Lecture 9

Counters & Shift Registers

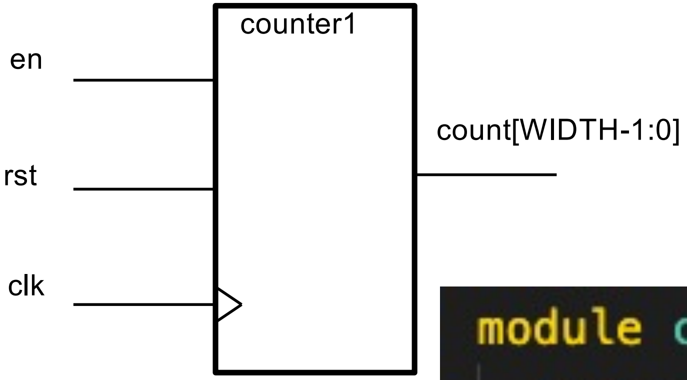
Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

Learning outcomes

- How to specify a simple binary counter?
- How to convert from binary to BCD format?
- How the generate various clock signals with different periods?
- How to specify shift registers?
- How to design a Linear Feedback Shift Register (LFSR) that produces pseudo-random binary sequence (PRBS)?
- How to specify ROM and RAM in SystemVerilog

Example: Simple Counter

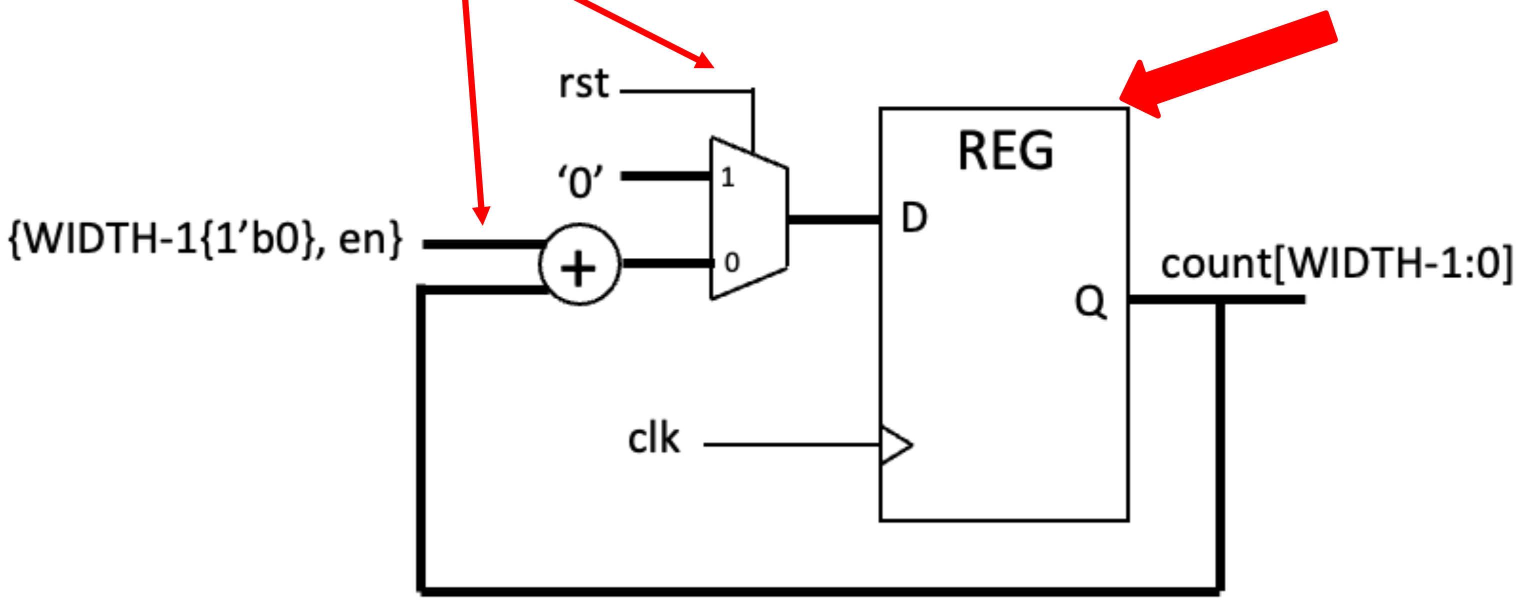



```
module counter #(
  parameter WIDTH = 4
)(                                  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // counter enable
  output logic [WIDTH-1:0] count     // count output
);
  always_ff @ (posedge clk)
    if (rst) count <= {WIDTH{1'b0}};
    else     count <= count + {{WIDTH-1{1'b0}}, en};
endmodule
```

