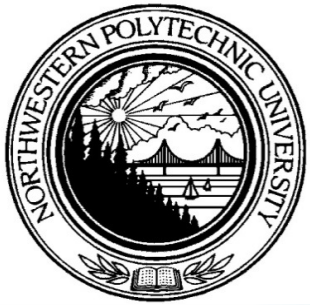



EE461 Verilog-HDL

Week 6 Behavioral Level Design



Alex Yang, Engineering School, NPU

Week 6 Outlines

- Behavioral Modeling (Non-synthesizable)
  - For design ideas verification
  - For testbench verification

Week 6 Outlines

- Behavioral Modeling (Non-synthesizable)
  - Procedural constructs
    - "initial" block
    - "always" block
  - Blocks
    - Sequential blocks
    - Parallel blocks
    - Special blocks
      - Named blocks & "disable" statement
  - Procedural continuous assignments
    - assign and deassign assignments
    - force and release assignments

Week 6 Outlines

- Behavioral Modeling (Non-synthesizable)
  - Non-synthesizable loop statements
    - forever loop
    - repeat loop
  - Timing control
    - Delay
    - Circuit delay models
    - Specify blocks
    - Timing checks
    - Event timing control

Week 6 Outlines

- Procedural constructs
  - "initial" block (Non-synthesizable)
    - Only execute each statement at one time during simulaiton if the statement isn't within sub-blocks.
    - LHS variable data type must be reg, integer, real or time for assignment.
    - Start execution at time 0.
    - Two more "initial" blocks in one module are executed concurrently.
    - "initial" block can't contain any "always" blocks.
    - "initial" block can't have nested "initial" blocks.
    - Any verilog statement must be either in "assign"/"always" blocks or "initial" block.

Procedural constructs

- "initial" block (Non-synthesizable)

```
module testInit1;
   reg a;
   initial begin
     assign a=1'b0;
     $display("a=%b",a);
      #10 $finish;
    end
endmodule
```

Works? Ans: Yes.
However, it is bad code.

How about changing "a" to wire? Ans: Error

```
module testInit2;
        reg a,b,c;
        wire clk;
        initial begin //Can't include "always"
                /*always@(a or b)begin
                        c=a+b;
                end*/
                /*always@(posedge clk)begin
                end*/
                always #1 clk = ~clk;
                $display("Try to print a=%b",a);
                #10 $finish;
        end
endmodule
```

Works? Ans: Error

Procedural constructs

- "initial" block(Non-synthesizable)

```
module testInit3;
    reg a,b,c;
    integer i;
    for(i=0;i<10;i=i+1)begin
            c = 1'b0;
      end
    initial begin
          $display( a=%b",a);
            #10 $finish;
    end
endmodule
```

Works? Ans: No.
for-loop must be in initial or always blocks.

```
module testInit4();
  reg [5:0] loc;
  reg [7:0] data;
  while (data[0] == 0) begin
    loc = loc + 1;
    data = data >> 1;
   end
  always @ (data or loc) begin
            loc = 0;
            if (data == 0) loc = 32;
            else data = data >> 1;
     $display ("DATA = %b   LOCATION = %d",data,loc);
   end
endmodule
```

Works? Ans: Error

Procedural constructs

- "always" block (For Non-synthesizable)
  - Starts at simulation time 0
  - Executes again and again during simulation.
  - LHS variable data type must be reg, integer, real or time for assignment.
  - "always" blocks, "initial" blocks and "assign" blocks in one module are executed concurrently.
  - "always" block can't contain any "initial" blocks.
  - "always" block can't have nested "always" blocks.
  - "always" block could include "assign", but not good.
  - Some statements must be in "always" blocks or "initial" block.
  - In behavioral level, blocking assignment(=) could be used in "always" sequential block.

Procedural constructs

- "always" block (For Non-synthesizable)

```
//always@(a or b)begin
always@(posedge clk)begin
// if-else, case, casex, casez
//for, while, repeat
//wait ...
// task, function calling
end
```

```
module clkGen(clk);
  output   clk;
  reg      clk;
  initial  clk = 1'b0;
  always #10 clk = ~clk;
endmodule
```

```
module testInit1;
    reg a,b,c,d,e;
    always@(a or b) begin
        //initial begin          //Doesn't work
            $display("Try to print a=%b",a);
            #10 $finish;
        //end
        always@(c or d)begin //Doesn't work
            e=a+b+c+d;
        end
    end
endmodule
```

Note: please don't do that. You will get errors

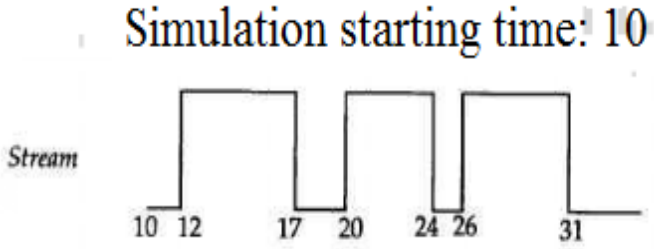
Blocks

- Sequential blocks (For Non-synthesizable)
  - begin-end block
  - In "initial" block, each statement in "begin-end" section is executed sequentially except non-blocking assignment and timing control statements.
  - Each statement delay timing in this block is relative value compared to that in the last statement.
  - More than one statement in verilog must be within "begin-end" or "fork-join" section.

Blocks

- Sequential blocks (For Non-synthesizable)
  - begin-end block

```
module stream();
  reg stream;
  initial  begin
           #10;
           #2 stream = 1;
           #5 stream = 0;
           #3 stream = 1;
           #4 stream = 0;
           #2 stream = 1;
           #5 stream = 0;
  end
endmodule
```




```
initial  begin
    // 1st clk posedge
    @(posedege clk) q = 0;
   // 2nd clk posedge
    @(posedege clk) q = 1;
end
```

Blocks

- Sequential blocks (For Non-synthesizable)
  - begin-end block

```
begin
        @trig r = 1;
        #250 r = 0;
 end
```

```
 begin
    @c     r = 4'h35;
    @c     r = 4'hE2;
    @c     r = 4'h00;
    @c     r = 4'hF7;
    @c     -> trig;
end
```

Note: 1. above two begin-end blocks are either in initial block or always block.

