BEE 271 Digital circuits and systems

Spring 2017

Lecture 9: Sequential circuits

Nicole Hamilton

https://faculty.washington.edu/kd1uj

Topics

1. Decoders
2. Undefined variables in Verilog
3. Introduction to sequential circuits

Decoders in Verilog

Select one of many outputs
or transform data.

```
module Decode2to4A( input [ 1:0 ] s, input enable,
      output reg [ 0:3 ] f );

   always @( * )
      if ( enable )
         case ( s )
            0: f = 4'b1000;
            1: f = 4'b0100;
            2: f = 4'b0010;
            3: f = 4'b0001;
         endcase
      else
         f = 4'b0000;

endmodule
```

A 2-to-4 binary decoder.

```
module Decode2to4B( input [ 1:0 ] s, input enable,
      output reg [ 0:3 ] f );

   always @( * )
      case ( { enable, s } )
         3'b100:  f = 4'b1000;
         3'b101:  f = 4'b0100;
         3'b110:  f = 4'b0010;
         3'b111:  f = 4'b0001;
         default: f = 4'b0000;
      endcase

endmodule
```

A 2-to-4 binary decoder.

```
module Decode2to4C( input [ 1:0 ] s, input enable,
      output reg [ 0:3 ] f );

   always @( * )
      casex ( { enable, s } )
         3'b0xx:  f = 4'b0000;
         3'b100:  f = 4'b1000;
         3'b101:  f = 4'b0100;
         3'b110:  f = 4'b0010;
         3'b111:  f = 4'b0001;
      endcase

endmodule
```

A 2-to-4 binary decoder.

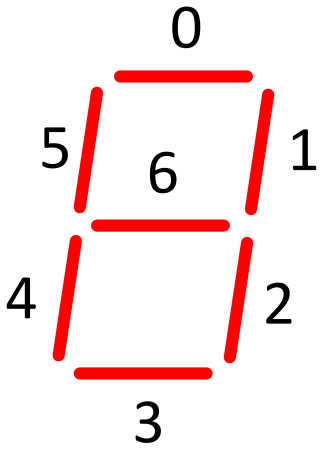
```
module Decode4to16A( input [ 3:0 ] s, input enable,
      output reg [ 0:15 ] f );

   wire [ 0:3 ] m;

   Decode2to4 d1( s[ 3:2 ], enable, m[  0:3  ] );
   Decode2to4 d2( s[ 1:0 ], m[ 0 ], f[  0:3  ] );
   Decode2to4 d3( s[ 1:0 ], m[ 1 ], f[  4:7  ] );
   Decode2to4 d4( s[ 1:0 ], m[ 2 ], f[  8:11 ] );
   Decode2to4 d5( s[ 1:0 ], m[ 3 ], f[ 12:15 ] );

endmodule
```

A 4-to-16 binary decoder.

```
module Decode4to16B(input [ 3:0 ] s, input enable,
      output reg [ 0:15 ] f );

   assign f = enable ? 1 << ( 15 - s ) : 0;

endmodule
```

A 4-to-16 binary decoder.

```
module SevenSegment ( input [ 3:0 ] hex,
   output reg [ 0:6 ] segments );

   always @( hex )
        case ( hex )
             0: segments = 7'b1111110;
             1: segments = 7'b0110000;
             2: segments = 7'b1101101;
             3: segments = 7'b1111001;
             4: segments = 7'b0110011;
             5: segments = 7'b1011011;
             6: segments = 7'b1011111;
             7: segments = 7'b1110000;
             8: segments = 7'b1111111;
             9: segments = 7'b1111011;
            10: segments = 7'b1110111;
            11: segments = 7'b0011111;
            12: segments = 7'b1001110;
            13: segments = 7'b0111101;
            14: segments = 7'b1001111;
            15: segments = 7'b1000111;
        endcase

endmodule
```



Exercise: Write a Verilog module that will count leading zeroes in an 8-bit value.

Exercise: Write a Verilog module that will count leading zeroes in an 8-bit value.

```
module clz ( input [ 7:0 ] u,
      output reg [ 3:0 ] count );

   always @( * )
      casex ( u )
         8'b1xxx_xxxx: count = 0;
         8'b01xx_xxxx: count = 1;
         8'b001x_xxxx: count = 2;
         8'b0001_xxxx: count = 3;
         8'b0000_1xxx: count = 4;
         8'b0000_01xx: count = 5;
         8'b0000_001x: count = 6;
         8'b0000_0001: count = 7;
         8'b0000_0000: count = 8;
      endcase

endmodule
```

Undefined variables in Verilog

Sometime you get an error.

Sometimes you don’t.

```
// Some scaffolding to allow tests of undefined variables
// and default values using the LEDs on the DE1-SoC boards.

module Display( input a, output [ 9:0 ] LEDR );

   // if a = 0, displays as 0001110000
   // if a = 1, displays as 0100100111

   assign LEDR[ 9:8 ] = a,
          LEDR[ 7:6 ] = !a,
          LEDR[ 5:4 ] = ~a,
          LEDR[ 3:2 ] = !( !a ),
          LEDR[ 1:0 ] = -a );

endmodule
```

```
module NoError1( output [ 9:0 ] LEDR );

   // Generates "created implicit net" warning and
   // causes zork to have the value 0.

   Display d( zork, LEDR );

endmodule
```

```
module NoError2( output [ 9:0 ] LEDR );

   // Causes zork to have the value 0.

   wire zork;
   Display d( zork, LEDR );

endmodule


module NoError3( output [ 9:0 ] LEDR );

   // Causes zork to have the value 0.

   reg zork;
   Display d( zork, LEDR );

endmodule
```

```
module Initialize1( output [ 9:0 ] LEDR );

   // Causes zork to have the initial value 1.

   wire zork = 1;
   Display d( zork, LEDR );

endmodule


module Initialize2( output [ 9:0 ] LEDR );

   // Causes zork to have the initial value 1.

   reg zork = 1;
   Display d( zork, LEDR );

endmodule
```

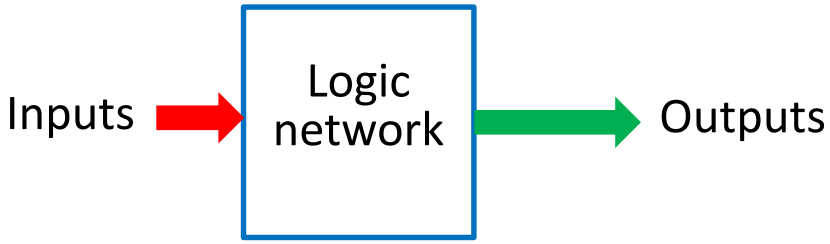
```
module CompileError( output [ 9:0 ] LEDR );

    // Generates an "object "zork" is not declared"
    // compile error if the wire definition is commented out.

    // wire zork;

    assign LEDR[ 9:8 ] = zork;
    assign LEDR[ 7:6 ] = !zork;
    assign LEDR[ 5:4 ] = ~zork;
    assign LEDR[ 3:2 ] = !( !zork );
    assign LEDR[ 1:0 ] = -a;

endmodule
```

