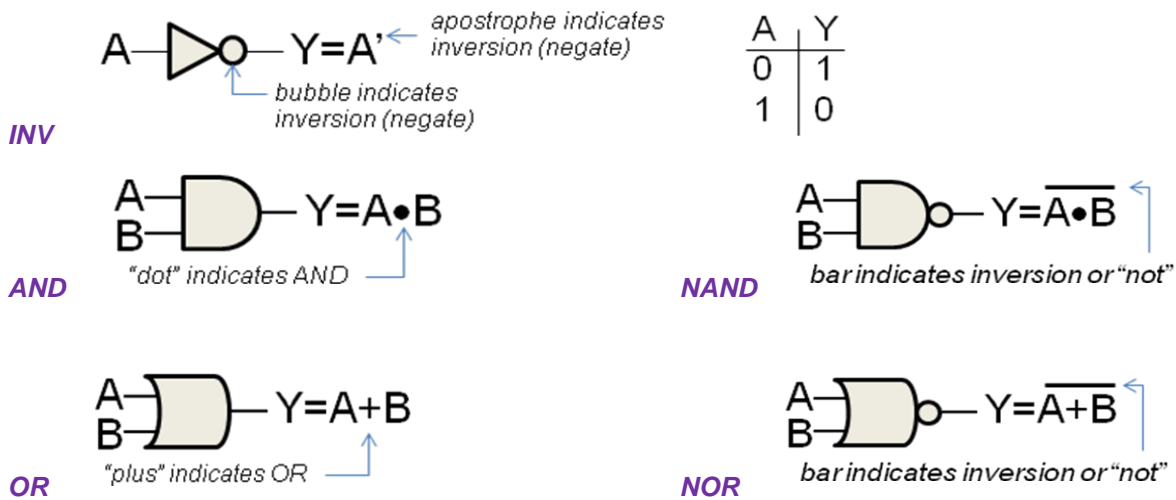


### 3. Digital Logic Review

#### 3.1 Combinational Logic Gates

##### INV, AND, OR, NAND, NOR

The INV (invert), AND, OR, NAND, NOR logic gates form the basic logic functions for all digital circuits. In fact, it is important to note that all digital logic circuits, from flip flops to microprocessors, can be formed from only INV, NAND and NOR gates. Gate symbols and logic truth tables for the INV (invert), AND, OR, NAND, NOR are shown below.



inputs		AND	NAND	OR	NOR
A	B	$A \cdot B$	$\overline{A \cdot B}$	$A + B$	$\overline{A + B}$
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	1	0

*Truth table for AND, NAND, OR, and NOR logic functions.*

#### DeMorgan's Relations

An important relationship between AND, OR, NAND and NOR logic functions are described by the DeMorgan's relations described and illustrated below. These equivalencies can be used to simplify logic expressions using Boolean algebra or to simplify logic circuits by swapping one gate for an equivalent gate.

NAND-OR rule: "An AND with inverted output (NAND) is the same as an OR with inverted inputs"

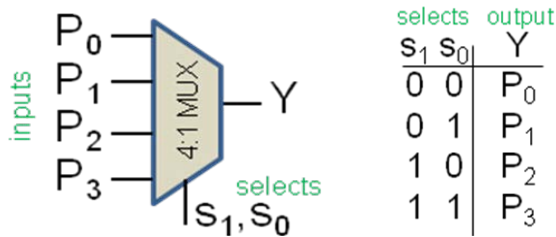
$$\overline{A \cdot B} = \overline{A} + \overline{B} \quad \begin{array}{c} A \\ B \end{array} \text{ AND } \text{bubble} \rightarrow \overline{A \cdot B} = \begin{array}{c} A \\ B \end{array} \text{ OR } \text{bubbles} \rightarrow \overline{A} + \overline{B}$$

NOR-AND rule: "An OR with inverted output (NOR) is the same as an AND with inverted inputs"

$$\overline{A + B} = \overline{A} \cdot \overline{B} \quad \begin{array}{c} A \\ B \end{array} \text{ OR } \text{bubble} \rightarrow \overline{A + B} = \begin{array}{c} A \\ B \end{array} \text{ AND } \text{bubbles} \rightarrow \overline{A} \cdot \overline{B}$$

## MUX

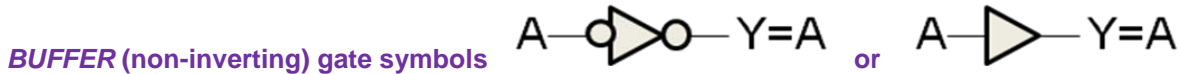
A multiplexer (MUX) routes multiple inputs signals into fewer (or a single) output using digital select lines.  $n$  select lines can multiplex  $2^n$  inputs.