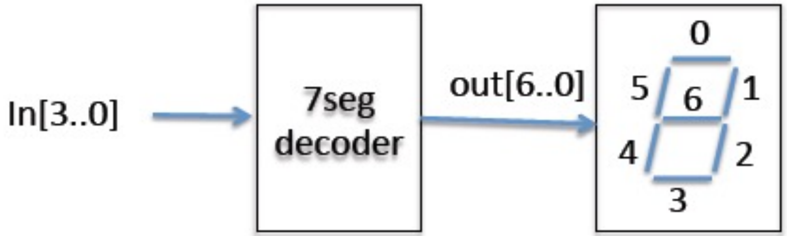
Mapping from SV to hardware

```
if (rst) count <= {WIDTH{1'b0}};
else     count <= count + {{WIDTH-1{1'b0}}, en};
```

```
   output reg  [WIDTH-1:0] count
always_ff @ (posedge clk)
```



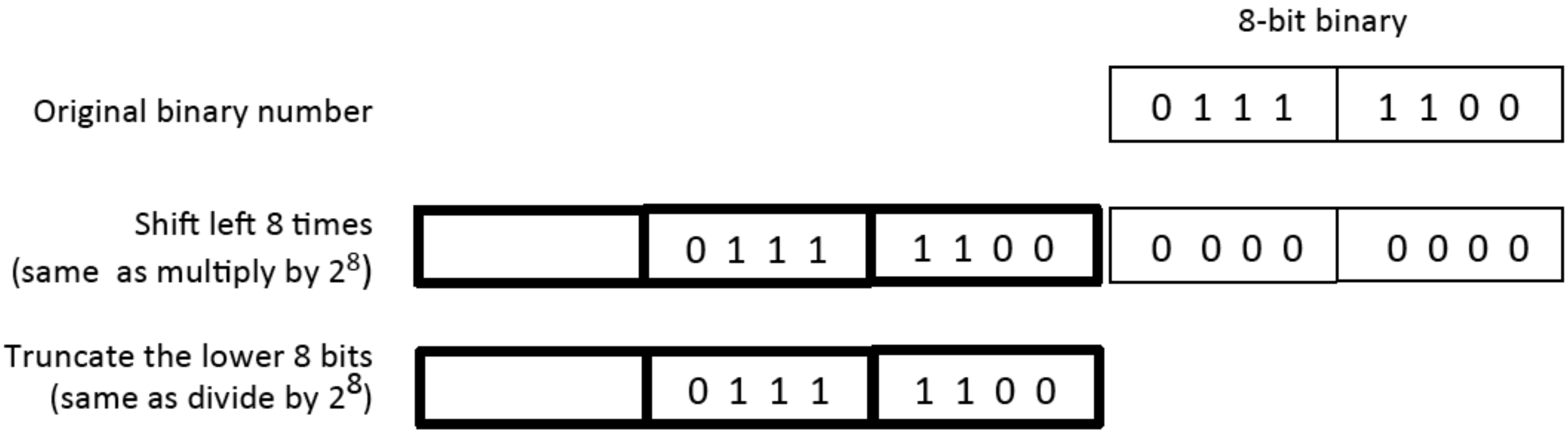
Displaying a binary number as decimal



- In Lab 4 Task 2, you are required to display the counter value as binary coded decimal number instead of hexadecimal. A SystemVerilog component **bin2bcd_16.sv** is provided.
- Hex numbers are difficult to interpret. Often we would like to see the binary value displayed as decimal. For that we need to design a combinational circuit to converter from binary to binary-coded decimal. For example, the value 8’hff or 8’b11111111 is converted to 8’d255 in decimal.

