

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

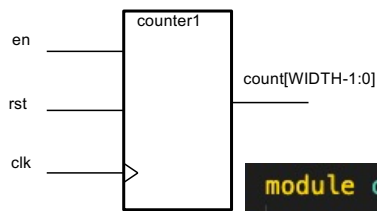
In this lecture, we will focus on two very important digital building blocks: **counters** which can either count events or keep time information, and shift **registers**, which is most useful in conversion between serial and parallel data formats. We will also learn about a special type shift register known as Linear Feedback Shift Registers, which are widely used to generate random digital numbers.

Learning outcomes

- ❖ How to specify a simple binary counter?
- ❖ How to convert from binary to BCD format?
- ❖ How to 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

Here are a list of learning outcome for this lecture. It is also tightly coupled with Lab 2, which will take you through the steps in designing with ROM, RAM and counter, to produce a variable frequency sinewave generator.

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
```

In Lab 4, you are to design an 4-bit counter as shown here. The counter has three inputs:

1. clk – the clock signal (positive edge triggered)
2. rst – the reset signal (high reset), synchronous to clk
3. en – the enable signal, i.e. counting only if en = '1'

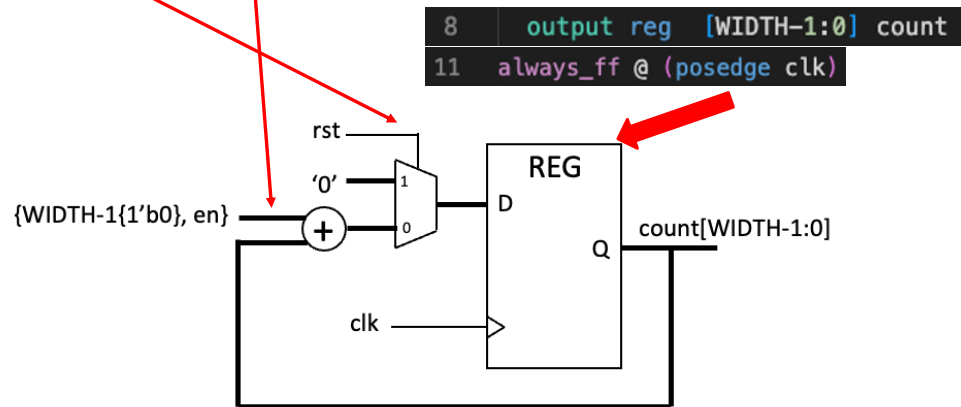
The counter has output count[3:0].

Note the following:

1. We use parameter to define the width of the counter to be 4-bit. The use of parameter allows the same module to be used with different counter width (covered in next lecture).
2. The use of concatenation {.,} to create 8-bit value with the LSB = en signal.

Mapping from SV to hardware

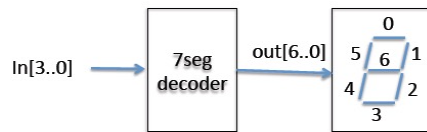
```
if (rst) count <= {WIDTH{1'b0}};  
else  
    count <= count + {{WIDTH-1{1'b0}}, en};
```



This is how the SV code is mapped to the actual hardware synthesized by Verilator.

The if-else statement is mapped to the MUX. The counting action is achieved via the adder on the feedback path of the register.

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 convert from binary to binary-coded decimal. For example, the value 8'hff or 8'b11111111 is converted to 8'd255 in decimal.

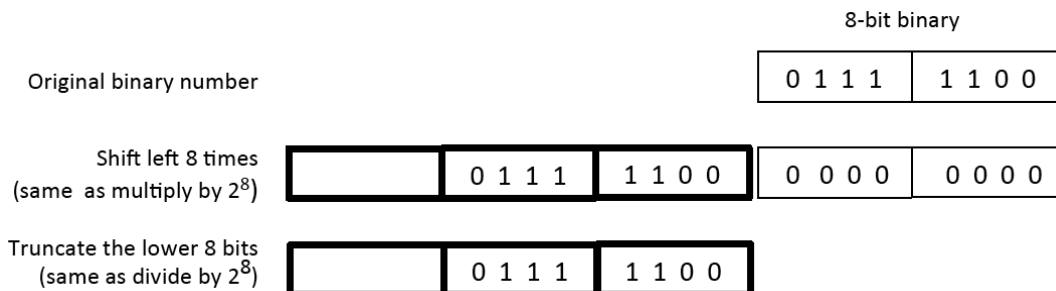
We now take another example of a relative complex combinational circuit, and see how we can specify our design in SystemVerilog.

The goal is to design a circuit that converts an 8-bit binary number into three x 4-bit binary coded decimal values (i.e. 12 bit).

There is a well-known algorithm called “**shift-and-add-3**” algorithm to do this conversion. For example, if we take 8-bit hexadecimal number 8'hff (i.e. all 1's), it has two hex digits. Once converted to binary coded decimal (BCD) it becomes 255 (3 BCD digits).

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)



Before we examine this algorithm in detail, let us consider the arithmetic operation of shifting left by one bit. This is the same as a $\times 2$ operation.

If we do it 8 times, then we have multiplied the original number by 256 or 2^8 .

Now if you ignore the bottom 8-bit through a truncation process, you effectively divide the number by 256. In other words, we get back to the original number in binary (or in hexadecimal).

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 is, 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

Our conversion algorithms works by shift the number left 8 times, but each time make an adjustment (or correction) if it is NOT a valid BCD digit.

