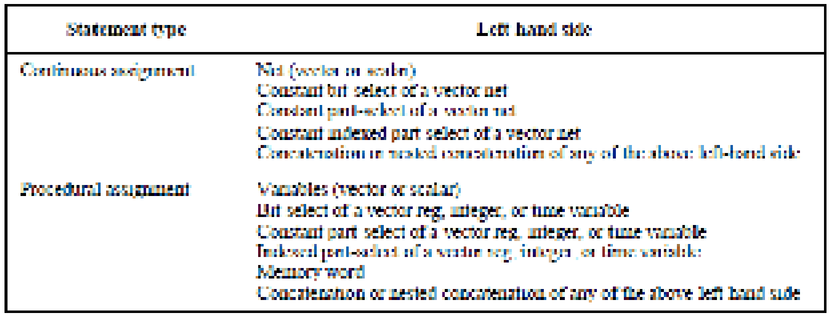
Assignments

- The assignment is the basic mechanism for placing values into nets and variables. There are two basic forms of assignments:
  - The continuous assignment, which assigns values to nets
  - The procedural assignment, which assigns values to variables



Statement type	Left hand side
Continuous assignment	Net (vector or scalar) Constant bit select of a vector net Constant part-select of a vector net Constant indexed part select of a vector net Concatenation or nested concatenation of any of the above left-hand side
Procedural assignment	Variables (vector or scalar) Bit select of a vector reg, integer, or time variable Constant part select of a vector reg, integer, or time variable Indexed part-select of a vector reg, integer, or time variable Memory word Concatenation or nested concatenation of any of the above left hand side

Continuous assignments

- Continuous assignments drives values onto nets, both vector and scalar.
- Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates.

Continuous assignments: Examples

- Exmple1:
```
wire mynet ;
assign (strong1, pull0) mynet = enable ;
```
- Example2
```
wire (strong1, pull0) mynet = enable ;
```
- Example3
```
module adder (sum_out, carry_out, carry_in, ina, inb);
output [3:0] sum_out;
output carry_out;
input [3:0] ina, inb;
input carry_in;
wire carry_out, carry_in;
wire [3:0] sum_out, ina, inb;
        assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

Continuous assignments: Examples

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
parameter n = 16;
parameter Zee = 16'bz;
output [1:n] busout;
input [1:n] bus0, bus1, bus2, bus3;
input enable;
input [1:2] s;
tri [1:n] data;          // net declaration
                         // net declaration with continuous assignment
tri [1:n] busout = enable ? data : Zee;
        // assignment statement with four continuous assignments
assign
        data = (s == 0) ? bus0 : Zee,
        data = (s == 1) ? bus1 : Zee,
        data = (s == 2) ? bus2 : Zee,
        data = (s == 3) ? bus3 : Zee;
endmodule
```

Behavioral modeling

- Verilog *behavioral models* contain
  - *procedural statements* that control the simulation and manipulate variables of the data types previously.
  - These statements are contained within procedures. Each procedure has an activity flow associated with it.
    - The activity starts at the control constructs **initial** and **always**.
      - Each initial construct and each always construct starts a separate activity flow.
      - All of the activity flows are concurrent to model the inherentconcurrence of hardware.

Procedural assignments

- procedural assignments put values in variables.
- Procedural assignments occur within procedures such as always, initial, task and function.
  - It executes when the flow of execution in the simulation reaches an assignment within a procedure.

Behavioral model Example

```
module behave;
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
```

During simulation of this model, all of the flows defined by the initial and always constructs start together at simulation time zero. The initial constructs execute once, and the always constructs execute repetitively.