Shift and Add 3 algorithm [1] – shifting operation

- Let us consider converting hexadecimal number 8'h7C (which is decimal 8'd124)
- Shift the 8-bit binary number left by 1 bit = multiply number by 2
- Shifting the number left 8 times = multiply number by 2^8
- Now truncate the number by dropping the bottom 8 bits = divide number by 2^8
- So far we have done nothing to the number – it has the same value
- The idea is that, as we shift the number left into the BCD digit "bins", we make the necessary adjustment to the hex number so that it conforms to the BCD rule (i.e. falls within 0 to 9, instead of 0 to 15)



Shift and Add 3 algorithm [2] – shift left with problem

- If we take the original 8-bit binary number and shift this three times into the BCD digit positions. After 3 shifts we are still OK, because the **ones digit** has a value of 3 (which is OK as a BCD digit).
- If we shift again (4th time), the digit now has a value of 7. This is still OK. However, no matter what the next bit it, another shift will make this digit illegal (either as hexadecimal "e" or "f", both not BCD).
- In our case, this will be a "F"!

	Hundreth BCD	Tens BCD	Ones BCD	8-bit binary	
Original binary number				0 1 1 1	1 1 0 0
Shift left 1 bit – no problem			0	1 1 1 1	1 0 0 0
Shift left 1 bit - no problem			0 1	1 1 1 1	0 0 0 0
Shift left 1 bit – no problem			0 1 1	1 1 1 0	0 0 0 0
Shift left 1 bit - no problem			0 1 1 1	1 1 0 0	0 0 0 0
Shift left 1 bit – problem, not BCD			1 1 1 1	1 0 0 0	0 0 0 0

Shift and Add 3 algorithm [3] – shift and adjust

- So on the fourth shift, we detect that the value is > or = 5, then we adjust this number by adding 3 before the next shift.
- In that way, after the shift, we move a 1 into the tens BCD digit as shown here.

	Hundreth BCD	Tens BCD	Ones BCD	8-bit binary	
Original binary number				0 1 1 1	1 1 0 0
Shift left 1 bit – no problem			0	1 1 1 1	1 0 0 0
Shift left 1 bit - no problem			0 1	1 1 1 1	0 0 0 0
Shift left 1 bit – no problem			0 1 1	1 1 1 0	0 0 0 0
Shift left 1 bit - no problem			0 1 1 1	1 1 0 0	0 0 0 0
Perform adjustment Before shifting by adding 3			1 0 1 0	1 0 0 0	0 0 0 0
We perform adjustment (if >=5, add 3) before shift		1	0 1 0 1	1 1 0 0	0 0 0 0

Shift and Add 3 algorithm [4] – full conversion

- In summary, the basic idea is to shift the binary number left, one bit at a time, into locations reserved for the BCD results.
- Let us take the example of the binary number 8'h7C. This is being shifted into a 12-bit/3 digital BCD result of 12'd124 as shown below.

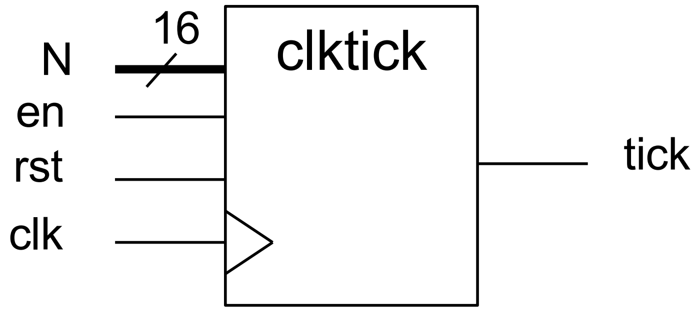
	Hundreth BCD	Tens BCD	Ones BCD	8-bit binary	
Original binary number				0 1 1 1	1 1 0 0
Shift left three times no adjust			0 1 1	1 1 1 0	0
Shift left Ones = 7, ≥5			0 1 1 1	1 1 0 0	
Add 3			1 0 1 0	1 1 0 0	
Shift left Ones = 5		1	0 1 0 1	1 0 0	
Add 3		1	1 0 0 0	1 0 0	
Shift left 2 times Tens = 6, ≥5		1 1 0	0 0 1 0	0	
Add 3		1 0 0 1	0 0 1 0	0	
Shift left BCD value is correct	1	0 0 1 0	0 1 0 0		