2. @trig or @c: once trig or c changes assignment will be executed.

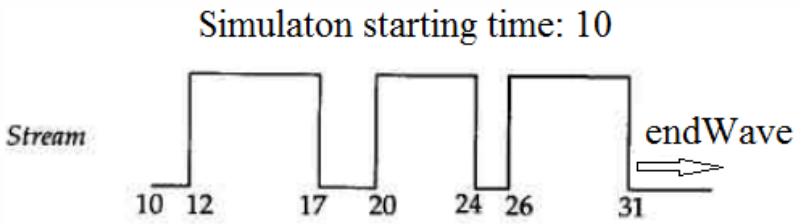
Blocks

- Parallel blocks (Non-synthesizable)
  - fork-join block, like fork-join all, completes all threads and then excute parent thread
  - Statements execute simultaneously.
  - Sequence of statements in fork-join block doesn't matter.
  - Delay values in each statement or timing control referrence point are from the same simulation starting time
  - fork-join must be either in initial block or always block.
  - fork-join could include "assign" block, but it is not good.
  - However, fork-join can't include any initial and always blocks.

Blocks

- Parallel blocks (Non-synthesizable)
  - fork-join block: create more threads

```
module stream();
  reg stream;
  initial  begin
    #10;
    fork
            #2 stream = 1;
            #7 stream = 0;
            #10 stream = 1;
            #14 stream = 0;
            #16 stream = 1;
            #21 stream = 0;
            #21 -> endWave;
    join
  end
endmodule
```



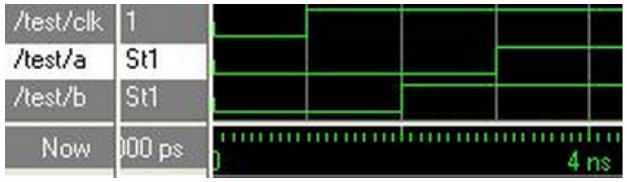

```
initial  begin
            fork
                        @Aevent;
                        @Bevent;
            join
            areg = breg;
end
```

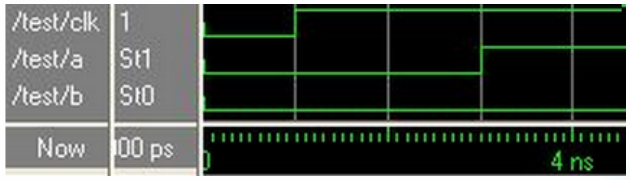
Note: Once any of Aevent/Bevent or both change, it/they start to execute.

Blocks

- Parallel blocks (Non-synthesizable)
  - fork-join block

```
module forkjoin(clk, a, b);
    input clk;
    output a;
    output b;
    reg a, b;
    initial begin
        a = 0;
        b = 0;
    end
    always @(posedge clk)
    fork
        #2 a = 1;
        #1 b = 1;
    join
endmodule
```





Which is the right and left module outputs respectively?

```
module forkjoin1(clk, a, b);
    input clk;
    output a;
    output b;
    reg a, b;
    initial begin
        a = 0;
        b = 0;
    end
    always @(posedge clk)
    fork
        #2 a = 1;
        #1 b = a;
    join
endmodule
```

Blocks

- Parallel blocks (Non-synthesizable)
  - fork-join block

```
module test(clk, a, b);
    input clk;
    output a;
    output b;
    reg a, b;
    initial begin
       a = 0;
       b = 0;
    end
    always @(posedge clk) begin
        #2 a = 1;
        #1 b = a;
    end
endmodule
```

Q: output wavefoms?

```
module nesting1(clk, a, b,
c, d, e, f);
    input clk;
    output a, b, c, d, e, f;

    reg a, b, c, d, e, f;

    initial begin
       a = 0;
       b = 0;
       c = 0;
       d = 0;
       e = 0;
       f = 0;
    end
```

```
    always @(posedge clk)
    fork
        #2 a = 1;
        #2 b = 1;
        begin //Sequential
            #2 c = 1;
            #2 d = 1;
            #2 e = 1;
        end
        #2 f = 1;
    join
endmodule
```

Q: output wavefoms?

Blocks

- Special blocks (Non-synthesizable)
  - Named block & "disable" block
    - Declare local variables in named blocks
    - No "break" & "continue" in Verilog
    - Named block could help to jump out from for-loops and continue to execute next iteration in loops.

```
module namedBlk();
    reg  x, y, z;
    initial begin
            x = 1'b1;
            #12 y = 1'b0;
            #10 z = 1'b0;
   end
```

```
initial begin: test
    reg x, y, z; // local variables
          x = 1'b0;
          #12 y = 1'b1;
          #10 z = 1'b1;
         $display("Local: x=%b,y=%b,z=%b",x,y,z);
end
 initial $monitor("x=%b,y=%b,z=%b",x,y,z);
endmodule
Results: x=1,y=0,z=0
```

Blocks

- Special blocks (Non-synthesizable)
  - Named block & "disable" block

```
module testDisable();
    reg[3:0]  a;
    integer          i;
    initial begin: test
       a = 4'b0000;
       for(i=0;i<4;i=i+1) begin
          a=a+1;
          if(2==i) disable test;
         a = a+1;
       end
    end
endmodule
```

Q: a = ? Ans: 5

Note: it is like break.

```
module testDisable1();
    reg[3:0]  a;
    integer          i;
    initial begin: test
       a = 4'b0000;
       for(i=0;i<4;i=i+1) begin: testThis
          a=a+1;
          if(2==i) disable testThis;
          a = a+1;
       end
    end
endmodule
```

Q: a = ? Ans: 7

Note: it is like continue

Blocks

- Special blocks (Non-synthesizable)
  - Named block & "disable" block - nested loop

```
initial begin:one
    // a = 0;
    a = 1;
    for(i=1;i<5;i=i+1) begin:two
      //Stop here and jump out from initial block;
      if(a==0) disable one;
     //Stop here and execute next iteration in loop;
      if(a==1) disable two;
   end
   #10 a=10;
end
```

