1) Write a verilog code to swap contents of two registers with and without a temporary register?

With temp reg ;

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

Without temp reg;

```
always @ (posedge clock)
begin
a <= b;
b <= a;
end
```

Click to view more

Following is the Verilog code for flip-flop with a positive-edge clock.

```
module flop (clk, d, q);
input  clk, d;
output q;
reg    q;

always @(posedge clk)
begin
   q <= d;
end
      endmodule
```

Following is Verilog code for a flip-flop with a negative-edge clock and asynchronous clear.

```
module flop (clk, d, clr, q);
input  clk, d, clr;
output q;
```

```
reg    q;
always @(negedge clk or posedge clr)
       begin
   if (clr)
      q <= 1'b0;
   else
      q <= d;
end
       endmodule
```

Following is Verilog code for the flip-flop with a positive-edge clock and synchronous set.

```
       module flop (clk, d, s, q);
       input  clk, d, s;
       output q;
       reg    q;
       always @(posedge clk)
       begin
          if (s)
             q <= 1'b1;
          else
             q <= d;
       end
       endmodule
```

Following is Verilog code for the flip-flop with a positive-edge clock and clock enable.

```
module flop (clk, d, ce, q);
input  clk, d, ce;
output q;
reg    q;
always @(posedge clk)
       begin
   if (ce)
             q <= d;
end
       endmodule
```

Following is Verilog code for a 4-bit register with a positive-edge clock, asynchronous set and clock enable.

```
module flop (clk, d, ce, pre, q);
```

```
input        clk, ce, pre;
input  [3:0] d;
output [3:0] q;
reg    [3:0] q;
always @(posedge clk or posedge pre)
       begin
   if (pre)
      q <= 4'b1111;
   else if (ce)
      q <= d;
       end
       endmodule
```

Following is the Verilog code for a latch with a positive gate.

```
module latch (g, d, q);
       input  g, d;
       output q;
reg    q;
always @(g or d)
       begin
          if (g)
             q <= d;
       end
       endmodule
```

Following is the Verilog code for a latch with a positive gate and an asynchronous clear.

```
       module latch (g, d, clr, q);
       input  g, d, clr;
       output q;
       reg    q;
always @(g or d or clr)
       begin
          if (clr)
             q <= 1'b0;
          else if (g)
             q <= d;
       end
       endmodule
```

Following is Verilog code for a 4-bit latch with an inverted gate and an asynchronous preset.

```
module latch (g, d, pre, q);
input        g, pre;
input  [3:0] d;
output [3:0] q;
reg    [3:0] q;
always @(g or d or pre)
begin
   if (pre)
      q <= 4'b1111;
   else if (~g)
      q <= d;
end
endmodule
```

Following is Verilog code for a tristate element using a combinatorial process and always block.

```
module three_st (t, i, o);
input  t, i;
output o;
reg    o;
always @(t or i)
begin
   if (~t)
      o = i;
   else
      o = 1'bZ;
end
endmodule
```

Following is the Verilog code for a tristate element using a concurrent assignment.

```
module three_st (t, i, o);
input  t, i;
output o;
   assign o = (~t) ? i: 1'bZ;
      endmodule
```

Following is the Verilog code for a 4-bit unsigned up counter with asynchronous clear.

```
module counter (clk, clr, q);
input        clk, clr;
output [3:0] q;
```

```
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
   if (clr)
      tmp <= 4'b0000;
   else
      tmp <= tmp + 1'b1;
end
   assign q = tmp;
endmodule
```

Following is the Verilog code for a 4-bit unsigned down counter with synchronous set.

```
module counter (clk, s, q);
input        clk, s;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk)
begin
   if (s)
      tmp <= 4'b1111;
   else
      tmp <= tmp - 1'b1;
end
   assign q = tmp;
endmodule
```

Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous load from the primary input.

```
module counter (clk, load, d, q);
input        clk, load;
input  [3:0] d;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge load)
begin
   if (load)
      tmp <= d;
   else
      tmp <= tmp + 1'b1;
end
   assign q = tmp;
endmodule
```

Following is the Verilog code for a 4-bit unsigned up counter with a synchronous load with a constant.

```
module counter (clk, sload, q);
input        clk, sload;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk)
begin
   if (sload)
            tmp <= 4'b1010;
   else
      tmp <= tmp + 1'b1;
end
   assign q = tmp;
      endmodule
```

Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous clear and a clock enable.

```
module counter (clk, clr, ce, q);
input        clk, clr, ce;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
   if (clr)
      tmp <= 4'b0000;
   else if (ce)
      tmp <= tmp + 1'b1;
end
   assign q = tmp;
      endmodule
```

Following is the Verilog code for a 4-bit unsigned up/down counter with an asynchronous clear.

```
module counter (clk, clr, up_down, q);
input        clk, clr, up_down;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
   if (clr)
      tmp <= 4'b0000;
```

```
    else if (up_down)
       tmp <= tmp + 1'b1;
    else
       tmp <= tmp - 1'b1;
 end
    assign q = tmp;
       endmodule
```

Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset.

```
module counter (clk, clr, q);
input               clk, clr;
output signed [3:0] q;
reg    signed [3:0] tmp;
always @ (posedge clk or posedge clr)
begin
   if (clr)
      tmp <= 4'b0000;
   else
      tmp <= tmp + 1'b1;
end
   assign q = tmp;
endmodule
```

Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset and a modulo maximum.

```
module counter (clk, clr, q);
parameter MAX_SQRT = 4, MAX = (MAX_SQRT*MAX_SQRT);
input                 clk, clr;
output [MAX_SQRT-1:0] q;
reg    [MAX_SQRT-1:0] cnt;
always @ (posedge clk or posedge clr)
begin
   if (clr)
      cnt <= 0;
   else
      cnt <= (cnt + 1) %MAX;
end
   assign q = cnt;
endmodule
```

Following is the Verilog code for a 4-bit unsigned up accumulator with an asynchronous clear.

```
module accum (clk, clr, d, q);
input        clk, clr;
input  [3:0] d;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
   if (clr)
      tmp <= 4'b0000;
   else
      tmp <= tmp + d;
end
   assign q = tmp;
endmodule
```

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, serial in and serial out.

```
module shift (clk, si, so);
input        clk,si;
output       so;
reg    [7:0] tmp;
always @(posedge clk)
begin
   tmp    <= tmp << 1;
   tmp[0] <= si;
end
   assign so = tmp[7];
      endmodule
```

Following is the Verilog code for an 8-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

```
module shift (clk, ce, si, so);
input        clk, si, ce;
output       so;
reg    [7:0] tmp;
always @(negedge clk)
begin
   if (ce) begin
      tmp    <= tmp << 1;
      tmp[0] <= si;
   end
end
```

```
   assign so = tmp[7];
      endmodule
```

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, asynchronous clear, serial in and serial out.

```
module shift (clk, clr, si, so);
input        clk, si, clr;
output       so;
reg    [7:0] tmp;
always @(posedge clk or posedge clr)
begin
   if (clr)
      tmp <= 8'b00000000;
   else
      tmp <= {tmp[6:0], si};
end
   assign so = tmp[7];
      endmodule
```

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous set, a serial in and a serial out.

```
module shift (clk, s, si, so);
input        clk, si, s;
output       so;
reg    [7:0] tmp;
always @(posedge clk)
begin
   if (s)
      tmp <= 8'b11111111;
   else
      tmp <= {tmp[6:0], si};
end
   assign so = tmp[7];
      endmodule
```

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a serial in and a parallel out.

```
      module shift (clk, si, po);
      input        clk, si;
      output [7:0] po;
      reg    [7:0] tmp;
```

```
always @(posedge clk)
begin
   tmp <= {tmp[6:0], si};
end
   assign po = tmp;
endmodule
```

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, an asynchronous parallel load, a serial in and a serial out.

```
module shift (clk, load, si, d, so);
input        clk, si, load;
input  [7:0] d;
output       so;
reg    [7:0] tmp;
always @(posedge clk or posedge load)
begin
   if (load)
            tmp <= d;
   else
      tmp <= {tmp[6:0], si};
end
   assign so = tmp[7];
      endmodule
```

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous parallel load, a serial in and a serial out.

```
module shift (clk, sload, si, d, so);
input        clk, si, sload;
input  [7:0] d;
output       so;
reg    [7:0] tmp;
always @(posedge clk)
begin
   if (sload)
      tmp <= d;
   else
      tmp <= {tmp[6:0], si};
end
   assign so = tmp[7];
endmodule
```

Following is the Verilog code for an 8-bit shift-left/shift-right register with a positive-edge clock, a serial in and a serial out.

```
module shift (clk, si, left_right, po);
input        clk, si, left_right;
output       po;
reg    [7:0] tmp;
always @(posedge clk)
begin
   if (left_right == 1'b0)
      tmp <= {tmp[6:0], si};
   else
      tmp <= {si, tmp[7:1]};
end
   assign po = tmp;
      endmodule
```

Following is the Verilog code for a 4-to-1 1-bit MUX using an If statement.

```
module mux (a, b, c, d, s, o);
input        a,b,c,d;
input  [1:0] s;
output       o;
reg          o;
always @(a or b or c or d or s)
begin
   if (s == 2'b00)
      o = a;
   else if (s == 2'b01)
      o = b;
   else if (s == 2'b10)
      o = c;
   else
      o = d;
end
      endmodule
```

Following is the Verilog Code for a 4-to-1 1-bit MUX using a Case statement.

```
module mux (a, b, c, d, s, o);
input        a, b, c, d;
input  [1:0] s;
output       o;
reg          o;
always @(a or b or c or d or s)
begin
   case (s)
      2'b00   : o = a;
```

```
      2'b01   : o = b;
      2'b10   : o = c;
      default : o = d;
   endcase
end
      endmodule
```

Following is the Verilog code for a 3-to-1 1-bit MUX with a 1-bit latch.

```
      module mux (a, b, c, d, s, o);
      input        a, b, c, d;
      input  [1:0] s;
      output       o;
      reg          o;
      always @(a or b or c or d or s)
      begin
         if (s == 2'b00)
            o = a;
         else if (s == 2'b01)
            o = b;
         else if (s == 2'b10)
            o = c;
      end
      endmodule
```

Following is the Verilog code for a 1-of-8 decoder.

```
      module mux (sel, res);
      input  [2:0] sel;
      output [7:0] res;
      reg    [7:0] res;
      always @(sel or res)
      begin
         case (sel)
            3'b000  : res = 8'b00000001;
            3'b001  : res = 8'b00000010;
            3'b010  : res = 8'b00000100;
            3'b011  : res = 8'b00001000;
            3'b100  : res = 8'b00010000;
            3'b101  : res = 8'b00100000;
            3'b110  : res = 8'b01000000;
            default : res = 8'b10000000;
         endcase
      end
      endmodule
```

Following Verilog code leads to the inference of a 1-of-8 decoder.

```
module mux (sel, res);
input  [2:0] sel;
output [7:0] res;
reg    [7:0] res;
always @(sel or res) begin
   case (sel)
      3'b000  : res = 8'b00000001;
      3'b001  : res = 8'b00000010;
      3'b010  : res = 8'b00000100;
      3'b011  : res = 8'b00001000;
      3'b100  : res = 8'b00010000;
      3'b101  : res = 8'b00100000;
      // 110 and 111 selector values are unused
      default : res = 8'bxxxxxxxx;
   endcase
end
      endmodule
```

Following is the Verilog code for a 3-bit 1-of-9 Priority Encoder.

```
module priority (sel, code);
input  [7:0] sel;
output [2:0] code;
reg    [2:0] code;
always @(sel)
begin
   if (sel[0])
      code = 3'b000;
   else if (sel[1])
      code = 3'b001;
   else if (sel[2])
      code = 3'b010;
   else if (sel[3])
      code = 3'b011;
   else if (sel[4])
      code = 3'b100;
   else if (sel[5])
      code = 3'b101;
   else if (sel[6])
      code = 3'b110;
   else if (sel[7])
      code = 3'b111;
   else
      code = 3'bxxx;
end
      endmodule
```

Following is the Verilog code for a logical shifter.

```
module lshift (di, sel, so);
      input  [7:0] di;
input  [1:0] sel;
output [7:0] so;
reg    [7:0] so;
always @(di or sel)
begin
   case (sel)
      2'b00   : so = di;
      2'b01   : so = di << 1;
      2'b10   : so = di << 2;
      default : so = di << 3;
   endcase
end
      endmodule
```

Following is the Verilog code for an unsigned 8-bit adder with carry in.

```
module adder(a, b, ci, sum);
input  [7:0] a;
input  [7:0] b;
input        ci;
output [7:0] sum;

   assign sum = a + b + ci;

      endmodule
```

Following is the Verilog code for an unsigned 8-bit adder with carry out.

```
module adder(a, b, sum, co);
input  [7:0] a;
input  [7:0] b;
output [7:0] sum;
output       co;
wire   [8:0] tmp;

   assign tmp = a + b;
   assign sum = tmp [7:0];
   assign co  = tmp [8];

      endmodule
```

Following is the Verilog code for an unsigned 8-bit adder with carry in and carry out.

```
module adder(a, b, ci, sum, co);
input        ci;
input  [7:0] a;
input  [7:0] b;
output [7:0] sum;
output       co;
wire   [8:0] tmp;

   assign tmp = a + b + ci;
   assign sum = tmp [7:0];
   assign co  = tmp [8];

endmodule
```

Following is the Verilog code for an unsigned 8-bit adder/subtractor.

```
module addsub(a, b, oper, res);
input        oper;
input  [7:0] a;
input  [7:0] b;
output [7:0] res;
reg    [7:0] res;
always @(a or b or oper)
begin
   if (oper == 1'b0)
      res = a + b;
   else
      res = a - b;
      end
      endmodule
```

Following is the Verilog code for an unsigned 8-bit greater or equal comparator.

```
module compar(a, b, cmp);
input  [7:0] a;
input  [7:0] b;
output       cmp;

   assign cmp = (a >= b) ?  1'b1 : 1'b0;

      endmodule
```

Following is the Verilog code for an unsigned 8x4-bit multiplier.

```
module compar(a, b, res);
input  [7:0]  a;
input  [3:0]  b;
output [11:0] res;

   assign res = a * b;

endmodule
```

Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_1, pipe_2, pipe_3;

   assign mult_res = a_in * b_in;

   always @(posedge clk)
   begin
a_in   <= a;
      b_in   <= b;
      pipe_1 <= mult_res;
      pipe_2 <= pipe_1;
      pipe_3 <= pipe_2;
      mult   <= pipe_3;
   end
   endmodule
```

Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
```

```
      reg    [35:0] mult;
      reg    [17:0] a_in, b_in;
      reg    [35:0] mult_res;
      reg    [35:0] pipe_2, pipe_3;
      always @(posedge clk)
      begin
   a_in     <= a;
         b_in     <= b;
         mult_res <= a_in * b_in;
         pipe_2   <= mult_res;
         pipe_3   <= pipe_2;
         mult     <= pipe_3;
      end
      endmodule
```

Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_1, pipe_2, pipe_3;

   assign mult_res = a_in * b_in;

always @(posedge clk)
begin
   a_in   <= a;
   b_in   <= b;
   pipe_1 <= mult_res;
   pipe_2 <= pipe_1;
   pipe_3 <= pipe_2;
   mult   <= pipe_3;
end
      endmodule
```

Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
```

```
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
reg    [35:0] mult_res;
reg    [35:0] pipe_2, pipe_3;
always @(posedge clk)
begin
   a_in     <= a;
   b_in     <= b;
   mult_res <= a_in * b_in;
   pipe_2   <= mult_res;
   pipe_3   <= pipe_2;
   mult     <= pipe_3;
end
      endmodule
```

Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as shift registers.

```
module mult3(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_regs [3:0];

   assign mult_res = a_in * b_in;

always @(posedge clk)
begin
   a_in <= a;
   b_in <= b;
   {pipe_regs[3],pipe_regs[2],pipe_regs[1],pipe_regs[0]} <=
         {mult, pipe_regs[3],pipe_regs[2],pipe_regs[1]};
end
      endmodule
```

Following templates to implement Multiplier Adder with 2 Register Levels on Multiplier Inputs in Verilog.

```
module mvl_multaddsub1(clk, a, b, c, res);
input         clk;
input  [07:0] a;
input  [07:0] b;
input  [07:0] c;
```

```
output [15:0] res;
reg    [07:0] a_reg1, a_reg2, b_reg1, b_reg2;
wire   [15:0] multaddsub;
always @(posedge clk)
begin
   a_reg1 <= a;
   a_reg2 <= a_reg1;
   b_reg1 <= b;
   b_reg2 <= b_reg1;
end
   assign multaddsub = a_reg2 * b_reg2 + c;
   assign res = multaddsub;
      endmodule
```

Following is the Verilog code for resource sharing.

```
module addsub(a, b, c, oper, res);
input        oper;
input  [7:0] a;
input  [7:0] b;
input  [7:0] c;
output [7:0] res;
reg    [7:0] res;
always @(a or b or c or oper)
begin
   if (oper == 1'b0)
      res = a + b;
   else
      res = a - c;
end
      endmodule
```

Following templates show a single-port RAM in read-first mode.

```
module raminfr (clk, en, we, addr, di, do);
input        clk;
input        we;
input        en;
input  [4:0] addr;
input  [3:0] di;
output [3:0] do;
reg    [3:0] RAM [31:0];
reg    [3:0] do;
always @(posedge clk)
begin
   if (en) begin
      if (we)
        RAM[addr] <= di;
```

```
            do <= RAM[addr];
   end
end
      endmodule
```

Following templates show a single-port RAM in write-first mode.

```
module raminfr (clk, we, en, addr, di, do);
input        clk;
input        we;
input        en;
input  [4:0] addr;
input  [3:0] di;
output [3:0] do;
reg    [3:0] RAM [31:0];
reg    [4:0] read_addr;
always @(posedge clk)
begin
   if (en) begin
      if (we)
        RAM[addr] <= di;
            read_addr <= addr;
   end
end
   assign do = RAM[read_addr];
      endmodule
```

Following templates show a single-port RAM in no-change mode.

```
module raminfr (clk, we, en, addr, di, do);
input        clk;
input        we;
input        en;
input  [4:0] addr;
input  [3:0] di;
output [3:0] do;
reg    [3:0] RAM [31:0];
reg    [3:0] do;
always @(posedge clk)
begin
   if (en) begin
      if (we)
        RAM[addr] <= di;
      else
        do <= RAM[addr];
   end
end
```

```
endmodule
```

Following is the Verilog code for a single-port RAM with asynchronous read.

```
module raminfr (clk, we, a, di, do);
input        clk;
input        we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
always @(posedge clk)
begin
if (we)
      ram[a] <= di;
end
   assign do = ram[a];
endmodule
```

Following is the Verilog code for a single-port RAM with "false" synchronous read.

```
module raminfr (clk, we, a, di, do);
input        clk;
input        we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [3:0] do;
always @(posedge clk)
begin
   if (we)
      ram[a] <= di;
   do <= ram[a];
end
      endmodule
```

Following is the Verilog code for a single-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, di, do);
input        clk;
input        we;
input  [4:0] a;
```

```
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;
always @(posedge clk)
begin
   if (we)
      ram[a] <= di;
   read_a <= a;
end
   assign do = ram[read_a];
      endmodule
```

Following is the Verilog code for a single-port block RAM with enable.