Let us consider this example. We can shift the number four time left, and it will give a valid BCD digit of 7.

However, if we shift left again, then 7 becomes hex F, which is NOT valid. Therefore the algorithm demands that 3 is added to 7 (7 is larger or equal to 5) before we do the shift.

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

- ◆ So on the fourth shift, we detect that the value is ≥ 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

The rationale of this algorithm is the following. If the number is 5 or larger, after shift left, we will get 10 or larger, which cannot fit into a BCD digit. Therefore if the number 5 (or larger) we add 3 to it (after shifting is adding 6), which measure we carry forward a 1 to the next BCD digit.

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		

PYKC 11 Nov 2025

EE2 Circuits and Systems

Lecture 9 Slide 9

To recap: 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 as shown above.

After 8 shift operations, the three BCD digits contain respectively: hundredth digit = 4'b0001, tens digit = 4'b0010 and ones digit = 4'b0100, thus representing the BCD value of 124.

The key idea behind the algorithm can be understood as follow (see the diagram in the slide):

1. Each time the number is shifted left, it is multiplied by 2 as it is shifted to the BCD locations;
2. The values in the BCD digits are the same as as binary if its value is 9 or lower. However if it is 10 or above, the number is wrong for BCD. Instead, it should carry over to the next digit. A correction must be made by adding 6 to this digit value.
3. The easiest way to do this is to detect if the value in the BCD digit locations are 5 or above BEFORE the shift (i.e. X2). If it is ≥ 5 , then add 3 to the value (i.e. adjust by +6 after the shift).

SystemVerilog implementation - bin2bcd_8.sv

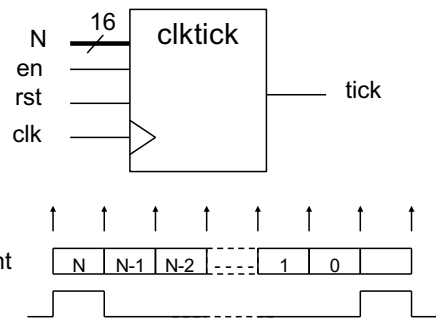
```
14 module bin2bcd_8 (  
15     input  logic [7:0]  x,      // value to be converted  
16     output logic [11:0] BCD     // BCD digits  
17 );  
18 // Concatenation of input and output  
19 logic [19:0] result; // = bit in x + 4 * no of digits  
20 integer i;  
21  
22 always_comb  
23 begin  
24     result[19:0] = 0;  
25     result[7:0] = x; // bottom 8 bits has input value  
26  
27     for (i=0; i<8; i=i+1) begin  
28         // Check if unit digit >= 5  
29         if (result[11:8] >= 5)  
30             result[11:8] = result[11:8] + 4'd3;  
31  
32         // Check if ten digit >= 5  
33         if (result[15:12] >= 5)  
34             result[15:12] = result[15:12] + 4'd3;  
35  
36         // Shift everything left  
37         result = result << 1;  
38     end  
39  
40     // Decode output from result  
41     BCD = result[19:8];  
42 end  
43  
44 endmodule
```

Here is the SystemVerilog implementation of the binary to BCD algorithm. You are invited to examine how the algorithm described in previous slides are implemented in this behavioural description in SV.

Note that although this description looks like a software function, synthesis program will produce hardware implementation of it, say, in FPGA.

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;
    
```

Counters are good in counting events (e.g. clock cycles). We can also use counters to provide some form of time measurement.

