Advanced Digital Design

Dr. Sudeendra kumar K

Department of Electronics and Communication Engineering

ADVANCED DIGITAL DESIGN

Lecture-2: Two Segment Coding in Verilog

Sudeendra kumar K
Department of Electronics and Communication Engineering

Advanced Digital Design

Contents

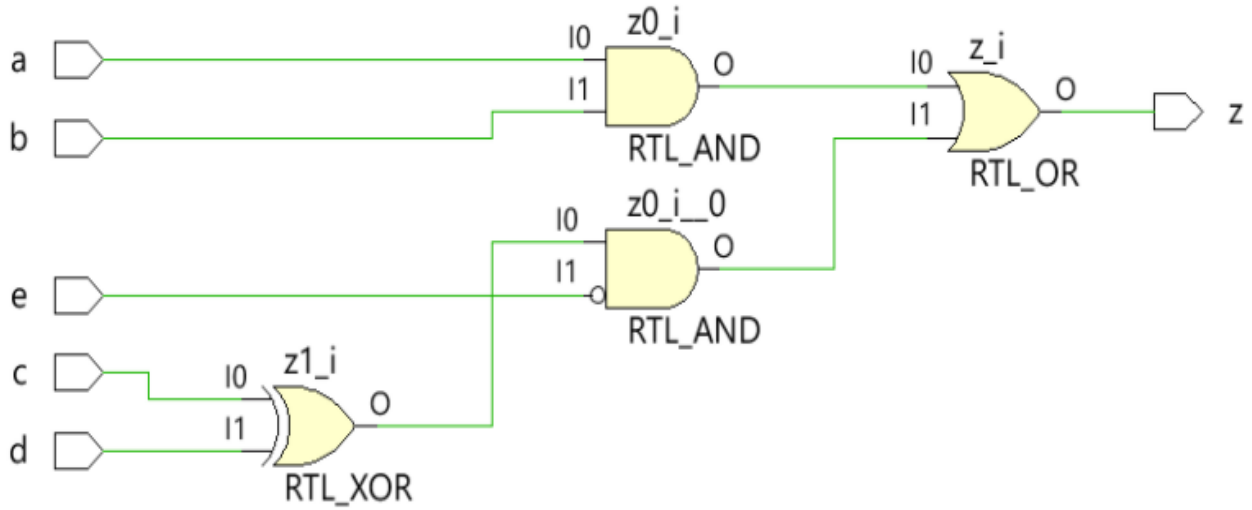
- Verilog Examples
- Two segment Coding
- Finite State machines
- FSM Examples

PES UNIVERSITY ONLINE

Advanced Digital Design

Simple Combinational Examples

```
module combo (  input   a, b, c, d, e,
                        output  z);

    assign z = ((a & b) | (c ^ d) & ~e);

endmodule
```



Courtesy: chipverify.com

Advanced Digital Design
Simple Combinational Examples

```
module tb;
    // Declare testbench variables
  reg a, b, c, d, e;
  wire z;
  integer i;

  // Instantiate the design and connect design inputs/outputs with
  // testbench variables
  combo u0 ( .a(a), .b(b), .c(c), .d(d), .e(e), .z(z));

  initial begin
    // At the beginning of time, initialize all inputs of the design
    // to a known value, in this case we have chosen it to be 0.
    a <= 0;
    b <= 0;
    c <= 0;
    d <= 0;
    e <= 0;

    // Use a $monitor task to print any change in the signal to
    // simulation console
    $monitor ("a=%0b b=%0b c=%0b d=%0b e=%0b z=%0b",
              a, b, c, d, e, z);
  end
endmodule
```

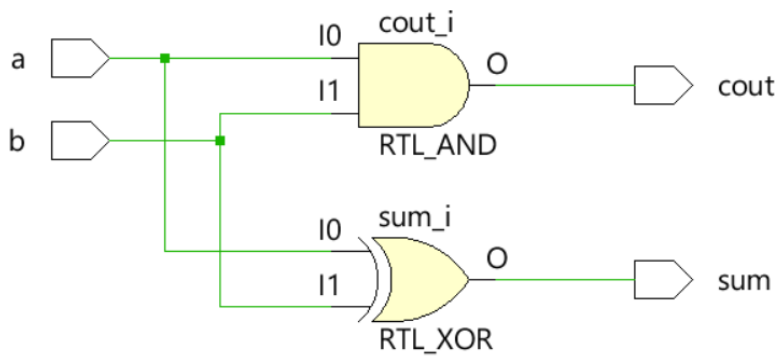
```
module combo ( input   a, b, c, d, e,
               output  z);

    assign z = ((a & b) | (c ^ d) & ~e);

endmodule
```

Courtesy: chipverify.com

Advanced Digital Design

Half Adder

```
module ha ( input   a, b,
                    output  sum, cout);

    assign sum  = a ^ b;
    assign cout = a & b;
endmodule
```

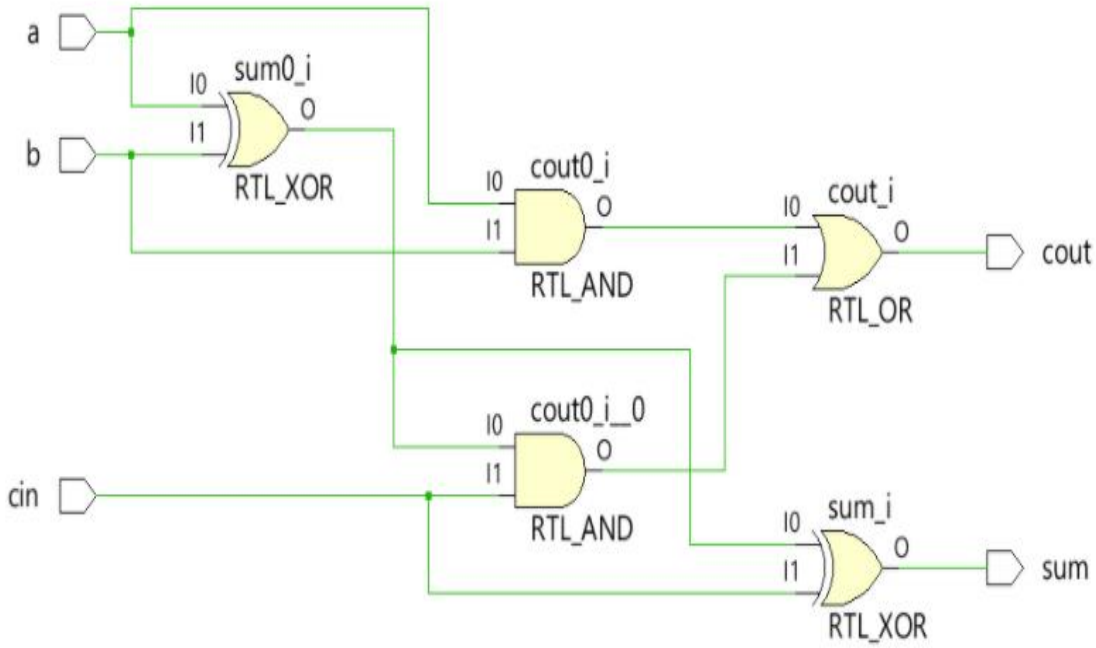



```
module tb;
    // Declare testbench variables
  reg a, b;
  wire sum, cout;
  integer i;

  // Instantiate the design and connect design inputs/outputs with
  // testbench variables
  ha u0 ( .a(a), .b(b), .sum(sum), .cout(cout));

  initial begin
    // At the beginning of time, initialize all inputs of the design
    // to a known value, in this case we have chosen it to be 0.
    a <= 0;
    b <= 0;

    // Use a $monitor task to print any change in the signal to
    // simulation console
    $monitor("a=%0b b=%0b sum=%0b cout=%0b", a, b, sum, cout);

    // Because there are only 2 inputs, there can be 4 different input combinations
    // So use an iterator "i" to increment from 0 to 4 and assign the value
    // to testbench variables so that it drives the design inputs
    for (i = 0; i < 4; i = i + 1) begin
      {a, b} = i;
      #10;
    end
  end
endmodule
```

Courtesy: chipverify.com

Advanced Digital Design
Full Adder

```
module fa ( input   a, b, cin,
                    output  sum, cout);

    assign sum  = (a ^ b) ^ cin;
    assign cout = (a & b) | ((a ^ b) & cin);
endmodule
```



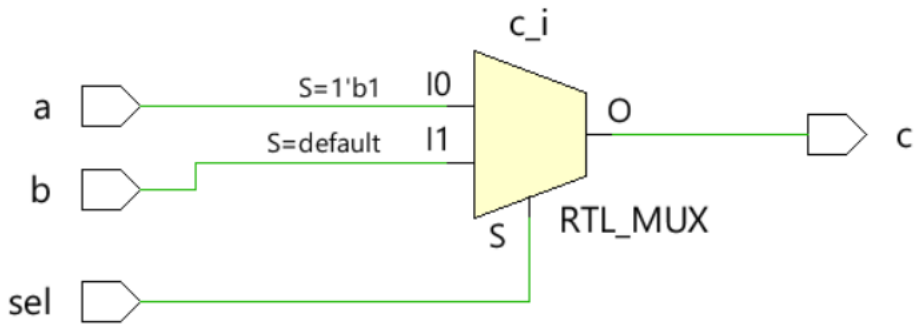

```
module tb;
  reg a, b, cin;
  wire sum, cout;
  integer i;

  fa u0 ( .a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));

  initial begin
    a <= 0;
    b <= 0;

    $monitor("a=%0b b=%0b cin=%0b sum=%0b cout=%0b", a, b, cin, sum, cout);

    for (i = 0; i < 7; i = i + 1) begin
      {a, b, cin} = i;
      #10;
    end
  end
endmodule
```

Courtesy: chipverify.com

Advanced Digital Design

2:1 Multiplexer

```
module mux_2x1 (input   a, b, sel,
                        output  c);


    assign c = sel ? a : b;
endmodule
```




```
module tb;
    // Declare testbench variables
  reg a, b, sel;
  wire c;
  integer i;

  // Instantiate the design and connect design inputs/outputs with
  // testbench variables
  mux_2x1 u0 ( .a(a), .b(b), .sel(sel), .c(c));

  initial begin
    // At the beginning of time, initialize all inputs of the design
    // to a known value, in this case we have chosen it to be 0.
    a <= 0;
    b <= 0;
    sel <= 0;

    $monitor("a=%0b b=%0b sel=%0b c=%0b", a, b, sel, c);

    for (i = 0; i < 3; i = i + 1) begin
      {a, b, sel} = i;
      #10;
    end
  end
endmodule
```

Courtesy: chipverify.com

Advanced Digital Design

Operators

Type of operation	Operator symbol	Description	Number of operands
Arithmetic	+	addition	2
	−	subtraction	2
	*	multiplication	2
	/	division	2
	%	modulus	2
	**	exponentiation	2
Shift	>>	logical right shift	2
	<<	logical left shift	2
	>>>	arithmetic right shift	2
	<<<	logical left shift	2
Relational	>	greater than	2
	<	less than	2
	>=	greater than or equal to	2
	<=	less than or equal to	2
Equality	==	equality	2
	!=	inequality	2
	===	case equality	2
	!==	case inequality	2
Bitwise	~	bitwise negation	1
	&	bitwise and	2
	|	bitwise or	2
	^	bitwise xor	2
Reduction	&	reduction and	1
	|	reduction or	1
	^	reduction xor	1
Logical	!	logical negation	1
	&&	logical and	2
	||	logical or	2
Concatenation	{ }	concatenation	any
	{ { } }	replication	any
Conditional	? :	conditional	3

Operator	Precedence
! ~ + − (unary)	highest
**	
* / %	
+ − (binary)	
>> << >>> <<<	
< <= > >=	
== != === !==	
&	
^	
|	
&&	
||	
?:	lowest

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design

Binary Decoder using if statement and case statement

```
module decoder_2_4_if
   (
    input wire [1:0] a,
    input wire en,
    output reg [3:0] y
   );

   always @*
      if (en==1'b0)          // can be written as (~en)
         y = 4'b0000;
      else if (a==2'b00)
         y = 4'b0001;
      else if (a==2'b01)
         y = 4'b0010;
      else if (a==2'b10)
         y = 4'b0100;
      else
         y = 4'b1000;

endmodule
```

```
module decoder_2_4_case
   (
    input wire [1:0] a,
    input wire en,
    output reg [3:0] y
   );

   always @*
      case({en,a})
         3'b000, 3'b001, 3'b010, 3'b011: y = 4'b0000;
         3'b100: y = 4'b0001;
         3'b101: y = 4'b0010;
         3'b110: y = 4'b0100;
         3'b111: y = 4'b1000; // default can also be used
      endcase
```

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design
Priority Encoder using case statement

```
module prio_encoder_case
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      case(r)
         4'b1000, 4'b1001, 4'b1010, 4'b1011,
         4'b1100, 4'b1101, 4'b1110, 4'b1111:
            y = 3'b100;
         4'b0100, 4'b0101, 4'b0110, 4'b0111:
            y = 3'b011;
         4'b0010, 4'b0011:
            y = 3'b010;
         4'b0001:
            y = 3'b001;
         4'b0000:      // default can also be used
            y = 3'b000;
     endcase

endmodule
```

Advanced Digital Design
Casex and casez statements

- There are two variations in addition to the regular case statement. In a casez statement, the z value and the ? character in the item expression are treated as don't-care.
- In a casex statement, the z and x values and the ? character in the item expression are treated as don't-care. Since the z and x values may appear in simulation, the ? character is preferred.

```
module prio_encoder_casez
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      casez(r)
         4'b1???: y = 3'b100;
         4'b01??: y = 3'b011;
         4'b001?: y = 3'b010;
         4'b0001: y = 3'b001;
         4'b0000: y = 3'b000; // default can also be used
      endcase

endmodule
```

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design
Full case and Parallel case

- When all possible binary values of the [case-exprl] expression are covered by the item expressions, the statement is known as a full case statement.

```
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   default: y = 1'b1; // y gets 1 for unspecified values
endcase
```

or as

```
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   3'b000: y = 1'b1;
   default: y = 1'bx; // y gets don't-care
endcase
```

Case (a)
2'b00: y = 2'b00;
2'b01
2'b10
2'b11: y = 2'b11;
default : y = 2'b00;

- When the values in the item expressions are mutually exclusive (i.e., a value appears in only one item expression), the statement is know as a parallel case statement. For example, the previous casez statement is not a parallel case statement since the value 3 ' b111 appears twice.

Advanced Digital Design

Sign Adder

- An integer can be represented in sign-magnitude format, in which the MSB is the sign and the remaining bits form the magnitude. For example, 3 and -3 become "00 1 1 " and " 10 1 1 “ in 4-bit sign-magnitude format.
- If the two operands have the same sign, add the magnitudes and keep the sign.
- If the two operands have different signs, subtract the smaller magnitude from the larger one and keep the sign of the number that has the larger magnitude.
- One possible implementation is to divide the circuit into two stages. The first stage sorts the two input numbers according to their magnitudes and routes them to the *max and min* signals.
- The second stage examines the signs and performs addition or subtraction on the magnitude accordingly.
- Note that since the two numbers have been sorted, the magnitude of *max* is always larger than that of *min* and the final sign is the sign of *max.*

Sign Adder

```
module sign_mag_add
   #(
    parameter N=4
   )
   (
    input wire [N-1:0] a, b,
    output reg [N-1:0] sum
   );

   // signal declaration
   reg [N-2:0]  mag_a, mag_b, mag_sum, max, m
   reg sign_a, sign_b, sign_sum;

   //body
   always @*
   begin
      // separate magnitude and sign
      mag_a = a[N-2:0];
      mag_b = b[N-2:0];
      sign_a = a[N-1];
      sign_b = b[N-1];
      // sort according to magnitude
      if (mag_a > mag_b)
         begin
            max = mag_a;
            min = mag_b;
            sign_sum = sign_a;
         end
      else
         begin
            max = mag_b;
            min = mag_a;
            sign_sum = sign_b;
         end
      // add/sub magnitude
      if (sign_a==sign_b)
         mag_sum = max + min;
      else
         mag_sum = max - min;
      // form output
      sum = {sign_sum, mag_sum};
   end
endmodule
```

Advanced Digital Design

Barrel Shifter

- Although Verilog has built-in shift functions, there is no rotation operation. we examine an 8-bit barrel shifter that rotates an arbitrary number of bits to the right.
- The circuit has an 8-bit data input, a, and a 3-bit control signal, ***amt,*** which specifies the amount to be rotated.
- The first design uses a case statement to exhaustively list all combinations of the ***amt*** signal and the corresponding rotated results.

Y = 1'b0;
Y = 3'd4;
Y = 8'h8;
Y = 3'o4;

```
module barrel_shifter_case
   (
     input wire [7:0] a,
     input wire [2:0] amt,
     output reg [7:0] y
    );

   // body
   always @*
      case(amt)
         3'o0: y = a;
         3'o1: y = {a[0], a[7:1]};
         3'o2: y = {a[1:0], a[7:2]};
         3'o3: y = {a[2:0], a[7:3]};
         3'o4: y = {a[3:0], a[7:4]};
         3'o5: y = {a[4:0], a[7:5]};
         3'o6: y = {a[5:0], a[7:6]};
         default: y = {a[6:0], a[7]};
      endcase

endmodule
```

Barrel Shifter

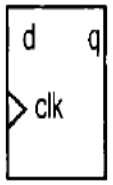
- Alternatively, we can construct the circuit by stages. In the ***n***th stage, the input signal is either passed directly to output or rotated right by positions. The ***n***th stage is controlled by the ***n***th bit of the ***amt*** signal. Assume that the bits of amt are $m_2m_1m_0$. The total rotated amount after three stages is which is the desired rotating amount.
- The total rotated amount after three stages is which is the desired rotating amount.

$$m_2 2^2 + m_1 2^1 + m_0 2^0,$$

```
module barrel_shifter_stage
   (
    input wire [7:0] a,
    input wire [2:0] amt,
    output wire [7:0] y
   );

   // signal declaration
   wire [7:0] s0, s1;

   // body
   // stage 0, shift 0 or 1 bit
   assign s0 = amt[0] ? {a[0], a[7:1]} : a;
   // stage 1, shift 0 or 2 bits
   assign s1 = amt[1] ? {s0[1:0], s0[7:2]} : s0;
   // stage 2, shift 0 or 4 bits
   assign y = amt[2] ? {s1[3:0], s1[7:4]} : s1;

endmodule
```

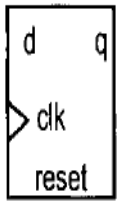
Courtesy: Pong P Chu

Advanced Digital Design
Simple Sequential Examples



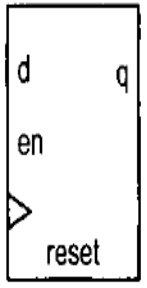
clk	q*
0	q
1	q
↓	d

(a) D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

(b) D FF with asynchronous reset



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

```
module d_ff_reset
   (
    input wire clk, reset,
    input wire d,
    output reg q
   );

   // body
   always @(posedge clk, posedge reset)
      if (reset)
         q <= 1'b0;
      else
         q <= d;

endmodule
```

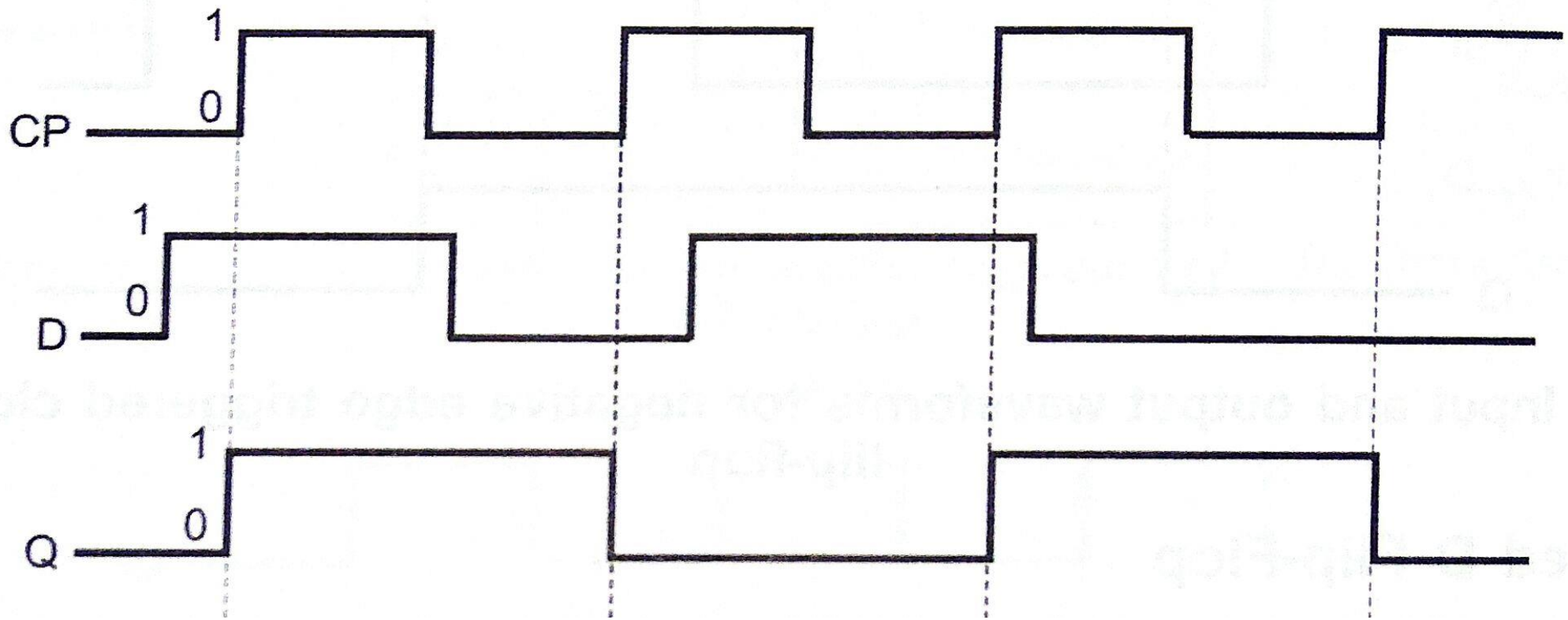
Reg q1;

Always @(posedge clk, posedge reset)
If (reset)
Q1 <= 1'b0;
Else
Q1 < = d;

Assign Q = Q1;
endmodule

Courtesy: Pong P Chu

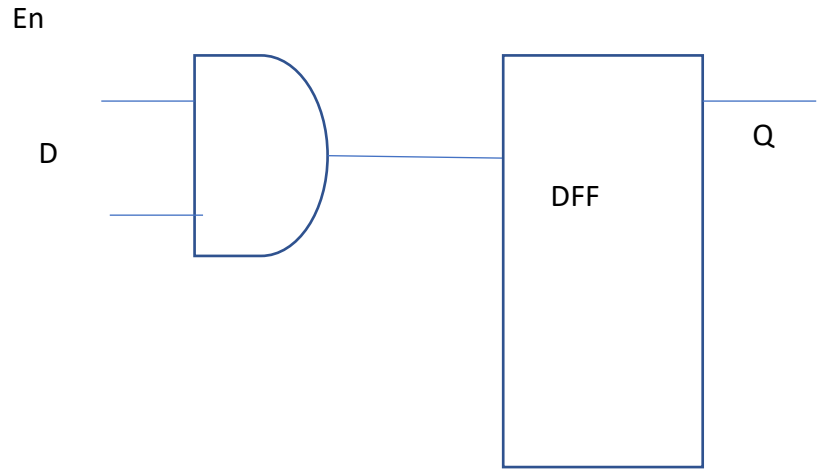
Always @ (posedge clk)
Q = D;



Advanced Digital Design

Two Segment Coding for DFF

```
module d_ff_en_2seg
    (
     input wire clk, reset,
     input wire en,
     input wire d,
     output reg q
    );

    // signal declaration
    reg r_reg, r_next;

    // body
    // D FF
    always @(posedge clk, posedge reset)
        if (reset)
            r_reg <= 1'b0;
        else
            r_reg <= r_next;

    // next-state logic
    always @*
        if (en)
            r_next = d;
        else
            r_next = r_reg;

    // output logic
    always @*
        q = r_reg;

endmodule
```



Always @ * == always @ (clk, reset, en, d)

Output q;
Reg q1;

Always @ *
Q1 = r_reg;

Assign q = q1;

Courtesy: Pong P Chu

Advanced Digital Design
Binary Counter

```
module free_run_bin_counter
   #(parameter N=8)
   (
    input wire clk, reset,
    output wire max_tick,
    output wire [N-1:0] q
   );

   //signal declaration
   reg [N-1:0] r_reg;
   wire [N-1:0] r_next;

   // body
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;   // {N{1b'0}}
      else
         r_reg <= r_next;

   // next-state logic
   assign r_next = r_reg + 1;
   // output logic
   assign q = r_reg;
   assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;
      //can also use (r_reg=={N{1'b1}})

endmodule
```

Courtesy: Pong P Chu

Advanced Digital Design

Universal Binary Counter

```
module univ_bin_counter
    #(parameter N=8)
    (
     input wire clk, reset,
     input wire syn_clr, load, en, up,
     input wire [N-1:0] d,
     output wire max_tick, min_tick,
     output wire [N-1:0] q
    );

    //signal declaration
    reg [N-1:0] r_reg, r_next;

    // body
    // register
    always @(posedge clk, posedge reset)
        if (reset)
            r_reg <= 0;   //
        else
            r_reg <= r_next;

    // next-state logic
    always @*
        if (syn_clr)
            r_next = 0;
        else if (load)
            r_next = d;
        else if (en & up)
            r_next = r_reg + 1;
        else if (en & ~up)
            r_next = r_reg - 1;
        else
            r_next = r_reg;

    // output logic
    assign q = r_reg;
    assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;
    assign min_tick = (r_reg==0) ? 1'b1 : 1'b0;

endmodule
```

Courtesy: Pong P Chu

```
module univcount (Resetn,Clock,Q,modsel);
input Resetn,Clock,modsel;
output[3:0] Q;

reg [3:0] Q;

always@(posedge Clock)
if (!Resetn)
Q <= 0;
else if (modsel == 0)
begin
if (Q == 4'b1111)
Q <= 4'b0000;
else
Q <= Q + 1;
end
else
begin
if (Q == 4'b0000)
Q <= 4'b1111;
else
Q <= Q – 1;
end
endmodule
```

How Verilog Coding Should not be done

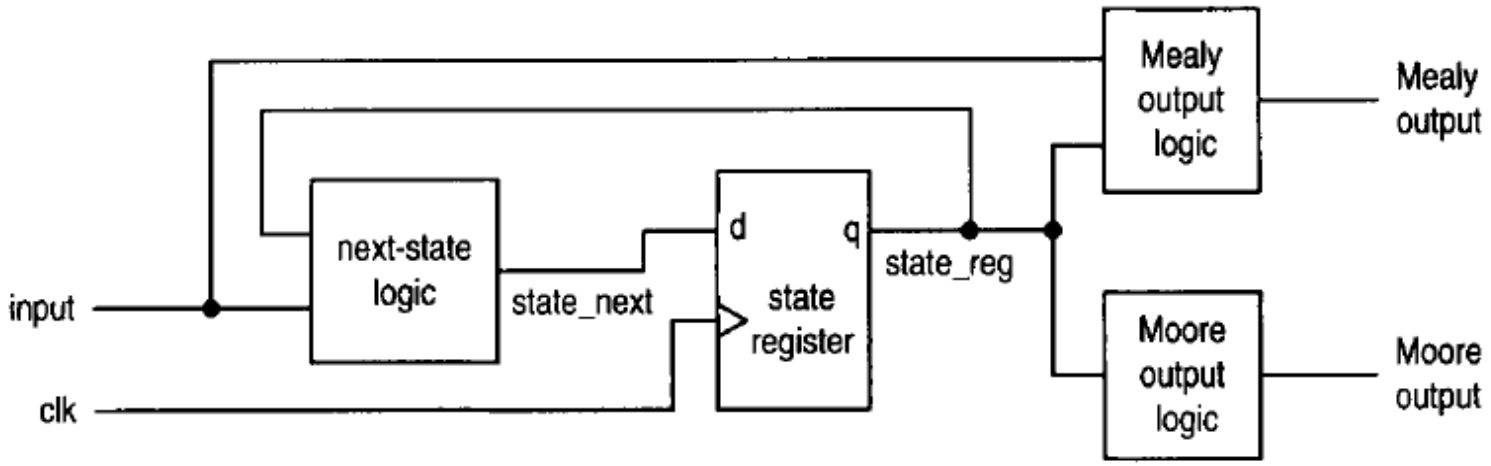
Advanced Digital Design

Finite State Machine

- An FSM (finite state machine) is used to model a system that transits among a finite number of internal states. The transitions depend on the current state and external input. Unlike a regular sequential circuit, the state transitions of an FSM do not exhibit a simple, repetitive pattern.
- Its next-state logic is usually constructed from scratch and is sometimes known as "random" logic. This is different from the next-state logic of a regular sequential circuit, which is composed mostly of "structured components, such as incrementors and shifters.