Variable declaration assignment

- Declare a 4-bit reg and assign it the value 4.
  - **reg** [3:0] a = 4'h4;
- This is equivalent to writing
  - **reg** [3:0] a;
  - **initial** a = 4'h4;
- The following example is not legal:
  - **reg** [3:0] array [3:0] = 0;
- Declare two integers; the first is assigned the value of 0.
  - **integer** i = 0, j;
- Declare two real variables, assigned to the values 2.5 and 300,000.
  - **real** r1 = 2.5, n300k = 3E6;
- Declare a time variable and realtime variable with initial values.
  - **time** t1 = 25;
  - **realtime** rt1 = 2.5;

Procedural assignments

- The Verilog HDL contains two types of procedural assignment statements:
  - *Blocking procedural assignment* statements
    - Executes before the execution of the statements that follow it in a sequential block.
    - It does not prevent the execution of statements that follow it **in a parallel block.**
  - *Nonblocking procedural assignment statements*
    - allows assignment scheduling without blocking the procedural flow.
    - used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other.

Procedural assignments

- The right-hand side of a procedural assignment can be any expression that evaluates to a value.
- The lefthand side shall be a variable that receives the assignment from the right-hand side.

Blocking procedural assignment

- The following examples show blocking procedural assignments:

```
      rega = 0;
      rega[3] = 1; // a bit-select
      rega[3:5] = 7; // a part-select
      mema[address] = 8'hff; // assignment to a mem
element
      {carry, acc} = rega + regb; // a concatenation
```

Procedural assignments

- The left-hand side of a procedural assignment can take one of the following forms:
  - **reg, integer, real, realtime**, or **time** data type: an assignment to the name reference of one of these data types.
  - Bit-select of a **reg, integer**, or **time** data type: an assignment to a single bit that leaves the other bits untouched.
  - Part-select of a **reg, integer**, or **time** data type: a part-select of one or more contiguous bits that leaves the rest of the bits untouched.
  - Memory word: a single word of a memory.
  - Concatenation or nested concatenation of any of the above: a concatenation or nested concatenation of any of the previous four forms. Such specification effectively partitions the result of the right hand expression and assigns the partition parts, in order, to the various parts of the concatenation or nested concatenation.

Example

```
//non_block1.v
module non_block1;
reg a, b, c, d, e, f;
//blocking assignments
initial begin
        a = #10 1; // a will be assigned 1 at time 10
        b = #2 0; // b will be assigned 0 at time 12
        c = #4 1; // c will be assigned 1 at time 16
end
//non-blocking assignments
initial begin
        d <= #10 1; // d will be assigned 1 at time 10
        e <= #2 0; // e will be assigned 0 at time 2
        f <= #4 1; // f will be assigned 1 at time 4
end
endmodule
```

scheduled changes at time 2

c = 0

scheduled changes at time 4

f = 1

scheduled changes at time 10

d = 1

Blocking procedural assignment

```
output out;
reg a, b, c;
initial begin
        a = 0;
        b = 1;
        c = 0;
end
always c = #5 ~c;
always @(posedge c) begin
   a <= b; // evaluates, schedules,
   b <= a; // and executes in two steps
end
endmodule
```

Step 1: At posedge c, the simulator evaluates the right-hand sides of the nonblocking assignments and schedules the assignments of the new values at the end of the nonblocking assign update events [a=0. b=1]

Step 2: When the simulator activates the nonblocking assign update events, the simulator updates the left-hand side of each nonblocking assignment statement [a=1, b=0]

Examples

```
reg a;
initial a = 1;
// The assigned value of the reg is
// determinate
initial begin
   a <= #4 0; // schedules a = 0 at time 4
   a <= #4 1; // schedules a = 1 at time 4
end        // At time 4, a = 1
endmodule
```

```
module multiple2;
reg a;
initial a = 1;
initial a <= #4 0; // schedules 0 at time 4
initial a <= #4 1; // schedules 1 at time 4
// At time 4, a = ??
// The assigned value of the reg is indeterminate
endmodule
```

Example

```
//non_block1.v
module non_block1;
reg a, b, c, d, e, f;
//blocking assignments
initial begin
        a = #10 1; // a will be assigned 1 at time 10
        b = #2 0; // b will be assigned 0 at time 12
        c = #4 1; // c will be assigned 1 at time 16
end
//non-blocking assignments
initial begin
        d <= #10 1; // d will be assigned 1 at time 10
        e <= #2 0; // e will be assigned 0 at time 2
        f <= #4 1; // f will be assigned 1 at time 4
end
endmodule
```