Note: 1. what is a's value? Ans: 0

2.disable blockName = stop + go to next in blockName

```
initial begin : A
  for( ... ... )begin : B
       ...
       for (... ...) begin: C
                ...
                if(...)disable C;
                if(...)disable B;
                ...
       end
       if(...) disable A;
 end
... ...
end
```

Procedural continuous assignments

- assign and deassign (Non-synthesizable)
  - Non-synthesizable in initial & always block.
  - Data type of variables in LHS must be reg, not wire like in continuous assignments. Of course, these variables can't be in continuous assign block in the same block.
  - Override values of the regular procedural assignments in initial & always blocks.
  - It is to inject values to reg variables for function simulation or verification.
  - Can't bound value to individual bit in assign-deassign.
  - deassign: return to privous value before "assign"

Procedural continuous assignments

- assign and deassign (Non-synthesizable)

```
module DFF(input clk, reset, d,
                output reg q, qbar
);
  always @(negedge clk) begin
    q <= d;
    qbar <= ~d;
  end
  always @(reset)
    if (reset) begin
      assign q = 1'b0;
      assign qbar = 1'b1;
   end else begin
     deassign q;
    deassign qbar;
   end
endmodule
```

Note: behavioral level for reset

```
module top();
... ...
submodule u ( .sig1 (sig1_w)... ...);
... ...
initial begin
  #5 top.u.sig1 = 4'b0000; //not sig1_w
end
initial begin
  #10 assign top.u.sig1 = 4'b0101;
  #10 deassign top.u.sig1;
end
endmodule
```

Note:

1. Inject a value to internal reg. And sig1 must be a reg type.
2. 4'b0101 overrides 4'b0000.

Procedural continuous assignments

- assign and deassign (Non-synthesizable)

```
module assignDeassign ();
  reg clk,rst,d,preset;
  wire q;
  initial begin
        $monitor("@%g clk %b rst   %b preset %b
d %b q %b", $time, clk, rst, preset, d, q);
      clk = 0; rst = 0; d  = 0; preset = 0;
     #10 rst = 1;
    #10 rst = 0;
    repeat (10) begin
       @ (posedge clk);
            d <= $random;
       @ (negedge clk) ;
          preset <= ~preset;
   end
   #1 $finish;
  end
always #1 clk = ~clk;// Clock generator
```

```
always @(preset)
  if (preset) begin
        assign U.Q = 1;  // Q must be reg in submodule
  end else begin
        deassign U.Q;
 end
 d_ff U (.clock (clk), .reset (rst), .D (d), .Q (q));
endmodule
-----------------------------------------------------
module d_ff (clock,reset,D,Q);
   input clock,reset,D; output Q;
   reg Q;
   always @ (posedge clock)
              if (reset) begin
                          Q <= 0;
              end else begin
                          Q <= D;
   end
endmodule
```

Procedural continuous assignments

- force and release (Non-synthesizable)
  - Non-synthesizable in initial & assign block.
  - Data type of variables in LHS can be either reg or wire.
  - force has higher precedence than assign-deassign, that is, after "force", "assign" can't change the value that is forced.
  - Can bound value to individual bit in force-release.
  - release: return to privous value before "force"
  - force more times, and then release. The value assigned by all the "force" is gone, and return to the privous value before "force".

Procedural continuous assignments

- force and release (Non-synthesizable)

```
module testForce;
        wire out, a, b;
        or #1 (out, a, b);
        assign a = 1'b0;
        assign b = 1'b1;
        initial begin
                #5 force out = a & b;
                $display("Time: %0g,out=%b",$time,out);
                #5 release out;
                $display("Time: %0g,out=%b",$time,out);
                #5 $finish;
        end
endmodule
```

Q: what are results?

Note: LHS variabls in force-release could be wire or reg

```
module testForce;
        wire out, a, b;
        reg out1;
        or #1 (out, a, b);
        assign a = 1'b0;
        assign b = 1'b1;
        always@(out)begin
                out1=out;
        end
        initial begin
                #5 assign out1=1'bx;
                #5 force out1 = 1'b0;
                // #5 assign out1=1'bx; // What is out1?
                $display("Time:%0g,out=%b,out1=%b",$time,out,out1);
                #5 force out1 = 1'b1;
                $display("Time:%0g,out=%b,out1=%b",$time,out,out1);
                #5 release out1;
                $display("Time:%0g,out=%b,out1=%b",$time,out,out1);
                #5 $finish;
        end
endmodule
```

```
Time:10,out=1,out1=0
Time:15,out=1,out1=1
Time:20,out=1,out1=x
```

Procedural continuous assignments

- force and release (Non-synthesizable)

```
module force_release ();
  reg clk,rst,d,preset;
 wire q;
 initial begin
      $monitor("@%g clk %b rst %b preset %b d %b
               q %b", $time, clk, rst, preset, d, q);
     clk = 0; rst = 0; d  = 0; preset = 0;
    #10 rst = 1;
    #10 rst = 0;
  repeat (10) begin
    @ (posedge clk);
              d <= $random;
    @ (negedge clk) ;
              preset <= ~preset;
  end
 #1 $finish;
end
always #1 clk = ~clk;
```

```
always @(preset)
  if (preset) begin
     force U.Q = preset; // Q could be wire in submodule
  end else begin
     release U.Q;
   end
 d_ff U (.clock (clk), .reset (rst), .d (D), .Q (q));
endmodule
-----------------------------------------------------
module d_ff (clock,reset,D,Q);
    input clock,reset,D; output Q;
    wire Q;
    reg Q_r;
    assign Q = Q_r;
    always @ (posedge clock)
                if (reset) begin
                   Q_r <= 0;
                end else begin
                   Q_r <= D;
end
endmodule
```

loop statements

- forever & repeat (Non-synthesizable)
  - forever
    - Infinite loop
    - Normally in initial block
    - In the same initial block, the statements after forever loop can't be executed.

