EECS 31L: Introduction to Digital Design Lab Lecture 4

Pooria M.Yaghini

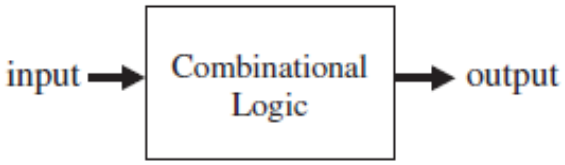
The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

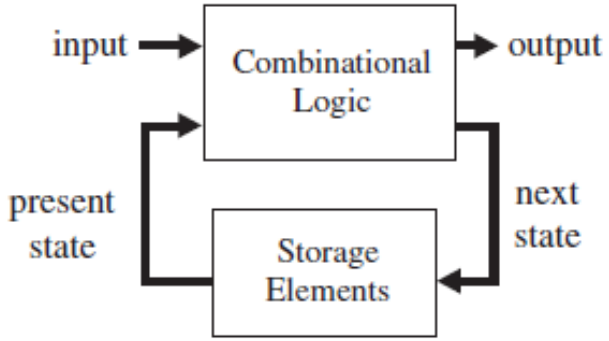
Lecture 4: Outline

- Sequential Coding
- If, case, for, while, …
- Storage Elements

Lecture 4: Combinational vs Sequential logic

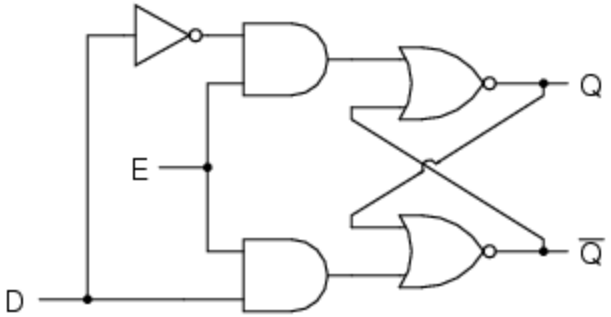
- By definition, combinational logic is that in which the output of the circuit depends solely on the current inputs. It is then clear that, in principle, the system requires no memory and can be implemented using conventional logic gates.
- In contrast, **sequential** logic is defined as that in which the output does depend on previous inputs. Therefore, storage elements are required, which are connected to the combinational logic block through a feedback loop, such that now the stored states (created by previous inputs) will also affect the output of the circuit.



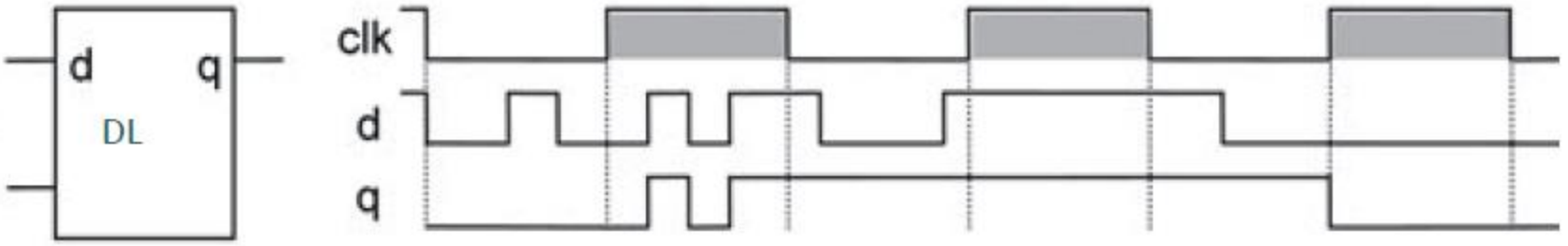


Lecture 4: Sequential Coding

- **Sequential Circuits**
  - D Latch

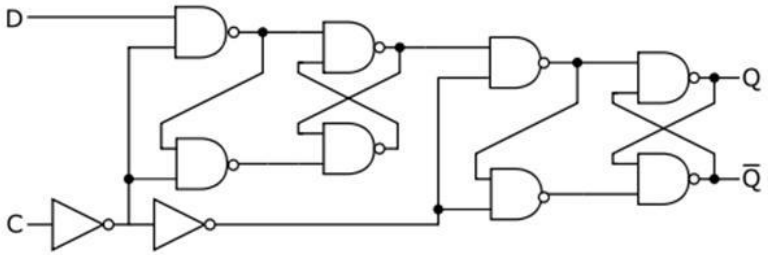


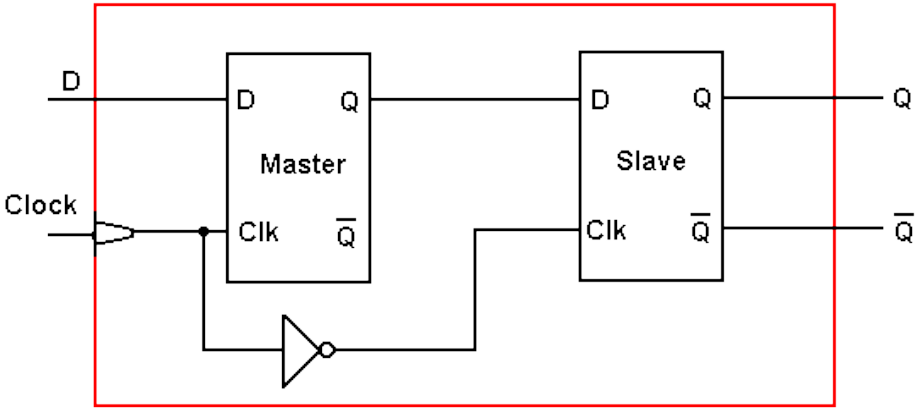
E	D	Q	$\overline{Q}$
0	**0**	latch	latch
0	1	latch	latch
1	**0**	**0**	1
1	1	1	**0**

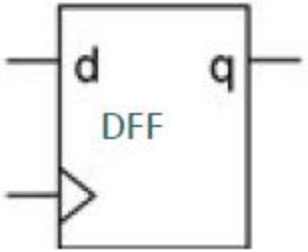


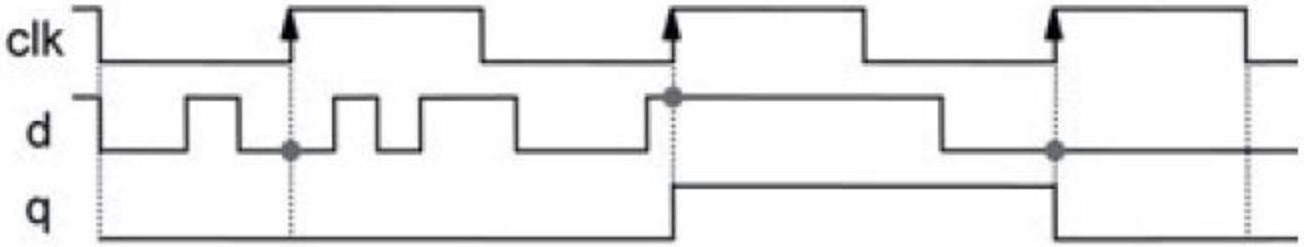
Lecture 4: Sequential Coding

- **Sequential Circuits**
  - D Flip-Flop









Lecture 4: Sequential Coding

Structured procedures

- **initial** procedure
- **always** procedure
  - always
  - always_comb
  - always_latch
  - always_ff
- **final** procedure
- **task**
- **function**

Lecture 4: Sequential Coding

- **initial**
  - is enabled **only once** at the beginning of a simulation
  - Not synthesizable
- **always**
  - is enabled at the beginning of a simulation
  - execute repeatedly until the simulation is terminated

- **final**
  - is enabled at the end of simulation time and execute only once
  - Not synthesizable

Lecture 4: Sequential Coding

There are four forms of always procedures

- **always**
  - represents a general purpose always procedure, which can be used to represent repetitive behavior such as clock oscillators. The construct can also be used with proper timing controls to represent combinational, latched, and sequential hardware behavior.
  - If an always procedure has no control for simulation time to advance, it will create a simulation deadlock condition.