```
module raminfr (clk, en, we, a, di, do);
input        clk;
input        en;
input        we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;
always @(posedge clk)
begin
   if (en) begin
      if (we)
         ram[a] <= di;
      read_a <= a;
   end
end
   assign do = ram[read_a];
endmodule
```

Following is the Verilog code for a dual-port RAM with asynchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input        clk;
input        we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
always @(posedge clk)
begin
```

```
   if (we)
      ram[a] <= di;
end
   assign spo = ram[a];
   assign dpo = ram[dpra];
      endmodule
```

Following is the Verilog code for a dual-port RAM with false synchronous read.

```
      module raminfr (clk, we, a, dpra, di, spo, dpo);
      input        clk;
      input        we;
      input  [4:0] a;
      input  [4:0] dpra;
      input  [3:0] di;
      output [3:0] spo;
      output [3:0] dpo;
      reg    [3:0] ram [31:0];
      reg    [3:0] spo;
      reg    [3:0] dpo;
always @(posedge clk)
      begin
         if (we)
            ram[a] <= di;

         spo = ram[a];
         dpo = ram[dpra];
      end
      endmodule
```

Following is the Verilog code for a dual-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input        clk;
input        we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;
reg    [4:0] read_dpra;
always @(posedge clk)
begin
   if (we)
      ram[a] <= di;
```

```
   read_a <= a;
   read_dpra <= dpra;
end
   assign spo = ram[read_a];
   assign dpo = ram[read_dpra];
      endmodule
```

Following is the Verilog code for a dual-port RAM with enable on each port.

```
module raminfr (clk, ena, enb, wea, addra, addrb, dia, doa, dob);
input        clk, ena, enb, wea;
input  [4:0] addra, addrb;
input  [3:0] dia;
output [3:0] doa, dob;
reg    [3:0] ram [31:0];
reg    [4:0] read_addra, read_addrb;
always @(posedge clk)
begin
   if (ena) begin
      if (wea) begin
         ram[addra] <= dia;
      end
   end
end

always @(posedge clk)
begin
   if (enb) begin
      read_addrb <= addrb;
   end
end
   assign doa = ram[read_addra];
   assign dob = ram[read_addrb];
      endmodule
```

Following is Verilog code for a ROM with registered output.

```
      module rominfr (clk, en, addr, data);
      input       clk;
      input       en;
      input [4:0] addr;
      output reg [3:0] data;
always @(posedge clk)
      begin
         if (en)
            case(addr)
               4'b0000: data <= 4'b0010;
               4'b0001: data <= 4'b0010;
```

```
                4'b0010: data <= 4'b1110;
                4'b0011: data <= 4'b0010;
                4'b0100: data <= 4'b0100;
                4'b0101: data <= 4'b1010;
                4'b0110: data <= 4'b1100;
                4'b0111: data <= 4'b0000;
                4'b1000: data <= 4'b1010;
                4'b1001: data <= 4'b0010;
                4'b1010: data <= 4'b1110;
                4'b1011: data <= 4'b0010;
                4'b1100: data <= 4'b0100;
                4'b1101: data <= 4'b1010;
                4'b1110: data <= 4'b1100;
                4'b1111: data <= 4'b0000;
                default: data <= 4'bXXXX;
             endcase
       end
       endmodule
```

Following is Verilog code for a ROM with registered address.

```
module rominfr (clk, en, addr, data);
input       clk;
input       en;
input [4:0] addr;
output reg [3:0] data;
reg   [4:0] raddr;
always @(posedge clk)
begin
   if (en)
      raddr <= addr;
end

always @(raddr)
begin
   if (en)
      case(raddr)
         4'b0000: data = 4'b0010;
         4'b0001: data = 4'b0010;
         4'b0010: data = 4'b1110;
         4'b0011: data = 4'b0010;
         4'b0100: data = 4'b0100;
         4'b0101: data = 4'b1010;
         4'b0110: data = 4'b1100;
         4'b0111: data = 4'b0000;
         4'b1000: data = 4'b1010;
         4'b1001: data = 4'b0010;
         4'b1010: data = 4'b1110;
         4'b1011: data = 4'b0010;
         4'b1100: data = 4'b0100;
         4'b1101: data = 4'b1010;
         4'b1110: data = 4'b1100;
```

```
         4'b1111: data = 4'b0000;
         default: data = 4'bXXXX;
      endcase
end
      endmodule
```

Following is the Verilog code for an FSM with a single process.

```
module fsm (clk, reset, x1, outp);
input        clk, reset, x1;
output       outp;
reg          outp;
reg    [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
   if (reset) begin
      state <= s1; outp <= 1'b1;
   end
   else begin
      case (state)
         s1: begin
                if (x1 == 1'b1) begin
                   state <= s2;
                          outp  <= 1'b1;
                end
                else begin
                   state <= s3;
                          outp  <= 1'b1;
                end
             end
         s2: begin
                state <= s4;
                       outp  <= 1'b0;
             end
         s3: begin
                state <= s4;
                       outp  <= 1'b0;
             end
         s4: begin
                state <= s1;
                       outp  <= 1'b1;
             end
      endcase
   end
end
      endmodule
```

Following is the Verilog code for an FSM with two processes.

```
module fsm (clk, reset, x1, outp);
input        clk, reset, x1;
output       outp;
reg          outp;
reg    [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
   if (reset)
      state <= s1;
   else begin
      case (state)
         s1: if (x1 == 1'b1)
                state <= s2;
             else
                state <= s3;
         s2: state <= s4;
         s3: state <= s4;
         s4: state <= s1;
      endcase
   end
end
always @(state) begin
   case (state)
      s1: outp = 1'b1;
      s2: outp = 1'b1;
      s3: outp = 1'b0;
      s4: outp = 1'b0;
   endcase
end
       endmodule
```

Following is the Verilog code for an FSM with three processes.

```
module fsm (clk, reset, x1, outp);
input        clk, reset, x1;
output       outp;
reg          outp;
reg    [1:0] state;
reg    [1:0] next_state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
   if (reset)
      state <= s1;
   else
```

```
      state <= next_state;
end

always @(state or x1)
begin
   case (state)
      s1: if (x1 == 1'b1)
             next_state = s2;
          else
             next_state = s3;
      s2: next_state = s4;
      s3: next_state = s4;
      s4: next_state = s1;
   endcase
       end
```

Home

2) Difference between blocking and non-blocking?(Verilog interview questions that is most commonly asked)

The Verilog language has two forms of the procedural assignment statement: blocking and non-blocking. The two are distinguished by the = and <= assignment operators. The blocking assignment statement (= operator) acts much like in traditional programming languages. The whole statement is done before control passes on to the next statement. The non-blocking (<= operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit. For example, the following Verilog program

```
// testing blocking and non-blocking assignment

module blocking;
reg [0:7] A, B;
initial begin: init1
A = 3;
#1 A = A + 1; // blocking procedural assignment
B = A + 1;

$display("Blocking: A= %b B= %b", A, B ); A = 3;
#1 A <= A + 1; // non-blocking procedural assignment
B <= A + 1;
#1 $display("Non-blocking: A= %b B= %b", A, B );
end
endmodule
```

produces the following output:

```
Blocking: A= 00000100 B= 00000101
Non-blocking: A= 00000100 B= 00000100
```

The effect is for all the non-blocking assignments to use the old values of the variables at the beginning of the current time unit and to assign the registers new values at the end of the current time unit. This reflects how register transfers occur in some hardware systems.
blocking procedural assignment is used for combinational logic and non-blocking procedural assignment for sequential

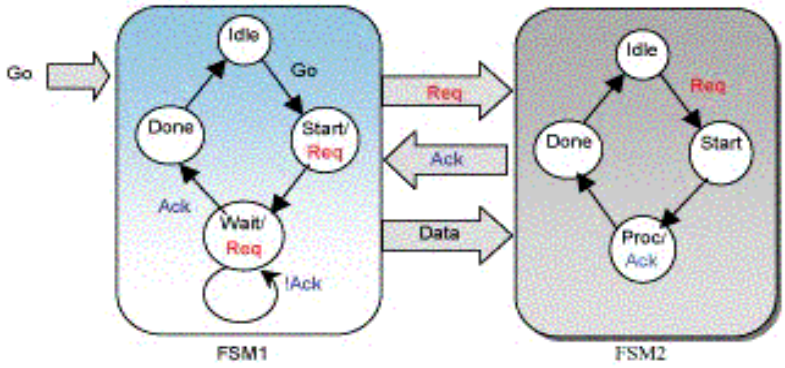
Click to view more

Clock Domain Crossing. . .

The following section explains clock domain interfacing

One of the biggest challenges of system-on-chip (SOC) designs is that different blocks operate on independent clocks. Integrating these blocks via the processor bus, memory ports, peripheral busses, and other interfaces can be troublesome because unpredictable behavior can result when the asynchronous interfaces are not properly synchronized

A very common and robust method for synchronizing multiple data signals is a handshake technique as shown in diagram below This is popular because the handshake technique can easily manage changes in clock frequencies, while minimizing latency at the crossing. However, handshake logic is significantly more complex than standard synchronization structures.



FSM1(Transmitter) asserts the req (request) signal, asking the receiver to accept the data on the data bus. FSM2(Receiver) generally a slow module asserts the ack (acknowledge) signal, signifying that it has accepted the data.

it has loop holes: when system Receiver samples the systems Transmitter req line and Transmitter samples system Receiver ack line, they have done it with respect to their internal clock, so there will be setup and hold time violation. To avoid this we go for double or triple stage synchronizers, which increase the MTBF and thus are immune to metastability to a good extent. The figure below shows how this is done.

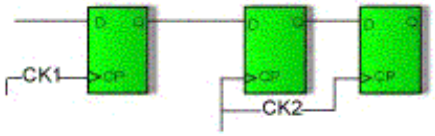


Figure 1a — Single-bit metastability sync

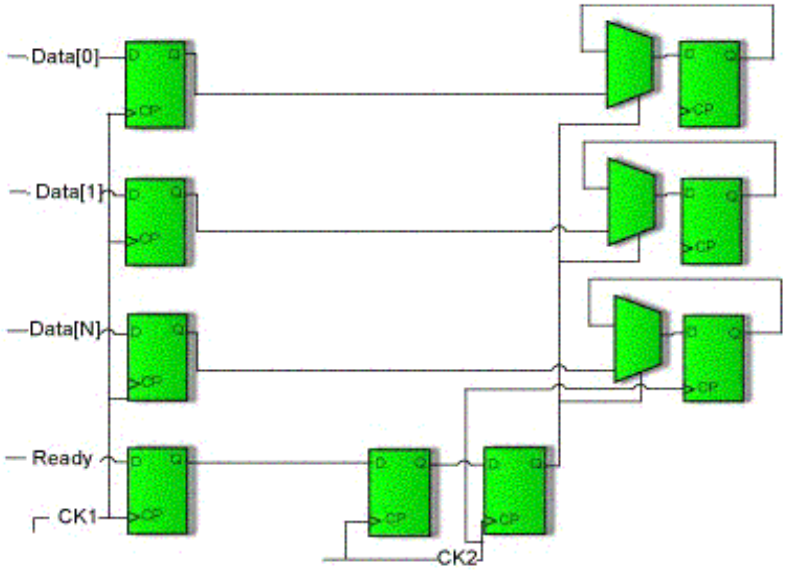


Figure 1b — Multi-bit sync

Blocking vs Non-Blocking. . .

self triggering blocks -

```
module osc2 (clk);
output clk;
reg clk;
initial #10 clk = 0;
always @(clk) #10 clk <= ~clk;
endmodule
```

After the first @(clk) trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the nonblocking assign updates event queue.
Before the nonblocking assign updates event queue is "activated," the @(clk) trigger statement is encountered and the always block again becomes sensitive to changes on the clk signal. When the nonblocking LHS value is updated later in the

same time step, the @(clk) is again triggered.

```
module osc1 (clk);
output clk;
reg clk;
initial #10 clk = 0;
always @(clk) #10 clk = ~clk;
endmodule
```

Blocking assignments evaluate their RHS expression and update their LHS value without interruption. The blocking assignment must complete before the @(clk) edge-trigger event can be scheduled. By the time the trigger event has been scheduled, the blocking clk assignment has completed; therefore, there is no trigger event from within the always block to trigger the @(clk) trigger.

Bad modeling: - (using blocking for seq. logic)

```
always @(posedge clk) begin
q1 = d;
q2 = q1;
q3 = q2;
end
```

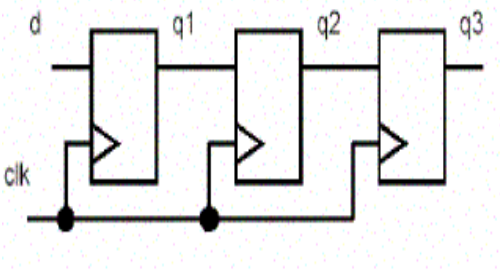
Race Condition

```
always @(posedge clk) q1=d;
always @(posedge clk) q2=q1;
always @(posedge clk) q3=q2;

always @(posedge clk) q2=q1;
always @(posedge clk) q3=q2;
always @(posedge clk) q1=d;

always @(posedge clk) begin
q3 = q2;
q2 = q1;
q1 = d;
end
```

Bad style but still works



Good modeling: -

```
always @(posedge clk) begin
q1 <= d;
q2 <= q1;
q3 <= q2;
end

always @(posedge clk) begin
q3 <= q2;
q2 <= q1;
q1 <= d;
end
```

No matter of sequence for Nonblocking

```
always @(posedge clk) q1<=d;
always @(posedge clk) q2<=q1;
always @(posedge clk) q3<=q2;

always @(posedge clk) q2<=q1;
always @(posedge clk) q3<=q2;
always @(posedge clk) q1<=d;
```

Good Combinational logic :- (Blocking)