Procedural continuous assignments

- The procedural continuous assignments (using keywords **assign** and **force**) are procedural statements that allow expressions to be driven continuously onto variables or nets.
  - The left-hand side of the assignment in the assign statement shall be a variable reference or a concatenation of variables.
- The **deassign** procedural statement shall end a procedural continuous assignment to a variable

Example

```
module multiple4;
reg r1;
reg [2:0] i;
initial begin
//makes assignments to r1 without
//cancelling previous assignments
for (i = 0; i <= 5; i = i+1)
   r1 <= # (i*10) i[0];
end
endmodule
```

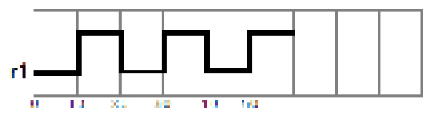
Example

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

The assign procedural continuous assignment statement shall override all procedural assignments to a variable

Example

```
module multiple4;
reg r1;
reg [2:0] i;
initial begin
//makes assignments to r1 without
//cancelling previous assignments
for (i = 0; i <= 5; i = i+1)
   r1 <= # (i*10) i[0];
end
endmodule
```



Conditional statement

- The conditional statement (or if-else statement) is used to make a decision about whether a statement is executed

```
if (index > 0)
    if (rega > regb)
        result = rega;
    else // else applies to preceding if
        result = regb;
```

Example: force, 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);
        assign d = a & b & c;
        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
```

If-else

```
if (index > 0) begin
    if (rega > regb)
        result = rega;
    end
else
    result = regb;
```

```
if (index > 0)
    if (rega > regb)
        result = rega;
    else
        result = regb;
```

Example: force, 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);
        assign d = a & b & c;
        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
```

Results:

```
 0  d=0,e=0
10  d=1,e=1
20  d=0,e=0
```

```
// declare regs and parameters
reg [31:0] instruction, segment_area[255:0];
reg [7:0]index;
reg [5:0]modify_seg1,
modify_seg2,
modify_seg3;
Parameter   segment1 = 0, inc_seg1 = 1,
            segment2 = 20, inc_seg2 = 2,
             segment3 = 64, inc_seg3 = 4,
              data = 128;
// test the index variable
if (index< segment2) begin
      instruction = segment_area [index + modify_seg1];
       index = index + inc_seg1;
end
else if (index < segment3) begin
            instruction = segment_area [index + modify_seg2];
            index = index + inc_seg2;
end
else if (index < data) begin
            instruction = segment_area [index + modify_seg3];
            index = index + inc_seg3;
end
else
            instruction = segment_area [index];
```

The following module fragment uses the if-else statement to test the variable index to decide whether one of three modify_segn regs has to be added to the memory address and which increment is to be added to the index reg. The first ten lines declare the regs and parameters.

Case statement

- The case statement is a multiway decision statement that tests whether an expression matches one of a number of other expressions and branches accordingly.

case(expr)

Case_item: statement

....

Default: statement

endcase

The default statement shall be optional.

Case statement

```
reg [15:0] rega;
reg [9:0] result;
case (rega)
   16'd0: result = 10'b0111111111;
   16'd1: result = 10'b1011111111;
   16'd2: result = 10'b1101111111;
    16'd3: result = 10'b1110111111;
    16'd4: result = 10'b1111011111;
    16'd5: result = 10'b1111101111;
    16'd6: result = 10'b1111110111;
    16'd7: result = 10'b1111111011;
    16'd8: result = 10'b1111111101;
    16'd9: result = 10'b1111111110;
    default result = 'bx;
endcase
```

- The case expression given in parentheses shall be evaluated exactly once and before any of the case item expressions.
- The case item expressions shall be evaluated and compared in the exact order in which they are given.
- If one of the case item expressions matches the case expression given in parentheses, then the statement associated with that case item shall be executed, and the linear search shall terminate.
- If all comparisons fail and the default is given then the default item statement is executed.
- If the default statement is not given and all of the comparisons fail, then none of the case item statements shall be executed.
- **the comparison only succeeds when each bit matches exactly with respect to the values 0, 1, x, and z.**

