Imperial College London

Lecture 5

Counters & Shift Registers

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/ee2_digital
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.

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)

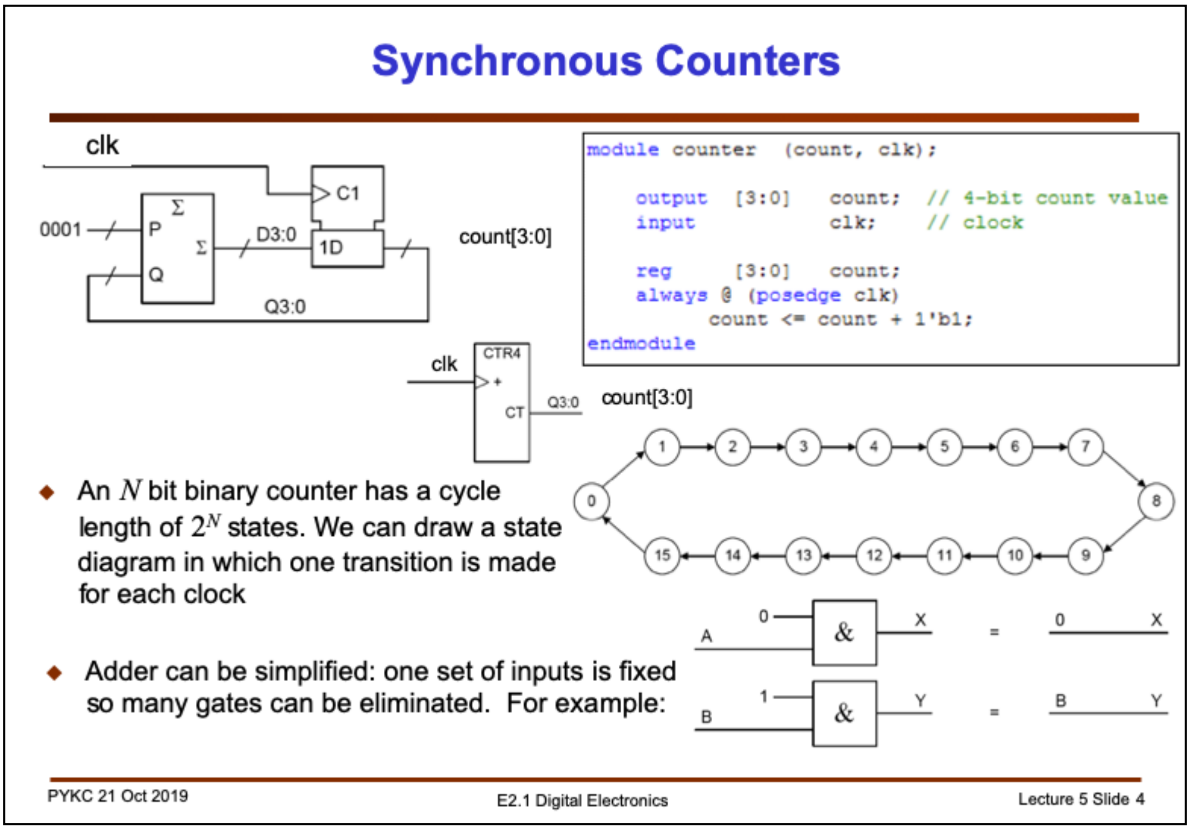
PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 3

Logic circuits generally consists of circuits in three categories:

- **Datapath blocks** – these are arithmetic blocks, logic units, floating point units, digital processing blocks etc.
- **Memory blocks** – these store information for digital circuit to manipulate or store the instructions for a microprocessor
- **Control logic blocks** – these are generally clocked and they make sure that data goes to the right places at the right time for computation

Control logic can be implemented either in a microprocessor or a microcontroller, a synchronous finite state machine (probably best) or using random logic with counters and shift registers.

In this lecture, we will introduce the use of counters and shift registers for producing control circuits.