```
always @(a or b or c or d) begin
tmp1 = a & b;
tmp2 = c & d;
y = tmp1 | tmp2;
end
```

Bad Combinational logic :- (Nonblocking)

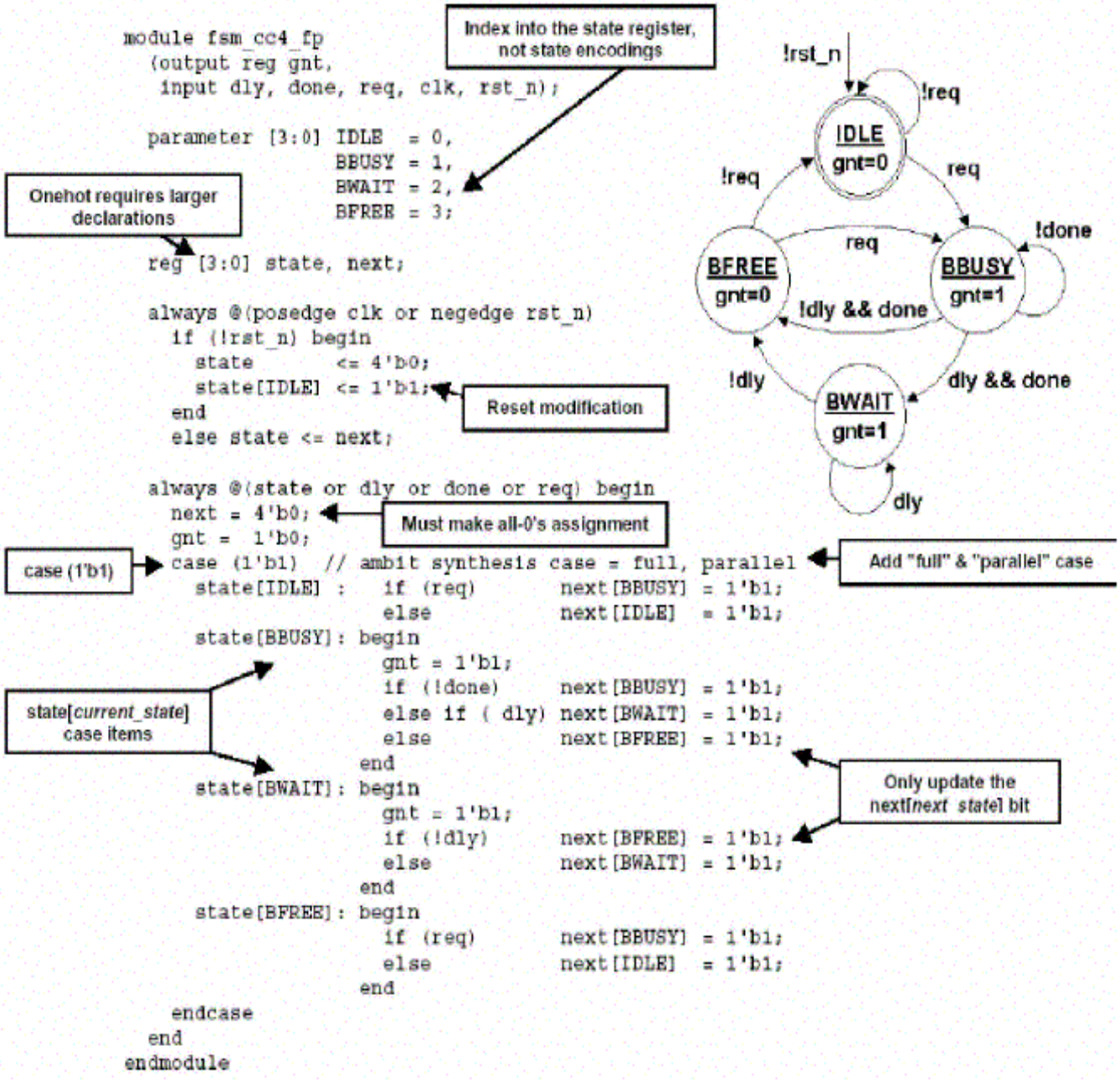
```
always @(a or b or c or d) begin will simulate incorrectly…
tmp1 <= a & b; need tmp1, tmp2 insensitivity
tmp2 <= c & d;
```

```
y <= tmp1 | tmp2;
end
```

Mixed design: -

Use Nonblocking assignment.
In case on multiple non-blocking assignments last one will win.

Verilog FSM




```
module fsm_cc4_fp
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [3:0] IDLE  = 0,
                  BBUSY = 1,
                  BWAIT = 2,
                  BFREE = 3;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state      <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt =  1'b0;
    case (1'b1)  // ambit synthesis case = full, parallel
      state[IDLE] :   if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
      state[BBUSY]: begin
                      gnt = 1'b1;
                      if (!done)     next[BBUSY] = 1'b1;
                      else if ( dly) next[BWAIT] = 1'b1;
                      else           next[BFREE] = 1'b1;
                    end
      state[BWAIT]: begin
                      gnt = 1'b1;
                      if (!dly)      next[BFREE] = 1'b1;
                      else           next[BWAIT] = 1'b1;
                    end
      state[BFREE]: begin
                      if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

Home

Tell me about verilog file I/O?

OPEN A FILE

```
integer file;
file = $fopenr("filename");
file = $fopenw("filename");
file = $fopena("filename");
```

The function $fopenr opens an existing file for reading. $fopenw opens a new file for writing, and $fopena opens a new file for writing where any data will be appended to the end of the file. The file name can be either a quoted string or a reg holding the file name. If the file was successfully opened, it returns an integer containing the file number (1..MAX_FILES) or NULL (0) if there was an error. Note that these functions are not the same as the built-in system function $fopen which opens a file for writing by $fdisplay. The files are opened in C with 'rb', 'wb', and 'ab' which allows reading and writing binary data on the PC. The 'b' is ignored on Unix.

CLOSE A FILE

```
integer file, r;
r = $fcloser(file);
r = $fclosew(file);
```

The function $fcloser closes a file for input. $fclosew closes a file for output. It returns EOF if there was an error, otherwise 0. Note that these are not the same as $fclose which closes files for writing.

Click to view more

Verilog File I/0,Verilog file handling

Verilog File I/0,Verilog file handling.

Overview

This application note describes how your Verilog model or testbench can read text and binary files to load memories, apply stimulus, and control simulation. Files can also be written. The format of the file I/O functions is based on the C stdio routines, such as fopen, fgetc, fprintf, and fscanf.

The Verilog language has a rich set of system functions to write files ($fdisplay, $fwrite, etc.) but only reads files with a single, fixed format ($readmem). In the past if you wanted to read a file that was not in $readmem format, you would have to learn the Programming Language Interface (PLI) and the C language, write C code to read the file and pass values into Verilog, then debug the combined C and Verilog code. In addition, the Verilog is limited to 32 open files at a time.

However, using the new file I/O system functions you can perform your file I/O directly from Verilog. You can write Verilog HDL to:

- read stimulus files to apply patterns to the inputs of a model
- read a file of expected values for comparison with your model
- read a script of commands to drive a simulation
- read either ASCII or binary files into Verilog registers and memories
- have hundreds of log files open simultaneously (though they are written to one at a time)

Code for all the examples in this file is included in the examples directory for the file I/O functions.

Note that these system tasks behave the same as the equivalent stdio routines. For example, $fscanf will skip over white-space, including blank lines, just like fscanf(). You can prototype code in C then convert it to Verilog.

Differences between fileio and IEEE-1364 Verilog-2001 (V2K) standard

The following list describes the differences between my file I/O package (fileio) and the IEEE-1364 Verilog-2001 standard (V2K).

1. In fileio $fopen has read, write, append variants:

```
file = $fopenr("filename");
file = $fopenw("filename");
file = $fopena("filename");
```

In V2K, there is a single $fopen for both multi-channel descriptors (MCD) and file descriptors (FD). Whether an FD or MCD is produced is indicated by the presence of a mode string added to $fopen in V2K:

```
file = $fopen("filename", "w");     // FD
file = $fopen("filename");          // MCD
```

Fileio supports the V2K $fopen format under a package compilation switch but that then blocks any use of MCDs since it hides the builtin $fopen.

2. Fileio $fclose has read and write variants that return a status:

```
r = $fcloser(file);
r = $fclosew(file);
```

In V2K, there is a single $fclose for both MCDs and FDs. It does not return a status. Errors can be determined by using $ferror.

Fileio supports the V2K $fclose format under a package compilation switch but that then blocks any use of MCDs since it hides the builtin $fclose.

3. Fileio $getchar is not directly supported in Verilog-2001. The operation can be done by using $fgetc('h8000_0000) which makes use of the reserved FD for stdin.
4. Fileio defines $fgets as:

```
r = $fgets(string, n, file);
```

V2K does not support a maximum count "n" of characters to read. Input in V2K always terminates at end of line and then string assignment to the target is done.

Fileio's $gets is not directly supported in Verilog 2001. The operation can be done by using:

```
$fgets(string, 'h8000_0000);
```

that makes use of the reserved FD for stdin.

5. Fileio $scanf is not directly supported in Verilog 2001. The operation can be done by using:

```
$fscanf('h8000_0000, format, args);
```

which makes use of the reserved FD for stdin.

6. Fileio does not support ? as an alias for X; V2K does.
7. Fileio does not support reading X or Z for %d format specification; V2K does.
8. Fileio supports %f, but not the synonyms %e and %g. V2K supports all three. Fileio does not support %f on in $sscanf; V2K supports all specifiers in $sscanf.
9. Fileio does not support %u, %z, %v, %t, or %m input format specifiers; V2K supports all of them.
10. Fileio supports special character input handling for \ (i.e. \\, \oNNN); V2K does not support this (not in LRM).

11. Fileio requires that $fread on a memory use "mem[0]" as the memory referend. V2K requires "mem" since "mem[0]" will be taken as a register read.
12. Fileio defines $sprintf and $fprintf which are not defined in V2K. V2K defines the $swrite family of tasks for string ouput and allows both MCDs and FDs in the $fwrite, $fdisplay, $fmonitor, and $fstrobe families of tasks. V2K supports $sformat where the difference in $sformat and $swrite is in the management of format specification strings. Fileio requires a single format string in $sprintf, etc; V2K follows the normal Verilog convention of treating any constant strings as format specifiers for $swrite. In V2K, all output format specifications are consistent and produce the same result independent of whether the target is a string, file, or standard output.
13. Fileio $ferror only returns a status. V2k $ferror takes a second parameter and stores the error string in that register. Additionally, V2K $ferror accepts a file descriptor with the value 0 and simple produces the most recent system error status.
14. Fileio requires an argument to $fflush; V2K permits a parameterless call and flushes all files (including MCD files) in that case. V2K $fflush supports either MCDs or FDs.
15. V2K supports $rewind which Fileio does not.
16. Fileio supports $fputc which V2K does not.
17. Fileio supports $feof which V2K does not. Some functions such as $fgetc return EOF (-1) but this is not the same.

File Input Functions

The file I/O system functions and tasks are based on the C stdio routines. For more information on the stdio routines, consult a C manual. The major differences between these system tasks and C are caused by the lack of a pointer variable in the Verilog language. Strings in Verilog are stored in registers, with 8 bits needed to store a single character.

OPEN A FILE

```
integer file;
file = $fopenr("filename");
file = $fopenw("filename");
file = $fopena("filename");
```

The function $fopenr opens an existing file for reading. $fopenw opens a new file for writing, and $fopena opens a new file for writing where any data will be appended to the end of the file. The file name can be either a quoted string or a reg holding the file name. If the file was successfully opened, it returns an integer containing the file number (1..MAX_FILES) or NULL (0) if there was an error. Note that these functions are not the same as the built-in system function $fopen which opens a file for writing by $fdisplay. The files are opened in C with 'rb', 'wb', and 'ab' which allows reading and writing binary data on the PC. The 'b' is ignored on Unix.

CLOSE A FILE

```
integer file, r;
r = $fcloser(file);
r = $fclosew(file);
```
The function $fcloser closes a file for input. $fclosew closes a file for output. It returns EOF if there was an error, otherwise 0. Note that these are not the same as $fclose which closes files for writing.

TEST FOR END OF FILE

```
integer file;
reg eof;
eof = $feof(file);
```
The function $feof tests for end of file. If an end-of-file has been reached while reading from the file, a non-zero value is returned; otherwise, a 0 is returned.

RETURN FILE STATUS

```
integer file;
reg error;
error = $ferror(file);
```

The function $ferror returns the error status of a file. If an error has occurred while reading from a file, $ferror returns a non-zero value, else 0. The error value is returned once, then reset to 0.

READ A SINGLE CHARACTER

```
integer file, char;
char = $fgetc(file);
char = $getc();
```
The function $fgetc reads a single character from the specified file and returns it. If the end-of-file is reached, $fgetc returns EOF. You should use a 32-bit register to hold the result from $fgetc to tell the difference between the character with the value 255 and EOF. $getc reads from stdin.

PUSH BACK A CHARACTER

```
integer file;
reg [7:0] char, r;
r = $ungetc(char, file);
```
The function $ungetc pushes the character back into the file stream. That character will be the next read by $fgetc. It returns the character if it was successfully pushed back or EOF if it fails.

Note that since there is no $ungetc for stdin in C, there will not be one in the file I/O package.

WRITE A SINGLE CHARACTER

```
integer stream, r, char;
r = $fputc(stream, char);
```

The function $fputc writes a single character to the specified file. It returns EOF if there was an error, 0 otherwise.

READ A STRING

```
integer file, n, r;
reg [n*8-1:0] string;
r = $fgets(string, n, file);
r = $gets(string);
```

The function $fgets reads a string from the file. Characters are read from the file into string until a newline is seen, end-of-file is reached, or n-1 characters have been read. If the end-of-file is encountered, $fgets returns a 0 and string is unchanged; otherwise, $fgets returns a 1. $gets reads from stdin.

The function $fgets has a string size limit of 1024 bytes. You can increase this by changing the constant MAX_STRING_SIZE.

The function $gets is no longer supported by default in fileio v3.4. If you want to use it, you must compile fileio.c with -DGETS. This is because some C compilers will give an error message when compiling fileio.c:

```
fileio.o: In function `fileio_gets_call`:
fileio.o: the gets function is dangerous and should not be used
```

You can either ignore this message, or stop using -DGETS to remove the gets function call from fileio.c.

READ FORMATTED TEXT

```
integer file, count;
count = $fscanf(file, format, args);
count = $sscanf(string, format, args);
count = $scanf(format, args);
```

The function $fscanf parses formatted text from the file according to the format and writes the results to args. $sscanf parses formatted text from a string. $scanf parses formated text from stdin. See a C reference manual for detailed information on fscanf, plus examples later in this note.

The format can be either a string constant or a reg. It can contain:

- Whitespace characters such as space, tab (\t), or newline (\n). One or more whitespace characters are treated as a single character, and can match zero or more whitespace characters from the input.
- Conversion specifications which start with a %. Next is an optional *, which suppresses assignment. Then is an optional field width in decimal. Lastly is the operator character as follows:
- b -- Binary values 0, 1, X, x, Z, z, _

- d -- Decimal values 0-9, _, no X, x, Z, or z. Note that negative numbers are NOT supported because of a Verilog language limitation.
- o -- Octal values 0-7, _, X, x, Z, z
- h or x -- Hexadecimal values, 0-9, A-F, a-f, _, X, x, Z, z
- c -- A single character
- f -- A floating point number, no _, X, x, Z, or z
- s -- A string
- % -- The percent character
- Other characters which must match the characters read from the file. Special characters are \" for the " character, \\ for the \ character, \oNNN is a single character whose ASCII value is specified by the octal number NNN, and %% for the character %.

The args is an optional list of registers to be assigned by $fscanf, $sscanf, and $scanf. There must be a register for each conversion operator (except those with %*). Bit subscripts are ignored.

Formatting & padding is closer to Verilog than C. For example, %x of 16'h24 is '0024', not '24', and %0x returns '24', not '0024'.

$fscanf, $sscanf, and $scanf return the number of successful assignments performed. If you do not want a return value from these routines, compile fileio.c (and veriuser.c for Cadence users) with -Dscanf_task. VCS users should switch from fileio.tab to fileio_task.tab

FIND THE FILE POSITION

```
integer file, position;
position = $ftell(file);
```

The function $ftell returns the position in the file for use by $fseek. If there is an error, it returns a -1.

POSITION A FILE

```
`define SEEK_SET 0
`define SEEK_CUR 1
`define SEEK_END 2
integer file, offset, position, r;
r = $fseek(file, 0, `SEEK_SET); /* Beginning */
r = $fseek(file, 0, `SEEK_CUR); /* No effect */
r = $fseek(file, 0, `SEEK_END); /* End of file */
r = $fseek(file, position, `SEEK_SET); /* Previous loc */
```

The function $fseek allows random access in a file. You can position at the beginning or end of a file, or use the position returned from $ftell.

READ BINARY DATA

```
integer r, file, start, count;
```

```
reg [15:0] mem[0:10], r16;
r = $fread(file, mem[0], start, count);
r = $fread(file, r16);
```
The function $fread reads a binary file into a Verilog memory. The first argument is either a register or a memory name, which must have a subscript, though the value of the subscript is ignored. start and count are optional.

By default $fread will store data in the first data location through the final location. For the memory up[10:20], the first location loaded would be up[10], then up[11]. For down[20:10], the first location would be down[10], then down[11].

start and count are ignored if $fread storing data in a reg instead of a memory. No warning is printed if the file contains more data than will fit in the memory.

start is the word offset from the lowest element in the memory. For start = 2 and the memory up[10:20], the first data would be loaded at up[12]. For the memory down[20:10] , the first location loaded would be down[12], then down[13].

$fread returns the number of elements read from the file, If $fread terminates early because of an error, it will return the number of elements successfully read. $feof can be used to determine whether the end-of-file was reached during $fread.

The data in the file is broken into bytes according to the width of the memory. An 8-bit wide memory takes one byte per location, while a 9-bit wide memory takes 2 bytes. Care should be taken when using memories with widths not evenly divisible by 8 as there may be gaps in the data in the memory vs. data in the file.

