

4. Digital Logic

At the hardware level, a computer consists of a very large number of interconnected logic elements. Groups of logic elements implement various logic functions, such as the addition of binary numbers. Most of these functions can be precisely and completely described with Boolean Algebra or as a Finite State Machine (FSM).

4.1 Boolean Algebra and Truth Tables

A *Boolean function* is a function of one or more *Boolean variables*, for example, $Y = f(X_{N-1}, \dots, X_1, X_0)$. Each input variable X_i and the output Y can only have one of two distinct values, for example, TRUE or FALSE, ON or OFF, and 1 or 0. Actually Boolean functions can have more than one output. A Boolean function can be defined by a *truth table*, which lists every distinct combination of values for the input variables along with the output values corresponding to each input combination. In that table, there is one column for each input and output variable and one row for each combination of input values. A function of N variables has N inputs, thus the number of distinct input combinations (nc) is 2^N and there will be nc rows in the truth table. There are 2^{nc} different functions of N variables since there are nc rows in the truth table and thus 2^{nc} distinct combinations of output values for the nc rows. For example, when $N=1$ there will be $2^1 (=2)$ distinct input combinations and $2^2 (=4)$ distinct functions.

X	Z ₀	Z ₁	Z ₂	Z ₃
0	0	0	1	1
1	0	1	0	1

This table can be interpreted as defining one function that has one input variable and four output variables. It can also be interpreted as defining four separate functions, each with one input variable and one output variable. Using the second interpretation, we see, for example, that Z_2 is the function NOT X . When $N=2$ there will be $2^2 (=4)$ distinct input combinations and $2^4 (=16)$ distinct functions.

X ₁	X ₀	V ₀	V ₁	V ₂	V ₃	V ₄	V ₅	V ₆	V ₇	V ₈	V ₉	V ₁₀	V ₁₁	V ₁₂	V ₁₃	V ₁₄	V ₁₅
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

This table defines sixteen functions, each with two input variables and one output variable. Here, for example, V_7 is the function X_1 IOR X_0 and V_1 is the function X_1 AND X_0 . The functions defined by the above two tables are the basic logic functions from which essentially all the logic components of a computer are constructed.

It is common practice to write the right-hand side of a Boolean function as a Boolean expression. A *Boolean expression* consists of Boolean variables connected with Boolean operators, with each operator corresponding to one of the basic functions defined in the above two tables. Not all of these functions will be used in this course. The following are the operators that we will use and the functions they represent.

Operator	Name	Function
X	Identity	$Y = X$ — column Z_1 in the first table above
$\sim X$	Complement	$Y = \text{NOT } X$ — column Z_2 in the first table above
$X_1 \mid X_0$	Inclusive or	$Y = X_1 \text{ IOR } X_0$ — column V_7 in the second table above
$X_1 \oplus X_0$	Exclusive or	$Y = X_1 \text{ XOR } X_0$ — column V_6 in the second table above
$X_1 \bullet X_0$	And	$Y = X_1 \text{ AND } X_0$ — column V_1 in the second table above
$X_1 \mid \mid X_0$	Nor	$Y = X_1 \text{ NOR } X_0 = \text{NOT}(X_1 \text{ IOR } X_0)$ — column V_8 in the second table above
$X_1 \mid \bullet X_0$	Nand	$Y = X_1 \text{ NAND } X_0 = \text{NOT}(X_1 \text{ AND } X_0)$ — column V_{14} in the second table above

In the table, two of the symbols are different than the symbols used in computer architecture books, which use a bar over the variable or the suffix ' for NOT, + for IOR, \oplus for XOR, and \bullet or juxtaposition for AND. Some books even have special symbols for NOR and NAND. To avoid confusion with the use of + for addition, we use \mid for IOR, which is the Java/C/C++ inclusive or operator. Because of the limitations of word processors like Word 97, which make it difficult to get bars over expressions, we use \sim for NOT, which is the Java/C/C++ bit-wise complement operator. We will use juxtaposition for AND when it is clear which elements are the operands.

Boolean algebra uses *algebraic identities*, such as the following to prove the truth or falsity of a Boolean equation. Here all variables have single letter names so using juxtaposition for AND should not be confusing.