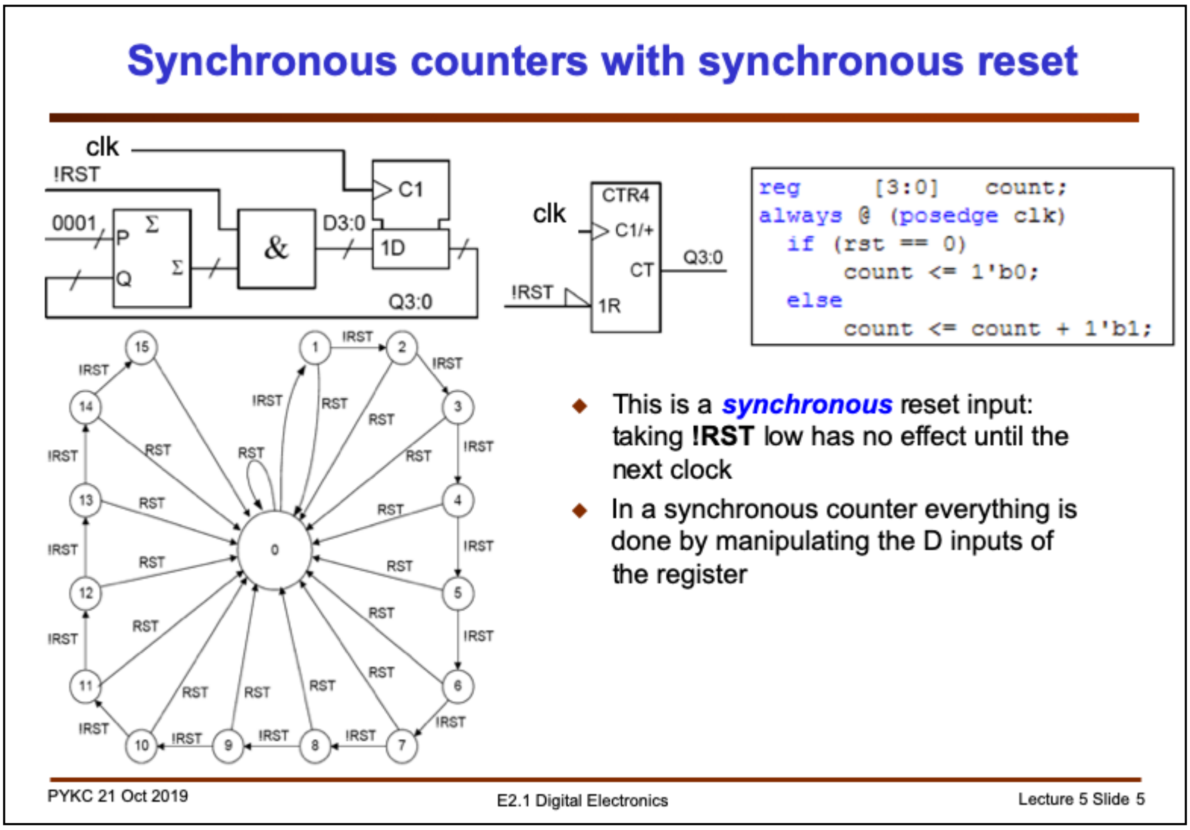
Here is the design of a 4-bit synchronous counter in schematic form. The four D-FFs store the current count value. The adder is used to compute the next count value and fed to the input of the D-FFs. On every positive edge of the signal **clk**, the count value **count[3:0]** goes up by one.

The Verilog description of this counter is shown. Note how easy it is to specify the D-FFs (via the always block with **posedge** keyword in the sensitivity list). The add-one is specified with the '+' operator.

In the actual implementation, the CAD software will NOT insert a 4-bit adder circuit. Instead the synthesis software will reduce this to an optimised gate-level implementation because one of the operand of the adder is the constant '1'. However, as a designer, you do not need to worry about this optimisation step (except may be in answering an examination question which asks you to simplify). The CAD software will deal with this for you. In fact, it is often much better to leave the specification as shown here – the design is more readable and therefore easier for others to understand what you are doing.

For example, one input to the adder is: **4'b0001**. Imagine you have many AND and OR gates inside the adder. The AND driven by 0 or 1 can be mapped to a simple wire. Similarly simplification can be applied to OR gates.

Finally, shown here is the state transition diagram. Each bubble represents a state (labeled with the count value) and on each rising edge of **clk**, it transits to the next state.

