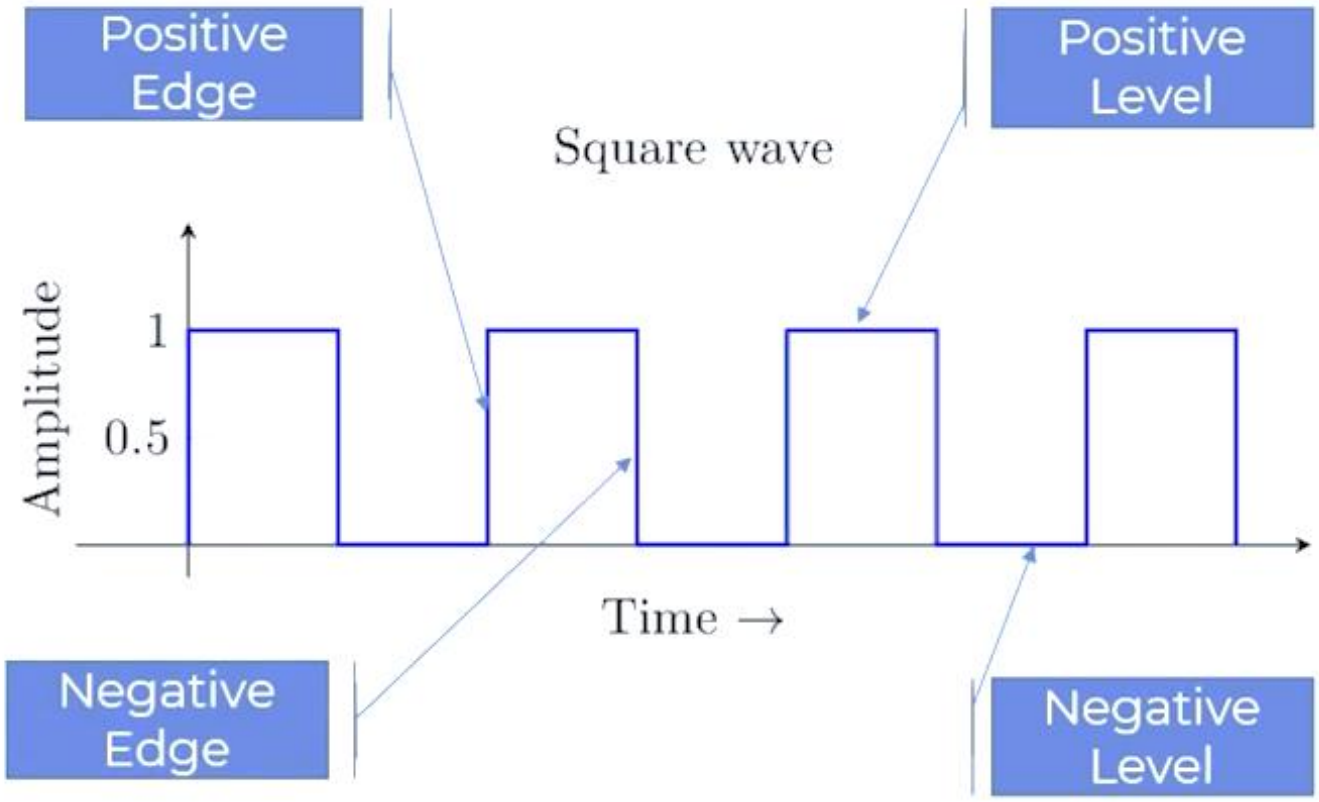
Verilog HDL:
Examples : Sequential Circuits

Pravin Zode

Outline

- D-Latch and D-FF
- Reset(Synchronous & Asynchronous)
- SR Latch
- JK Flip-Flop
- Shift Register
- Counters

Types of Triggering



D-Latch & D-FF

```
1   module latch (D, clk, Q);
2   input D, clk;
3   output reg Q;
4   always @(D, clk)
5   if (clk)
6       Q = D;
7   endmodule
```

D-Latch

```
1   module flipflop (D, Clock, Q);
2   input D, Clock;
3   output reg Q;
4   always @(posedge Clock)
5       Q <= D;
6   endmodule
```

D-FlipFlop

D-Latch & D-FF

```
1   module latch (D, clk, Q);
2   input D, clk;
3   output reg Q;
4   always @(D, clk)
5   if (clk)
6       Q = D;
7   endmodule
```

D-Latch

```
1   module flipflop (D, Clock, Q);
2   input D, Clock;
3   output reg Q;
4   always @(posedge Clock)
5       Q <= D;
6   endmodule
```

D-FlipFlop

Reset (Synchronous & Asynchronous)

