Procedural timing controls

- The Verilog HDL has two types of explicit timing control over when procedural statements can occur.
  - ***delay control***, in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes.
  - ***event expression***, which allows statement execution to be delayed until the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure.

Delay control

- The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

- Execution of the assignment is delayed by the amount of simulation time specified by the value of the expression

```
#d rega = regb; // d is defined as a parameter
#((d+e)/2) rega = regb; // delay is average of d and e
#regr regr = regr + 1; // delay is the value in regr
```

Event control

- The execution of a procedural statement can be synchronized with a value change on a net or variable or the occurrence of a declared event.
  - The value changes on nets and variable can be used as events to trigger the execution of a statement.

Events

- A ***negedge*** shall be detected on the transition from 1 to x, z, or 0, and from x or z to 0.
- A ***posedge*** shall be detected on the transition from 0 to x, z, or 1, and from x or z to 1

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

- Example 1
```
always @(*) // equivalent to @(a or b or c or d or f)
    y = (a & b) | (c & d) | myfunction(f);
```
- Example 2
```
always @* begin // equivalent to @(a or b or c or d or tmp1 or tmp2)
    tmp1 = a & b;
    tmp2 = c & d;
    y = tmp1 | tmp2;
end
```
- Example 3
```
always @* begin // equivalent to @(b)
    @(i) kid = b; // i is not added to @*
end
```
- Example 4
```
always @* begin // equivalent to @(a or b or c or d)
    x = a ^ b;
    @* // equivalent to @(c or d)
    x = c ^ d;
end
```
- Example 5
```
always @* begin // same as @(a or en)
    y = 8'hff;
    y[a] = !en;
end
```

Edge controlled statements

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

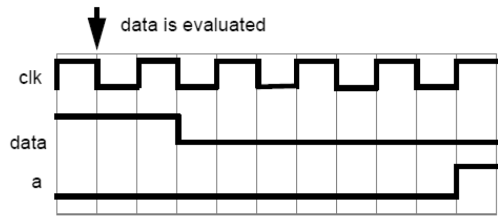
```
always @* begin // same as @(state or go or ws)
    next = 4'b0;
    case (1'b1)
    state[IDLE]: if (go) next[READ] = 1'b1;
                 else next[IDLE] = 1'b1;
                 state[READ]: next[DLY ] = 1'b1;
    state[DLY ]: if (!ws) next[DONE] = 1'b1;
                 else next[READ] = 1'b1;
    state[DONE]: next[IDLE] = 1'b1;
    endcase
end
```

Named events

- A new data type, in addition to nets and variables, called event can be declared.
- An identifier declared as an event data type is called a named event. A named event can be triggered explicitly.
  - It can be used in an event expression to control the execution of procedural statements in the same manner as event controls

```
event a;
-> a;
@a r=b;
```

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



- a <= **repeat**(a+b) @(**posedge** phi1 **or negedge** phi2) data;

Level-sensitive event control

- The execution of a procedural statement can also be delayed until a condition becomes true. This is accomplished using the wait statement.

```
wait ( expression ) statement_or_null
begin
    wait (!enable) #10 a = b;
    #10 c = d;
end
```

Block statements

- There are two types of blocks in the Verilog HDL:
  - There are two types of blocks in the Verilog HDL:
    - The sequential block is delimited by the keywords **begin** and **end**.
    - The procedural statements are executed sequentially in the given order.
  - *Parallel block*, also called *fork-join block*
    - The parallel block is delimited by the keywords **fork** and **join**.
    - The procedural statements in parallel block are executed concurrently.

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

Parallel blocks

- A *parallel block* shall have the following characteristics:
  - Statements shall execute concurrently.
  - Delay values for each statement shall be considered relative to the simulation time of entering the block.
  - Delay control can be used to provide time-ordering for assignments.
  - Control shall pass out of the block when the last time-ordered statement executes.

Sequential blocks

- A sequential block shall have the following characteristics:
  - Statements areexecuted in sequence.
  - Delay values for each statement is relative to the simulation time of the execution of the previous statement.
  - Control exits the block after the last statement executes.

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

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

```
begin
        #50 r = 'h35;
        #50 r = 'hE2;
        #50 r = 'h00;
        #50 r = 'hF7;
        #50 -> end_wave;
end
```

- *Example 1*—A sequential block enables the following two assignments to have a deterministic result:

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

The first assignment is performed, and areg is updated before control passes to the second assignment.

