Timing Pulses

- Important element of laboratory electronics

- Pulses can control logical sequences with precise timing.

  → If your detector "sees" a charged particle or a photon, you might want to signal a clock to store the time at which it occurred.

  → You could use the event to generate a standard pulse so that your clock always responds in the same way.

- Alternatively, you might need to reset your electronics after the event

  → Clearly you want the reset pulse to arrive as soon as possible after the data has been processed

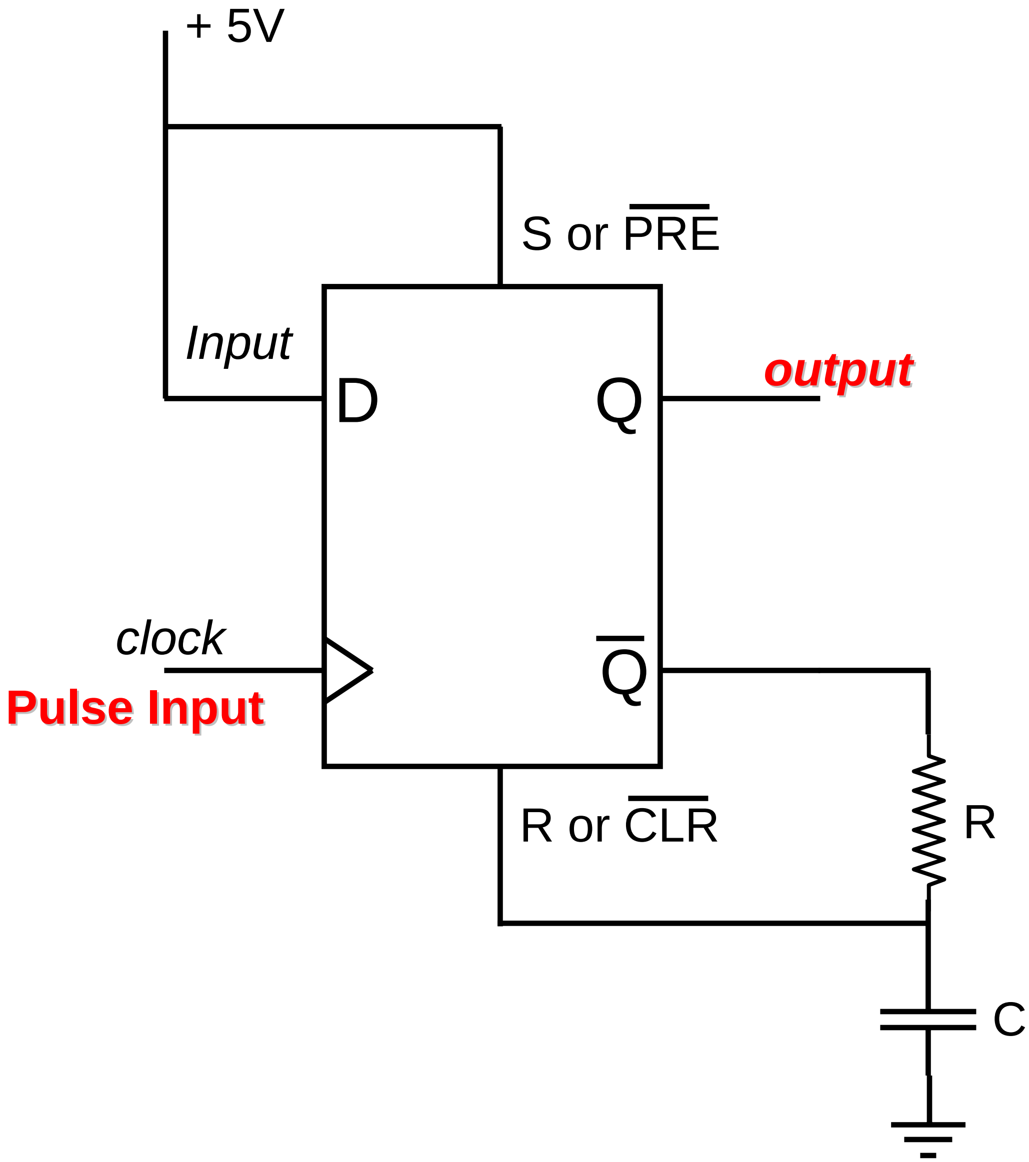
  → This requires a precision time *delay generator*

Timing Pulses

- A simple type of delay generator…
  1. A **D-type flip-flop** receives a clock edge and goes from low to high at the output
  2. The output charges up an **RC circuit** after going high.
  3. The charged capacitor also serves as the **clear input** to the D flip-flop.
  4. **So, that after a fixed time (roughly *RC*) the flip-flop resets back to its initial state.**
  5. The net result is a single pulse that has a duration (or *pulse width*) determined by the combination of the resistor & capacitor