Mealy and Moore Outputs

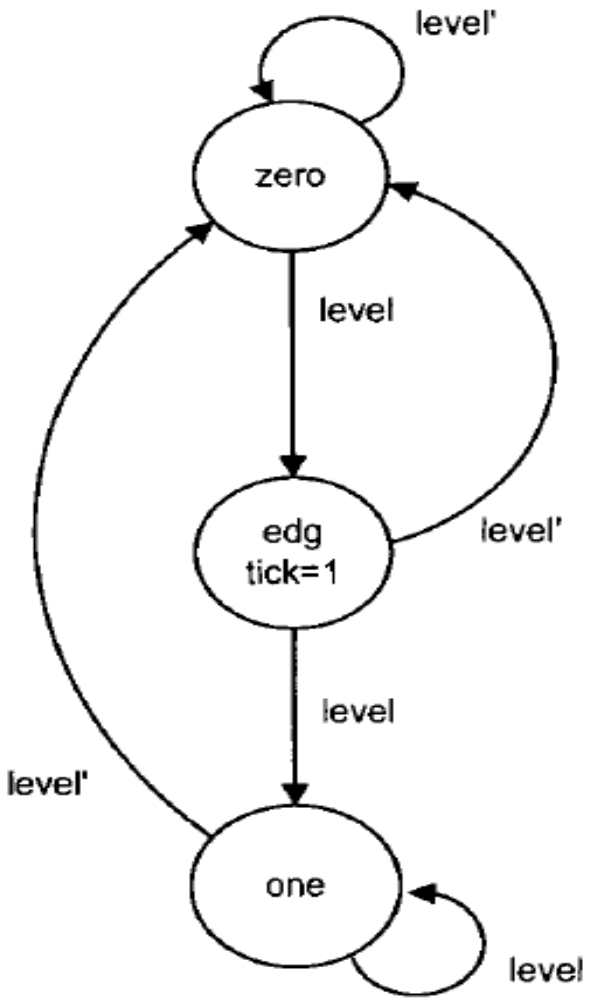
- An FSM is known as a Moore machine if the output is only a function of state, and is known as a Mealy machine if the output is a function of state and external input.
- Both types of output may exist in a complex FSM, and we simply refer to it as containing a Moore output and a Mealy output. The Moore and Mealy outputs are similar but not identical.



FSM Example: - Rising Edge Detector

- The rising-edge detector is a circuit that generates a short one-clock-cycle tick when the input signal changes from 0 to 1.
- It is usually used to indicate the onset of a slow time varying input signal.

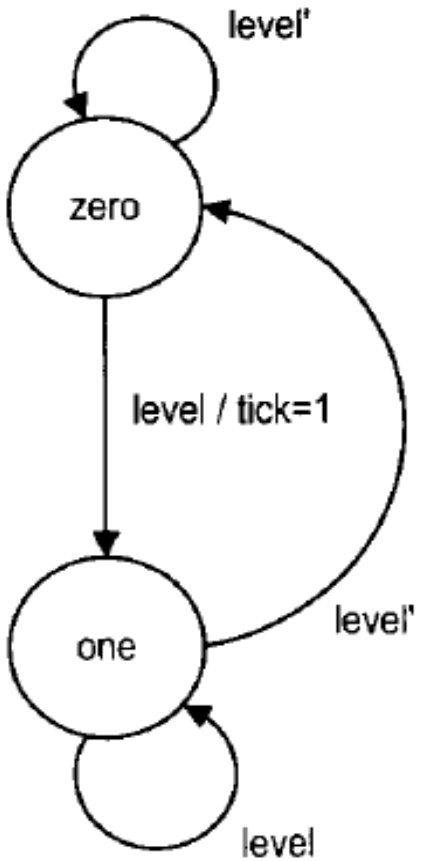
Moore FSM: Rising edge detector



- The ***z e r o*** and ***one*** states indicate that the input signal has been 0 and 1 for a while.
- The rising edge occurs when the input changes to 1 in the ***z e r o*** state.
- The FSM moves to the **e d g** state and the output, ***t i c k*** , is asserted in this state.

Courtesy: Pong P Chu

Advanced Digital Design
Mealy FSM: Rising Edge Detector

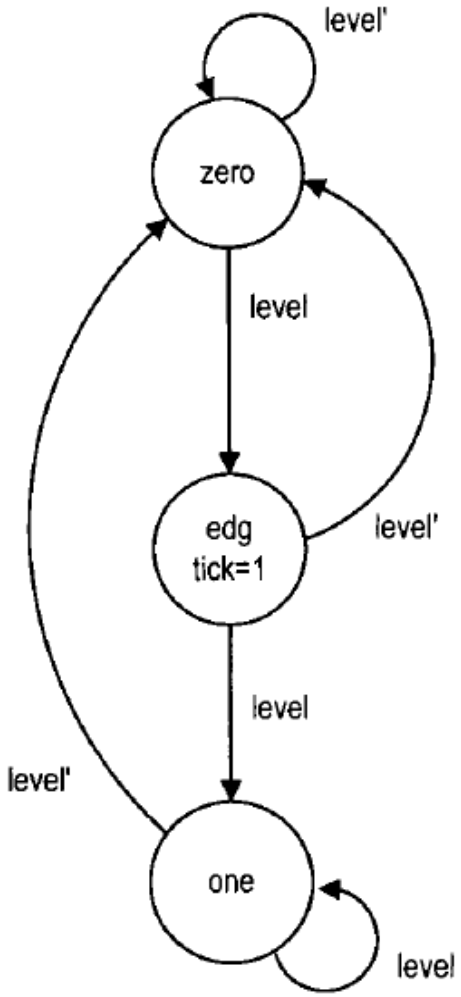


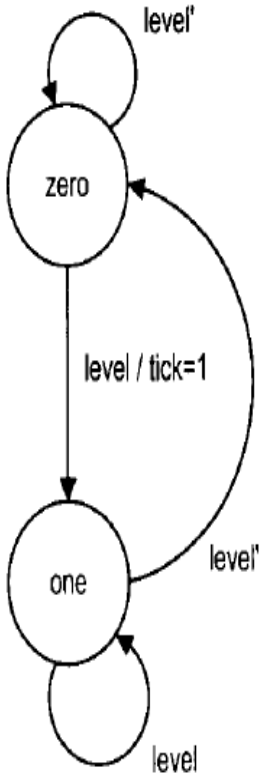
- The **zero** and **one** states have a similar meaning.
- When the FSM is in the zero state and the input changes to 1, the output is asserted immediately.
- The FSM moves to the one state at the rising edge of the next clock and the output is de-asserted.

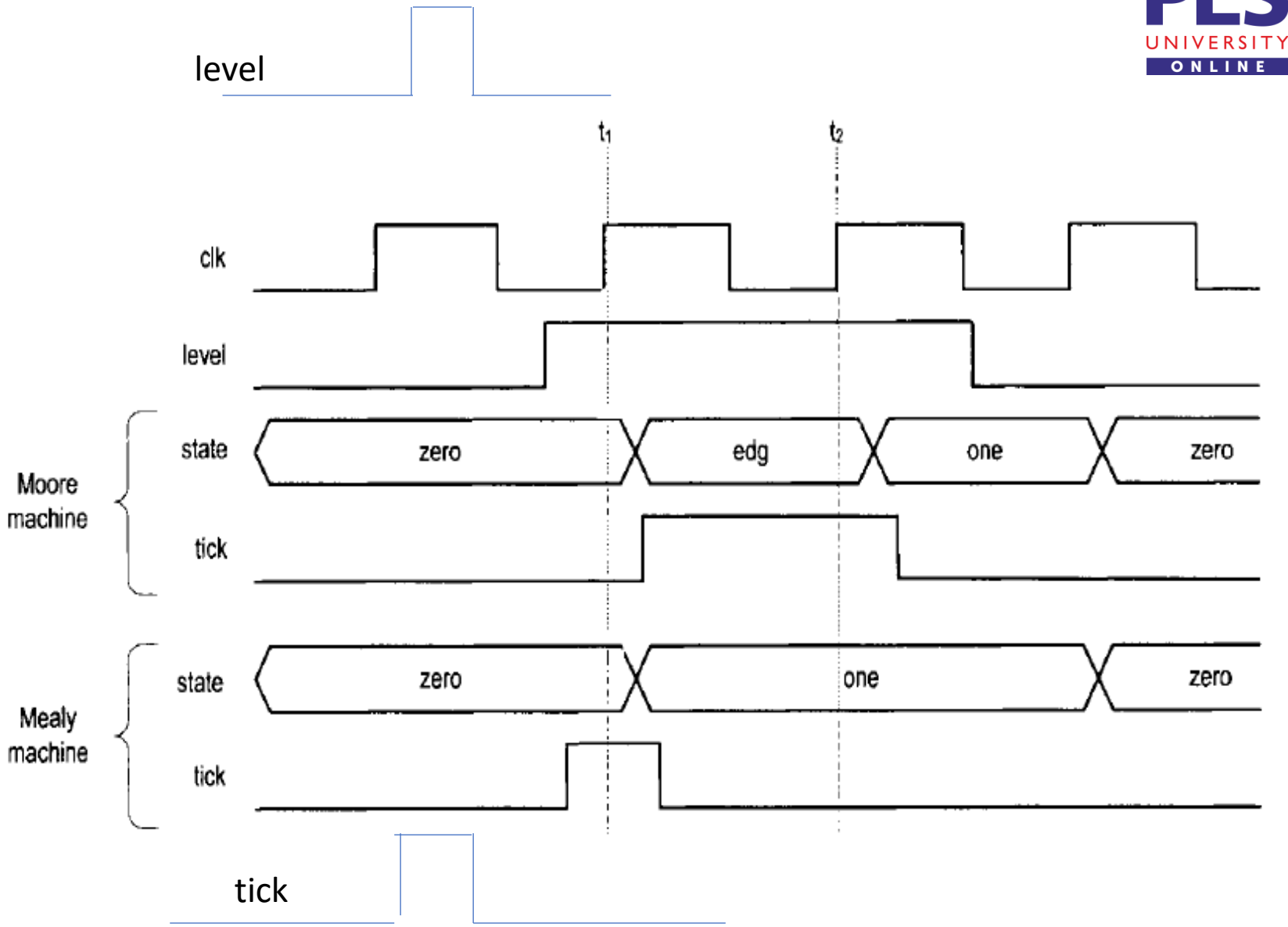
Courtesy: Pong P Chu

Advanced Digital Design

Waveforms of Mealy and Moore : Rising Edge Detector







Courtesy: Pong P Chu

Comparison

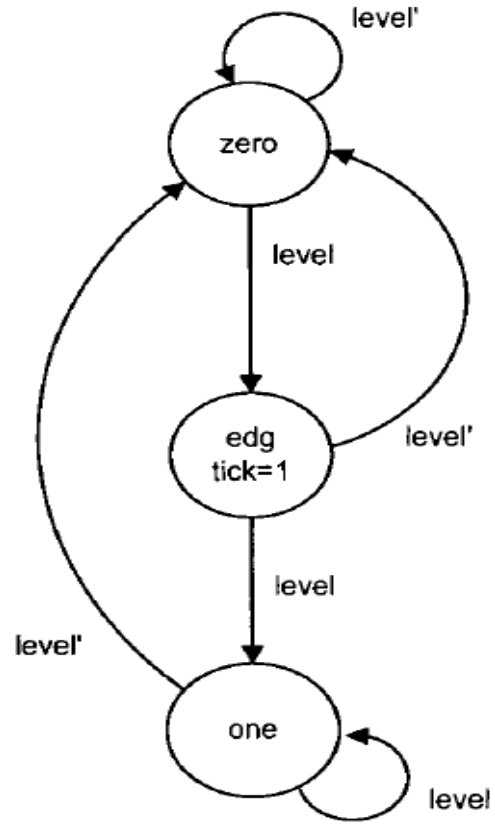
- The Mealy machine-based design requires fewer states and responds faster, but the width of its output may vary and input glitches may be passed to the output.
- The choice between the two designs depends on the subsystem that uses the output signal.
- Most of the time the subsystem is a synchronous system that shares the same clock signal.
- Since the FSM's output is sampled only at the rising edge of the clock, the width and glitches do not matter as long as the output signal is stable around the edge.

Courtesy: Pong P Chu

Advanced Digital Design

Rising Edge Detector: Moore

```
module edge_detect_moore
   (
    input wire clk, reset,
    input wire level,
    output reg tick
   );

   // symbolic state declaration
   localparam [1:0]
      zero = 2'b00,
      edg = 2'b01,
      one = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
   begin
      state_next = state_reg;  // default state: the same
      tick = 1'b0;             // default output: 0
      case (state_reg)
         zero:
            if (level)
               state_next = edg;
         edg:
            begin
               tick = 1'b1;
               if (level)
                  state_next = one;
               else
                  state_next = zero;
             end
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
      endcase
   end
endmodule
```

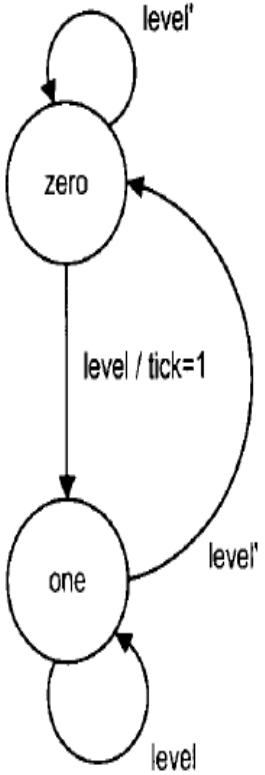


Courtesy: Pong P Chu

Advanced Digital Design

Rising Edge Detector: Mealy

```
module edge_detect_mealy
   (
    input wire  clk, reset,
    input wire  level,
    output reg tick
   );

   // symbolic state declaration
   localparam zero = 1'b0,
              one = 1'b1;

   // signal declaration
   reg state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
   begin
      state_next = state_reg;  // default state: the same
      tick = 1'b0;             // default output: 0
      case (state_reg)
         zero:
            if (level)
               begin
                  tick = 1'b1;
                  state_next = one;
               end
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
      endcase
   end

endmodule
```



Courtesy: FPGA Prototyping, Pong P Chu

References

- FPGA Prototyping using Verilog Examples, Pong P Chu

THANK YOU

Sudeendra kumar K

Department of Electronics and Communication Engineering

sudeendrakumark@pes.edu

Advanced Digital Design

Dr. Sudeendra kumar K

Department of Electronics and Communication Engineering

ADVANCED DIGITAL DESIGN

Lecture-3: Verilog Coding Guidelines

Sudeendra kumar K
Department of Electronics and Communication Engineering

Advanced Digital Design

Contents

- Verilog Coding Guidelines
- Verilog Race Conditions

Verilog Coding Guidelines

- Following are common errors found in combinational circuit codes:
  - A Variable assigned in multiple always blocks
  - A Incomplete sensitivity list
  - Incomplete branch and incomplete output assignment

Verilog Coding Guidelines

- Although the code is syntactically correct and can be simulated, it cannot be synthesized.
- Recall that each always block can be interpreted as a circuit part. The code above indicates that 'y' is the output of both circuit parts and can be updated by each part. No physical circuit exhibits this kind of behavior and thus the code cannot be synthesized.

```
reg y;
reg a, b, clear;
. . .
always @*
   if (clear) y = 1'b0;

always @*
   y = a & b;
```

```
always @*
   if (clear)
      y = 1'b0;
   else
      y = a & b;
```

Incomplete Sensitivity list in "Always" Block

```
always @(a, b)    // both a and b are in sensitivity list
   y = a & b;
```

If we forget to include b, the code becomes

```
always @(a)       // a missing from sensitivity list
   y = a & b;
```

Advanced Digital Design
Incomplete Branch Assignment

- Incomplete branch and incomplete output assignment are two common errors that lead to unintended memory. To avoid these, we should observe the following rules while developing code for combinational circuit:
  - Include all the branches of an if or case statement.
  - Assign a value to every output signal in every branch.

```
always @*
   if (a > b)        // eq not assigned in this branch
      gt = 1'b1;
   else if (a == b) // gt not assigned in this branch
      eq = 1'b1;
                     // final else branch is omitted
```

Verilog Coding Guidelines: Incomplete Branch Assignment

```
always @*
    if (a > b)
        begin
            gt = 1'b1;
            eq = 1'b0;
        end
    else if (a == b)
        begin
            gt = 1'b0;
            eq = 1'b1;
        end
    else    // i.e., a < b
        begin
            gt = 1'b0;
            eq = 1'b0;
        end
```

Advanced Digital Design
Verilog Coding Guidelines

- Assign a variable only in a single always block.
- Use blocking statements for combinational circuits.
- Use @*to include all inputs automatically in the sensitivity list.
- Make sure that all branches of the if and case statements are included.
- Make sure that the outputs are assigned in all branches.
- One way to satisfy the two previous guidelines is to assign default values for outputs in the beginning of the always block.
- Describe the desired full case and parallel case in code rather than using software directives or attributes.
- Be aware of the type of routing network inferred by different control constructs.
- Think hardware, not C code.

Advanced Digital Design
Verilog Coding Guidelines: Blocking and Non-Blocking

- A procedural assignment can only be used within an always block or initial block. There are two types of assignments: blocking assignment and non-blocking assignment.
- In a blocking assignment, the expression is evaluated and then assigned to the variable immediately, before execution of the next statement (the assignment thus "blocks" the execution of other statements). It behaves like the normal variable assignment in the C language.
- In a non-blocking assignment, the evaluated expression is assigned at the end of the always block (the assignment thus does not block the execution of other statements).
- The basic rule of thumb is:
  - Use blocking assignments for a combinational circuit.
  - Use non-blocking assignments for a sequential circuit.

```
[variable_name] = [expression];    // blocking assignment
[variable_name] <= [expression];   // nonblocking assignment
```

```
always (posedge clk)
begin
X1 = a + b; //blocking
Y = a + 2 ;
…
X2<= a + b; //Non-blocking
…..
Z = b +2;
…..
End
```

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design
Verilog Coding Guidelines: Non-Blocking Assignment

- A non-blocking assignment can be viewed as a 2-step assignment. At the beginning of a simulation time step, the right-hand-side (RHS) of the non-blocking assignment is (1) evaluated and at the end of the non-blocking assignment the left-hand-side (LHS) variable is (2) updated.
- A non-blocking assignment does not "block" other assignments from being executed between the evaluation and update steps of a non-blocking assignment; hence, the name "non-blocking."

Advanced Digital Design
Verilog Coding Guidelines

- When modeling sequential logic, use nonblocking assignments.
- When modeling latches, use nonblocking assignments.
- When modeling combinational logic with an always block, use blocking assignments.
- When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
- Do not mix blocking and nonblocking assignments in the same always block.
- Do not make assignments to the same variable from more than one always block.
- Use $strobe to display values that have been assigned using nonblocking assignments.
- Do not make assignments using #0 delays.

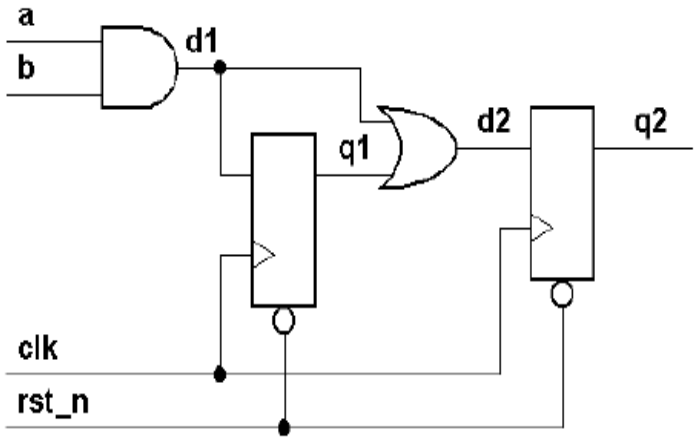
```
Always (posedge clk)
s = a xor b;
```

```
Always (posedge clk)
If (reset)
Q <= 0;
Else
Q < = d;
```

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design

Verilog Coding Guidelines: Example




```
module sblk1 (
  output reg q2,
  input      a, b, clk, rst_n);
  reg        q1, d1, d2;

  always @(a or b or q1) begin
    d1 = a & b;
    d2 = d1 | q1;
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      q2 <= 0;
      q1 <= 0;
    end
    else begin
      q2 <= d2;
      q1 <= d1;
    end
endmodule
```

Verilog Coding Guidelines

- HDL code frequently uses constant values in expressions and array boundaries. These values are fixed within the module and cannot be modified. One good design practice is to replace the "hard literals" with symbolic constants. It makes code clear and helps future maintenance and revision.
- In Verilog, a constant can be declared using the localparam (for"local parameter") keyword.

```
localparam DATA_WIDTH = 8,
           DATA_RANGE = 2**DATA_WIDTH - 1;
```

or define a symbolic port name:

```
localparam UART_PORT  = 4'b0001,
           LCD_PORT   = 4'b0010,
           MOUSE_PORT = 4'b0100;
```

Verilog Coding Guidelines: Blocking and Non-Blocking

- Use blocking assignments in always blocks that are written to generate combinational logic.
- Use nonblocking assignments in always blocks that are written to generate sequential logic.
- But why? In general, the answer is simulation related. Ignoring the above guidelines can still infer the correct synthesized logic, but the pre-synthesis simulation might not match the behavior of the synthesized circuit.

Verilog Coding Guidelines: Race conditioning

- 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.
- A problem with blocking assignments occurs when the RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block and both equations are scheduled to execute in the same simulation time step, such as on the same clock edge.
- If blocking assignments are not properly ordered, a race condition can occur. When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

Verilog Coding Guidelines: Race Conditions

- According to the IEEE Verilog Standard, the two always blocks can be scheduled in any order.
- If the first always block executes first after a reset, both y1 and y2 will take on the value of 1.
- If the second always block executes first after a reset, both y1 and y2 will take on the value 0. This clearly represents a Verilog race condition.

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 = 0;  // reset
    else     y1 = y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 = 1;  // preset
    else     y2 = y1;
endmodule
```

Courtesy: Pong P Chu

Verilog Coding Guidelines: Non-Blocking

- 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.

- Nonblocking assignments are only made to register data types and are therefore only permitted inside of procedural blocks, such as initial blocks and always blocks. Nonblocking assignments are not permitted in continuous assignments.

Courtesy: Pong P Chu

Verilog Coding Guidelines: Non-Blocking

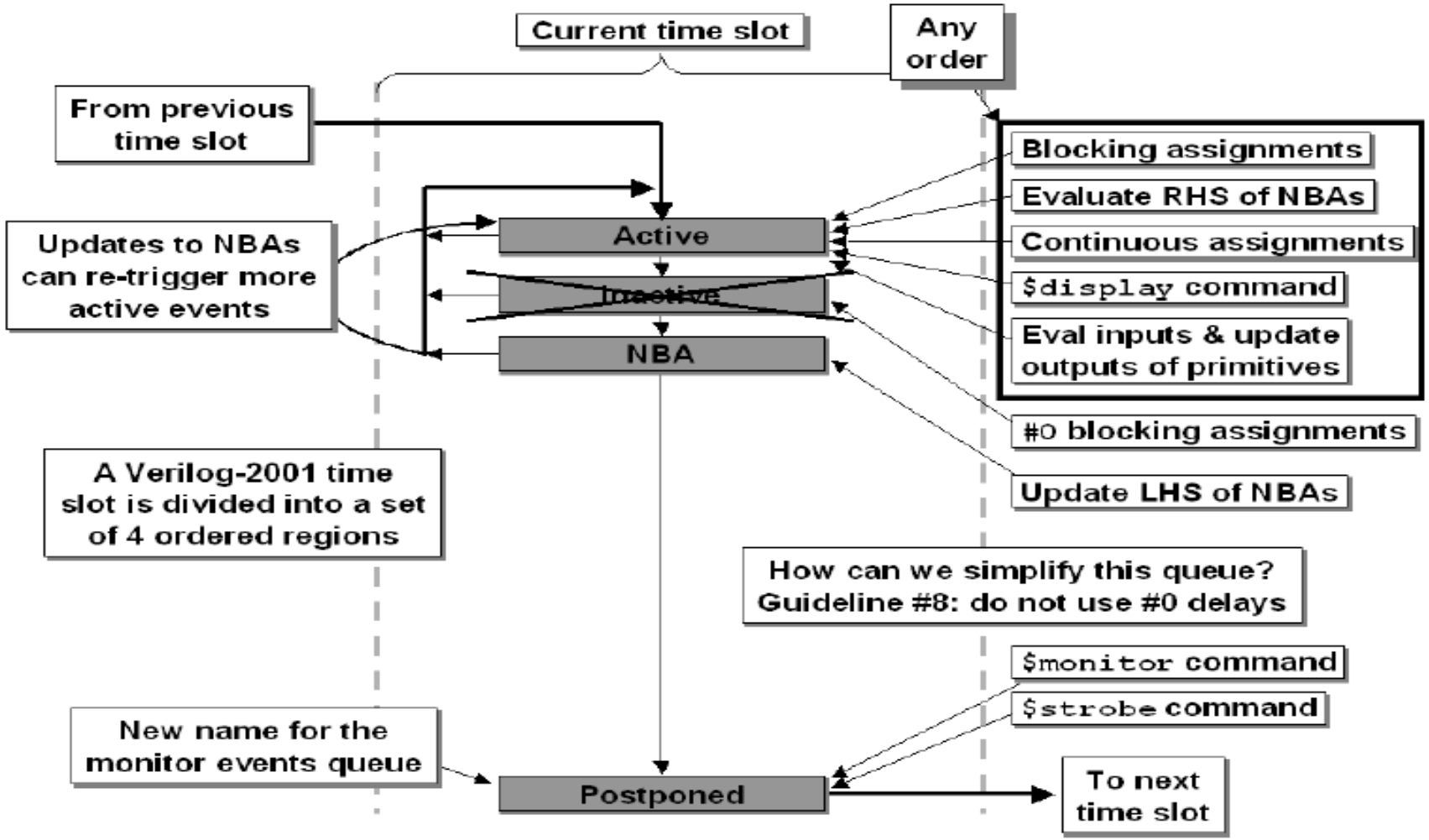
- Again, according to the IEEE Verilog Standard, the two always blocks can be scheduled in any order.
- No matter which always block starts first after a reset, both nonblocking RHS expressions will be evaluated at the beginning of the time step and then both nonblocking LHS variables will be updated at the end of the same time step.
- From a users perspective, the execution of these two nonblocking statements happen in parallel.