Chapter 5

Flip-Flops, Registers, and Counters

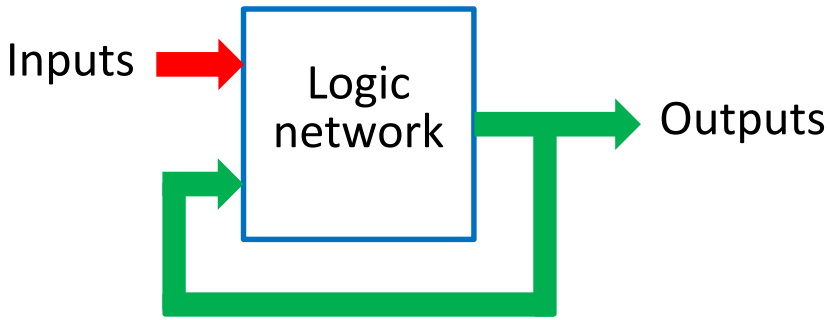
Combinatorial vs. Sequential Logic

Combinatorial or combinational logic



1. No memory elements.
2. No feedback from the outputs to the inputs.
3. Outputs depend only on the inputs.

Sequential logic



1. Contains memory that can remember a present state.
2. Outputs feed back to the inputs.
3. Outputs depend on both inputs and the present state.

Two problems

1. Would like to save state, e.g., count things.
2. Need to wait until signals settle

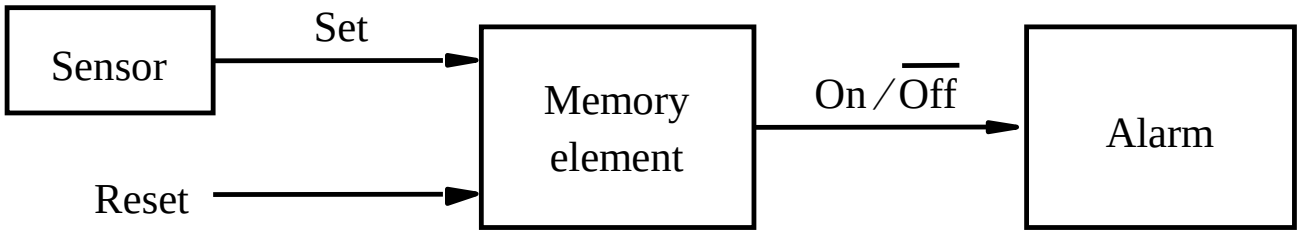
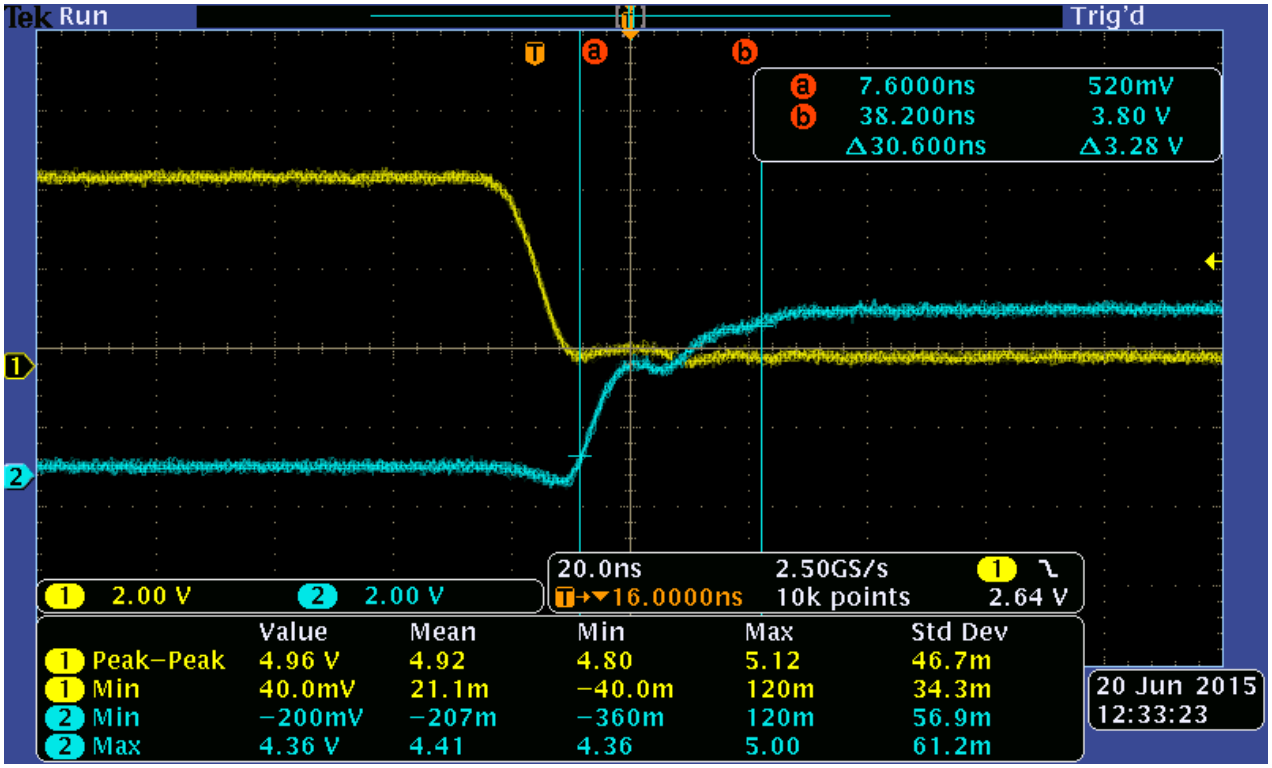


Figure 5.1. Control of an alarm system.

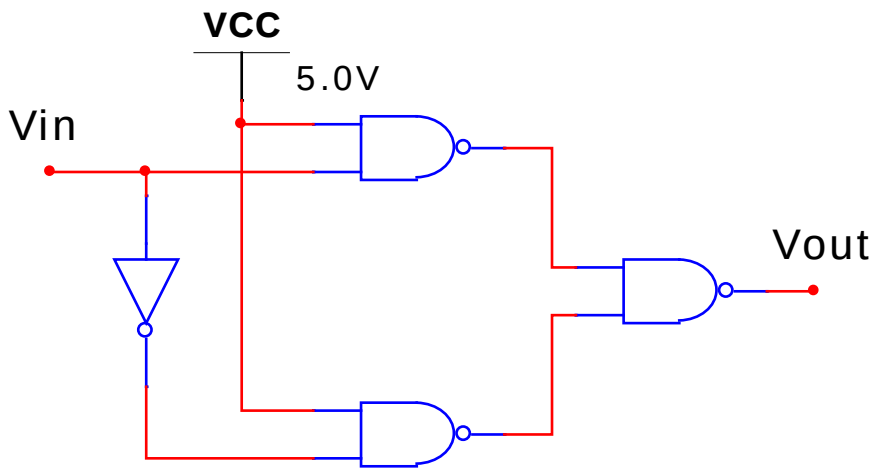
Real gates have propagation and rise and fall times.