Synchronous Reset (waits for clock edge)

```
always @(posedge clk) begin
    if (reset) q <= 0;
    else       q <= q + 1;
end
```

Resets q to 0 only on the rising edge of clk

Asynchronous Reset (immediate reset)

```
always @(posedge clk or posedge reset) begin
    if (reset)
        q <= 0;
    else
        q <= q + 1;
end
```

Resets q to 0 as soon as reset goes high, even if clock hasn't ticked

Flip-Flop Synchronous & Asynchronous Reset

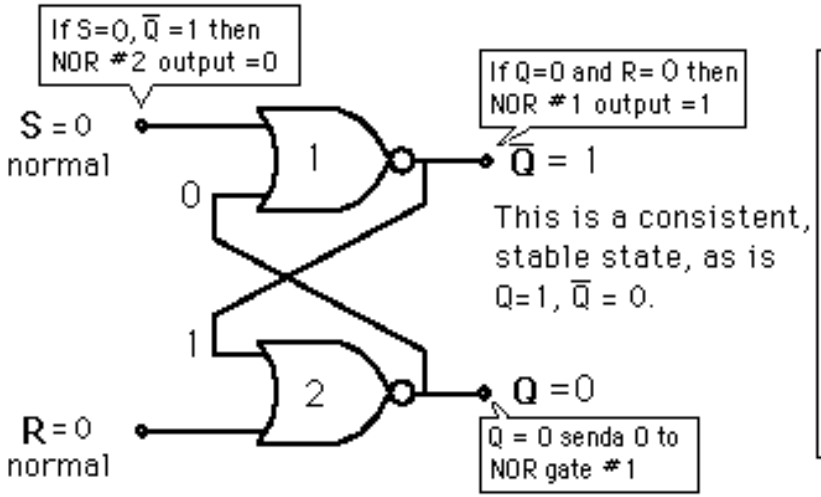
```
1   module flipflop_ar (D, Clock, Resetn, Q);
2   input D, Clock, Resetn;
3   output reg Q;
4   always @(posedge Clock, negedge Resetn)
5   if (Resetn == 0)
6       Q <= 0;
7   else
8       Q <= D;
9   endmodule
```

D flip-flop with asynchronous reset

D flip-flop with synchronous reset

```
1   module flipflop_sr (D, Clock, Resetn, Q);
2   input D, Clock, Resetn;
3   output reg Q;
4   always @(posedge Clock)
5   if (Resetn == 0)
6       Q <= 0;
7   else
8       Q <= D;
9   endmodule
```

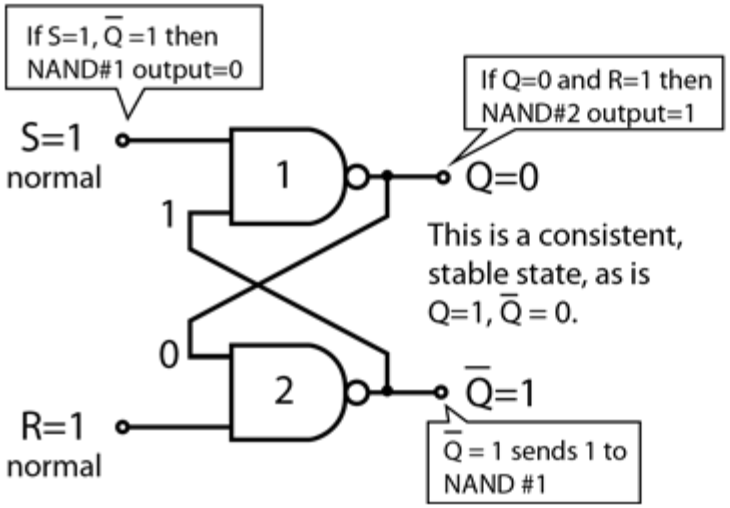
SR Latch



Truth Table

Set	Reset	Output
0	0	No change*
1	0	Q=1
0	1	Q=0
1	1	Invalid state

*can be used for data storage



Truth Table

Set	Reset	Output
1	1	No change*
0	1	Q=1
1	0	Q=0
0	0	Invalid state

* can be used for data storage

SR Latch

Dataflow Model

```
// SR Latch using NOR gates
module sr_latch (
    input S, // Set input
    input R, // Reset input
    output Q,// Normal output
    output Qbar // Complement
);

    // Cross-coupled NOR gates
    assign Q    = ~(R | Qbar);
    assign Qbar = ~(S | Q);

endmodule
```

Gate Level Model

```
// SR latch - gate-level
modeling using NOR primitives
module sr_latch_gatelevel (
    input  S,   // Set
    input  R,   // Reset
    output Q,   // Normal output
    output Qbar // Complement
);

 nor u1(Q,R, Qbar);
 nor u2(Qbar, S, Q);

endmodule
```