Here is a useful component called a “clock tick” circuit. We are not interested in the actual count value. What is needed, however, is that the circuit generates a **single clock pulse** (i.e. lasting for one clock period) for every $N+1$ **rising edge** of the clock input signal **clk**.

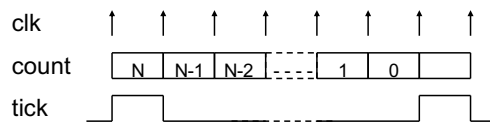
We also add an enable signal **en**, which must be set to ‘1’ in order to enable the internal counting circuit.

Shown here is the module interface for this circuit in SystemVerilog.

Note that the **parameter** keyword is used to define the number of bits of the internal counter (or the count value N). This makes the module easily adaptable to different size of counter.

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
```

The actual SystemVerilog specification for this module is shown here.

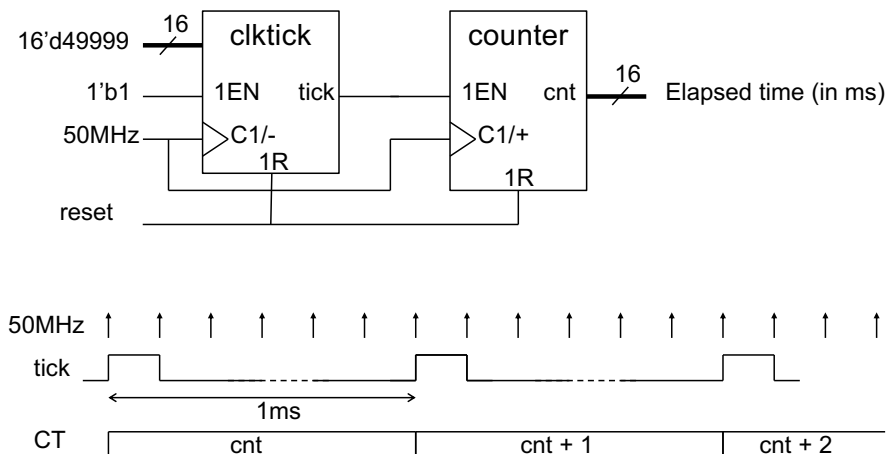
There has to be an internal counter **count** whose output is NOT visible external to this module. This is created with the **reg [N_BIT-1:0] count;** statement.

The output **tick** has to be declared as **reg** because its value is updated inside the **always** block.

Also note that instead of adding '1' on each positive edge of the clock, this design uses a **down counter**. The counter counts from N to 0 (hence N+1 clock cycles). When that happens, it is reset to N and the tick output is high for the next clock cycle.

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.



Using this style to design a clock tick circuit allows us to easily connect multiple counters in series as shown here.

The **clktick** module is producing a pulse on the **tick** output every 50,000 cycles of a 50MHz clock. Therefore **tick** goes high for 20 nanosecond once every 1 msec (or 1KHz).

The **clktick** module is sometimes called a **prescaler** circuit. It prescale the input clock signal (50MHz) in order for the second counter to count at a lower frequency (i.e. 1KHz).

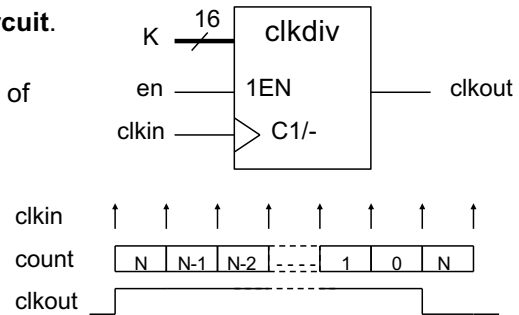
The second counter is now counting the number of millisecond that has elapsed since the last time reset signal (**1R**) goes high.

The design of this circuit is left as a Laboratory task for you to do.

In case you are not familiar with the schematic notation here (which is a IEEE standard), **C1/-** indicates that the clock input is synchronized to the enable and reset input (**1EN** and **1R**), and it results in circuit counting DOWN ('-' sign).

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 $2*(K+1)$.