- This is called a *monostable multivibrator* or *one-shot.*

One-shot: D-type flip-flop

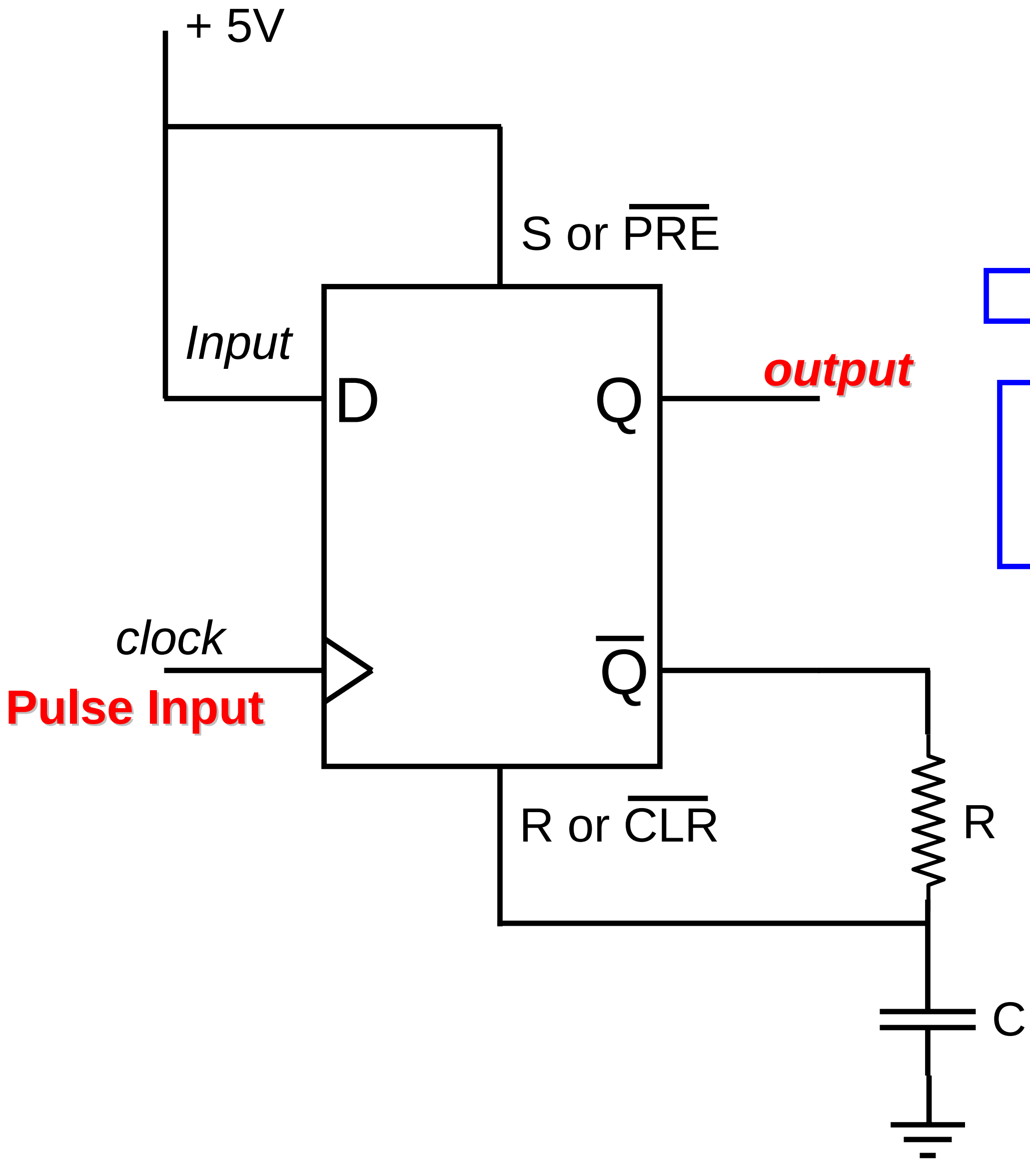


FUNCTION TABLE

INPUTS				OUTPUTS	
$\overline{PRE}$	$\overline{CLR}$	CLK	D	Q	$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H↑	H↑
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$

[Texas Instruments 74LS74 flip-flop datasheet]

One-shot: D-type flip-flop



FUNCTION TABLE

INPUTS				OUTPUTS	
$\overline{PRE}$	$\overline{CLR}$	CLK	D	Q	$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H†	H†
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$

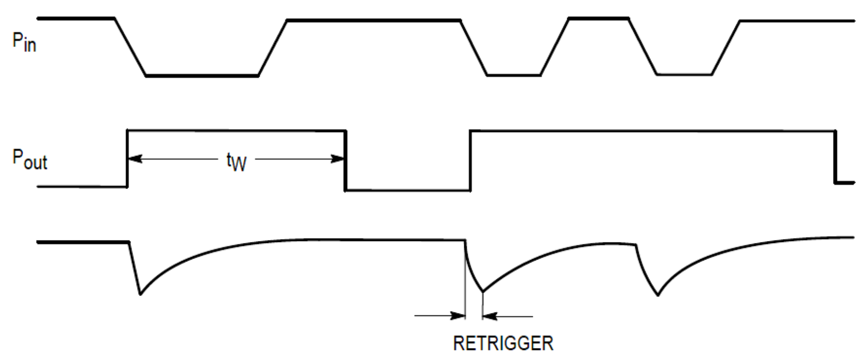
[Texas Instruments 74LS74 flip-flop datasheet]