- **always_comb**
  - For modeling combinational logic
- **always_latch**
  - for modeling latched logic behavior

- **always_ff**
  - can be used to model synthesizable sequential logic behavior
  - imposes the restriction that it contains one and only one event control and no blocking timing controls.
  - Variables on the left-hand side of assignments within an always_ff procedure, including variables from the contents of a called function, shall not be written to by any other process.

Lecture 4: Sequential Coding

```
always clock = ~clock;          /* Deadlock Example */

always #10 clock = ~clock;      /* Correct Example */
```

```
                    /* Latch Example */
always_latch
   if( en ) q <= d;
```

```
                    /* Flip Flop Example */
always_ff @(posedge clock)
         q <= d;
```

Lecture 4: Sequential Coding

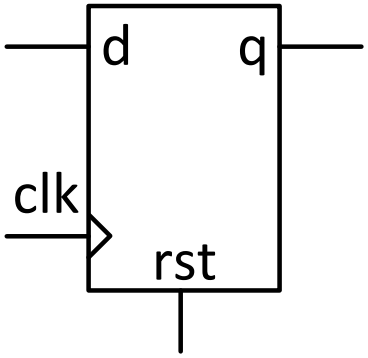
- **Sensitivity list**

  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.
  - The always keyword must be followed by an edge-sensitive event control (the @ token).
  - The sensitivity list of the event control cannot contain **posedge** or **negedge** qualifiers.

Lecture 4: Sequential Coding

- **Sensitivity list**

  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.

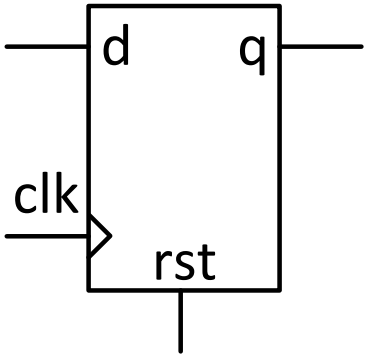


Lecture 4: Sequential Coding

- **Sensitivity list**

  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.

When does the output **q** change?



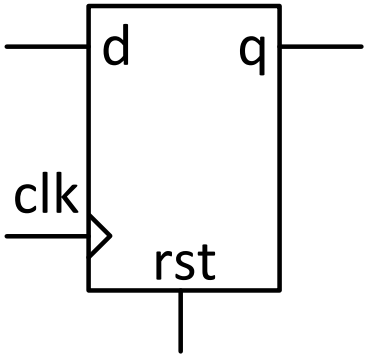
Lecture 4: Sequential Coding

- **Sensitivity list**

  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.

When does the output **q** change?

If **clk** is triggered and **rst** is asserted or **d** value has changed.

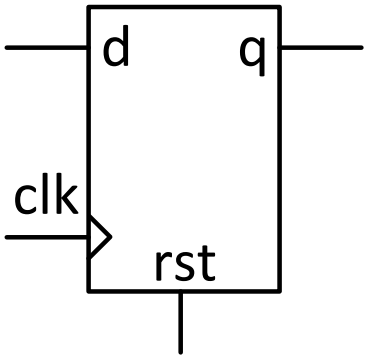


Lecture 4: Sequential Coding

- **Sensitivity list**

  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.

```
always_ff
begin
    if ( rst == 1'b1) begin
       q <= 1'b0;
    end else begin
       q <= d;
    end
end
```



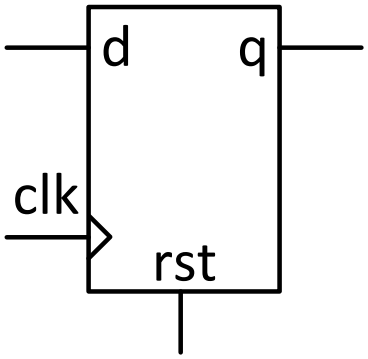
Lecture 4: Sequential Coding

- **Sensitivity list**

  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.

When does the following code should run to model a flip-flop?

```
always_ff
begin
    if ( rst == 1'b1) begin
       q <= 1'b0;
    end else begin
       q <= d;
    end
end
```



Lecture 4: Sequential Coding

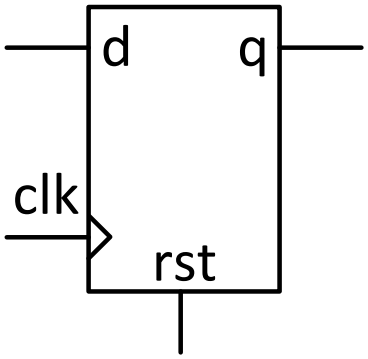
- **Sensitivity list**

  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.

When does the following code should run to model a flip-flop?

→ If **clk** is triggered and **rst** is asserted or **d** value has changed.

```
always_ff
begin
    if ( rst == 1'b1) begin
       q <= 1'b0;
    end else begin
       q <= d;
    end
end
```



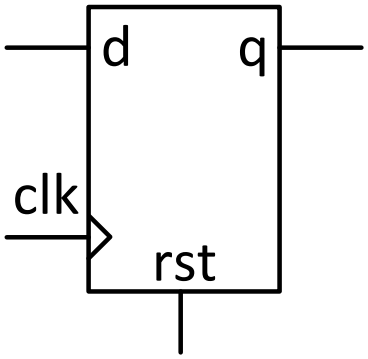
Lecture 4: Sequential Coding

- **Sensitivity list**
  A list of signals a process is sensitive to.
  - The sensitivity list is a compact way of specifying the set of signals, events on which may resume a process. A sensitivity list is specified right after the keyword **always** or **always_ff**.
  - Upon initialization, all always blocks are executed once
  - After that, always blocks are executed in a data-driven manner, they are activated by events on signals in the sensitivity list.

When does the following code should run to model a flip-flop?

→ So the always block should be sensitive to **clk** signal.

```
always_ff @(posedge clk)
begin
    if ( rst == 1'b1) begin
       q <= 1'b0;
    end else begin
       q <= d;
    end
end
```



Lecture 4: Blocking vs Non-blocking Assignment

Assignment statements

```
/* Example blocking vs non-blocking */

module nonblock1;
   logic a, b, c, d, e, f;

   // blocking assignments
   initial begin
      a = #10 1; // a will be assigned 1 at time 10
      b = #2 0;  // b will be assigned 0 at time 12
      c = #4 1;  // c will be assigned 1 at time 16
   end

   // nonblocking assignments
   initial begin
      d <= #10 1; // d will be assigned 1 at time 10
      e <= #2 0;  // e will be assigned 0 at time 2
      f <= #4 1;  // f will be assigned 1 at time 4
   end

endmodule
```