Case Statement

- The following example illustrates the use of a case statement to handle x and z values properly:

```
case (select[1:2])
        2'b00: result = 0;
        2'b01: result = flaga;
        2'b0x,
        2'b0z: result = flaga ? 'bx : 0;
        2'b10: result = flagb;
        2'bx0,
        2'bz0: result = flagb ? 'bx : 0;
        default result = 'bx;
endcase
```

In this example, if select[1] is 0 and flaga is 0, then even if the value of select[2] is x or z, result should be 0—which is resolved by the third case.

Casez Statement

- If the most significant bit of ir is a 1, then the task instruction1 is called, regardless of the values of the other bits of ir.

```
reg [7:0] ir;
casez (ir)
        8'b1???????: instruction1(ir);
        8'b01??????: instruction2(ir);
        8'b00010???: instruction3(ir);
        8'b000001??: instruction4(ir);
endcase
```

Case statement

- The following example shows another way to use a case statement to detect x and z values:

```
case (sig)
    1'bz: $display("signal is floating");
    1'bx: $display("signal is unknown");
    default: $display("signal is %b", sig);
endcase
```

Casex statement

- The following is an example of the casex statement. In this case, if r = 8'b01100110, then the task stat2 is called.

```
reg [7:0] r, mask;
mask = 8'bx0x0x0x0;
casex (r ^ mask)
        8'b001100xx: stat1;
        8'b1100xx00: stat2;
        8'b00xx0011: stat3;
        8'bxx010100: stat4;
endcase
```

Case statement with do-not-cares

- Two other types of case statements are provided to allow handling of do-not-care conditions in the case comparisons.
  - One of these treats high-impedance values (z) as do-not-cares (**casez**),
  - the other treats both high-impedance and unknown (x) values as do-not-cares (**casex**).

Looping example

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
always @(opa or opb)
        begin : mult
                reg [longsize:1] shift_opa, shift_opb;
                shift_opa = opa;
                shift_opb = opb;
                result = 0;
                repeat (size) begin
                        if (shift_opb[1])
                        result = result + shift_opa;
                        shift_opa = shift_opa << 1;
                        shift_opb = shift_opb >> 1;
                end
        end
```

Evaluation order use

- The following example demonstrates the usage by modeling a 3-bit priority encoder

```
        reg [2:0] encode ;
        case (1)
                encode[2] : $display("Select Line 2") ;
                encode[1] : $display("Select Line 1") ;
                encode[0] : $display("Select Line 0") ;
                default : $display("Error: One of the bits expected
ON");
        endcase
```

While Looping

```
begin : count1s
        reg [7:0] tempreg;
        count = 0;
        tempreg = rega;
        while (tempreg) begin
                if (tempreg[0])
                        count = count + 1;
                        tempreg = tempreg >> 1;
        end
end
```

Looping statements

- forever Continuously executes a statement.
- repeat Executes a statement a fixed number of times. If the expression evaluates to unknown or high impedance, it shall be treated as zero, and no statement shall be executed.
- while Executes a statement until an expression becomes false. If the expression starts out false, the statement shall not be executed at all.
- for Controls execution of its associated statement(s) by a three-step process, as follows:
  - a) Executes an assignment normally used to initialize a variable that controls the number of loops executed.
  - b) Evaluates an expression. If the result is zero, the for loop shall exit. If it is not zero, the for loop shall execute its associated statement(s) and then perform step c). If the expression evaluates to an unknown or high-impedance value, it shall be treated as zero.
  - c) Executes an assignment normally used to modify the value of the loop-control variable, then repeats step b).

For statement

- *For statement:* The for statement accomplishes the same results as the following pseudo-code that is based on the while loop:

```
begin
    initial_assignment;
    while (condition) begin
        statement
        step_assignment;
    end
end
```

- The for loop implements this logic while using only two lines, as shown in the pseudo-code below:

```
for (initial_assignment; condition; step_assignment)
    statement
```