One-shot: 74LS123

Characteristics:

- 2 clock inputs triggered by either a rising edge or a falling edge.
- 2 outputs (Q & $\overline{Q}$).
- A **reset or clear input**, instantly sets the output to a standard condition regardless of the current state or clock level.
- Can be confused a little by pulses in quick succession.

INPUTS			OUTPUTS	
CLEAR	A	B	Q	$\overline{Q}$
L	X	X	L	H
X	H	X	L	H
X	X	L	L	H
H	L	↑	⊓	⊔
H	↓	H	⊓	⊔
↑	L	H	⊓	⊔



One-shot: 74LS123

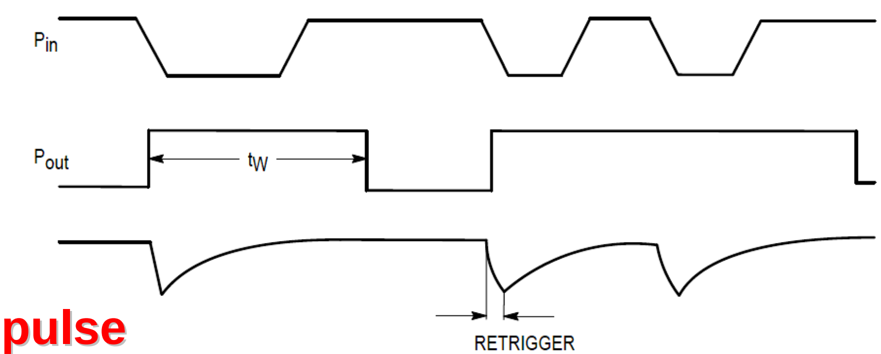
Characteristics:

- 2 clock inputs triggered by either a rising edge or a falling edge.
- 2 outputs (Q & $\overline{Q}$).
- A **reset or clear input**, instantly sets the output to a standard condition regardless of the current state or clock level.
- Can be confused a little by pulses in quick succession.

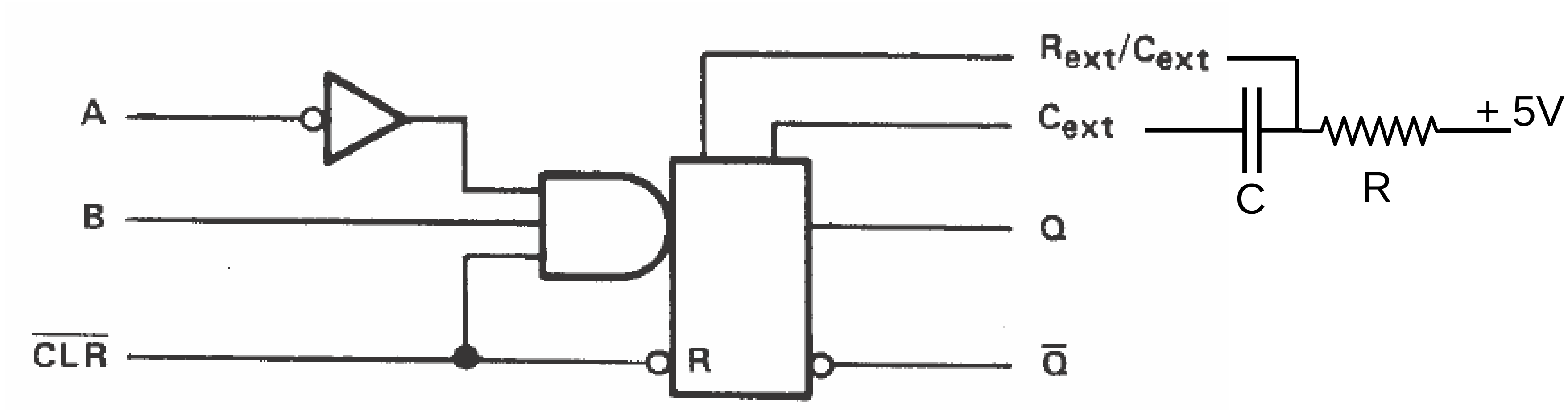
INPUTS			OUTPUTS	
CLEAR	A	B	Q	$\overline{Q}$
L	X	X	L	H
X	H	X	L	H
X	X	L	L	H
H	L	↑	⊓	⊔
H	↓	H	⊓	⊔
↑	L	H	⊓	⊔

armed (rows 1–3)