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

Note: what is clock waveform?

```
reg clock, x, y;
initial begin
    forever @(posedge clock) x <= y;
end
```

Note: Simulate sequential logic in behavioral level.

loop statements

- forever & repeat (Non-synthesizable)

```
module testForever;
   reg clk0;
   reg [7:0] num;
   initial begin
         clk0 = 0;
         num = 0;
         forever begin
               #20 clk0=1;
               #10 clk0=0;
         end
        repeat(8'b11100111) begin
               num = num+1;
        end
    end
endmodule
```

Note: Statements in repeat block can never be executed.

```
module testForever1;
     reg clk0;
     reg [7:0] cnt, num;
     initial begin
          clk0 = 0;
          num = 0;
          begin: A
                    forever begin
                         cnt = cnt+1;
                         #20 clk0=1;
                         #10 clk0=0;
                         if(cnt>='d100) begin
                              disable A;
                         end
                    end
          end
          repeat (8'b11100111) begin
                    num = num+1;
           end
     end
endmodule
```

Note: Jump out from forever loop by disable statement.

loop statements

- forever & repeat (Non-synthesizable)
  - repeat
    - repeat (# of loop iteration) begin ... ... end
    - # of loop iteration: constant number or variable
    - if # of loop iteration is x or z, what will happen?

```
module testRepeat;
  reg [7:0] num;
  initial begin
            num = 0;
            repeat(8'bx)begin
               num = num+1;
            end
  end
endmodule
```

Note: any bit with x/z in repeat number, # of iteration = 0

```
module testRepeat1;
  reg [7:0] num;
  initial begin
            num = 0;
            repeat(8'b0110111x)begin
               num = num+1;
            end
  end
endmodule
```

Note: num=? Ans: num=0;

loop statements

- forever & repeat (Non-synthesizable)
  - repeat

```
module testRepeat;
  reg [7:0] num;
  initial begin
            num = 1;
            repeat(num)begin
               num = num+1;
            end
  end
endmodule
```

Q: what is num's value? Ans: 2

Note: if number in repeat loop is variable, total number iteration of loop is fixed to initial value and never change with this variable.

Timing control

- Delays (Non-synthesizable)
  - Delays in Combinational & Sequential Logic
    - Delays in Combinational Logic
      - Rise time delays
      - Fall time delays
      - Turn-on & Turn-off delays
    - Delays in Sequential Logic
      - Propagation delays (Tclk-q)
      - Setup time (Ts)
      - Hold time (Th)
      - Determining Clk Max Frequency
      - Holding time validation
      - Timing Analysis Examples

Timing control

- Delays (Non-synthesizable)
  - Dataflow Delays
  - Behavioral Level Delays
  - Distributed Delays
- Circuit Delay Models
  - Distributed Delay
  - Lumped Delay
  - Pin-to-pin Delay

Timing control

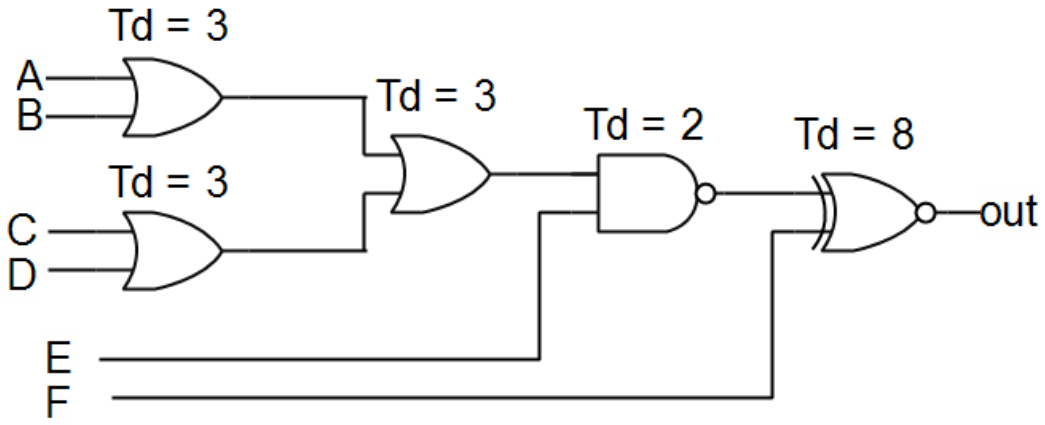
- Delays (Non-synthesizable)
  - Delays in Combinational Logic
    - Rise time delays (0,1,x,z -> 1)
    - Fall time delays (1, x, z -> 0)
    - Turn-on & Turn-off delays (0, 1, x -> z)

    e.g. Gates: and #(5, 10) u0 (f, a,b);
    Buffer: bufif0 #(5, 10, 15) u0 (f, a, ctrl);
    Switch: pmos #(5, 10, 15) u0 (d, s, g);
    Bidirection: tranif0 #(5, 10) u0 (f, a, ctrl);
    Pullup/down: pullup (neta),(netb); //No delay in pullup & pulldown

    what are the delay vaules, like Tr/Tf/Ton in each examples?

Timing control

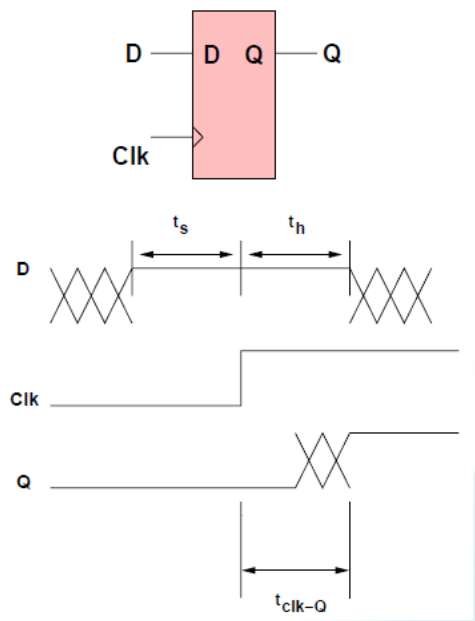
- Delays
  - Delays in Combinational Logic



what is the longest delay time from A,B,C,D,E,F to out?