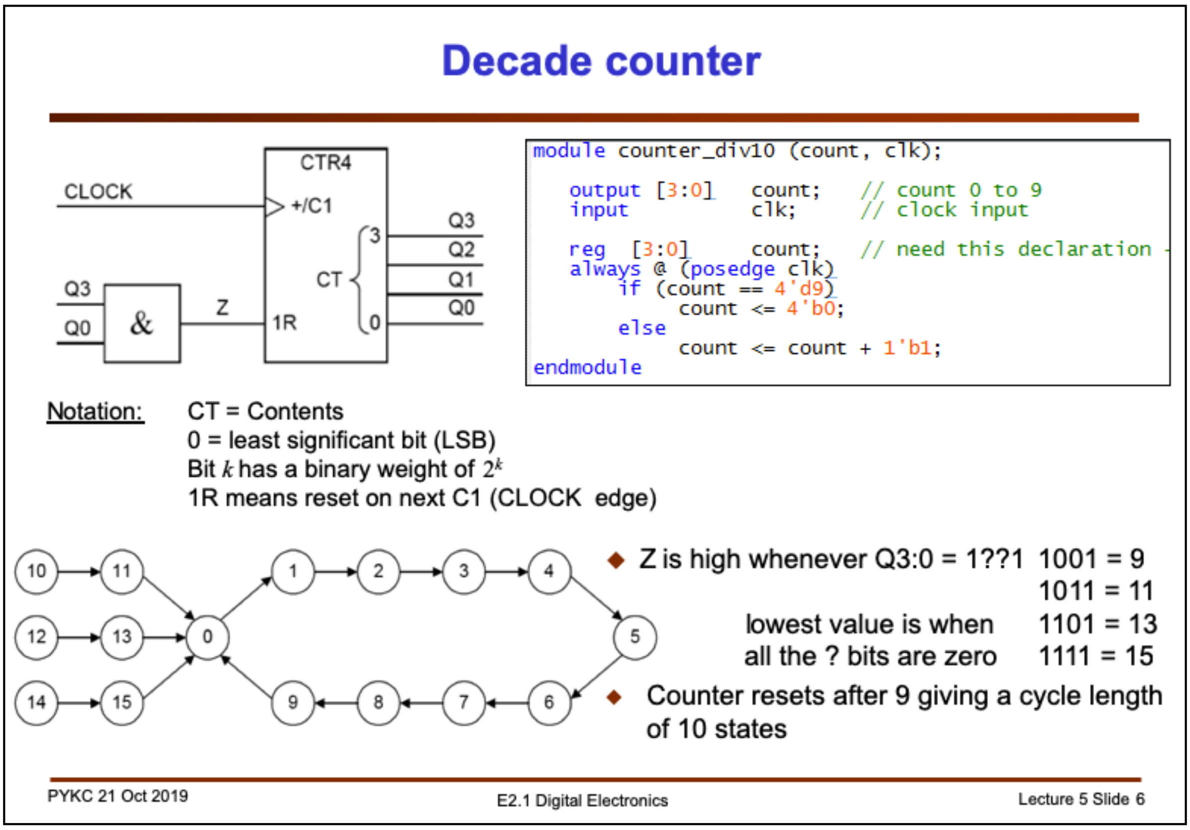


Here is a modification from the previous counter by adding a low-active reset signal **RST** (shown as !RST).

The implementation is simple in Verilog – just add a if-else statement to specify that when RST = 0, reset on the **next** clock rising edge.

The state transition diagram is modified to include the reset action.

The symbol for the counter is worth noting. CTR4 indicates that it is a 4-bit counter. CT means count vaue. C1/+ label on the **clk** signal indicates that this clock signal controls either the action of input 1 or will increment the counter. The 1R label indicates that it is a reset input and it is controlled by the clock signal 1. Finally the triangle at the input of 1R says that this input is low active (i.e. low to reset).



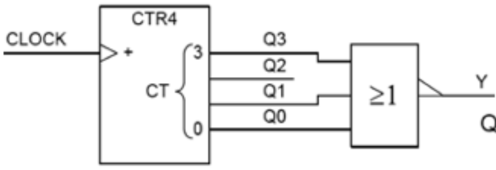
What if you want a counter that only counts from 0 to 9, then back to 0 again? This is really easy with Verilog. A simple if-else construct testing for the terminal counter of 9 will do the trick.