4:1 MUX, 4 (inputs) to 1 (output) MUX: symbol and truth table

## BUFFER

A buffer is used to increase the drive strength of a signal and/or to isolate output signals from the input signal. An inverter is a type of buffer; however we can also use non-inverting buffers.



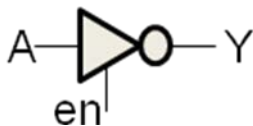
## TRI-STATE BUFFER

Often in digital circuits it is desired to disable a signal so that it is neither 1 (hi) or 0 (low). In this case, the signal is essentially disconnected and said to be in an “open circuit” or “high impedance” (generally called “hi Z” because Z is the symbol for impedance) state. A basic circuit for achieving a “high Z” state is the tri-state buffer, which is an inverter with a third state where the output is disconnected or at high impedance. This third state is controlled by an enable signal where enable = false generally activates the high impedance state.

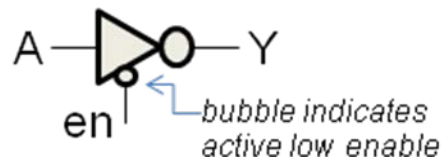


Tri-state buffer gate symbol and truth table.

Other variations of tri-state buffer include the inverting tri-state (basically an inverter with an enable), and active low tri-state buffers and inverters that are high impedance when  $en = 1$ .



Tri-state inverter symbol

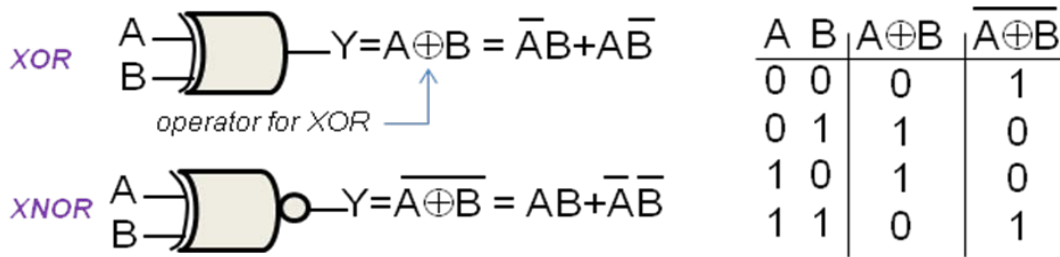


Active low tri-state inverter

## XOR and XNOR

Exclusive OR (XOR) sets the output true (hi, 1) when the inputs are different. That is the output is true if *either* input A or input B, but not both, are true.

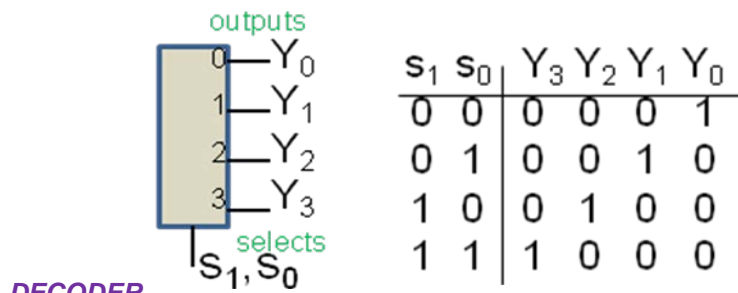
Exclusive NOR (XNOR) sets the output true when the inputs are the same. XNOR is the inverse of XOR.



**XOR and XNOR gate symbols and truth tables.**

**Decoder**

Decoder circuits implement “one hot” code, where only one of many outputs is active-high (true) at a time. Decoders can also be implemented as active low circuits, where only the active output is low and all others are high. Decoders are commonly found in memory circuits to select which of many memory cells is active. The inputs to a decoder are binary encoded select lines as shown below.



**DECODER**

**4-output decoder: gate symbol and truth table.**

**3.2 Sequential Logic Circuits**

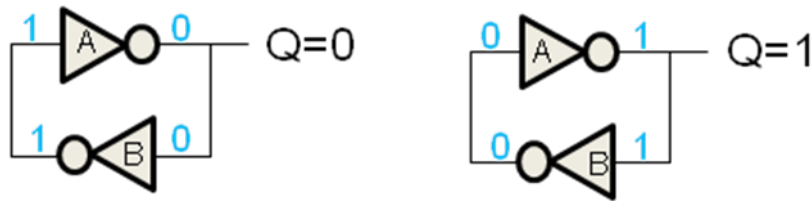
In the combination logic circuits shown above, the outputs are determined entirely by the current inputs. In contrast, there are many useful digital circuits with outputs that are determined (in part) by outputs from prior states (at an earlier time than present). These circuits are generally referred to as sequential logic circuits because their output response can be described by a time sequence pattern.