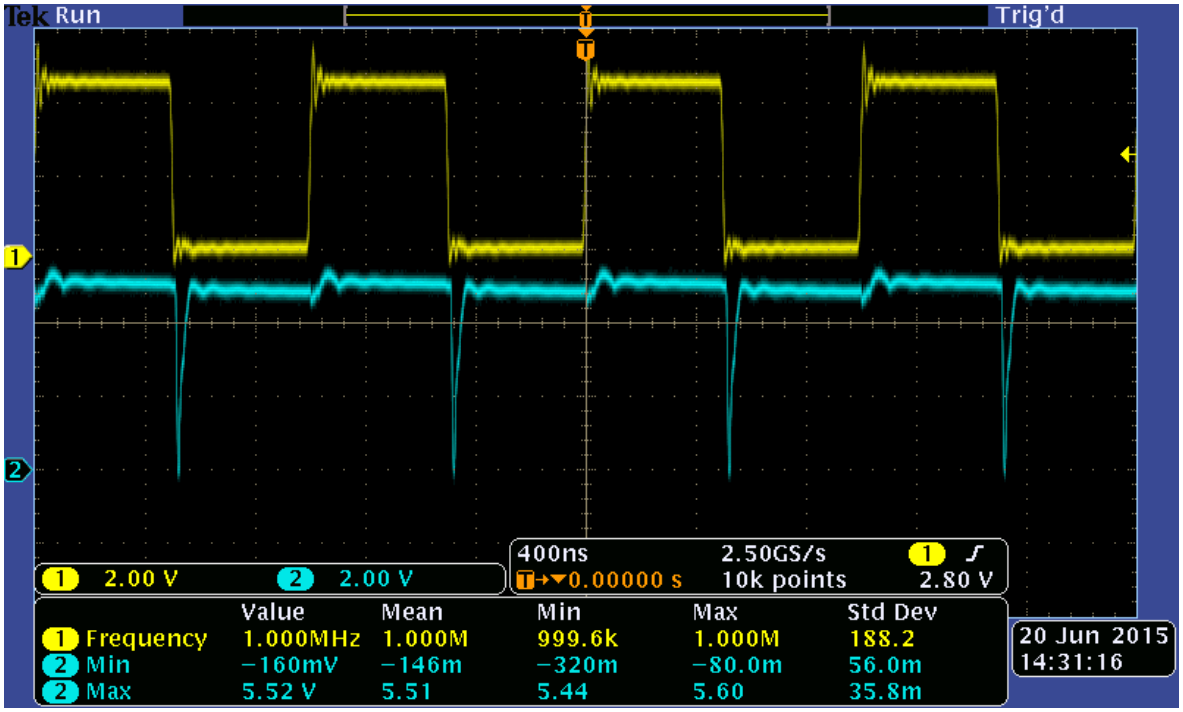
Causes hazards where outputs change when they shouldn't.

Must wait until signals have settled.



A circuit with a hazard in lab 1.

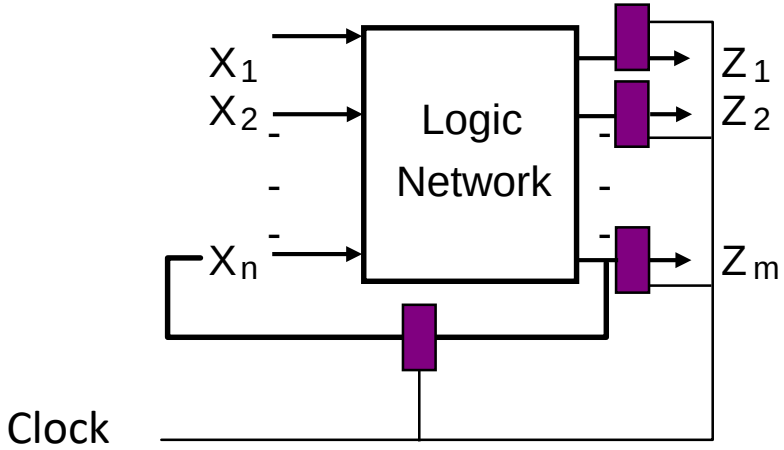


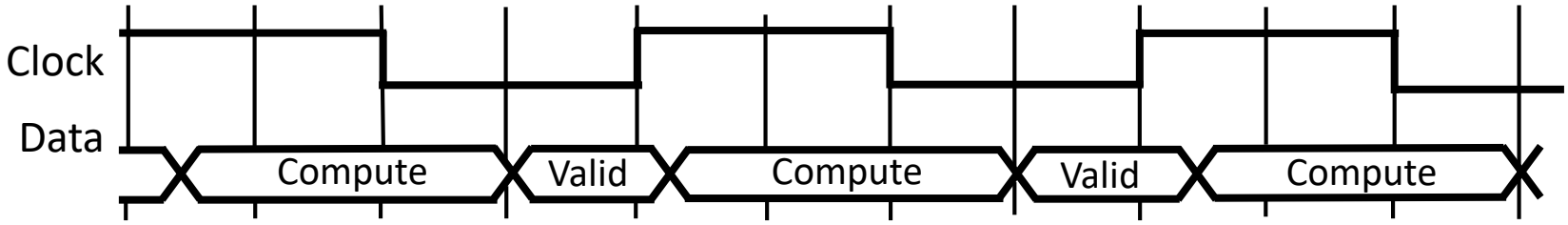


Safe Sequential Circuits

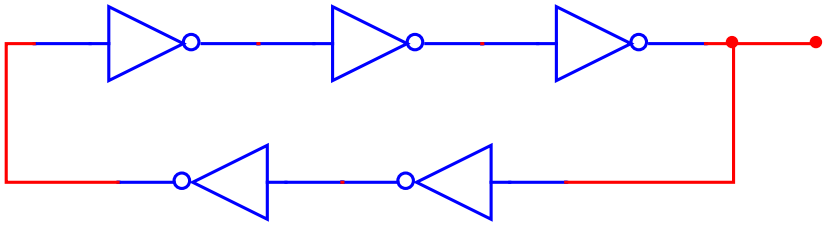
Clocked elements on feedback, perhaps outputs