I1:	$A = \sim\sim A$
I2:	$A A = A$
I4:	$A 0 = A$
I6:	$A 1 = 1$
I8:	$A \sim A = 1$
I10:	$A B = B A$
I12:	$A (B C) = (A B) C$
I14:	$A (B C) = AB AC$
I15:	$(B C) A = BA CA$
I18:	$A \sim B B = A B$
I19:	$\sim AB A \sim B = A \oplus B$

I3:	$A \bullet A = A$
I5:	$A \bullet 0 = 0$
I7:	$A \bullet 1 = A$
I9:	$A \sim A = 0$
I11:	$AB = BA$
I13:	$A (BC) = (AB) C$
I16:	$\sim (A B) = \sim A \sim B$
I17:	$\sim (AB) = \sim A \sim B$
I20:	$\sim A \sim B AB = \sim (A \oplus B)$

The precedence of these operators is the same as the precedence of the Boolean operators in Java/C/C++. Some of these identities have names – I10 and I11 are *commutative laws*, I12 and I13 are *associative laws*, I14 and I15 are *distributive laws*, and I16 and I17 are *De Morgan's laws*. All these identities can be verified using truth tables. For example, the following truth table verifies the I16 part of de Morgan's laws.

A	B	A B	$\sim (A B)$	$\sim A$	$\sim B$	$\sim A \sim B$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

We use these identities to prove the truth or falsity of Boolean statements. For example, we claim that $\sim AB \sim C | \sim ABC | ABC = \sim AB | B \sim C$ is true. Using the above identities we can derive the right side from the left side.

$\sim AB \sim C | \sim ABC | AB \sim C$
 $\sim ABC | \sim AB \sim C | AB \sim C$ reorder terms using I10
 $\sim ABC | (\sim A | A) B \sim C$ factor using I15
 $\sim ABC | 1 \bullet B \sim C$ reduce using I8
 $\sim ABC | B \sim C$ reduce using I7
 $(\sim AC | \sim C) B$ factor using I15
 $(\sim A \sim \sim C | \sim C) B$ expand using I1
 $(\sim A | \sim C) B$ reduce using I8
 $\sim AB | B \sim C$ distribute using I15

We could also have proved this using a truth table as we did with De Morgan's law above. In fact, truth tables can be used to prove or disprove the equality of any two Boolean expressions.


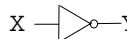
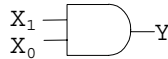

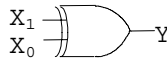
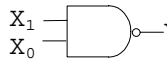
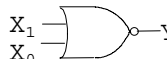
A truth table for $Y = f(X_{N-1}, \dots, X_1, X_0)$ can be transformed into a *sum-of-products* (SOP) Boolean expression. First form a *product* (AND) for each row in which $Y = 1$ and then form the *sum* (IOR) of all these products. The product for a row is $X_1 \bullet \dots \bullet \sim X_j \bullet \dots$ for all X_i in the row that equal 1 and for all X_j in the row that equal 0. For example

X_1	X_0	Y
0	0	1
0	1	0
1	0	1
1	1	1

SOP for this truth table is $Y = \sim X_1 \sim X_0 | X_1 \sim X_0 | X_1 X_0$

4.2 Logic Gates

The basic logic elements used to build a computer are *logic gates*. A logic gate is an electronic circuit, composed of one or more transistors, that implements a simple Boolean function.

- a) buffer $Y=X$ 
- b) NOT gate $Y=\sim X$ 
- c) AND gate $Y=X_1 X_0$ 
- d) IOR gate $Y=X_1 | X_0$ 
- e) XOR gate $Y=X_1 \oplus X_0$ 
- f) NAND gate $Y=X_1 ! \bullet X_0$ 
- g) NOR gate $Y=X_1 ! | X_0$ 

The first gate shown is the identify function, that is, its output is equal to its input. All it does is to introduce a time delay (we will see later that there is a time delay in every gate, which is proportional to the number of transistors in the longest path through the gate). The little bubble in the output of some of the gates is called *an*

inversion and is shorthand for NOT. Logically, a NAND gate is NOT applied to an AND gate and a NOR gate is NOT applied to an IOR gate. However, we will soon learn that NAND and NOR are not implemented this way.