scheduled changes at time 2

e = 0

scheduled changes at time 4

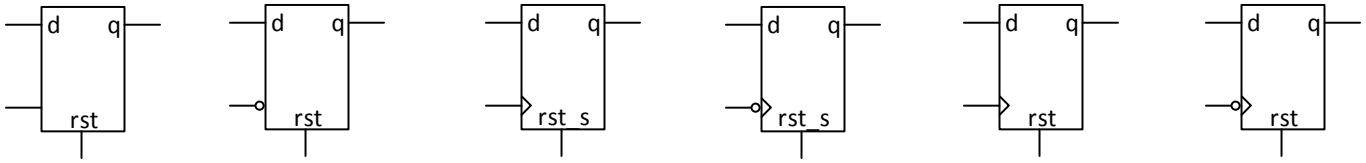
f = 1

scheduled changes at time 10

d = 1

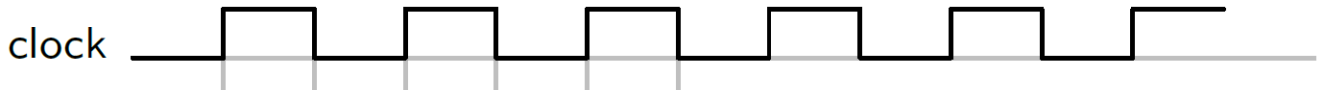
Lecture 4: Storage Elements

- **Latch**
  - **Positive-Level D Latch with asynchronous reset**
  - **Negative-Level D Latch with asynchronous reset**
- **Flip-Flop**
  - **Positive-Edge D Flip-Flop with synchronous reset**
  - **Negative-Edge D Flip-Flop with synchronous reset**
  - **Positive-Edge D Flip-Flop with asynchronous reset**
  - **Negative-Edge D Flip-Flop with asynchronous reset**