- Clock signal synchronizes operation
- Clocked elements hide glitches/hazards

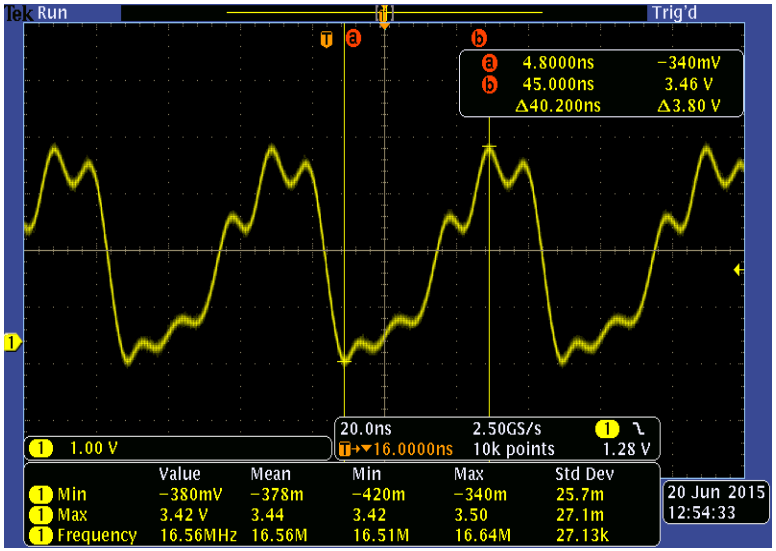


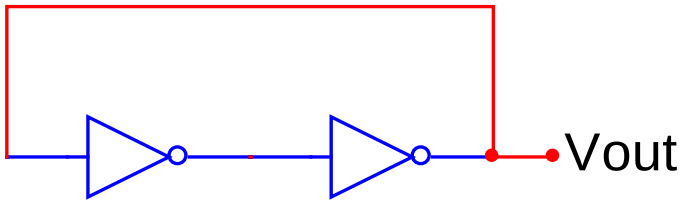


Feedback



Ring oscillator with an odd number of inverters

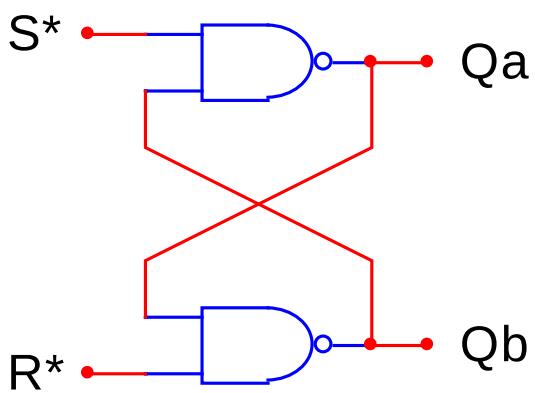


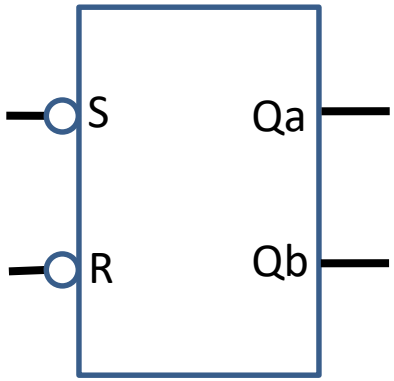


Bistable latch with an even number of inverters

Vout can be either 1 or 0 but whatever it is will be stable.

Set/reset latches





S'R' latch

S*	R*	Qa	Qb
0	0	1	1
0	1	1	0
1	1	no change	no change
1	0	0	1

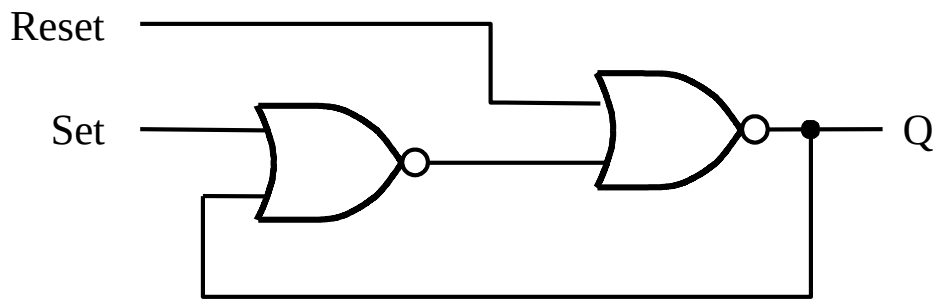


Figure 5.3. A memory element with NOR gates.