A single transistor connected as shown in (a) below, functions as an *inverter* (NOT gate). The transistor acts as a switch, which is closed when V_A is high (normally +5 volts or less) and open when V_A is low (0 volts). When the switch is closed, the supply voltage V_{supply} is effectively short circuited to the ground so V_X will equal zero. When the switch is open, V_X will equal V_{supply} . Let V_0 (0 volts) be the voltage that represents logic 0 and V_1 (+5 volts) be the voltage that represents logic 1. If $V_A = V_0$ then $V_X = V_1$ and if $V_A = V_1$ then $V_X = V_0$, that is, $V_X = \sim V_A$. The range $V_0 \dots V_1$ is divided into three sections. Voltages in the range $V_0 \dots V_{0,max}$ will represent logic 0; voltages in the range $V_{1,min} \dots V_1$ will represent logic 1; and voltages in the range $V_{0,max}$ to $V_{1,min}$, called the *forbidden range*, will have an indeterminate interpretation due to noise and other effects, so logic gates are not normally operated in this range. We define the *switching time* of a gate as the time between when the input has changed, from 0 to 1 or 1 to 0, and when the output has settled into a steady state in response to the input change. In the NOT gate, the switching time is essentially equal to the time it takes for the voltage to change from V_0 to V_1 or V_1 to V_0 . This time, which we call Δt , is the shortest delay for any gate.

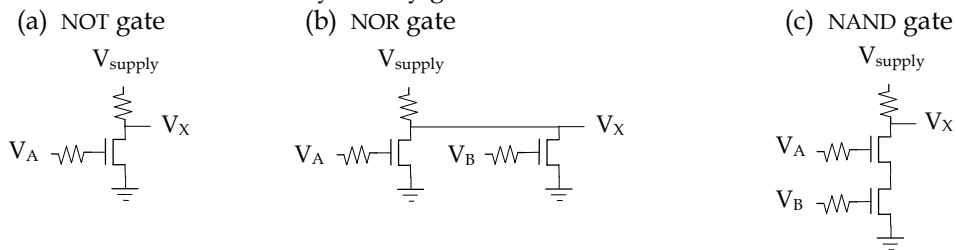
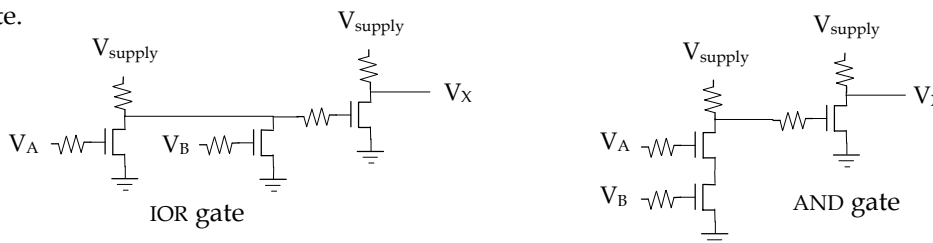


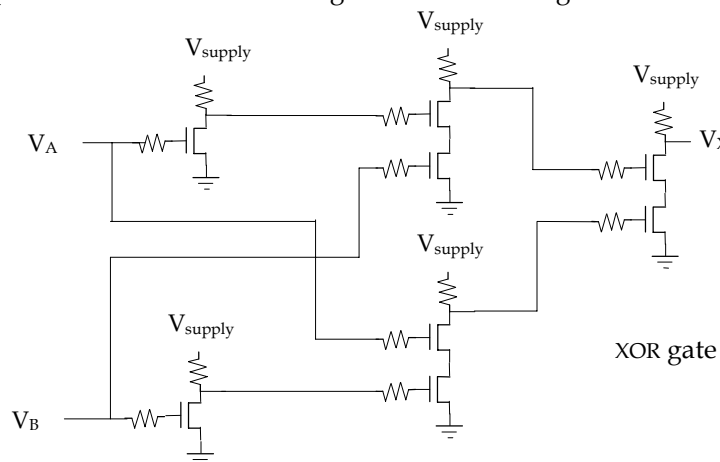
Diagram (b) above shows how two transistors can be connected to make a NOR gate, $V_X = V_A \mid \mid V_B$, and (c) shows how two transistors can be connected to make a NAND gate, $V_X = V_A \bullet \bullet V_B$. An IOR gate is constructed by putting a NOT gate in the output of a NOR gate and an AND gate is constructed by putting a NOT gate in the output of a NAND gate.



An XOR gate can easily be constructed from NAND and NOT gates. Using Boolean algebra

$$A \oplus B = \sim AB \mid A \sim B = \sim \sim (\sim AB \mid A \sim B) = \sim (\sim (\sim AB) \bullet \sim (A \sim B)) = \sim (\sim AB) \bullet \sim (A \sim B) = (\sim A \bullet B) \bullet (A \bullet \sim B)$$

So an XOR gate can be implemented with three NAND gates and two NOT gates.