- *Example 2*—Delay control can be used in a sequential block to separate the two assignments in time.

```
begin
        areg = breg;
        @(posedge clock) creg = areg; // assignment delayed until
end // posedge on clock
```

- *Example 3*—The following example shows how the combination of the sequential block and delay control can be used to specify a time-sequenced waveform:

```
parameter d = 50; // d declared as a parameter and
reg [7:0] r; // r declared as an 8-bit reg
event end_wave;

begin // a waveform controlled by sequential delay
        #d r = 'h35;
        #d r = 'hE2;
        #d r = 'h00;
        #d r = 'hF7;
        #d -> end_wave; //trigger an event called end_wave
end
```

This example shows two sequential blocks

- All procedures in the Verilog HDL are specified within one of the following four statements:
  - initial construct
  - always construct
  - Task
  - Function

Joining of Events

- When an assignment is to be made after two separate events have occurred, known as the joining of events, a fork-join block can be useful.

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

initial construct

- Is used to initialize variables at the start of simulation.

```
initial begin
    areg = 0; // initialize a reg
    for(index=0; index<size; index=index + 1)
        memory[index] = 0; //initialize memory word
end

initial begin
    inputs = 'b000000; // initialize at time zero
    #10 inputs = 'b011001; // first pattern
    #10 inputs = 'b011011; // second pattern
    #10 inputs = 'b011000; // third pattern
    #10 inputs = 'b001000; // last pattern
end
```

Parallel execution of sequential blocks

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

Distinctions between tasks and functions

- A function shall execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function shall have at least one input type argument and shall not have an output or inout type argument; a task can have zero or more arguments of any type.
- A function shall return a single value; a task shall not return a value.

Always construct

- The always construct repeats continuously throughout the duration of the simulation.
  - **This creates a zero-delay infinite loop**
    - **always** areg = ~areg;
  - **Providing a timing control to the above code creates a potentially useful description**
    - **alway**s #half_period areg = ~areg;

Tasks and task enabling

- A task shall be enabled from a statement that defines the argument values to be passed to the task and the variables that receive the results.
- Control shall be passed back to the enabling process after the task has completed.
  - the time of enabling a task can be different from the time at which the control is returned
- A task can enable other tasks
  - no limit on the number of tasks enabled

Tasks and functions

- Tasks and functions provide the ability to execute common procedures from several different places in a description.
- They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions

The use of tasks

```
module traffic_lights;
reg clock, red, amber, green;
parameter on = 1, off = 0, red_tics = 350, amber_tics = 30, green_tics = 200;
initial  begin red = off; amber = off; green = off; end                    // initialize colors.
always begin // sequence to control the lights.
        red = on; // turn red light on
        light(red, red_tics); // and wait.
        green = on; // turn green light on
        light(green, green_tics); // and wait.
        amber = on; // turn amber light on
        light(amber, amber_tics); // and wait.
end
// task to wait for 'tics' positive edge clocks before turning 'color' light off.
task light;
output color;
input [31:0] tics;
begin
        repeat (tics) @ (posedge clock);
        color = off; // turn light off.
end
endtask
always begin // waveform for the clock.
        #100 clock = 0;   #100 clock = 1;
end
endmodule // traffic_lights.
```

Task: Examples

- **The following example illustrates the basic structure of a task definition with five arguments**:

```
task my_task;
input a, b;
inout c;
output d, e;
begin
. . . // statements that perform the work of the task
. . .
c = foo1; // the assignments that initialize result regs
d = foo2;
e = foo3;
end
endtask
```

- **Or using the second form of a task declaration, the task could be defined as follows:**

```
task my_task (input a, b, inout c, output d, e);
begin
        . . . // statements that perform the work of the task
        . . .
        c = foo1; // the assignments that initialize result regs
        d = foo2;
        e = foo3;
end
endtask
```

Disabling of named blocks and tasks

- The disable statement provides the ability to terminate the activity associated with concurrently active procedures.
- The disable statement gives a mechanism
  - terminating a task before it executes all its statements
  - breaking from a looping statement
  - skipping statements in order to continue with another iteration

task my_task (input a, b, inout c, output d, e);

- The following statement enables the task:
  my_task (v, w, x, y, z);
- The task-enabling arguments (v, w, x, y, and z) correspond to the arguments (a, b, c, d, and e) defined by the task. At task-enabling time, the input and inout type arguments (a, b, and c) receive the values passed in v, w, and x. Thus, execution of the task-enabling call effectively causes the following assignments:
  a = v;
  b = w;
  c = x;
- my_task shall place the computed result values into c, d, and e. When the task completes, the following assignments to return the computed values to the calling process are performed:
  x = c;
  y = d;
  z = e;