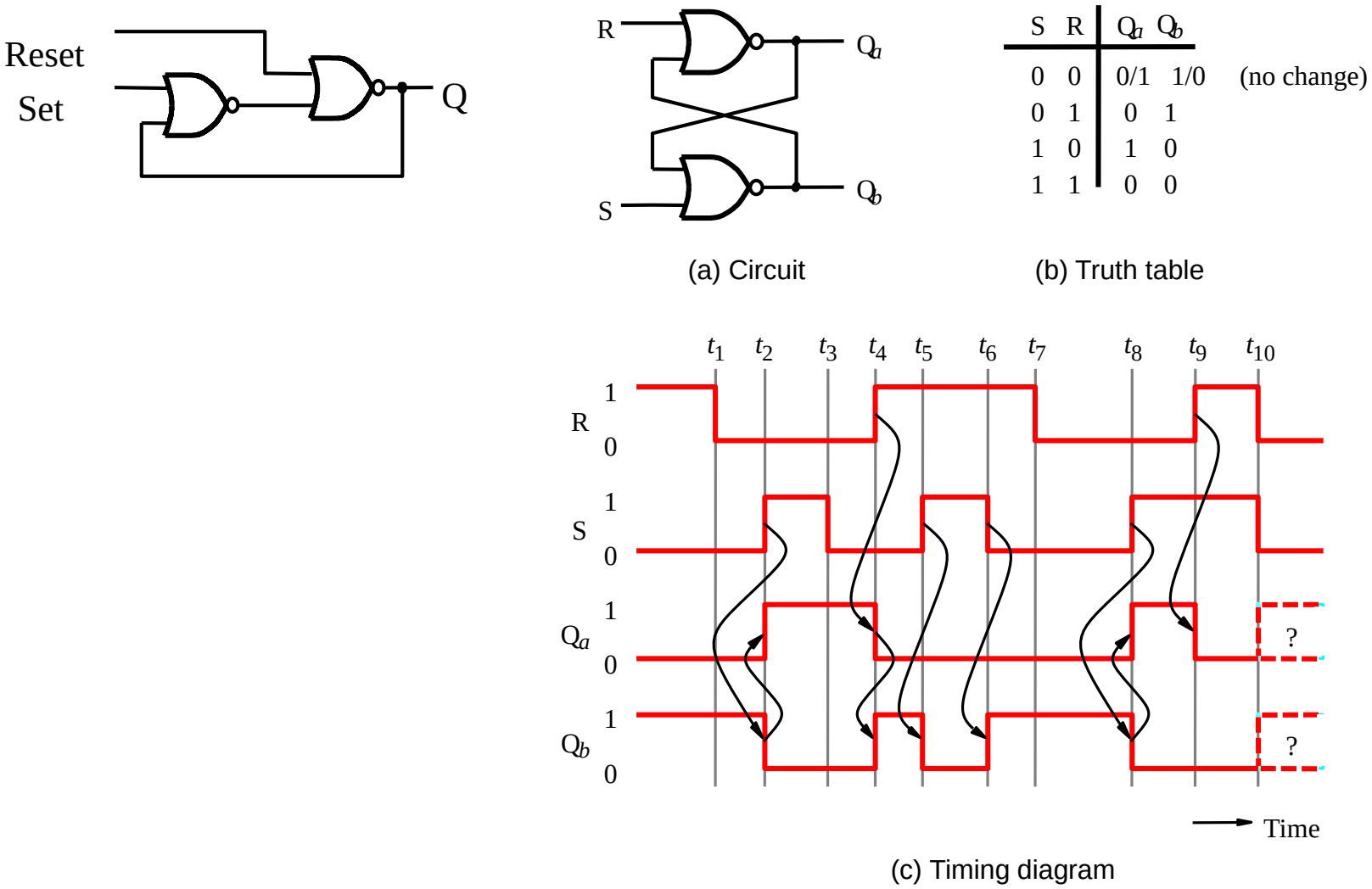


Figure 5.4. A basic latch built with NOR gates.

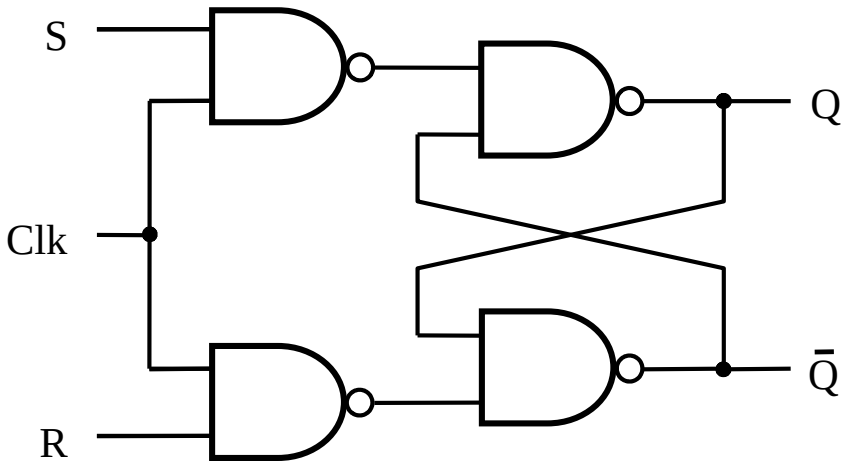
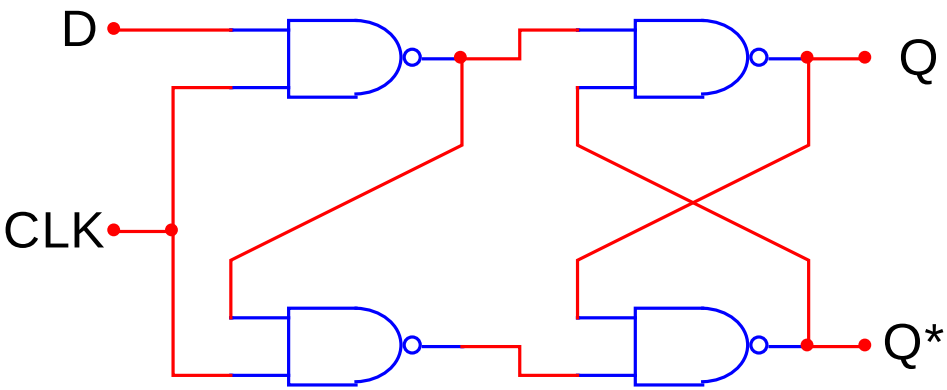
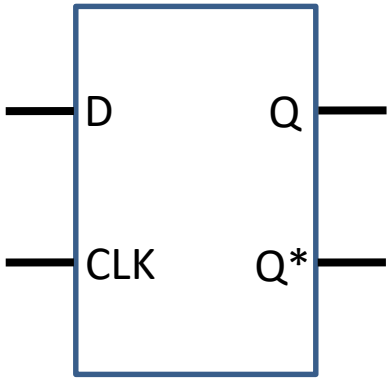


Figure 5.6. Gated SR latch with NAND gates.

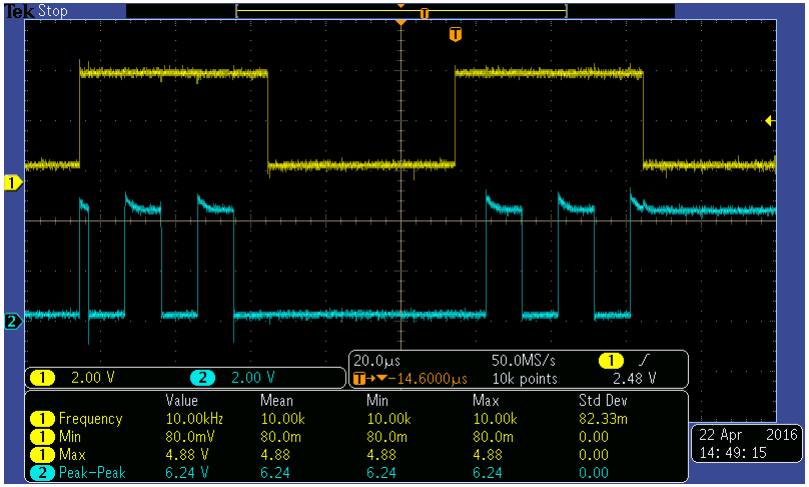
Gated latches



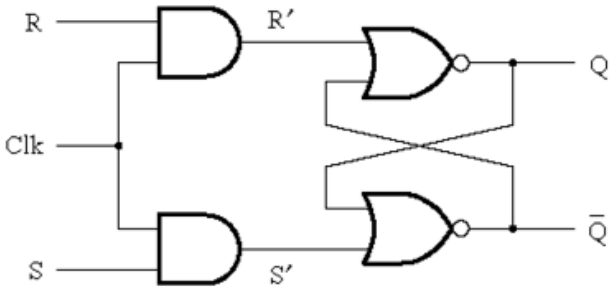


Gated D latch

Clock used to sample the input when the clock is high and hold it when the clock is low.