SR Flip-Flop (version-1)

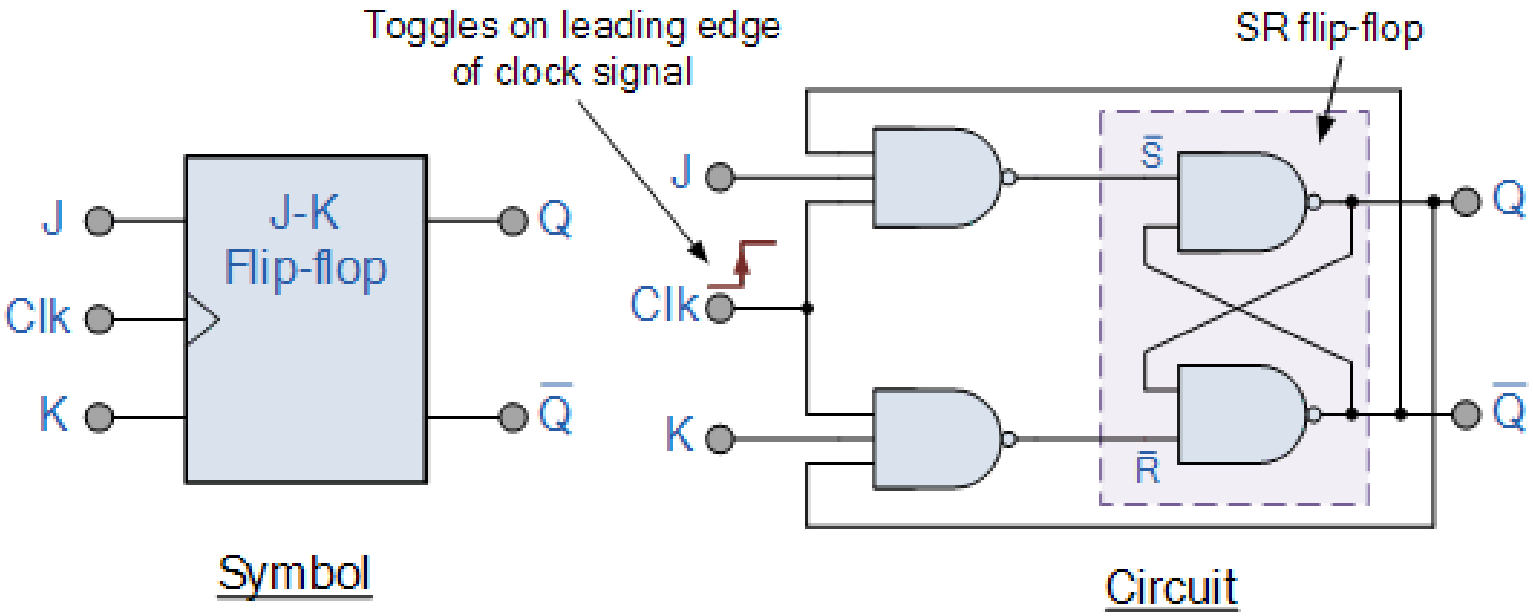
```
// Clocked SR FF (Behavioral using if-else with explicit condn)
module sr_ff_ifelse (
    input  clk,   // Clock input
    input  S,     // Set input
    input  R,     // Reset input
    output reg Q, // Normal output
    output reg Qbar // Complement output
);
    always @(posedge clk) begin
        if (S==1 && R==0)
            Q <= 1;           // Set
        else if (S==0 && R==1)
            Q <= 0;           // Reset
        else if (S==0 && R==0)
            Q <= Q;           // Hold (no change)
        else if (S==1 && R==1)
            Q <= 1'bx;        // Invalid case
    end
    assign Qbar = ~Q;
endmodule
```

SR Flip-Flop (version-2)

```
// Clocked SR Flip-Flop (Behavioral using if-else)
module sr_ff_ifelse (
    input  clk,   // Clock input
    input  S,     // Set input
    input  R,     // Reset input
    output reg Q, // Normal output
    output reg Qbar // Complement output
);
    always @(posedge clk) begin
        if (S && ~R)
            Q <= 1;           // Set
        else if (~S && R)
            Q <= 0;           // Reset
        else if (~S && ~R)
            Q <= Q;           // Hold (no change)
        else
            Q <= 1'bx;        // Invalid case (S=R=1)
    end
    assign Qbar = ~Q;
endmodule
```

JK Flip-Flop (version-1)