Sequential logic circuits are said to have “memory” because their outputs are based on states from an earlier time. The ability of sequential circuits to “store” a state for use in the future is critical to the operation of many common digital circuits. Unlike combinational circuits that are described by truth tables, sequential circuit often use state tables to describe what the next state will be for all possible input (and prior state) combinations.

**Latch**

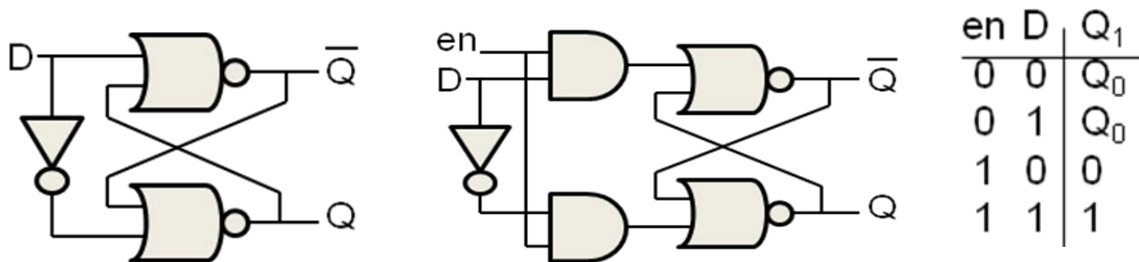
A latch is a circuit that has two stable states and can be used to store state information at one or two outputs. The latch can change states by applying signals to one or more control inputs. Because the state is actively stored by the latch circuit, the output value (state) will be maintained as long as power is applied to the circuit. Latches are often incorporated in static (will not change state over time) memory and are a building block in clock-triggered flip flops.

One of the most basic latch structures is the static digital latch that can be formed by *cross coupling* two inverters in a positive feedback arrangement as shown below. This circuit forces itself to maintain the data value at the output, Q, as follows: If the input to inverter A is a 1, it generates a 0 at its output, which is also the input of inverter B. Inverter B in turn generates an output of 1, which is also the input of inverter A and the circuit achieves a stable state shown in the figure below. Alternatively, if the input to inverter A is a 0, the logic is reversed an a second stable state is achieved. Because this circuit has two stable states, it is often called a bistable circuit.



**Circuit schematic for static digital latch (or bistable circuit). The two possible states are shown.**

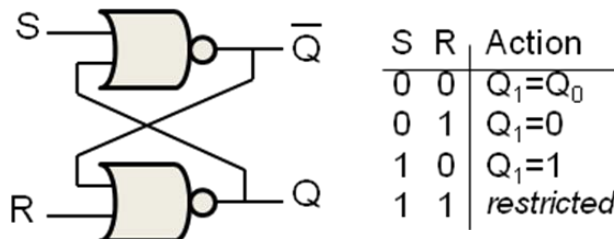
The static digital latch shown above is of limited practical use because it does not provide a means to change the state. To provide access to the latch to set the value it will store, a modified circuit is needed with an input that can set the latch value. The D-latch is a simple circuit that allows the output  $Q$  to be set by input  $D$ . Functionally, the D-latch is simply an inverter, but positive feedback within the latch structure allows the output to hold its value even if the input is removed, something an inverter cannot do. A D-latch can be implemented using multiple configurations of combinational logic gates, and a simple NOR-based D-latch is shown below. To improve the utility of the D-latch, an enable input can be added that permits the latch to either a) set the output  $Q$  based on input  $D$  or b) hold  $Q$  to the internal value using positive feedback, in which case input  $D$  is disabled and will not affect the output. This is generally called a gated D-latch because the enable acts as a gate to let  $D$  pass or not.



**Circuit schematic for a D-latch (left) and gated D-latch (center), and state table for gated D-latch (right).**

The state table for the D-latch with enable is shown above where the next-state output,  $Q_1$ , is defined in terms of the last-state output,  $Q_0$ . When  $en = 0$ , the circuit is in hold mode and the next-state output is held to the last-state value. When  $en = 1$ , the input is enabled and the output is equal to input  $D$ .