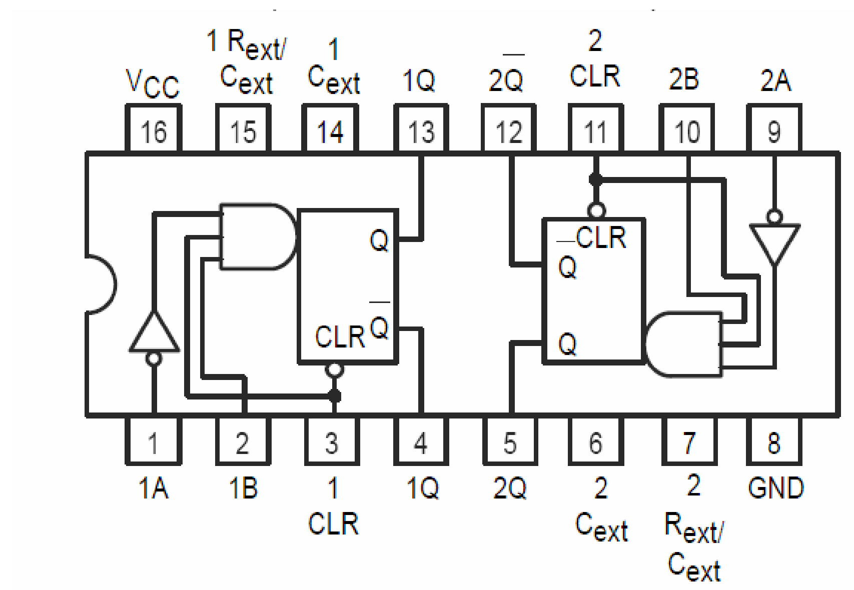
output pulse (rows 4–6)



74LS123 usage



INPUTS			OUTPUTS	
CLEAR	A	B	Q	$\overline{Q}$
L	X	X	L	H
X	H	X	L	H
X	X	L	L	H
H	L	↑	⊓	⊔
H	↓	H	⊓	⊔
↑	L	H	⊓	⊔



[Texas Instruments 74LS123 datasheet]

Pulse Delay Generator

- A **single one-shot** will produce a **variable delay** pulse.

- **2 one-shots** can be combined to produce a pulse of **variable duration/width** produced at **variable delay** after a trigger.

  → Pulse Delay Generator … very useful in a research lab.

Pulse Delay Generator

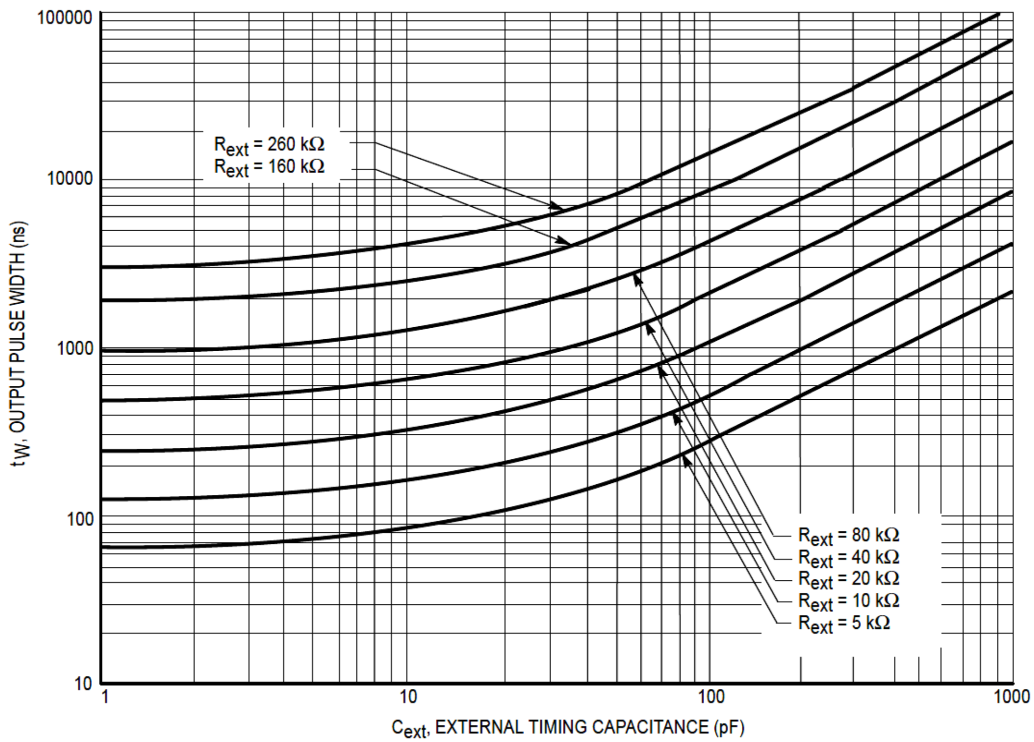
- A **single one-shot** will produce a **variable duration/width** pulse.

- **2 one-shots** can be combined to produce a pulse of **variable duration/width** produced at **variable delay** after a trigger.