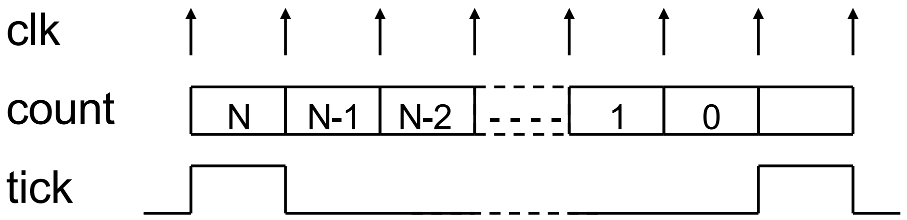
SystemVerilog implementation - bin2bcd_8.sv

```
module bin2bcd_8 (
    input  logic [7:0]   x,       // value ot be converted
    output logic [11:0]  BCD     // BCD digits
);
    // Concatenation of input and output
    logic  [19:0] result;  // = bit in x + 4 * no of digits
    integer i;

    always_comb
    begin
        result[19:0] = 0;
        result[7:0] = x;     // bottom 8 bits has input value

        for (i=0; i<8; i=i+1) begin
            // Check if unit digit >= 5
            if (result[11:8] >= 5)
                result[11:8] = result[11:8] + 4'd3;

            // Check if ten digit >= 5
            if (result[15:12] >= 5)
                result[15:12] = result[15:12] + 4'd3;

            // Shift everything left
            result = result << 1;
        end

        // Decode output from result
        BCD = result[19:8];
    end

endmodule
```

A Flexible Timer – clktick.sv

- 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 uses a clock reference, and produces a pulse lasting for one cycle every N+1 clock cycles.
- If "en" signal is low (not enabled), the clkin pulses are ignored.

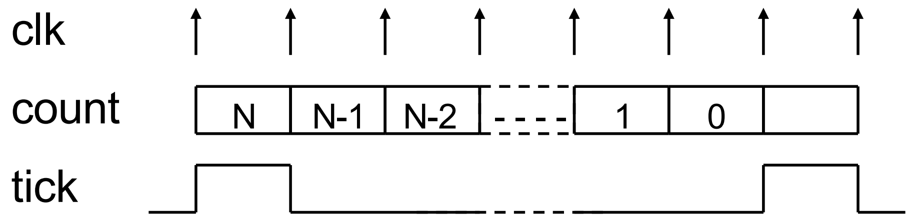





```
module clktick #(
    parameter WIDTH = 16
)(
  // interface signals
  input  logic             clk,      // clock
  input  logic             rst,      // reset
  input  logic             en,       // enable signal
  input  logic [WIDTH-1:0] N,        // clock divided by N+1
  output logic             tick      // tick output
);

logic [WIDTH-1:0] count;
```