The above implementations of logic gates are members of the RTL (resistor-transistor logic) family. These are the simplest gate implementations, but they are no longer used because of their slow speed and high power consumption. However, because of their simplicity, they are useful for illustrating the basics of logic gate implementation. The TTL (transistor-transistor logic) family was popular until about 1990. The MOS (metal-oxide semiconductor) family evolved into the CMOS (complementary metal-oxide semiconductor) family, which is extensively used in VLSI. CMOS uses pairs of complementary transistors, called *p-channel* and *n-channel*

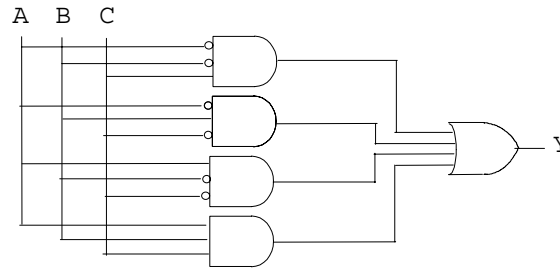
transistors. The CMOS family has high noise immunity, low power consumption, and very small size. Small size is a big advantage in VLSI. The ECL (emitter-coupled logic) family has the highest speed (i.e., shortest switching time) of all the families, but is the most expensive.

Current trends in chip design are to include more transistors, use lower voltages, use higher clock frequencies, and use transistors with shorter switching times. Power is proportional to the square of the voltage, so the decrease in voltage from 5.0v to 3.3v, and even lower, would tend to decrease the power consumption. However, this decrease in power has been overwhelmed by the increase in the number of transistors (42 million in the Pentium 4) and the use of higher clock frequencies (more than 1 GHz). Moore's Law, which states that the number of transistors doubles every two years, has held over the past few decades. If it continues to hold, by the year 2010 chips will contain over a billion transistors and clock frequencies should be as high as 30 GHz. If this happens, the power consumption will increase to the point where the internal temperature will be greater than the temperature inside a nuclear reactor!

4.3 Combinational Circuits

A *combinational circuit* is composed entirely of logic gates and has the property that it has no memory, that is, the output produced by a particular set of input values will always be the same, no matter what has happened in the past. A *logic gate diagram* consists of only the symbols for basic logic gates and lines that connect some gates to other gates. To transform a Boolean expression into a logic gate diagram, we replace each operator in the expression with the corresponding logic gate. Product terms composed of more than two variables map into multiple input AND gates, sum terms composed of more than two variables map into multiple input IOR gates, and complement terms map into NOT gates or inversion bubbles at the inputs of other gates. For example

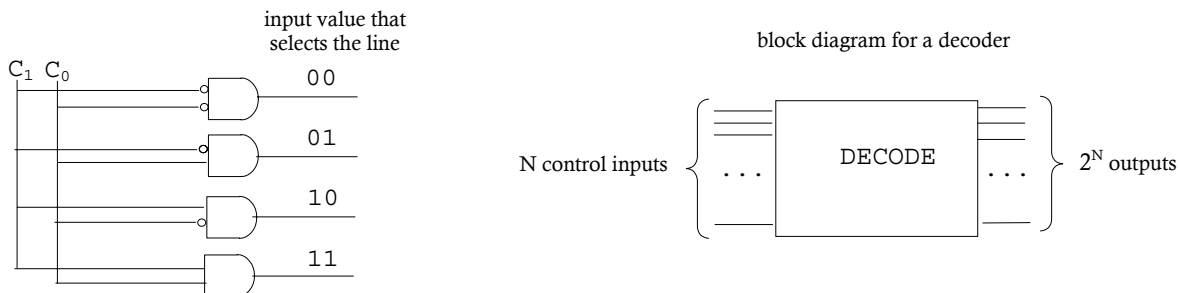
$$Y = \sim A \sim B C \mid \sim A B \sim C \mid A \sim B \sim C \mid A B C$$



When two lines completely cross each other there is no connection unless there is a solid dot on the connection. When one line branches from another, both branches are connected to the main line. When the direction of current flow is not clear, there will be an arrow head indicating the direction. Here no arrow heads are needed because current can only flow into the input side of a logic gate.