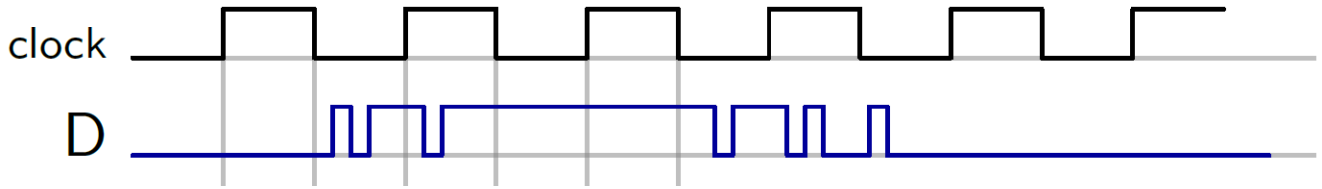
Lecture 4: Storage Elements

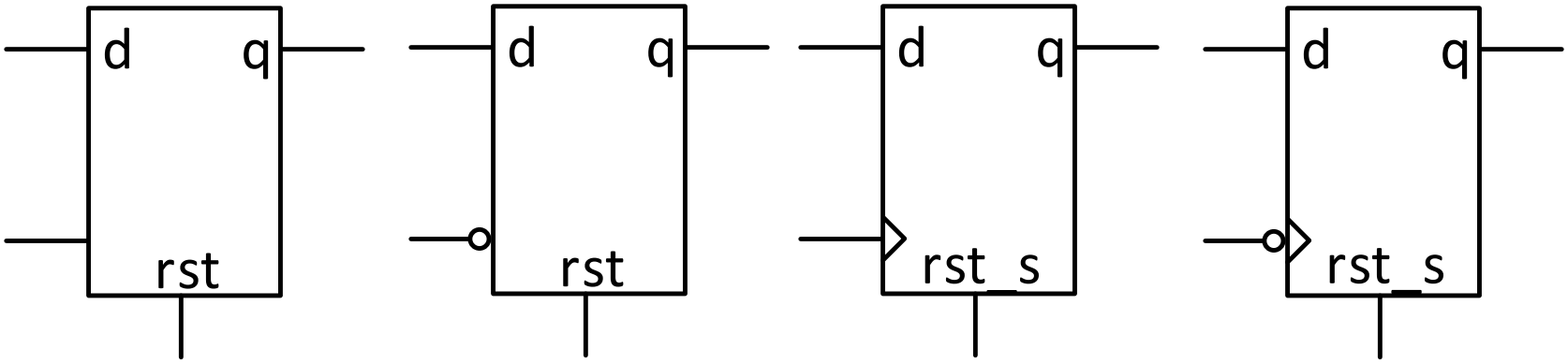
- **Clock**



Lecture 4: Storage Elements

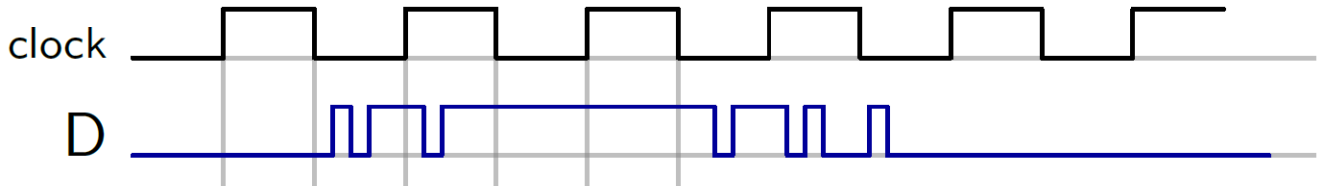
- **Clock**

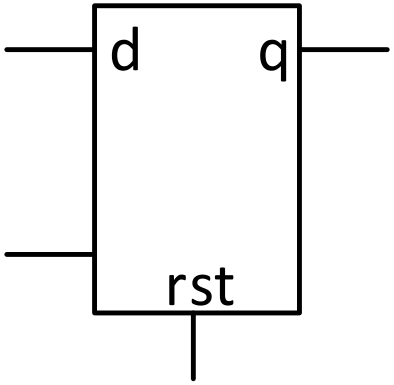




Lecture 4: Storage Elements

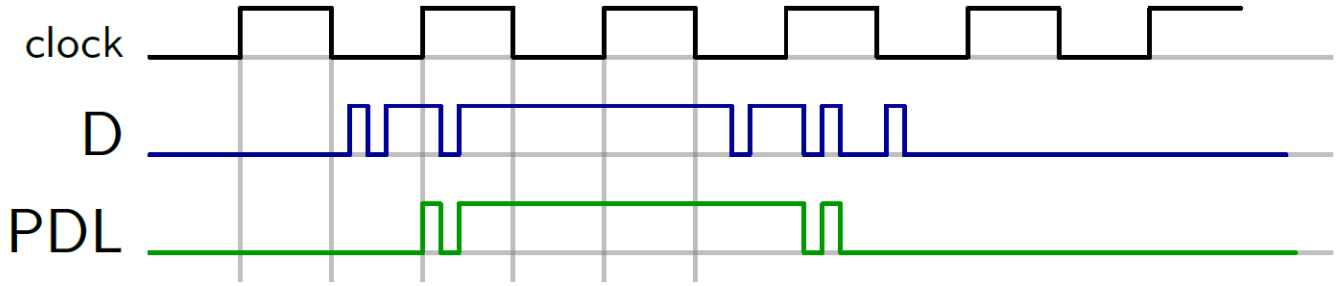
- **Clock**

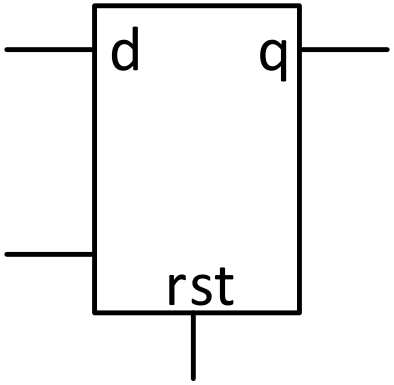




Lecture 4: Storage Elements

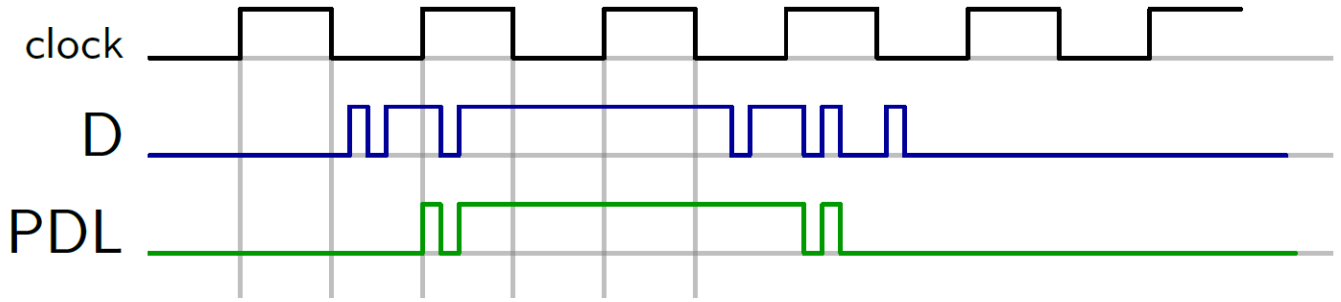
- **Clock**

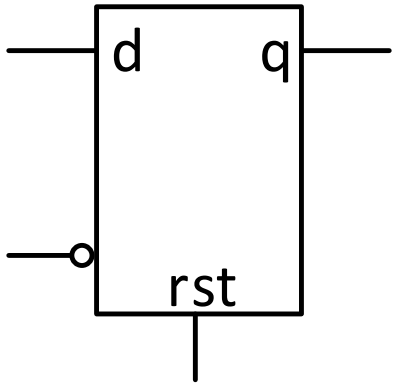




Lecture 4: Storage Elements

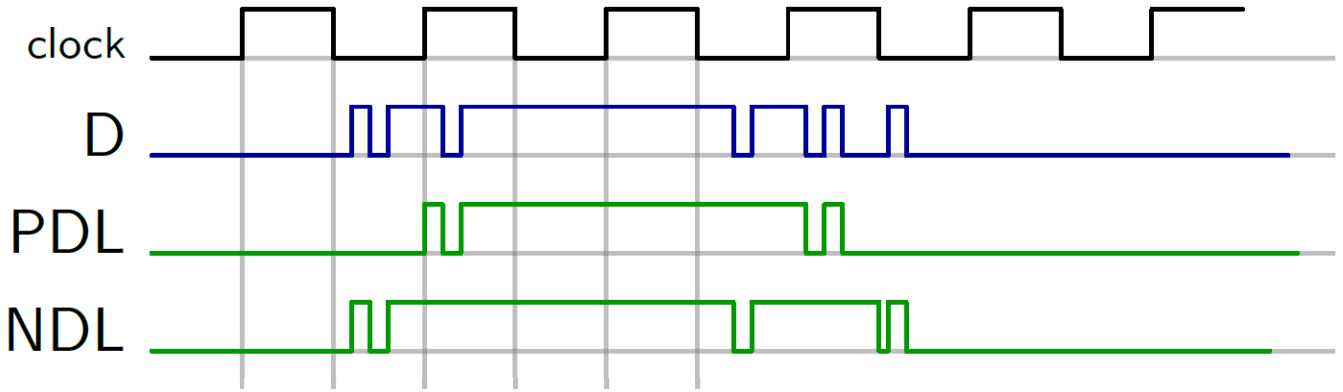
- **Clock**

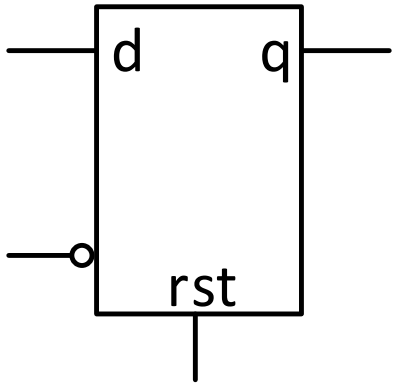