clktick.sv explained

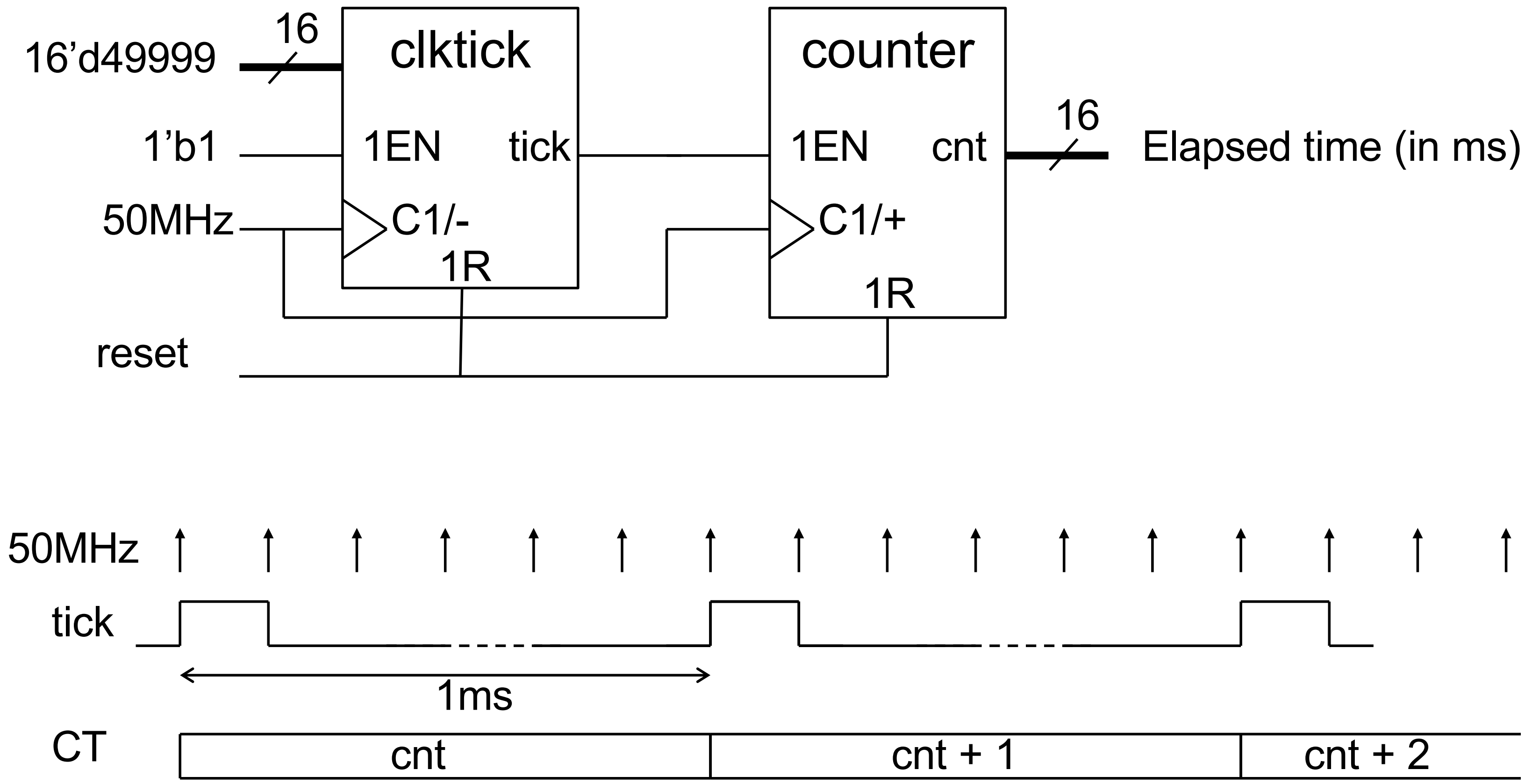
- "count" is an internal counter with WIDTH bits
- We use this as a down (instead of up) counter
- The counter value goes from N to 0, hence there are N+1 clock cycles for each tick pulse




```
always_ff @ (posedge clk)
    if (rst) begin
        tick <= 1'b0;
        count <= N;
        end
    else if (en) begin
        if (count == 0) begin
        tick <= 1'b1;
        count <= N;
          end
        else begin
        tick <= 1'b0;
        count <= count - 1'b1;
          end
        end
endmodule
```

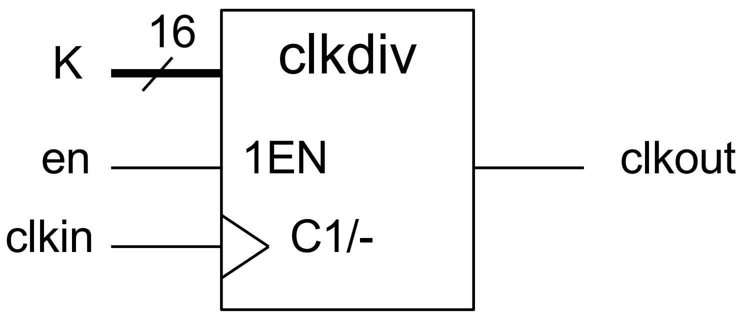
Cascading counters

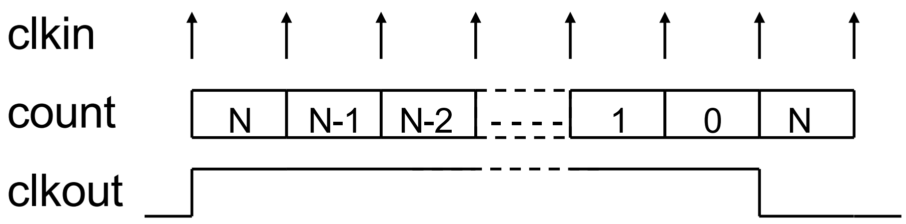
- By connecting **clktick** module in series with a counter module, we can produce a counter that counts the number of millisecond elapsed as shown below.