- It is modified version of SR FF invented by Jack Kilby from Texas Instruments



JK Flip-Flop Truth Table

	Clock	Input		Output		Description
	Clk	J	K	Q	Q	
same as for the SR Latch	X	0	0	1	0	Memory no change
	X	0	0	0	1	
	⎺↓_	0	1	1	0	Reset Q » 0
	X	0	1	0	1	
	⎺↓_	1	0	0	1	Set Q » 1
	X	1	0	1	0	
toggle action	⎺↓_	1	1	0	1	Toggle
	⎺↓_	1	1	1	0	

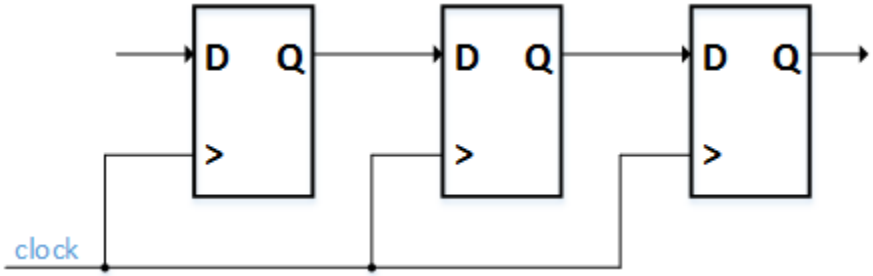
JK Flip-Flop Verilog Code (Version-1)

```
// JK Flip-Flop (Behavioral, using if-else)
module jk_ff (
    input  clk,   // Clock input
    input  J,     // J input
    input  K,     // K input
    output reg Q, // Normal output
    output    Qbar // Complement output
);
    always @(posedge clk) begin
        if (J==0 && K==0)
            Q <= Q;           // Hold (no change)
        else if (J==0 && K==1)
            Q <= 0;           // Reset
        else if (J==1 && K==0)
            Q <= 1;           // Set
        else if (J==1 && K==1)
            Q <= ~Q;          // Toggle
    end
    assign Qbar = ~Q;
endmodule
```

JK Flip-Flop Verilog Code (Version-2)

```
// JK Flip-Flop (Behavioral, using case statement)
module jk_ff_case (
    input  clk,    // Clock input
    input  J,      // J input
    input  K,      // K input
    output reg Q,  // Normal output
    output    Qbar // Complement output
);
    always @(posedge clk) begin
        case ({J,K})
            2'b00: Q <= Q;      // Hold (no change)
            2'b01: Q <= 1'b0;   // Reset
            2'b10: Q <= 1'b1;   // Set
            2'b11: Q <= ~Q;     // Toggle
        endcase
    end
    assign Qbar = ~Q;
endmodule
```

Shift Register




```
1   module shift3 (w, Clock, Q);
2   input w, Clock;
3   output reg [1:3] Q;
4   always @(posedge Clock)
5   begin
6       Q[3] <= w;
7       Q[2] <= Q[3];
8       Q[1] <= Q[2];
9   end
10  endmodule
```

Three-bit shift register

```
1   module shift3 (w, Clock, Q);
2   input w, Clock;
3   output reg [1:3] Q;
4   always @(posedge Clock)
5   begin
6       Q[3] = w;
7       Q[2] = Q[3];
8       Q[1] = Q[2];
9   end
10  endmodule
```

