```
• case (operator)
     2'd0 : z = x + y;
     2'd1 : z = x - y;
     2'd2 : z = x * y;
    default : $display("Invalid operation");
  endcase
• initial begin                    // 50 MHz clock
     clock = 0;
      forever #10 clock = ~clock;
  end                              // precision 1 ns
• repeat (2) @ (posedge clk); // add 2 clock delay
• repeat (5) @ (posedge clk) bus <=data;
          // evaluate data when the assignment is encountered
          // and assign to bus after 5 clocks.
• repeat (flag) begin              // looping
     .... action ...
  end
• while ( i < 10) begin
     .... action ...
  end
• for ( i = 0; i < 9; i = i+1 ) begin
     .... action ...
  end
• wait ( !oe ) #5 data = d_in;
• @ ( negedge clock ) q = d;
• begin                            // finishes at time #25
     #10 x = y;
     #15 a = b;
  end
• fork                             // finishes at time #15
     #10 x = y;
     #15 a = b;
  join
```

## 5. Gate Primitives

| | | | |
|---|---|---|---|
| **and** | (out, $in_1$, ..., $in_n$); | **nand** | (out, $in_1$, ..., $in_n$); |
| **or** | (out, $in_1$, ..., $in_n$); | **nor** | (out, $in_1$, ..., $in_n$); |
| **xor** | (out, $in_1$, ..., $in_n$); | **xnor** | (out, $in_1$, ..., $in_n$); |
| **buf** | ($out_1$, .. ,$out_n$, in); | **not** | ($out_1$, .. ,$out_n$, in); |
| **bufif0** | (out, in, control); | **bufif1** | (out, in, control); |
| **notif0** | (out, in, control); | **notif1** | (out, in, control); |
| **pullup** | (out); | **pulldown** | (out); |

## 6. Delays

```
Single delay                    : and  #5 my_and(...);
Rise/fall                       : and  #(5,7) my_and(...);
Rise/fall/Transport             : bufif1 #(10,15,5) my_buf(...);
All delays as min:typ:max       : or  #(4:5:6, 6:7:8) my_or (...);
```

**Compiler options for delays**

+maxdelays, +typdelays(default), +mindelays

e.g. **verilog** +maxdelays test.v

## 7. Operators

| | |
|---|---|
| {}, {{}} | concatenation |
| + - * / | arithmetic |
| % | modulus |
| > >= < <= | relational |
| ! | logical negation |
| && | logical and |
| \|\| | logical or |
| == | logical equality |
| != | logical inequality |
| === | case equality |
| !== | case inequality |
| ~ | bit-wise negation |
| & | bit-wise and |
| \| | bit-wise inclusive or |
| ^ | bit-wise exclusive or |
| ^~ or ~^ | bit-wise equivalence |
| & | reduction and |
| ~& | reduction nand |
| \| | reduction or |
| ~\| | reduction nor |
| ^ | reduction xor |
| ~^ or ^~ | reduction xnor |
| << | left shift |
| >> | right shift |
| ? : | conditional |
| or | Event or |

## 8. Specify Blocks

```
specify
          // specparam declarations (min:typ:max)
  specparam t_setup = 8:9:10, t_hold = 11:12:13;
          // timing constraints checks
  $setup (data, posedge clock, t_rise);
  $hold (posedge clear, data, t_hold);
          // simple pin to pin path delay
  (a => out) = 9; // => means parallel connection
          // edge sensitive pin to pin path delay
  (posedge clock => (out +: in)) = (10,8);
          // state dependent pin to pin path delay
  if (state_a = = 2'b01) (a, b *> out) = 15;
          // *> means full connection
endspecify
```

## 9. Memory Instantiation

```
module mem_test;
reg  [7:0]  memory [0:10];      // memory declaration
integer i ;
initial begin
  // reading the memory content file
  $readmemh ("contents.dat", memory);
  // display contents of initialized memory
  for  ( i = 0; i < 9; i = i+1 )
      $display ("Memory [%d] = %h", i, memory[i]);
end
endmodule

"contents.dat" contains
@02       ab    da
@06       00    01
```

- This simple memory model can be used for feeding input data values to simulation environment.
- $readmemb can be used for feeding binary values from contents file.

## 10. Blocking and Non-blocking Statements

```
// These blocking statements exhibit race condition.
always @(posedge clock)
    a = b;
always @(posedge clock)
    b = a;
// This Non-blocking statement  removes above race condition
// and gives true swapping operation
always @(posedge clock)
    a <= b;
always @(posedge clock)
    b <= a;
```

## 11. Functions and Tasks

**Function**

- A function can enable another function but not another task.
- Function always executes in 0 simulation time.
- Functions must not contain any delay, event, or timing control statement.
- Functions must have at least one input argument. They can have more than one input.
- Functions always return a single value. They cannot have output or inout argument.

e.g.

```
....
parity = calc_parity(addr);
....
function calc_parity;
input [31:0] address;
begin
    calc_parity = ^address;
end
endfunction
```

**Task**

- A task can enable other tasks and functions
- Tasks may execute in non-zero simulation time.
- Tasks may contain delay, event or timing control statements
- Tasks may have zero or more arguments of type input, output or inout.
- Tasks do not return with a value but can pass multiple values through output and inout arguments.

```
....
Cycle_read (read_in, oe_in, data, addr );
....
task  Cycle_read;
input         read, oe; // notice the sequence
output [7:0]  data;
input [15:0]  address;
begin
    #10 read_pin = read;
    # 05 oe_pin = oe;
    data = some_function(address);
end
```