```
module fbosc2 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 <= 0;  // reset
    else     y1 <= y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 <= 1;  // preset
    else     y2 <= y1;
endmodule
```

Advanced Digital Design

Verilog Coding Guidelines: Verilog Stratified Queue

The "stratified event queue" is a fancy name for the different Verilog event queues that are used to schedule simulation events.



Courtesy: Pong P Chu

Verilog Coding Guidelines: stratified queue

- The active events queue is where most Verilog events are scheduled, including blocking assignments, continuous assignments, $display commands, evaluation of instance and primitive inputs followed by updates of primitive and instance outputs, and the evaluation of nonblocking RHS expressions.
- The LHS of nonblocking assignments are not updated in the active events queue.
- Events are added to any of the event queues but are only removed from the active events queue. Events that are scheduled on the other event queues will eventually become "activated," or promoted into the active events queue.
- Two other commonly used event queues in the current simulation time are the nonblocking assign updates event queue and the monitor events queue.
- The nonblocking assign updates event queue is where updates to the LHS expression of nonblocking assignments are scheduled. The RHS expression is evaluated in random order at the beginning of a simulation time step along with the other active events described above.

Courtesy: Pong P Chu

Verilog stratified queue

- The monitor events queue is where $strobe and $monitor display command values are scheduled.
- $strobe and $monitor show the updated values of all requested variables at the end of a simulation time step, after all other assignments for that simulation time step are complete.
- Adding #0-delay assignments to Verilog models needlessly complicates the analysis of scheduled events.

Courtesy: FPGA Prototyping, Pong P Chu

Non Self Triggering Always Blocks

- In general, a Verilog always block cannot trigger itself. This oscillator uses blocking assignments. 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.**

```
module osc1 (clk);
  output clk;
  reg    clk;

  initial #10 clk = 0;

  always @(clk) #10 clk = ~clk;
endmodule
```

```
Always   #10 clk = ~clk;
```

```
Always
begin
Clk = 1'b0;
#10;
Clk = 1'b1;
end
```

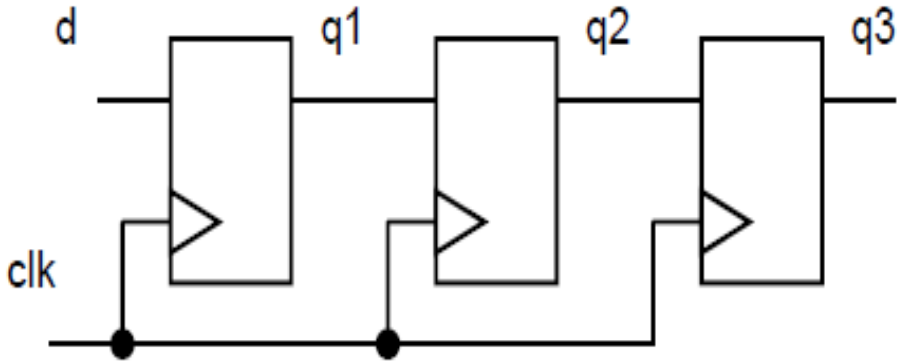
Courtesy: FPGA Prototyping, Pong P Chu

Self Triggering Always Block (not recommended)

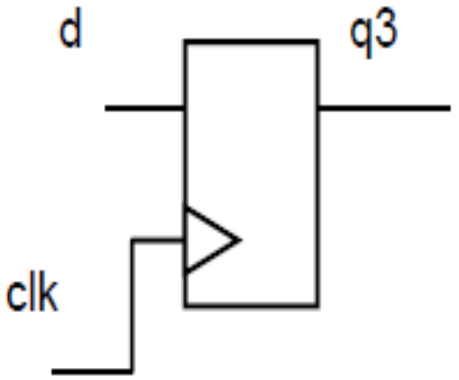
- After the first @(clk) trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the nonblocking assign updates event queue.
- Before the nonblocking assign updates event queue is "activated," the **@(clk) trigger** statement is encountered and the always 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. The osc2 example is self-triggering (which is not necessarily a recommended coding style).

```
module osc2 (clk);
  output clk;
  reg    clk;

  initial #10 clk = 0;

  always @(clk) #10 clk <= ~clk;
endmodule
```

Courtesy: FPGA Prototyping, Pong P Chu

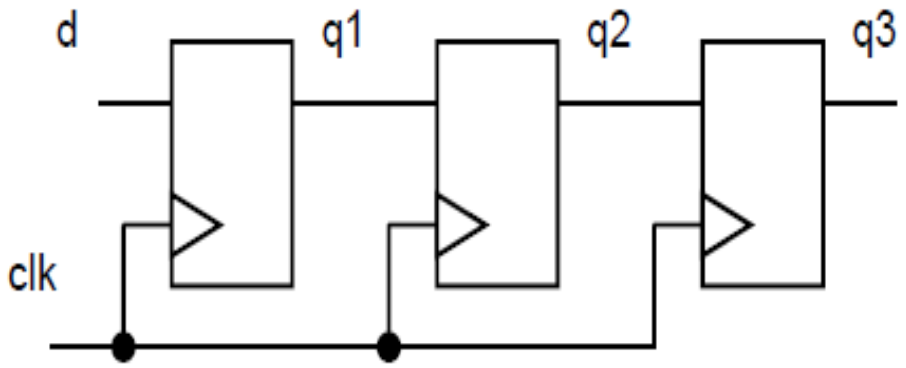



```
module pipeb1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

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



```
module pipeb2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

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

The blocking assignments have been carefully ordered to cause the simulation to correctly behave like a pipeline register.



Register/Pipeline Modelling

```
module pipeb3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1=d;

  always @(posedge clk) q2=q1;

  always @(posedge clk) q3=q2;
endmodule
```

- The blocking assignments have been split into separate always blocks.
- Verilog is permitted to simulate the always blocks in any order, which might cause this pipeline simulation to be wrong.
- This is a Verilog race condition! Executing the always blocks in a different order yields a different result.
- This Verilog code will synthesize to the correct pipeline register. This means that there might be a mismatch between the pre-synthesis and post-synthesis simulations.

```
module pipeb4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

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

Any other ordering of the same always block statements will also synthesize to the correct pipeline logic, but might not simulate correctly.

Register/Pipeline Modelling: Good Styles

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

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

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

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

Sequential Feedback Modelling

A Linear Feedback Shift-Register (LFSR) is a piece of sequential logic with a feedback loop.

```
module lfsrb1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 = 1'b1;
      q2 = 1'b1;
      q1 = 1'b1;
    end
    else begin
      q3 = q2;
      q2 = n1;
      q1 = q3;
    end
endmodule
```

This Style Won't Work
(using Blocking Assignments)

```
module lfsrb2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} = 3'b111;
    else        {q3,q2,q1} = {q2,(q1^q3),q3};
endmodule
```

Advanced Digital Design

LFSR Modelling (This works)

```
module lfsrn1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 <= 1'b1;
      q2 <= 1'b1;
      q1 <= 1'b1;
    end
    else begin
      q3 <= q2;
      q2 <= n1;
      q1 <= q3;
    end
endmodule
```

```
module lfsrn2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} <= 3'b111;
    else        {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

The y-output will reflect the old values of tmp1 and tmp2, not the values calculated in the current pass of the always block.

```
module ao4 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

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

```
module ao5 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

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

- **tmp1 and tmp2** have been added to the sensitivity list. The always block will self-trigger and update the y-outputs with the newly calculated tmp1 and tmp2 values.

- The y-output value will now be correct after taking two passes through the always block.

- Multiple passes through an always block equates to degraded simulation performance and should be avoided if a reasonable alternative exists.

Combinational Logic: Working

```
module ao2 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

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

Advanced Digital Design

Non-Blocking Assignments and $display

Myth: "Using the $display command with nonblocking assignments does not work"

Truth: Nonblocking assignments are updated after all $display commands.

```
module display_cmds;
  reg a;

  initial $monitor("\$monitor: a = %b", a);

  initial begin
    $strobe  ("\$strobe : a = %b", a);
    a = 0;
    a <= 1;
    $display ("\$display: a = %b", a);
    #1 $finish;
  end
endmodule
```

```
$display: a = 0
$monitor: a = 1
$strobe : a = 1
```

Advanced Digital Design

#0 Delay Assignments

Myth: "#0 forces an assignment to the end of a time step"

Truth: #0 forces an assignment to the "inactive events queue"

```
module nb_schedule1;
  reg a, b;

  initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;

       $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
    #0 $display ("%0dns: #0       : a=%b  b=%b", $stime, a, b);

    #1 $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
       $display ("%0dns: #0       : a=%b  b=%b", $stime, a, b);

    #1 $finish;
  end
endmodule
```

Advanced Digital Design

#0 Delay Assignments

- The #0-delay command was executed in the inactive events queue, before the nonblocking assign update events were executed.
- Use $strobe to display values that have been assigned using nonblocking assignments.

```
0ns: $display: a=0   b=1
0ns: #0      : a=0   b=1
0ns: $monitor: a=1   b=0
0ns: $strobe : a=1   b=0

1ns: $display: a=1   b=0
1ns: #0      : a=1   b=0
1ns: $monitor: a=1   b=0
1ns: $strobe : a=1   b=0
```

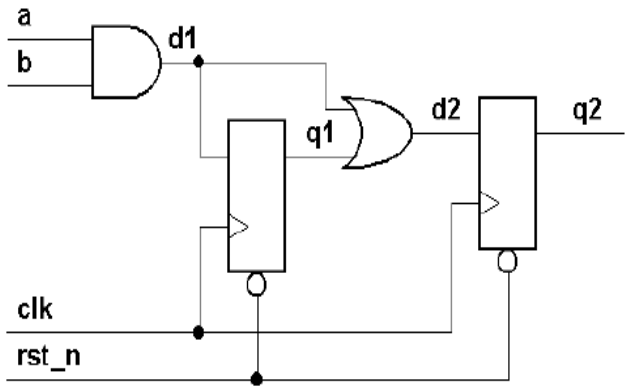
Multiple nonblocking assignments to the same variable

Myth: "Making multiple nonblocking assignments to the same variable in the same always block is undefined"

Truth: Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins!

Last nonblocking assignment wins

Simple sequential logic with one clock

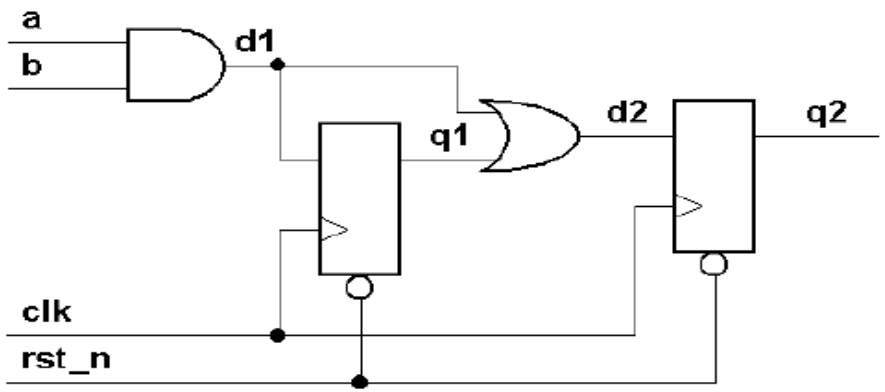


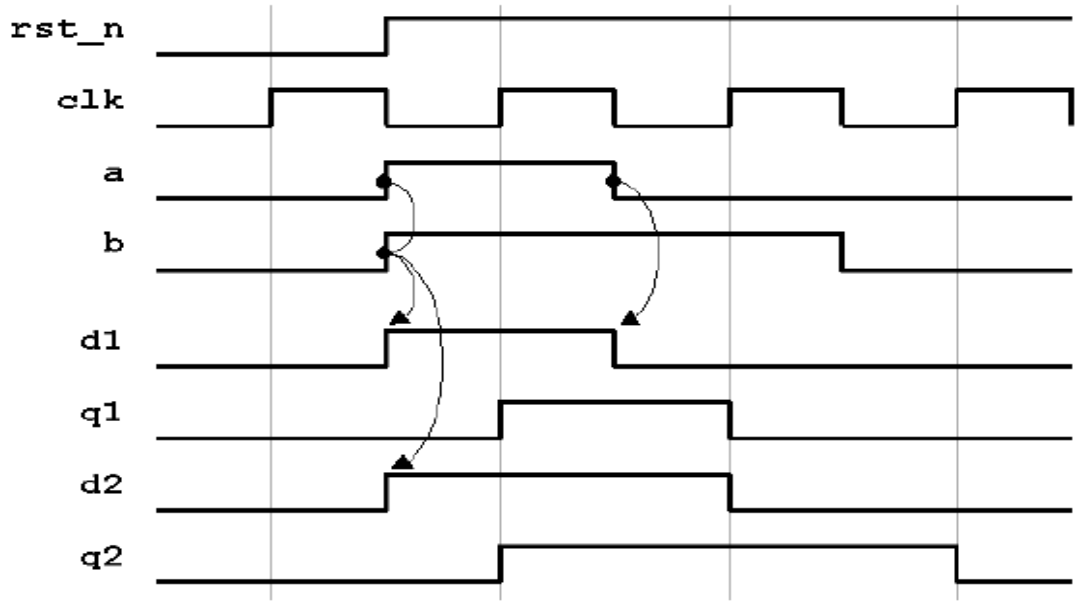

```
module sblk1 (
  output reg q2,
  input      a, b, clk, rst_n);
  reg        q1, d1, d2;

  always @(a or b or q1) begin
    d1 = a & b;
    d2 = d1 | q1;
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      q2 <= 0;
      q1 <= 0;
    end
    else begin
      q2 <= d2;
      q1 <= d1;
    end
endmodule
```

Advanced Digital Design

Testbench

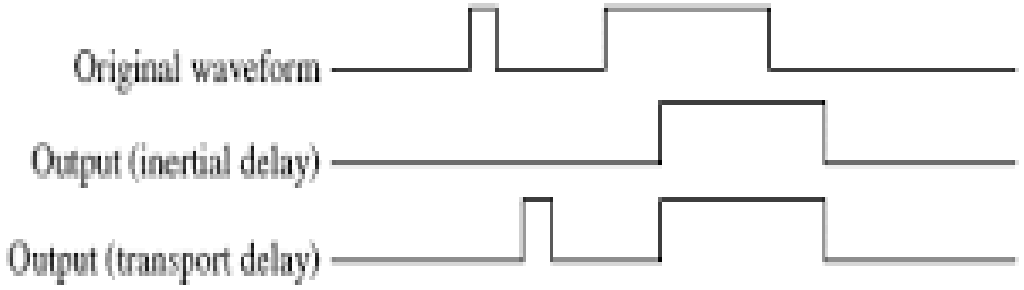
```
module tb;
  reg a, b, clk, rst_n;

  initial begin // clock oscillator
    clk = 0;
    forever #10 clk = ~clk;
  end

  sblk1 u1 (.q2(q2), .a(a), .b(b), .clk(clk), .rst_n(rst_n));

  initial begin // stimulus
    a = 0; b = 0;
    rst_n <= 0;
    @(posedge clk);
    @(negedge clk) rst_n = 1;
    a = 1; b = 1;
    @(negedge clk) a = 0;
    @(negedge clk) b = 0;
    @(negedge clk) $finish;
  end
endmodule
```



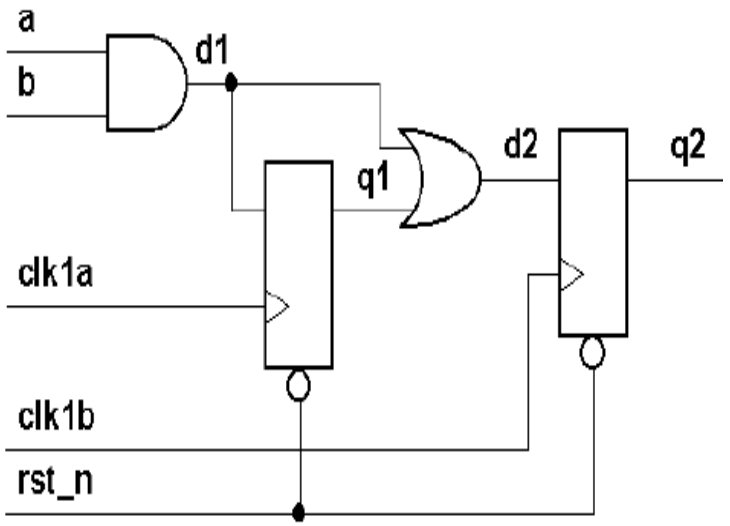


Advanced Digital Design
Inertial and Transport Delays

- Inertial delay models are simulation delay models that filter pulses that are shorter than the propagation delay of Verilog gate primitives or continuous assignments.
- Inertial delays swallow glitches!
- Inertial delays are very easy for a simulator to implement because the simulator only keeps track of what the next assignment value is going to be and when it will occur.
- By default, both Verilog and VHDL simulate using inertial delays.
- Transport delay models are simulation delay models that pass all pulses, including pulses that are shorter than the propagation delay of corresponding Verilog procedural assignments. Transport delays pass glitches, delayed in time.
- The VHDL language models transport delays by adding the key word "transport" to assignments.
- Verilog can model RTL transport delays by adding explicit delays to the right-hand-side (RHS) of a nonblocking assignment.



Advanced Digital Design
Multiple common clocks and race conditions



- In this case posedge clk1a and posedge clk1b occur at the same simulation time.
- Can there be a race condition caused by these two clock signals being generated from different blocks of RTL code?
- If the sequential logic driven by these two clocks is properly coded with no-delay nonblocking assignments, the answer is no.

- For this example, all posedge clk1a nonblocking assignments will be scheduled to be updated in the nonblocking assignments update queue.
- Then all of the posedge clk1b nonblocking assignments will be scheduled to be updated in the nonblocking assignments update queue before the clk1a updates have been activated in the same time step.
- This insures that all registered logic will be correctly pipelined between the no-skew clock domains before the combinational logic is updated.

References

- FPGA Prototyping using Verilog Examples, Pong P Chu
- Verilog Coding Guidelines “Sunburst Design Papers”

THANK YOU

Sudeendra kumar K

Department of Electronics and Communication Engineering

sudeendrakumark@pes.edu

Advanced Digital Design

Dr. Sudeendra kumar K

Department of Electronics and Communication Engineering

ADVANCED DIGITAL DESIGN

Lecture-IV: Verilog Recap, FSM, FSM + D

Sudeendra kumar K
Department of Electronics and Communication Engineering

Advanced Digital Design

Contents

- Verilog Recap
- Circular FIFO
- De-bouncing Circuit
- Code Coverage

PES UNIVERSITY ONLINE

Difference between Continuous and Procedural Statements

Continuous assignment	Procedural assignment
Assigns values primarily to **nets**	Assigns values primarily to **reg** variables
Variables and nets **continuously** drive values onto ports	Results of calculations involving variables and nets can be **stored** into variables
Used to infer combinatorial logic	Used to infer both storage elements like Flip-flops and latches and also combinatorial logic
Assignment occurs whenever the value on the RHS of the expression changes as a continuous process	The value of the previous assignment is held until another assignment is made to the variable
Occurs in assignments to ***wire, port, and net*** type	Occurs in constructs like ***always, initial, task, function***

For example,
`wire out1 = in1 & in2;`
or
`assign out1 = in1 & in2;`
	For example,
	`always @(posedge clk)`
	`  reg1 <= in1;`
	`always @(a or b or s)`
	`  y = (s == 1) ? a : b;`

Advanced Digital Design

Difference between initial and always blocks

initial	always
Assignments in an initial block begin to execute from time 0 in simulation, and proceed in the specified sequence.	Assignments in an always block also begin from time 0, and repeat forever as a function of the changes on the blocks sensitivity list
Execution of statements in an initial begin-end block stops when the end of the block is reached, i.e., executed only once during simulation	Execution continuously repeats from the begin to the end of the process unless held by a ***wait*** construct throughout the simulation session
Non-synthesizable construct	Synthesizable construct

For example,
`reg [1:0] out1, out2;`
`initial begin`
`  out1 = 2'b10;`
`  #5 out2 = 2'b01;`
`end`
	For example,
	`reg [1:0] out1, out2;`
	`always @(posedge clk)`
	`begin`
	`  out1 <= in1;`
	`  out2 <= out1 & in2;`
	`end`

Difference between blocking and non-blocking 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

Difference between task and function

task	*function*
Can contain time control statements like @(*posedge* .), delay operator (#)	Executes in zero simulation time
Can call any number of function's or tasks within itself	Can call any number of *function*'s within itself
Cannot return any value when called; instead the *task* can have output arguments	Returns a single value when called. In SytemVerilog the return value can be optionally voided
For example, `gt_result` is an output of a *task* to calculate the result of the greater of two input arguments `arg1` and `arg2` `greater_val(arg1, arg2, gt_result);`	For example, `gt_result` is assigned the return of a *function* call to calculate the result of the greater of two input arguments `arg1` and `arg2` `gt_result = greater_val(arg1, arg2)`

Difference between task and module

task	module
A *task* cannot instantiate a *module* within it	Fundamentally *task*s can be called only within a module
The logic of a task cannot be identified as a block to be moved around during floorplanning. It is a part of the sea-of-gates	A module instantiation has a hierarchy that can be fully identified and placed as a block during floorplanning
Prior to the advent of Verilog 2001, tasks in Verilog are not re-entrant. Therefore, if a task uses internal local variables, it could be multiply invoked in overlapping time-domains	By definition, modules can be instantiated multiple times. Each instance will carry its own context, including all of its internal registers and other variables. Therefore, processes within these instances are inherently concurrent among themselves and also across instances

Advanced Digital Design

Difference between latch and flip-flop

Latch	Flip-flop
Area of a latch is typically less than that of a Flip-flop	Area of a Flip-flop for same features is more than that of a latch
Consumes lesser power, due to lesser switching activity and lesser area	Power consumption is typically higher, due to the area and free running clock. Additional controls required to save power
Facilitates time borrowing or cycle stealing; Helps increase pipeline depth with lesser area.; Even if the path is longer than a clock cycle for a latch based pipeline, it is okay as long as it meets the next latch setup margin	Since the clock boundaries are rigid, the facility of time borrowing or cycle stealing doesn't exist with FFs. A negative slack cannot be propagated to the timing of the next stage in pipeline and hence must execute within a clock period
In multiple clock schemes, the clock edges must not be overlapping; It makes the logic design, vector generation for verification and clock tree synthesis difficult	Clock tree synthesis is less tedious in FF based designs. Since the stimulus needs to be stable before the setup time of the clock, the vector generation is relatively easier
With time borrowing* and cycle stealing, the operating frequency is higher than the slowest logic path	Due to rigid timing boundaries, the slowest path pretty much decides the operating frequency
Makes time budgeting and characterizing the interfaces tedious	The time budgeting is clearer and characterizing the interface is easier

Advanced Digital Design

Difference between Async Reset and Sync Reset

Asynchronous reset	Synchronous reset
Reset signal is not a part of the data path, that is, not a part of logic for D input of the FF	Reset signal is part of the data path, that is, the D input of the FF
Effect of reset can happen anytime asynchronously	Effect of reset will happen only on the active edge of a clock
Doesn't depend upon the presence of an active clock signal	Depends upon the presence of the clock signal for the reset to happen
Asynchronous event is an overload, compared to synchronous reset in the cycle based simulators	Works well when using cycle based simulators
Not recommended for internally generated resets, due to glitches	For internally generated resets, synchronous approach is the best mechanism
Reset input from external sources can be prone to glitches, the final reset signal needs to be synchronized before applying it to all storage elements	Not prone to glitches from internal or external sources
Asynchronous reset input still needs the double FF synchronization to avoid race condition during de-assertion	The additional synchronization circuitry is not required as it is a part of the default synchronous logic requirement
Needs to meet only the minimum reset pulse width required for the FF	Reset pulse width has to be long enough to be sampled on an active clock edge

```
always (posedge clk) //Sync
If (reset)
Q <= 0;
Else
Q <=d;
```

```
always (posedge clk, reset)
If (reset)      //Aysnc
Q <= 0;
Else
Q < = d;
```

Advanced Digital Design
Verilog Coding Guidelines: conditional and if-else

Conditional "*?:*" operator	***if-else***
Typically used in procedural or continuous assignments	Typically used within ***initial*** or ***always*** blocks

A TRUE and a FALSE expression is always required to be fully specified, that is, in the example:
`assign a=(b == 0) ? c:d;`
both expression `c` and `d` are required
	The ***else*** portion is optional in the ***if--else*** statement, in the following example:
	`if (en)`
	`  q = d;`
	`// else not necessarily`
	`// required`

Expressions to the left and right of the colon ":" can only be a single expression, that is, not a block of expressions within begin-end. For example, the following is a syntax error:
`a = (b==0)? c : begin d; e;`
`end; // wrong`
	The expressions within the ***if-else*** can be block code enclosed within a begin-end. For example,
	`if (en) begin`
	`… // many expressions`
	`end else begin`
	`… // many expressions`
	`end`

While the "***?:***" operator is useful in specifying simple expressions, readability is an issue when it is deeply nested	***if-else*** is visually more readable code in all expressions, especially when it is nested

Advanced Digital Design
Verilog Coding Guidelines

full_case	parallel_case
Indicates that the case statement has been fully specified, and all unspecified case expressions can be optimized away	Indicates that all case items need to be evaluated in parallel and not infer any priority encoding logic
All control paths are specified explicitly or by using a default	There is no overlap among the case items
Helps avoid latches as all cases are fully specified	Results in multiplexor logic as a parallel logic
Although not recommended, the default clause can be avoided, and still not infer a latch	A priority encoder is NOT synthesized, as each path is unique

An example of a case statement that is full (and parallel) is shown below:
`reg var1 [1:0];`
`always @(a or b or c) begin`
`  case (var1)`
`  2'b00 : out1 = a;`
`  2'b01 : out1 = b;`
`  2'b10 : out1 = c;`
`  2'b11 : out1 = a&b;`
`  endcase`
`end`
	An example of a case statement that is parallel (not full) is shown as follows:
	`reg var1 [2:0];`
	`always @(a or b or c) begin`
	`  case (var1)`
	`  3'b000 : out1 = a;`
	`  3'b001 : out1 = b;`
	`  3'b010 : out1 = c;`
	`// rest of the cases are //`
	`not defined`
	`  endcase`
	`end`

Reg var1 [1:0]
Always …..
Case (var1)
2'b00: out 1 = a;
2'b01: out 1 = b;
Default : out1 = a & b & c;
// This is full-case, but not parallel case.