Wrong code for a three-bit shift register

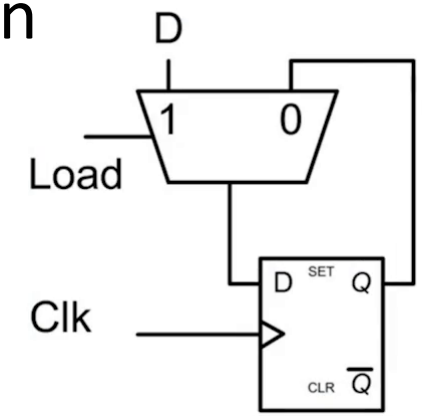
Shift Register

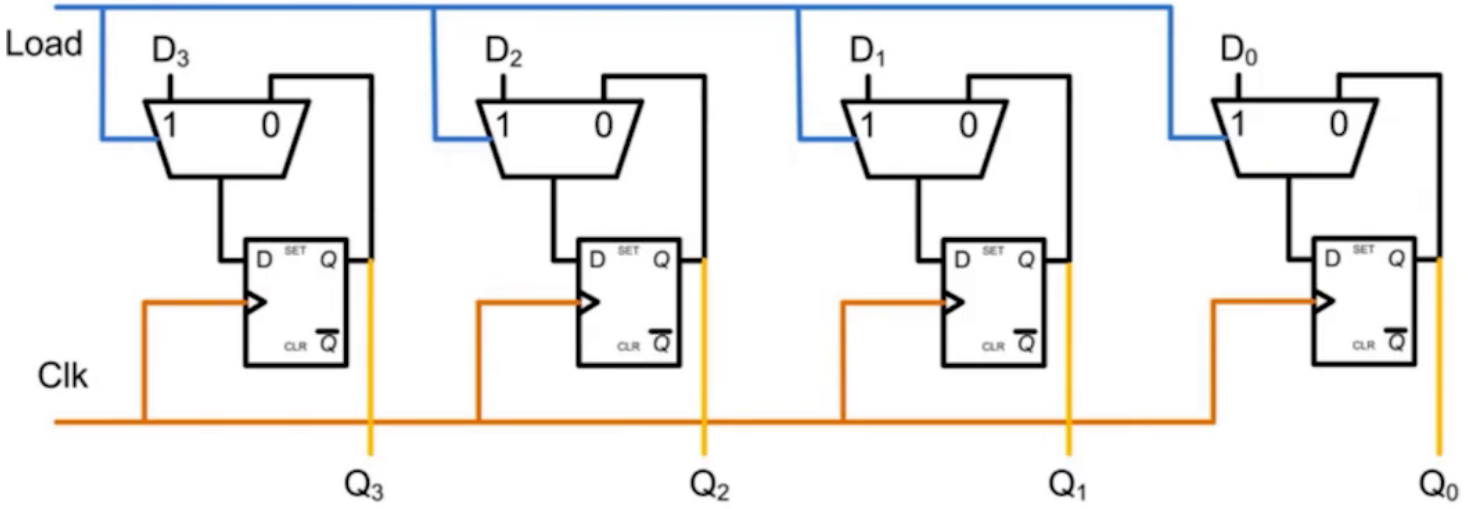
```
1   module shiftreg_4bit (clock, clear, A, E);
2   input clock, clear, A;
3   output reg E;
4   reg B, C, D;
5   always @(posedge clock or negedge clear)
6   begin
7   if (!clear)
8   begin
9       B<=0; C<=0; D<=0; E<=0;
10  end
11  else
12  begin
13      E <= D;
14      D <= C;
15      C <= B;
16      B <= A;
17  end
18  end
19  endmodule
```

```
1   module shiftreg_4bit_tb;
2   reg clk, clr, in; wire out; integer i;
3   shiftreg_4bit SR (clk, clr, in, out);
4   initial
5   begin clk = 1'b0; #2 clr = 0; #5 clr = 1;
6   end
7   always #5 clk = ~clk;
8   initial begin #2;
9   repeat (2)
10  begin #10 in=0; #10 in=0; #10 in=1; #10 in=1;
11  end
12  end
13  initial
14  begin
15  $dumpfile ("shifter.vcd");
16  $dumpvars (0, shift_test);
17  #200 $finish;
18  end
19  endmodule
```

Shift Register (Loadable)

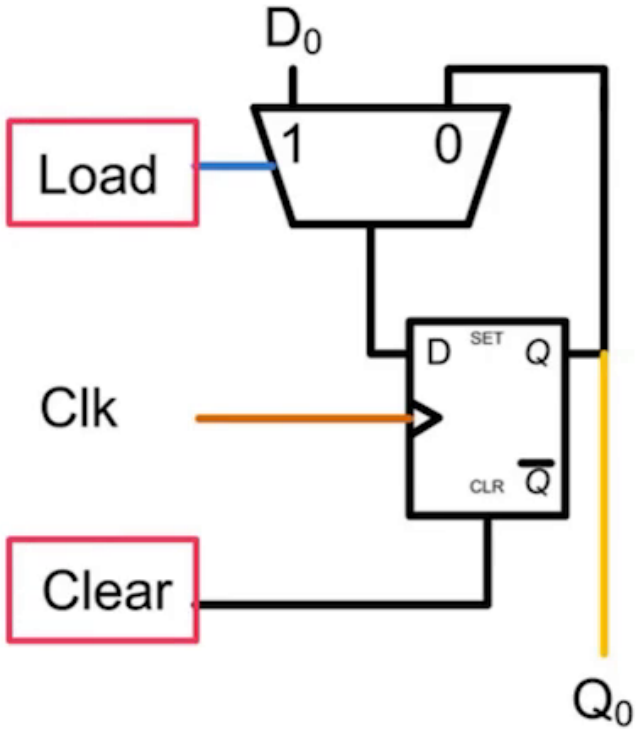
- Load signal is used to decide the operation
- If load =0, it holds
- If load =1, it load new value





Shift Register (Clear & Loadable)