4.3.1 Decoders

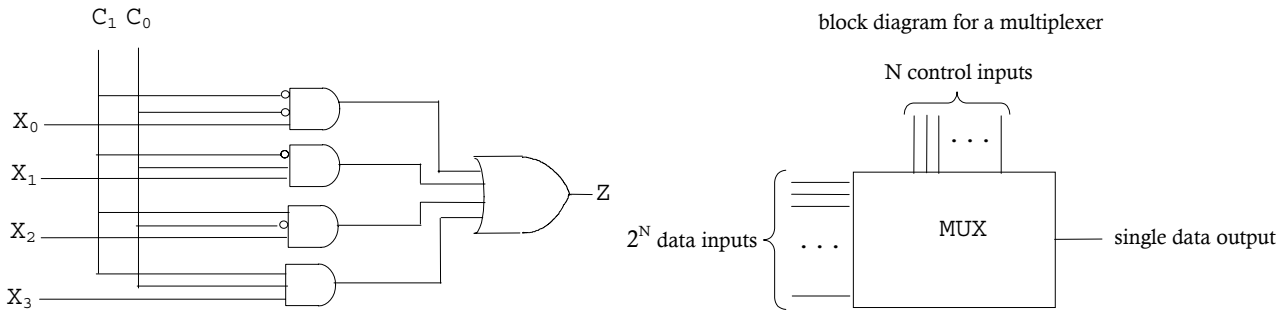
A *decoder* has N control input lines and 2^N output lines. Each combination of control inputs causes exactly one of the output lines to be 1, with all the other output lines 0.



The mapping is unique, that is, every different set of inputs selects a different output line. In the 2 to 4 decoder shown above, the input is a 2-bit number C_1C_0 . Each different value of C_1C_0 selects a different output line, which is set to 1 and all other output lines are set to 0. In the diagram, the value of C_1C_0 that selects each output line is written above that line. On the right is the *block diagram* for an N to 2^N decoder. A decoder can be used to select a particular register in a register file. Another use for a decoder is to select a particular operation in the ALU. A diagram containing block diagrams, and possibly some basic logic gates, is a *logic diagram*, while a logic gate diagram contains only basic logic gates.

4.3.2 Multiplexers

A *multiplexer* has N control input lines, 2^N data input lines, and one data output line. Each distinct set of control inputs selects one data input line and passes its value, unchanged, to the output line.



For example, in the diagram above, if $C_1C_0 = 10$ then the Z output will equal the X_2 input. On the right is the block diagram for a 2^N data input multiplexer. One use for a multiplexer is to select the input to a component, such as the PC, from several different sources – for the PC, either from an incrementing unit when the next instruction to be executed is the instruction immediately following the current instruction or from the current instruction itself when the instruction is a branch instruction.

4.4 Addition and Subtraction Units

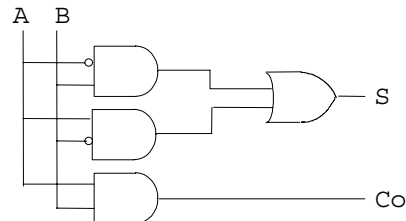
An add-subtract unit is composed of a number of simple components, one for each bit position. These components are called full-adders. So an add-subtract unit that can add or subtract 32-bit naturals and integers (recall that a single algorithm works for both naturals and integers) consists of 32 full-adders. It is easier to understand the design of a full-adder if we first look at a half-adder.

4.4.1 Half-Adders

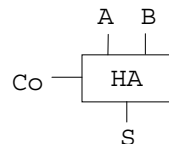
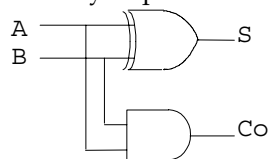
A *half-adder* (HA) has two 1-bit inputs, A and B , and two 1-bit outputs, S and C_o . The S output is the sum of $A+B$ modulo 2 and C_o is the carry-out from the sum of $A+B$. Both S and C_o are defined by the following truth table. The logic gate diagram on the right is synthesized directly from the corresponding SOP expressions for S and C_o .

A	B	S	C_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S = \sim AB \mid A\sim B$
 $C_o = AB$



Note that $A\oplus B = \sim AB + A\sim B$ so the following logic gate diagram is equivalent to the one above. Although this equivalent diagram looks simpler, we will see later that it is actually not. The reason for this has to do with the way the various gates are actually implemented with electronic components.



The diagram on the right is the block diagram for a HA. It represents the function of a HA without indicating any particular implementation. It is used as a component in a larger logic diagram when the HA function is needed but a particular implementation of that function is not important.