Another common latch is the SR latch, which has the storage capability inherent to latches but operates very differently from the D-latch. Here,  $S$  stands for *set* and  $R$  stands for *reset* and the SR latch has three states, a hold state, a set state, and a reset state. When inputs  $S$  and  $R$  are both low, the latch is in hold state and the output remains constant. If  $S$  goes hi (while  $R$  is low) the output is set to 1, and if  $R$  goes hi (while  $S$  is low) the output is reset to 0. The SR latch can be formed using only two NOR gates as shown below. Having similar operation, an  $S'R'$  latch can be formed using only two NAND gates with the set and reset functions becoming active low. The problem with both SR and  $S'R'$  latches is that they have a forth “restricted state” that generates improper outputs, and this state must be avoided when using these latches, limiting their practical value.



**Circuit schematic and operation table for a SR latch.**

Notice that the D-latch has the same feedback structure as the SR latch. It avoids the restricted state of the SR latch by forcing the two inputs to the NOR feedback structure to opposite values, but does not exhibit the set/reset capability of the SR latch.

### But why, Professor?

Why does the SR latch have a restricted state? It won't blow up or anything will it?

If you take a moment to observe how these latches work, you'll find you can figure out their state tables on your own as long as you know how simple INV, AND and OR gates work.

Let's start with the SR latch and begin by assuming  $S$ ,  $R$ , and  $Q$  are 0. Recall that the NOR gate has a low output unless both inputs are low, in which case the output is hi. So,  $R$  and  $Q$  low force  $Q'$  to hi, and with  $S$  low and  $Q'$  hi,  $Q$  is low thus holding its original low value. Now let  $Q$  be hi and start over and you'll find  $Q'$  is low and  $Q$  stay hi, again holding the state. This confirms the action of the  $S=R=0$  state.

Now look at  $R$  hi while  $S$  is low. Because  $R$  is hi,  $Q$  has to be low (NOR output is always low unless both inputs are hi). With  $Q$  and  $S$  low,  $Q'$  is hi. This is the 'reset' state, where  $Q$  is forced to 0.

Similarly, when  $S$  is hi while  $R$  is low,  $Q'$  is forced to low and thus  $Q$  is hi. This is the 'set' state, where  $Q$  is forced to 1.

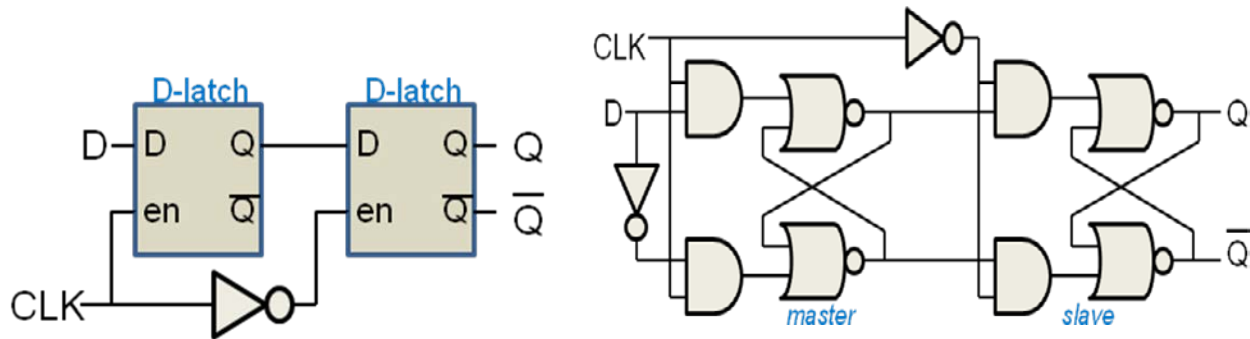
Finally, there is the restricted state. If both  $S$  and  $R$  are hi, the output of both NOR gates must be low, so both  $Q$  and  $Q'$  would be low. Because  $Q'$  is no longer the inverse of  $Q$ , this is not a reliable state. But the circuit is logically stable and nothing should explode!

### Flip Flop

Latches are referred to as level-triggered because their outputs can transition any time the input level (logic value) changes. It is often desirable in digital circuit design to use circuits that only evaluate their outputs during the transition of an input signal. Such edge-triggered circuits often use a clock signal to determine when their output should be evaluated, and the output is set immediately after the edge-triggering event and held at that value until the next edge-trigger event, regardless of changes in any other input levels. A flip flop is an edge-triggered storage circuit that behaves similar to a latch but only changes input in the rising, or falling, edge of a triggering input. When multiple circuits are triggered by the same signal, they become synchronized, i.e. their outputs change at the same time. This synchronizing trigger signal is generally referred to as a clock, which is a periodic square wave that sets the speed of operation in most sequential digital circuits.