- load and clear decides the operation
- if clear is 1, load is not considered, and output is 0
- if load =0, it holds
- if load =1, it loads new value

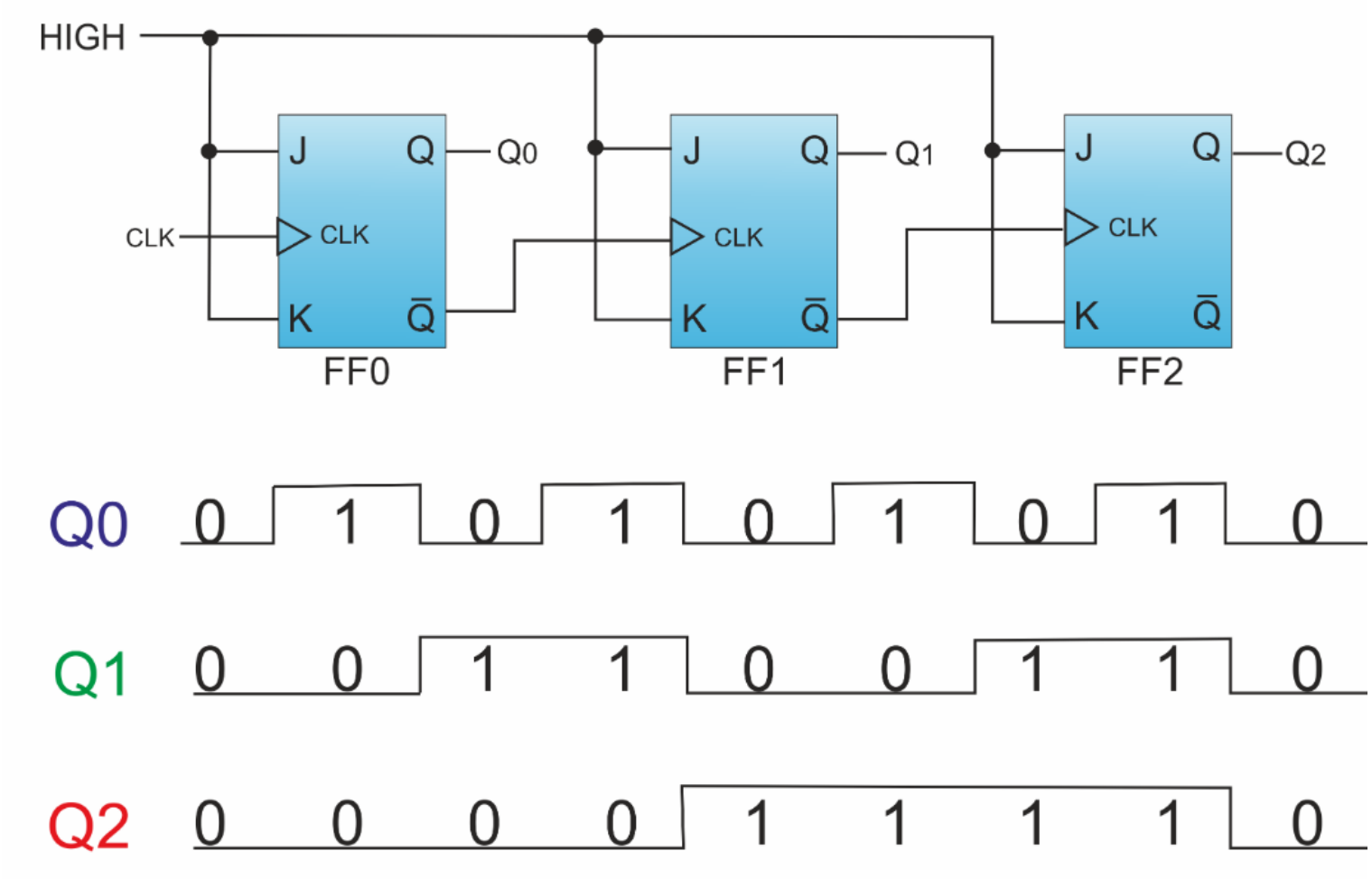


Extend this to 4-bit shift register

Counter

- **Purpose of Counters**
  - Used to count various operations and events
  - Store and display the number of occurrences and repetitions
- **Types of Counters**
  - Synchronous Counters – all flip-flops triggered by the same clock
  - Asynchronous Counters – flip-flops triggered in sequence
- **Implementation**
  - Mostly designed using JK Flip-Flops (JK FFs)