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design

Verilog Coding Guidelines: Example

Use of == or != operators	Use of === or !== operators
These operators can be used in a synthesizable code	Cannot be used in a synthesizable code
If either of the operands have x or z value, the result is unknown	The operands will be compared, even if they have x and z values in the bits
If any of the operators is x or z, the logical result of comparison is **always** FALSE	The x and z bits will be used in comparison, and the logical result will be a TRUE or FALSE, based on actual comparison
Since the operands contain x and z, the result will be an x. Hence, the comparison can be non-deterministic	Since x and z are also used in comparison, the result of comparison will be Boolean 1 or 0. Hence the comparison can be deterministic

Example of using (== or !=) operators
`if (a == b)`
`  out1 = a & b;`
`else`
`  out1 = a | b;`
 	
If either a or b becomes x or z, the else clause will be executed and out1 will be driven by OR gate
	Example of using (=== or !==) operators
	`if (a === b)`
	`  out1 = a & b;`
	`else`
	`  out1 = a | b;`
	
	If a and b are identical, even if they becomes x or z, the if clause will be executed and out1 will be driven by AND gate

Asynchronous state machines	Synchronous state machines
State transitions depend upon the order in which the input signals change	State transitions are controlled by a clock signal
State transitions happen after propagation delay of the state line	State transitions happen at intervals of the clock period
Delay lines act as memory elements	Edge triggered FFs or level sensitive latches act as storage elements
Output response time is not predictable	Output response time is predictable; will happen at clock period intervals

Verilog Coding Guidelines: Mealy and Moore FSM

Mealy machine	Moore machine
Outputs are a function of current state and input signals	Outputs are a function of current state only
Output can change between changes between state	Outputs change only when the current state changes
Output can changes any number of times during a clock cycle, which may result in glitches on the outputs	Output is delayed by one clock cycle, but is stable
More output combinations are possible as the outputs are a function of inputs too	Since the outputs are a function only of the current state, the numbers of output combinations are fewer with the Mealy machine
If the inputs are not registered, the combinatorial paths could potentially be larger than Moore machine; Hence, a relatively lower frequency is expected compared to a Moore machine	Can expect higher frequency compared to Mealy machine, as the combinatorial paths are typically shorter, and no input paths are involved

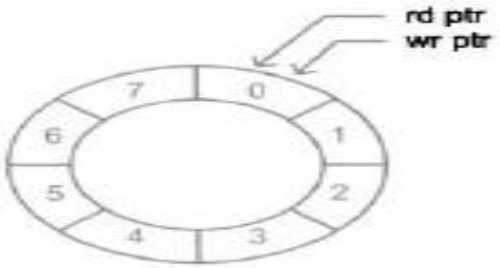
Advanced Digital Design

Circular FIFO Design

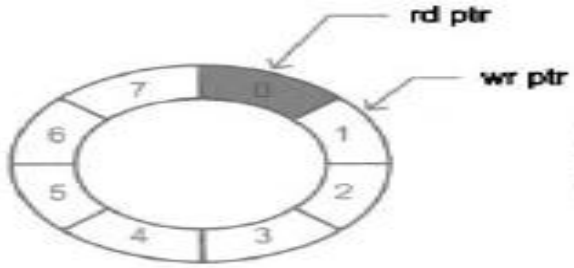
- A FIFO (first-in-first-out) buffer is an "elastic" storage between two subsystems.
- It has two control signals, **wr** and **rd**, for write and read operations. When **wr** is asserted, the input data is written into the buffer.
- The head of the FIFO buffer is normally always available and thus can be read at any time.
- The **rd** signal actually acts like a "remove“ signal. When it is asserted, the first item (i.e., head) of the FIFO buffer is removed and the next item becomes available.
- The write pointer points to the head of the queue, and the read pointer points to the tail of the queue.
- The pointer advances one position for each write or read operation.
- A FIFO buffer usually contains two status signals, full and empty, to indicate that the FIFO is full (i.e., cannot be written) and empty (i.e., cannot be read), respectively.

Advanced Digital Design

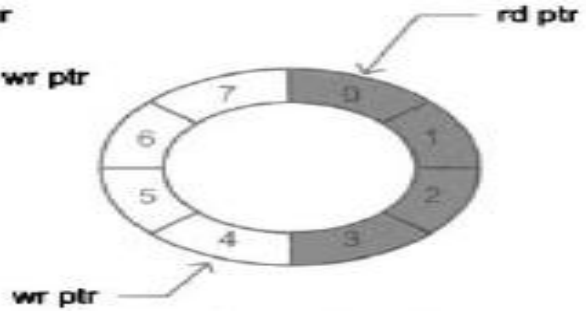
Circular FIFO Design



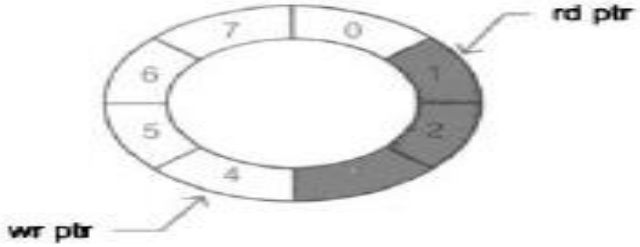
(a). initial (empty)



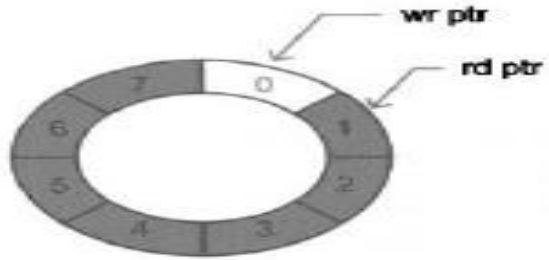
(b). after a write



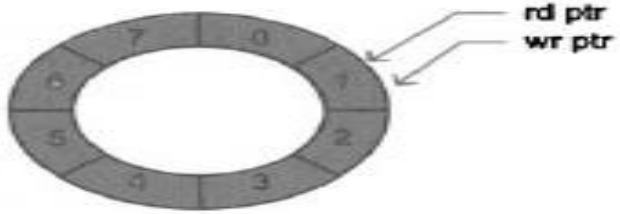
(c). 3 more writes



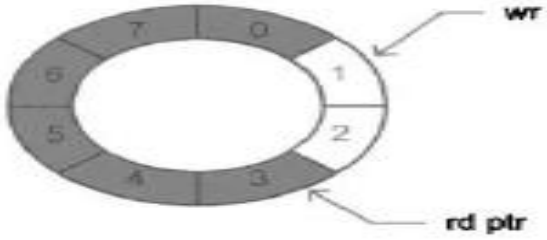
(d). after a read



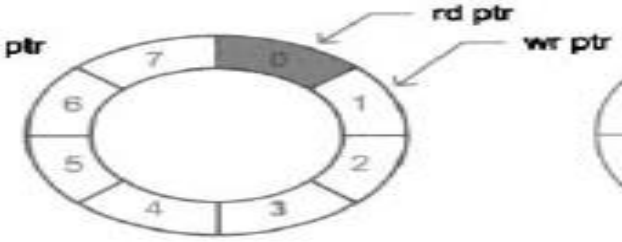
(e). 4 more writes



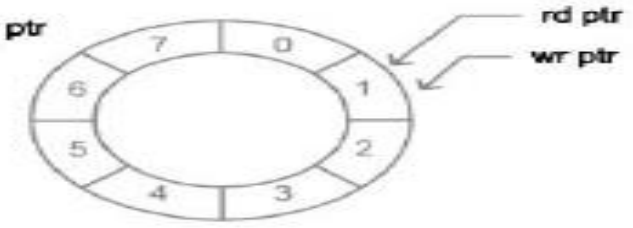
(f). 1 more write (full)



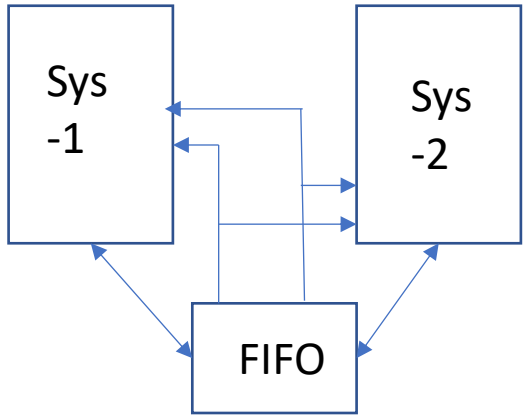
(g). 2 reads



(h). 5 more reads



(i). 1 more read (empty)



Circular FIFO Design

- A FIFO (first-in-first-out) buffer is an "elastic" storage between two subsystems.
- It has two control signals, **wr** and **rd,** for write and read operations. When **wr** is asserted, the input data is written into the buffer.
- The head of the FIFO buffer is normally always available and thus can be read at any time.
- The **rd** signal actually acts like a "remove“ signal. When it is asserted, the first item (i.e., head) of the FIFO buffer is removed and the next item becomes available.

Advanced Digital Design
Circular FIFO

```
module fifo
   #(
    parameter B=8, // number of bits in a word
              W=4  // number of address bits
   )
   (
    input wire clk, reset,
    input wire rd, wr,
    input wire [B-1:0] w_data,
    output wire empty, full,
    output wire [B-1:0] r_data
   );

   //signal declaration
   reg [B-1:0] array_reg [2**W-1:0];  // register array
   reg [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
   reg [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
   reg full_reg, empty_reg, full_next, empty_next;
   wire wr_en;

   // body
   // register file write operation
   always @(posedge clk)
      if (wr_en)
         array_reg[w_ptr_reg] <= w_data;
   // register file read operation
   assign r_data = array_reg[r_ptr_reg];
   // write enabled only when FIFO is not full
   assign wr_en = wr & ~full_reg;
```

Courtesy: Pong P Chu

Advanced Digital Design
Circular FIFO

```
// fifo control logic
// register for read and write pointers
always @(posedge clk, posedge reset)
   if (reset)
      begin
         w_ptr_reg <= 0;
         r_ptr_reg <= 0;
         full_reg <= 1'b0;
         empty_reg <= 1'b1;
      end
   else
      begin
         w_ptr_reg <= w_ptr_next;
         r_ptr_reg <= r_ptr_next;
         full_reg <= full_next;
         empty_reg <= empty_next;
      end
```

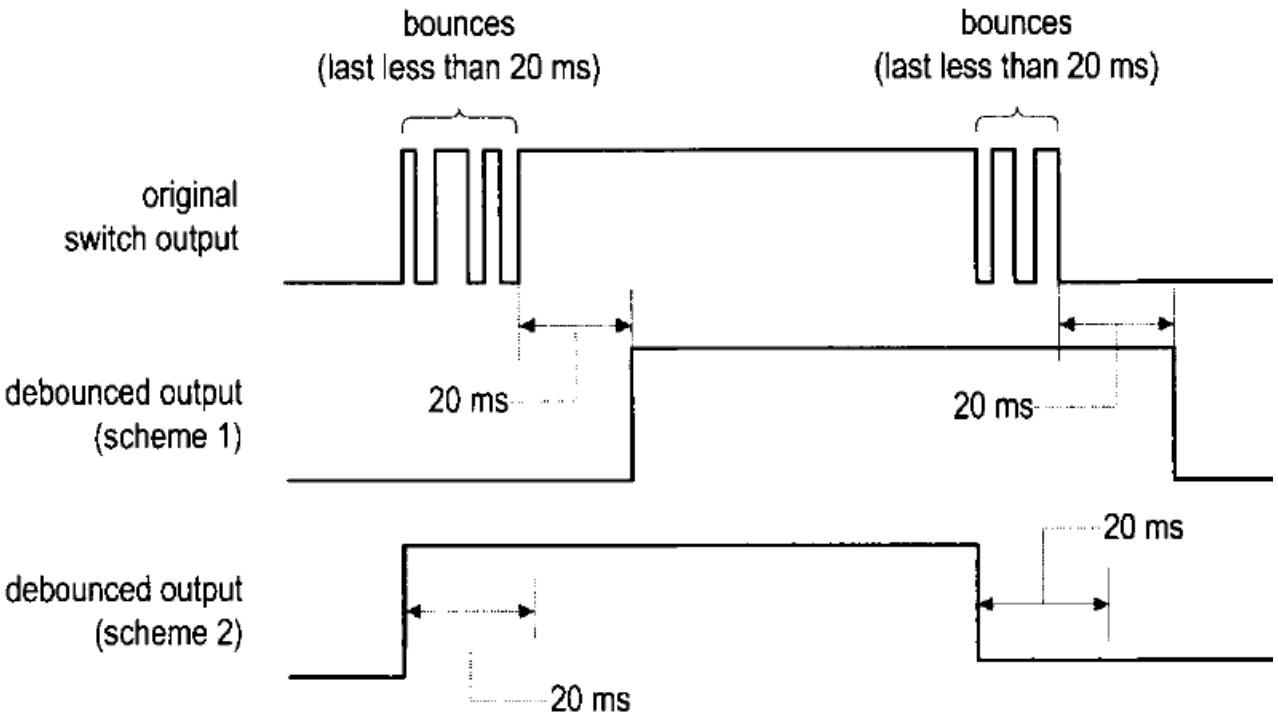
```
// next-state logic for read and write pointers
always @*
begin
   // successive pointer values
   w_ptr_succ = w_ptr_reg + 1;
   r_ptr_succ = r_ptr_reg + 1;
   // default: keep old values
   w_ptr_next = w_ptr_reg;
   r_ptr_next = r_ptr_reg;
   full_next = full_reg;
   empty_next = empty_reg;
   case ({wr, rd})
      // 2'b00:  no op
      2'b01: // read
         if (~empty_reg) // not empty
            begin
               r_ptr_next = r_ptr_succ;
               full_next = 1'b0;
               if (r_ptr_succ==w_ptr_reg)
                  empty_next = 1'b1;
            end
      2'b10: // write
         if (~full_reg) // not full
```

Courtesy: Pong P Chu

```
                begin
                    w_ptr_next = w_ptr_succ;
                    empty_next = 1'b0;
                    if (w_ptr_succ==r_ptr_reg)
                        full_next = 1'b1;
                end
            2'b11: // write and read
                begin
                    w_ptr_next = w_ptr_succ;
                    r_ptr_next = r_ptr_succ;
                end
        endcase
    end

    // output
    assign full = full_reg;
    assign empty = empty_reg;

endmodule
```

Courtesy: Pong P Chu

Advanced Digital Design
De-bouncing

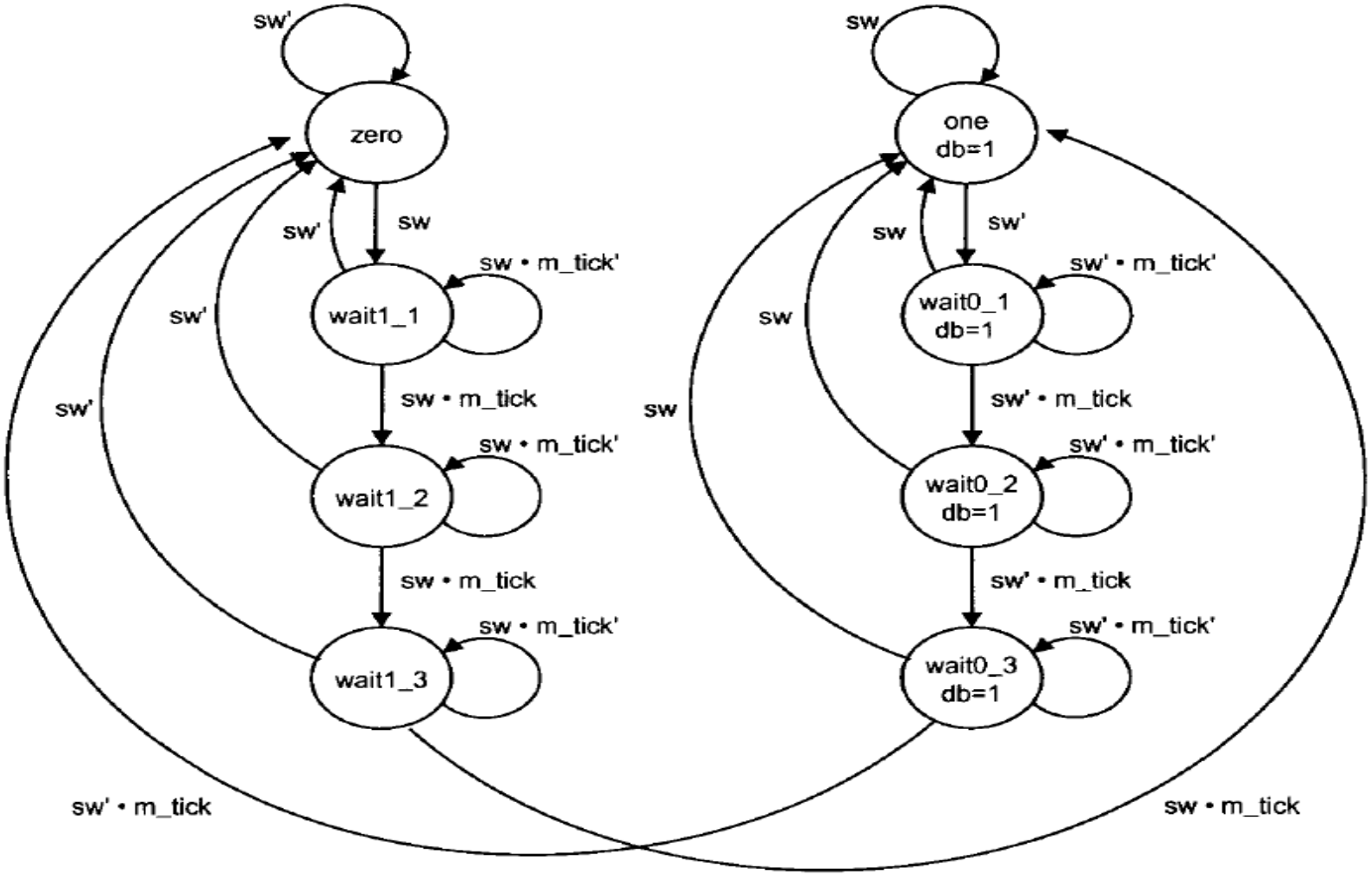


- The slide and pushbutton switches on the prototyping board are mechanical devices. When pressed, the switch may bounce back and forth a few times before settling down. The bounces lead to glitches in the signal, as shown at the top of Figure.
- The bounces usually settle within 20 ms. The purpose of a debouncing circuit is to filter out the glitches associated with switch transitions.

Courtesy: Pong P Chu

Advanced Digital Design

Debouncing Circuit



Courtesy: Pong P Chu

Advanced Digital Design
De-bouncing Circuit

```
module db_fsm
    (
     input wire clk, reset,
     input wire sw,
     output reg db
    );

    // symbolic state declaration
    localparam  [2:0]
                zero    = 3'b000,
                wait1_1 = 3'b001,
                wait1_2 = 3'b010,
                wait1_3 = 3'b011,
                one     = 3'b100,
                wait0_1 = 3'b101,
                wait0_2 = 3'b110,
                wait0_3 = 3'b111;

    // number of counter bits (2^N * 20ns = 10ms tick)
    localparam N =19;
```

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design
De-bouncing Circuit

```
// signal declaration
reg [N-1:0] q_reg;
wire [N-1:0] q_next;
wire m_tick;
reg [2:0] state_reg, state_next;

// body

//=============================================
// counter to generate 10 ms tick
//=============================================
always @(posedge clk)
   q_reg <= q_next;
// next-state logic
assign q_next = q_reg + 1;
// output tick
assign m_tick = (q_reg==0) ? 1'b1 : 1'b0;

//=============================================
```

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design
De-bouncing Circuit

```
//=================================
// debouncing FSM
//=================================
// state register
 always @(posedge clk, posedge reset)
    if (reset)
       state_reg <= zero;
    else
       state_reg <= state_next;
```

```
// next-state logic and output logic
always @*
begin
   state_next = state_reg;  // default state: the same
   db = 1'b0;               // default output: 0
   case (state_reg)
      zero:
         if (sw)
            state_next = wait1_1;
      wait1_1:
         if (~sw)
            state_next = zero;
         else
            if (m_tick)
               state_next = wait1_2;
      wait1_2:
         if (~sw)
            state_next = zero;
         else
            if (m_tick)
               state_next = wait1_3;
      wait1_3:
         if (~sw)
            state_next = zero;
```

Courtesy: FPGA Prototyping, Pong P Chu

Advanced Digital Design

De-bouncing Circuit

```
            else
                if (m_tick)
                    state_next = one;
        one:
            begin
               db = 1'b1;
               if (~sw)
                  state_next = wait0_1;
            end
        wait0_1:
            begin
                db = 1'b1;
                if (sw)
                    state_next = one;
                else
                   if (m_tick)
                      state_next = wait0_2;
            end
        wait0_2:
            begin
                db = 1'b1;
                if (sw)
                    state_next = one;
                else
                   if (m_tick)
                      state_next = wait0_3;
            end
        wait0_3:
            begin
                db = 1'b1;
                if (sw)
                    state_next = one;
                else
                   if (m_tick)
                      state_next = zero;
            end
        default: state_next = zero;
    endcase
end
```

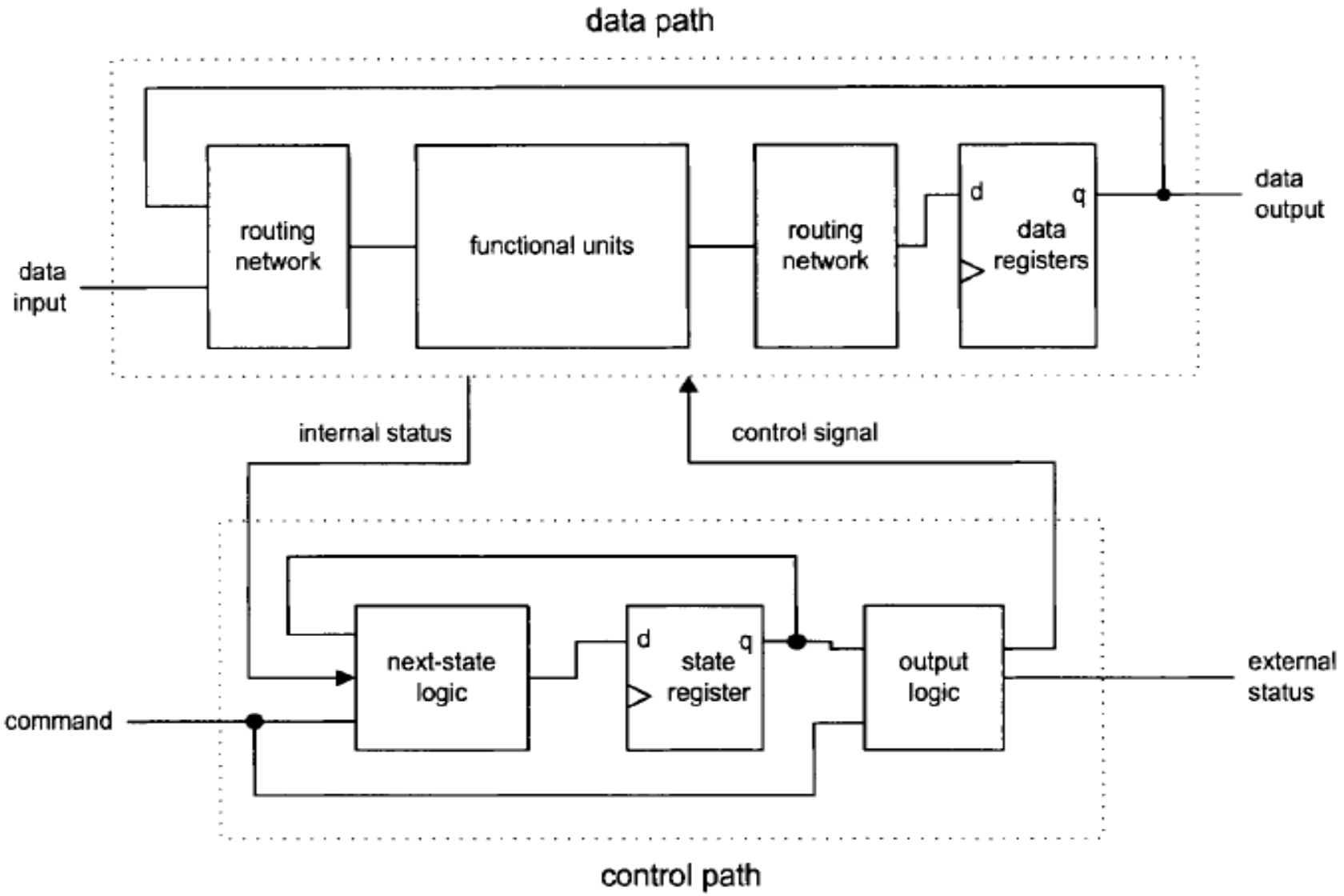
Advanced Digital Design

Finite State Machine + Datapath (FSMD)

- An FSMD (finite state machine with data path) combines an FSM and regular sequential circuits. The FSM, which is sometimes known as a control path, examines the external commands and status and generates control signals to specify operation of the regular sequential circuits, which are known collectively as a data path.
- The FSMD is used to implement systems described by RT(register transfer) methodology, in which the operations are specified as data manipulation and transfer among a collection of registers.

Advanced Digital Design

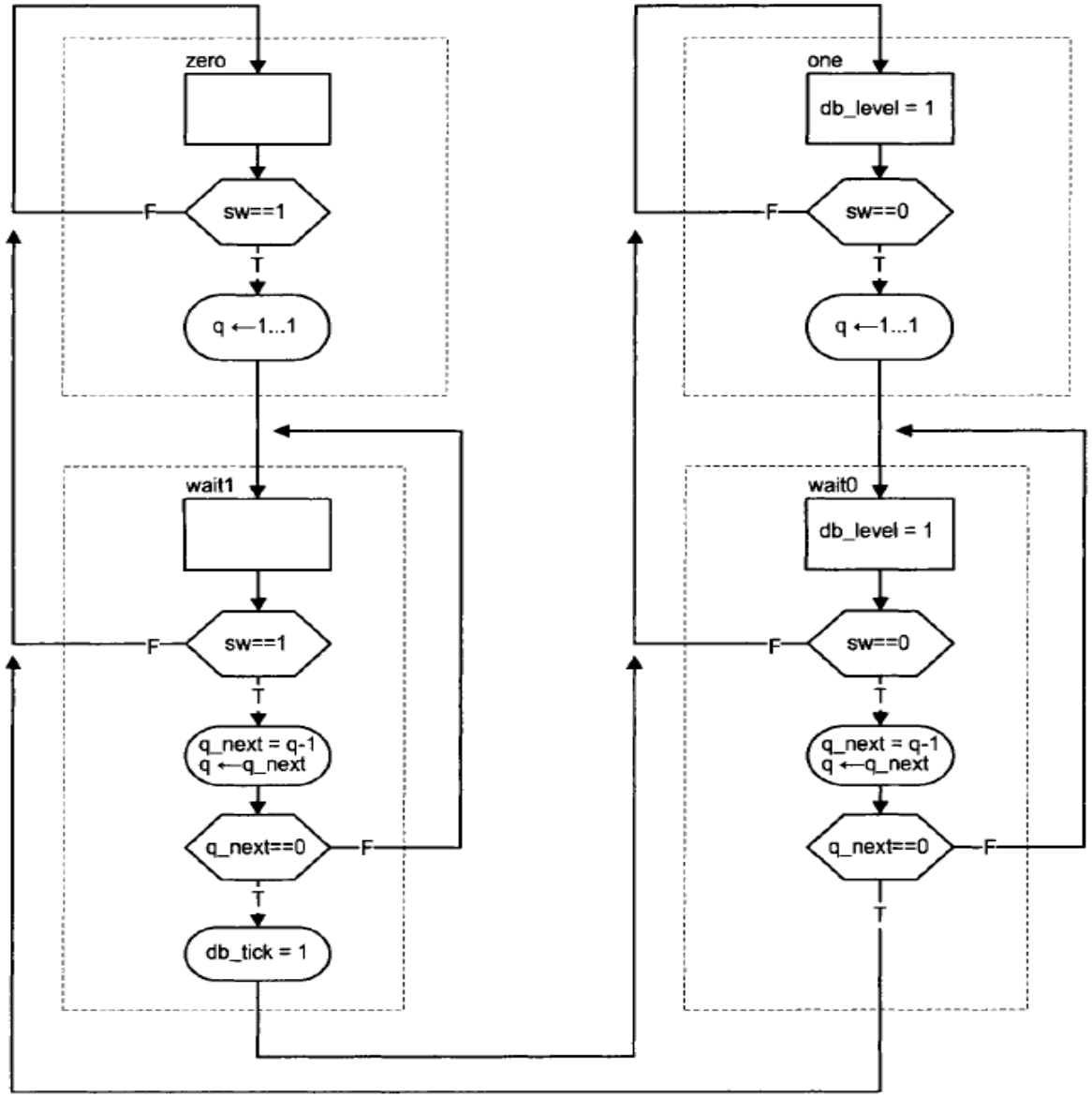
Finite State Machine + Datapath (FSMD)



- The data path performs the required RT operations. It consists of: -
  - Data registers: store the intermediate computation results
  - Functional units: perform the functions specified by the RT operations
  - Routing network: routes data between the storage registers and the functional units
- The data path follows the control signal to perform the desired RT operations and generates the internal status signal.
- The control path is an FSM. As a regular FSM, it contains a state register, next-state
- logic, and output logic. It uses the external command signal and the data path's status signal as the input and generates the control signal to control the data path operation.
- The FSM also generates the external status signal to indicate the status of the FSMD operation.
- FSMD consists of two types of sequential circuits, both circuits are controlled by the same clock, and thus the FSMD is a synchronous system.