Flip flops (FF) are data storage elements for synchronous circuits that have circuit blocks triggered simultaneously and periodically by a clock signal. FFs are used to store the logic state of a signal until the next rising (or falling) edge of the clock signal when the output is reevaluated and stored again. They are typically controlled by one or two input signals and a clock, and they typically have differential (inverted) outputs  $Q$  and  $Q'$ . Some FFs include asynchronous set or reset input signals that immediately force the output hi or low, respectively, regardless of the state of the clock.

To better understand the operation of a FF, consider the master-slave D-type FF composed of two cascaded gated D-type latches. The output of the master block is connected to the input of the slave block, and the enable signal, which we'll now consider to be a clock ( $CLK$ ), is inverted between the master and slave blocks. Each of the latches is individually level-triggered so that when the master is enabled the slave is not (is in hold state), and when the master is not enabled (is in hold state), the slave is enabled. Thus, when  $CLK$  is HI, input  $D$  can pass through the master block but will be held from entering the slave block. Then the  $CLK$  goes LO, and inputs are blocked from the master while the master's output is enabled to pass through the slave. Notice that the output  $Q$  (of the slave latch) only changes as  $CLK$  goes LO; because input  $D$  is blocked at the master stage while  $CLK$  is low, the input to the slave cannot change while  $CLK$  is low and output  $Q$  is thus constant until the next low-going clock edge. Because of the master-slave interactions, the output is only triggered by the low-going, or falling, edge of the clock, and this FF is considered to be *falling edge triggered*. Similarly, by inverting the clock (or moving the  $CLK$  input inverter from the slave to the master stage), a *rising edge triggered* FF can be implemented.



*Block-level (left) and gate-level (right) schematic for D-type master-slave flip flop.*

Just as there are a variety of latches (e.g., D-latch, SR latch), several variations of FFs are available. In a D-type FF, the value of input D is transferred to the output Q at the rising (or falling) clock edge and held there until the next rising (or falling) edge. In contrast, a JK FF implements different functions at the rising (or falling) clock edge based on the values of inputs J and K and then holds the output value until the next rising (or falling) edge. Similar to the SR latch, the JK FF has hold, set, and reset states. However, rather than having a useless (and troublesome) restricted state like the SR latch, the JK FF implements a fourth state that toggles (or inverts) the output. The last FF described here is the T-type FF, or toggle FF. This can be considered as a simplified version of the JK FF that will either a) hold the output or b) toggle (invert) the output. It cannot hold/store any specific input value like the DFF; it can only hold the prior value or invert it, which can be useful in some digital circuits.

D	$Q_1$
0	0
1	1

J	K	$Q_1$
0	0	$Q_0$
0	1	0
1	0	1
1	1	$\bar{Q}_0$

T	$Q_1$
0	$Q_0$
1	$\bar{Q}_0$

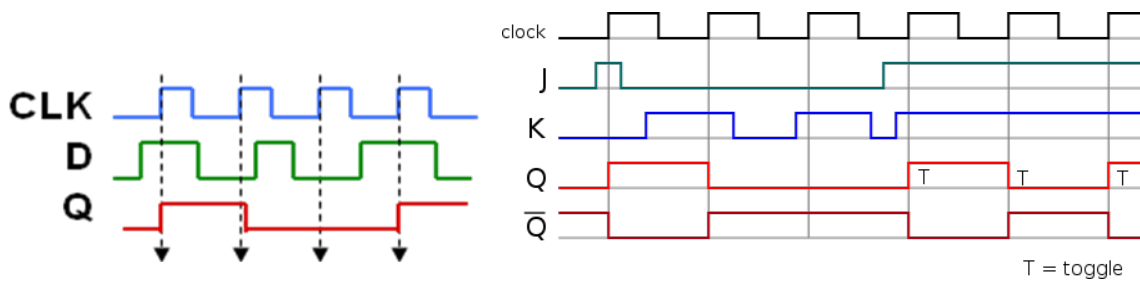
*State tables for D-type, JK, and T-type flip flops.*

In some ways the JK FF can “do more” than the D-type FF; it has 4 operations while the DFF only has one. However, the JK FF cannot directly store/hold an input value like a DFF, which is the most commonly required function of a FF in most digital circuits. For this reason, the DFF is the most common FF used in VLSI circuits.

It should be noted that, by setting K to J', the JK FF can implement the function of a D-type FF. Similarly, by adding an AND and two OR gates to a DFF, the function of a JK FF can be realized. Also, a JK FF (or a DFF converted to JK FF function) can realize the function of a T-type FF by simply connecting the J and K inputs together. Thus, it is possible to implement the functions of D, JK, and T flip flops with either a JK FF or a DFF (and some additional logic).