Timing control

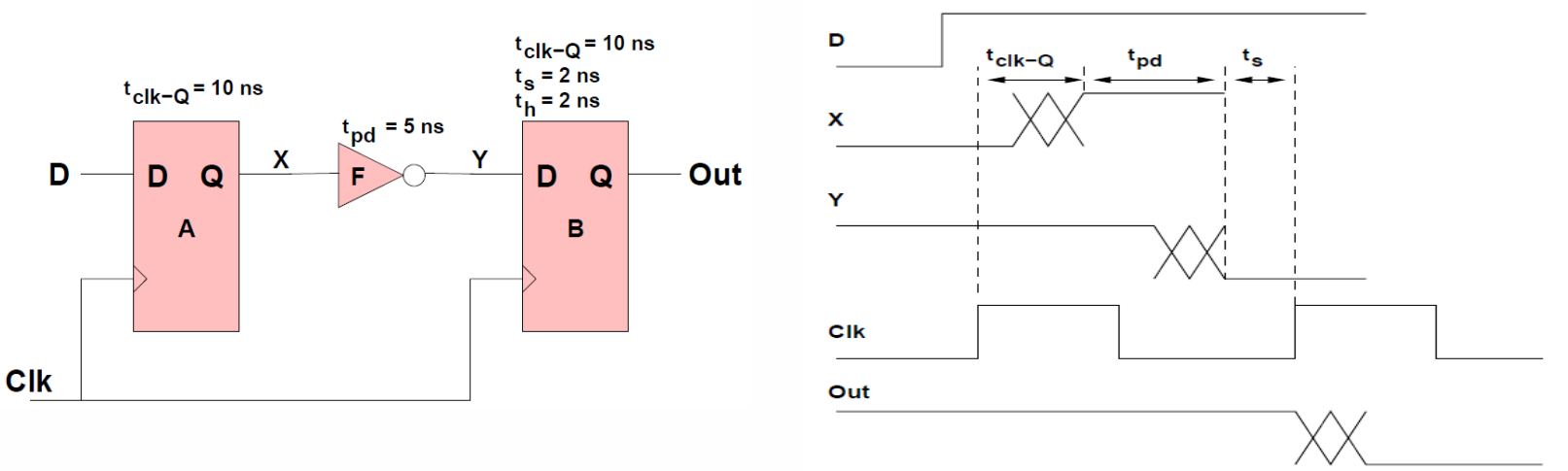
- Delays
  - Delays in Sequential Logic



Time parameter diagram

Timing control

- Delays
  - Determining Clk Max Frequency

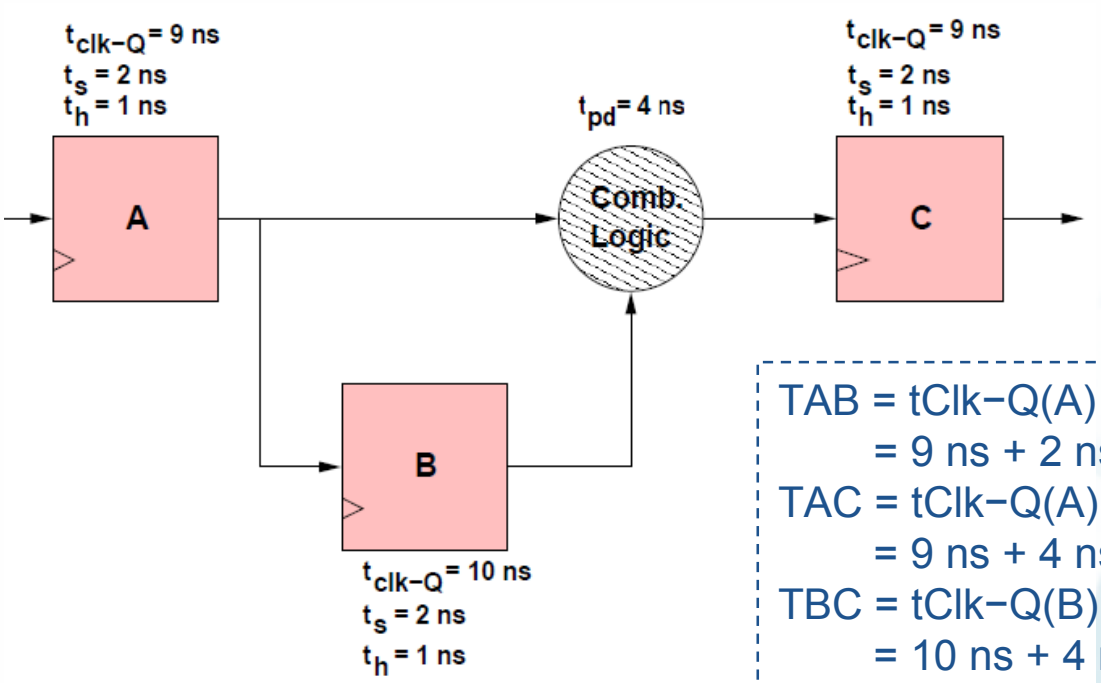


Min Clk Period: Tmin = Tclk-q+Tpd+Ts = 10ns+5ns+2ns=17ns

Max Clk Frequency: f = 1/Tmin = 1/17ns = 58.8MHz

Timing control

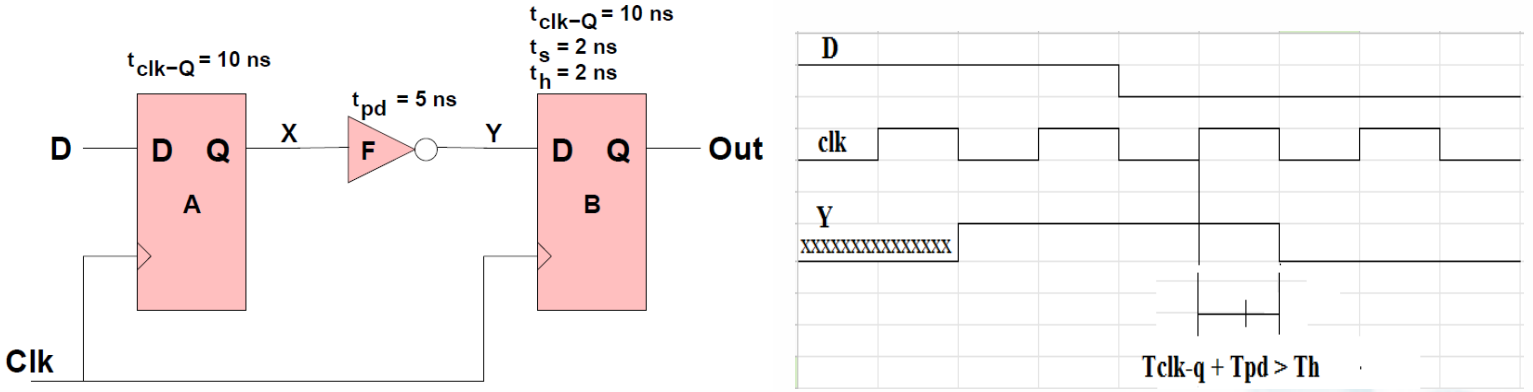
- Delays
  - Determining Clk Max Frequency