Advanced Digital Design

Constructing De-bouncing Circuit in FSM + D



Advanced Digital Design
FSM + D for Debouncing Circuit

```
module debounce
    (
     input wire clk, reset,
     input wire sw,
     output reg db_level, db_tick
    );

    // symbolic state declaration
    localparam  [1:0]
                zero  = 2'b00,
                wait0 = 2'b01,
                one   = 2'b10,
                wait1 = 2'b11;

    // number of counter bits (2^N * 20ns = 40n
    localparam N=21;

    // signal declaration
    reg [N-1:0] q_reg, q_next;
    reg [1:0] state_reg, state_next;

    // body
    // fsmd state & data registers
     always @(posedge clk, posedge reset)
        if (reset)
           begin
              state_reg <= zero;
              q_reg <= 0;
           end
        else
           begin
              state_reg <= state_next;
              q_reg <= q_next;
           end
```

Advanced Digital Design

FSM + D for Debouncing Circuit

```
// next-state logic & data path functional units/routing
always @*
begin
   state_next = state_reg;    // default state: the same
   q_next = q_reg;            // default q: unchnaged
   db_tick = 1'b0;            // default output: 0
   case (state_reg)
      zero:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  state_next = wait1;
                  q_next = {N{1'b1}}; // load 1..1
               end
         end
      wait1:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  q_next = q_reg - 1;
                  if (q_next==0)
                     begin
                        state_next = one;
                        db_tick = 1'b1;
                     end
               end
            else // sw==0
               state_next = zero;
         end
      one:
         begin
            db_level = 1'b1;
            if (~sw)
               begin
                  state_next = wait0;
                  q_next = {N{1'b1}}; // load
               end
         end
      wait0:
         begin
            db_level = 1'b1;
            if (~sw)
               begin
                  q_next = q_reg - 1;
                  if (q_next==0)
                     state_next = zero;
               end
            else // sw==1
               state_next = one;

         end
      default: state_next = zero;
   endcase
end

endmodule
```

References

- FPGA Prototyping using Verilog Examples, Pong P Chu

THANK YOU

Sudeendra kumar K

Department of Electronics and Communication Engineering

sudeendrakumark@pes.edu

Advanced Digital Design

Dr. Sudeendra kumar K

Department of Electronics and Communication Engineering

ADVANCED DIGITAL DESIGN

Lecture-IV: Fibonacci Circuit & Code Coverage

Sudeendra kumar K
Department of Electronics and Communication Engineering

Advanced Digital Design

Contents

- Fibonacci Circuit
- Code Coverage

Fibonacci Number Circuit

- The Fibonacci numbers constitute a sequence defined as:

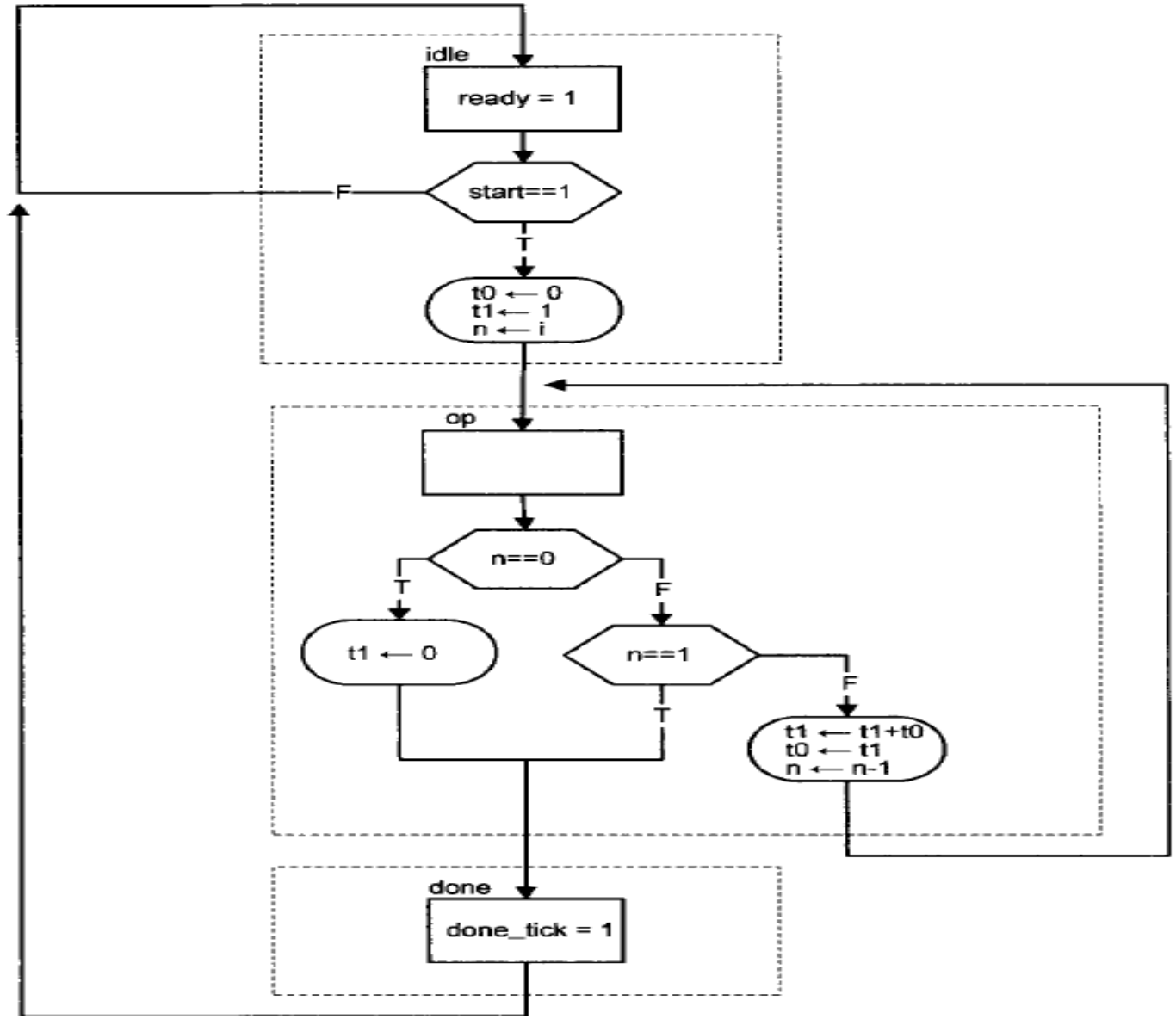
$$fib(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ fib(i-1) + fib(i-2) & \text{if } i > 1 \end{cases}$$

- One way to calculate **fib(i)** is to construct the function iteratively, from 0 to the desired **i**. This approach requires two temporary registers to store the two most recently calculated values [i.e., **fib(i - 1) and fib(i - 2)**] and one index register to keep track of the number of iterations.

- The ASMD chart is shown in Figure, in which **t1** and **t0** are temporary storage registers and **n** is the index register. In addition to the regular data input and output signals, **i** and **f** , we include a command signal, **start** , which signals the beginning of operation, and two status signals: **ready**, which indicates that the circuit is idle and ready to take new input, and **done-tick,** which is asserted for one clock cycle when the operation is completed.

Advanced Digital Design

Fibonacci Number Circuit



The ASMD chart has three states. The **idle** state indicates that the circuit is currently idle. When **start** is asserted, the FSMD moves to the **op** state and loads initial values to three registers. The **t0** and **t1** registers are loaded with 0 and 1, which represent **fib(0)** and **fib(1)**, respectively. The **n** register is loaded with **i**, the desired number of iterations.

Advanced Digital Design

Fibonacci Number Circuit

```
module fib
    (
     input wire clk, reset,
     input wire start,
     input wire [4:0] i,
     output reg ready, done_tick,
     output wire [19:0] f
    );

    // symbolic state declaration
    localparam [1:0]
        idle = 2'b00,
        op   = 2'b01,
        done = 2'b10;

    // signal declaration
    reg [1:0] state_reg, state_next;
    reg [19:0] t0_reg, t0_next, t1_reg, t1_next;
    reg [4:0] n_reg, n_next;

    // body
    // FSMD state & data registers
    always @(posedge clk, posedge reset)
        if (reset)
            begin
                state_reg <= idle;
                t0_reg <= 0;
                t1_reg <= 0;
                n_reg <= 0;
            end
        else
            begin
                state_reg <= state_next;
                t0_reg <= t0_next;
                t1_reg <= t1_next;
                n_reg <= n_next;
            end
    // FSMD next-state logic
    always @*
    begin
        state_next = state_reg;
        ready = 1'b0;
        done_tick = 1'b0;
        t0_next = t0_reg;
        t1_next = t1_reg;
        n_next = n_reg;
        case (state_reg)
            idle:
                begin
                    ready = 1'b1;
                    if (start)
                        begin
                            t0_next = 0;
                            t1_next = 20'd1;
                            n_next = i;
                            state_next = op;
                        end
                end
```

Fibonacci Number Circuit

```
        op:
            if (n_reg==0)
               begin
                  t1_next = 0;
                  state_next = done;
               end
            else if (n_reg==1)
               state_next = done;
            else
               begin
                  t1_next = t1_reg + t0_reg;
                  t0_next = t1_reg;
                  n_next = n_reg - 1;
               end
        done:
            begin
               done_tick = 1'b1;
               state_next = idle;
            end
        default: state_next = idle;
    endcase
end
// output
    assign f = t1_reg;

endmodule
```

Advanced Digital Design

Code Coverage

- Coverage is a generic term for measuring progress to complete design verification.
- The coverage tools gather information during a simulation and then postprocess it to produce a coverage report.
- The easiest way to measure verification progress is with code coverage.
- We are measuring how many lines of code have been executed (line coverage), which paths through the code and expressions have been executed (path coverage), which single bit variables have had the values 0 or I (toggle coverage), and which states and transitions in a state machine have been visited (FSM coverage).
- The end result is a measure of how much your tests exercise the design code. Untested design code could conceal a hardware bug, or may be just redundant code.

Functional Coverage

- The goal of verification is to ensure that a design behaves correctly in its real environment, be that an MP3 player, network router, or cell phone.
- The design specification details how the device should operate, whereas the verification plan lists how that functionality is to be stimulated, verified, and measured.
- Functional coverage is tied to the design intent and is sometimes called "specification coverage," while code coverage measures the design implementation.
- Consider what happens if a block of code is missing from the design. Code coverage cannot catch this mistake, but functional coverage can.

Advanced Digital Design

Types of Code Coverage

- Statement Coverage (line coverage)
- Branch Coverage
- Toggle Coverage
- FSM Coverage

Types of Code Coverage

- **Statement Coverage (line coverage): -** is a white box testing technique in which all the executable statements in the source code are executed at least once. It is used for calculation of the number of statements in source code which have been executed. The main purpose of Statement Coverage is to cover all the possible paths, lines and statements in source code.
- **Branch Coverage** is a white box testing method in which every outcome from a code module(statement or loop) is tested. The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once.
- **Condition Coverage** or **expression coverage** is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement. The goal of condition coverage is to check individual outcomes for each logical condition.

FSM Coverage

- **Finite state machine (FSM) coverage** is certainly the most complex type of code coverage method. This is because it works on the behavior of the design. In this coverage method, you need to look for how many time-specific states are visited, transited. It also checks how many sequences are included in a finite state machine.

Code Coverage and Functional Coverage

Code Coverage	Functional Coverage
Code coverage tells you how well the source code has been exercised by your test bench.	Functional coverage measures how well the functionality of the design has been covered by your test bench.
Never use a design specification	Use design specification

References

- FPGA Prototyping using Verilog Examples, Pong P Chu
- This slide and online class demo for Code coverage and Functional coverage.

THANK YOU

Sudeendra kumar K

Department of Electronics and Communication Engineering

sudeendrakumark@pes.edu

Advanced Digital Design

Dr. Sudeendra kumar K

Department of Electronics and Communication Engineering

ADVANCED DIGITAL DESIGN

Lecture-IV: Fibonacci Circuit & Code Coverage

Sudeendra kumar K
Department of Electronics and Communication Engineering

Advanced Digital Design

Contents

- Fibonacci Circuit
- Code Coverage

Fibonacci Number Circuit

- The Fibonacci numbers constitute a sequence defined as:

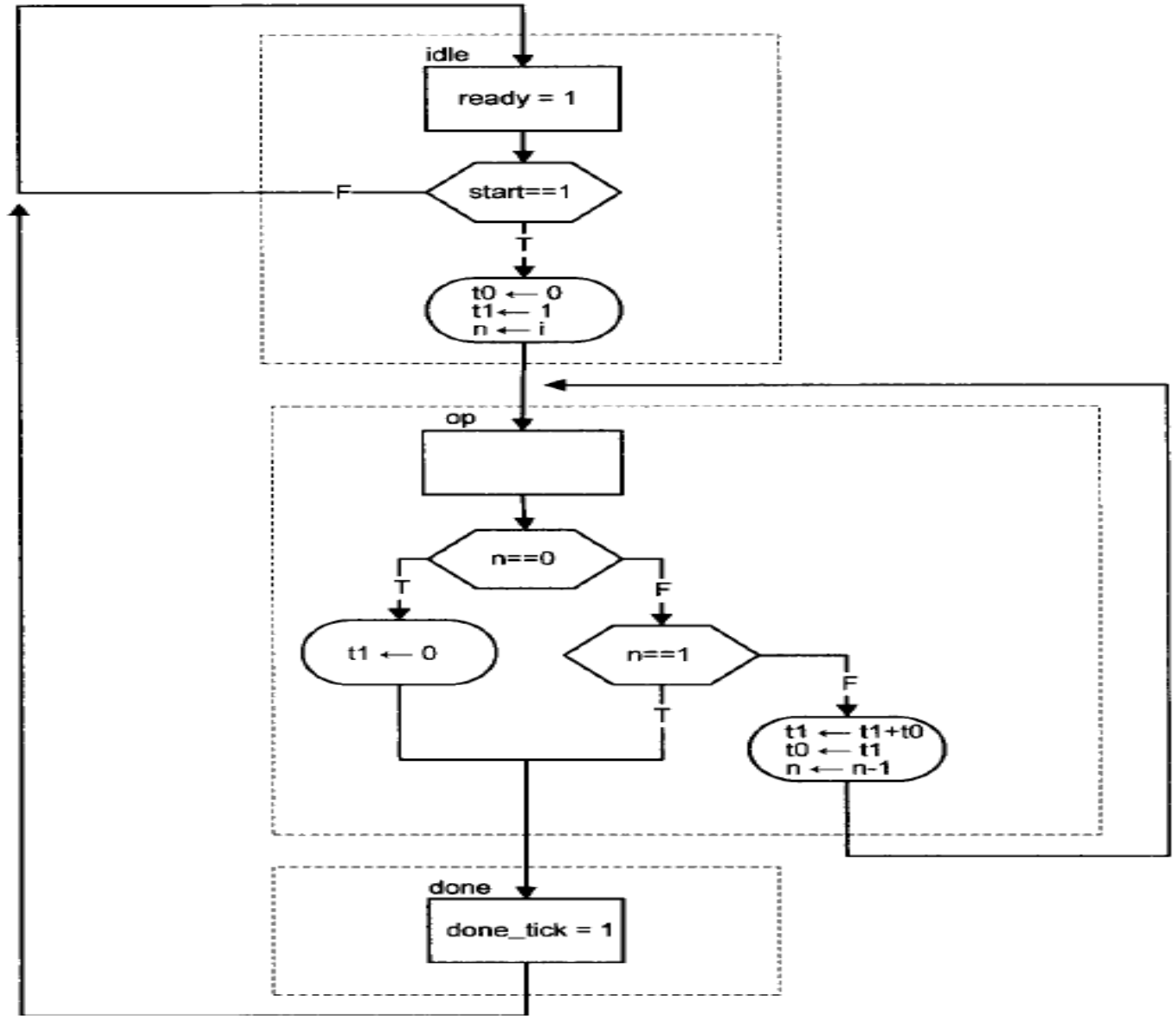
$$fib(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ fib(i-1) + fib(i-2) & \text{if } i > 1 \end{cases}$$

- One way to calculate **fib(i)** is to construct the function iteratively, from 0 to the desired **i**. This approach requires two temporary registers to store the two most recently calculated values [i.e., **fib(i - 1) and fib(i - 2)**] and one index register to keep track of the number of iterations.

- The ASMD chart is shown in Figure, in which **t1** and **t0** are temporary storage registers and **n** is the index register. In addition to the regular data input and output signals, **i** and **f** , we include a command signal, **start** , which signals the beginning of operation, and two status signals: **ready**, which indicates that the circuit is idle and ready to take new input, and **done-tick,** which is asserted for one clock cycle when the operation is completed.

Advanced Digital Design

Fibonacci Number Circuit



The ASMD chart has three states. The **idle** state indicates that the circuit is currently idle. When **start** is asserted, the FSMD moves to the **op** state and loads initial values to three registers. The **t0** and **t1** registers are loaded with 0 and 1, which represent **fib(0)** and **fib(1)**, respectively. The **n** register is loaded with **i**, the desired number of iterations.

Advanced Digital Design

Fibonacci Number Circuit

```
module fib
    (
     input wire clk, reset,
     input wire start,
     input wire [4:0] i,
     output reg ready, done_tick,
     output wire [19:0] f
    );

    // symbolic state declaration
    localparam [1:0]
        idle = 2'b00,
        op   = 2'b01,
        done = 2'b10;

    // signal declaration
    reg [1:0] state_reg, state_next;
    reg [19:0] t0_reg, t0_next, t1_reg, t1_next;
    reg [4:0] n_reg, n_next;

    // body
    // FSMD state & data registers
    always @(posedge clk, posedge reset)
        if (reset)
            begin
                state_reg <= idle;
                t0_reg <= 0;
                t1_reg <= 0;
                n_reg <= 0;
            end
        else
            begin
                state_reg <= state_next;
                t0_reg <= t0_next;
                t1_reg <= t1_next;
                n_reg <= n_next;
            end
    // FSMD next-state logic
    always @*
    begin
        state_next = state_reg;
        ready = 1'b0;
        done_tick = 1'b0;
        t0_next = t0_reg;
        t1_next = t1_reg;
        n_next = n_reg;
        case (state_reg)
            idle:
                begin
                    ready = 1'b1;
                    if (start)
                        begin
                            t0_next = 0;
                            t1_next = 20'd1;
                            n_next = i;
                            state_next = op;
                        end
                end
```

Fibonacci Number Circuit

```
        op:
            if (n_reg==0)
               begin
                  t1_next = 0;
                  state_next = done;
               end
            else if (n_reg==1)
               state_next = done;
            else
               begin
                  t1_next = t1_reg + t0_reg;
                  t0_next = t1_reg;
                  n_next = n_reg - 1;
               end
        done:
            begin
               done_tick = 1'b1;
               state_next = idle;
            end
        default: state_next = idle;
    endcase
end
// output
    assign f = t1_reg;

endmodule
```

Advanced Digital Design

Code Coverage

- Coverage is a generic term for measuring progress to complete design verification.
- The coverage tools gather information during a simulation and then postprocess it to produce a coverage report.
- The easiest way to measure verification progress is with code coverage.
- We are measuring how many lines of code have been executed (line coverage), which paths through the code and expressions have been executed (path coverage), which single bit variables have had the values 0 or I (toggle coverage), and which states and transitions in a state machine have been visited (FSM coverage).
- The end result is a measure of how much your tests exercise the design code. Untested design code could conceal a hardware bug, or may be just redundant code.

Functional Coverage

- The goal of verification is to ensure that a design behaves correctly in its real environment, be that an MP3 player, network router, or cell phone.
- The design specification details how the device should operate, whereas the verification plan lists how that functionality is to be stimulated, verified, and measured.
- Functional coverage is tied to the design intent and is sometimes called "specification coverage," while code coverage measures the design implementation.
- Consider what happens if a block of code is missing from the design. Code coverage cannot catch this mistake, but functional coverage can.

Advanced Digital Design

Types of Code Coverage

- Statement Coverage (line coverage)
- Branch Coverage
- Toggle Coverage
- FSM Coverage

Types of Code Coverage

- **Statement Coverage (line coverage): -** is a white box testing technique in which all the executable statements in the source code are executed at least once. It is used for calculation of the number of statements in source code which have been executed. The main purpose of Statement Coverage is to cover all the possible paths, lines and statements in source code.
- **Branch Coverage** is a white box testing method in which every outcome from a code module(statement or loop) is tested. The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once.
- **Condition Coverage** or **expression coverage** is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement. The goal of condition coverage is to check individual outcomes for each logical condition.

FSM Coverage

- **Finite state machine (FSM) coverage** is certainly the most complex type of code coverage method. This is because it works on the behavior of the design. In this coverage method, you need to look for how many time-specific states are visited, transited. It also checks how many sequences are included in a finite state machine.

Code Coverage and Functional Coverage

Code Coverage	Functional Coverage
Code coverage tells you how well the source code has been exercised by your test bench.	Functional coverage measures how well the functionality of the design has been covered by your test bench.
Never use a design specification	Use design specification

References

- FPGA Prototyping using Verilog Examples, Pong P Chu
- This slide and online class demo for Code coverage and Functional coverage.

THANK YOU

Sudeendra kumar K

Department of Electronics and Communication Engineering

sudeendrakumark@pes.edu

Advanced Digital Design

Dr. Sudeendra kumar K

Department of Electronics and Communication Engineering

ADVANCED DIGITAL DESIGN

Lecture-1: Timing Properties of Flip-flop

Unit - 2

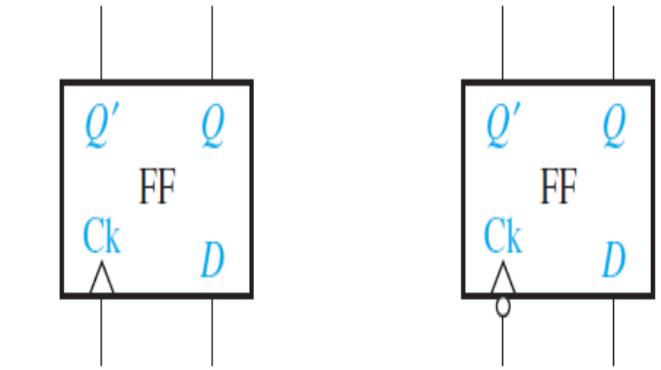
Sudeendra kumar K
Department of Electronics and Communication Engineering

Advanced Digital Design

Edge Triggered Flipflop

- The term active edge refers to the clock edge (rising or falling) that triggers the flip-flop state change.

- The state of a D flip-flop after the active clock edge (Q+) is equal to the input (D) before the active edge.
- For example, if D = 1 before the clock pulse, Q = 1 after the active edge, regardless of the previous value of Q.
- Therefore, the characteristic equation is Q+ = D. If D changes at most once following each clock pulse, the output of the flip-flop is the same as the D input, except that the output changes are delayed until after the active edge of the clock pulse