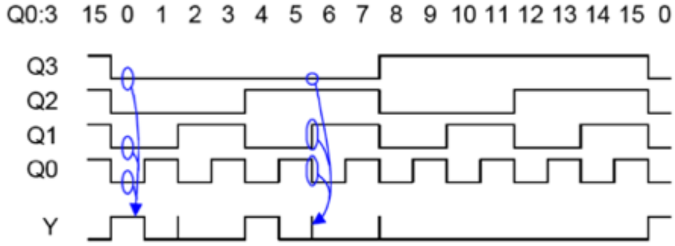
The actual implementation could use of 4-bit binary counter with a simple AND gate to detect when the counter value of 9 is reached, and then synchronously reset the counter back to 0.

The state transition diagram is as shown. Beware of what happens when count value is outside the range of 0 to 9.

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

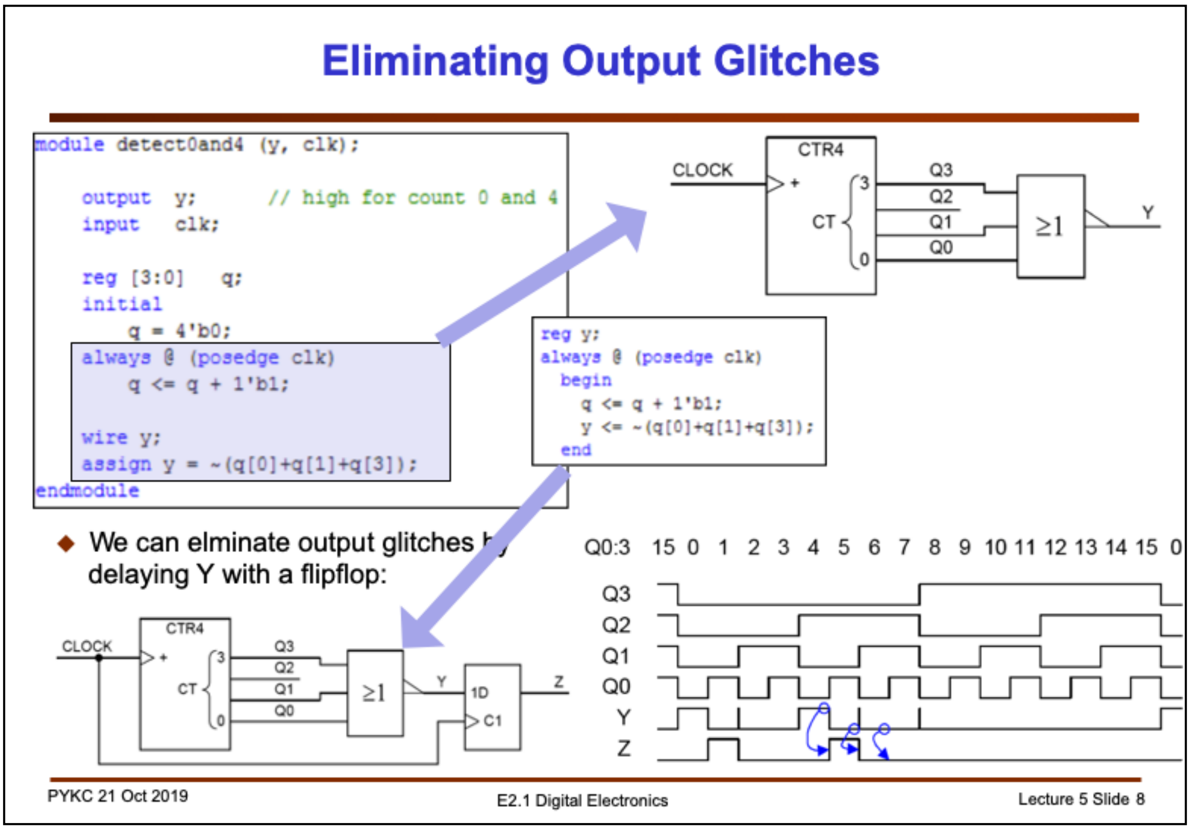
- Transition 1 → 2: Q=00?? which includes 0000
- Transition 5 → 6: Q=01?? which includes 0100
- Transition 7 → 8: Q=???? which includes both

PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 7

When you use gates to combine values from a counter, beware that you will get glitches at the output of the gates.