TAB = tClk−Q(A) + ts(B)
= 9 ns + 2 ns = 11 ns
TAC = tClk−Q(A) + tpd(Z) + ts(C)
= 9 ns + 4 ns + 2 ns = 15 ns
TBC = tClk−Q(B) + tpd(Z) + ts(C)
= 10 ns + 4 ns + 2 ns = 16 ns
What is max clk frequency?

Timing control

- Delays
  - Holding time validation



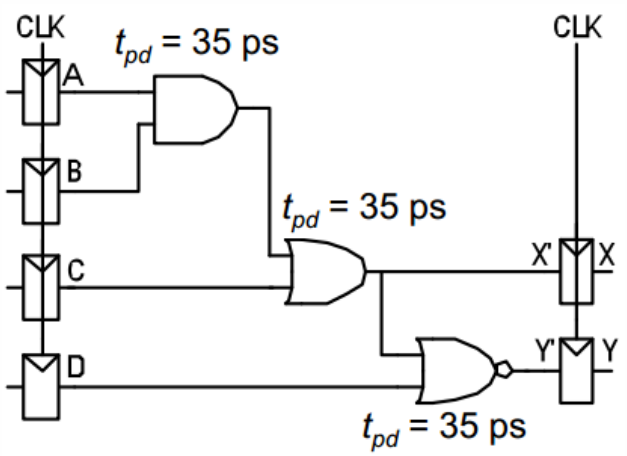
Y's holding time for B flip flop must be: Tclk-q + Tpd > Thold

e.g. Tclk-q = 0.5ns Tpd = 1ns Thold = 2ns, current Y = 0 when clk posedge is coming, Y -> 1 after 1.5ns. However, Y should hold 0 at least 2ns. So it has holding time issue.

Timing control

- Delays (Non-synthesizable)
  - Timing Analysis Examples

Tmin ≥ Tclk-q+Tpd(Combo logic)+Ts & Tclk-q + Tpd(Combo logic) > Thold

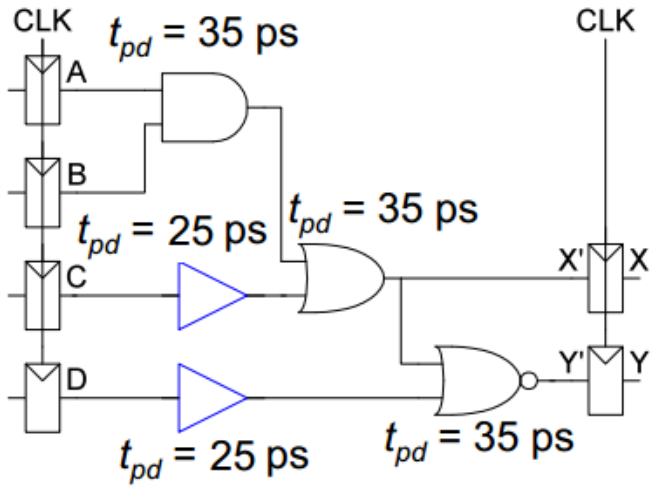


Tclk-q = 50ps Ts = 60ps Th = 90ps
What is max clk frequency? holding time violateion?
Tmin=50+3x35+60=215ps =>fmax=4.65GHz
50ps+35ps >? 90ps => violation

Timing control

- Delays (Non-synthesizable)
  - Timing Analysis Examples

Tmin ≥ Tclk-q+Tpd(Combo logic)+Ts & Tclk-q + Tpd(Combo logic) > Thold



Tclk-q = 50ps Ts = 60ps Th = 90ps
What is max clk frequency? holding time violateion?
Tmin=50+3x35+60=215ps =>fmax=4.65GHz

Fixing holding time violation
50+25+35 >? 90ps => No violation

Timing control

- Delays (Non-synthesizable)
  - Dataflow delays

- Net declaration delays (inertial delay)

  e.g. wire #10 out; assign out = in1 & in2;

  in1 & in2 ==> #10 (through wire) out

- Regular assignment delays

  e.g. wire out; assign #(1:2:3) out = (en) ? in1 : in2;

  in1/in2(saved to variable) ==> #2 out

  1:2:3 => min: typical: max

- Implicit continuous assigment

  e.g. wire #10 out = in1 & in2; //Like net delays

  in1 & in2 ==> #10 (through wire) out

Timing control

- Delays (Non-synthesizable)
  - Behavioral level delay
    - Inter-assignment delay

```
e.g.    reg     c;
        always@(a or b) begin
                #5 c = a + b;
        end
```

Note: Once one of a or b changes, compiler will wait for #5 to execute the whole assignment statement.