Clock divider (clkdiv.sv)

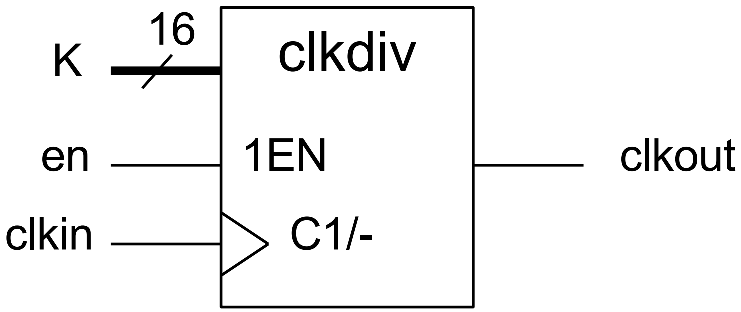
- Another useful module is a **clock divider circuit**.
- This produces a **symmetrical** clock output, dividing the input clock frequency by a factor of 2*(K+1).

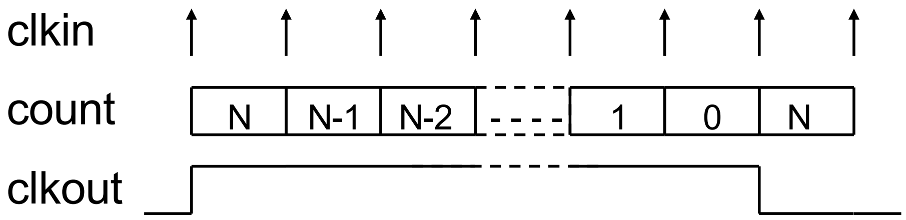





```
module clkdiv #(
    parameter WIDTH = 16
)(
  input  logic               clkin,  // Clock input signal to be divided
  input  logic               en,     // enable clk divider when high
  input  logic [WIDTH-1:0]   K,      // half clock period counts K+1 clkin cycles
  output logic               clkout  // symmetric clock output Fout = Fin / 2*(K+1)
);          //  End of port list

logic [WIDTH-1:0] count;        // internal counter
```