Lecture 4: Storage Elements

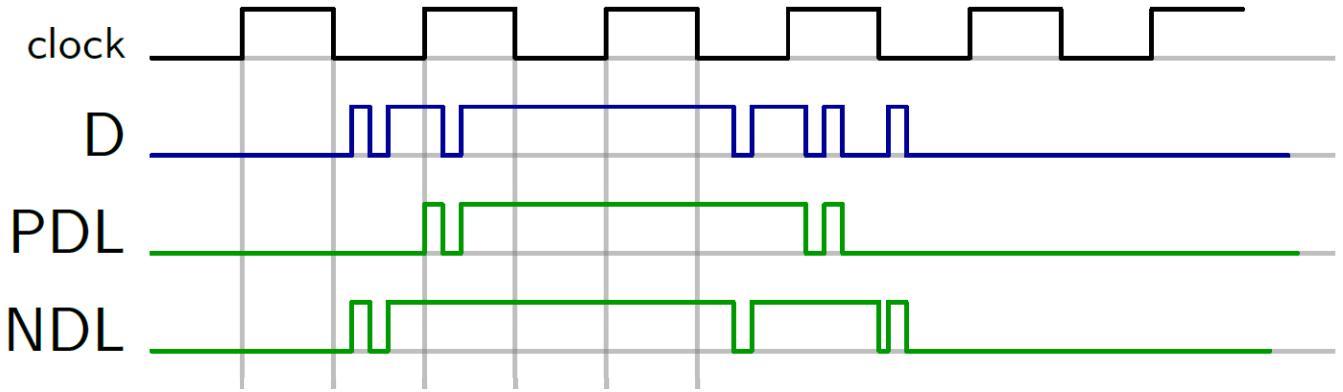
- **Clock**

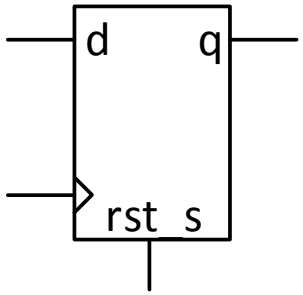




Lecture 4: Storage Elements

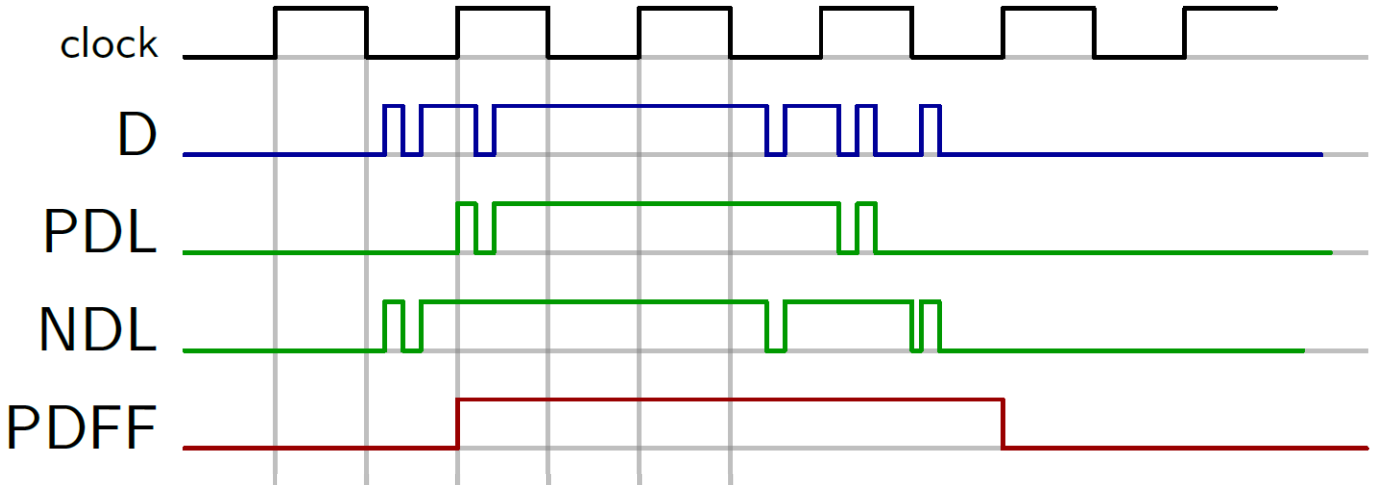
- **Clock**

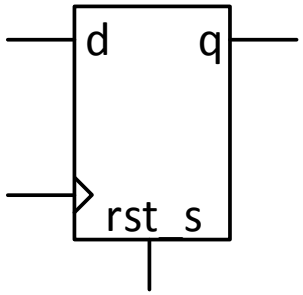




Lecture 4: Storage Elements

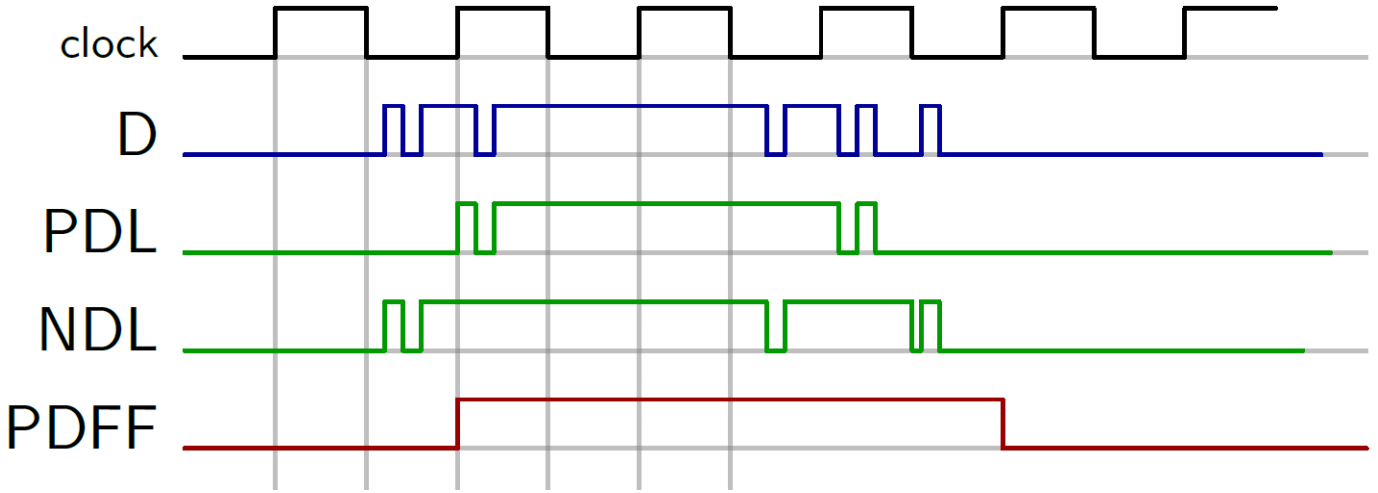
- **Clock**

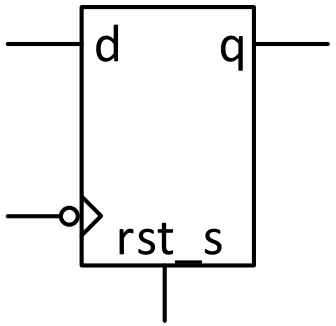




Lecture 4: Storage Elements

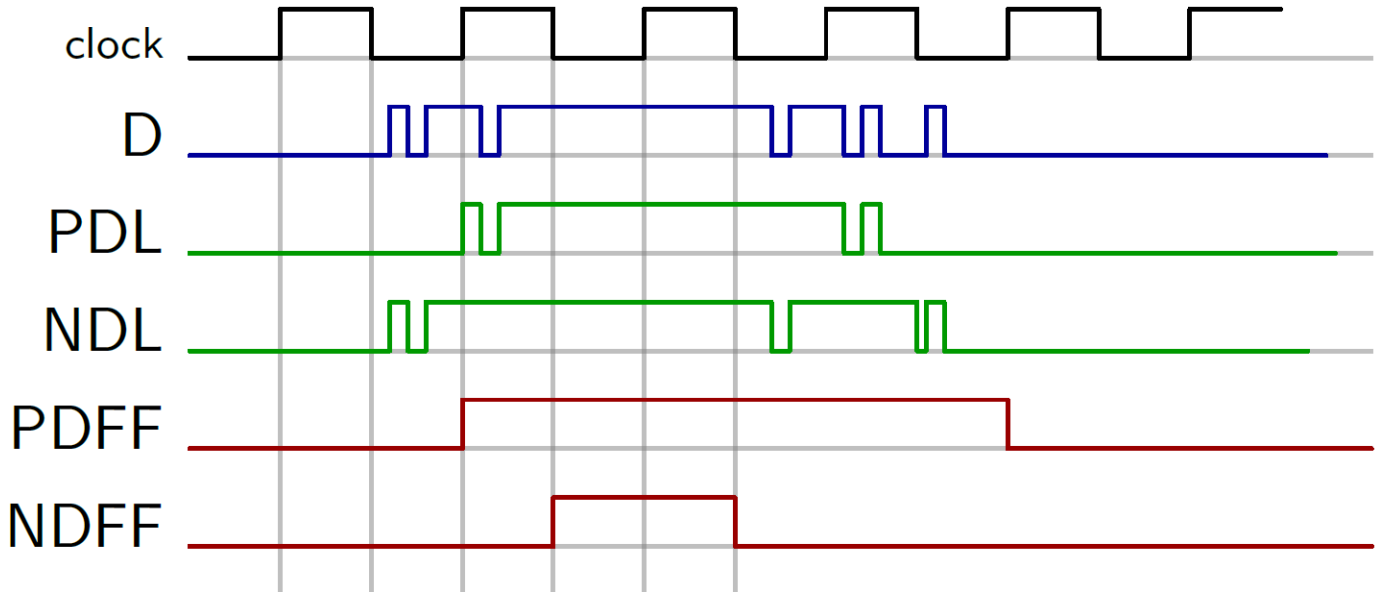
- **Clock**



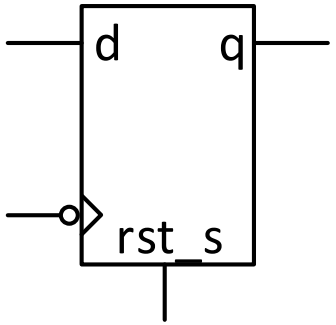


Lecture 4: Storage Elements

- **Clock**

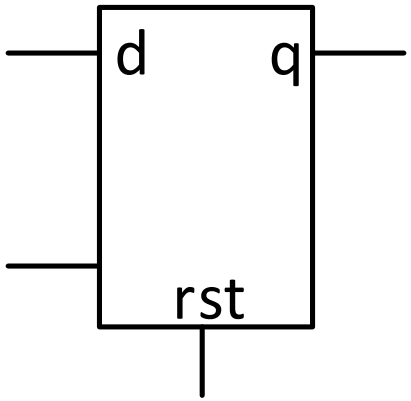


Is there anything wrong?