  → Pulse Delay Generator … very useful in a research lab.

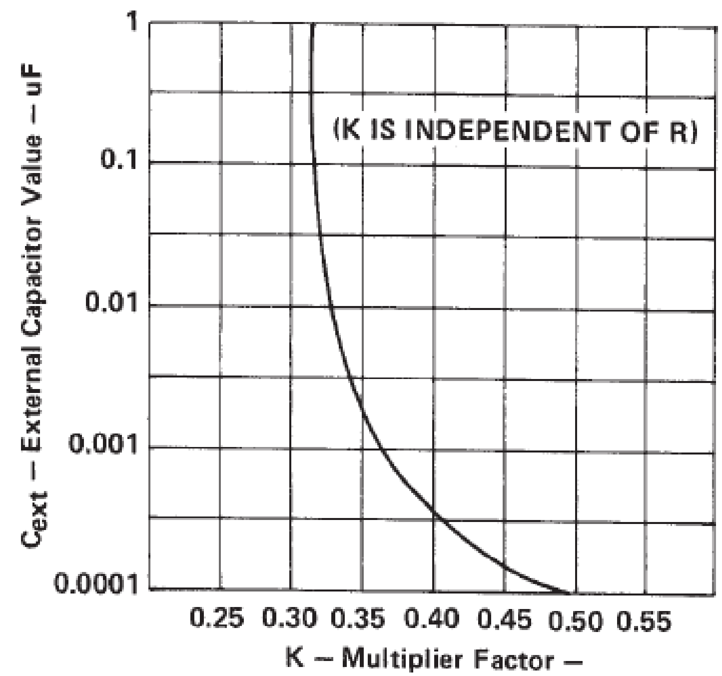
[image from www.thinksrs.com]

Setting the Pulse Width



$$t_W = K R_{ext} C_{ext}$$

with



The Problem with One-Shots

1. One-shots are very useful in a research laboratory as pulse delay generator.

2. **One-shots should be avoided in regular circuitry, because**

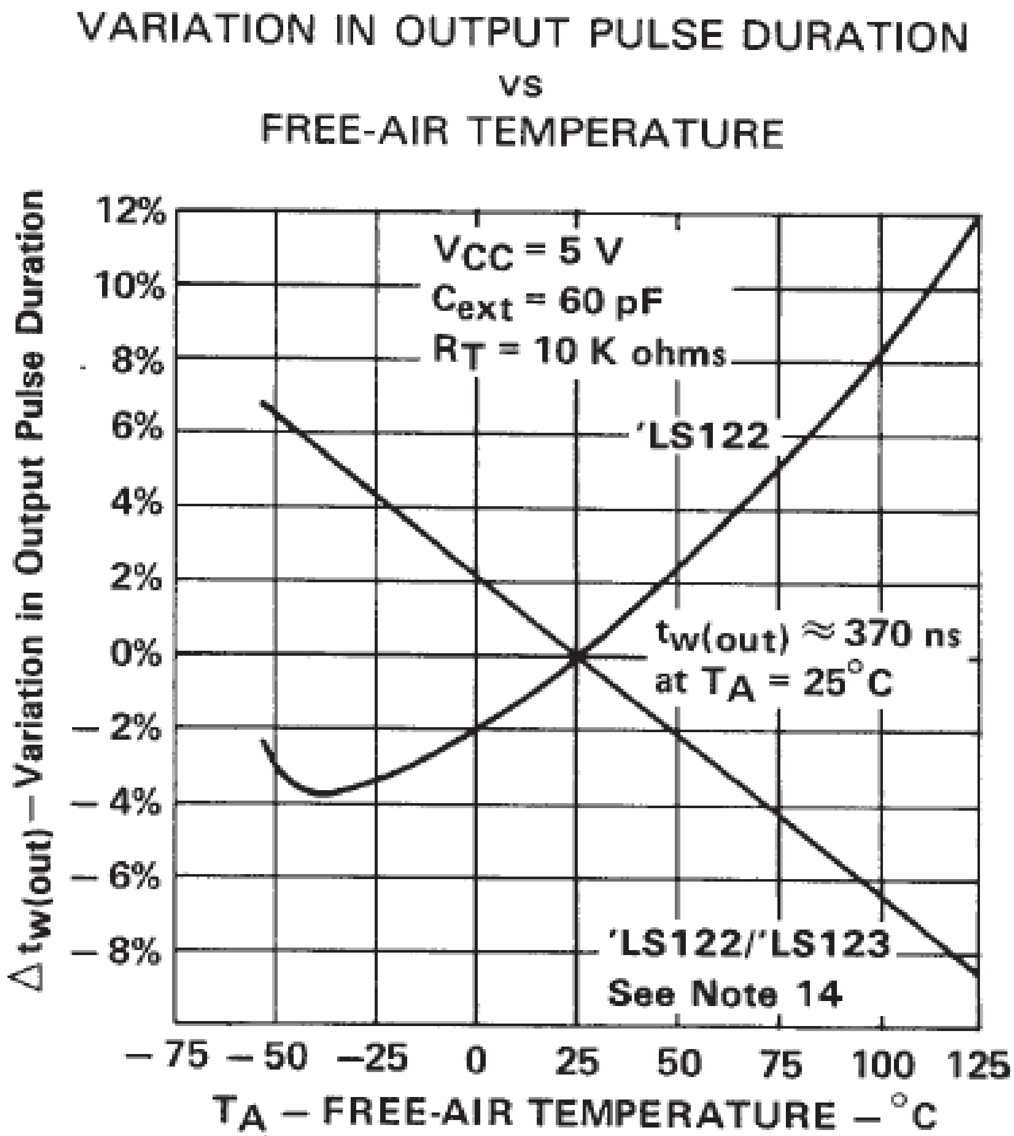
→ They are useful in **asynchronous circuits** for avoiding glitches and signal races …