The $fread system task only works with NC-Verilog if you use the -MEMPACK switch as in:

```
ncverilog +ncvlogargs+-NOMEMPACK foo.v
```

WRITING A FORMATTED STRING

```
integer file, r, a, b;
reg [80*8:1] string;
file = $fopenw("output.log");
r = $sformat(string, "Formatted %d %x", a, b);
r = $sprintf(string, "Formatted %d %x", a, b);
r = $fprintf(file, "Formatted %d %x", a, b);
```
The functions $sformat and $sprintf writes a formatted string into a Verilog register. The two functions are identical. $fprintf writes a formatted string to a file. It has most, but not the formatting capabilites of C stdio package. The first argument is a register to receive the formatted data, and the second is a format string. Additional arguments may be included.

The supported formats include b, c, d, e, f, g, h, m, o, r, s, and x. %t for printing formatted time is NOT supported yet.

FLUSHING THE FILE STREAM

```
integer file, r;
file = $fopenw("output.log");
r = $fflush(file);
```
The function $fflush(stream) causes any buffered data waiting to be written for the named stream to be written to that file. If the stream is 0, all files open for writing are flushed.

Restrictions and Caveats

You should be aware of following restrictions in using these Verilog functions vs. the stdio functions in C, which are imposed by the Verilog language :

- Because these are Verilog system functions, you must always use the return value as in:

```
r = $fscanf(...)
```

- Verilog does not allow assignments inside a conditional. Thus the C code fragment:

```
while (c=fgetc(stream) != EOF) {
    <process input>
    }
```
turns into the Verilog code:
```
c = $fgetc(file);
while (c !== `EOF)
    begin
    <process input>
    c = $fgetc(file);
    end
```

- $fgets and $gets return only a single bit, 0 = error, 1 = no error, unlike fgets / gets in C, which return a string pointer.
- $fread is very different from fread in C. The order of arguments is different, and the arguments are oriented towards writing to a Verilog memory instead of a C character string. See page 5 for more info.
- $fscanf can not be used to read binary files, or any files with null characters. $fprintf can not be used to write binary files with null characters. Because of the string processing that $fscanf uses, a null in the middle of an ASCII field will prematurely terminate the field.

In addition, $save / $restart are loosely supported with VCS on SunOS. In a test, $feof never returned EOF when running from a save file, but $fgetc did. On other simulators and hardware platforms you can mimic these functions using $ftell and $fseek to find the position in a file and later jump to that location.

The maximum number of files (MAX_FILES) is set in the C code to 12. The maximum string size is 1000 characters. There is no known limit to the number of conversion operators in $fscanf or $sscanf.

Reading pattern files

This first example shows how to read input stimulus from a text file.

This is the pattern file - read_pattern.pat , included in the examples directory:

```
// This is a pattern file
// time bin dec hex
10: 001 1 1
20.0: 010 20 020
50.02: 111 5 FFF
62.345: 100 4 DEADBEEF
75.789: XXX 2 ZzZzZzZz
```

Note that the binary and hexadecimal values have X and Z values, but these are not allowed in the decimal values. You can use white space when formatting your file to make it more readable. Lastly, any line beginning with a / is treated as a comment.

The module read_pattern.v reads the time for the next pattern from an ASCII file. It then waits until the absolute time specified in the input file, and reads the new values for the input signals (bin, dec, hex). The time in the file is a real value and, when used in a delay, is rounded according to the timescale directive. Thus the time 75.789 is rounded to 75.79 ns.

```
`timescale 1ns / 10 ps
`define EOF 32'hFFFF_FFFF
`define NULL 0
`define MAX_LINE_LENGTH 1000

module read_pattern;
integer file, c, r;
reg [3:0] bin;
reg [31:0] dec, hex;
real real_time;
reg [8*`MAX_LINE_LENGTH:0] line; /* Line of text read from file */

initial
    begin : file_block
    $timeformat(-9, 3, "ns", 6);
    $display("time bin decimal hex");
    file = $fopenr("read_pattern.pat");
    if (file == `NULL) // If error opening file
        disable file_block; // Just quit

    c = $fgetc(file);
    while (c != `EOF)
        begin
        /* Check the first character for comment */
        if (c == "/")
            r = $fgets(line, `MAX_LINE_LENGTH, file);
        else
            begin
            // Push the character back to the file then read the next time
            r = $ungetc(c, file);
            r = $fscanf(file," %f:\n", real_time);
```

```
            // Wait until the absolute time in the file, then read stimulus
            if ($realtime > real_time)
                $display("Error - absolute time in file is out of order - %t",
                        real_time);
                else
                    #(real_time - $realtime)
                        r = $fscanf(file," %b %d %h\n",bin,dec,hex);
                end // if c else
            c = $fgetc(file);
        end // while not EOF

    r = $fcloser(file);
    end // initial

// Display changes to the signals
always @(bin or dec or hex)
    $display("%t %b %d %h", $realtime, bin, dec, hex);

endmodule // read_pattern
```

Comparing outputs with expected results

The following model, compare.v, reads a file containing both stimulus and expected results. The input signals are toggled at the beginning of a clock cycle and the output is compared just before the end of the cycle.

```
`define EOF 32'hFFFF_FFFF
`define NULL 0
`define MAX_LINE_LENGTH 1000
module compare;
integer file, r;
reg a, b, expect, clock;
wire out;
reg [`MAX_LINE_LENGTH*8:1];
parameter cycle = 20;

initial
    begin : file_block
    $display("Time Stim Expect Output");
    clock = 0;

    file = $fopenr("compare.pat");
    if (file == `NULL)
        disable file_block;

    r = $fgets(line, MAX_LINE_LENGTH, file); // Skip comments
    r = $fgets(line, MAX_LINE_LENGTH, file);

    while (!$feof(file))
        begin
        // Wait until rising clock, read stimulus
        @(posedge clock)
```

```
        r = $fscanf(file, " %b %b %b\n", a, b, expect);

        // Wait just before the end of cycle to do compare
        #(cycle - 1)
        $display("%d %b %b %b %b", $stime, a, b, expect, out);
        $strobe_compare(expect, out);
        end // while not EOF

    r = $fcloser(file);
    $stop;
    end // initial

always #(cycle / 2) clock = !clock; // Clock generator

and #4 (out, a, b); // Circuit under test
endmodule // compare
```

Reading script files

Sometimes a detailed simulation model for a device is not available, such as a microprocessor. As a substitute, you can write a bus-functional model which reads a script of bus transactions and performs these actions. The following, script.v, reads a file with commands plus data values.

```
`define EOF 32'hFFFF_FFFF
`define NULL 0
module script;
integer file, r;
reg [80*8:1] command;
reg [31:0] addr, data;

initial
    begin : file_block
    clock = 0;

    file = $fopenr("script.txt");
    if (file == `NULL)
        disable file_block;

    while (!$feof(file))
        begin
        r = $fscanf(file, " %s %h %h \n", command, addr, data);
        case (command)
        "read":
            $display("READ mem[%h], expect = %h", addr, data);
        "write":
            $display("WRITE mem[%h] = %h", addr, data);
        default:
            $display("Unknown command '%0s'", command);
        endcase
        end // while not EOF

    r = $fcloser(file);
    end // initial
endmodule // script
```

The file script.txt is the script read by the above model:

```
read 9 0
write 300a feedface
read 2FF xxxxxxxx
bad
```

Reading data files into memories

Reading a formatted ASCII file is easy with the system tasks. The following is an example of reading a binary file into a Verilog memory. $fread can also read a file one word at a time and copy the word into memory, but this is about 100 times slower than using $fread to read the entire array directly.

This is the file load_mem.v

```
`define EOF 32'HFFFF_FFFF
`define MEM_SIZE 200_000

module load_mem;

integer file, i;
reg [7:0] mem[0:`MEM_SIZE];
reg [80*8:1] file_name;

initial
    begin
    file_name = "data.bin";
    file = $fopenr(file_name);
    i = $fread(file, mem[0]);
    $display("Loaded %0d entries \n", i);
    i = $fcloser(file);
    $stop;
    end

endmodule // load_mem
```

The file data.bin contains the 200 binary values 0 to 199. You can look at the program data.c which generated the file. To dump out the binary file in Unix use the command od data.bin

Linking with VCS

Note that VCS 6.1 and later supports the IEEE-1364 2001 standard. To use the file I/O system functions with VCS, you will need to:

1. Compile fileio.c with the command:
   ```
   cc -c fileio.c -I$VCS_HOME/include
   ```

   On the DEC/Alpha use:
   ```
   cc -c fileio.c -I$VCS_HOME/`vcs -platform`/lib -taso -xtaso_short
   ```

On Windows, use the Microsoft C++ compiler included with VCS:

```
cl -c -Zp4 fileio.c
```

The -Zp4 switch tells the compiler to use longword alignment. Note that the compiler produces fileio.obj, not fileio.o. In the example below, if you compile on Windows, change the file extension.

Note that the system variable "include" must contain a reference to the VCS include files, such as:

```
c:\vcs422\Windows_NT\lib
```

2. Compile your Verilog model with the fileio routines on Unix with:

```
% vcs load_mem.v fileio.o -P fileio.tab -R
                        Chronologic VCS (TM)
              Version 5.1 -- Tue Jan 11 09:00:41 2000
               Copyright (c) 1991-2000 by Synopsys Inc.
                        ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys Inc.
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

Compiling load_mem.v
Top Level Modules:
load_mem
( cd csrc ; make -f Makefile DEFAULT_RUNTIME=TRUE product )
../simv up to date
Chronologic VCS simulator copyright 1991-2000
Contains Synopsys proprietary information.
Compiler version 5.1; Runtime version 5.1;  Jan 11 09:00 2000

Loaded 200 entries

$stop at time 0 Scope: load_mem File: load_mem.v Line: 13
cli_0 >
```

Linking with Verilog-XL

Verilog-XL does not natively support the IEEE-1364 2001 tasks except through this PLI application.

Note: this information is based on an older version of Verilog-XL. Send me an update if you have one.

To use the file I/O system functions with Verilog-XL, you will need to:

1. Modify your veriuser.c to point to the system functions in fileio.c . You should copy these two files into your current directory from the examples directory.

2. Type vconfig to generate a script to link Verilog-XL. Use the following table to choose your responses.

Running vconfig	
Prompt issued from vconfig :	**You type:**
Please enter the name of the output script	cr_vlog
Please choose a target	1 Stand Alone
Please choose how to link in PLI application	4 Static with user PLI application
What do you want to name the Verilog-XL target?	verilog_fileio
Do you want to compile for the SimVision environment?	n
Do you want to include GR_WAVES	n
Do you want to include the Simulation History Manager	n
The LAI interface in no longer supported	n
Do you want to include the LMSI HARDWARE MODELER interface software in this executable?	return
Do you want to include the Verilog Mixed-Signal interface software in this executable?	return
Do you want to include the Standard Delay File Annotator in this executable?	return
The user template file 'veriuser.c' must always be included in the link statement. What is the path name of this file?	./veriuser.c
Please list any other user files to be linked with this Verilog-XL ... terminating with a single '.'	./fileio.c . (period)

- Add the switch **-Dverilogxl** to cr_vlog to compile fileio.c:
- ```
  cc -Dverilogxl -o verilog_read $1 $2 -g \
  ...
  fileio.c \
  ...
  ```


- Run cr_vlog and link Verilog-XL, producing the new executable verilog_read
- Run the new executable and simulate your models with calls to the read functions.
- You should see the following printed when you run verilog_read to simulate the model load_mem.v :
  ```

```
% ./verilog_read load_mem.v
VERILOG-XL 2.1.12 Jun 21, 1995 14:55:32

Copyright (c) 2000 Cadence Design Systems, Inc. All Rights Reserved.
.
.
.
<<< Routines for file read linked in >>
Compiling source file "load_mem.v"
Highest level modules:
load_mem

Loaded 200 entries

L13 "load_mem.v": $stop at simulation time 0
Type ? for help
C1>
```

Linking with MTI

MTI's ModelSim 5.5 and later offer native support these IEEE-1364 2001 system tasks.

Create fileio.so, the shareable object:

```
make MTI=1 fileio.so
```

or:

```
gcc -c -g fileio.c -DMTI -I$MTI_PATH/include
ld -G -Bdynamic -o fileio.so fileio.o
```

Compile and run your design with:

```
rm -rf work
vlib work


vlog test1.v    # Your Verilog code here
vsim -c test1 -pli fileio.so -do "run -all"
```

You might have to set the environment variable LD_LIBRARY_PATH to point to the directory where fileio.so resides.

Linking with NC-Verilog

NC-Verilog 3.3 and later offer native support for these IEEE-1364 2001 system tasks.

NC-Verilog does not support the tf_nodeinfo PLI call which is used in several places by fileio.c. If you want to use these system functions with NC-Verilog, compile fileio.c with the -DNCVerilog switch. This will disable the $fread routine, and passing some strings using a register, such as the format string to $fscanf.

If you have any information on linking with NC-Verilog, please pass it on to me!

You can use makefile_fileio_ncv to compile and link with NC-Verilog. I make no promises!

Home

3) Difference between task and function?

Function:
A function is unable to enable a task however functions can enable other functions.
A function will carry out its required duty in zero simulation time. (The program time will not be incremented during the function routine)
Within a function, no event, delay or timing control statements are permitted
In the invocation of a function their must be at least one argument to be passed.
Functions will only return a single value and can not use either output or inout statements.

Tasks:
Tasks are capable of enabling a function as well as enabling other versions of a Task
Tasks also run with a zero simulation however they can if required be executed in a non zero simulation time.
Tasks are allowed to contain any of these statements.
A task is allowed to use zero or more arguments which are of type output, input or inout.
A Task is unable to return a value but has the facility to pass multiple values via the output and inout statements .

4) Difference between inter statement and intra statement delay?

```
//define register variables
reg a, b, c;

//intra assignment delays
initial
begin
a = 0; c = 0;
b = #5 a + c; //Take value of a and c at the time=0, evaluate
//a + c and then wait 5 time units to assign value
//to b.
end

//Equivalent method with temporary variables and regular delay control
initial
begin
a = 0; c = 0;
temp_ac = a + c;
```

```
#5 b = temp_ac; //Take value of a + c at the current time and
//store it in a temporary variable. Even though a and c
//might change between 0 and 5,
//the value assigned to b at time 5 is unaffected.
end
```

5) What is delta simulation time?

6) Difference between $monitor,$display & $strobe?

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools like cwaves?. $display and $strobe display once every time they are executed, whereas $monitor displays every time one of its parameters changes.
The difference between $display and $strobe is that $strobe displays the parameters at the very end of the current simulation time unit rather than exactly where it is executed. The format string is like that in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time), %m (hierarchy level). %5d, %5b etc. would give exactly 5 spaces for the number instead of the space needed. Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

Syntax:
$display (“format_string”, par_1, par_2, ...);
$strobe (“format_string”, par_1, par_2, ...);
$monitor (“format_string”, par_1, par_2, ...);

7) What is difference between Verilog full case and parallel case?

A "full" case statement is a case statement in which all possible case-expression binary patterns can be matched to a case item or to a case default. If a case statement does not include a case default and if it is possible to find a binary case expression that does not match any of the defined case items, the case statement is not "full."
A "parallel" case statement is a case statement in which it is only possible to match a case expression to one and only one case item. If it is possible to find a case expression that would match more than one case item, the matching case items are called "overlapping" case items and the case statement is not "parallel."

8) What is meant by inferring latches,how to avoid it?

Consider the following :

```
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
2'd0 : out = i0;
2'd1 : out = i1;
```

```
2'd2 : out = i2;
endcase
```

in a case statement if all the possible combinations are not compared and default is also not specified like in example above a latch will be inferred ,a latch is inferred because to reproduce the previous value when unknown branch is specified.
For example in above case if {s1,s0}=3 , the previous stored value is reproduced for this storing a latch is inferred.
The same may be observed in IF statement in case an ELSE IF is not specified.
To avoid inferring latches make sure that all the cases are mentioned if not default condition is provided.

9) Tell me how blocking and non blocking statements get executed?

Execution of blocking assignments can be viewed as a one-step process:
1. Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement. A blocking assignment "blocks" trailing assignments in the same always block from occurring until after the current assignment has been completed

Execution of nonblocking assignments can be viewed as a two-step process:
1. Evaluate the RHS of nonblocking statements at the beginning of the time step. 2. Update the LHS of nonblocking statements at the end of the time step.

10) Variable and signal which will be Updated first?

Signals

11) What is sensitivity list?

The sensitivity list indicates that when a change occurs to any one of elements in the list change, begin…end statement inside that always block will get executed.

12) In a pure combinational circuit is it necessary to mention all the inputs in sensitivity disk? if yes, why?

Yes in a pure combinational circuit is it necessary to mention all the inputs in sensitivity disk other wise it will result in pre and post synthesis mismatch.

13) Tell me structure of Verilog code you follow?

A good template for your Verilog file is shown below.