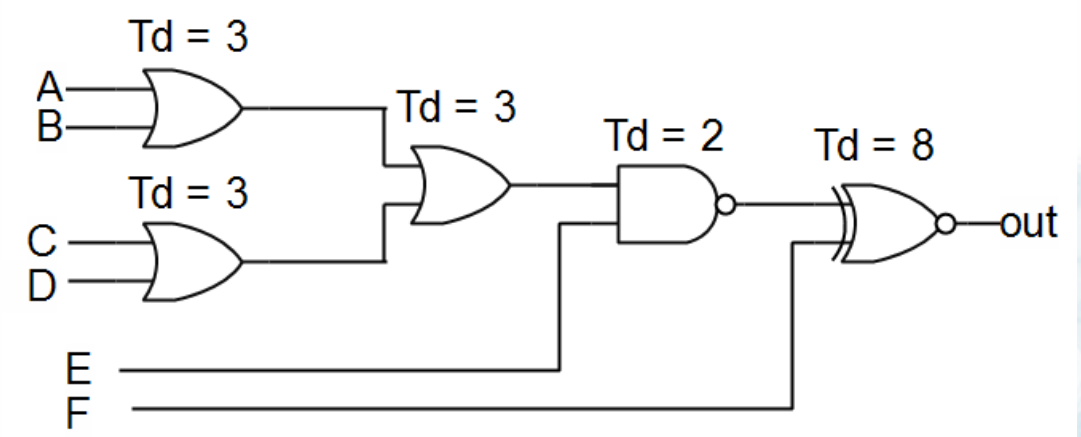
- Intra-assignment delay

```
e.g.    reg     c;
        always@(posedge clk) begin
                c <= #5 a + b;
        end
```

Note: Once clk posedge is coming, (a+b) is executed immediately and then wait for #5 to assignment the value to c.

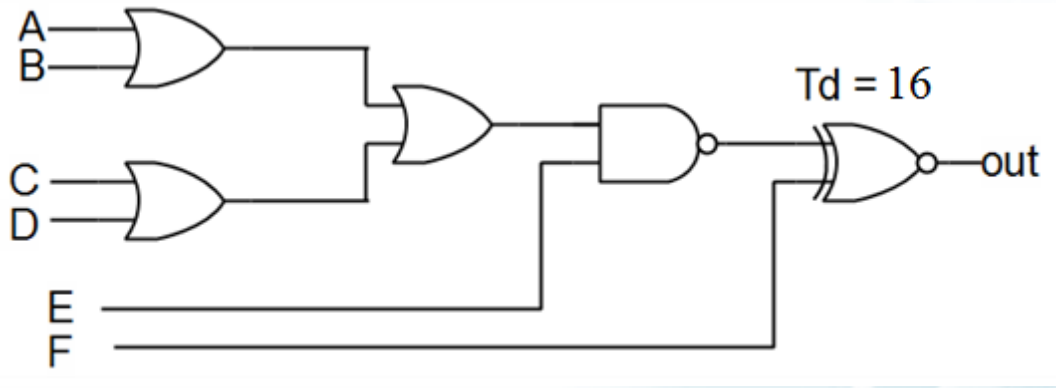
Timing control

- Circuit Delay Models (Non-synthesizable)
  - Distributed Delay
    - Is delay assigned to each gate in a module



Timing control

- Circuit Delay Models (Non-synthesizable)
  - Lumped Delay
    - Is delay assigned as a single delay in each module, mostly to the output gate of the module with the sum delay of the longest path



Timing control

- Circuit Delay Models (Non-synthesizable)
  - Pin-to-pin Delay
    - Is specified for each input to output pin pairing, rather than being associated with specific elements. This can be advantageous as it means that details of the internals of the module need not be known for the analysis to be carried out.
    - Parallel connection:

      - input pins => output pins. e.g. (a,b=>q,q') = 10

        same as: (a=>q) = 10; (b=>q') = 10

      - # of input pins must be equal to that of output pins.
    - Full connection:

      - input pins *> output pins. (a,b *> q,q') = 10

        same as: (a=>q) = 10; (a=>q') = 10; (b=>q) = 10; (b=>q') = 10

      - # of pins in both sides may not be equal.

      How about this connection (a[3:0] *> b[1:0])?

Timing control

- Specify blocks (Non-synthesizable)
  - Pin-to-pin timing specification
    - Timing/Delay info within *specify... ... endspecify* block
    - "specify" block can't be in "initial"/"always"/"assign" block.

```
module ornor(o,a,b,c);
        output          o;
        input           a, b, c;
        nor u1(n1, a, b);
        or  u2(o, c, n1);
        specify
                (a => o) = 2;
                (b => o) = 3;
                (c => o) = 1;
        endspecify
endmodule
```

Timing control

- Specify blocks (Non-synthesizable)
  - Pin-to-pin timing specification
    - "specparam" Statements

```
module ornor(o,a,b,c);
    output  o;
    input    a, b, c;
    nor u1(n1, a, b);
    or   u2(o, c, n1);
    specify
      specparam ao=2, bo=3, co=1;
                (a => o) = ao;
                (b => o) = bo;
                (c => o) = co;
      endspecify
endmodule
```

```
module adder(a,b,sum);
   parameter width = 8;


   input [width-1:0] a;
   input [width-1:0] b;
   output [width-1:0] sum;

   assign #5 sum = a + b
endmodule
```

Timing control

- Specify blocks (Non-synthesizable)
  - Difference between "specparam" and "parameter"
    - parameter:
      1. must be declared outside of "specify" block
      2. must be used outside of "specify" block
      3. can be changed by "defparam" keyword
    - specparam:
      1. must be declared within "specify" block
      2. must be used within "specify" block
      3. can't be changed by "defparam" keyword
      4. can be changed by SDF(Standard Delay Format) file

Timing control

- Specify blocks (Non-synthesizable)
  - Pin-to-pin path delay
    - Simple path delay statement

(input ± =>(or *>) output) = (delay)

e.g (in +=> out) = 10; (in -*> out) = 10

+ : out = in; - : out = ~in

delay:

1. one value (in -*> out) = 10
2. two values (in -*> out) = (10, 20) // Rise/fall time
3. three values (in -*> out) = (10, 20, 30) // 30: turnOff
4. six values (in -*> out) = (10,20,30,40,50,60)

   0->1, 1->0, 0->z, z->1, 1->z, z->0

5. twelve valus (in -*> out) = (1,2,3,4,5,6,7,8,9,10,11,12)

   0-1, 1-0, 0-z, z-1, 1-z, z-0, 0-x, x-1, 1-x, x-0, x-z, z-x

Rise/fall/turnOff = min: typical: max

e.g. (in -*> out) = (10:11:12, 20:21:22, 30:31:32)