→ It's very easy to put them all over your asynchronous circuit with all the pulse timing set just right. It is very hard to figure out how the circuit works just by looking at it (or even a circuit diagram).

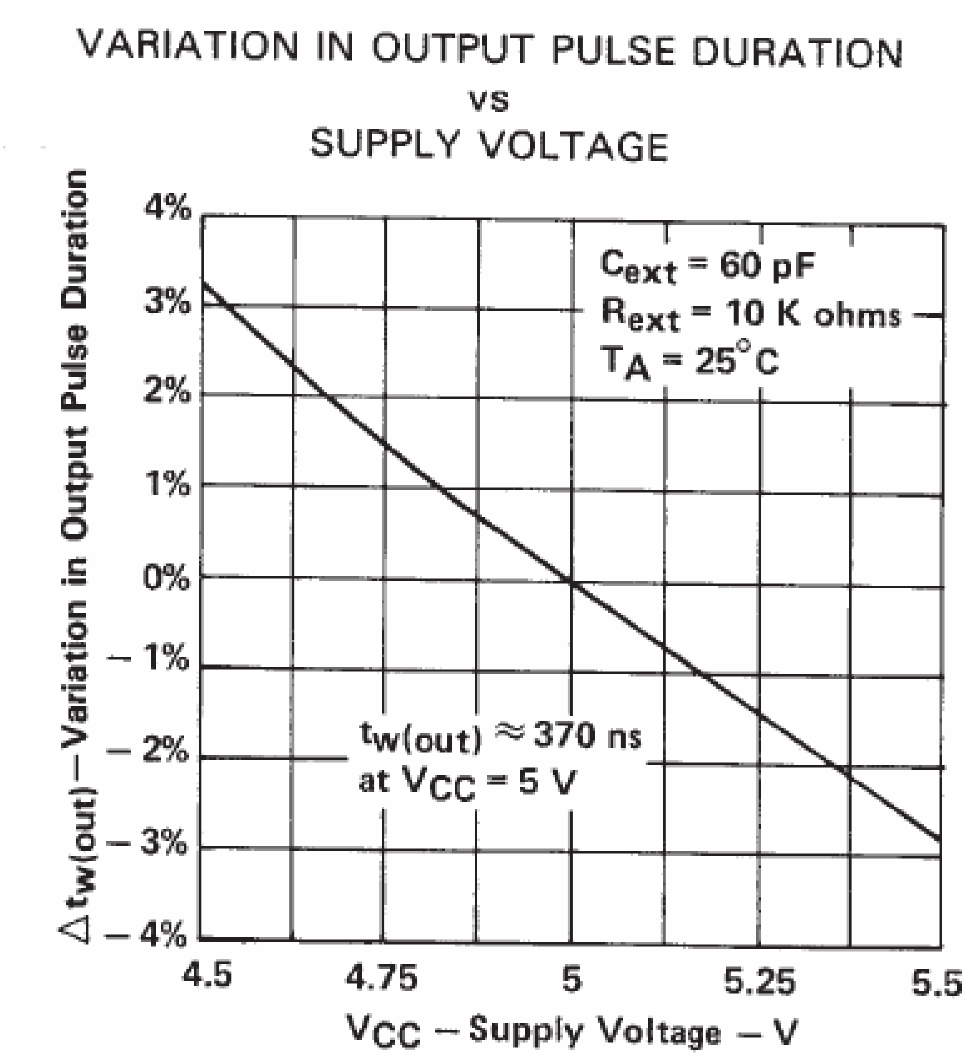
→ The pulse width depends on temperature (R, C, and chip).

→ The pulse width depends on supply voltage.