Here is an example where Y could contain glitches. This is because the outputs Q[3:0] will not change exactly at the rising edge of the clock signal. Instead, there will be some delays and the delays are likely not be the same for different Q outputs.

For example a count value transition from 7 to 8 could temporarily go to 1111 before going 1000. (i.e. 0111 -> 1111 -> 1000). Similarly transitions from 5 to 6 and 0 to 1 could result in glitches in Y.

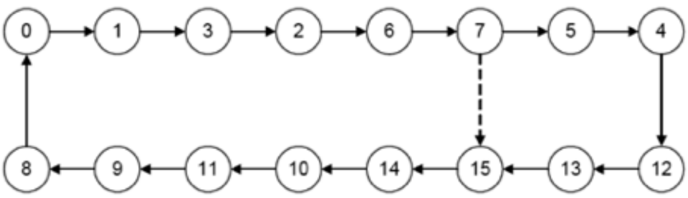


Eliminating glitches can be achieved by adding an extra D-FF at Y output. Z will be glitch free. However**, beware that Z is one cycle delayed relative to Y**.

If Z is instead of Y is to be used to control other circuits, we need to time synchronous all the signals. This may involve adding D-FFs to datapaths that are controlled by Z in order to make sure that data signals and control signals arrive at the same clock cycle.

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
```

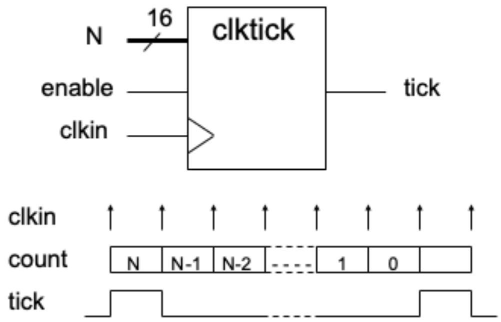
PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 9

If the one-cycle delay is not desirable and you need to avoid any glitches, one could use a special counter that counts in a special sequence that guarantees NOT to produce glitches when its count outputs are gated together. The count sequence is known as Gray code.

In a Gray code counter, successive values only have ONE bit changing. Here is a 4-bit gray code counter and it is specified as a case statement in Verilog.

A Flexible Timer – clktick.v

- 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;
```


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 which I can 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 **clkin**.

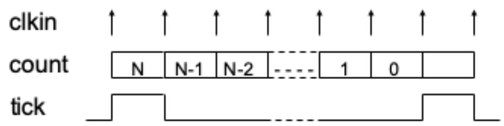
We also add an enable signal, which must be set to '1' in order to enable the internal counting circuit.

Shown below is the module interface for this circuit in Verilog.

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.v 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.