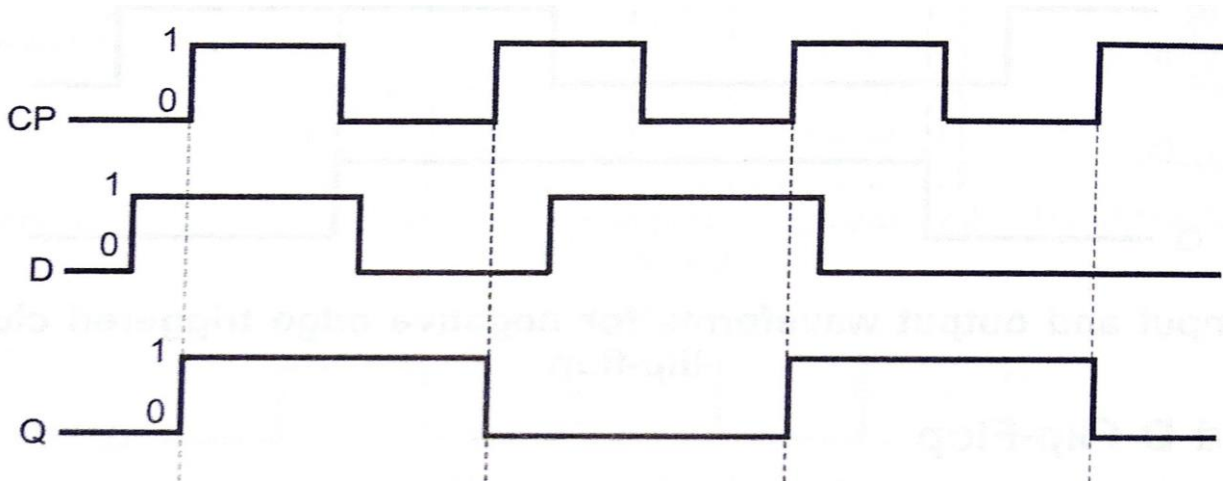
(a) Rising-edge trigger

(b) Falling-edge trigger

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

(c) Truth table

$Q^+ = D$

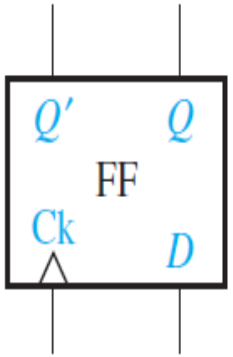


Positive (rising) Edge Triggered

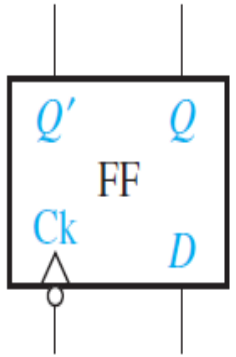
```
Always @ (posedge CP)
Q = D;
```

Advanced Digital Design

Edge Triggered Flipflop



(a) Rising-edge trigger

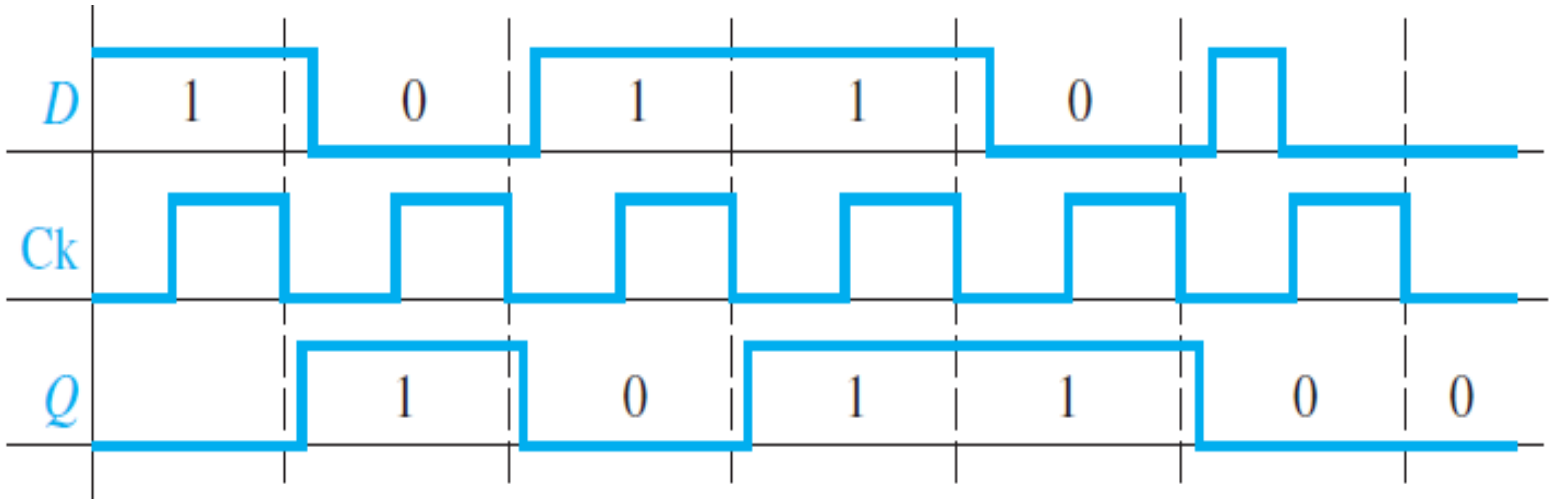


(b) Falling-edge trigger

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

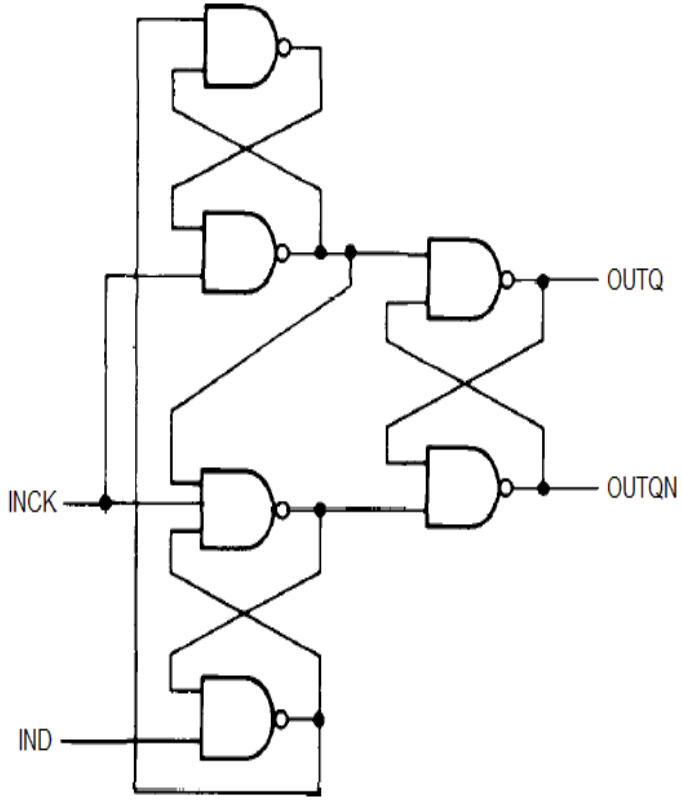
(c) Truth table

$$Q^+ = D$$



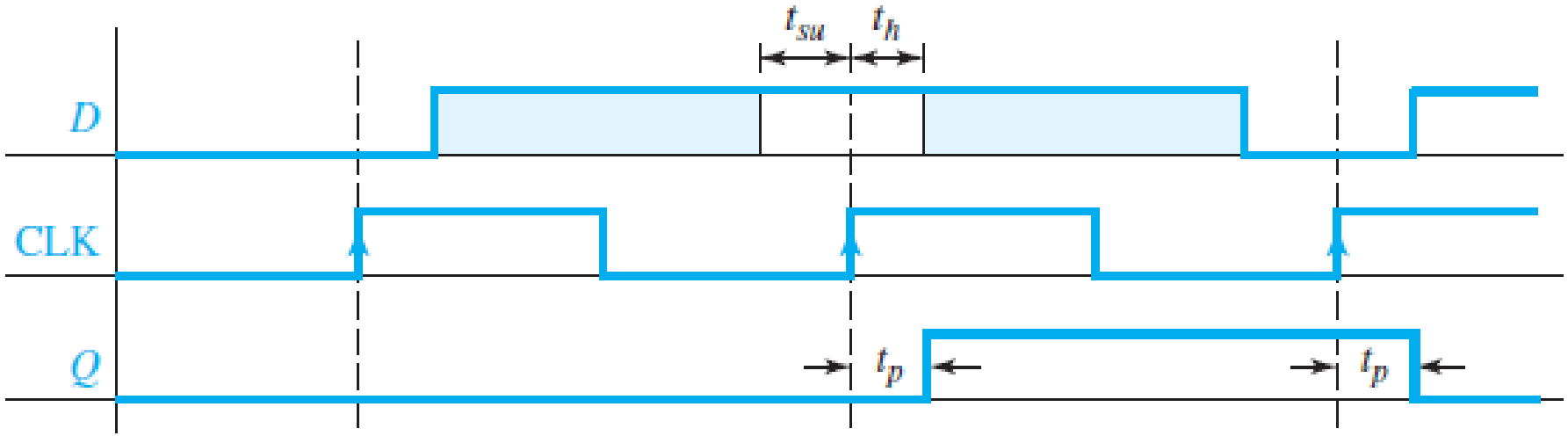
Negative (Falling) Edge Triggered

Always @(negedge Ck)
Q = D;

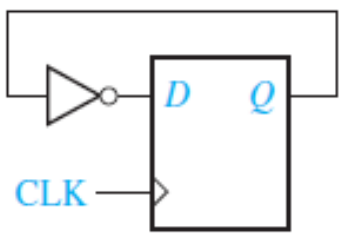


Timing Parameters of Flip-flops

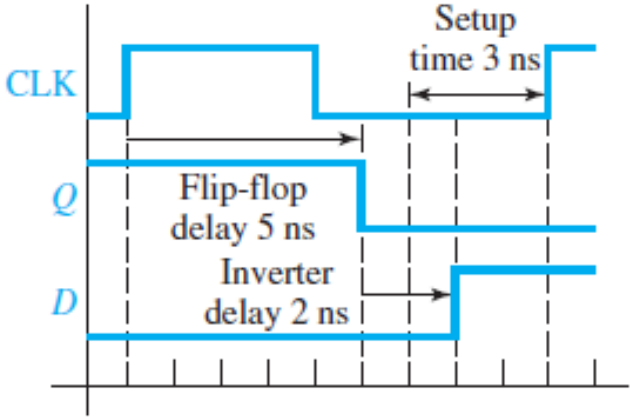
- A flip-flop changes state only on the active edge of the clock, the **propagation delay** of a flip-flop is the time between the active edge of the clock and the resulting change in the output.
- To function properly, the D input to an edge-triggered flip-flop must be held at a constant value for a period of time before and after the active edge of the clock. If D changes at the same time as the active edge, the behavior is unpredictable (metastable = X).
- The amount of time that D must be stable before the active edge is called the setup time (**tsu**), and the amount of time that D must hold the same value after the active edge is the hold time (**th**).
- The propagation delay **(tp)** from the time the clock changes until the **Q** output changes is also indicated.



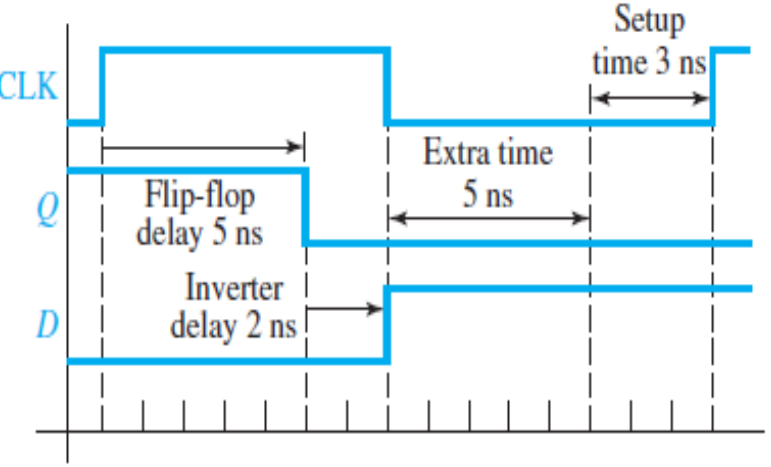
Advanced Digital Design
Determination of Minimum Clock Period



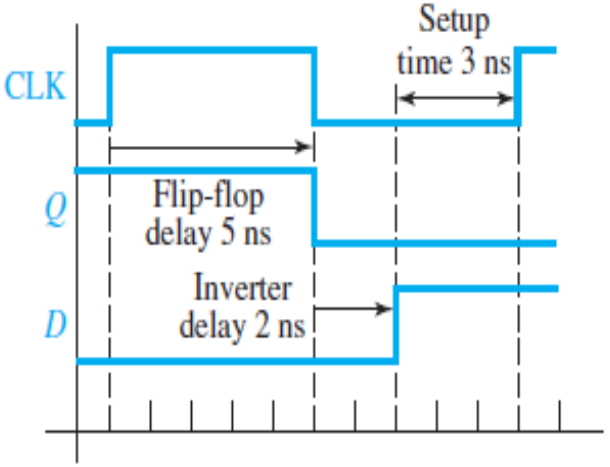
(a) Simple flip-flop circuit



(b) Setup time not satisfied



(c) Setup time satisfied

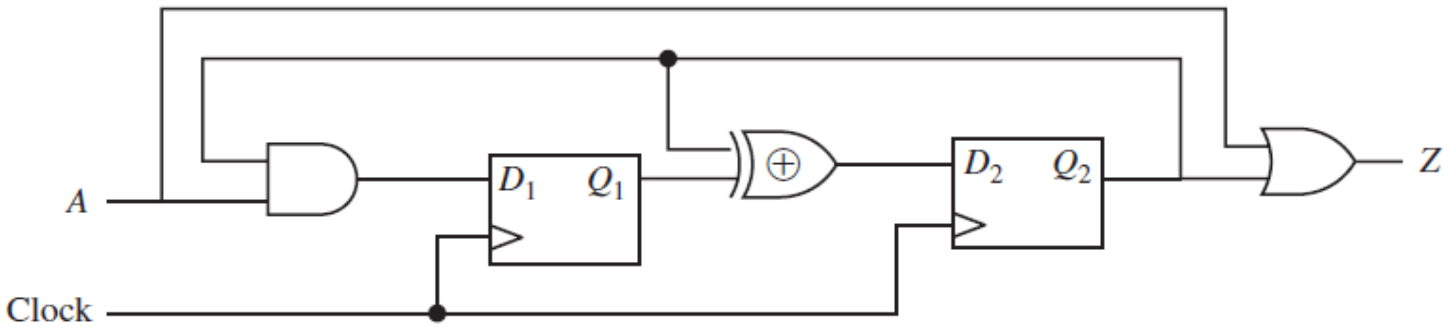


(d) Minimum clock period

- Suppose the inverter has a propagation delay of 2 ns, and suppose the flip-flop has a propagation delay of 5 ns and a setup time of 3 ns. Suppose, as in Figure, that the clock period is 9 ns, i.e., 9 ns is the time between successive active edges.
- Suppose instead that the clock period were 15 ns, as in Figure (c). Again, the input to the flip-flop will change 7 ns after the rising edge. However, because the clock is slower, this is 8 ns before the next rising edge. Therefore, the flip-flop will work properly. Note in Figure(c) that there is 5 ns of extra time between the time the D input is correct and the time when it must be correct for the setup time to be satisfied. Therefore, we can use a shorter clock period, and have less extra time, or no extra time. Figure(d) shows that 10 ns is the minimum clock period which will work for this circuit.

Advanced Digital Design

Timing Conditions for Proper Operation



There are six static timing paths in this circuit:

- From A to D1 (primary input to flip-flop)
- From D1 to D2 including the XOR (flip-flop to flip-flop)
- From D2 via XOR to D2 (flip-flop to flip-flop)
- From D2 to D1 via AND (flip-flop to flip-flop)
- From D2 to Z via the OR gate (flip-flop to output)
- From A to Z via the OR gate (input to output)

Advanced Digital Design

Timing Conditions for Proper Operation

- Static timing analysis checks how the data arrives with respect to clock. It detects setup and hold-time violations in the design so that they can be corrected.
- A setup time violation occurs if the data changes just before the clock without providing enough setup time for the flip-flop.
- A hold-time violation occurs if the data changes just after the clock without providing enough hold time for the flip-flop.
- Slack is the amount of time still left before a signal will violate a setup or hold-time constraint. Paths must have a positive or zero slack in order to have no violations.
- Paths that have a zero or very small slack are the speed-limiting paths in the design, because any small changes in clock or gate delays will lead to violations in such circuits.
- Paths that have a negative slack time have already violated a setup or hold constraint.

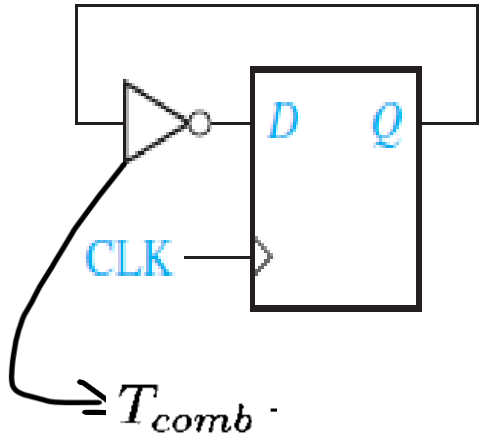
Maximum Operating Frequency

- The timing of a sequential circuit is characterized by **fmax,** the maximal clock frequency, which specifies how fast the circuit can operate. The reciprocal off specifies **Tclock**, the minimal clock period, which can be interpreted as the interval between two sampling edges of the clock.
- To ensure correct operation, the next value must be generated and stabilized within this interval. Assume that the maximal propagation delay of next-state logic is **Tcomb**. The minimal clock period can be obtained by adding the propagation delays and setup time constraint of the closed loop

$$T_{clock} = T_{cq} + T_{comb} + T_{setup}$$

and the maximal clock rate is the reciprocal:

$$f_{max} = \frac{1}{T_{clock}} = \frac{1}{T_{cq} + T_{comb} + T_{setup}}$$



References

- Digital Systems Engineering by William Dally and Poulton

THANK YOU

Sudeendra kumar K

Department of Electronics and Communication Engineering

sudeendrakumark@pes.edu

Advanced Digital Design

Dr. Sudeendra kumar K

Department of Electronics and Communication Engineering

ADVANCED DIGITAL DESIGN

Lecture-1: Skew, Jitter and Clock Properties

Unit - 2

Sudeendra kumar K
Department of Electronics and Communication Engineering

Advanced Digital Design

Contents

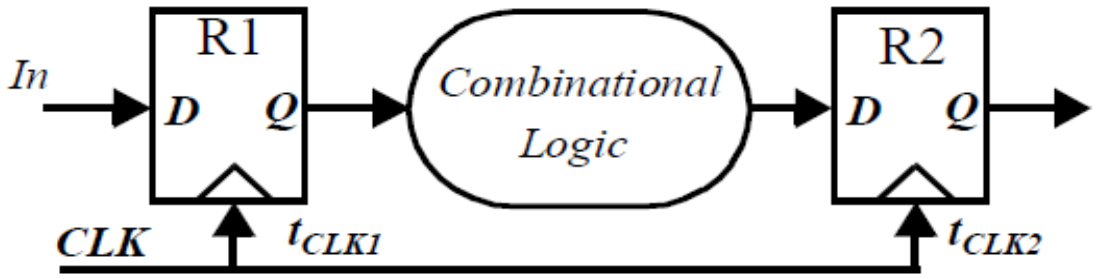
- Skew and Jitter
- Timing Properties of Combinational Circuits
- Timing Properties of Sequential Circuits (Clocked Circuits)
- Eye Diagram

Skew

- We analyze the impact of **spatial variations** of the clock signal, called clock skew, and temporal variations of the clock signal, called clock jitter, and introduce techniques to cope with it.
- The spatial variation in arrival time of a clock transition on an integrated circuit is commonly referred to as clock skew.
- The clock skew between two points *i* and *j* on a IC is given by $\delta(i,j) = t_i - t_j$, where t_i and t_j are the position of the rising edge of the clock with respect to a reference.

Skew

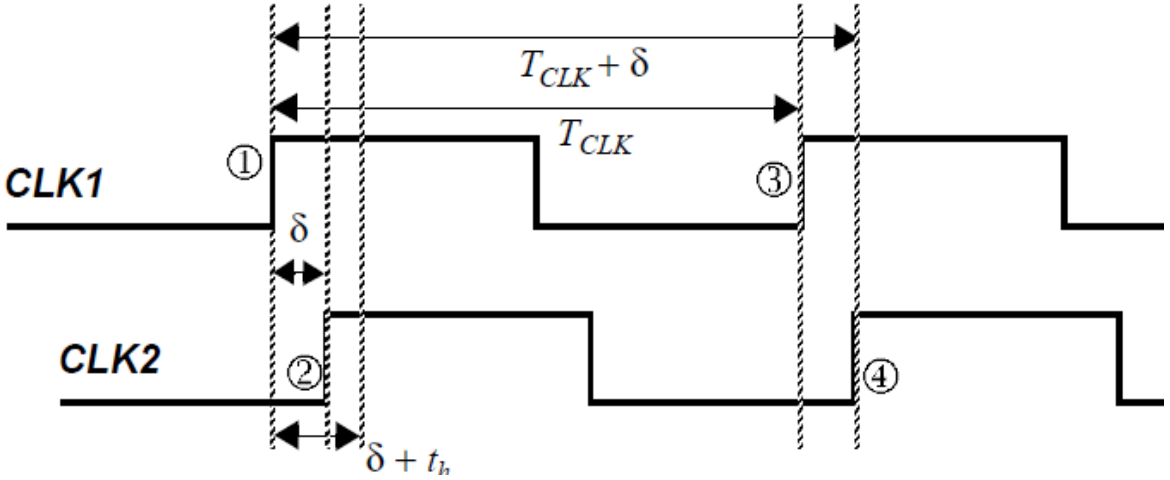
- Consider the transfer of data between registers R1 and R2 in Figure.

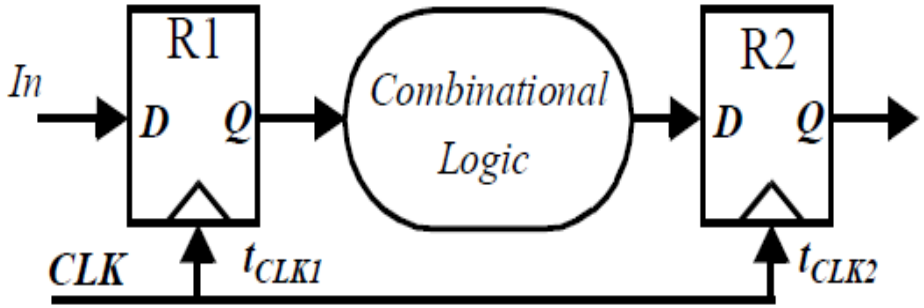


- The clock skew can be positive or negative depending upon the routing direction and position of the clock source. The timing diagram for the case with positive skew is shown in Figure.

Skew

- As the figure illustrates, the rising clock edge is delayed by a positive δ at the second register.



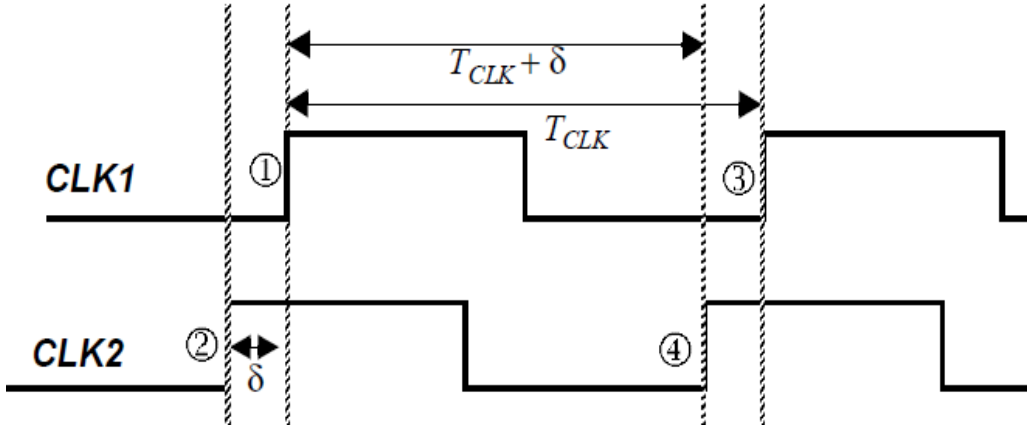


Timing Diagram to study the impact of Clock skew on performance and Functionality. In this sample timing diagram, δ > 0

- Skew has strong implications on performance and functionality of a sequential system. First consider the impact of clock skew on performance. From Figure, a new input In sampled by R1 at edge 1 will propagate through the combinational logic and be sampled by R2 on edge 4. If the clock skew is positive, the time available for signal to propagate from R1 to R2 is increased by the skew δ. The output of the combinational logic must be valid one set-up time before the rising edge of CLK2 (point 4).

Skew

- Clock skew is caused by static path-length mismatches in the clock load and by definition skew is constant from cycle to cycle. That is, if in one cycle CLK2 lagged CLK1 by δ, then on the next cycle it will lag it by the same amount. It is important to note that clock skew does not result in clock period variation, but rather phase shift.



Timing Diagram for the case when δ < 0. The rising edge of CLK2 arrives earlier than the edge of CLK1

- Figure shows the timing diagram for the case when δ < 0. For this case, the rising edge of CLK2 happens before the rising edge of CLK1. On the rising edge of CLK1, a new input is sampled by R1.
- The new sampled data propagates through the combinational logic and is sampled by R2 on the rising edge of CLK2, which corresponds to edge 4.

Jitter

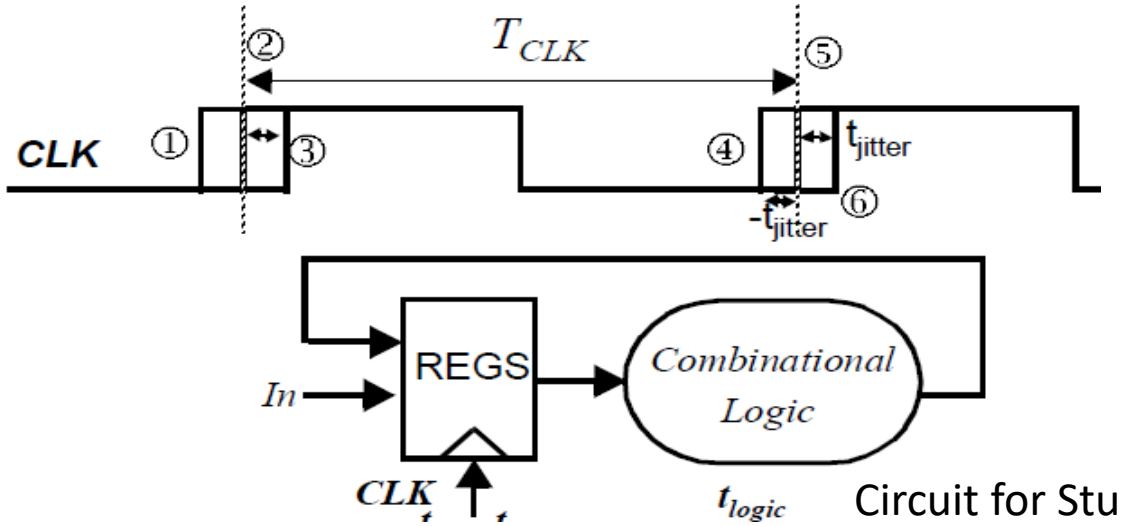
- Clock jitter refers to the temporal variation of the clock period at a given point — that is, the clock period can reduce or expand on a cycle-by-cycle basis.
- It is strictly a temporal uncertainty measure and is often specified at a given point on the chip. Jitter can be measured and cited in one of many ways.
- Cycle-to-cycle jitter refers to time varying deviation of a single clock period and for a given spatial location i is given as:

$$T_{jitter,i}(n) = Ti, n+1 - Ti,n$$

- TCLK, where ***Ti,n*** is the clock period for period **n, Ti, n+1** is clock period for period **n+1,** and TCLK is the nominal clock period.

Jitter

- Jitter directly impacts the performance of a sequential system. Figure shows the nominal clock period as well as variation in period.
- Ideally the clock period starts at edge 2and ends at edge 5 and with a nominal clock period of TCLK.
- However, as a result of jitter, the worst case scenario happens when the leading edge of the current clock period is delayed (edge 3), and the leading edge of the next clock period occurs early (edge 4).

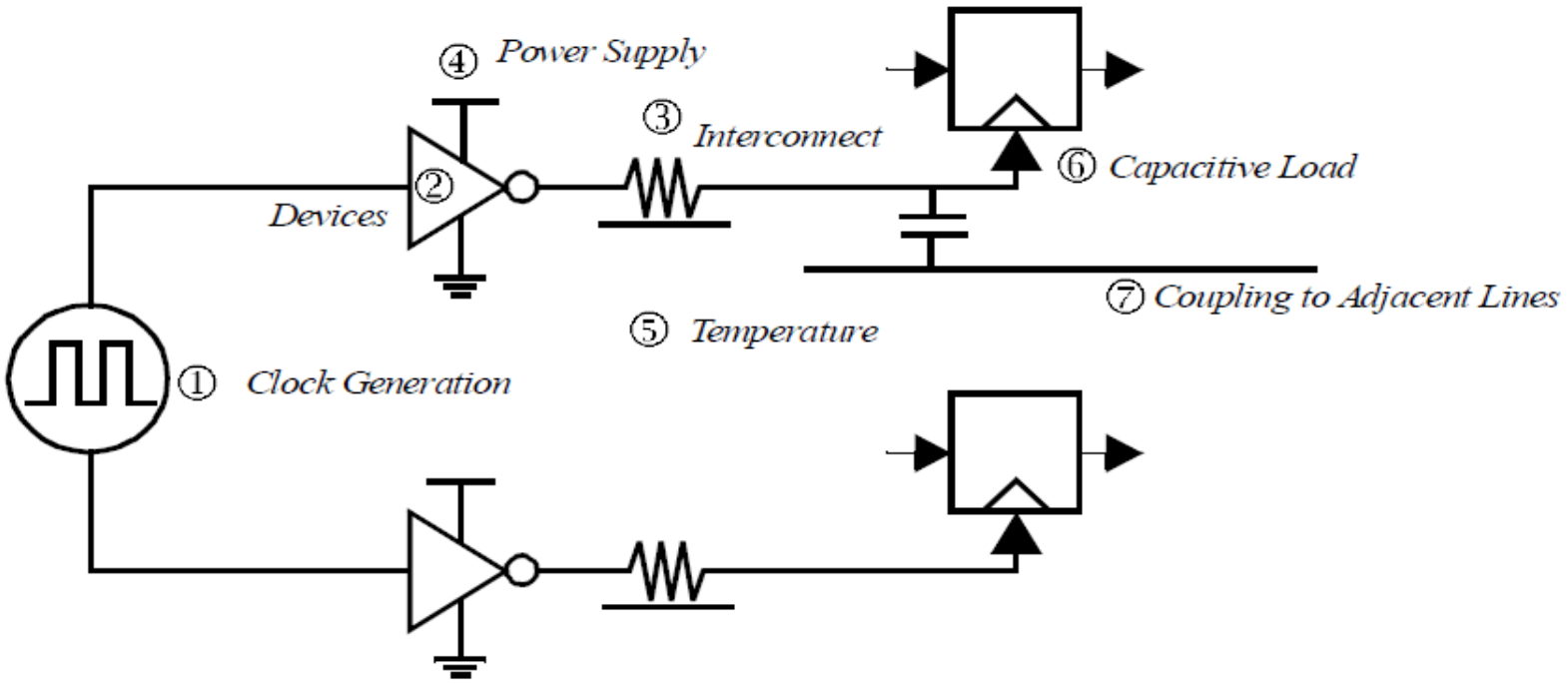


Circuit for Studying the impact of Jitter

Advanced Digital Design

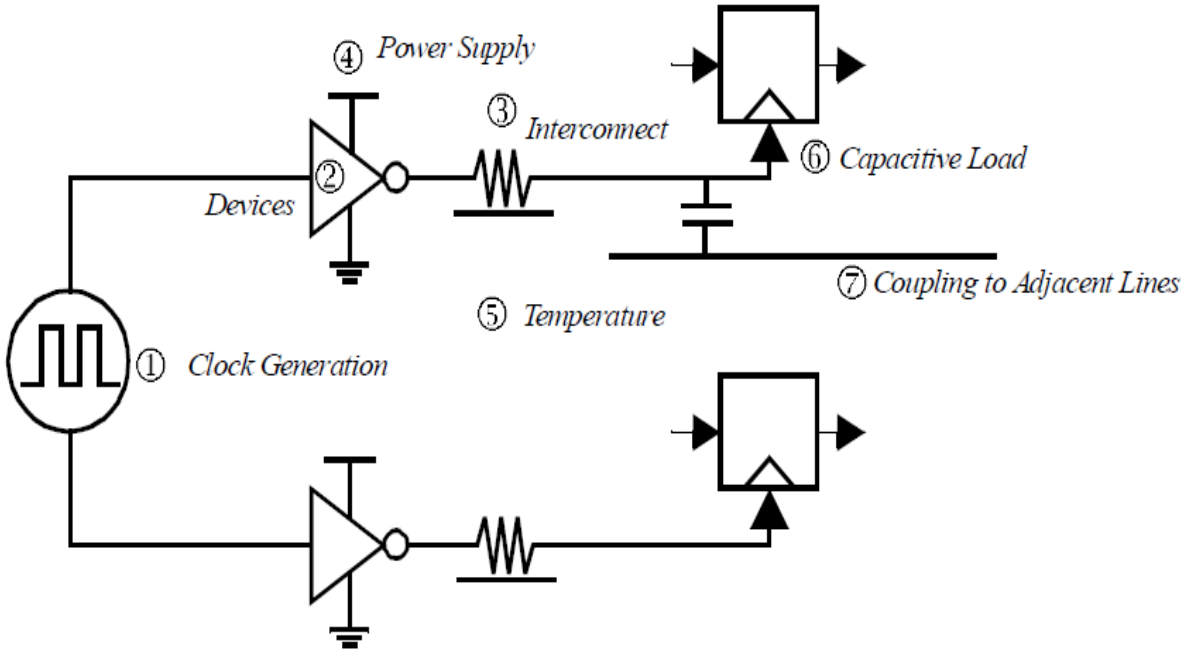
Sources of Clock skew and Jitter

- A perfect clock is defined as perfectly periodic signal that is simultaneous triggered at various memory elements on the chip. However, due to a variety of process and environmental variations, clocks are not ideal.
- To illustrate the sources of skew and jitter, consider the simplistic view of clock generation and distribution as shown in Figure.