Pulse Width vs. Temperature



Pulse Width vs. Supply Voltage



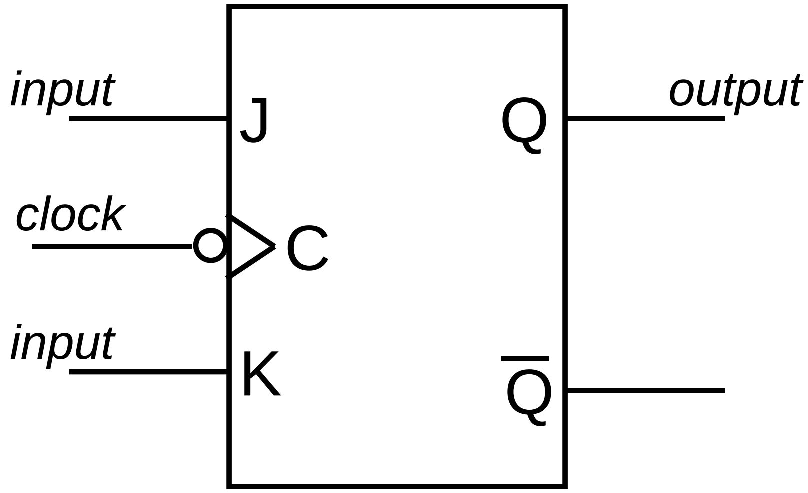
Counters … 1 2 3

1. Frequency dividers.

2. Ripple counter.

3. Synchronous counter.

JK-type flip-flop

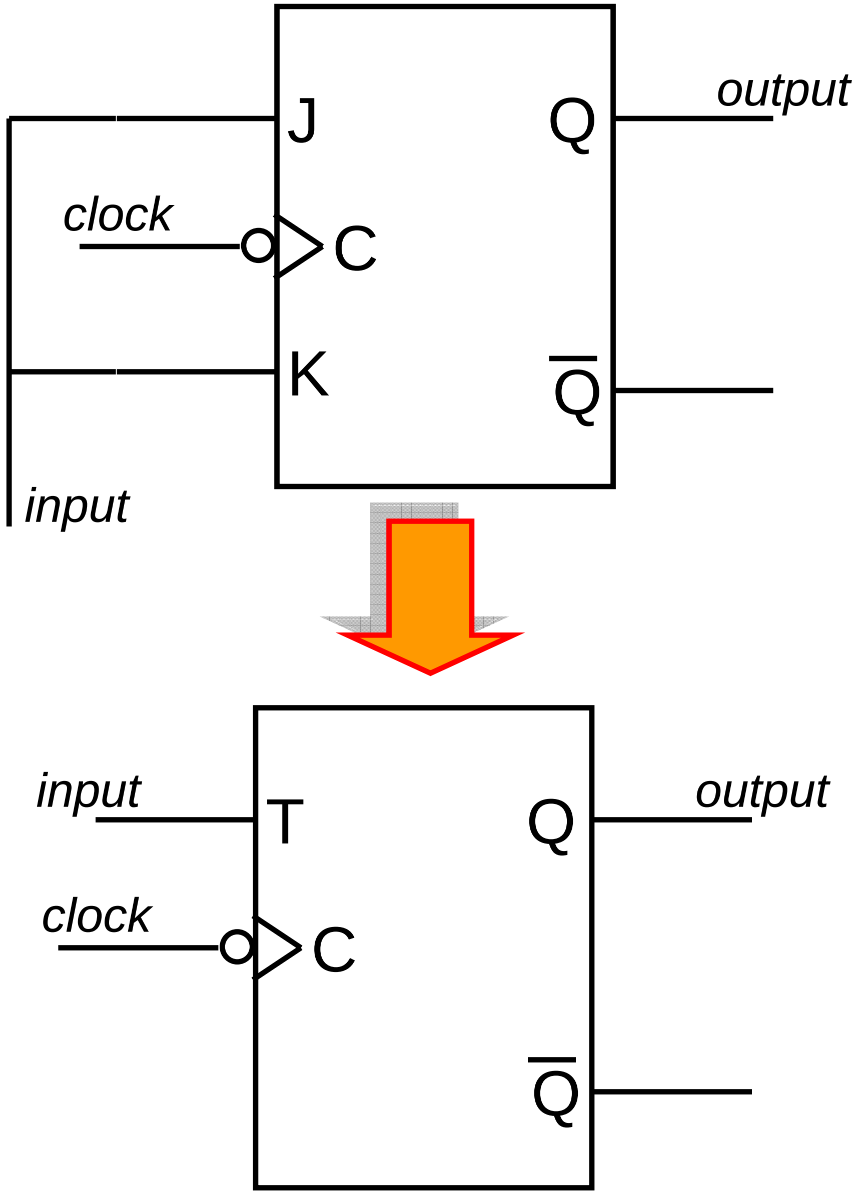


Logic table
for clock falling edge

J	K	Q_{n+1}
0	0	Q_n
1	0	0
0	1	1
1	1	$\overline{Q}_n$

JK-type flip-flops are used in counters.

T-type flip-flop



JK Logic table

J	K	Q_{n+1}
0	0	Q_n
1	0	0
0	1	1
1	1	$\overline{Q_n}$

T Logic table
(clock falling edge)

T	Q_{n+1}
0	Q_n
1	$\overline{Q_n}$

T-type flip-flops are used in counters.