Asynchronous Counter



Asynchronous Up-Counter

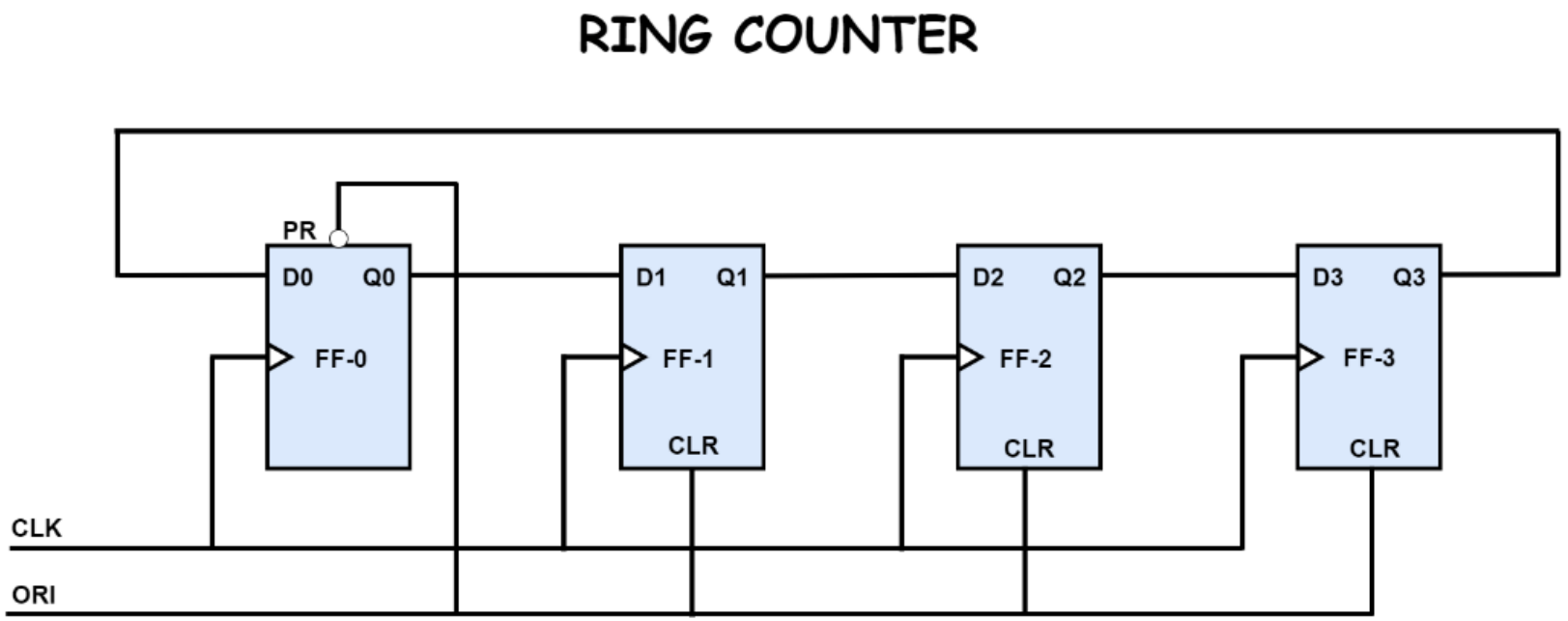
```
1   module async_counter(
2       input clk,
3     output reg [3:0] count );
4
5       initial count = 0;
6
7       always @(posedge clk) begin
8           count[0] <= ~count[0];
9       end
10
11      always @(posedge count[0]) begin
12          count[1] <= ~count[1];
13      end
14
15      always @(posedge count[1]) begin
16          count[2] <= ~count[2];
17      end
18
19      always @(posedge count[2]) begin
20          count[3] <= ~count[3];
21      end
22
23  endmodule
```

Pattern Counter-Ring Counter

- A circular shift register where the output of the last flip-flop is fed back to the first.
  - Requires n flip-flops for n states.
  - Example: 4-bit Ring Counter → 4 states
- 1000 → 0100 → 0010 → 0001 → 1000 ...
- Applications:
  - Sequence generation
  - Timing signals
  - Control logic

