שפת תכנון חומרה
ורילוג - Verilog

Verilog – Counters & Shift registers

Dr. Avihai Aharon

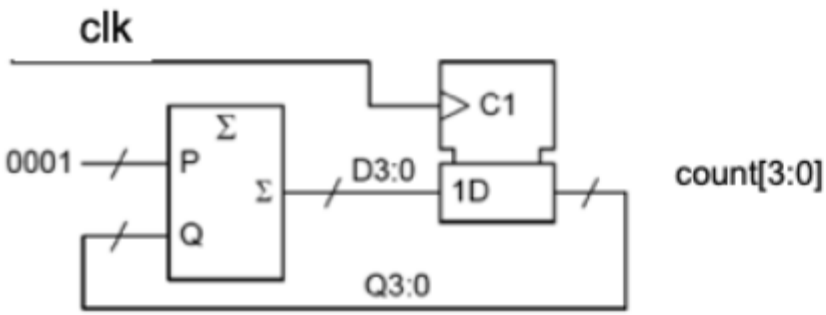
Lecture Objectives

- Understand how digital systems may be divided into datapath and control logic
- Appreciate the different ways of implementing control logic
- Understand how shift registers and counters can be used to generate arbitrary pulse sequences
- Understand the circumstances that give rise to output glitches
- Able to design various types of counters and timers

Control Logic

- Most digital systems can be divided into
  Data Path Blocks: adder, subtractor, ALU, floating point unit etc
  Memory Blocks: RAM, ROM, registers etc for storing data or instructions
  Control Logic Blocks: generates timing signals at the right time and in the right order
- Control logic can be implemented with:
  - **Microprocessor/Microcontroller**
    - + Cheap, very flexible, design easy (software)
    - – Slow: most actions require >20 instructions = 2 μs @ clock speed of 10 MHz
    - Use for slow applications
  - **Synchronous State Machine**
    - + Fast (20 ns/action), Cheap using programmable logic
    - – Hard to design complex systems. Limited data storage
    - Use for fast, moderately complex systems
  - **Counters/Shift Registers**
    - + Fast, Cheap, Very easy design
    - – Simple systems only
    - A special case of synchronous state machines
    - Use for very simple systems (fast or slow)

Synchronous Counters

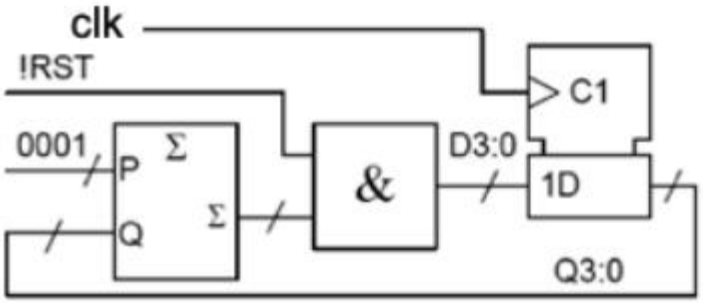


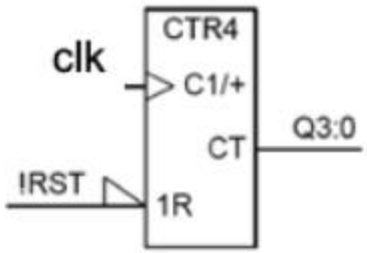

```
module counter  (count, clk);

    output  [3:0]   count;  // 4-bit count value
    input           clk;    // clock

    reg     [3:0]   count;
    always @ (posedge clk)
          count <= count + 1'b1;
endmodule
```

- An N bit binary counter has a cycle length of 2^N states. We can draw a state diagram in which one transition is made for each clock

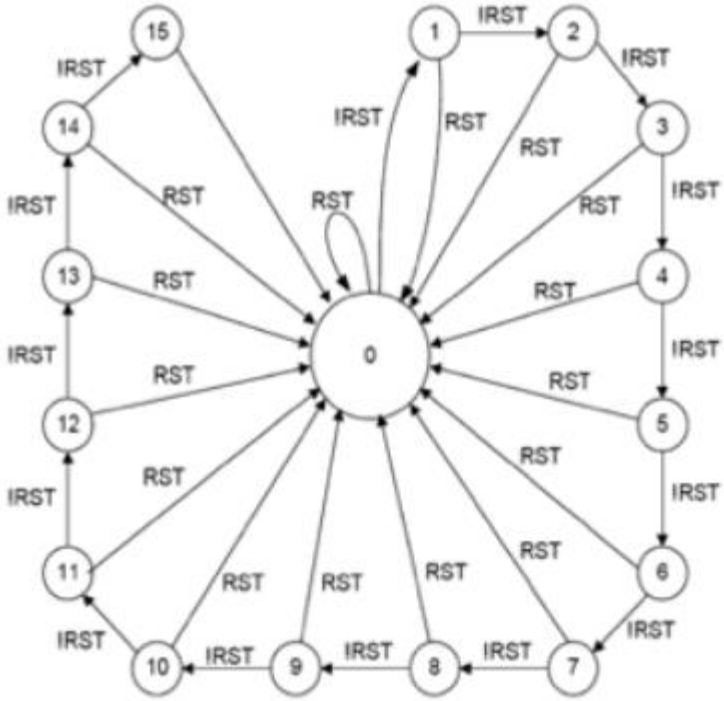
- Adder can be simplified: one set of inputs is fixed so many gates can be eliminated. For example:

Synchronous counters with synchronous reset



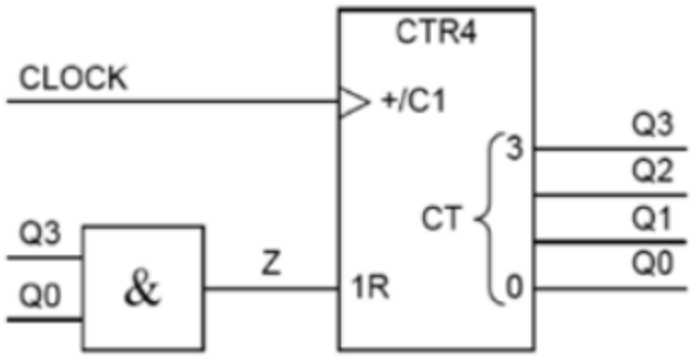



```
reg      [3:0]   count;
always @ (posedge clk)
  if (rst == 0)
      count <= 1'b0;
  else
      count <= count + 1'b1;
```



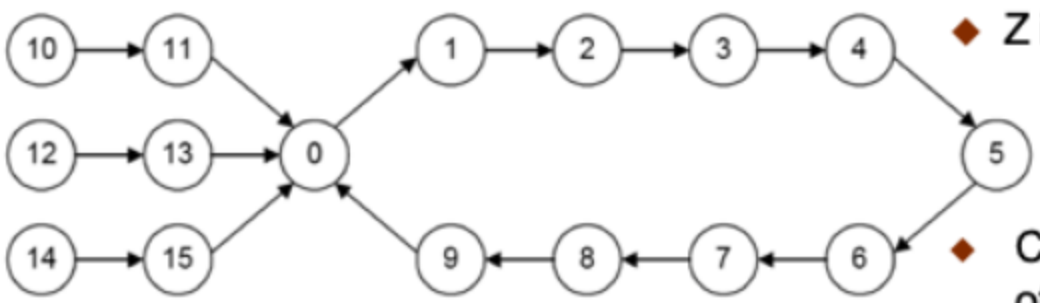
- This is a ***synchronous*** reset input: taking **!RST** low has no effect until the next clock
- In a synchronous counter everything is done by manipulating the D inputs of the register

Decade counter




```
module counter_div10 (count, clk);

   output [3:0]   count;    // count 0 to 9
   input          clk;      // clock input

   reg  [3:0]     count;    // need this declaration
   always @ (posedge clk)
       if (count == 4'd9)
            count <= 4'b0;
       else
            count <= count + 1'b1;
endmodule
```

<u>Notation:</u> CT = Contents
0 = least significant bit (LSB)
Bit k has a binary weight of 2^k
1R means reset on next C1 (CLOCK edge)

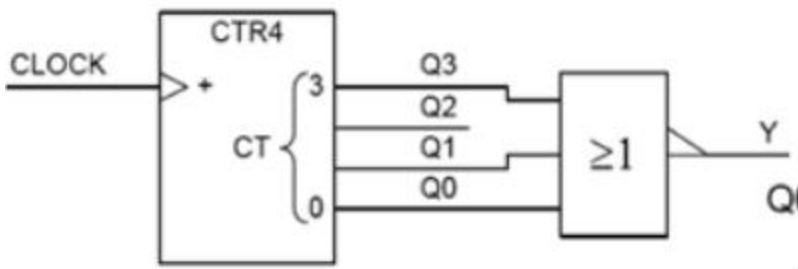


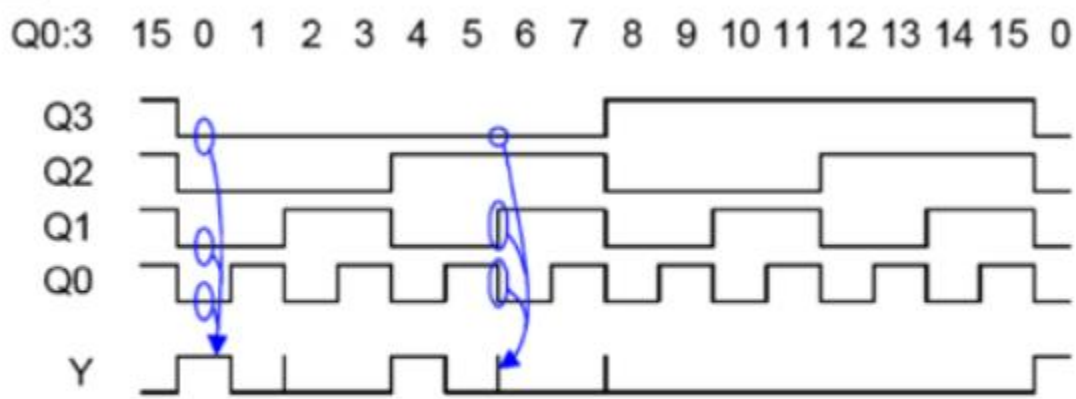
- Z is high whenever Q3:0 = 1??1 1001 = 9
  1011 = 11
  1101 = 13
  1111 = 15

  lowest value is when all the ? bits are zero
- Counter resets after 9 giving a cycle length of 10 states