Lecture 4: Storage Elements

- **Latch**
  - **Positive-Level D Latch with asynchronous reset**

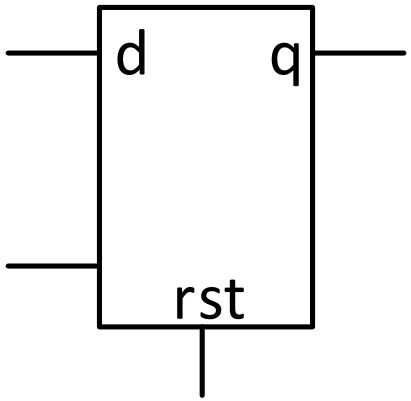



```
module PDL ( input d, input en, input rst;
             output logic q);

   always( ?  , ? ) begin
      if (? == '?1')
         ? <= '?';
      else if(? == '?')
         ? <= ?;

   end


endmodule
```

Lecture 4: Storage Elements

- **Latch**
  - **Positive-Level D Latch with asynchronous reset**

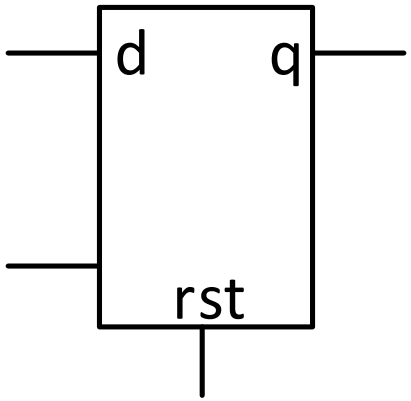
```
module PDL ( input d, input en, input rst;
             output logic q);

   always( rst  , en ) begin
      if ( rst == 1'b1)
         q <= 1'b0;
      else if( en == 1'b1 )
         q <= d;

   end


endmodule
```



Lecture 4: Storage Elements

- **Latch**
  - **Positive-Level D Latch with asynchronous reset**

```
module PDL ( input d, input en, input rst;
             output logic q);

   always( rst  , en ) begin
      if ( rst == 1'b1)
         q <= 1'b0;
      else if( en == 1'b1 )
         q <= d;

   end


endmodule
```

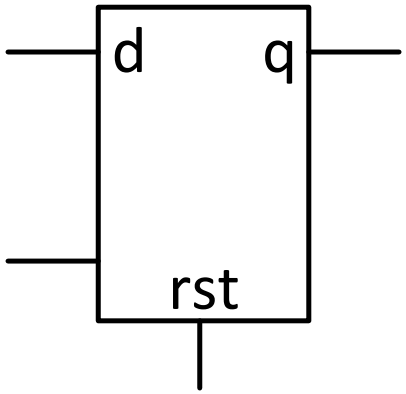


Is this implementation exact and correct?

Lecture 4: Storage Elements

- **Latch**
  - **Positive-Level D Latch with asynchronous reset**

```
module PDL ( input d, input en, input rst;
             output logic q);

   always( rst  , en ) begin
      if ( rst == 1'b1)
         q <= 1'b0;
      else if( en == 1'b1 )
         q <= d;

   end


endmodule
```

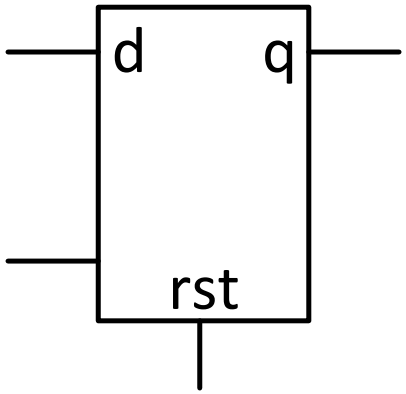


Is this implementation exact and correct?
NO !!!
What if **en** is 1'b1 and value of **d** changes?

Lecture 4: Storage Elements

- **Latch**
  - **Positive-Level D Latch with asynchronous reset**

```
module PDL ( input d, input en, input rst;
             output logic q);

   always( rst , en, d ) begin
      if ( rst == 1'b1)
         q <= 1'b0;
      else if( en == 1'b1 )
         q <= d;

   end


endmodule
```

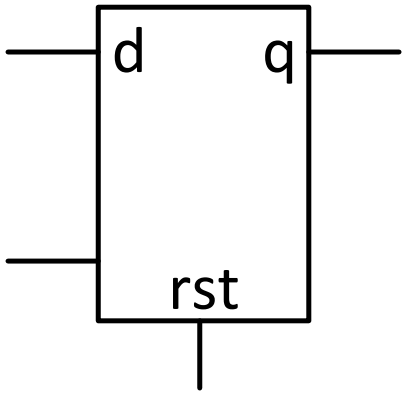


Add the **d** to the sensitivity list.

Lecture 4: Storage Elements

- **Latch**
  - **Positive-Level D Latch with asynchronous reset**

```
module PDL ( input d, input en, input rst;
             output logic q);

   always_latch begin
      if ( rst == 1'b1)
         q <= 1'b0;
      else if( en == 1'b1 )
         q <= d;

   end


endmodule
```

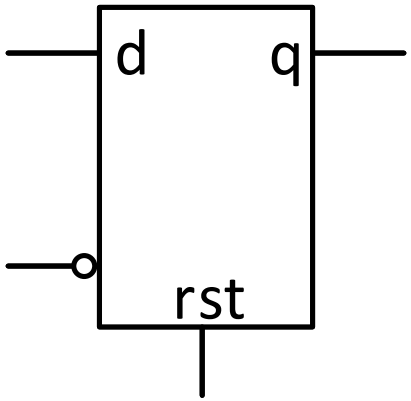


Or use **always_latch**

Lecture 4: Storage Elements

- **Latch**
  - **Negative-Level D Latch with asynchronous reset**

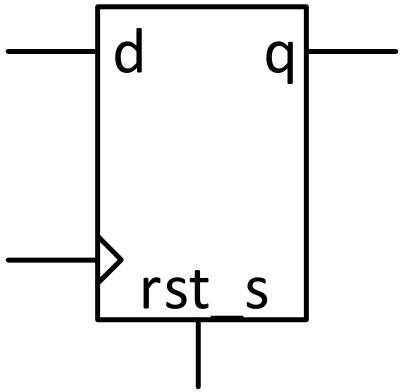
```
module PDL ( input d, input en, input rst;
             output logic q);

   always_latch begin
      if ( rst == 1'b1)
         q <= 1'b0;
      else if( en == 1'b0 )
         q <= d;

   end


endmodule
```



Lecture 4: Storage Elements

- **Flip-Flop**
  - **Positive-Edge D Flip-Flop with synchronous reset**

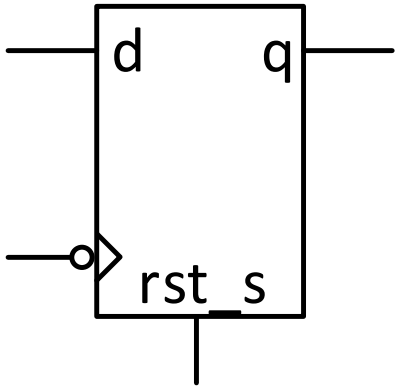
```
module PDFF ( input d, input clk, input rst;
              output logic q);

    always_ff @( posedge clk ) begin

        if ( rst == 1'b1) begin
           q <= 1'b0;
        end else begin
           q <= d;
        end

    end

endmodule
```