4.4.2 Full-Adders

A *full-adder* (FA) has three 1-bit inputs A , B , and C_i , and two 1-bit outputs, S and C_o . C_i is a carry-in bit, usually the C_o output from some other adder and S is the sum of $A+B+C_i$ modulo 2. C_o is the carry-out from the sum of $A+B+C_i$. Both S and C_o are defined by the following truth table.

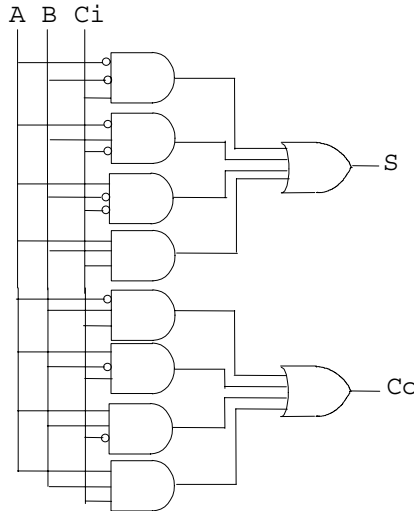
A	B	C _i	S	C _o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The SOPs for the sum output S and the carry-out output C_o are

$$S = \sim A \sim B C_i \mid \sim A B \sim C_i \mid A \sim B \sim C_i \mid A B C_i$$

$$C_o = \sim A B C_i \mid A \sim B C_i \mid A B \sim C_i \mid A B C_i$$

The following gate diagram was synthesized from these two SOPs. We call this the SOP FA.



Using Boolean algebra we can derive simpler expressions for S and C_o and thus construct a simpler logic gate diagram for an FA, which we call the XOR FA.

$$S = \sim A \sim B C_i \mid \sim A B \sim C_i \mid A \sim B \sim C_i \mid A B C_i$$

$$S = \sim A B \sim C_i \mid A \sim B \sim C_i \mid \sim A \sim B C_i \mid A B C_i$$

$$S = (\sim A B \mid A \sim B) \sim C_i \mid \sim A \sim B C_i \mid A B C_i$$

$$S = (A \oplus B) \sim C_i \mid (\sim A \sim B \mid A B) C_i$$

$$S = (A \oplus B) \sim C_i \mid \sim (A \oplus B) C_i$$

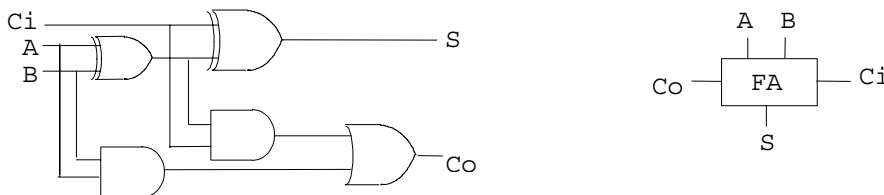
$$S = (A \oplus B) \oplus C_i$$

$$C_o = \sim A B C_i \mid A \sim B C_i \mid A B \sim C_i \mid A B C_i$$

$$C_o = (\sim A B \mid A \sim B) C_i \mid A B (\sim C_i \mid C_i)$$

$$C_o = (A \oplus B) C_i \mid A B$$

The logic gate diagram for an XOR FA is synthesized from the simpler expressions for S and C_o derived above. However, this circuit is not simpler if we count the number of transistors inside the gates and the manner in which they are connected, which we will do in Section 4.5.

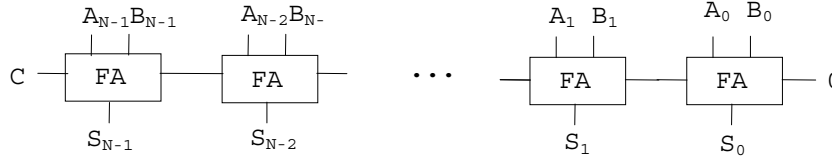


The diagram on the right is the block diagram for a FA that is used as a component in a larger logic diagram when the FA function is needed but a particular implementation of that function is not important.