```

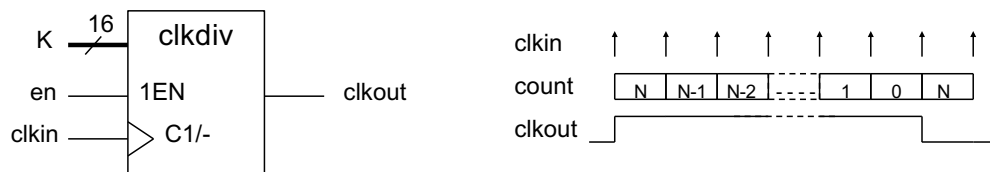
module clkdiv #(
    parameter WIDTH = 16
) (
    input logic      clk,    // Clock input signal to be divided
    input logic      en,     // enable clk divider when high
    input logic [WIDTH-1:0] K, // half clock period counts K+1 clk cycles
    output logic      clkout // symmetric clock output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

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

Here is yet another useful form of a counter. I call this a **clock divider**. Unlike the **clktick** module, which produces a one cycle **tick** signal every $N+1$ cycle of the clock, this produces a **symmetric clock** output **clkout** at a frequency which is the input clock frequency divided by $2*(K+1)$.

Shown here is the module interface in SystemVerilog. Again we have used the **parameter** statement to make this design ease of modification for different internal counter size.

clkdiv.v explained



```

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

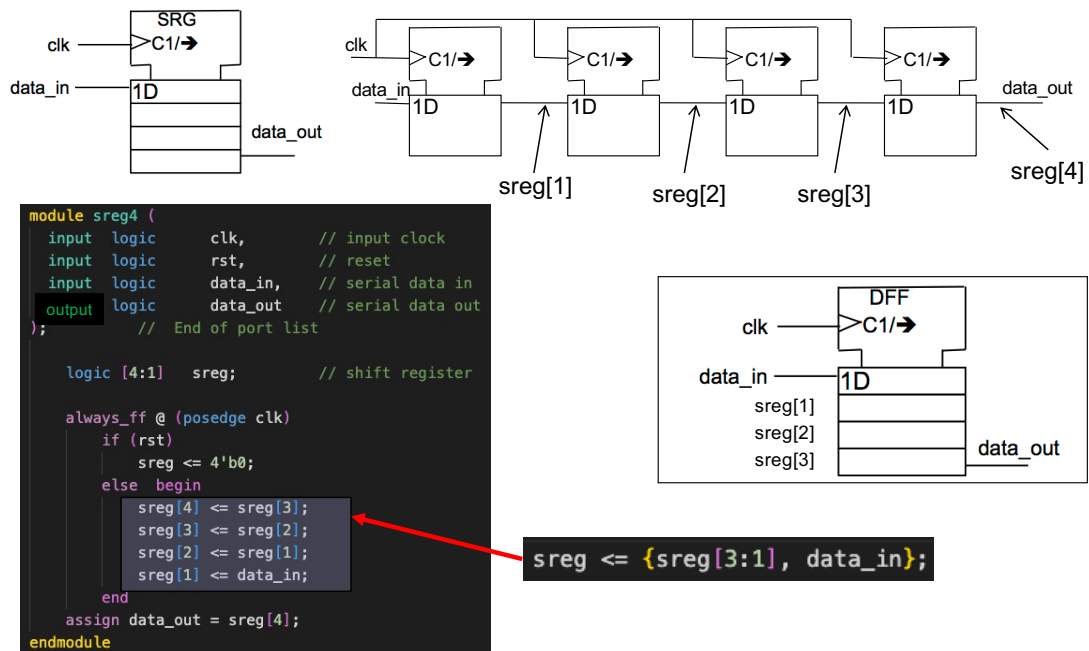
//----- Main Body of the module -----

always_ff @ (posedge clkdiv)
    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

```

The Verilog specification is similar to that for `clktick`. This also has an internal counter that counts from `K` to 0, then the output `clkout` is toggled whenever the count value reaches 0.

Shift Register specification in SystemVerilog



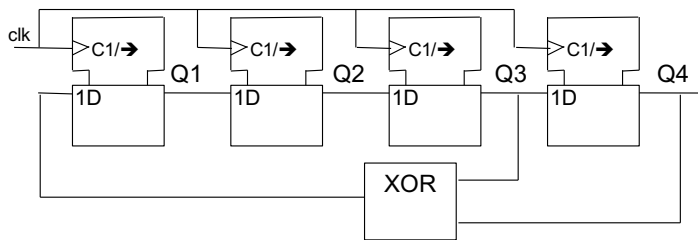
To specify a shift register in SystemVerilog, use the code shown here. We use the `<=` assignment to make sure that `sreg[4:1]` are updated only at the end of the `always` block.