```
// timescale directive tells the simulator the base units and precision of the simulation
`timescale 1 ns / 10 ps
module name (input and outputs);
```

```
// parameter declarations
parameter parameter_name = parameter value;
// Input output declarations
input in1;
input in2; // single bit inputs
output [msb:lsb] out; // a bus output
// internal signal register type declaration - register types (only assigned within always
statements). reg register variable 1;
reg [msb:lsb] register variable 2;
// internal signal. net type declaration - (only assigned outside always statements) wire net
variable 1;
// hierarchy - instantiating another module
reference name instance name (
.pin1 (net1),
.pin2 (net2),
..
.pinn (netn)
);
// synchronous procedures
always @ (posedge clock)
begin
..
end
// combinatinal procedures
always @ (signal1 or signal2 or signal3)
begin
..
end
assign net variable = combinational logic;
endmodule
```

14) Difference between Verilog and vhdl?

Compilation
VHDL. Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each design unit in it's own system file in which case separate compilation should not be an issue.

Verilog. The Verilog language is still rooted in it's native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.

Data types
VHDL. A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

Verilog. Compared to VHDL, Verilog data types a re very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit. Objects, that is signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of it's simplicity.

Design reusability
VHDL. Procedures and functions may be placed in a package so that they are avail able to any design-unit that wishes to use them.

Verilog. There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include compiler directive.

15) What are different styles of Verilog coding I mean gate-level,continuous level and others explain in detail?

16) Can you tell me some of system tasks and their purpose?

$display, $displayb, $displayh, $displayo, $write, $writeb, $writeh, $writeo.
The most useful of these is $display.This can be used for displaying strings, expression or values of variables.
Here are some examples of usage.
$display("Hello oni");
--- output: Hello oni
$display($time) // current simulation time.
--- output: 460
counter = 4'b10;
$display(" The count is %b", counter);
--- output: The count is 0010
$reset resets the simulation back to time 0; $stop halts the simulator and puts it in interactive mode where the
user can enter commands; $finish exits the simulator back to the operating system

17) Can you list out some of enhancements in Verilog 2001?

In earlier version of Verilog ,we use 'or' to specify more than one element in sensitivity list . In Verilog 2001, we can use comma as shown in the example below.

```
// Verilog 2k example for usage of comma
always @ (i1,i2,i3,i4)
```

Verilog 2001 allows us to use star in sensitive list instead of listing all the variables in RHS of combo logics . This removes typo mistakes and thus avoids simulation and synthesis mismatches,

Verilog 2001 allows port direction and data type in the port list of modules as shown in the example below

```
module memory (
input r,
input wr,
input [7:0] data_in,
input [3:0] addr,
output [7:0] data_out
);
```

18)Write a Verilog code for synchronous and asynchronous reset?

Synchronous reset, synchronous means clock dependent so reset must not be present in sensitivity disk eg:

```
always @ (posedge clk )

begin if (reset)
. . . end
```

Asynchronous means clock independent so reset must be present in sensitivity list.

Eg

```
Always @(posedge clock or posedge reset)
begin
if (reset)
. . . end
```

19) What is pli?why is it used?

Programming Language Interface (PLI) of Verilog HDL is a mechanism to interface Verilog programs with programs written in C language. It also provides mechanism to access internal databases of the simulator from the C program.

PLI is used for implementing system calls which would have been hard to do otherwise (or impossible) using Verilog syntax. Or, in other words, you can take advantage of both the paradigms - parallel and hardware related features of Verilog and sequential flow of C - using PLI.

20) There is a triangle and on it there are 3 ants one on each corner and are free to move along sides of triangle what is probability that they will collide?

Ants can move only along edges of triangle in either of direction, let's say one is represented by 1 and another by 0, since there are 3 sides eight combinations are possible, when all ants are going in same direction they won't collide that is 111 or 000 so probability of not collision is 2/8=1/4 or collision probability is 6/8=3/4

STAGE 2

Verilog interview Questions
How to write FSM is verilog?

there r mainly 4 ways 2 write fsm code
1) using 1 process where all input decoder, present state, and output decoder r combine in one process.
2) using 2 process where all comb ckt and sequential ckt separated in different process
3) using 2 process where input decoder and persent state r combine and output decoder seperated in other process
4) using 3 process where all three, input decoder, present state and output decoder r separated in 3 process.

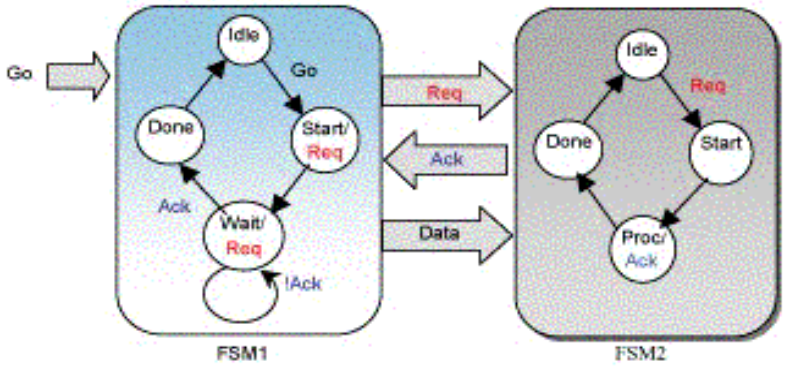
Click to view more

Clock Domain Crossing. . .

The following section explains clock domain interfacing

One of the biggest challenges of system-on-chip (SOC) designs is that different blocks operate on independent clocks. Integrating these blocks via the processor bus, memory ports, peripheral busses, and other interfaces can be troublesome because unpredictable behavior can result when the asynchronous interfaces are not properly synchronized

A very common and robust method for synchronizing multiple data signals is a handshake technique as shown in diagram below This is popular because the handshake technique can easily manage changes in clock frequencies, while minimizing latency at the crossing. However, handshake logic is significantly more complex than standard synchronization structures.



FSM1(Transmitter) asserts the req (request) signal, asking the receiver to accept the data on the data bus. FSM2(Receiver) generally a slow module asserts the ack (acknowledge) signal, signifying that it has accepted the data.

it has loop holes: when system Receiver samples the systems Transmitter req line and Transmitter samples system Receiver ack line, they have done it with respect to their internal clock, so there will be setup and hold time violation. To avoid this we go for double or triple stage synchronizers, which increase the MTBF and thus are immune to metastability to a good extent. The figure below shows how this is done.

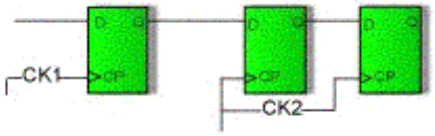


Figure 1a — Single-bit metastability sync

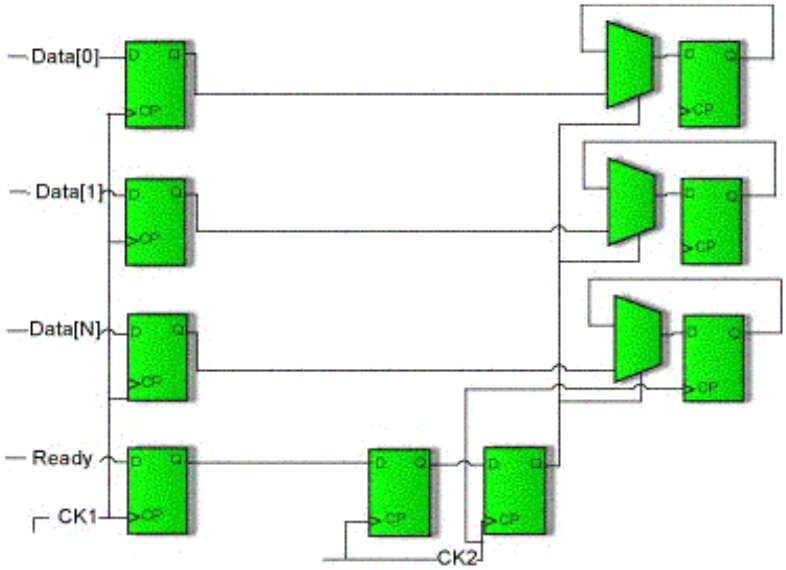


Figure 1b — Multi-bit sync

Blocking vs Non-Blocking. . .

self triggering blocks -

```
module osc2 (clk);
output clk;
reg clk;
initial #10 clk = 0;
always @(clk) #10 clk <= ~clk;
endmodule
```

After the first @(clk) trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the nonblocking assign updates event queue.
Before the nonblocking assign updates event queue is "activated," the @(clk) trigger statement is encountered and the always block again becomes sensitive to changes on the clk signal. When the nonblocking LHS value is updated later in the

same time step, the @(clk) is again triggered.

```
module osc1 (clk);
output clk;
reg clk;
initial #10 clk = 0;
always @(clk) #10 clk = ~clk;
endmodule
```

Blocking assignments evaluate their RHS expression and update their LHS value without interruption. The blocking assignment must complete before the @(clk) edge-trigger event can be scheduled. By the time the trigger event has been scheduled, the blocking clk assignment has completed; therefore, there is no trigger event from within the always block to trigger the @(clk) trigger.

Bad modeling: - (using blocking for seq. logic)

```
always @(posedge clk) begin
q1 = d;
q2 = q1;
q3 = q2;
end
```

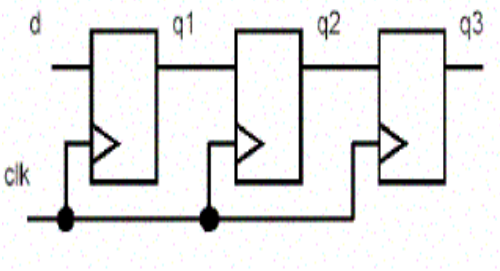
Race Condition

```
always @(posedge clk) q1=d;
always @(posedge clk) q2=q1;
always @(posedge clk) q3=q2;

always @(posedge clk) q2=q1;
always @(posedge clk) q3=q2;
always @(posedge clk) q1=d;

always @(posedge clk) begin
q3 = q2;
q2 = q1;
q1 = d;
end
```

Bad style but still works



Good modeling: -

```
always @(posedge clk) begin
q1 <= d;
q2 <= q1;
q3 <= q2;
end

always @(posedge clk) begin
q3 <= q2;
q2 <= q1;
q1 <= d;
end
```

No matter of sequence for Nonblocking

```
always @(posedge clk) q1<=d;
always @(posedge clk) q2<=q1;
always @(posedge clk) q3<=q2;

always @(posedge clk) q2<=q1;
always @(posedge clk) q3<=q2;
always @(posedge clk) q1<=d;
```

Good Combinational logic :- (Blocking)

```
always @(a or b or c or d) begin
tmp1 = a & b;
tmp2 = c & d;
y = tmp1 | tmp2;
end
```

Bad Combinational logic :- (Nonblocking)

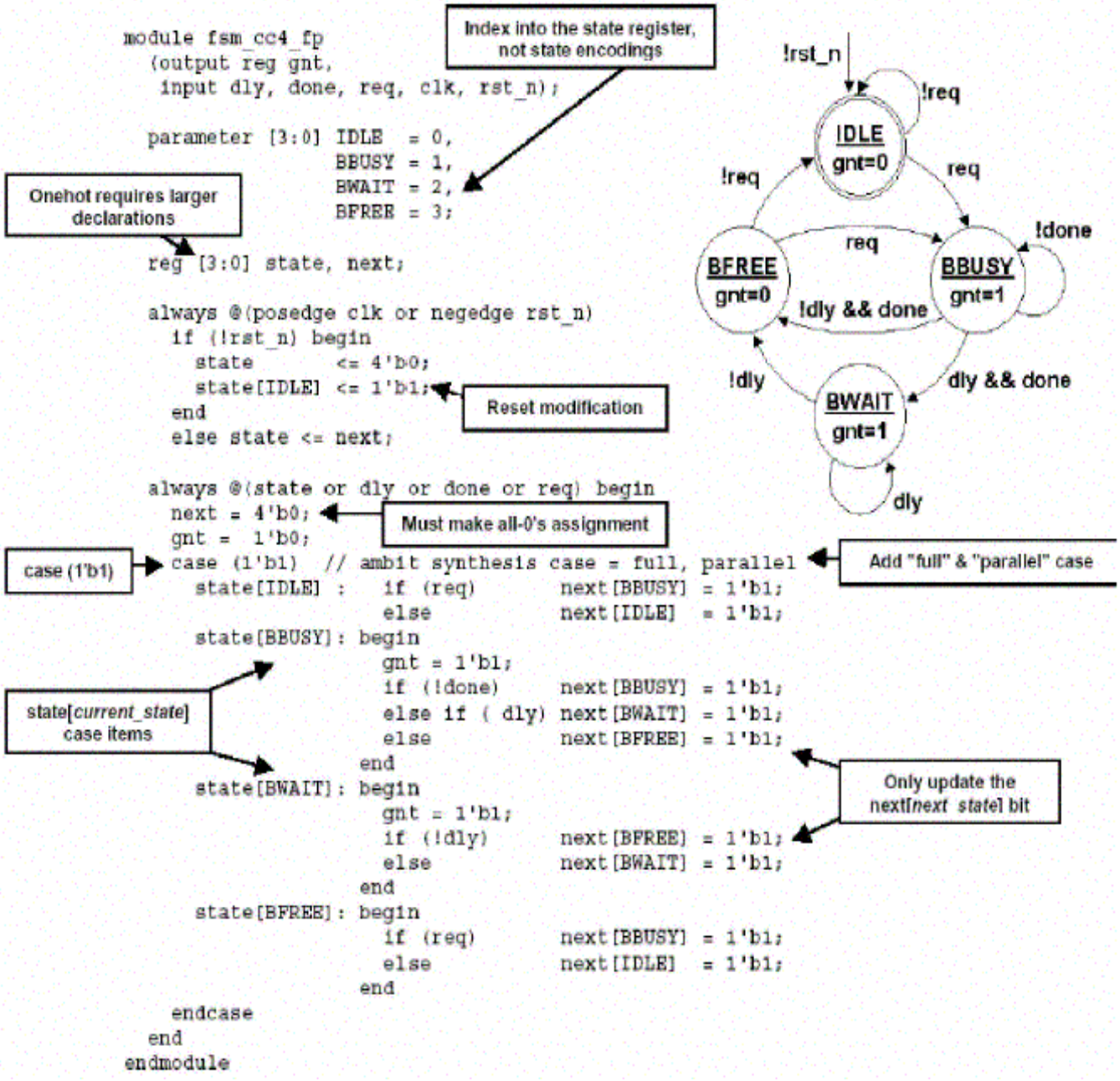
```
always @(a or b or c or d) begin will simulate incorrectly…
tmp1 <= a & b; need tmp1, tmp2 insensitivity
tmp2 <= c & d;
```

```
y <= tmp1 | tmp2;
end
```

Mixed design: -

Use Nonblocking assignment.
In case on multiple non-blocking assignments last one will win.

Verilog FSM




```
module fsm_cc4_fp
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [3:0] IDLE  = 0,
                  BBUSY = 1,
                  BWAIT = 2,
                  BFREE = 3;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state      <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt =  1'b0;
    case (1'b1)  // ambit synthesis case = full, parallel
      state[IDLE] :   if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
      state[BBUSY]: begin
                      gnt = 1'b1;
                      if (!done)     next[BBUSY] = 1'b1;
                      else if ( dly) next[BWAIT] = 1'b1;
                      else           next[BFREE] = 1'b1;
                    end
      state[BWAIT]: begin
                      gnt = 1'b1;
                      if (!dly)      next[BFREE] = 1'b1;
                      else           next[BWAIT] = 1'b1;
                    end
      state[BFREE]: begin
                      if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

Home

(Also refer to Tutorial section for more)

Verilog interview Questions
21)What is difference between freeze deposit and force?

$deposit(variable, value);
This system task sets a Verilog register or net to the specified value. variable is the register or net to be changed; value is the new value for the register or net. The value remains until there is a subsequent driver transaction or another $deposit task for the same register or net. This system task operates identically to the ModelSim force -deposit command.

The force command has -freeze, -drive, and -deposit options. When none of these is specified, then -freeze is assumed for unresolved signals and -drive is assumed for resolved signals. This is designed to provide compatibility with force files. But if you prefer -freeze as the default for both resolved and unresolved signals.

Verilog interview Questions
22)Will case infer priority register if yes how give an example?

yes case can infer priority register depending on coding style
reg r;
// Priority encoded mux,

```
always @ (a or b or c or select2)
begin
r = c;
case (select2)
2'b00: r = a;
2'b01: r = b;
endcase
end
```

Verilog interview Questions

23)Casex,z difference,which is preferable,why?

CASEZ :
Special version of the case statement which uses a Z logic value to represent don't-care bits.
CASEX :
Special version of the case statement which uses Z or X logic values to represent don't-care bits.

CASEZ should be used for case statements with wildcard don't cares, otherwise use of CASE is required; CASEX should never be used.
This is because:
Don't cares are not allowed in the "case" statement. Therefore casex or casez are required. Casex will automatically match any x or z with anything in the case statement. Casez will only match z's -- x's require an absolute match.