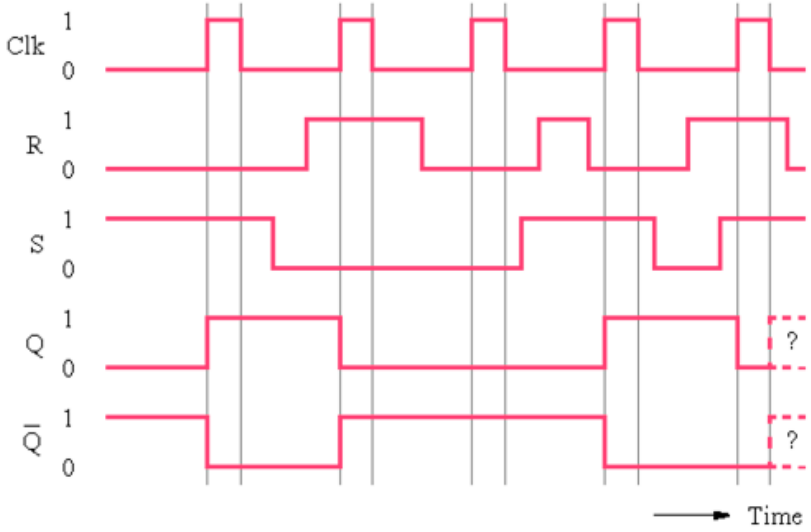
clock = 10 KHz / D = 52 KHz



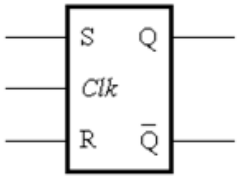
(a) Circuit

Clk	S	R	Q(t + 1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

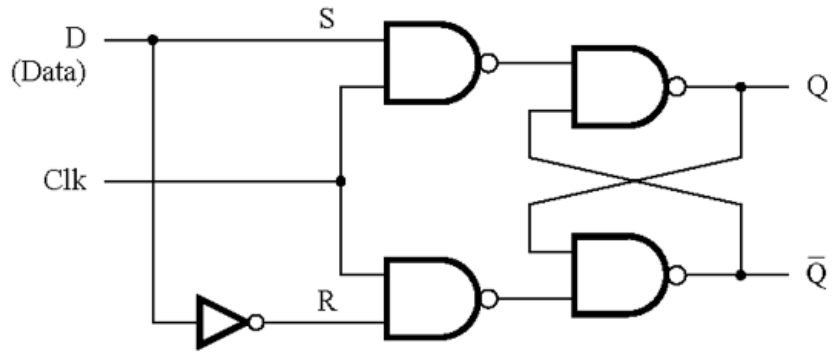


(c) Timing diagram



(d) Graphical symbol

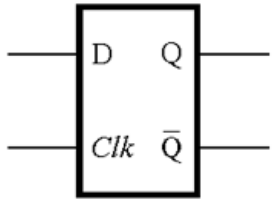
Figure 5.5.
Gated SR latch.



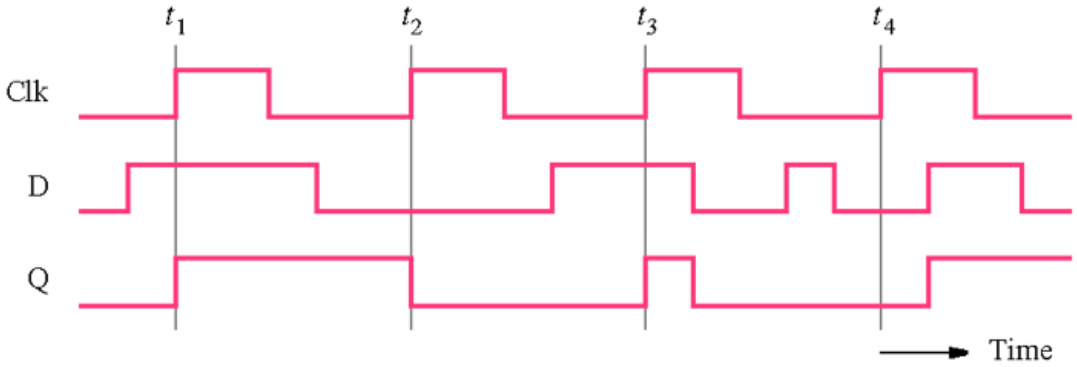
(a) Circuit

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

(b) Characteristic table



(c) Graphical symbol



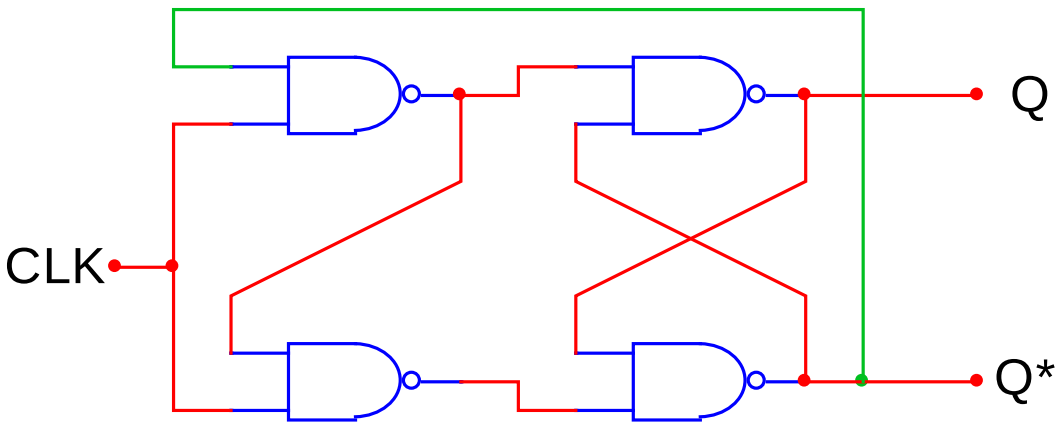
(d) Timing diagram

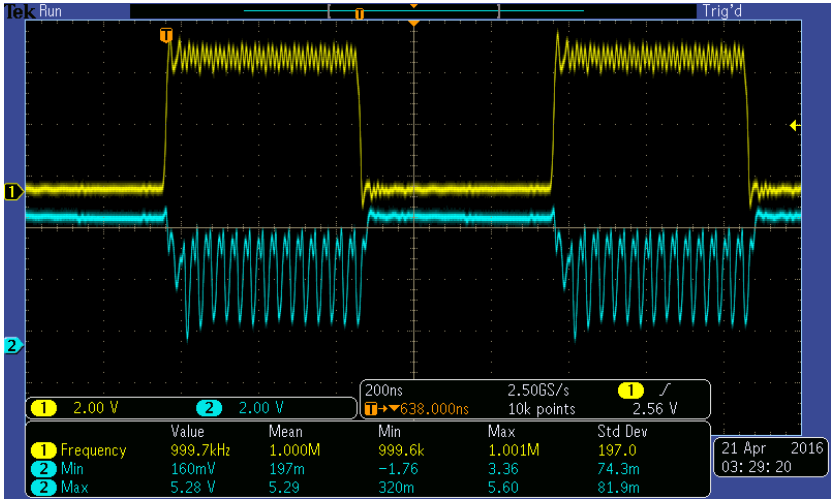
Figure 5.7. Gated D latch.