Advanced Digital Design
Sources of Clock skew and Jitter

- Typically, a high frequency clock is either provided from off chip or generated on-chip.
- From a central point, the clock is distributed using multiple matched paths to low-level memory element registers.
- In the picture, two paths are shown. The clock paths include wiring and the associated distributed buffers required to drive interconnects and loads.
- A key point to realize in clock distribution is that the absolute delay through a clock distribution path is not important; what matters is the relative arrival time between the output of each path at the register points (i.e., it is perfectly acceptable for the clock signal to take multiple cycles to get from a central distribution point to a low-level register as long as all clocks arrive at the same time to different registers on the chip).

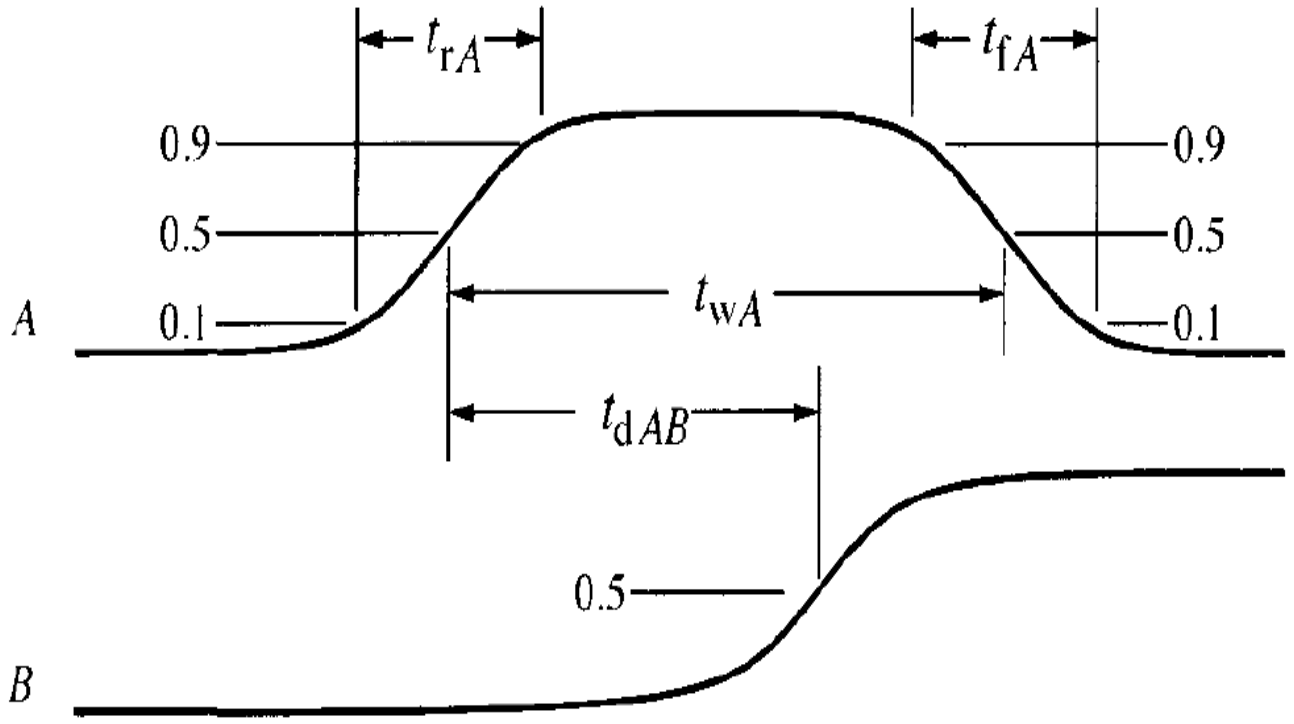


Sources of Clock skew and Jitter

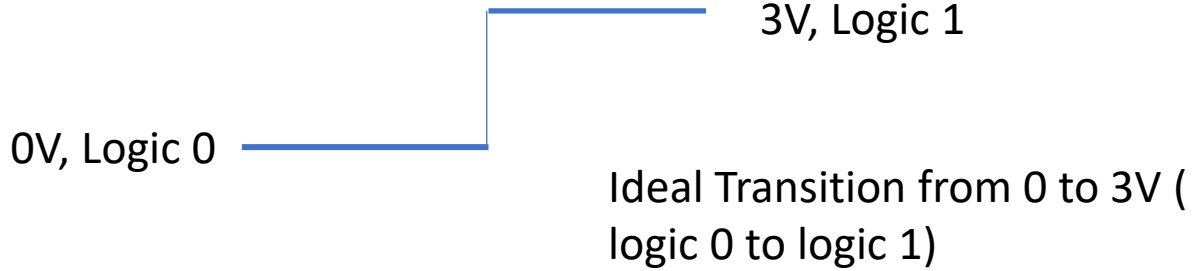
- The are many reasons why the two parallel paths don't result in exactly the same delay. The sources of clock uncertainty can be classified in several ways.
- First, errors can be divided into systematic or random. Systematic errors are nominally identical from chip to chip, and are typically predictable.
- Random errors are due to manufacturing variations (e.g., dopant fluctuations that result in threshold variations) that are difficult to model and eliminate.
- A clock network tuned by a one-time calibration or trimming would be vulnerable to time-varying mismatch due to varying thermal gradients (Temperature Variations)
- Power supply variations also cause skews and Jitters in Clocks.

Advanced Digital Design
Delay and Transition Times

- Figure illustrates the terms used to describe signal transitions and delays between transitions.
- We measure times associated with a single transition from when the signal has completed 10% of the transition to when the signal has completed 90% of the transition.
- The relative timing between two transitions is measured from the 50% point of one transition to the 50% of the other transition.
- For example, signal A's rise time**, trA** , is the time from the start (10%) to the end (90%) of a rising transition on A. Similarly, the fall time of A, **tfA**, is the time from the start to the end of a falling transition on A.



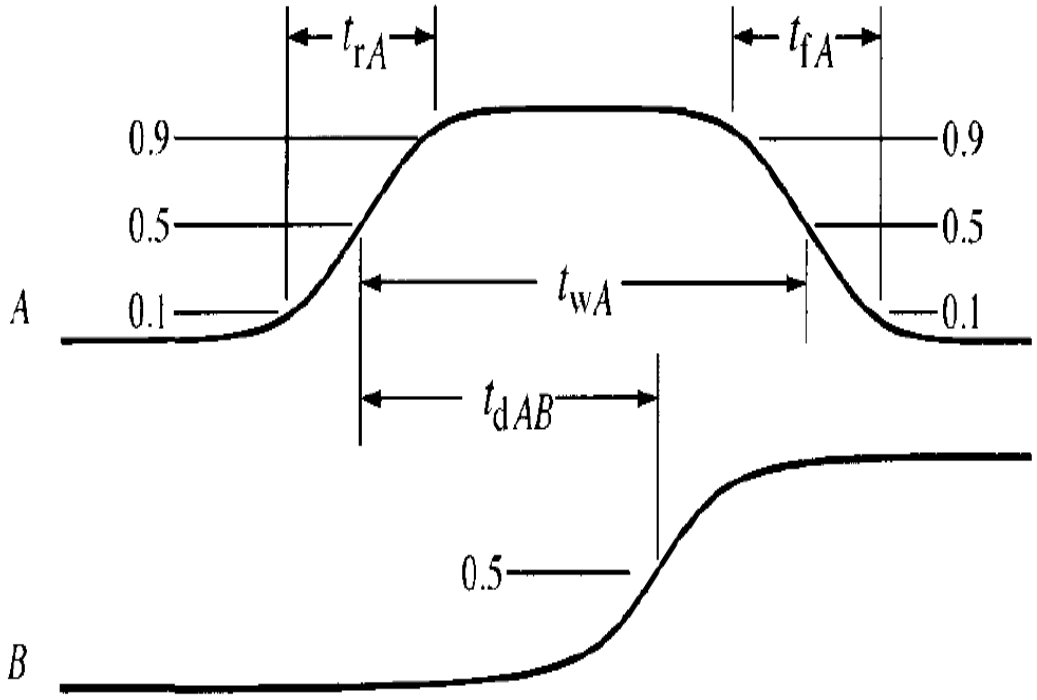
Rise Time, Fall Time, Pulse Width and Delay



Advanced Digital Design

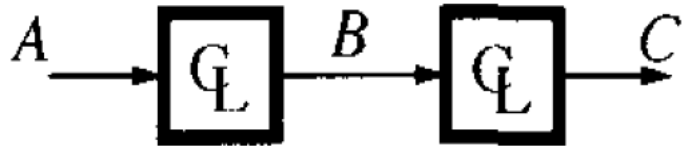
Delay and Transition Times

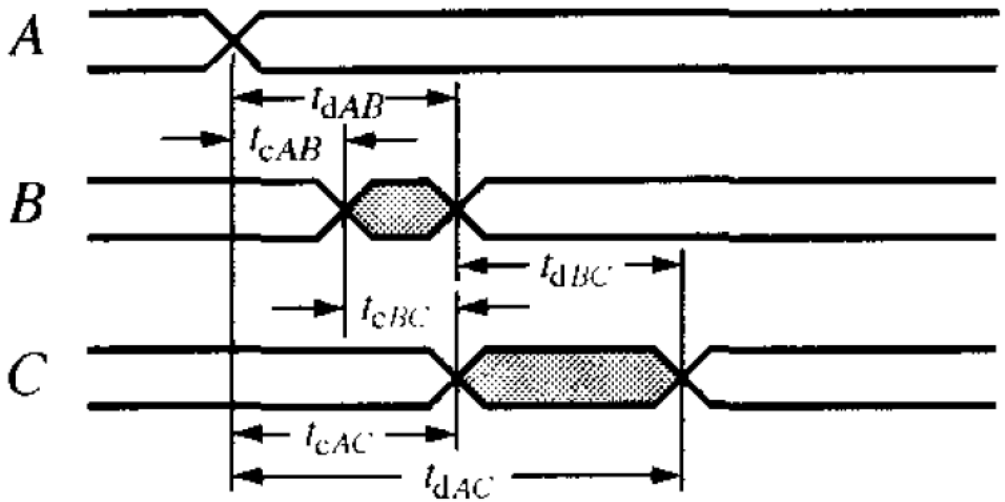
- We denote the time of the $\mathbf{i_{th}}$ transition on a signal A as $\mathbf{t_{Ai}}$, the time when A crosses its midpoint for the $\mathbf{i_{th}}$ time after some reference time.
- The delay from a transition on A to a transition on B, **tdAB** or just **tAB,** is just **tBj - tAi**, for some i and j. This definition holds even when A and B have very different signal amplitudes.
- The pulse width of the signal, **twA,** is the time from one transition on A to the next transition on A, **tA(i+1) - tAi**.
- Often we will distinguish between the high pulse width, **twhA**, and the low pulse width, **twlA,** of signal A.



Composition of Combinational Logic Delays

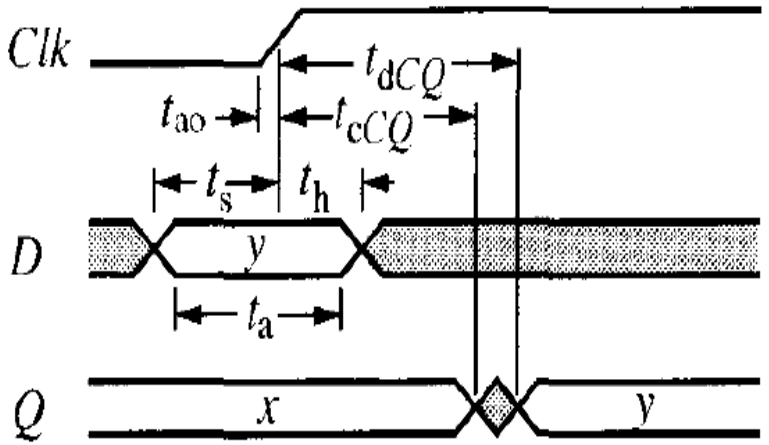
- When combinational logic circuits are composed in series, the propagation delay of the composite circuit is the sum of the delays of the individual elements.
- The same is true for contamination delay. As shown in Figure, because propagation delay is in general larger than contamination delay, the shaded region of uncertainty grows as circuits are combined.
- The propagation delay from an input i to an output j is the maximum sum of element delays over all paths from i to j. Similarly, the contamination delay is the minimum sum of element contamination delays over all paths from i to j.

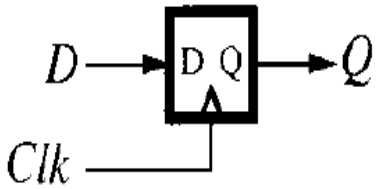


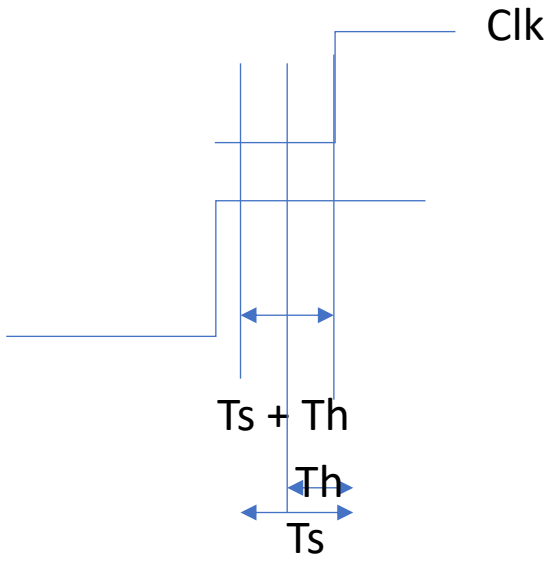


Advanced Digital Design
Edge Triggered Flip-flop

- Figure illustrates the timing properties of an edge-triggered flip-flop. The input, D, must have a stable S value (y in the figure) during the aperture time of the flip-flop.
- The aperture has width, **tao** The aperture offset time, **tao**, is the time from the center of the aperture to the rising edge of the clock.
- The output remains at its previous value (x in the figure) until at least a contamination delay, **tccQ**, after the clock, and the output changes to its new value (y) at most a propagation delay, **tdDQ**, after the clock.

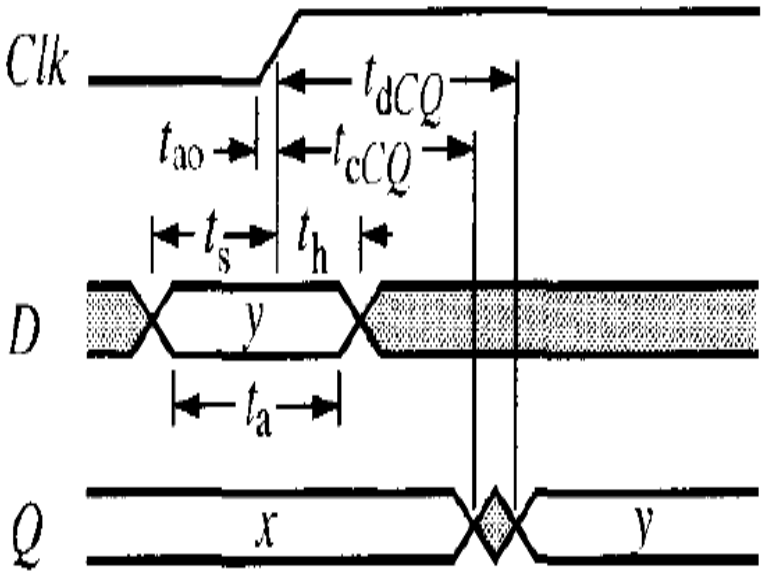


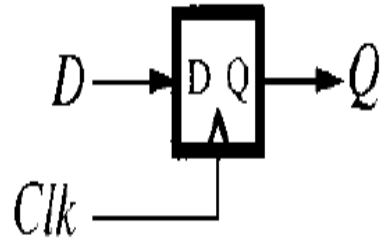




Advanced Digital Design
Edge Triggered Flip-flop

- Most data sheets and texts on logic design describe the position of the aperture relative to the clock in terms of a setup time, **ts**, and a hold time, **th.**
- The setup time is the delay from the data's becoming valid to the rising edge of the clock.
- Because delays are specified from the 50% point of the waveform, whereas the aperture time is specified from the.10% or 90% point, the setup time is from **tr /2** before the beginning of the aperture to the rising edge of the clock.
- Similarly, the hold time is the delay from the clock to the data's becoming invalid.



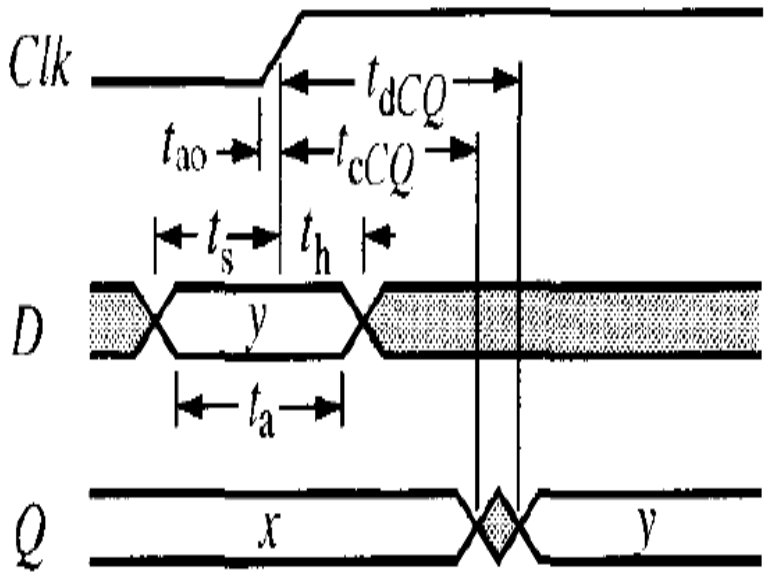


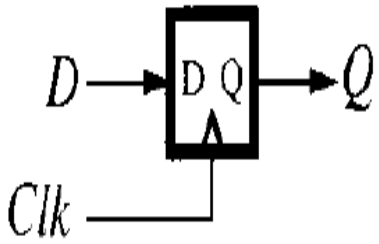
Advanced Digital Design

Edge Triggered Flip-flop

- It is straightforward to convert between these two representations of the aperture interval Using below equation.
- The setup and hold representation is more convenient in open-loop timing systems where setup figures in the maximum delay calculation and hold figures in the minimum delay calculation.
- The aperture representation is more convenient in closed-loop timing systems where the width of the aperture and the uncertainty of delay are the important factors.

$$t_s = 0.5t_a - t_{ao} + t_r/2$$
$$t_h = 0.5t_a + t_{ao} + t_r/2$$
$$t_a = t_s + t_h - t_r$$
$$t_{ao} = 0.5(t_s - t_h)$$





Edge Triggered Flip-flop

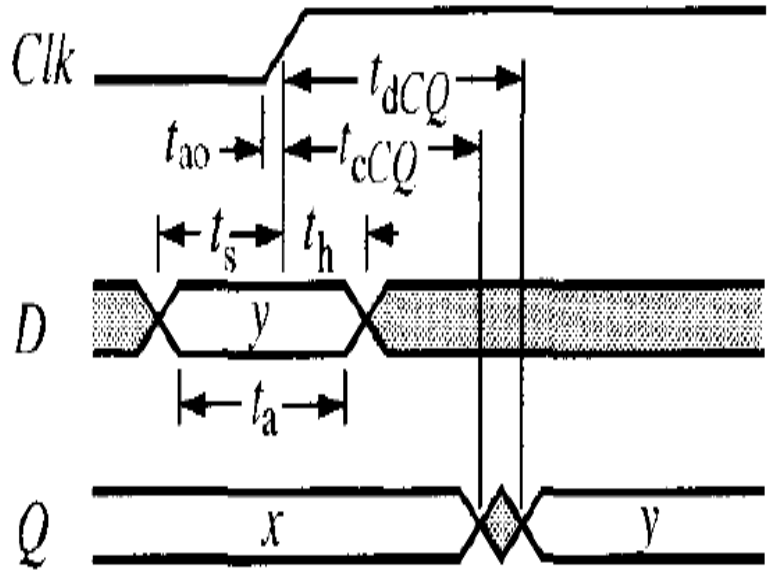
- The aperture time, t_a, is specified in conjunction with the delay, **tdCQ**. The aperture characteristics are really described by a curve in (**ta, tdCQ**) space. A smaller input valid period may result in the correct output, but with larger delay.
- However, there is a minimum valid period below which the correct output will never occur. Also, for some flip-flops the aperture time for a 1 may be different than the aperture time for a 0. For present purposes we ignore this effect and consider the device to be symmetrical.

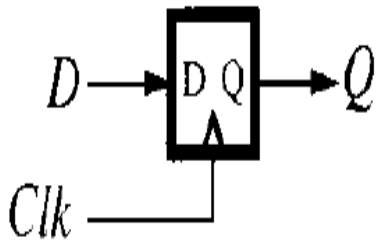
$$t_s = 0.5t_a - t_{ao} + t_r/2$$

$$t_h = 0.5t_a + t_{ao} + t_r/2$$

$$t_a = t_s + t_h - t_r$$

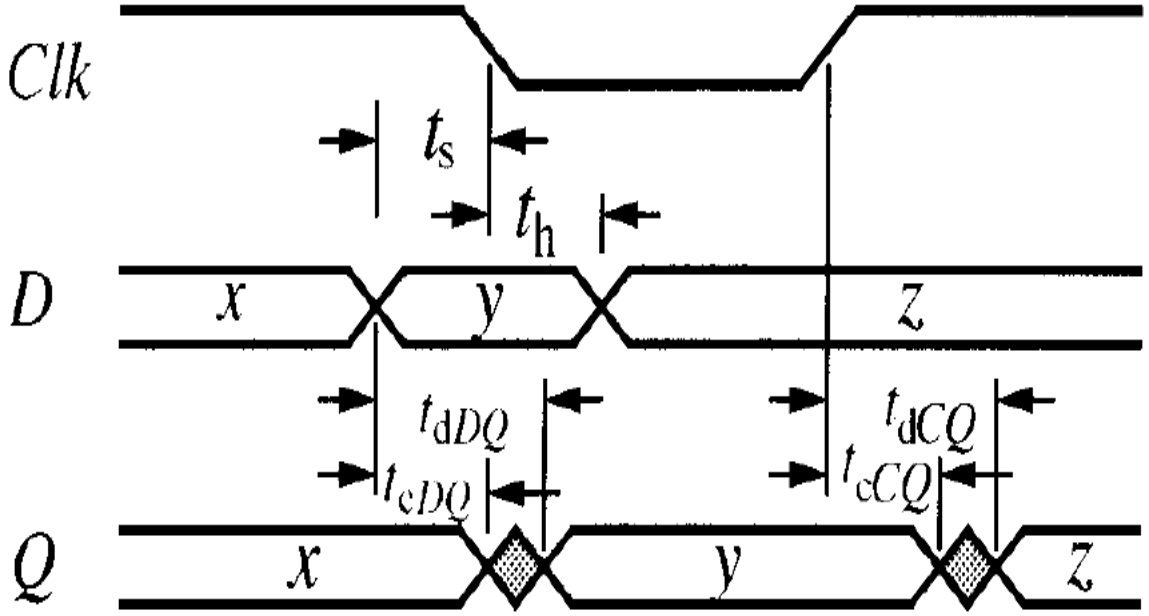
$$t_{ao} = 0.5(t_s - t_h)$$

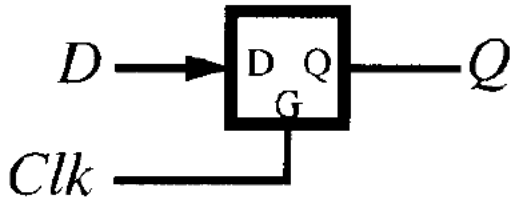




Level Sensitive latch

- The level-sensitive latch is a clocked storage element that logically connects its input to its output when the clock is high and holds the output value stable when the clock is low.
- In the figure, when input D changes from x to y, the output follows immediately (after a propagation delay) because the clock is high. When D changes from y to Z, however, the output does not change until the clock goes high.





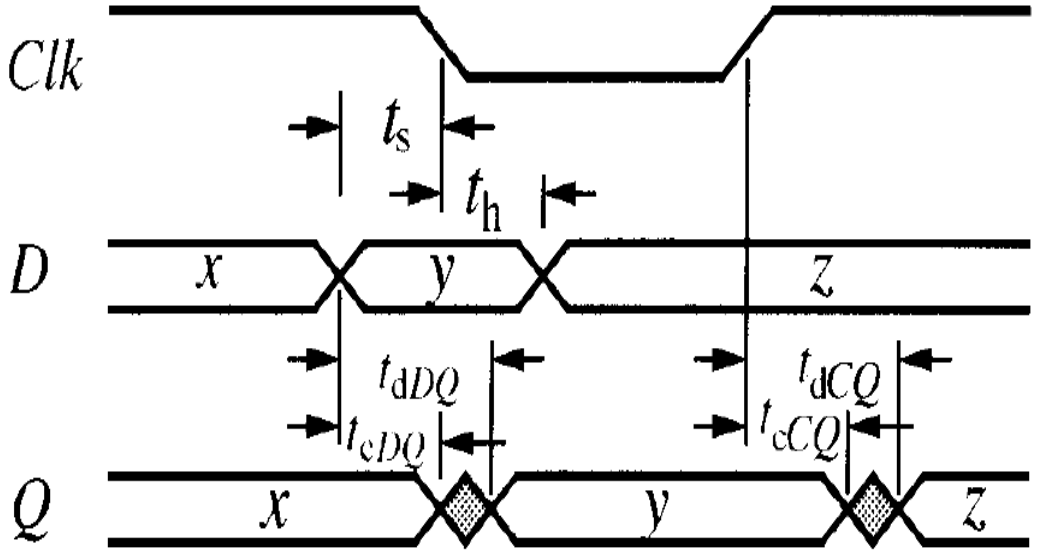
+ve Edge sensitive:-
Always @(Posedge clk)
…..
…..//flip-flops
-ve edge: -
Always @ (negedge clk)
……
….