Verilog interview Questions

24)Given the following Verilog code, what value of "a" is displayed?

```
always @(clk) begin
a = 0;
a <= 1;
$display(a);
end
```

This is a tricky one! Verilog scheduling semantics basically imply a
four-level deep queue for the current simulation time:
1: Active Events (blocking statements)
2: Inactive Events (#0 delays, etc)
3: Non-Blocking Assign Updates (non-blocking statements)
4: Monitor Events ($display, $monitor, etc).
Since the "a = 0" is an active event, it is scheduled into the 1st "queue".
The "a <= 1" is a non-blocking event, so it's placed into the 3rd queue.
Finally, the display statement is placed into the 4th queue. Only events in the active queue are completed this sim cycle, so the "a = 0" happens, and then the display shows a = 0. If we were to look at the value of a in the next sim cycle, it would show 1.

25) What is the difference between the following two lines of Verilog code?

#5 a = b;
a = #5 b;

#5 a = b; Wait five time units before doing the action for "a = b;".
a = #5 b; The value of b is calculated and stored in an internal temp register,After five time units, assign this stored value to a.

26)What is the difference between:

c = foo ? a : b;
and
if (foo) c = a;
else c = b;

The ? merges answers if the condition is "x", so for instance if foo = 1'bx, a = 'b10, and b = 'b11, you'd get c = 'b1x. On the other hand, if treats Xs or Zs as FALSE, so you'd always get c = b.

27)What are Intertial and Transport Delays ??

28)What does `timescale 1 ns/ 1 ps signify in a verilog code?

'timescale directive is a compiler directive.It is used to measure simulation time or delay time. Usage : `timescale / reference_time_unit : Specifies the unit of measurement for times and delays. time_precision: specifies the precision to which the delays are rounded off.

29) What is the difference between === and == ?

output of "==" can be 1, 0 or X.
output of "===" can only be 0 or 1.
When you are comparing 2 nos using "==" and if one/both the numbers have one or more bits as "x" then the output would be "X" . But if use "===" outpout would be 0 or 1.
e.g A = 3'b1x0
B = 3'b10x
A == B will give X as output.
A === B will give 0 as output.
"==" is used for comparison of only 1's and 0's .It can't compare Xs. If any bit of the input is X output will be X
"===" is used for comparison of X also.

30)How to generate sine wav using verilog coding style?

A: The easiest and efficient way to generate sine wave is using CORDIC Algorithm.

31) What is the difference between wire and reg?

Net types: (wire,tri)Physical connection between structural elements. Value assigned by a

continuous assignment or a gate output. Register type: (reg, integer, time, real, real time) represents abstract data storage element. Assigned values only within an always statement or an initial statement. The main difference between wire and reg is wire cannot hold (store) the value when there no connection between a and b like a->b, if there is no connection in a and b, wire loose value. But reg can hold the value even if there in no connection. Default values:wire is Z,reg is x.

32)How do you implement the bi-directional ports in Verilog HDL?

```
module bidirec (oe, clk, inp, outp, bidir);

// Port Declaration
input oe;
input clk;
input [7:0] inp;
output [7:0] outp;
inout [7:0] bidir;
reg [7:0] a;
reg [7:0] b;
assign bidir = oe ? a : 8'bZ ;
assign outp = b;
// Always Construct
always @ (posedge clk)
begin
b <= bidir;
a <= inp;
end
endmodule
```

34)what is verilog case (1) ?

```
wire [3:0] x;
always @(...) begin
case (1'b1)
x[0]: SOMETHING1;
x[1]: SOMETHING2;
x[2]: SOMETHING3;
x[3]: SOMETHING4;
endcase
end
```

The case statement walks down the list of cases and executes the first one that matches. So here, if the lowest 1-bit of x is bit 2, then something3 is the statement that will get executed (or selected by the logic).

35) Why is it that "if (2'b01 & 2'b10)..." doesn't run the true case?

This is a popular coding error. You used the bit wise AND operator (&) where you meant to use the logical AND operator (&&).

36)What are Different types of Verilog Simulators ?

There are mainly two types of simulators available.

Event Driven
Cycle Based

Event-based Simulator:

This Digital Logic Simulation method sacrifices performance for rich functionality: every active signal is calculated for every device it propagates through during a clock cycle. Full Event-based simulators support 4-28 states; simulation of Behavioral HDL, RTL HDL, gate, and transistor representations; full timing calculations for all devices; and the full HDL standard. Event-based simulators are like a Swiss Army knife with many different features but none are particularly fast.

Cycle Based Simulator:

This is a Digital Logic Simulation method that eliminates unnecessary calculations to achieve huge performance gains in verifying Boolean logic:

1.) Results are only examined at the end of every clock cycle; and
2.) The digital logic is the only part of the design simulated (no timing calculations). By limiting the calculations, Cycle based Simulators can provide huge increases in performance over conventional Event-based simulators.
Cycle based simulators are more like a high speed electric carving knife in comparison because they focus on a subset of the biggest problem: logic verification.
Cycle based simulators are almost invariably used along with Static Timing verifier to compensate for the lost timing information coverage.

37)What is Constrained-Random Verification ?

Introduction

As ASIC and system-on-chip (SoC) designs continue to increase in size and complexity, there is an equal or greater increase in the size of the verification effort required to achieve functional coverage goals. This has created a trend in RTL verification techniques to employ constrained-random verification, which shifts the emphasis from hand-authored tests to utilization of compute resources. With the corresponding emergence of faster, more complex bus standards to handle the massive volume of data traffic there has also been a renewed significance for verification IP to speed the time taken to develop advanced testbench environments that include

randomization of bus traffic.

Directed-Test Methodology

Building a directed verification environment with a comprehensive set of directed tests is extremely time-consuming and difficult. Since directed tests only cover conditions that have been anticipated by the verification team, they do a poor job of covering corner cases. This can lead to costly re-spins or, worse still, missed market windows.

Traditionally verification IP works in a directed-test environment by acting on specific testbench commands such as read, write or burst to generate transactions for whichever protocol is being tested. This directed traffic is used to verify that an interface behaves as expected in response to valid transactions and error conditions. The drawback is that, in this directed methodology, the task of writing the command code and checking the responses across the full breadth of a protocol is an overwhelming task. The verification team frequently runs out of time before a mandated tape-out date, leading to poorly tested interfaces. However, the bigger issue is that directed tests only test for predicted behavior and it is typically the unforeseen that trips up design teams and leads to extremely costly bugs found in silicon.

Constrained-Random Verification Methodology

The advent of constrained-random verification gives verification engineers an effective method to achieve coverage goals faster and also help find corner-case problems. It shifts the emphasis from writing an enormous number of directed tests to writing a smaller set of constrained-random scenarios that let the compute resources do the work. Coverage goals are achieved not by the sheer weight of manual labor required to hand-write directed tests but by the number of processors that can be utilized to run random seeds. This significantly reduces the time required to achieve the coverage goals.

Scoreboards are used to verify that data has successfully reached its destination, while monitors snoop the interfaces to provide coverage information. New or revised constraints focus verification on the uncovered parts of the design under test. As verification progresses, the simulation tool identifies the best seeds, which are then retained as regression tests to create a set of scenarios, constraints, and seeds that provide high coverage of the design.

STAGE 3

What are the differences between blocking and nonblocking assignments?

While both blocking and nonblocking assignments are procedural assignments, they differ in behaviour with respect to simulation and logic
synthesis as follows:

Table 1-3. Differences between blocking and nonblocking assignments

Blocking assignments	Nonblocking assignments
In a blocking assignment, the evaluation of the expression on the RHS is updated to the LHS variable autonomously based on the delay value (either 0 if no delay specified, or scheduled as a future event if a non-0 value is specified)	Nonblocking assignment to LHS is *scheduled* to occur when the next evaluation cycle occurs in simulation and not immediately. Updates are not available immediately within the same time unit
When multiple blocking assignments are present in a process, the trailing assignments are blocked from occurring until the current assignment is completed	Multiple nonblocking assignments can be scheduled to occur concurrently on the next evaluation cycle in simulation
There is a possibility of race conditions on the variables of blocking assignments if assignments happen to it from two processes concurrently	The race conditions are avoided as the updated value is assigned after evaluation
Recommended to use within combinatorial **always** blocks	Recommended to use within the sequential **always** blocks
Can be used in procedural assignments like ***initial***, ***always*** and continuous assignments to nets like ***assign*** statements	Can be used only in the procedural blocks like ***initial*** and ***always***; Continuous assignment to nets like the ***assign*** statement is not permitted
Represented by "=" operator sign between LHS and RHS	Represented by "<=" operator sign between LHS and RHS

How can I model a bi-directional net with assignments influencing both source and destination?

The assign statement constitutes a continuous assignment. The changes on the RHS of the

statement immediately reflect on the LHS net. However, any changes on the LHS don't get reflected on the RHS. For example, in the following statement, changes to the rhs net will update the lhs net, but not vice versa.

System Verilog has introduced a keyword alias, which can be used only on nets to have a two-way assignment. For example, in the following code, any changes to the rhs is reflected to the lhs , and vice versa.

```
wire rhs , lhs
assign lhs=rhs;
```

System Verilog has introduced a keyword alias, which can be used only on nets to have a two-way assignment. For example, in the following code, any changes to the rhs is reflected to the lhs , and vice versa.

```
module test ();
wire rhs,lhs;


alias lhs=rhs;
```

In the above example, any change to either side of the net gets reflected on the other side.

Are tasks and functions re-entrant, and how are they different from static task and function calls?

In Verilog-95, tasks and functions were not re-entrant. From Verilog version 2001 onwards, the tasks and functions are reentrant. The reentrant tasks have a keyword automatic between the keyword task and the name of the task. The presence of the keyword automatic replicates and allocates the variables within a task dynamically for each task entry during concurrent task calls, i.e., the values don’t get overwritten for each task call. Without the keyword, the variables are allocated statically, which means these variables are shared across different task calls, and can hence get overwritten by each task call.

Reentrant task	Static task
Has the keyword ***automatic*** between the ***task*** keyword and identifier	*Doesn't* have the keyword ***autom*** between the ***task*** keyword and the identifier
Variables declared within the task are allocated dynamically for each concurrent task call	Variable declarations within the ta are allocated statically
All variables will be replicated in each concurrent call to store state specific to that invocation	Each concurrent call to the task w OVERWRITE the statically allocated local variables of the tas from all other concurrent calls to t task
Variables declared are de-allocated at the end of task invocation	Variables retain their values betw invocations
Task items cannot be accessed by hierarchical inferences	Task items can be accessed by hierarchical inferences
Task items shall be allocated new across all uses of the task executing concurrently	Task items can be shared across a uses of the task executing concurrently

How can I override variables in an automatic task?

By default, all variables in a module are static, i.e., these variables will be replicated for all

instances of a module. However, in the case of task and function, either the task/function itself or the variables within them can be defined as static or automatic. The following explains the inferences through different combinations of the task/function and/or its variables, declared either as static or automatic:

No automatic definition of task/function or its variables This is the Verilog-1995 format, wherein the task/function and its variables were implicitly static. The variables are allocated only once. Without the mention of the automatic keyword, multiple calls to task/function will override their variables.

static task/function definition

System Verilog introduced the keyword static. When a task/function is explicitly defined as static, then its variables are allocated only once, and can be overridden. This scenario is exactly the same scenario as before.

automatic task/function definition

From Verilog-2001 onwards, and included within SystemVerilog, when the task/function is declared as automatic, its variables are also implicitly automatic. Hence, during multiple calls of the task/function, the variables are allocated each time and replicated without any overwrites.

static task/function and automatic variables

SystemVerilog also allows the use of automatic variables in a static task/function. Those without any changes to automatic variables will remain implicitly static. This will be useful in scenarios wherein the implicit static variables need to be initialised before the task call, and the automatic variables can be allocated each time.

automatic task/function and static variables

SystemVerilog also allows the use of static variables in an automatic task/function. Those without any changes to static variables will remain implicitly automatic. This will be useful in scenarios wherein the static variables need to be updated for each call, whereas the rest can be allocated each time.

What are the rules governing usage of a Verilog function?

The following rules govern the usage of a Verilog function construct:

A function cannot advance simulation-time, using constructs like #, @. etc.
A function shall not have nonblocking assignments.
A function without a range defaults to a one bit reg for the return value.
It is illegal to declare another object with the same name as the function in the scope where the function is declared.

How do I prevent selected parameters of a module from being overridden during instantiation?

If a particular parameter within a module should be prevented from being overridden, then it should be declared using the localparam construct, rather than the parameter construct. The localparam construct has been introduced from Verilog-2001. Note that a localparam variable is fully identical to being defined as a parameter, too. In the following example, the localparam construct is used to specify num_bits, and hence trying to override it directly gives an error message.

```
module localparam_list (addr, data);
parameter width = 32;
parameter depth = 64;
localparam num_bits = width * depth;
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
...
endmodule
```

Note, however, that, since the width and depth are specified using the parameter construct, they can be overridden during instantiation or using defparam, and hence will indirectly override the num_bits values. In general, localparam constructs are useful in defining new and localized identifiers whose values are derived from regular parameters.

What are the pros and cons of specifying the parameters using the defparam construct vs. specifying during instantiation?

The advantages of specifying parameters during instantiation method are:

All the values to all the parameters don't need to be specified. Only those parameters that are assigned the new values need to be specified. The unspecified parameters will retain their default

values specified within its module definition.

The order of specifying the parameter is not relevant anymore, since the parameters are directly specified and linked by their name.

The disadvantage of specifying parameter during instantiation are:

This has a lower precedence when compared to assigning using defparam.

The advantages of specifying parameter assignments using defparam are:

This method always has precedence over specifying parameters during instantiation.

All the parameter value override assignments can be grouped inside one module and together in one place, typically in the top-level testbench itself.

When multiple defparams for a single parameter are specified, the parameter takes the value of the last defparam statement encountered in the source if, and only if, the multiple defparam's are in the same file. If there are defparam's in different files that override the same parameter, the final value of the parameter is indeterminate.

The disadvantages of specifying parameter assignments using defparam are:

The parameter is typically specified by the scope of the hierarchies underneath which it exists. If a particular module gets ungrouped in its hierarchy, [sometimes necessary during synthesis], then the scope to specify the parameter is lost, and is unspecified. B

For example, if a module is instantiated in a simulation testbench, and its internal parameters are then overridden using hierarchical defparam constructs (For example, defparam U1.U_fifo.width = 32;). Later, when this module is synthesized, the internal hierarchy within U1 may no longer exist in the gate-level netlist, depending upon the synthesis strategy chosen. Therefore post-synthesis simulation will fail on the hierarchical defparam override.

Can there be full or partial no-connects to a multi-bit port of a module during its instantiation?

No. There cannot be full or partial no-connects to a multi-bit port of a module during instantiation

What happens to the logic after synthesis, that is driving an unconnected output port that is left open (, that is, noconnect) during its module instantiation?

An unconnected output port in simulation will drive a value, but this value does not propagate to any other logic. In synthesis, the cone of any combinatorial logic that drives the unconnected output will get optimized away during boundary optimisation, that is, optimization by synthesis tools across hierarchical boundaries.

How is the connectivity established in Verilog when connecting wires of different widths?

When connecting wires or ports of different widths, the connections are right-justified, that is, the rightmost bit on the RHS gets connected to the rightmost bit of the LHS and so on, until the MSB of either of the net is reached.

Can I use a Verilog function to define the width of a multi-bit port, wire, or reg type?

The width elements of ports, wire or reg declarations require a constant in both MSB and LSB. Before Verilog 2001, it is a syntax error to specify a function call to evaluate the value of these widths. For example, the following code is erroneous before Verilog 2001 version.

reg [port1(val1:vla2) : port2 (val3:val4)] reg1;

In the above example, get_high and get_low are both function calls of evaluating a constant result for MSB and LSB respectively. However, Verilog-2001 allows the use of a function call to evaluate the MSB or LSB of a width declaration

What is the implication of a combinatorial feedback loops in design testability?

The presence of feedback loops should be avoided at any stage of the design, by periodically checking for it, using the lint or synthesis tools. The presence of the feedback loop causes races and hazards in the design, and 104 RTL Design
leads to unpredictable logic behavior. Since the loops are delay-dependent, they cannot be tested with any ATPG algorithm. Hence, combinatorial loops should be avoided in the logic.

What are the various methods to contain power during RTL coding?

Any switching activity in a CMOS circuit creates a momentary current flow from VDD to GND during logic transition, when both N and P type transistors are ON, and, hence, increases power consumption.
The most common storage element in the designs being the synchronous FF, its output can change whenever its data input toggles, and the clock triggers. Hence, if these two elements can be asserted in a controlled fashion, so that the data is presented to the D input of the FF only when required, and the clock is also triggered only when required, then it will reduce the switching activity, and, automatically the power.

The following bullets summarize a few mechanisms to reduce the power consumption:

- Reduce switching of the data input to the Flip-Flops.
- Reduce the clock switching of the Flip-Flops.
- Have area reduction techniques within the chip, since the number of gates/Flip-Flops that toggle can be reduced.

How do I model Analog and Mixed-Signal blocks in Verilog?

First, this is a big area.Analog and Mixed-Signal designers use tools like Spice to fully characterize and model their designs.My only involvement with Mixed-Signal blocks has been to utilize behavioral models of things like PLLs, A/Ds, D/As within a larger SoC.There are some specific Verilog tricks to this which is what this FAQ is about (I do not wish to trivialize true Mixed-Signal methodology, but us chip-level folks need to know this trick).