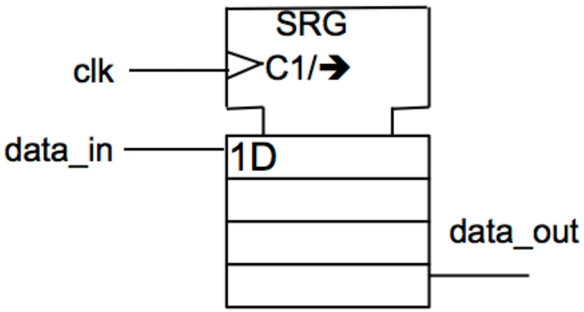
clkdiv.v explained

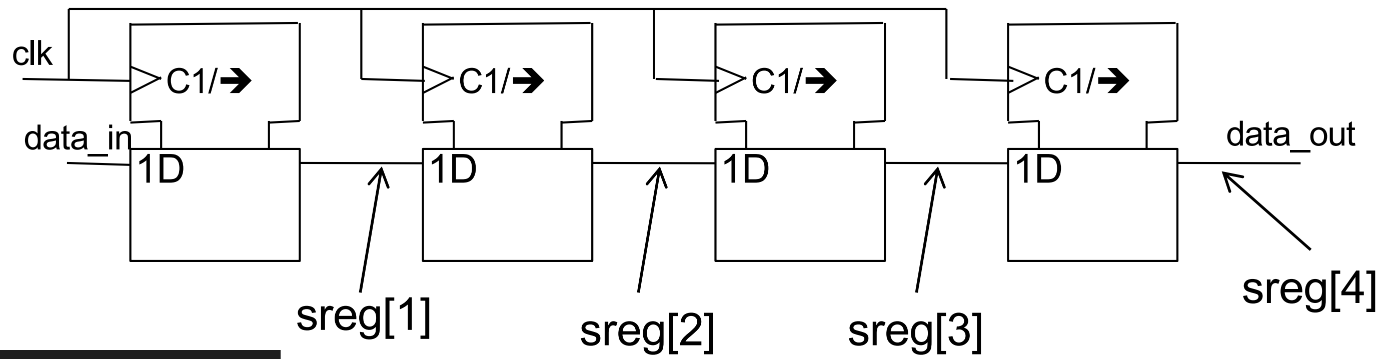




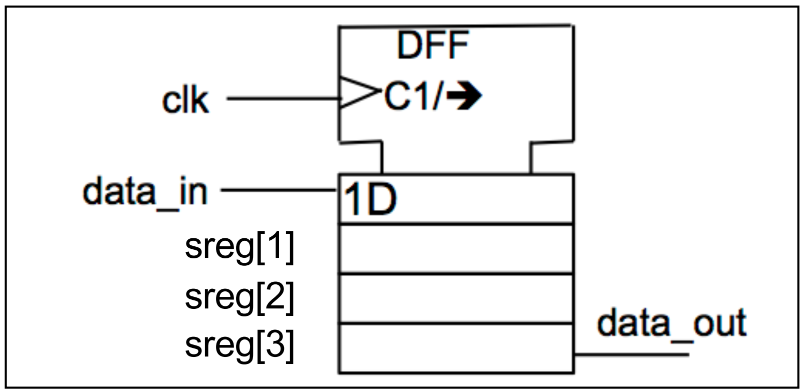

```
initial clkout = 1'b0;          // alternative way to initialise logic
initial count = {WIDTH{1'b0}}   // ... or FF (Good for FPGA designs)

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

    always_ff @ (posedge clkin)
        if (en == 1'b1)
            if (count == {WIDTH{1'b0}}  begin
                clkout <= ~clkout;              // toggle the clock output
                count <= K;  // shift right one bit
                end
            else
                count <= count - 1'b1;
endmodule // End of Module clkdiv
```

Shift Register specification in SystemVerilog



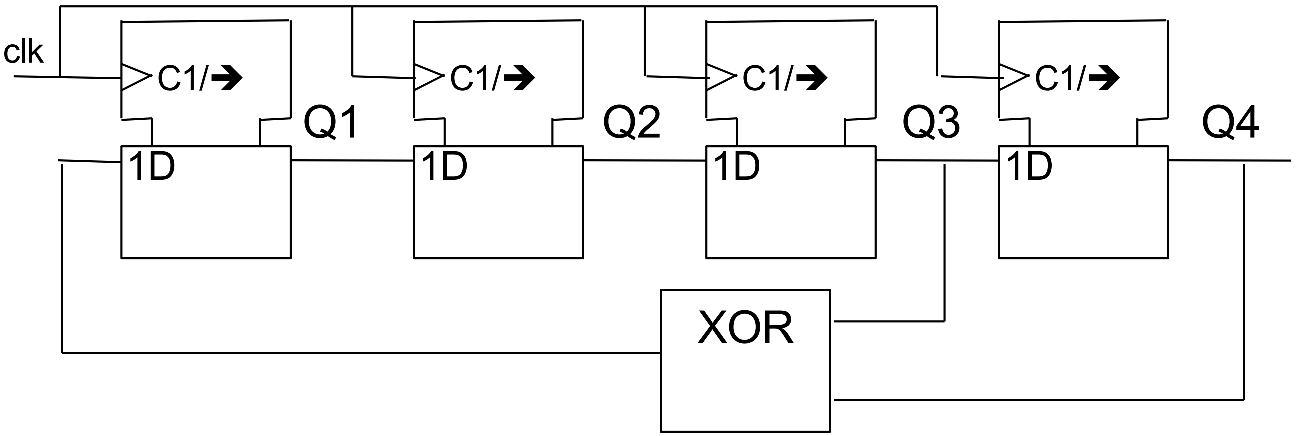



```
module sreg4 (
  input  logic      clk,       // input clock
  input  logic      rst,       // reset
  input  logic      data_in,   // serial data in
  output logic      data_out   // serial data out
);        //  End of port list

   logic [4:1]   sreg;          // shift register

   always_ff @ (posedge clk)
      if (rst)
         sreg <= 4'b0;
      else  begin
         sreg[4] <= sreg[3];
         sreg[3] <= sreg[2];
         sreg[2] <= sreg[1];
         sreg[1] <= data_in;
      end
   assign data_out = sreg[4];
endmodule
```