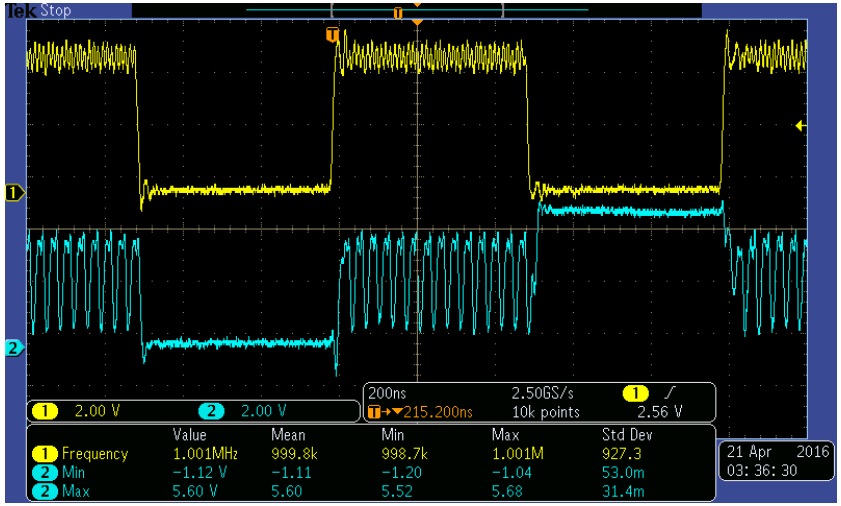
Flip Flops

- Problem: latches are sensitive to any changes that occur with the input while the clock or control signal is high
  - Glitches/Hazards
  - Unsynchronized changes
- Solution: use flip-flops, devices that react only on the clock edge

But we cannot build reliable counters or anything else requiring feedback with latches.







What you get depends on small differences in the length of the green wire.

Solution is to isolate inputs from outputs

Three popular alternatives:

1. Two-phase clocking
2. Master-slave
3. Edge-triggered

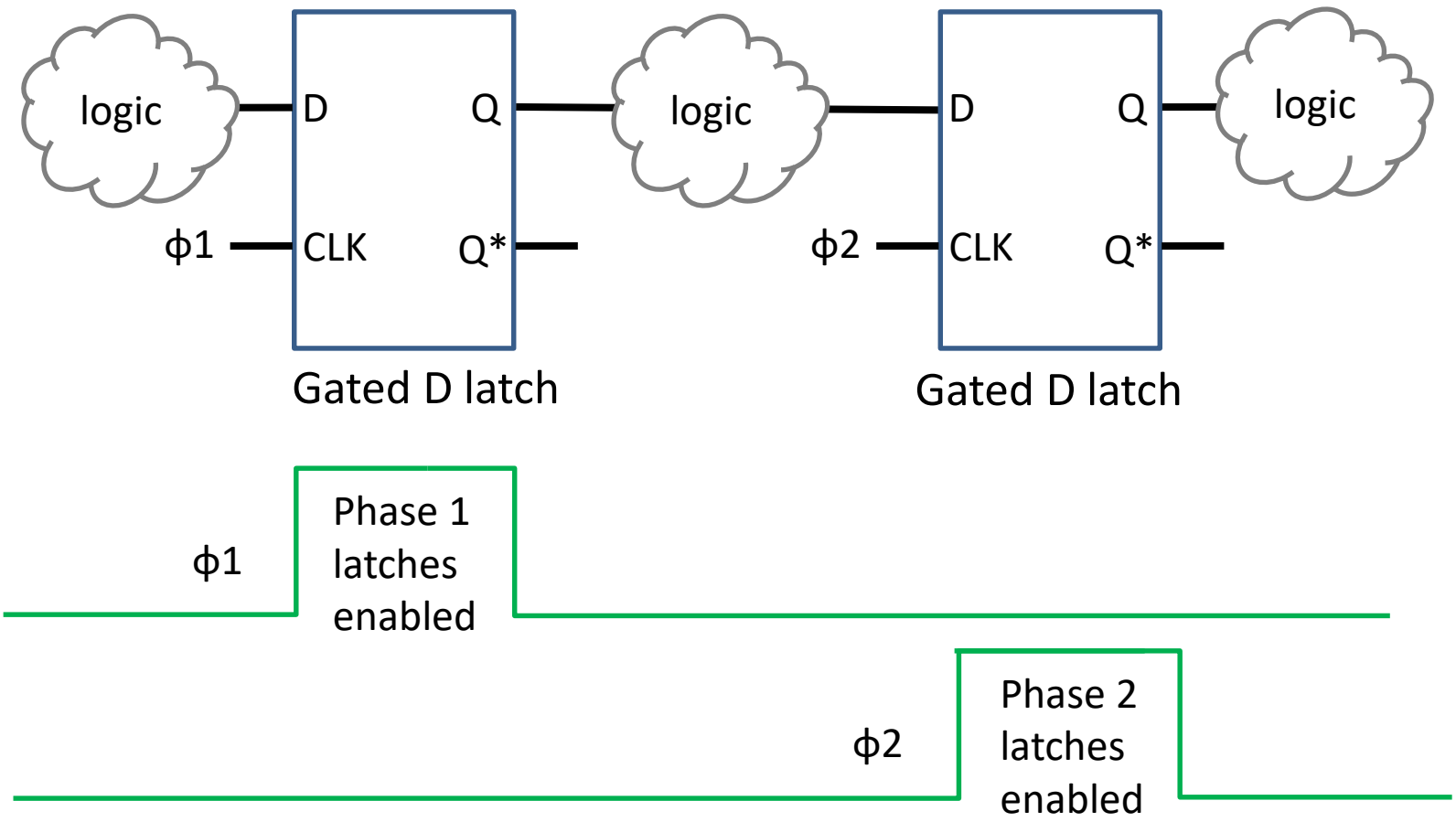
Terminology

Basic latch: A feedback connection of two NORs or two NANDs that can store 1 bit. Set using the S input and reset using the R input.

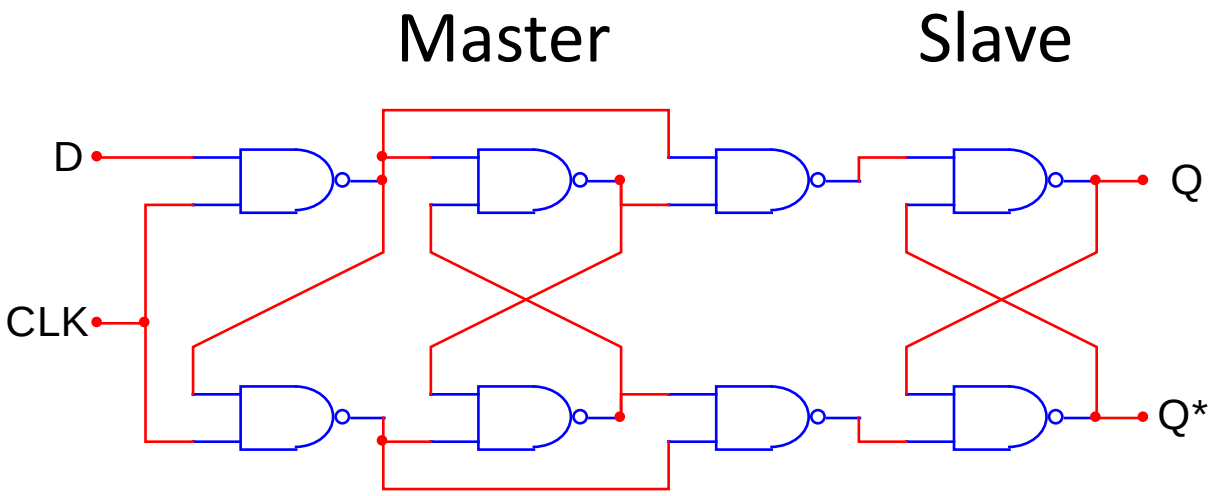
Gated latch: A basic latch that includes input gating and a control input signal. Retains its value when the control signal is 0, changes state when the control signal is 1.