On the right is a short-hand version of the four assignment statements:

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

This way of specifying the right-hand side of the assignment is powerful. We use the **concatenation operation** `{ }` to make up four bits from `sreg[3:1]` and `data_in` (with `data_in` being the LSB) and assign it to `sreg[4:1]`.

Linear Feedback Shift Register (LFSR) (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 every $2^N - 1$ 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

PYKC 11 Nov 2025

EE2 Circuits and Systems

Lecture 9 Slide 17

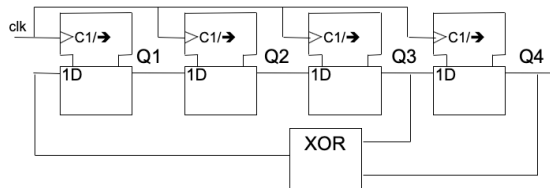
We can also make a shift register count in binary, but in an interesting sequence.

Consider the above circuit with an initial state of the shift register set to 4'b0001.

The sequence that this circuit goes through is shown in the table here. It is NOT counting binary. Instead it is counting in a sequence that is sort of **random**. This is often called a **pseudo random binary sequence (PRBS)**.

The shift register connect this way is also known as a “**Linear Feedback Shift Register**” or LFSR. There is a whole area of mathematics devoted to this type of computation, known as “finite fields” which we will not consider on this course.

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	$1 + X^3 + X^{23}$
13	$1 + X + X^3 + X^4 + X^{13}$	24	$1 + X + X^2 + X^7 + X^{24}$

PYKC 11 Nov 2025

EE2 Circuits and Systems

Lecture 9 Slide 18

The circuit shown below is effectively implementing a sequence defined by a polynomial shown: $1 + X^3 + X^4$. The term "1" specifies the input to the left-most D-FF. This signal is derived as an XOR function (which is the finite field '+') of two signals "tapped" from stage 3 (i.e. X^3) and stage 4 (i.e. X^4) of the shift register.

For a m -stage LFSR, where m is an integer, one could always find a polynomial (i.e. tap configuration) that will provide **maximal length**. This means that the sequence will only repeat after $2^m - 1$ cycles. Such a polynomial is known as a "**primitive polynomial**".

The table shown in the slide has primitive polynomial at various order. For example, a 15-bit LFSR that produces a maximal length PRBS can be achieved by implementing the primitive polynomial:

$$1 + X + X^{15}$$

This results in a 15-bit shift register with one XOR gate from Q1 and Q15, feeding back to input of FF1 (which we would label Q0). This effectively implements the equation:

$$X^0 = X + X^{15}$$

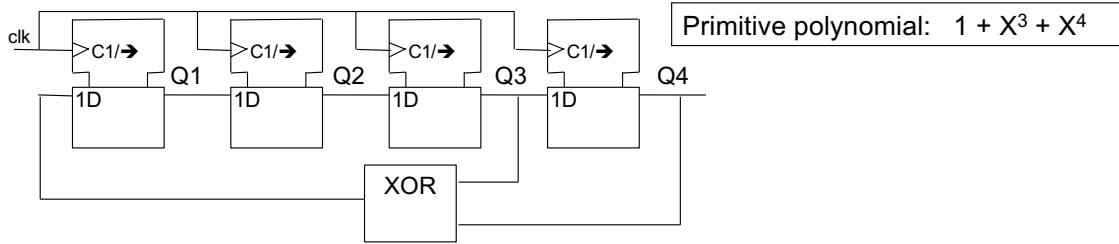
Note that for a given order, the primitive polynomial shown here is NOT unique.

For example, for order 4, the table shows an alternative primitive polynomial:

$$1 + X + X^4$$

This will produce a pseudo-random sequence which is also maximal length different from that using the polynomial in the slide ($1 + X^3 + X^4$).

lfsr4.sv



```
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
```

Here is the implementation of a 4-bit LFSR of the primitive polynomial

$$1 + X^3 + X^4$$

This is essentially a shift register with data_in feed from an XOR gate with Q3 and Q4. Note that we MUST initialize the shift register to a value other than 4'b0000 (e.g. 4'b0001 will do).

This module has not been parameterized because for different WIDTH, we need different primitive polynomial!