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

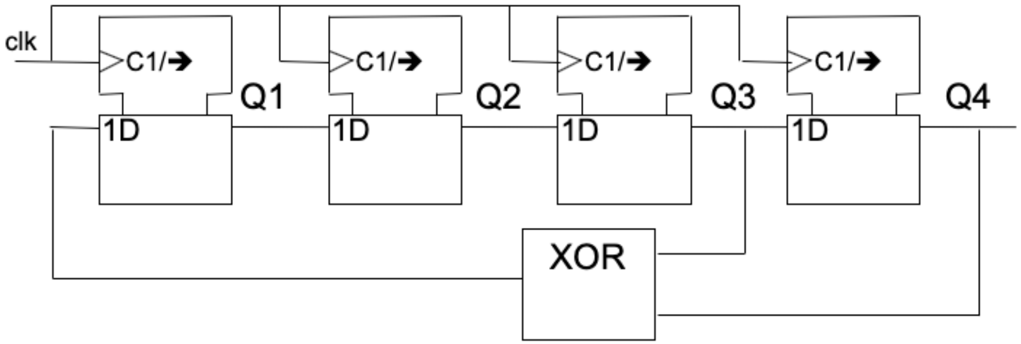
Linear Feedback Shift Register (LFSR) (1)



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

- 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.

Primitive Polynomial

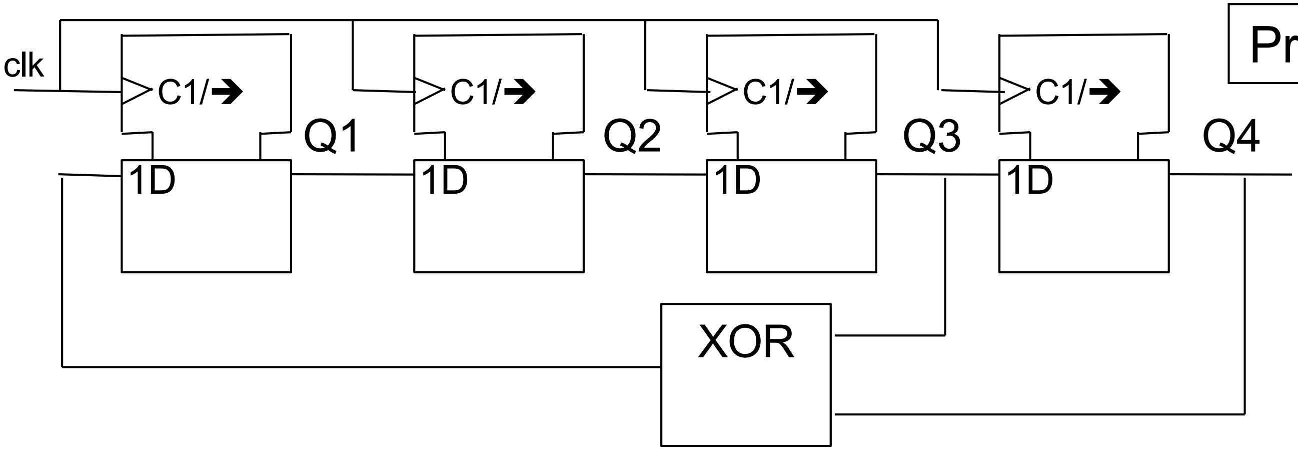


Primitive polynomial: $1 + X^3 + X^4$

- This circuit implements the LFSR based on this **primitive polynomial**:
- 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.sv



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

```
module lfsr4 (
    input logic         clk,        // clock
    input logic         rst,        // reset
    output logic [4:1]  data_out    // pseudo-random output
);

always_ff @ (posedge clk, posedge rst)
    if (rst)
        sreg <= 4'b1;
    else
        sreg <= {sreg[3:1], sreg[4] ^ sreg[3]};

assign data_out = sreg;
endmodule
```