Example: disable blocks

- *Example 1*—This example illustrates how a block disables itself.

```
begin : block_name
        rega = regb;
        disable block_name;
        regc = rega; // this assignment will never execute
end
```

- *Example 2*—This example shows the disable statement being used within a named block in a manner similar to a forward *goto*. The next statement executed after the disable statement is the one following the named block.

```
begin : block_name
        ...
        if (a == 0)
        disable block_name;
        ...
end // end of named block
// continue with code following named block
...
```

Example 3: disable blocks

```
task proc_a;
begin
        ...
        ...
        if (a == 0)
                disable proc_a; // return if true
        ...
        ...
end
endtask
```

Example 4: disable blocks

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

Example 5

- This example shows the disable statement being used to disable concurrently a sequence of timing controls and the task action when the reset event occurs.

```
fork
        begin : event_expr
                @ev1;
                repeat (3) @trig;
                #d action (areg, breg);
        end
        @reset disable event_expr;
join
```

- The sequential block and the wait for reset execute in parallel. The event_expr block waits for one occurrence of event ev1 and three occurrences of event trig. When these four events have happened, plus a delay of d time units, the task action executes. When the event reset occurs, regardless of events within the sequential block, the fork-join block terminates—including the task

Function declarations

- The following example defines a function called getbyte, using a range specification:

```
function [7:0] getbyte;
input [15:0] address;
begin
        // code to extract low-order byte from addressed word
        . . .
        getbyte = result_expression;
end
endfunction
```

- Or using the second form of a function declaration, the function could be defined as follows:

```
function [7:0] getbyte (input [15:0] address);
begin
        // code to extract low-order byte from addressed word
        . . .
        getbyte = result_expression;
end
endfunction
```

Example 6: is a behavioral description of a retriggerable monostable

- The named event retrig restarts the monostable time period. If retrig continues to occur within 250 time units, then q will remain at 1.

```
always begin : monostable
        #250 q = 0;
end
always @retrig begin
        disable monostable;
        q = 1;
end
```

Returning a value from a function

- The function definition shall implicitly declare a variable, internal to the function, with the same name as the function.
  - is the same type as the type specified in the function declaration
  - It is illegal to declare another object with the same name as the function in the scope where the function is declared

```
getbyte = result_expression;
```

Functions and function calling

- The purpose of a function is to return a value that is to be used in an expression.
  - Declaration
  - Use

Example: a function factorial that returns an integer value

```
module tryfact;
// define the function
function automatic integer factorial;
input [31:0] operand;
integer i;
if (operand >= 2)
        factorial = factorial (operand - 1) * operand;
else
        factorial = 1;
endfunction
// test the function
integer result;
integer n;
initial begin
        for (n = 0; n <= 7; n = n+1) begin
                result = factorial(n);
                $display("%0d factorial=%0d", n, result);
        end
end
endmodule    //tryfact
```

The simulation results are as follows:
0 factorial=1
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
7 factorial=5040

Calling a function

- A function call is an operand within an expression
- The order of evaluation of the arguments to a function call is undefined.
- Example:

word=control? {getbyte(msbyte),getbyte(lsbyte)}:0;

Constant function calls

```
module ram_model (address, write, chip_select, data);
        parameter data_width = 8;
        parameter ram_depth = 256;
        localparam addr_width = clogb2(ram_depth);
        input [addr_width - 1:0] address;
        input write, chip_select;
        inout [data_width - 1:0] data;
        //define the clogb2 function
        function integer clogb2;
        input [31:0] value;
        begin
                value = value - 1;
                for (clogb2 = 0; value > 0; clogb2 = clogb2 + 1)
                        value = value >> 1;
        end
endfunction

reg [data_width - 1:0] data_store[0:ram_depth - 1];
//the rest of the ram model
```

- An instance of this ram_model with parameters assigned is as follows:

```
        ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);
```

Function rules

- Functions are more limited than tasks. The following rules govern their usage:
  a) A function definition shall not contain any time-controlled statements, that is, any statements containing **#**, **@**, or **wait**.
  b) Functions shall not enable tasks.
  c) A function definition shall contain at least one input argument.
  d) A function definition shall not have any argument declared as output or inout.
  e) A function shall not have any nonblocking assignments or procedural continuous assignments.
  f) A function shall not have any event triggers.