### **Timing Diagrams**

It is often necessary to interpret timing diagrams that show the state of inputs and outputs over time. For FF circuits and registers (described below), this typically involves observing how the outputs change at the rising or falling edge of a clock. For rising-edge-triggered FFs, the outputs can only change on the high-going edge of the clock signal; the resulting value will then be held until the next rising edge when the outputs should be checked again. Similarly, falling-edge-triggered circuits are checked at the low-going clock edge. To check the outputs, simply follow the state transition rules of the state table for whatever type of FF is being observed. The examples below show timing diagrams for D-type and JK FFs. For ECE331, **you should be capable of finding the output logic levels over time for any given set of FF inputs.**



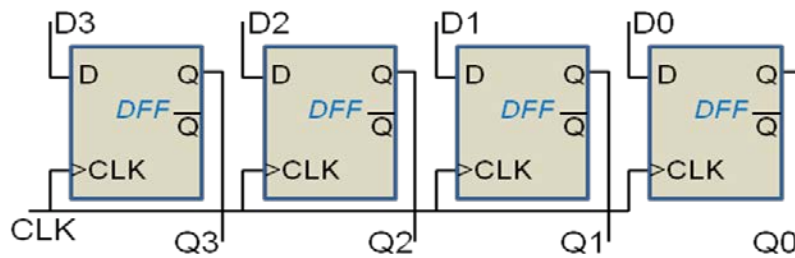
Timing diagrams for rising-edge triggered (left) D-type FF and (right) JK FF. Note that the output only changes when the clock is going high. Any transitions of inputs between clocks should be ignored.

**Factoid!?!**

The rising edge and falling edges of a clock are often referred to as the positive and negative edges. Don't let the terminology confuse you!

**3.3 Data Registers**

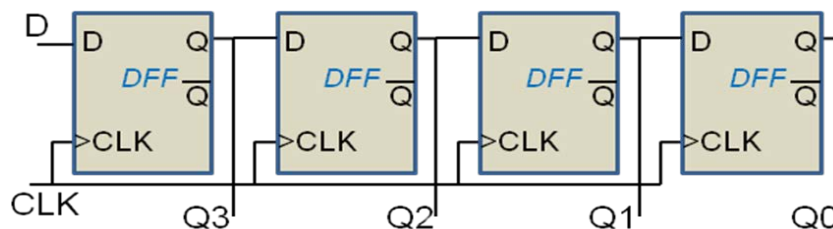
A register is a circuit with a bank of memory elements that is used to store (or manipulate) blocks of binary data. Registers typically have 8, 16 or 32 memory elements to store bytes or words of digital data. Registers are often described by whether their inputs/outputs are written/read parallel (all at the same time) or serially (one bit at a time in sequence). The example below shows a simple 4-bit data register formed from four DFF gates. The inputs D0, D1, D2, D3, often written D(3:1), go into the four FFs in parallel, and the outputs Q(3:1) come out in parallel. The clock signal is shared by all bits of the register so that they all read in new values and write out resulting outputs simultaneously. Whether this data register is positive or negative edge triggered depends on the triggering of the FF blocks.



Block diagram of a 4-bit parallel-in, parallel-out data register formed with DFF gates.

**Shift Register**

Consider what would happen if we modified the 4-bit register above so that the input to each block was taken from the output of the register to its left, as shown in the diagram below. Bit 2 (Q2) would now get its input from the output Q3, and bit 1 (Q1) would come from Q2, etc. Thus, data entering at D (bit 3) would be shifted through the register from left to right. This would be described as a serial-in, parallel out data register. Because data is shifted from left to right, it could also be called a shift-right register. With some additional logic, the register could be modified to be capable of shifting data both left and right.



Block diagram of a 4-bit serial-in, parallel-out shift register that can only shift right.

### Thought Exercise

In the 4-bit serial-in shift register shown above, to load in a 4-bit word, which data bit should be input first, D0 or D3?

### Shift and Rotate Functions

Common functions implemented within the CPU of a microprocessor are the ability to shift and rotate data to the left and to the right. In the shift and rotate functions, all bit values are moved by one (or more) places to the left or right. However, this leaves a void where the first bit was moved from. For example, assume the shift register above is loaded with 4-bits of data and disconnected from input D. What happens then when the data is shifted to the right? Bit 3 goes to 2, bit 2 goes to 1, and bit 1 goes to 0, but what happens to bit 0 and what comes into bit 3? In a shift function, the bits shift as just described, bits shifted out the front end (bit 0 in this example) are lost (ignored), and a preset value (normally 0) is loaded into the back end (the new bit 3 in this example). In contrast, for a rotate function, the bits going out the front end are rotated back into the opposite end, so no bits are lost they are just cycled around. In most CPUs, both shift and rotate functions can execute multiple-bit cycling at once. For example, shift left by 2, or rotate right by 4.