Output Glitches

- If k counter bits change "simultaneously", other logic circuits using them may briefly see any of 2^k possible values
- Glitches are possible at the logic circuit output if:
  1. These 2^k values include any that would cause the logic circuit output to change
  2. The logic circuit output is meant to remain at a constant value



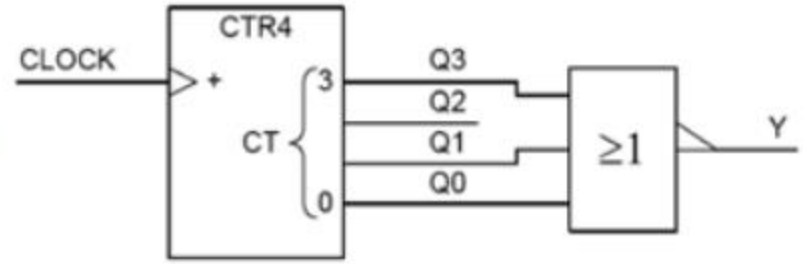


Y is high when Q=0000 or 0100

- Transition 1 → 2: Q=00?? which includes 0000
- Transition 5 → 6: Q=01?? which includes 0100
- Transition 7 → 8: Q=???? which includes both

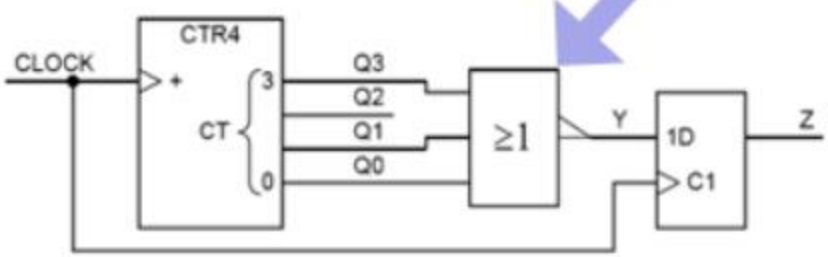
Eliminating Output Glitches

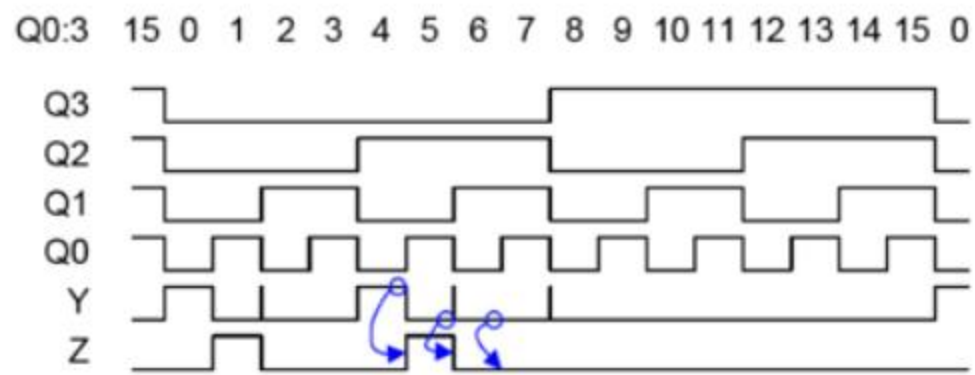
```
module detect0and4 (y, clk);

    output  y;        // high for count 0 and 4
    input   clk;

    reg [3:0]   q;
    initial
        q = 4'b0;
    always @ (posedge clk)
        q <= q + 1'b1;

    wire y;
    assign y = ~(q[0]+q[1]+q[3]);
endmodule
```




```
reg y;
always @ (posedge clk)
  begin
    q <= q + 1'b1;
    y <= ~(q[0]+q[1]+q[3]);
  end
```

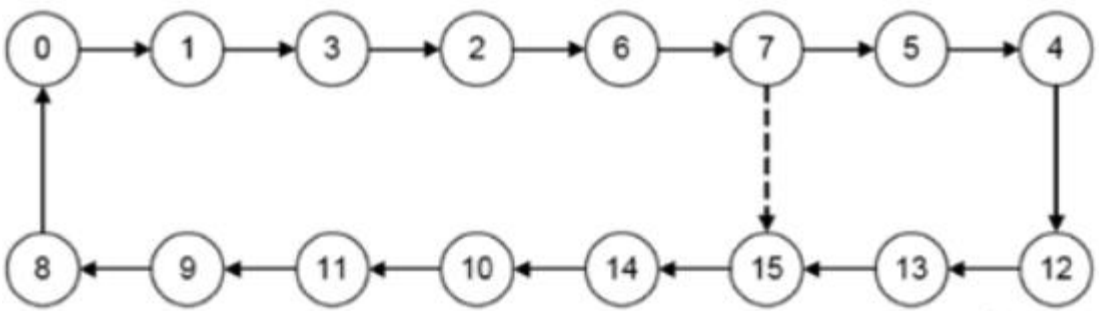
- We can elminate output glitches by delaying Y with a flipflop:





Gray code counter

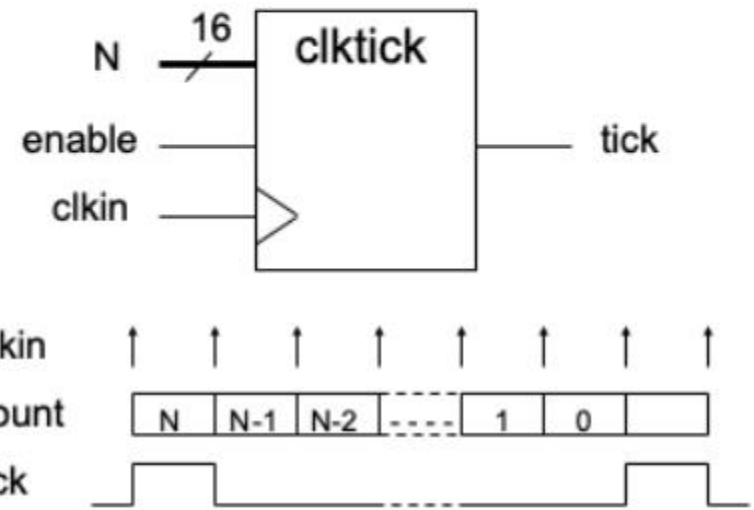
- Alternatively use a count sequence where only one bit changes at a time (e.g. Gray code)
- Top and bottom rows differ only in the MSB ⇒ any even count length can be made by branching to the bottom row after half the counts. Dashed line gives a ÷12 counter




```
module graycode_counter(count, clk);

    output [3:0]    count;
    input           clk;

    reg [3:0]   count;

    initial
        count = 4'b0;

    reg [3:0]   d_in;

    always  @ (posedge clk)
        count <= d_in;

    always @ (count)    // circuit to evaluate next count
        case (count)
            4'd0:   d_in = 4'd1;
            4'd1:   d_in = 4'd3;
            4'd2:   d_in = 4'd6;
            4'd3:   d_in = 4'd2;
            4'd4:   d_in = 4'd12;
            4'd5:   d_in = 4'd4;
            4'd6:   d_in = 4'd7;
            4'd7:   d_in = 4'd5;
            4'd8:   d_in = 4'd0;
            4'd9:   d_in = 4'd8;
            4'd10:  d_in = 4'd11;
            4'd11:  d_in = 4'd9;
            4'd12:  d_in = 4'd13;
            4'd13:  d_in = 4'd15;
            4'd14:  d_in = 4'd10;
            4'd15:  d_in = 4'd14;
        endcase
endmodule
```

A Flexible Timer – clock tick

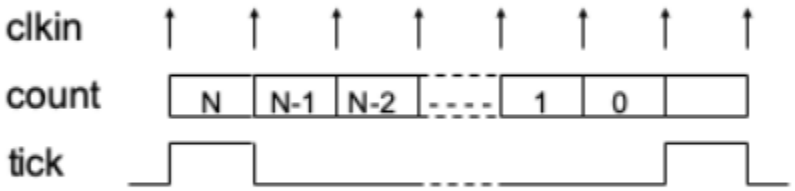
- Instead of having a counter that count events, we often want a counter to provide a measure of **time**. We call this a timer.
- Here is a useful **timer** component that use a clock reference, and produces a pulse lasting for one cycle pulse every N+1 clock cycles.
- If "enable" is low (not enabled), the clkin pulses will be ignored.




```
module clktick (
  clkin,     // Clock input to the design
  enable,    // enable clk divider
  N,         // Clock division factor is N+1
  tick       // pulse_out goes high for one cycle (n+1) clock cycles
);           //    End of port list

parameter    N_BIT = 16;
//-------------Input Ports----------------------------
input clkin;
input enable;
input [N_BIT-1:0] N;

//-------------Output Ports---------------------------
output   tick;
```