+ve Level Sensitive: -
Always @ (clk)
….
…. //Latches
-ve Level Sensitive:
Always @ (~clk)
…..
……..

Advanced Digital Design

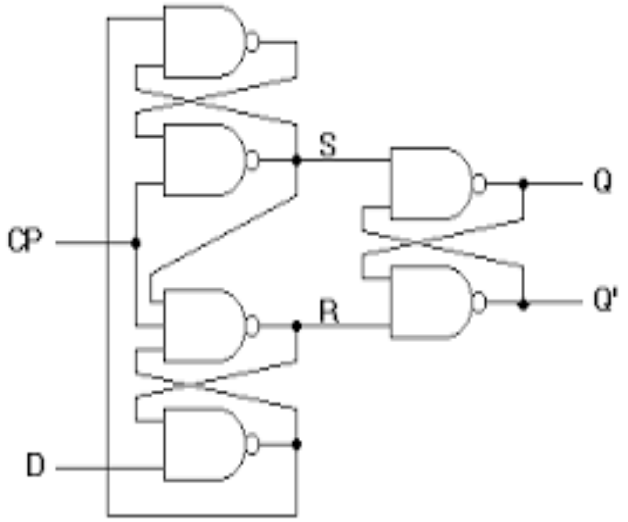
Level Sensitive latch

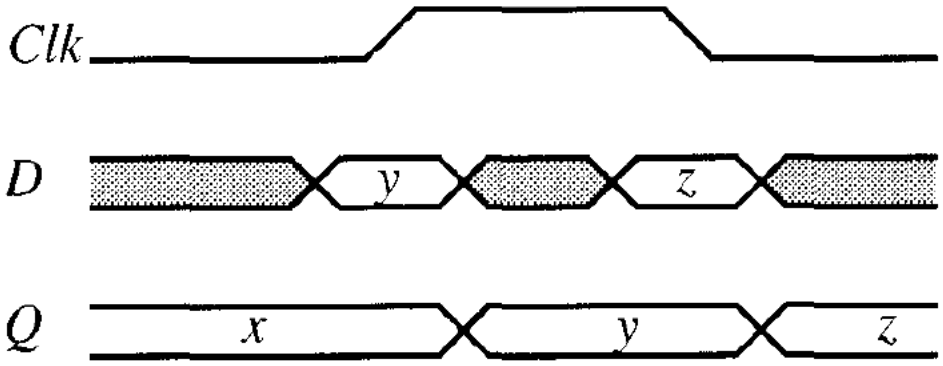
- With a flip-flop, the aperture time and delay are both referenced to the positive edge of the clock and there is no delay property referenced to the data input.
- With a latch, however, the aperture and delay timing properties are referenced to different edges of the clock, and an additional delay property is referenced to the data input.
- The aperture time of a latch is referenced to the falling edge of the clock because this is the edge that samples the data. As illustrated in the figure, the input must remain stable with value y from **ts** before the clock until **th** after the clock for the output to take on the value y reliably while the clock is low.
- The delay is referenced to the rising edge of the clock because this is the edge that may cause the output to change if the input changed while the clock was low.

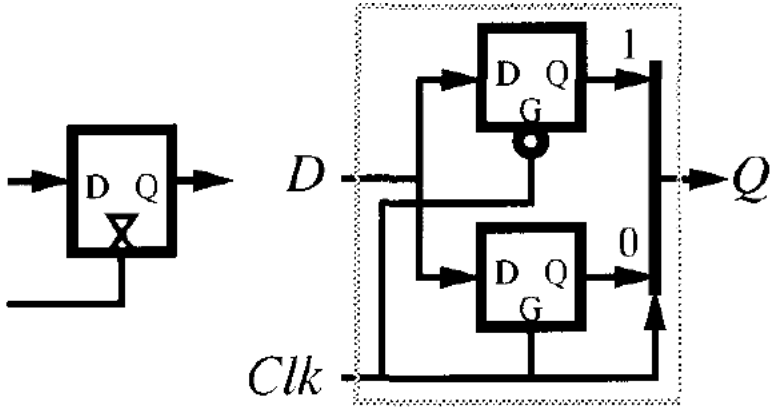


Advanced Digital Design
Double-Edge-Triggered Flip-Flop

- A dual-edge-triggered flip-flop samples its input and updates its output on both the rising and falling edges of the clock.
- However, it does require careful control of the clock's duty factor to ensure that the combinational logic has adequate time to operate during both the clock high and the clock low cycles.
- Figure shows (from left to right) timing waveforms, a symbol, and the construction of a double-edge-triggered flip-flop.
- The waveforms illustrate that each edge of the clock is associated with an aperture and a delay. The input must be stable during the aperture.
- The flip-flop samples the value during the aperture (y during the rising edge and z during the falling edge) and updates the output after a short







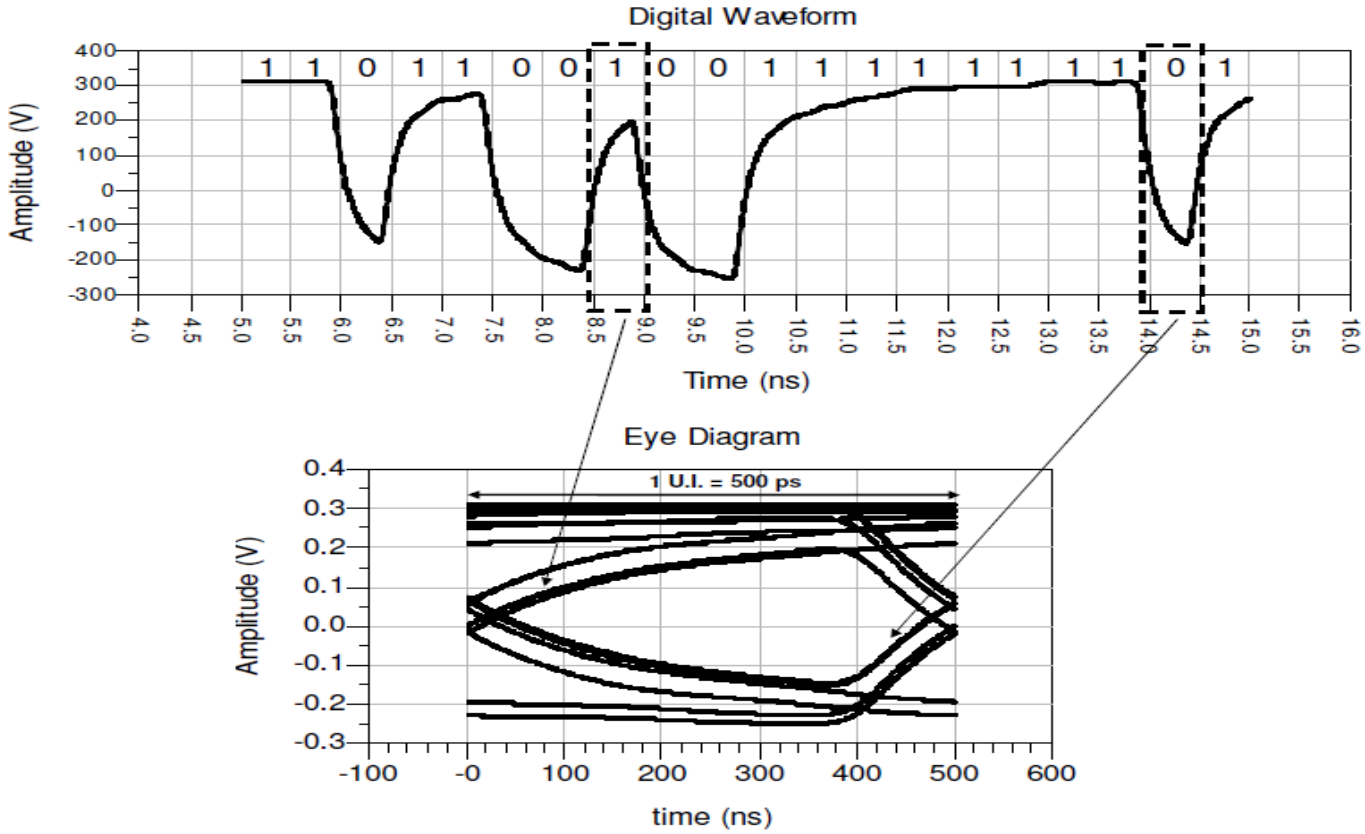
The Eye Diagram

- An eye diagram of a signal overlays the signal's waveform over many cycles.
- Each cycle's waveform is aligned to a common timing reference, typically a clock.
- An eye diagram provides a visual indication of the voltage and timing uncertainty associated with a signal.
- It is easy to generate by synchronizing an oscilloscope to the timing reference.

Advanced Digital Design

The Eye Diagram

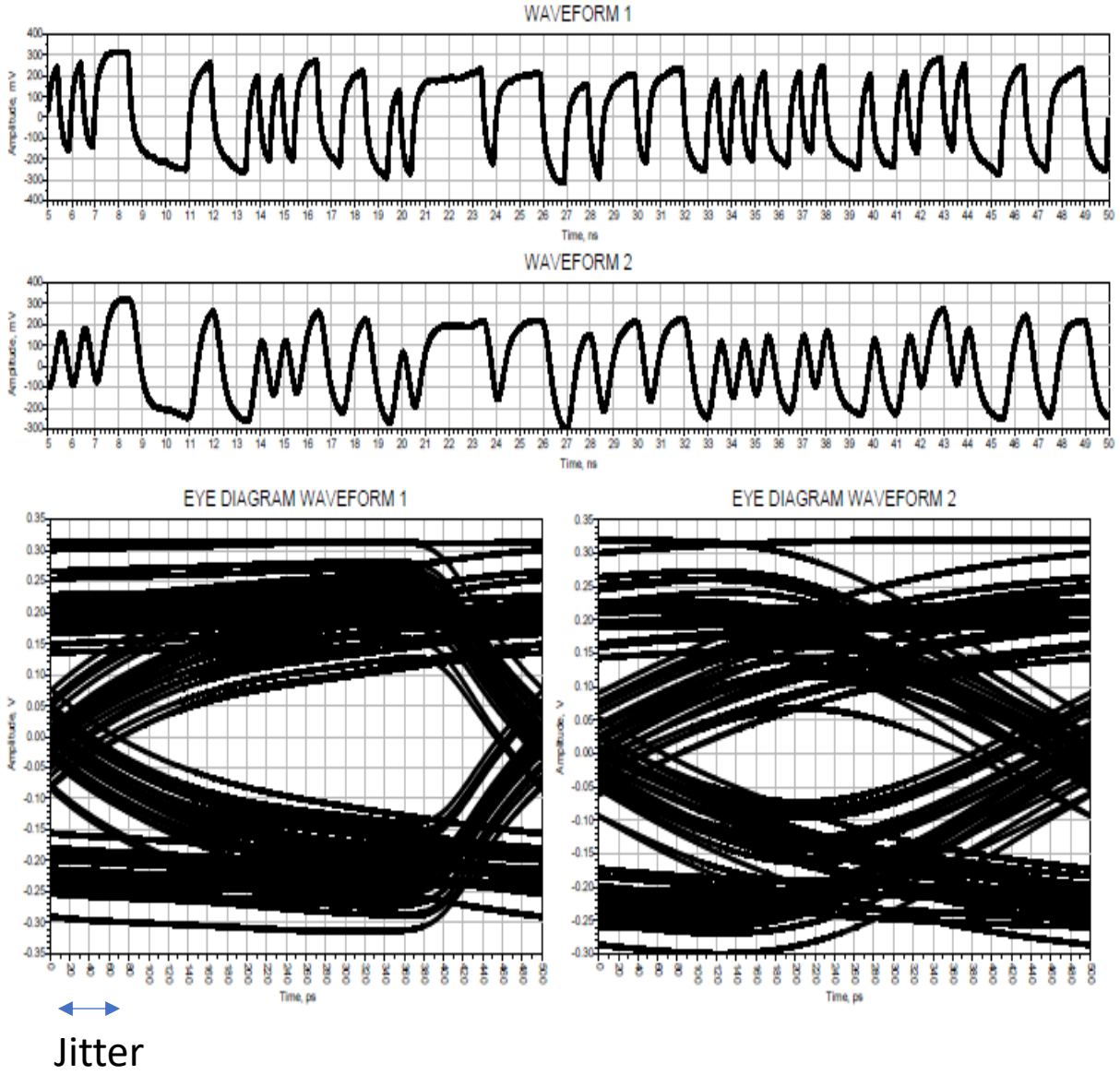
- The data eye diagram is a methodology to display and characterize a highspeed digital signal in the time domain.
- The data eye diagram is derived from the digital waveform by folding the parts of the waveform corresponding to each individual bit into a single graph with a one unit interval (UI) width on the timing axis as shown in Figure.



Advanced Digital Design

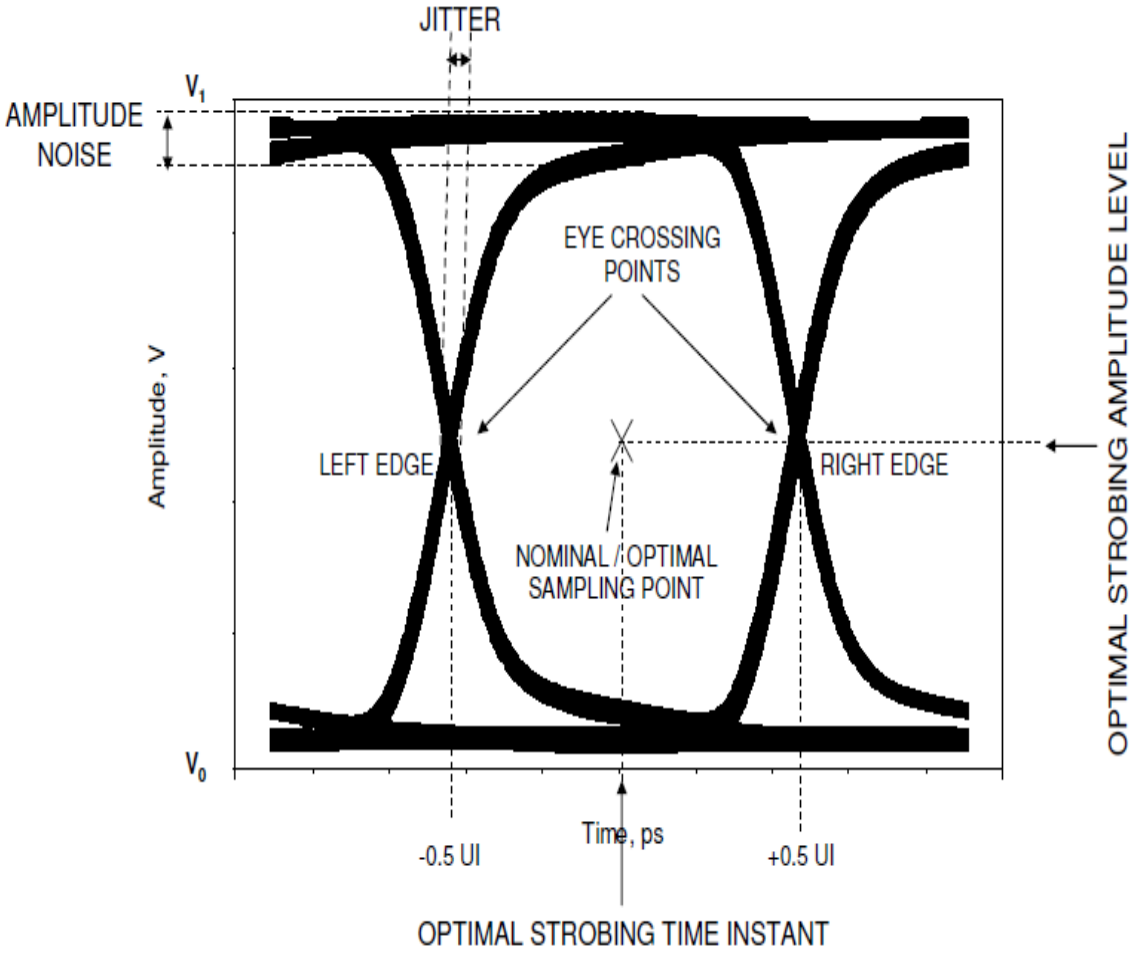
The Eye Diagram

- The data eye diagram does not allow the identification of the individual bits in the bit pattern.
- However, showing all the bits folded into a single UI allows the easy visualization of the quality of the digital signal.
- This is because both the best and the worst case bit transitions on the waveform will be folded together as shown in Figure.
- In the figure two different waveforms are displayed, and although it is not easy to compare the performance of both digital signals when looking at the full waveforms, using the data eye diagram it is obvious that waveform 2 is worse than waveform 1.



Advanced Digital Design

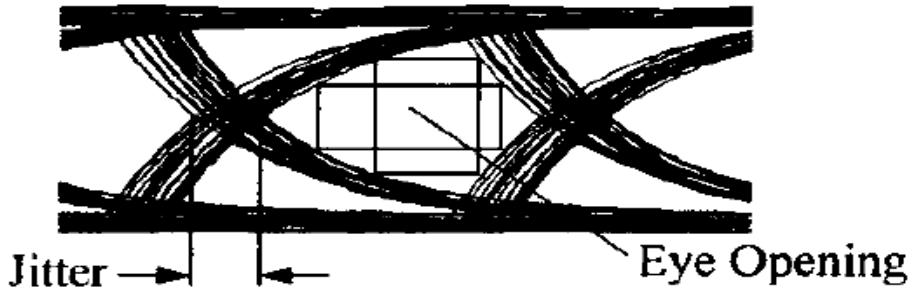
The Eye Diagram



- Figure shows some of the typical nomenclature associated with an eye diagram.
- One key item is the optimal strobing point in the data eye diagram. This is the point with the maximum timing and level margin.
- Another important item is the amount of timing jitter and amplitude noise that are present on the data eye that reduce the timing and level margin.

The Eye Diagram

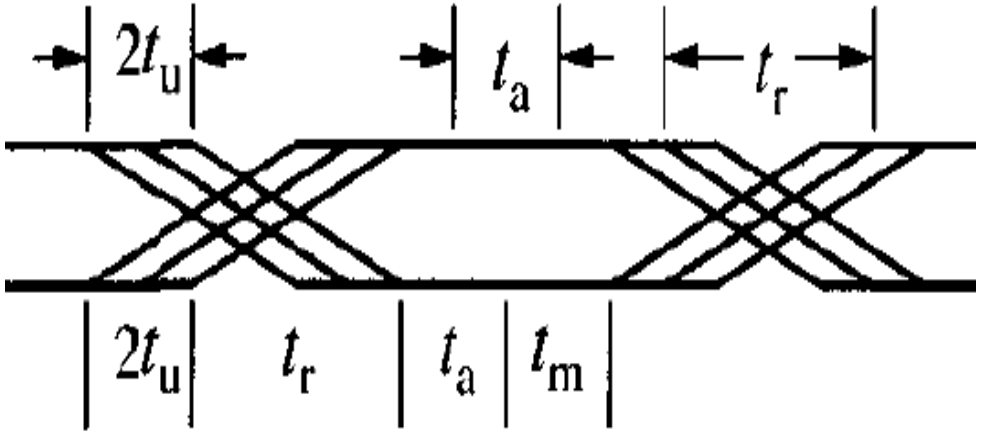
- The size of the eye opening in the center of the diagram indicates the amount of voltage and timing margin available to sample this signal.
- If a margin rectangle with width equal to the required timing margin and height equal to the required voltage margin fits in the opening, then the signal has adequate margins.
- One can trade off voltage margin against timing margin, as illustrated by the two rectangles inscribed in the eye opening.
- One rectangle has a large timing margin that is achieved at the expense of a small voltage margin, whereas the other rectangle trades some of the timing margin to give a larger voltage margin.

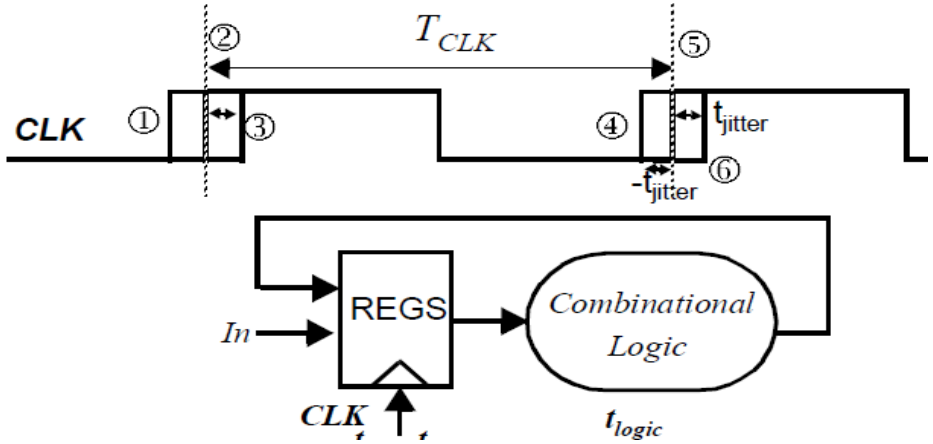


Advanced Digital Design

The Eye Diagram

- Often we will draw an abstract eye diagram, as shown in Figure, to visualize a timing budget.
- The figure shows three superimposed waveform pairs, one each with nominal timing, worst-case early timing, and worst-case late timing.
- The three major components of a timing budget are labeled along the top of the waveforms. The uncertainty, **tu** , is the difference between the nominal waveform and the early or late waveform.?
- The aperture time, **ta**, is a property of the receiver, its flip-flops, or both. The transition time, **tr** is the time required for each waveform to switch states.
- These three components are stacked up along the bottom of the diagram along with a net timing margin, **tm**, to round out the cycle. It is easy to see from the abstract eye diagram that it is the transition time component of the timing budget that is traded off against voltage margin when fitting a margin rectangle into the eye opening.

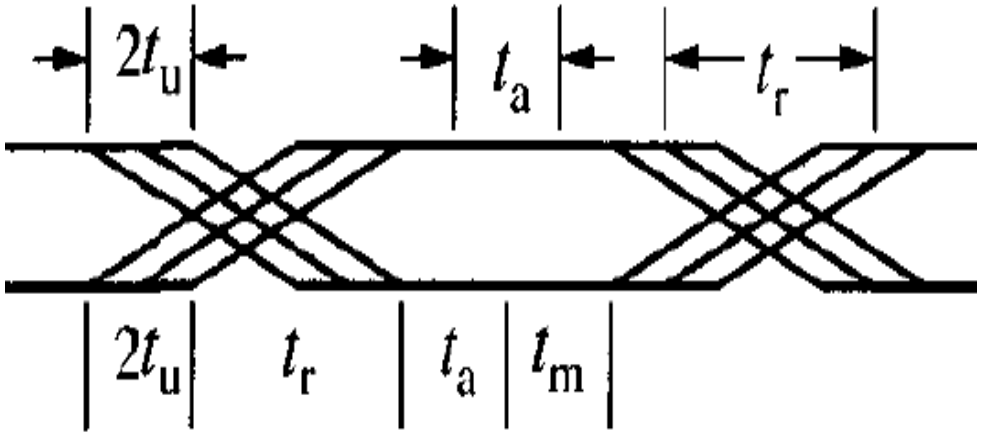




The Eye Diagram

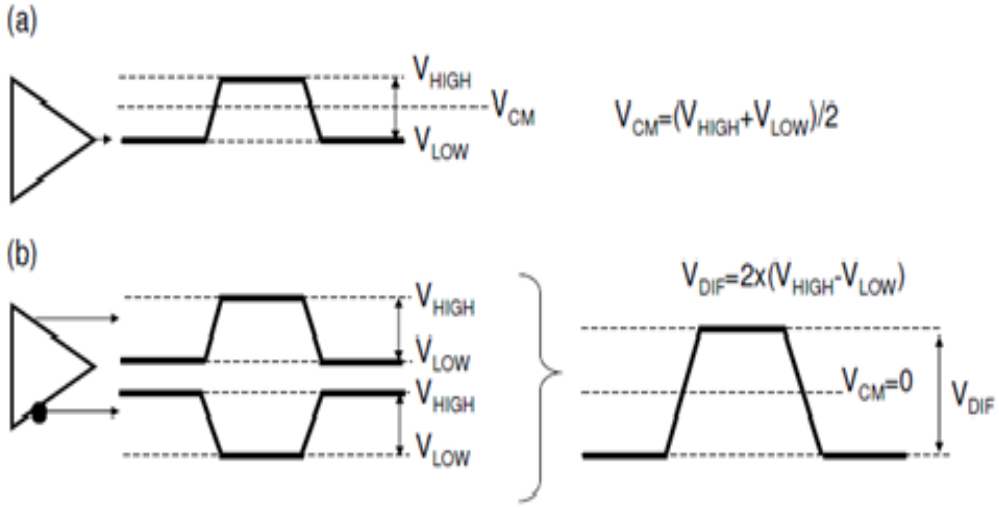
- This diagram illustrates the fundamental equation that limits the cycle time of a logic or transmission module. The cycle time must be large enough to account for uncertainty, aperture, and rise time, as expressed by:

$$t_{cy} \geq 2t_u + t_a + t_r$$

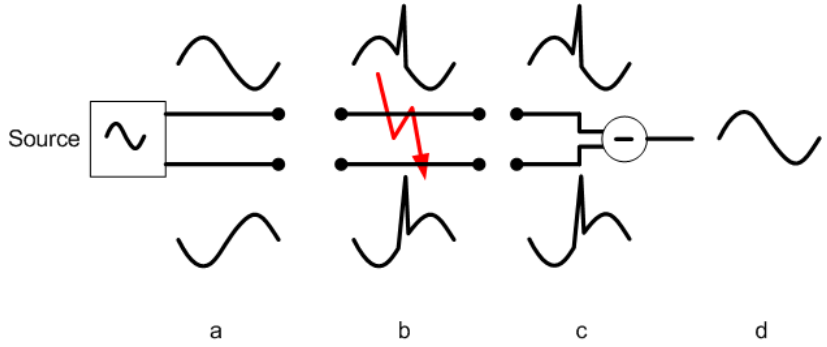


Advanced Digital Design
Differential Signalling

- Most high-speed digital interfaces use differential signaling for electrical transmission. Differential signaling uses two complementary output drivers (differential legs) to drive two independent signal transmission lines.
- Figure shows a comparison between single-ended and differential signaling and how differential signaling is defined.
- While a single-ended signal is characterized by its high voltage level value VHIGH and its low voltage level value VLOW, a differential signal is usually characterized by its differential amplitude VDIF and the common mode value VCM of the two differential legs.
- Differential signaling has several properties that make its use advantageous for high-speed digital interfaces. They include lower **dI/dt,** lower EMI, higher gain, and more immunity to crosstalk if transmitted through a coupled transmission line.



Comparison of (a) single-ended and (b) differential signaling.



References

- Digital Systems Engineering by William Dally and Poulton

THANK YOU

Sudeendra kumar K

Department of Electronics and Communication Engineering

sudeendrakumark@pes.edu