Illustrations of shift right and rotate left functions.

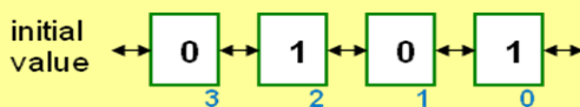
- **Shift** move each bit (left or right) to adjacent register, load in preset value (normally 0) into open registers
- **Rotate** move each bit (left or right) to adjacent register, rotate exiting bits back into other side of register

It is worth noting that shifting a data byte left by 1 is equivalent to multiplying by 2 (and shifting by 2 is like multiplying by 4, etc). Similarly, shifting a data byte right by 1 is equivalent to dividing by 2.

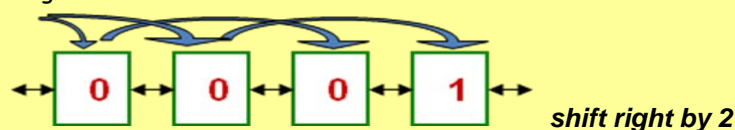
#### EXAMPLE: Shift and Rotate functions

Assume you have a 4-bit shift register loaded with the initial values below. What values would be in each bit of the register after

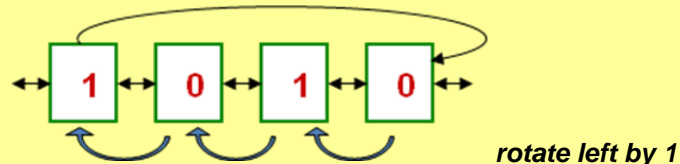
a) shift right by 2, b) rotate left by 1



a) For shift right by 2, bits would be moved to the right by 2 places and 0 values (assumed) would be loaded into any registered emptied by the shift. Thus, position 3 would move to 1, 2 would move to 0, and 1 and 0 would be lost. Positions 3 and 2 would get a new 0 value.



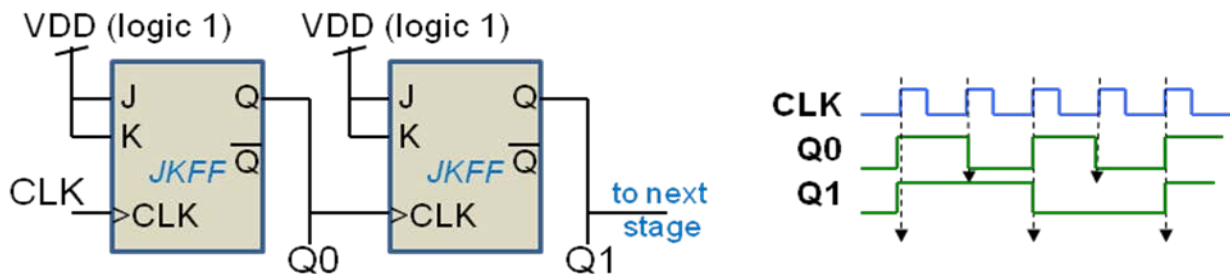
b) For rotate left by 1, bits would be moved to the left by 1 place and the left-most bit (3) would be rotated into the bit 0. Thus, bit 0 goes to 1, bit 1 goes to 2, bit 2 goes to 3, and bit 3 goes to position 0.



## Counters

A counter is a special type of register that counts the number of times an input signal changes value, from 1 to 0 or 0 to 1 depending on the polarity of the counter. Counters are commonly employed to count clock cycles in timer hardware. For a periodic input like a clock, each successive bit of the counter will toggle (change state) at  $\frac{1}{2}$  the frequency of its input bit. Thus, counters can also be used to divide higher frequency clocks down to lower frequencies.

There are many ways to implement counters using various circuit configurations with flip flops. In a binary ripple counter, a T-type (toggle) FF is used as the first stage and set to toggle mode. Because the output will toggle value only when the input clock shows a rising (or falling) edge, the output would be  $\frac{1}{2}$  the frequency of the input clock. That output is then used to clock another T-type FF so that its output changes at  $\frac{1}{2}$  its input frequency, which would be  $\frac{1}{4}$  the frequency of the main clock. The structure continues through as many states as needed to count the desired duration or generate the desired frequency. In a synchronous counter, all FFs are clocked at the same time which allows more precise timing. Logic gates are included between bits to implement the desired counting function. In a counter-by-n counter,  $m$  FFs are used to count from 0 to  $2^m-1$  before the counter resets to zero and starts over.