clock tick explained

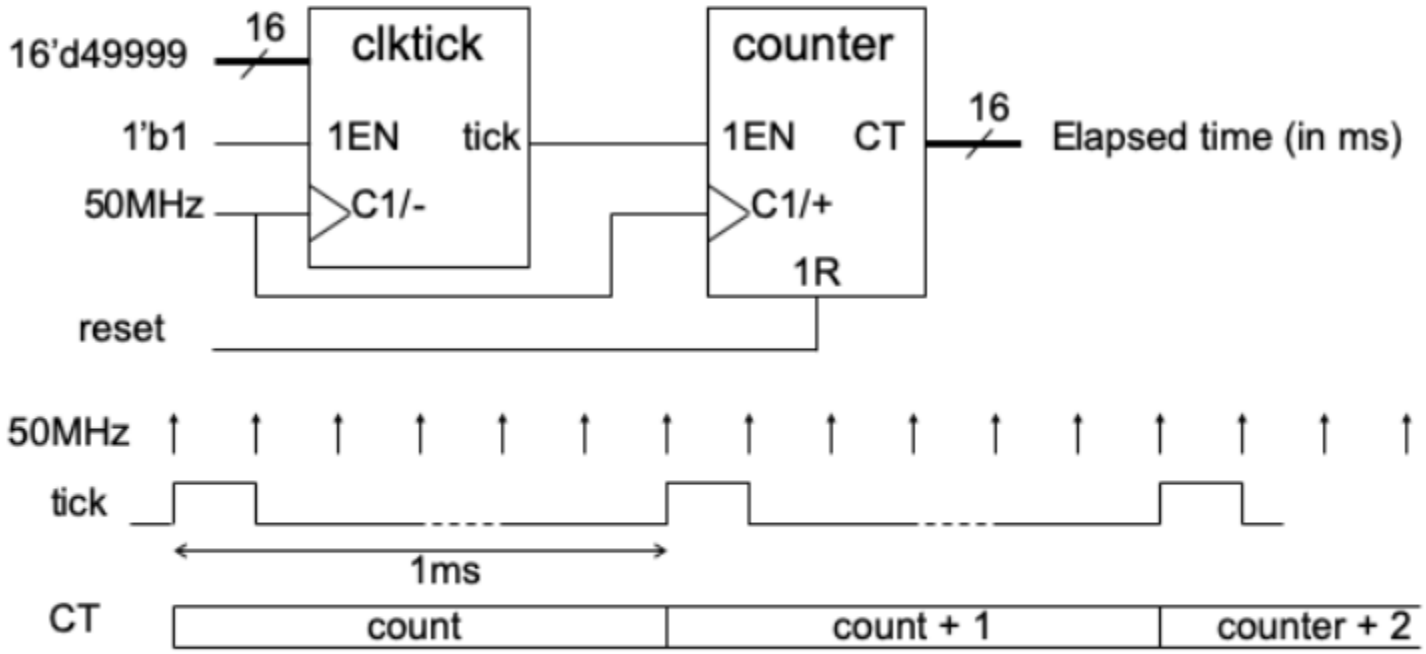
- "count" is an internal N_BIT counter.
- We use this as a down (instead of up) counter.
- The counter value goes from N to 0, hence there are N+1 clock cycles for each tick pulse.




```
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [N_BIT-1:0] count;
reg          tick;

initial      tick = 1'b0;

//------------ Main Body of the module ------------------

   always @ (posedge clkin)
      if (enable == 1'b1)
         if (count == 0)   begin
            tick <= 1'b1;
            count <= N;
            end
         else  begin
            tick <= 1'b0;
            count <= count - 1'b1;
            end

endmodule // End of Module clktick
```

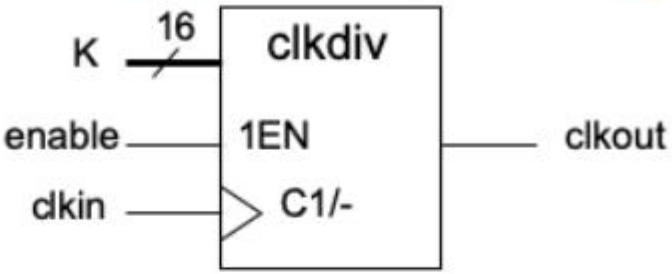
Cascading counters

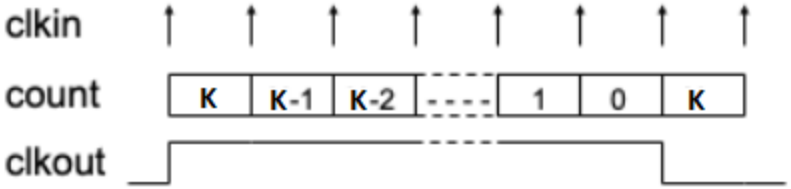
- By connecting **clktick** module in series with a counter module, we can produce a counter that counts the number of millisecond elapsed as shown below.



A clock divider

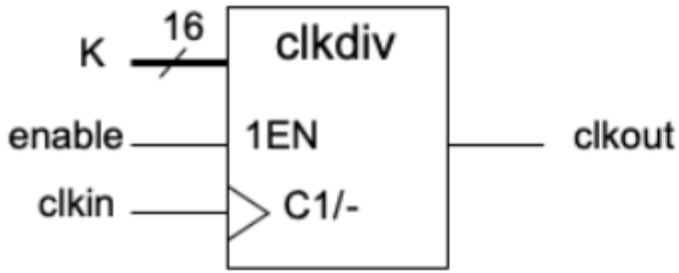
- Another useful module is a clock divider circuit.
- This produces a symmetrical clock output, dividing the input clock frequency by a factor of 2*(K+1).