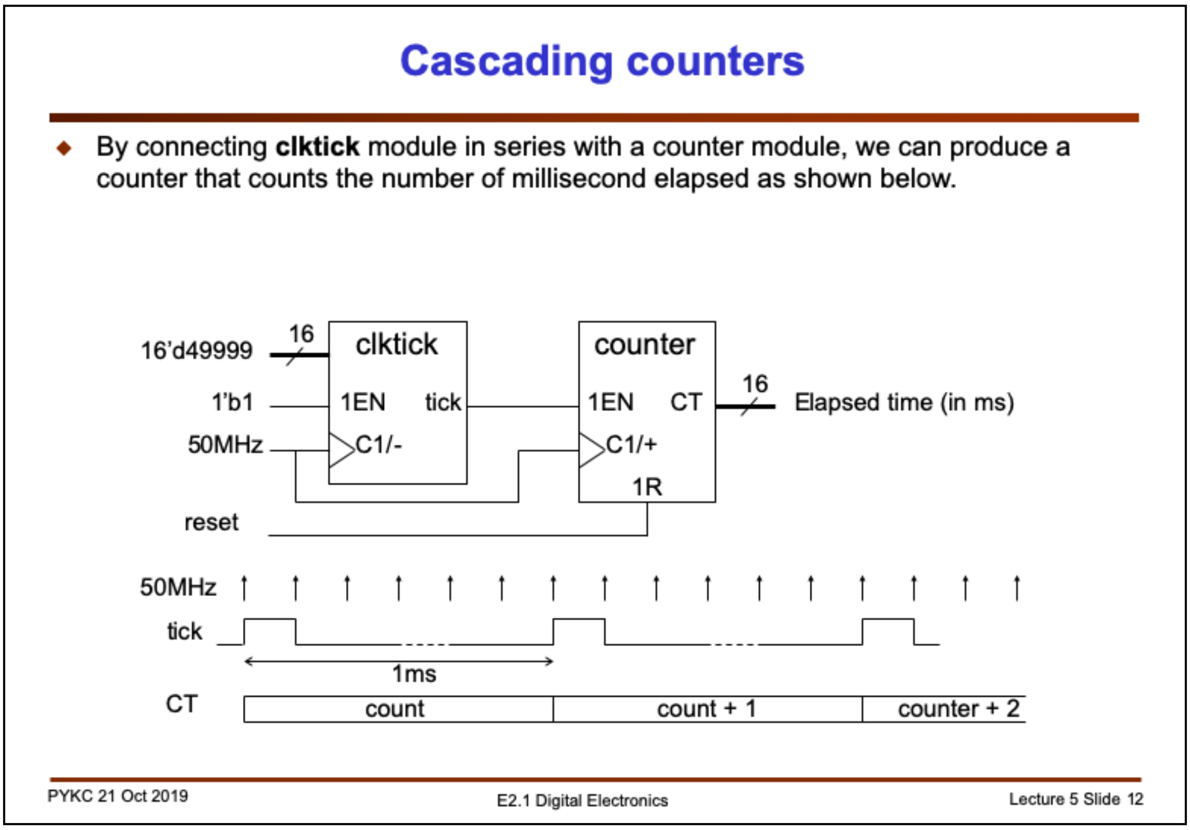

```verilog
//------------Output Ports Data Type-----------------
// Output port can be a storage element (reg) or a wire
reg [N_BIT-1:0] count;
reg             tick;

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
```


The actual Verilog 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** here 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.



Using this style of designing 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 the 50MHz clock. Therefore **tick** goes high for 20 microsecond once every 1 msec (or 1KHz).

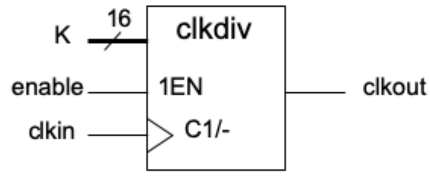
The **clktick** module is sometimes called a **prescaler** circuit. It prescale the input clock signal (50MHz) in order for the second counter to count at a lower frequency (i.e. 1KHz).

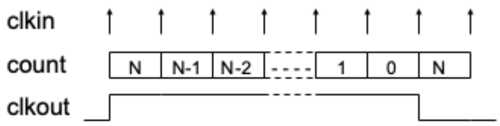
The second counter is now counting the number of millisecond that has elapsed since the last time reset signal (1R) goes high.

The design of this circuit is left as a tutorial problem for you to do.

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).





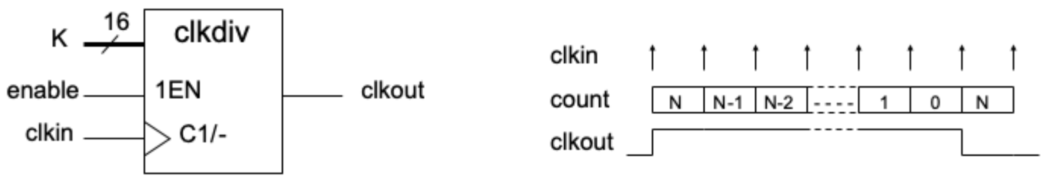

```
module clkdiv (
  clkin,          // Clock input signal to be divided
  enable,         // enable clk divider when high
  K,              // clock frequency divider is 2*(K+1)
  clkout       // symmetric clock output Fout = Fin / 2*(K+1)
);             //    End of port list

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;
```

PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 13

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 Verilog. Again we have used the **parameter** statement to make this design ease of modification for different internal counter size.

clkdiv.v explained

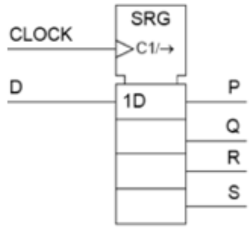


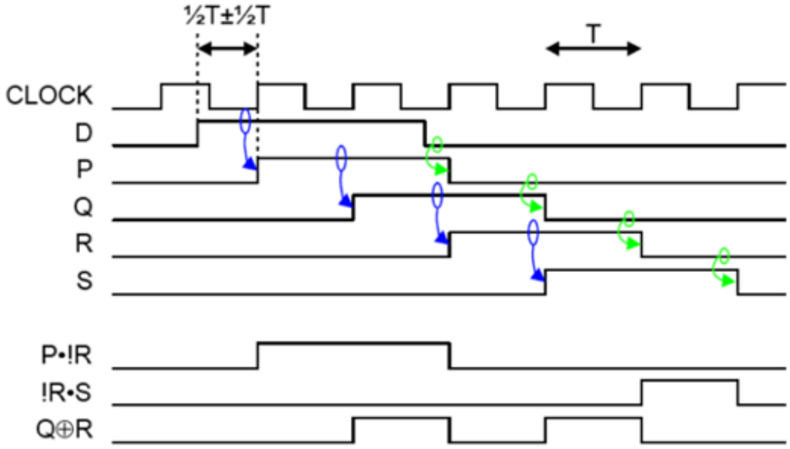

```
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [K_BIT-1:0] count;
reg             clkout;

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
```


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 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 & ↓

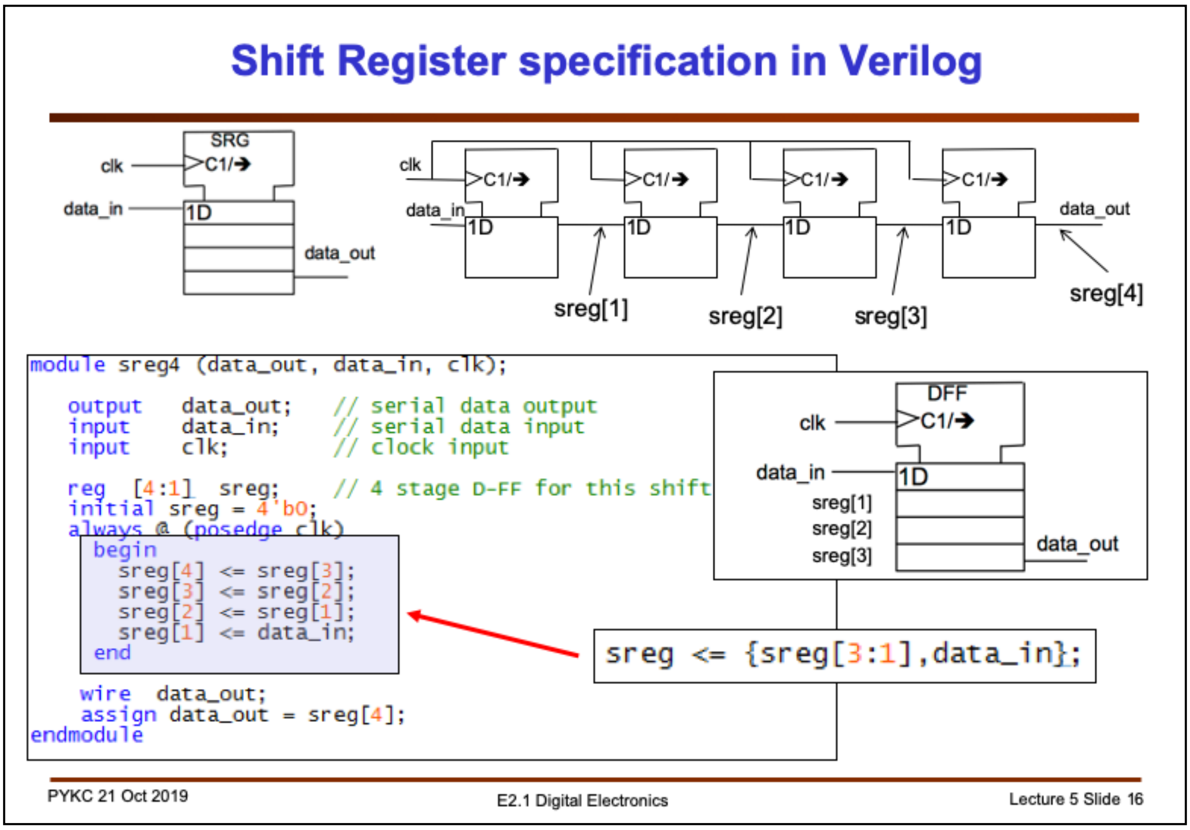
PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 15