Lecture 4: Storage Elements

- **Flip-Flop**
  - **Negative-Edge D Flip-Flop with synchronous reset**

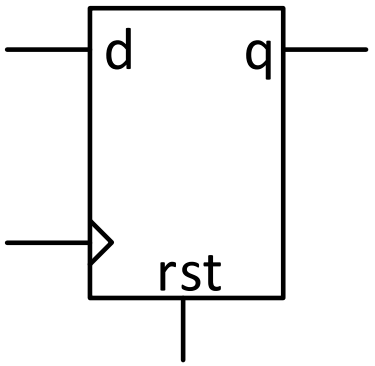
```
module NDFF( input d, input clk, input rst;
             output logic q);

    always_ff @( negedge clk ) begin

        if ( rst == 1'b1) begin
           q <= 1'b0;
        end else begin
           q <= d;
        end

    end

endmodule
```



Lecture 4: Storage Elements

- **Flip-Flop**
  - **Positive-Edge D Flip-Flop with asynchronous reset**

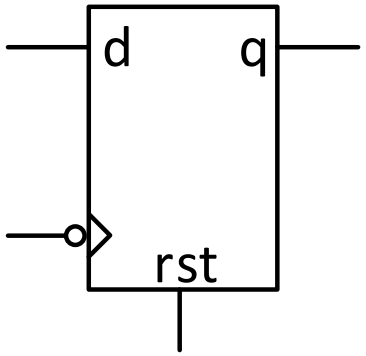
```
module PDFF ( input d, input clk, input rst;
              output logic q);

    always_ff @( posedge clk, rst ) begin

        if ( rst == 1'b1 ) begin
           q <= 1'b0;
        end else begin
           q <= d;
        end

    end

endmodule
```



Lecture 4: Storage Elements

- **Flip-Flop**
  - **Negative-Edge D Flip-Flop with asynchronous reset**

```
module PDFF ( input d, input clk, input rst;
              output logic q);

    always_ff @( negedge clk, rst ) begin

        if ( rst == 1'b1 ) begin
           q <= 1'b0;
        end else begin
           q <= d;
        end

    end

endmodule
```



Lecture 4: Sequential Coding

- Purely sequential statements:
  - **if**
  - **case**
  - **for**

Lecture 4: Sequential Coding

- **If** statement
  - The if statement is used to choose which statement should be executed depending on the conditional expression.

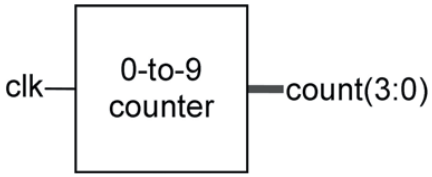
```
if (conditional expression) begin
  statement1;
  statement2;
end else if (conditional expression)
  statement3;
else
   statement4;
```

Example

```
always_comb
    if (a)
      b = 4;
    else if (d)
      b = 5;
    else
      b = 1;
```

Lecture 4: Sequential Coding

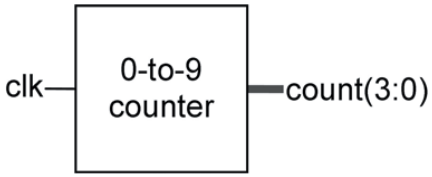
- Example: **1-digit Decimal Counter**




```
module counter (
         input logic clk,
         output logic [3:0]  count);









endmodule
```

Lecture 4: Sequential Coding

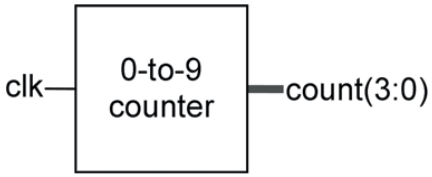
- Example: **1-digit Decimal Counter**




```
module counter (
         input logic clk,
         output logic [3:0]  count);

    logic [3:0] temp;

    always_ff @( posedge clk ) begin
         if ( temp == 10 ) begin
             temp <= 0;
         end else begin
             temp <= temp + 1;
         end
    end

    assign count <= temp;

endmodule
```

Lecture 4: Sequential Coding

- Example: **1-digit Decimal Counter**

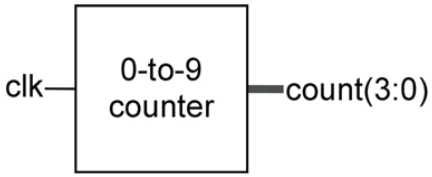



```
module counter (
         input logic clk,
         output logic [3:0]  count);

    logic [3:0] temp;

    always_ff @( posedge clk ) begin
         if ( temp == 10 ) begin
             temp <= 0;
         end else begin
             temp <= temp + 1;
         end
    end

    assign count <= temp;

endmodule
```

What happens if temp value is not zero?

Lecture 4: Sequential Coding

- Example: **1-digit Decimal Counter**

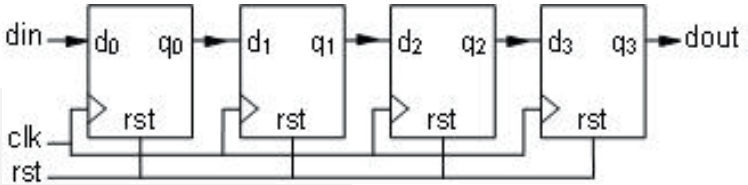



```
module counter (
         input logic clk,
         input logic reset,
         output logic [3:0]  count);

    logic [3:0] temp;

    always_ff @( posedge clk ) begin

        if ( reset ) begin
            temp <= 0;
        end else begin
            if ( temp == 10 ) begin
                temp <= 0;
            end else begin
                    temp <= temp + 1;
            end
        end
    end

    assign count <= temp;

endmodule
```

What happens if temp value is not zero? It is better to reset the internal variables first

Lecture 4: Sequential Coding

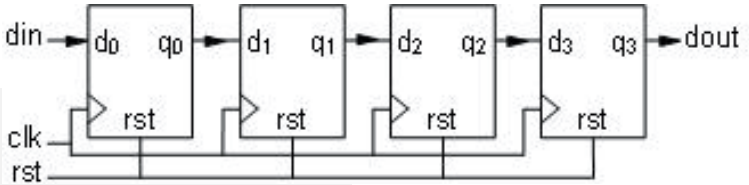
- Example: **Shift Register**




```
module shift_register
   #(N = 4); // number of stages
   ( input din, input clk, input rst;
     output dout);









endmodule
```

Lecture 4: Sequential Coding

- Example: **Shift Register**




```
module shift_register
   #(N = 4); // number of stages
   ( input din, input clk, input rst;
     output dout);

    logic [0:N-1] q;

    always_ff @( posedge clk, rst ) begin

        if ( rst ) begin
           q <= { N {1'b0} };
        end else begin
           q <= { din , q[0:N-2] };
        end
    end

    assign dout = q[N-1];

endmodule
```

Lecture 4: Sequential Coding

- **case** Statement

  The case statement selects for execution one of several alternative sequences of statements; the alternative is chosen based on the value of the associated expression.

```
case ( expression )
  expression : statement
  expression {, expression} : statement
  default : statement
endcase
```

Lecture 4: Sequential Coding

- **case** Statement

  The case statement selects for execution one of several alternative sequences of statements; the alternative is chosen based on the value of the associated expression.

```
case ( expression )
  expression : statement
  expression {, expression} : statement
  default : statement
endcase
```

Example

```
always_comb
    case (opcode)
        2'b00: y = a + b;
        2'b01: y = a - b;
        2'b10: y = a * b;
        2'b11: y = a / b;
    endcase
```

Lecture 4: Sequential Coding

Which statement will be executed if a == 1'bZ

```
logic a;
case ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
logic a;
casez ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
logic a;
casex ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

Lecture 4: Sequential Coding

Which statement will be executed if a == 1'bZ

```
logic a;
case ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
logic a;
casez ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

```
logic a;
casex ( a )
  1'b0 : statement1;
  1'b1 : statement2;
  1'bx : statement3;
  1'bz : statement4;
endcase
```

- The **casez** statement treats high-impedance (z) values as don't-care values and the **casex** statement treats high-impedance and unknown (x) values as don't care values. If any of the bits in the case expression or case item expression is a don't-care value then that bit position will be ignored.

