cares.
- casex treats all x and z values as don't cares.
- In addition to specifying bits as z, they may also be specified with question marks "?" which also indicates don't-care

Selection Constructs

```
// an example illustrating how to count the trailing zeros in a nibble.
module trailing_zero_4b (data, out);
input   [3:0] data;
output reg [2:0] out;   //output declared as register
always @(data)
  casex (data)          // treat both x and z as don't care conditions.
     4'bxxx1: out = 0;
     4'bxx10: out = 1;
     4'bx100: out = 2;
     4'b1000: out = 3;
     4'b0000: out = 4;
     default:  out = 3'b111; //using default to include all other possible cases.
  endcase
endmodule
```

Selection Constructs

```
// an example of casex.
module decode;
reg   [7:0] r, mask;

always
   begin
      // other statements
      r      =  8'b01100110;
      mask = 8'bx0x0x0x0;
     casex (r  ^  mask)          // r ^ mask = 8b'x1x0x1x0
                                 // treat both x and z as don't care conditions.
        8'b001100xx: statement1;
        8'b1100xx00: statement2;  // matches and executed
        8'b00xx0011: statement3;
        8'bxx001100: statement4; // matches
     endcase
endmodule
```

Loop Constructs

- Loop constructs control the execution of a statement zero, one, or more times.
- Loop constructs
  - can appear only inside an initial or always block.
  - may contain delay expressions.
- Four types
  - while loop executes a statement until an expression becomes false.
  - for loop repeatedly executes a statement.
  - repeat loop executes a statement a fixed number of times.
  - forever loop continuously executes a statement.

While Loop Structure

- A while loop
  - executes until the condition is false.
  - shall not be executed at all if the condition_expr starts out false.

```
while (condition_expr) statement;
```

```
while (count < 12) count <= count + 1;
while (count <= 100 && flag) begin
// put statements wanted to be carried out here.
end
```

While Loop Structure

```
// an example illustrating how to count the zeros in a byte.
module zero_count_while (data, out);
input   [7:0] data;
output reg [3:0] out;        //output declared as register
integer i;
always @(data) begin
   out = 0; i = 0;
   while (i <= 7) begin     // simple condition
       if (data[i] == 0) out = out + 1; // may be replaced with out = out + ~data[i].
       i = i + 1; end
end
endmodule
```

While Loop Structure

```
// an example illustrating how to count the trailing zeros in a byte.
module trailing_zero_while (data, out);
input   [7:0] data;
output reg [3:0] out;  // output declared as register
integer i;                    // loop counter
always @(data) begin
   out = 0; i = 0;
   while (data[i] == 0 && i <= 7) begin // complex condition
      out = out + 1;
         i = i + 1;
   end
end
endmodule
```

Note that: Please distinguish the difference between this example and the previous one.

For Loop Structure

- A for loop is used to perform a counting loop. It
  - behaves like the for statement in C programming language.

```
for (init_expr; condition_expr; update_expr) statement;
```

  - is equivalent to

```
init_expr;
while (condition_expr) begin
    statement;
    update_expr;
end
```

For Loop Structure

```
// an example illustrating how to count the zeros in a byte.
module zero_count_for (data, out);
input   [7:0] data;
output reg [3:0] out;   // output declared as register
integer i;
always @(data) begin
  out = 0;
  for (i = 0; i <= 7; i = i + 1) // simple condition
      if (data[i] == 0)
          out = out + 1;          // may be replaced with out = out + ~data[i].
end
endmodule
```

For Loop Structure

```
// an example illustrating how to count the trailing zeros in a byte.
module trailing_zero_for (data, out);
input   [7:0] data;
output reg [3:0] out;  // output declared as register
integer i;                    // loop counter
always @(data) begin
   out = 0;
   for (i = 0; data[i] == 0 && i <= 7; i = i + 1) // complex condition
         out = out + 1;
end
endmodule
```

Note that: Please distinguish the difference between this example and the previous one.

Repeat Loop Structure

- A repeat loop performs a loop a fixed number of times.

```
repeat (counter_expr) statement;
```

  - counter_expr can be a constant, a variable or a signal value.
  - counter_expr is evaluated only once before starting the execution of statement (loop).

Repeat Loop Structure

- Examples:

```
i = 0;
repeat (32) begin
    state[i] = 0;      // initialize to zeros
    i  =  i  +  1;      // next item
end
repeat (cycles) begin                    // cycles must be evaluated to a number
@(posedge clock) buffer[i] <= data;  // before entering the loop.
      i  <=  i  + 1;      // next item
end
```

Forever Loop Structure

- A forever loop continuously performs a loop until the $finish task is encountered. It
  - is equivalent to a while loop with an always true expression such as while (1).

```
forever  statement;
```

  - can be exited by the use of disable statement.

Forever Loop Structure

- The forever statement example

```
initial  begin
    clock <= 0;
    forever begin
        #10 clock <= 1;
        #5   clock <= 0;
    end
end
```

- The forever statement is usually used with timing control statements.

```
reg clock, x, y;

initial
    forever @(posedge clock) x <= y;
```