Instead of producing binary signals using a counter, one could use a shift register to produce a sequence of pulses delayed relative to each other, and use gates to merge these together and produce different binary signals.

Shown here is a D-input to a shift register, producing P Q R and S, delayed from the previous signal by one clock cycle.

Using AND, NOT, OR or XOR gates, we can produce the various digital signals with different delays and pulse widths.

Beware that producing control signals this way may generate glitches.



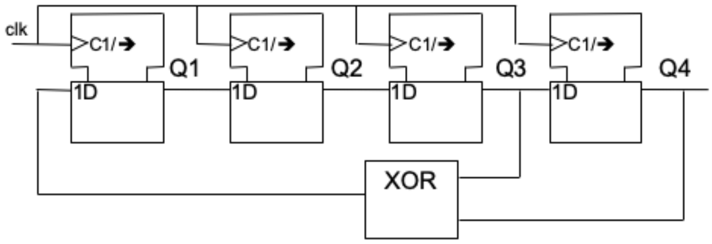
To specify a shift register in Verilog, use the code shown here (in blue box). We use the **<=** assignment to make sure that **sreg[4:1]** are updated only at the end of the **always** block.

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)



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.

PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 17

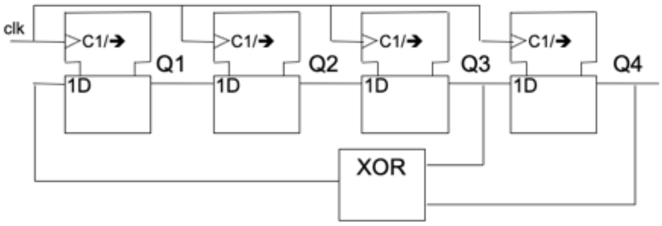
We can also make a shift register count in binary, but in an interesting sequence.

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**: $1 + X^3 + X^4$
- 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	$1+X+X^2+X^7+X^{24}$

PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 18

The circuit shown below is effective 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:

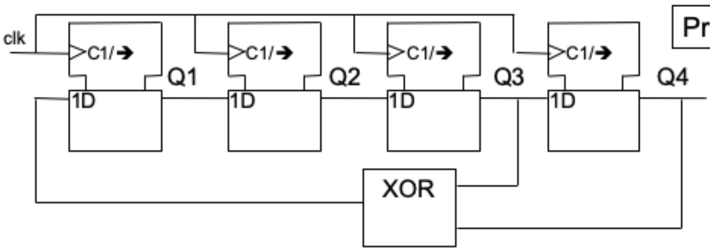
$$1 + X + X^{15}$$

This result 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 effective implements the equation:

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

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

lfsr4.v



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
```

PYKC 21 Oct 2019 E2.1 Digital Electronics Lecture 5 Slide 19

Here is the Verilog specification for a 4-bit LFSR. Note how the concatenation operator is used in the always block.

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

Quiz Questions

1. If the CLOCK period is T, what is the range of possible time delays between a change in the DATA input of a shift register and the resultant change in the output of the first stage?
2. How do you combine the outputs of a shift register to generate a pulse for both the rising and the falling edges of its input signal?
3. In order to guarantee that a shift register will notice a pulse on its DATA input, how long must a pulse last?
4. If an AND gate is used to combine 2 of the outputs from a 4-bit counter, how many different count values will make the AND gate output go high?
5. Why do output glitches not occur when a counter counts from 6 to 7?
6. Name two ways in which output glitches may be avoided.

Answers are all in the notes.