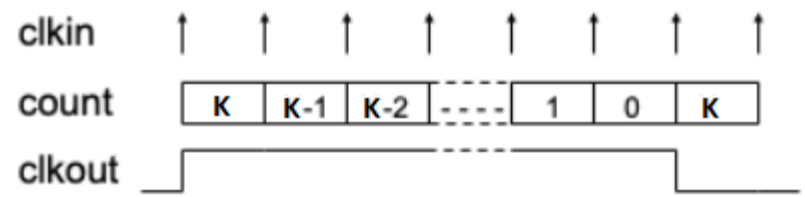





```
module clkdiv (
  clkin,        // Clock input signal to be divided
  enable,       // enable clk divider when high
  K,            // clock frequency divider is 2*(K+1)
  clkout     // symmetric clock output Fout = Fin / 2*(K+1)
);           //    End of port list

parameter   K_BIT = 16;     // change this for different number of bits divider
//-------------Input Ports----------------------------
input clkin;
input enable;
input [K_BIT-1:0] K;

//-------------Output Ports----------------------------
output         clkout;
```

clock divider explained

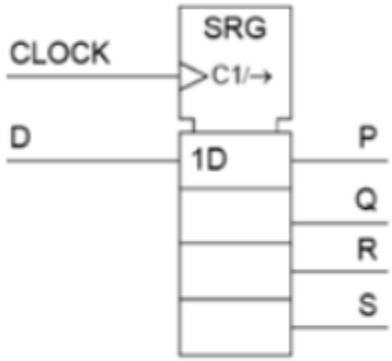


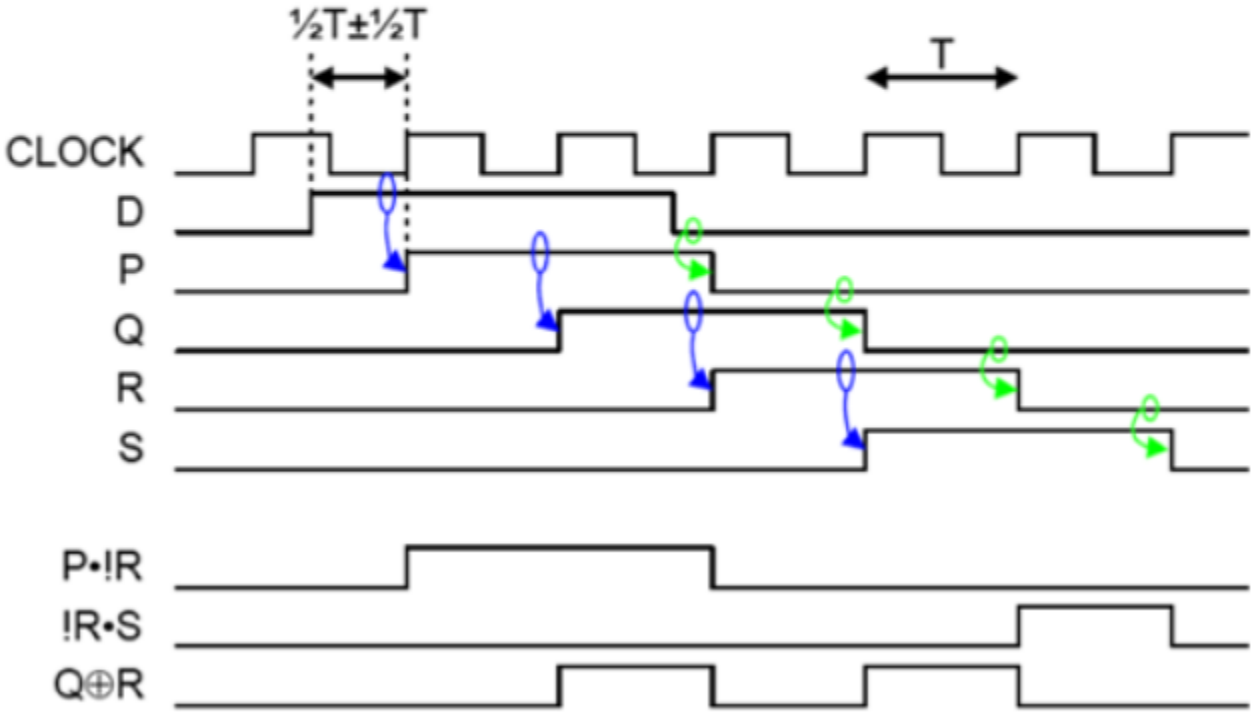



```
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [K_BIT-1:0] count;
reg          clkout;

initial  clkout = 1'b0;

//------------ Main Body of the module ------------------------

   always @ (posedge clkin)
      if (enable == 1'b1)
         if (count == 10'b0)  begin
            clkout <= ~clkout;              // toggle the clock output signal
            count <= K;   // shift right one bit
            end
         else
            count <= count - 1'b1;

endmodule // End of Module clkdiv
```

Shift Registers as control circuits

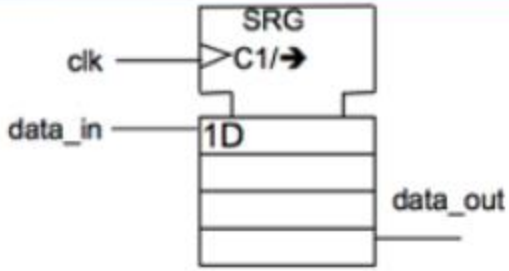
- Easy way to make a sequence of events happen in response to a trigger:
  - P, Q, R and S are delayed versions of D but with all transitions on the CLOCK
  - Delay from D to P is between 0 and 1 clock cycle.