Timing control

- Specify blocks (Non-synthesizable)
  - Pin-to-pin path delay

    - Edge-sensitive path delay:

    (posedge/negedge input1 =>/*> (output ±: input2)) = (delay);

    e.g. (posedge clk =>(q -: d))=(10,20);
    clk's edge => q, and q =~d; q's Lo->Hi delay=10 fall delay=20

    - State-dependent path delay:

    if (a & b) (posedge/negedge input1 =>/*> (output ±: input2)) = (delay);
    if (!a) (simple) or (edge-sensitive path delay)
    ifnone (input ± =>(or *>) output) = (delay)
    Note: no else statement

    e.g. if ({c, d} == 2'b10) (c, d *> q) = 15;
    if ({c, d} != 2'b10) (c, d *> q) = 12;

Timing control

- Specify blocks (Non-synthesizable)

Specify Block Examples
`(a => b) = 1.8;` //parallel connection path; one delay for all output transitions
`(a -*> b) = 2:3:4;/*`full connection path; one **`min:typ:max`** delay range for all output transitions; **`b`** receives the inverted value of **`a */`**

`specparam t1 = 3:4:6,` //different path delays for rise, fall transitions
`t2 = 2:3:4;`
`(a => y) = (t1,t2);`

`(a *> y1,y2) = (2,3,4,3,4,3);`//different delays for 6 output transitions
`(posedge clk => (qb -: d)) = (2.6, 1.8);` `/*` edge-sensitive path delay; timing path is positive edge of clock to qb; qb receives the inverted value of data`*/`
`if (rst && pst) (posedge clk=> (q +: d))=2;` //state-dependent edge sensitive path delay
`if (opcode == 3'b000)` // state-dependent path delays; `(a,b *> o) = 15;` // an ALU with different delays for certain `if (opcode == 3'b001)` // operations; `(a,b *> o) = 25;` `ifnone (a,b *> o) = 10;` // (default delay has no condition)

Timing control

- Timing checks(Non-synthesizable)
  - Must be inside the specify blocks
  - The most commonly used timing checks
    - $setup: $setup(data_change, reference, time_limit);
      e.g. $setup(d, posedge clk, 2)
    - $hold: $hold(reference, data_change, time_limit);
      e.g. $hold(posedge clk, d, 2)
    - $width: $width(reference1, time_limit);
      e.g. $width(posedge clk, 2)
    - $setuphold
    - $skew
    - $period
    - $recovery

Timing control

- Timing checks(Non-synthesizable)

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

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

```
`include "dff.v"
module dffTB;
   reg  clk, d;
   wire q, clk2, d2;
   dff uDFF(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
     #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
```

Timing control

- Timing checks(Non-synthesizable)

```
 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
```

```
"dffTB.v", 36: Timing violation in dffTB
    $setup( d2:14, posedge clk2:15, limit: 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
```

```
"dffTB.v", 37: Timing violation in dffTB
    $hold( posedge clk2:21, d2:22, limit: 2 );
                23   1   1 0
                24   0   0 0
                25   0   1 0
"dffTB.v", 38: Timing violation in dffTB
    $width( negedge d2:24,  : 25, limit: 2 );
```

Timing control

- Event Timing Control (Non-synthesizable)
  - Edge-triggered event control

    such as @(posedge clock) or @(negedge clock)

```
always @( posedge clk) begin
    reg1 <= #25 in_1;
    reg2 <= @(negedge clk) in_2 ^ in_3;
    reg3 <= in_1;
end
```

Note: @(negedge clk): just one clock cycle

```
module edgeTrig();
reg enable, clk, trigger;
always @ (posedge enable) begin
  trigger = 0;
    repeat (5) begin   // Wait for 5 clock cycles
       @ (posedge clk) ;
    end
  trigger = 1;
end
endmodule
```

  - Named event control
    - Timing control by event variable
    - event variable doesn't have a value, like pulse.
    - trigger operator: ->

Timing control

- Event Timing Control (Non-synthesizable)
  - Edge-triggered event control
    - Named event control

      - event variable is not vector either

      - event variable can't be bounded to any values in initial/assign/always blocks.

      - Don't try to print event variable value. You will get error.

```
module testEvent();
    reg clock;
    event event1, event2;// Error: event[1:0] a,b;
    always @(posedge clock)
            -> event1; // Trigger event1.
    always @(event1) begin
// Error: $display("Strike1!!,event1=%b",event1);
                -> event2;
     end
```

```
      always @ event2 begin
                $display("Strike 2!!");
                $finish;
        end
        always #10 clock = ~ clock;
initial clock = 0;
endmodule
```

Timing control

- Event Timing Control (Non-synthesizable)
  - Level-sensitive event controlol
    - wait (<expression>) statement

    e.g. wait (a==1) b=0

    - expession can't be posedge/negedge
    - wait statement could be in initial/always block
    - Don't do that like wait (a==1'bx) to control timing
    - difference between wait() and @()
      - expression in wait is level sensitive (true/false)
      - expression in @ is edge sensitive (at changing moment)

Timing control

- Event Timing Control (Non-synthesizable)
  - Level-sensitive event controlol

```
module testWait;
        reg D, Clock,Reset;
        dffWait u1(D, Q, Clock, Reset);
        initial begin
                D=1; Clock=0;Reset=1'b1;
                #15 Reset=1'b0;
                #20 D=0;
        end
        always #10 Clock = !Clock;
        initial begin
                $display("T  Clk D Q Reset");
$monitor("%2g",$time,,Clock,,,,D,,Q,,Reset);
                #50 $finish;
        end
endmodule
```

```
module dffWait(D, Q, Clock, Reset);
        output Q;
        input D, Clock,Reset;
        reg Q;
        wire D;
        always @(posedge Clock)
                if (Reset != 1) Q = D;
        always begin
                //wait (posedge Reset) Q = 0;
                wait (Reset == 1) Q = 0;
                //wait (Reset != 1);
        end
endmodule
```

what will happen?if no wait (Reset != 1);
Ans: infinite loop
Note: Error posedge in wait control