Counters in Verilog

Counters in Verilog are easy → just use `always` (synchronous).

→ and a self-referential "add 1" assignment.

```
module counter_v3(input1,output1);      // module for an 8-bit synchronous counter
    input input1;                       // 1-bit input
    output reg [7:0] output1;           // 8-bit output register

    always@ (posedge input1)            // synchronous loop, clocked on input1 rising edge
        begin

        output1 <= output1 + 1;         // self-referential add+1 assignment.
//      output1 = output1 + 8'b00000001;   could have used this instruction line instead.

        end

endmodule
```

Initializing a register

```
module counter_v3(input1,output1);       // module for an 8-bit synchronous counter
    input input1;                       // 1-bit input
    output reg [7:0] output1;           // 8-bit output register

    initial                             // this block initializes the output register
        begin                           // to zero.
        output1 = 8'b00000000;
        end

    always@ (posedge input1)            // synchronous loop, clocked on input1 rising edge
        begin

        output1 <= output1 + 1;         // self-referential add+1 assignment.
//      output1 = output1 + 8'b00000001;  could have used this instruction line instead.

        end

endmodule
```

Initializing a register

```
module counter_v3(input1,output1);        // module for an 8-bit synchronous counter
    input input1;                       // 1-bit input
    output reg [7:0] output1;           // 8-bit output register

    initial                             // this block initializes the output register
        begin                           // to zero.
        output1 = 8'b00000000;
        end

    always@ (posedge input1)            // synchronous loop, clocked on input1 rising edge
        begin

        output1 <= output1 + 1;         // self-referential add+1 assignment.
//      output1 = output1 + 8'b00000001;   could have used this instruction line instead.

        end

endmodule
```

This section initializes the register to zero.
(Code should not rely on this too much!)

“if” statement

```
module counter_v3(input1,output1, output2);     // module for an 8-bit synchronous counter
    input input1;                          // 1-bit input
    output reg [7:0] output1;              // 8-bit output register
    output reg [2:0] output2;              // 3-bit output register

    initial                                // this block initializes the output registers
        begin                              // to zero.
        output1 = 8'b00000000;
        output2 = 3'b000;
        end

    always@ (posedge input1)               // synchronous loop, clocked on input1 rising edge
        begin

        output1 <= output1 + 1;            // self-referential add+1 assignment.
//      output1 = output1 + 8'b00000001;  could have used this instruction line instead.

        if (output1 <= 8'b11100111)
            begin
            output2 = 3'b000;   // output2 stays at zero for output1 <= 231.
            end
        else
            begin
            output2 = output2 + 1;  // output2 starts counting for output1 > 231.
            end

        end

endmodule
```

Variable Registers

```
module counter_v3(input1,output1, output2);      // module for an 8-bit synchronous counter
    input input1;                       // 1-bit input
    output reg [7:0] output1;           // 8-bit output register
    output reg [2:0] output2;           // 3-bit output register
    reg [1:0] temp;     // "temp" variable 2-bit register.

    initial                             // this block initializes the output registers
        begin                           // to zero.
        output1 = 8'b00000000;
        output2 = 3'b000;
        temp = 2'b00;
        end

    always@ (posedge input1)            // synchronous loop, clocked on input1 rising edge
        begin

        output1 <= output1 + 1;         // self-referential add+1 assignment.

        temp <= temp + 1;               // temp is used as counter.

        if (output1 <= 8'b11100111)
            begin
            output2 = 3'b000 + temp;    // output2 counts continuously to 2 for output1 <= 231.
            end
        else
            begin
            output2 = output2 + 1 ; // output2 starts counting for output1 > 231.
            end

        end

endmodule
```

Recommendation: check the Technology Map Viewer after compiling.

The “function” command (I)

```
// Module translates 2-bit inputs to an 8-bit output code
module Input_to_Output_converter(input_register1, input_register2, input_register3,
                                 output_register1, output_register2, output_register3);
    input [1:0] input_register1;
    input [1:0] input_register2;
    input [1:0] input_register3;
    output reg [7:0] output_register1;
    output reg [7:0] output_register2;
    output reg [7:0] output_register3;

    always
        begin

        // 1st output
        output_register1 = output_8bit(input_register1);

        // 2nd output
        output_register2 = output_8bit(input_register2);

        // 3rd output
        output_register3 = output_8bit(input_register3);

        end

```

The "function" command (II)

```
// This function defines the output codes given a 2-bit input
    function [7:0] output_8bit;
        input [1:0] input_number_2bit;
        begin
        output_8bit = 7'b1111111;

        if (input_number_2bit == 4'b00)
            output_8bit = 7'b1111111;

        if (input_number_2bit == 4'b01)
            output_8bit = 7'b1001011;

        if (input_number_2bit == 4'b10)
            output_8bit = 7'b1000000;

        if (input_number_2bit == 4'b11)
            output_8bit = 7'b0000000;


        end

    endfunction

endmodule
```