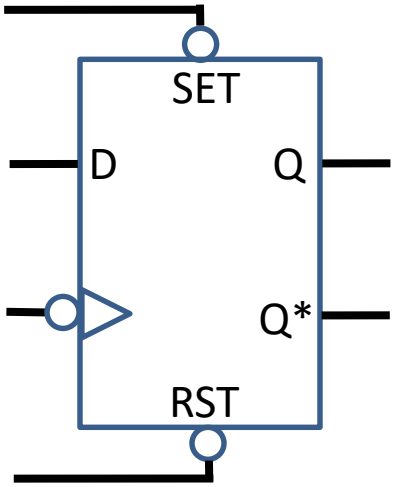
Flip-flop: A storage element whose output changes only on the edge of a controlling clock. Can be either positive or negative edge-triggered.

Two-phase clock

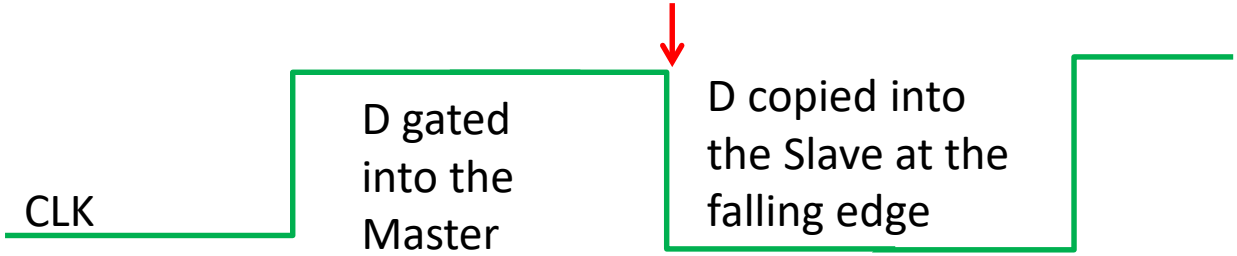


Non-overlapping clocks for phases 1 and 2. Advantages are fewer gates per bit and that you can put logic between both phases so long phase 1 latches only use phase 2 inputs and vice versa.



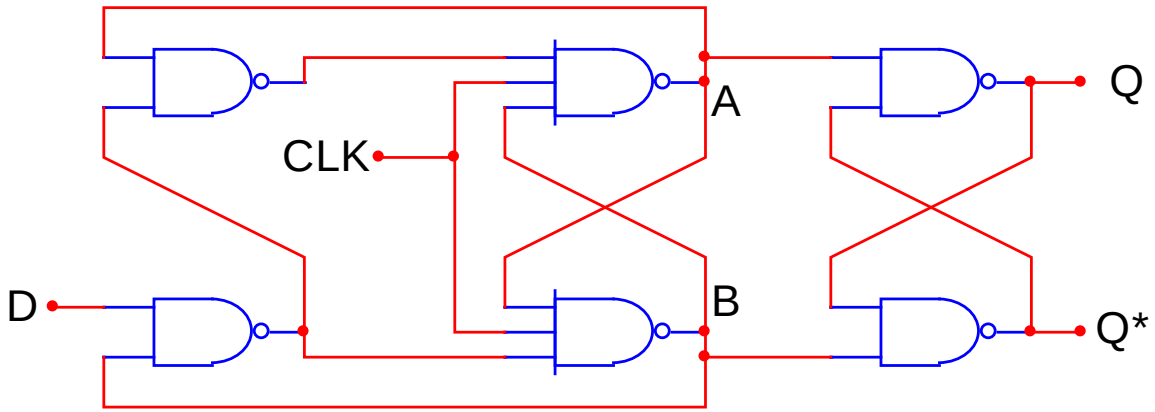


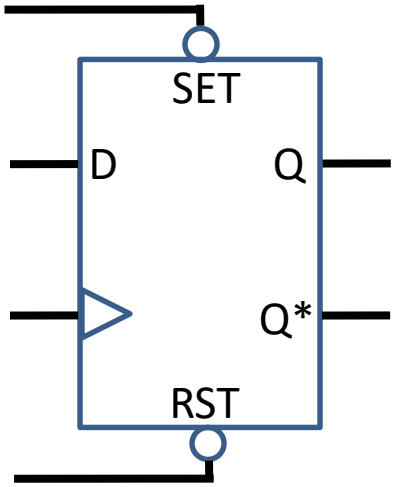
Master-slave
D flip-flop



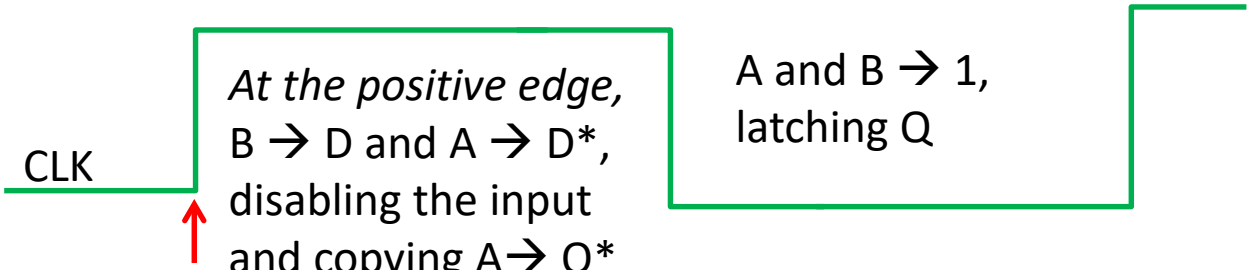
Because of the signal path is longer to the slave, we can guarantee the input to the master will be disabled before the input to the slave is enabled.

Edge-triggered





Edge-triggered
D flip-flop



At the positive edge,
B → D and A → D*,
disabling the input
and copying A→ Q*
and B → Q

A and B → 1,
latching Q

Requires a hold time: D is not allowed to change until the rising edge of the clock has propagated through A or B back to the input NANDs, latching whichever side was a zero.