A mixed-signal behavioral model might model the digital and analog input/output behavior of, for example, a D/A (Digital to Analog Converter).So, digital input in and analog voltage out.Things to model might be the timing (say, the D/A utilizes an internal Success Approximation algorithm), output range based on power supply voltages, voltage biases, etc.A behavioral model may not have any knowledge of the physical layout and therefore may not offer any fidelity whatsoever in terms of noise, interface, cross-talk, etc.A model **might** be parameterized given a specific characterization for a block.Be very careful about the assumptions and limitations of the model!

Issue #1; how do we model analog voltages in Verilog.Answer: use the Verilog real data type, declare “analog wires” as wire[63:0] in order to use a 64-bit floating-type represenation, and use the built-in PLI functions:

$rtoi converts reals to integers w/truncation e.g. 123.45 -> 123

$itor converts integers to reals e.g. 123 -> 123.0

$realtobits converts reals to 64-bit vector

$bitstoreal converts bit pattern to real

That was a lot.This is a trick to be used in vanilla Verilog.The 64-bit wire is simply a ways to actually interface to the ports of the mixed-signal block.In other words, our example D/A module may have an output called AOUT which is a voltage.Verilog does not allow us to declare an output port of type REAL.So, instead declare AOUT like this:

```
module dtoa (clk, reset..... aout.....);

....

wire [63:0]aout;// Analog output

....
```

We use 64 bits because we can use floating-point numbers to represent out voltage output (e.g. 1.22x10-3 for 1.22 millivolts).The floating-point value is relevant only to Verilog and your workstation and processor, and the IEEE floating-point format has NOTHING to do with the D/A implementation.Note the disconnect in terms of the netlist itself.The physical “netlist” that you might see in GDS may have a single metal interconnect that is AOUT, and obviously NOT

64 metal wires.Again, this is a trick.The 64-bit bus is only for wiring.You may have to do some quick netlist substitutions when you hand off a netlist.

In Verilog, the **real** data type is basically a floating-point number (e.g. like double in C).If you want to model an analog value either within the mixed-signal behavorial model, or externally in the system testbench (e.g. the sensor or actuator), use the real data type.You can convert back and forth between real and your wire [63:0] using the PLI functions listed above.A trivial D/A model could simply take the digital input value, convert it to real, scale it according to some #defines, and output the value on AOUT as the 64-bit "psuedo-analog" value.Your testbench can then do the reverse and print out the value, or whatever.More sophisticated models can model the Successive Approximation algorithm, employ look-ups, equations, etc. etc.

That's it.If you are getting a mixed-signal block from a vendor, then you may also receive (or you should ask for) the behavioral Verilog models for the IP.

How do I synthesize Verilog into gates with Synopsys?

The answer can, of course, occupy several lifetimes to completely answer.. BUT.. a straight-forward Verilog module can be very easily synthesized using Design Compiler (e.g. dc_shell). Most ASIC projects will create very elaborate synthesis scripts, CSH scripts, Makefiles, etc. This is all important in order automate the process and generalize the synthesis methodology for an ASIC project or an organization. BUT don't let this stop you from creating your own simple dc_shell experiments!

Let's say you create a Verilog module named foo.v that has a single clock input named 'clk'. You want to synthesize it so that you know it is synthesizable, know how big it is, how fast it is, etc. etc. Try this:

```
target_library = { CORELIB.db } <--- This part you need to get from your vendor...
read -format verilog foo.v
create_clock -name clk -period 37.0
set_clock_skew -uncertainty 0.4 clk
set_input_delay 1.0 -clock clk all_inputs() - clk - reset
set_output_delay 1.0 -clock clk all_outputs()
compile
report_area
```

report_timing

write -format db -hierarchy -output foo.db

write -format verilog -hierarchy -output foo.vg

quit

You can enter all this in interactively, or put it into a file called 'synth_foo.scr' and then enter:

dc_shell -f synth_foo.scr

You can spend your life learning more and more Synopsys and synthesis-related commands and techniques, but don't be afraid to begin using these simple commands.

How can I pass parameters to my simulation?

A testbench and simulation will likely need many different parameters and settings for different sorts of tests and conditions. It is definitely a good idea to concentrate on a single testbench file that is parameterized, rather than create a dozen seperate, yet nearly identical, testbenches. Here are 3 common techniques:

- Use a **define**. This is almost exactly the same approach as the #define and -D compiler arg that C programs use. In your Verilog code, use a `define to define the variable condition and then use the Verilog preprocessor directives like `ifdef. Use the '+define+' Verilog command line option. For example:

... to run the simulation ..

```
verilog testbench.v cpu.v +define+USEWCSDF
```

... in your code ...

```
`ifdef USEWCSDF
initial $sdf_annotate (testbench.cpu, "cpuwc.sdf");
`endif
```

The +define+ can also be filled in from your Makefile invocation, which in turn, can be finally filled in the your UNIX promp command line.

Defines are a blunt weapon because they are very global and you can only do so much with them since they are a pre-processor trick. Consider the next approach before resorting to defines.

- Use **parameters** and parameter definition modules. Parameters are not preprocessor definitions and they have *scope* (e.g. parameters are associated with specific modules). Parameters are therefore more clean, and if you are in the habit of using a lot of defines; consider switching to parameters. As an example, lets say we have a test (e.g. test12) which needs many parameters to have particular settings. In your code, you might have this sort of stuff:

  module testbench_uart1 (....)

  parameter BAUDRATE = 9600;

  ...

  if (BAUDRATE > 9600) begin

  ... E.g. use the parameter in your code like you might any general variable

  ... BAUDRATE is completely local to this module and this instance. You might

  ... have the same parameters in 3 other UART instances and they'd all be different

  ... values...

  Now, your test12 has all kinds of settings required for it. Let's define a special module called **testparams** which specifies all these settings. It will itself be a module instantiated under the testbench:

  module testparams;

  defparam testbench.cpu.uart1.BAUDRATE = 19200;

  defparam testbench.cpu.uart2.BAUDRATE = 9600;

  defparam testbench.cpu.uart3.BAUDRATE = 9600;

```
defparam testbench.clockrate CLOCKRATE = 200; // Period in ns.

... etc ...

endmodule
```

The above module always has the same module name, but you would have many different filenames; one for each test. So, the above would be kept in test12_params.v. Your Makefile includes the appropriate params file given the desired make target. (BTW: You may run across this sort of technique by ASIC vendors who might have a module containing parameters for a memory model, or you might see this used to collect together a large number of system calls that turn off timing or warnings on particular troublesome nets, etc. etc.)

- Use **memory blocks**. Not as common a technique, but something to consider. Since Verilog has a very convenient syntax for declaring and loading memories, you can store your input data in a hex file and use $readmemh to read all the data in at once.

In your testbench:

```
module testbench;

...

reg [31:0] control[0:1023];

...

initial $readmemh ("control.hex", control);

...

endmodule
```

You could vary the filename using the previous techniques. The control.hex file is just a file of hex values for the parameters. Luckily, $readmemh allows embedded comments, so you can keep the file very readable:

```
A000 // Starting address to put boot code in
10 // Activate all ten input pulse sources
... etc...
```

Obviously, you are limitied to actual hex values with this approach. Note, of course, that you are free to mix and match all of these techniques!

STAGE 4

Verilog gate level expected questions.

1) Tell something about why we do gate level simulations?

a. Since scan and other test structures are added during and after synthesis, they are not checked by the rtl simulations and therefore need to be verified by gate level simulation.
b. Static timing analysis tools do not check asynchronous interfaces, so gate level simulation is required to look at the timing of these interfaces.

c. Careless wildcards in the static timing constraints set false path or mutlicycle path constraints where they don't belong.

d. Design changes, typos, or misunderstanding of the design can lead to incorrect false paths or multicycle paths in the static timing constraints.

e. Using create_clock instead of create_generated_clock leads to incorrect static timing between clock domains.

f. Gate level simulation can be used to collect switching factor data for power estimation.

g. X's in RTL simulation can be optimistic or pessimistic. The best way to verify that the design does not have any unintended dependence on initial conditions is to run gate level simulation.

f. It's a nice "warm fuzzy" that the design has been implemented correctly.

2) Say if I perform Formal Verification say Logical Equivalence across Gatelevel netlists(Synthesis and post routed netlist). Do you still see a reason behind GLS.?

If we have verified the Synthesized netlist functionality is correct when compared to RTL and when we compare the Synthesized netlist versus Post route netlist logical Equivalence then I think we may not require GLS after P & R. But how do we ensure on Timing . To my knowledge Formal Verification Logical Equivalence Check does not perform Timing checks and dont ensure that the design will work on the operating frequency , so still I would go for GLS after post route database.

3)An AND gate and OR gate are given inputs X & 1 , what is expected output?

AND Gate output will be X

OR Gate output will be 1.

4) What is difference between NMOS & RNMOS?

RNMOS is resistive nmos that is in simulation strength will decrease by one unit , please refer to below Diagram.

Input Strength	Output Strength
supply	pull
strong	pull
pull	weak
weak	medium
large	medium
medium	small
small	small
highz	highz

4) Tell something about modeling delays in verilog?

Verilog can model delay types within its specification for gates and buffers. Parameters that can be modelled are T_rise, T_fall and T_turnoff. To add further detail, each of the three values can have minimum, typical and maximum values

T_rise, t_fall and t_off

Delay modelling syntax follows a specific discipline;
gate_type #(t_rise, t_fall, t_off) gate_name (paramters);
When specifiying the delays it is not necessary to have all of the delay values specified. However, certain rules are followed.

and #(3) gate1 (out1, in1, in2);
When only 1 delay is specified, the value is used to represent all of the delay types, i.e. in this example, t_rise = t_fall = t_off = 3.

or #(2,3) gate2 (out2, in3, in4);
When two delays are specified, the first value represents the rise time, the second value represents the fall time. Turn off time is presumed to be 0.

buf #(1,2,3) gate3 (out3, enable, in5);
When three delays are specified, the first value represents t_rise, the second value represents t_fall and the last value the turn off time.

Min, typ and max values

The general syntax for min, typ and max delay modelling is;
gate_type #(t_rise_min:t_ris_typ:t_rise_max, t_fall_min:t_fall_typ:t_fall_max, t_off_min:t_off_typ:t_off_max) gate_name (paramteters);

Similar rules apply for th especifying order as above. If only one t_rise value is specified then this value is applied to min, typ and max. If specifying more than one number, then all 3 MUST be scpecified. It is incorrect to specify two values as the compiler does not know which of the parameters the value represents.

An example of specifying two delays;
and #(1:2:3, 4:5:6) gate1 (out1, in1, in2);
This shows all values necessary for rise and fall times and gives values for min, typ and max for both delay types.

Another acceptable alternative would be;
or #(6:3:9, 5) gate2 (out2, in3, in4);
Here, 5 represents min, typ and max for the fall time.

N.B. T_off is only applicable to tri-state logic devices, it does not apply to primitive logic gates because they cannot be turned off.

5) With a specify block how to defining pin-to-pin delays for the module ?

```
module A( q, a, b, c, d )
        input a, b, c, d;
        output q;
        wire e, f;
        // specify block containing delay statements
        specify
                ( a => q ) = 6;   // delay from a to q
                ( b => q ) = 7;   // delay from b to q
                ( c => q ) = 7;   // delay form c to q
                ( d => q ) = 6;   // delay from d to q
        endspecify
        // module definition
        or o1( e, a, b );
        or o2( f, c, d );
        exor ex1( q, e, f );
endmodule
```

```
module A( q, a, b, c, d )
        input a, b, c, d;
        output q;
```

```
        wire e, f;
        // specify block containing full connection statements
        specify
                ( a, d *> q ) = 6;     // delay from a and d to q
                ( b, c *> q ) = 7;     // delay from b and c to q
        endspecify
        // module definition
        or o1( e, a, b );
        or o2( f, c, d );
        exor ex1( q, e, f );
endmodule
```

6) What are conditional path delays?

Conditional path delays, sometimes called *state dependent path delays*, are used to model delays which are dependent on the values of the signals in the circuit. This type of delay is expressed with an **if** conditional statement. The operands can be scalar or vector module input or inout ports, locally defined registers or nets, compile time constants (constant numbers or specify block parameters), or any bit-select or part-select of these. The conditional statement can contain any bitwise, logical, concatenation, conditional, or reduction operator. The **else** construct cannot be used.

```
//Conditional path delays
Module A( q, a, b, c, d );
        output q;
        input a, b, c, d;
        wire e, f;
        // specify block with conditional timing statements
        specify
                // different timing set by level of input a
                if (a) ( a => q ) = 12;
                if ~(a) ( a => q ) = 14;
                // delay conditional on b and c
                // if b & c is true then delay is 7 else delay is 9
                if ( b & c ) ( b => q ) = 7;
                if ( ~( b & c )) ( b => q ) = 9;
                // using the concatenation operator and full connections
                if ( {c, d} = 2'b10 ) ( c, d *> q ) = 15;
                if ( {c, d} != 2'b10 ) ( c, d *> q ) = 12;
        endspecify
        or o1( e, a, b );
        or o2( f, c, d );
        exor ex1( q, e, f );
endmodule
```

6) Tell something about Rise, fall, and turn-off delays?

Timing delays between pins can be expressed in greater detail by specifying rise, fall, and turn-off delay values. One, two, three, six, or twelve delay values can be specified for any path. The order in which the delay values are specified must be strictly followed.

```
// One delay used for all transitions
specparam delay = 15;
( a => q ) = delay;
// Two delays gives rise and fall times
specparam rise = 10, fall = 11;
( a => q ) = ( rise, fall );
// Three delays gives rise, fall and turn-off
// rise is used for 0-1, and z-1, fall for 1-0, and z-0, and turn-off for 0-z, and 1-z.
specparam rise = 10, fall = 11, toff = 8;
( a => q ) = ( rise, fall, toff );
// Six delays specifies transitions 0-1, 1-0, 0-z, z-1, 1-z, z-0
// strictly in that order
specparam t01 = 8, t10 = 9, t0z = 10, tz1 = 11, t1z = 12, tz0 = 13;
( a => q ) = ( t01, t10, t0z, tz1, t1z, tz0 );
// Twelve delays specifies transitions:
// 0-1, 1-0, 0-z, z-1, 1-z, z-0, 0-x, x-1, 1-x, x-0, x-z, z-x
// again strictly in that order
specparam t01 = 8, t10 = 9, t0z = 10, tz1 = 11, t1z = 12, tz0 = 13;
specparam t0x = 11, tx1 = 14, t1x = 12, tx0 = 10, txz = 8, tzx = 9;
( a => q ) = ( t01, t10, t0z, tz1, t1z, tz0, t0x, tx1, t1x, tx0, txz, tzx );
```

7)Tell me about In verilog delay modeling?

Distributed Delay

Distributed delay is delay assigned to each gate in a module. An example circuit is shown below.

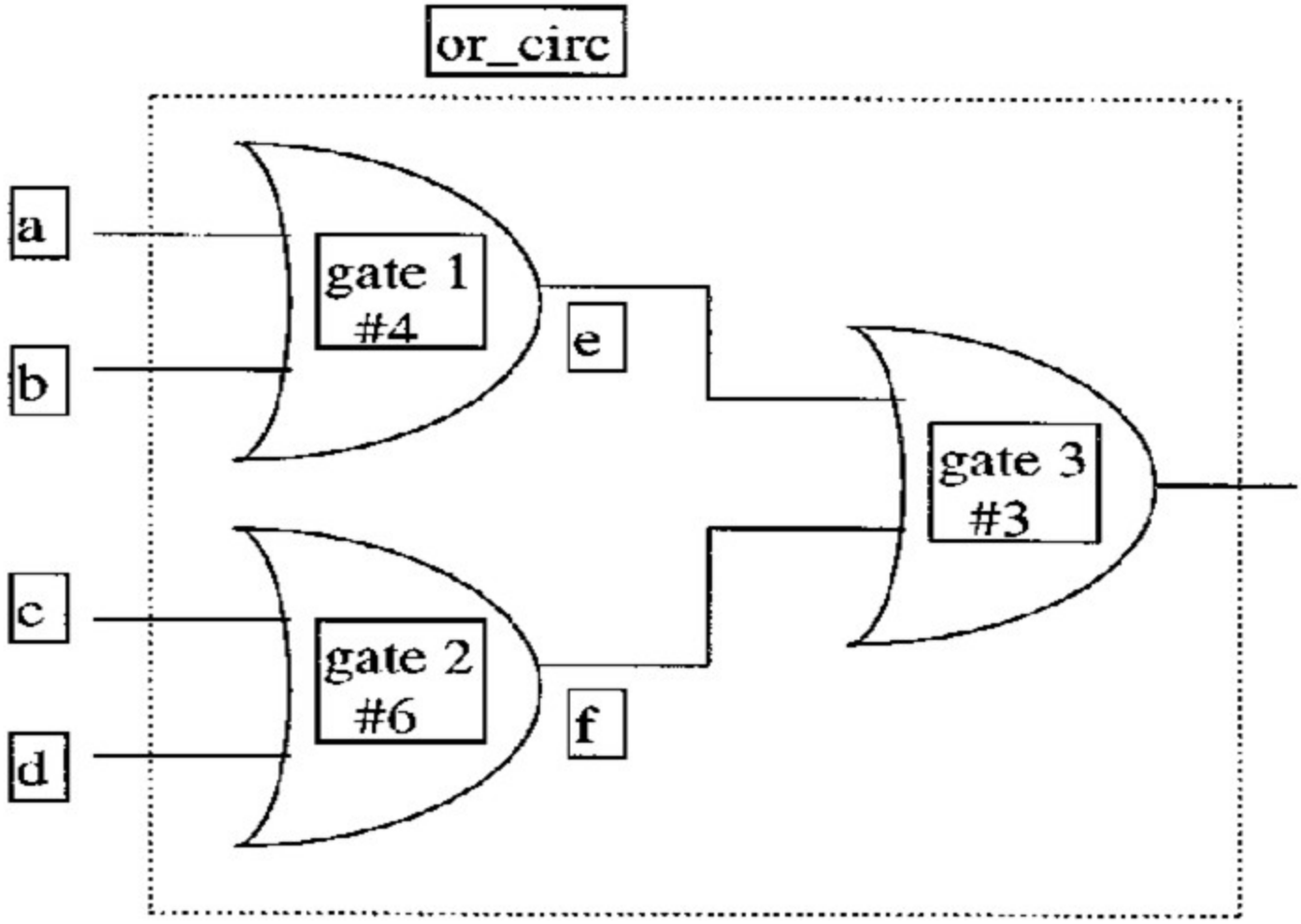


Figure 1: Distributed delay

As can be seen from Figure 1, each of the or-gates in the circuit above has a delay assigned to it:

- gate 1 has a delay of 4
- gate 2 has a delay of 6
- gate 3 has a delay of 3

When the input of any gate change, the output of the gate changes after the delay value specified.