- The don't-care value can be also specified by the question mark (?), which is equal to z value.

Lecture 4: Sequential Coding

The case expression can be a constant expression.

```
logic a;
case ( 1'b1 )
  a : statement1;
endcase
```

Lecture 4: Sequential Coding

SystemVerilog provides special unique and priority modifiers to case, casex, and casez decisions. These modifiers are placed before the case, casex, or casez keywords:

NOTE—By specifying unique or priority, it is not necessary to code a default case to trap unexpected case values.

```
unique case (<case_expression>)
         ... // case items
endcase



priority case (<case_expression>)
         ... // case items
endcase
```

Lecture 4: Sequential Coding

A unique case statement specifies that:

- ✓ Only one case select expression matches the case expression when it is evaluated
- ✓ One case select expression must match the case expression when it is evaluated

```
always_comb
    unique case (opcode)
        2'b00: y = a + b;
        2'b01: y = a - b;
        2'b10: y = a * b;
        2'b11: y = a / b;
    endcase
```

Lecture 4: Sequential Coding

A **priority case** statement specifies that:

- ✓ At least one case select expression must match the case expression when it is evaluated
- ✓ If more than one case select expression matches the case expression when it is evaluated, the first matching branch must be taken

```
always_comb
    priority case (1'b1)
        irq0: irq = 4'b0001;
        irq1: irq = 4'b0010;
        irq2: irq = 4'b0100;
        irq3: irq = 4'b1000;
    endcase
```

Lecture 4: Sequential Coding

Looping Statements

Loop statements provide a means of modeling blocks of procedural statements.

- **forever**
  - continuously repeats the statement that follows it. Therefore, it should be used with procedural timing controls (otherwise it hangs the simulation).
- **repeat**
  - executes a given statement a fixed number of times. The number of executions is set by the expression, which follows the repeat keyword. If the expression evaluates to unknown, high-impedance, or a zero value, then no statement will be executed.
- **while**
  - executes a given statement until the expression is true. If a while statement starts with a false value, then no statement will be executed.
- **for**
  - executes a given statement until the expression is true. At the initial step, the first assignment will be executed. At the second step, the expression will be evaluated. If the expression evaluates to an unknown, high-impedance, or zero value, then the for statement will be terminated. Otherwise, the statement and second assignment will be executed. After that, the second step is repeated.

Lecture 4: Sequential Coding

```
always begin
  counter = 0;
  forever #10 counter = counter + 1;
end
```

```
initial begin
  repeat (10) a = a + ~b;
end
```

```
initial begin
  for (index=0; index < 10; index = index + 2)
      mem[index] = index;
end
```

Lecture 4: Sequential Coding

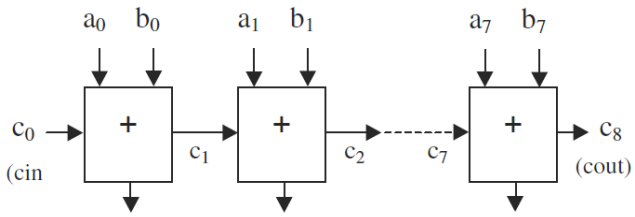
```
module test();
    parameter MSB = 8;
    logic [MSB-1:0] Vector;
    integer t;

    initial begin
      t = 0;

      while (t < MSB);
        begin
          //Initializes vector elements
          Vector[t] = 1'b0;
          t = t + 1;
        end
end
```

Lecture 4: Sequential Coding

Example: **Ripple Carry Adder**

$$s_j = a_j \text{ XOR } b_j \text{ XOR } c_j$$

$$c_{j+1} = (a_j \text{ AND } b_j) \text{ OR } (a_j \text{ AND } c_j) \text{ OR } (b_j \text{ AND } c_j)$$



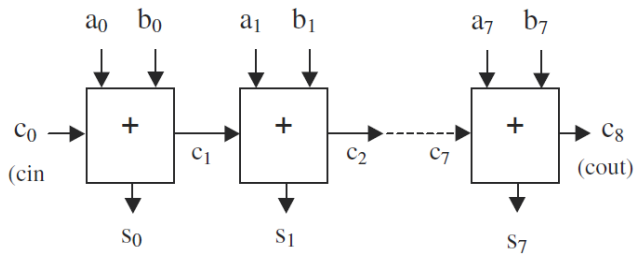

```
module adder
   #(length = 8);
   ( input  [length-1] a, input [length-1] b, input cin,
     output [length-1] s, output cout);









endmodule
```

Lecture 4: Sequential Coding

Example: **Ripple Carry Adder**

$$s_j = a_j \text{ XOR } b_j \text{ XOR } c_j$$

$$c_{j+1} = (a_j \text{ AND } b_j) \text{ OR } (a_j \text{ AND } c_j) \text{ OR } (b_j \text{ AND } c_j)$$

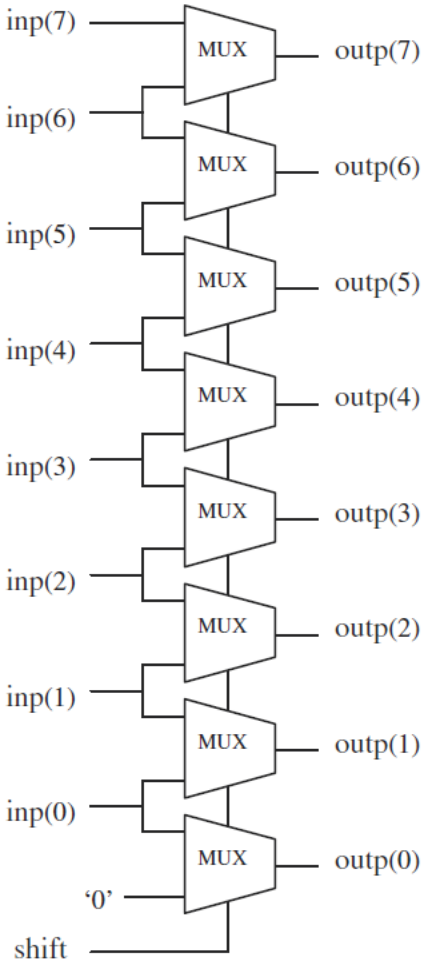



```
module adder
   #(length = 8);
   ( input  [length-1] a, input [length-1] b, input cin,
     output [length-1] s, output cout);

   always_comb begin
      logic [length:0] carry;
      carry[0] = cin;
      for ( int i=0; i< length; i= i + 1 ) begin
         s[i]       =  a[i] ^ b[i] ^ carry[i];
         carry[i+1] = (a[i] & b[i]) | (a[i] & carry[i]) | (b[i] & carry[i]);
      end
      cout = carry[length];
   end
endmodule
```

Lecture 4: Sequential Coding

- Example: **Barrel Shifter**

```
module barrel
   #(n = 8);
   (input  [n-1:0] inp,
    input          shift,
    output [n-1:0] outp );









endmodule
```



Lecture 4: Sequential Coding

- Example: **Barrel Shifter**

```
module barrel
   #(n = 8);
   (input  [n-1:0] inp,
    input          shift,
    output [n-1:0] outp );

   always (inp, shift) begin
      if ( shift == 0)
         outp <= inp;
      else begin
         outp[0] <= 1'b0;
         for( int i=1; i<n; i++ ) begin
            outp[i] <= inp[i-1];
         end
      end
   end
endmodule
```