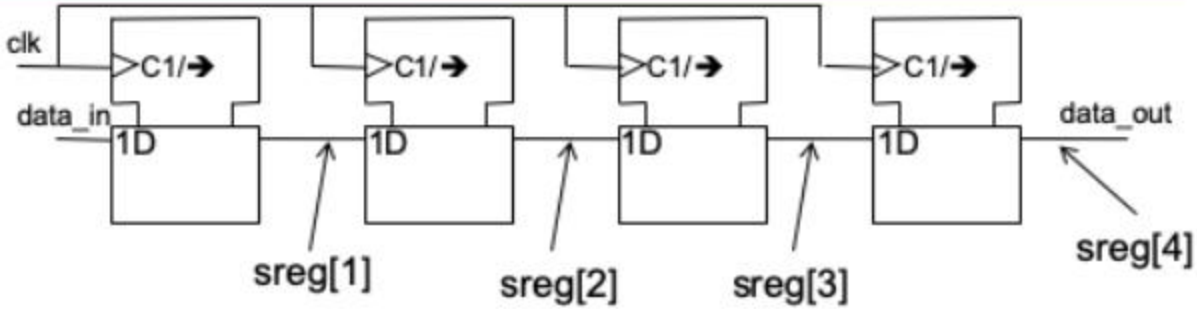




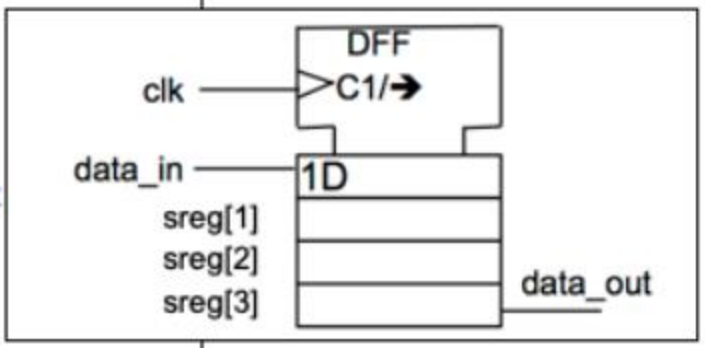
- P•!R gives pulse of length $2T$ approx $\frac{1}{2}T$ after D
- !R•S gives pulse of length T approx $2\frac{1}{2}T$ after D↓
- Q⊕R gives pulses of length T approx $1\frac{1}{2}T$ after D & ↓

Shift Register specification in Verilog



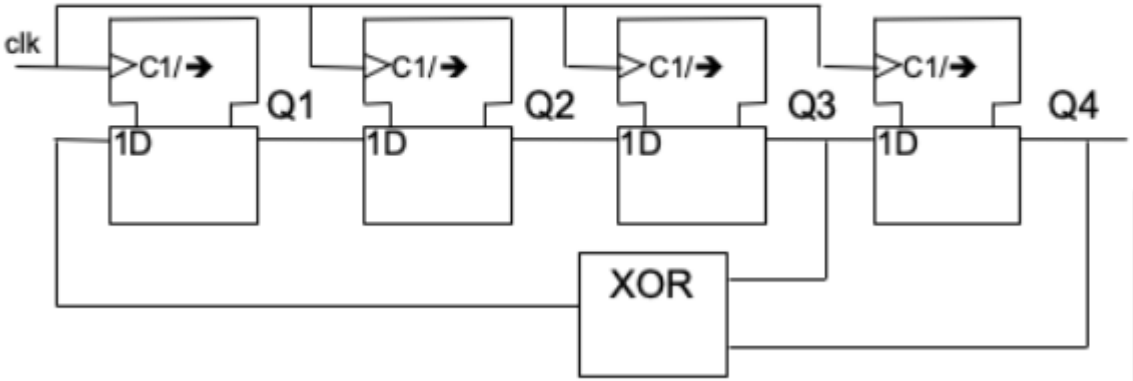



```
module sreg4 (data_out, data_in, clk);

   output    data_out;    // serial data output
   input     data_in;     // serial data input
   input     clk;         // clock input

   reg  [4:1]  sreg;      // 4 stage D-FF for this shift
   initial sreg = 4'b0;
   always @ (posedge clk)
     begin
       sreg[4] <= sreg[3];
       sreg[3] <= sreg[2];
       sreg[2] <= sreg[1];
       sreg[1] <= data_in;
     end

    wire  data_out;
    assign data_out = sreg[4];
endmodule
```




```
sreg <= {sreg[3:1],data_in};
```