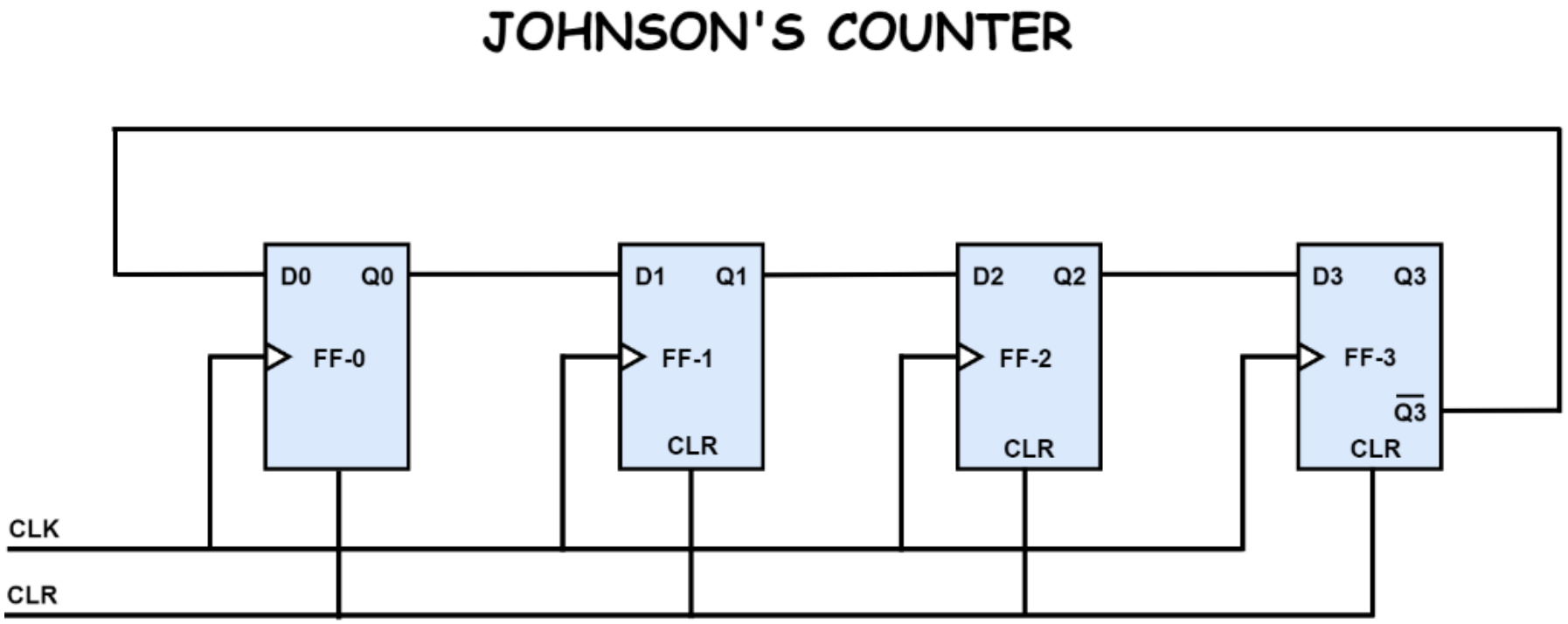
Pattern Counter-Ring Counter

- The ring counter is application of shift register, in which the output of last flip flop is connected to input of first flip flop
- In ring counter if the output of any flip flop is 1, then the output of remaining flip flops is 0.



Pattern Counter-Johnson Counter

A Johnson counter is a type of shift register counter where the inverted output of the last flip-flop is fed back to the input of the first flip-flop (Twisted Ring Counter)



Pattern Counter-Johnson Counter

Clock no.	D3	D2	D1	D0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1
8	0	0	0	0
9	1	0	0	0

```
1   module Counter_4_Johnson(clk,reset,count);
2   input clk,reset;
3   output reg[3:0]count;
4   reg [3:0]temp;
5   always @(posedge clk or reset)
6   begin
7   if (reset==1)
8       temp =4'b0000;
9   else
10  begin
11      temp = {~temp[0],temp[3:1]};
12  end
13  assign count =temp;
14  end
15  endmodule
```

Counter

```
1   module counter (clear, clock, count);
2   parameter N = 7;
3   input clear, clock;
4   output reg [0:N] count;
5   always @(negedge clock)
6   if (clear)
7       count <= 0;
8   else
9       count <= count + 1;
10  endmodule
```

```
1   module test_counter;
2   reg clk, clr;
3   wire [7:0] out;
4   counter CNT (clr, clk, out);
5   initial clk = 1'b0;
6   always #5 clk = ~clk;
7   initial
8   begin
9   clr = 1'b1;
10  #15 clr = 1'b0;
11  #200 clr = 1'b1;
12  #10 $finish;
13  end
14  initial
15  begin
16  $dumpfile ("counter.vcd");
17  $dumpvars (0, test_counter);
18  $monitor ($time, " Count: %d", out);
19  end
20  endmodule
```

Clock Divider

LED Blinking Example

```
1   module ledblink(clk,led);
2   input clk; output led; reg led;
3
4   reg[23:0] cnt;
5
6   always @(posedge clk)
7   begin
8       cnt<= cnt + 1'b1;
9       led<=cnt[23];
10  end
11  endmodule
```

Thank you !

Happy Learning