*A 2-bit binary ripple counter and associated timing diagram showing frequency decreasing.*

### EXAMPLE: Counters

How many counter bits (flip flops) would be needed to count for 1ms if the input clock is 1MHz?

A 1MHz clock changes state every 1 $\mu$ s, so 1000 clocks would be needed to count to 1ms. Since  $2^{10}$  is 1024, anything less than 10 bits would not be able to count to 1ms. Thus, **10 counters bits are needed.**

## 3.4 Complete Set Concept

As noted at the beginning of section 3, all digital logic functions can be implemented with only INV, NAND, and NOR gates, and thus these gate are the primary tools in any VLSI designers toolbox. It is a cornerstone of VLSI design that all complex digital circuits are composed of more simple base gates. In practice, to optimize speed and power, many “building block” logic circuits like flip flops and adders are designed at the transistor level rather than at the gate-level, but it is possible to construct all logic functions from simple gates like NANDs and NORs. In fact, the complete set concept states that all logic functions can be formed using only NAND gates or only NOR gates. An entire microprocessor can be built using only NAND (or only NOR) gates! As proof of this concept, note that registers are composed of flip flops and, as shown above, a D-type flip flops can be constructed from only INV, AND, and NOR gates. Thus, if one could show that INV, AND, and NOR could be implemented using only NAND gates it would prove that data registers could be implemented using only NAND gates. This proof will be explored in homework (aren't you excited!). But to get you started, here's an illustration showing how a NAND gate can implement the INV function.

$$A \text{---} \text{INV} \text{---} \bar{A} = A \text{---} \text{NAND} \text{---} \bar{A} \cdot A = \bar{A}$$

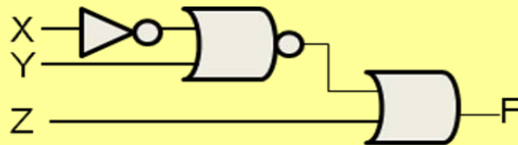
## Bubble Pushing Technique

The circle symbols at the end of an INV or NAND gate are referred to as “bubbles”. They represent an inversion (negation) operation. And just as the mathematical product of two negatives is a positive, two bubbles can be combined to remove them from a schematic. It is just like  $A = (A')' = A$ . Similarly, two bubbles can be added to a single node in a schematic without changing it functionally. These options may seem useless, but they can be used to simplify a digital logic schematic. Recall that DeMorgan’s relations show how a gate with an inverted output can be replaced by a gate with inverted inputs. Combining DeMorgan’s equivalent circuits with bubble pushing techniques can often allow you to manipulate logic at the schematic level the same way you might use Boolean algebra principles to manipulate a logic expression. The following example helps to illustrate this concept.

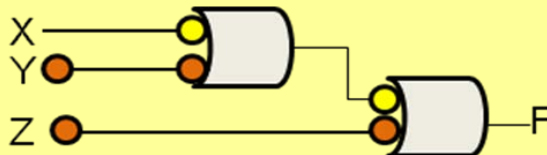
### EXAMPLE: Complete Set Concept

This example illustrates the use of DeMorgan’s relations and the bubble pushing technique to modify the gates used in a circuit so that it follows the complete set concept. The result is not necessarily the optimal circuit with the best performance, but it exercises techniques useful in the design of digital circuits.

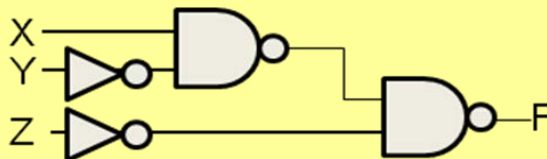
**Manipulate the logic circuit below using DeMorgan’s relations and bubble pushing to realize the same logic function using only NAND gates.**



Step 1: By DeMorgan’s, NAND is equivalent to OR with inverted inputs, so let’s first try to get this circuit into only NAND and OR gates. We can slide the bubble on the NOR up to the input of the OR and replace the INV with a bubble at the input of the NOR. Then we can add bubble pairs to node Y and Z. The result looks like this, with yellow bubbles that were moved and orange that were added.



Step 2: The only logic gates remaining are ORs with inverted inputs, so by DeMorgan’s we can replace them in NANDs. That just leaves the bubbles at the Y and Z input, which we’ll replace for now with INV gates.



Step 3: As shown in the Complete Set Concept section above, INV can be replaced by NAND with the inputs tied together. This gives the **final circuit using only NAND gates**.