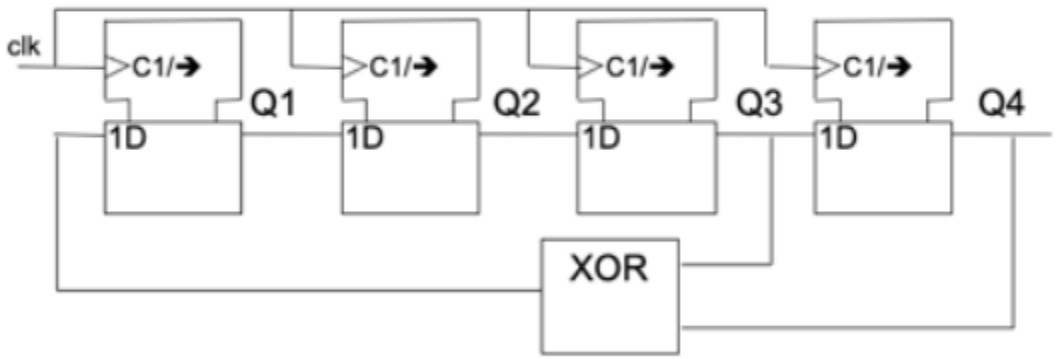
Linear Feedback Shift Register (LFSR)



- Assuming that the initial value is 4'b0001.
- This shift register counts through the sequence as shown in the table here.
- This is now acting as a 4-bit counter, whose count value appears somewhat random.
- This type of shift register circuit is called "**Linear Feedback Shift Register**" or LFSR.
- Its value is sort of random, but repeat very 2^N-1 cycles (where N = no of bits).
- The "taps" from the shift register feeding the XOR gate(s) is defined by a polynomial as shown above.

Q4	Q3	Q2	Q1	count
0	0	0	1	1
0	0	1	0	2
0	1	0	0	4
1	0	0	1	9
0	0	1	1	3
0	1	1	0	6
1	1	0	1	13
1	0	1	0	10
0	1	0	1	5
1	0	1	1	11
0	1	1	1	7
1	1	1	1	15
1	1	1	0	14
1	1	0	0	12
1	0	0	0	8
0	0	0	1	1

Primitive Polynomial

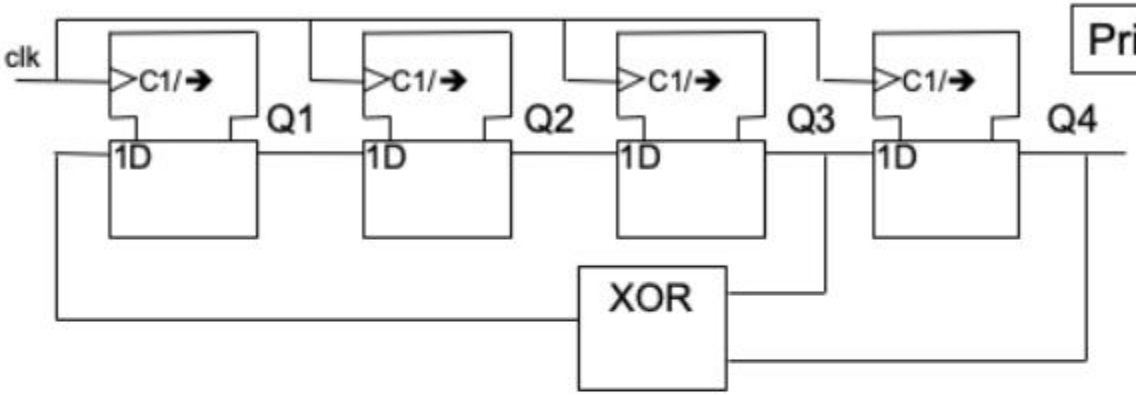


Primitive polynomial: $1 + X^3 + X^4$

- This circuit implements the LFSR based on this **primitive polynomial**: $1 + X^3 + X^4$
- The polynomial is of order 4 (highest power of x)
- This produces a **pseudo random binary sequence** (PRBS) of length $2^4 - 1 = 15$
- Here is a table showing primitive polynomials at different sizes (or orders)

m		m	
3	$1+X+X^3$	14	$1+X+X^6+X^{10}+X^{14}$
4	$1+X+X^4$	15	$1+X+X^{15}$
5	$1+X^2+X^5$	16	$1+X+X^3+X^{12}+X^{16}$
6	$1+X+X^6$	17	$1+X^3+X^{17}$
7	$1+X^3+X^7$	18	$1+X^7+X^{18}$
8	$1+X^2+X^3+X^4+X^8$	19	$1+X+X^2+X^5+X^{19}$
9	$1+X^4+X^9$	20	$1+X^3+X^{20}$
10	$1+X^3+X^{10}$	21	$1+X^2+X^{21}$
11	$1+X^2+X^{11}$	22	$1+X+X^{22}$
12	$1+X+X^4+X^6+X^{12}$	23	$1+X^5+X^{23}$
13	$1+X+X^3+X^4+X^{13}$	24	$1+X+X^2+X^7+X^{24}$

lfsr4



Primitive polynomial: $1 + X^3 + X^4$

```
module lfsr4 (data_out, clk);

   output [4:1]   data_out;    // four bit random output
   input          clk;         // clock input

   reg  [4:1]  sreg;     // 4 stage D-FF for this shift register

   initial sreg = 4'b1;

   always @ (posedge clk)
       sreg <= {sreg[3:1], sreg[4] ^ sreg[3]};

   assign data_out = sreg;
endmodule
```