The gate function and delay, for example for gate 1, can be described in the following manner:

or #4 a1 (e, a, b);

A delay of 4 is assigned to the or-gate. This means that the output of the gate, e, is delayed by 4 from the inputs a and b.

The module explaining Figure 1 can be of two forms:

```
1)
Module or_circ (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
        //Delay distributed to each gate
        or #4 a1 (e, a, b);
        or #6 a2 (f, c, d);
        or #3 a3 (out, e, f);
endmodule
2)
Module or_circ (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
        //Delay distributed to each expression
        assign #4 e = a & b;
        assign #6 e = c & d;
        assign #3 e = e & f;
endmodule
```

Version 1 models the circuit by assigning delay values to individual gates, while version 2 use delay values in individual assign statements. (An assign statement allows us to describe a combinational logic function without regard to its actual structural implementation. This means that the assign statement does not contain any modules with port connections.)

The above or_circ modules results in delays of (4+3) = 7 and (6+3) = 9 for the 4 connections part from the input to the output of the circuit.

Lumped Delay

Lumped delay is delay assigned as a single delay in each module, mostly to the output gate of the module. The cumulative delay of all paths is lumped at one location. The figure below is an example of lumped delay. This figure is similar as the figure of the

distributed delay, but with the sum delay of the longest path assigned to the output gate: (delay of gate 2 + delay of gate 3) = 9.

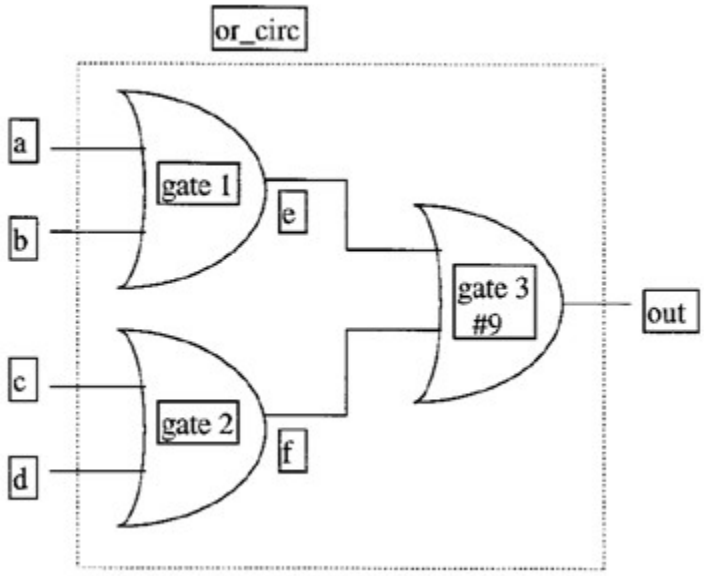


Figure 2: Lumped delay

As can be seen from Figure 2, gate 3 has got a delay of 9. When the input of this gate changes, the output of the gate changes after the delay value specified.

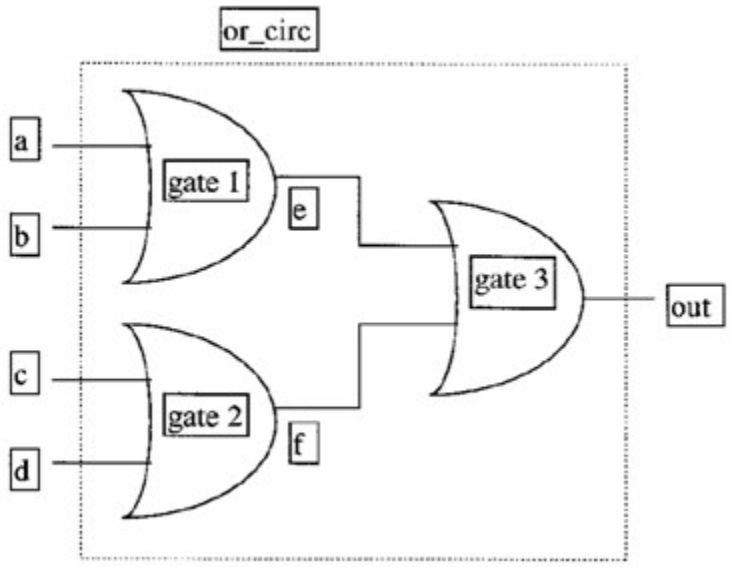
The program corresponding to Figure 2, is very similar to the one for distributed delay. The difference is that only or - gate 3 has got a delay assigned to it:

```
1)
Module or_circ (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
        or a1 (e, a, b);
        or a2 (f, c, d);
        or #9 a3 (out, e, f); //delay only on the output gate
endmodule
```

This model can be used if delay between different inputs is not required.

Pin - to Pin Delay

Pin - to - Pin delay, also called path delay, is delay assigned to paths from each input to each output. An example circuit is shown below.



path a - e - out, delay = 7
path b - e - out, delay = 7
path c - f - out, delay = 9
path d - f - out, delay = 9

Figure 3: Pin - to Pin delay

The total delay from each input to each output is given. The same example circuit as for the distributed and lumped delay model is used. This means that the sum delay from each input to each output is the same.

The module for the above circuit is shown beneath:

```
Module or_circ (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
//Blocks specified with path delay
specify
        (a => out) = 7;
        (b => out) = 7;
        (c => out) = 9;
        (d => out) = 9;
endspecify
//gate calculations
```

or a1(e, a, b);
or a2(f, c, d);
or a3(out, e, f);
endmodule

Path delays of a module are specified incide a specify block, as seen from the example above. An example of delay from the input, a, to the output, out, is written as (a => out) = delay, where delay sets the delay between the two ports. The gate calculations are done after the path delays are defined.

For larger circuits, the pin - to - pin delay can be easier to model than distributed delay. This is because the designer writing delay models, needs to know only the input / output pins of the module, rather than the internals of the module. The path delays for digital circuits can be found through different simulation programs, for instance SPICE. Pin - to - Pin delays for standard parts can be found from data books. By using the path delay model, the program speed will increase.

8) Tell something about delay modeling timing checks?

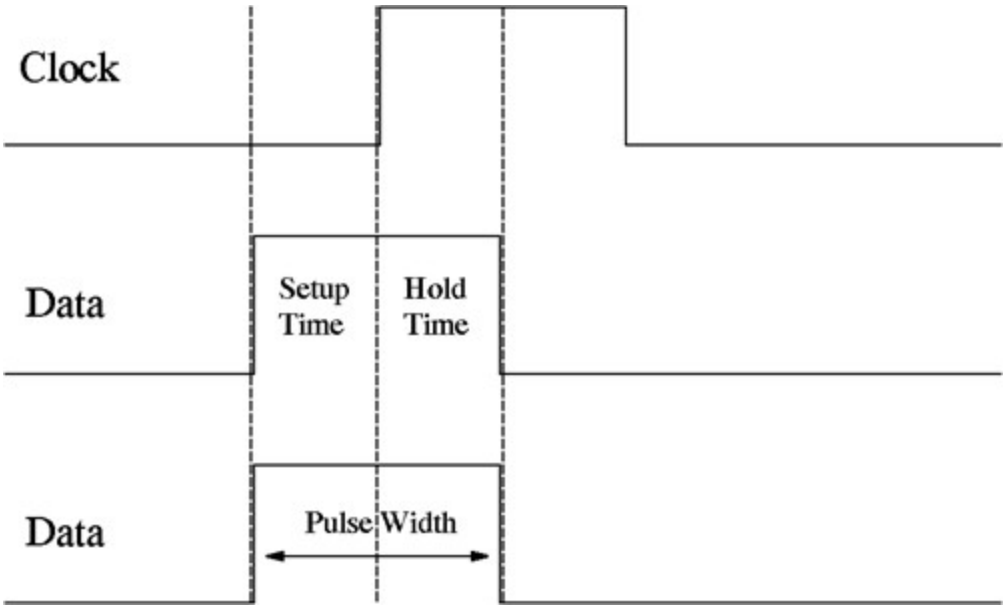
Delay Modeling: Timing Checks.

Keywords: $setup, $hold, $width

This section, the final part of the delay modeling chapter, discusses some of the various system tasks that exist for the purposes of timing checks. Verilog contains many timing-check system tasks, but only the three most common tasks are discussed here: **$setup**, **$hold** and **$width**. Timing checks are used to verify that timing constraints are upheld, and are especially important in the simulation of high-speed sequential circuits such as microprocessors. All timing checks must be contained within **specify** blocks as shown in the example below.

The **$setup** and **$hold** tasks are used to monitor the *setup* and *hold* constraints during the simulation of a sequential circuit element. In the example, the setup time is the minimum allowed time between a change in the input *d* and a positive clock edge. Similarly, the hold time is the minimum allowed time between a positive clock edge and a change in the input *d*.

The **$width** task is used to check the minimum width of a positive or negative-going pulse. In the example, this is the time between a negative transition and the transition back to 1.



Syntax:

NB: *data_change*, *reference* and *reference1* must be declared *wires*.

$setup(*data_change*, *reference*, *time_limit*);

data_change: signal that is checked against the *reference*

reference: signal used as reference

time_limit: minimum time required between the two events.

Violation if: Treference - Tdata_change < time_limit.

$hold(*reference*, *data_change*, *time_limit*);

reference: signal used as reference

data_change: signal that is checked against the *reference*

time_limit: minimum time required between the two events.

Violation if: Tdata_change - Treference < time_limit

$width(*reference1*, *time_limit*);

reference1: first transition of signal

time_limit: minimum time required between *transition1* and *transition2*.

Violation if: Treference2 - Treference1 < time_limit

Example:

```
module d_type(q, clk, d);
  output q;
  input  clk, d;

  reg    q;

  always @(posedge clk)
    q = d;

endmodule // d_type

module stimulus;

  reg  clk, d;
  wire q, clk2, d2;

  d_type dt_test(q, clk, d);

  assign d2=d;
  assign clk2=clk;

  initial
    begin
      $display ("\t\t    clock d q");
      $display ($time,"   %b   %b %b", clk, d, q);
      clk=0;
      d=1;
      #7 d=0;
      #7 d=1; // causes setup violation
      #3 d=0;
      #5 d=1; // causes hold violation
      #2 d=0;
      #1 d=1; // causes width violation
    end // initial begin

  initial
    #26 $finish;
```

```
__
  always
    #3 clk = ~clk;

__
  always
    #1 $display ($time,"   %b   %b %b", clk, d, q);

__
  specify
    $setup(d2, posedge clk2, 2);
    $hold(posedge clk2, d2, 2);
    $width(negedge d2, 2);
  endspecify
endmodule // stimulus
```

Output:

```
                clock d q
              0   x   x x
              1   0   1 x
              2   0   1 x
              3   1   1 x
              4   1   1 1
              5   1   1 1
              6   0   1 1
              7   0   0 1
              8   0   0 1
              9   1   0 1
             10   1   0 0
             11   1   0 0
             12   0   0 0
             13   0   0 0
             14   0   1 0
             15   1   1 0

"timechecks.v", 46: Timing violation in stimulus
    $setup( d2:14, posedge clk2:15, 2 );

             16   1   1 1
             17   1   0 1
             18   0   0 1
             19   0   0 1
             20   0   0 1
             21   1   0 1
             22   1   1 0
```

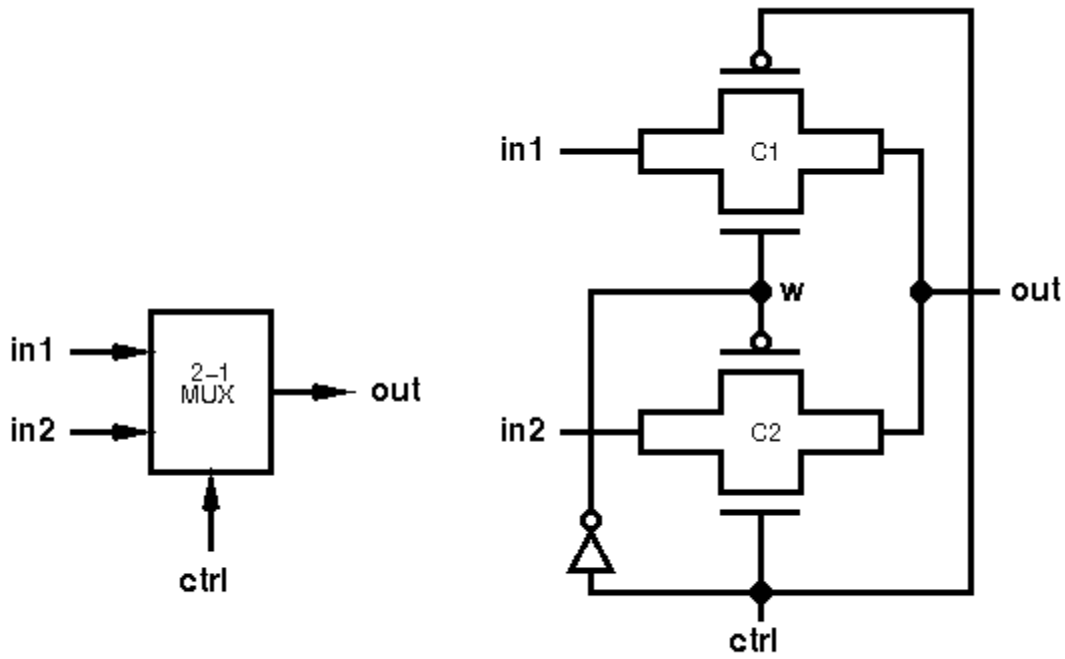
"timechecks.v", 47: Timing violation in stimulus
    $hold(posedge clk2:21, d2:22, 2);

            23   1   1 0
            24   0   0 0
            25   0   1 0

"timechecks.v", 48: Timing violation in stimulus
    $width(negedge d2:24, : 25, 2);

9) Draw a 2:1 mux using switches and verilog code for it?

1-bit 2-1 Multiplexer



Gate representation Switch–level model

This circuit assigns the output **out** to either inputs **in1** or **in2** depending on the low or high values of **ctrl** respectively.

```
// Switch-level description of a 1-bit 2-1 multiplexer
// ctrl=0, out=in1; ctrl=1, out=in2

module mux21_sw (out, ctrl, in1, in2);

  output out;              // mux output
```

```
  input  ctrl, in1, in2;      // mux inputs
  wire   w;                   // internal wire

  inv_sw I1 (w, ctrl);        // instantiate inverter module

  cmos C1 (out, in1, w, ctrl);  // instantiate cmos switches
  cmos C2 (out, in2, ctrl, w);

endmodule
```

An inverter is required in the multiplexer circuit, which is instantiated from the previously defined module.

Two transmission gates, of instance names C1 and C2, are implemented with the *cmos* statement, in the format cmos [instancename]([output],[input],[nmosgate],[pmosgate]). Again, the instance name is optional.

10)What are the synthesizable gate level constructs?

The <GATETYPE> Keywords				
and	buf	nmos	tran	pullup
nand	not	pmos	tranif0	pulldown
nor	bufif0	cmos	tranif1	
or	bufif1	rnmos	rtran	
xor	notif0	rpmos	rtranif0	
xnor	notif1	rcmos	rtranif1	

The above table gives all the gate level constructs of only the constructs in first two columns are synthesizable.