4.4.3 Ripple-Carry Adders

An N-bit *ripple-carry adder* adds two N-bit numbers. It consists of a sequence of N FAs with the C_o output of each FA, except the last, connected to the C_i input of the next FA in the sequence. There is one FA for each bit position in the N-bit numbers with the corresponding bits of the two numbers going to the A and B inputs of the FAs. For

example, let $A = A_{N-1} \dots A_1 A_0$ and $B = B_{N-1} \dots B_1 B_0$ be N -bit numbers, $S = S_{N-1} \dots S_1 S_0$ be the sum $(A+B) \bmod 2^N$, and C be the carry-out from the high bit of the sum. The logic diagram for an N -bit ripple-carry adder is



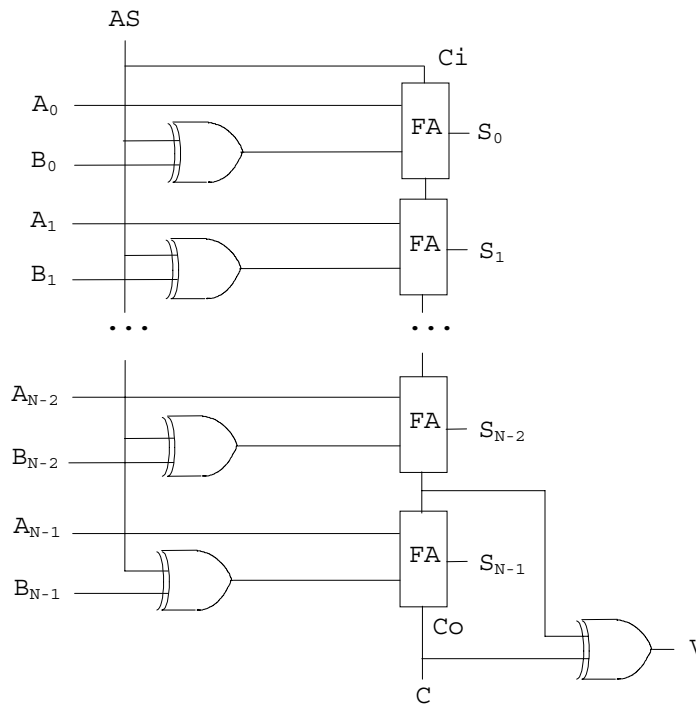
The C_i input of the low bit is zero since there is never any carry into that bit position. The delay in an FA is the delay along the longest path, which runs from the C_i input of the FA for the lo bit position to the C_o output of the FA for the hi bit position. The delay for an N -bit adder is N times the delay for the C_o output of a single FA. A more complex design, called a *carry-lookahead adder*, has a much shorter delay than a pure ripple-carry adder, but at a cost of significantly more gates.

4.4.4 Add-Subtract Unit for Naturals and Integers

The logic diagram for an add-subtract unit is on the next page. Like a ripple-carry adder, the add-subtract unit for an N -bit number contains N FAs. The subtraction $S = A - B$ is done by taking the one's complement of B and then adding that result and 1 to A . The one's complement is done by N XOR gates. For example, let

$$A = A_{N-1} \dots A_1 A_0 \quad B = B_{N-1} \dots B_1 B_0 \quad S = S_{N-1} \dots S_1 S_0$$

be N -bit numbers. A and B can both be naturals or both be integers, but they cannot be mixed. Let AS be the add-subtract control. When $AS = 0$, $S = (A+B) \bmod 2^N$ and when $AS = 1$, $S = (A-B) \bmod 2^N$. When AS is 0, one input to each XOR gate is 0 and, since $(0 \oplus B_k)$ equals B_k , the input to the full-adders is just B . But when $AS = 1$, one input to each XOR gate is 1 and $(1 \oplus B_k)$ equals the one's complement of B_k , so the input to the full-adders is the one's complement of B . The addition of the extra 1 needed to make the two's complement is accomplished by sending AS to the C_i input of the full-adder for the lo bit. The V output goes to the V flag in the program status register (PSR) and the C output goes to the C flag in the PSR. Recall that the V flag is set to 1 when there is a carry into the hi bit position or a carry out of the hi bit position, but not both. An XOR gate generates the proper V flag value. The C flag is set to 1 when there is a carry out of the hi bit position. In a computer like the Pentium that sets the C flag to 1 if the actual result is equal to the true result, the value of the C flag would be the output of an XOR gate whose inputs are C_o and AS .



4.5 Minimization of Logic Gate Circuits

The objective of *minimizing* the implementation of a logic function is to reduce the cost according to some criterion. One such criteria is the total number of gates. Using this definition, the cost of the SOP version of an FA is 19 — 8 AND gates, 2 IOR gates, and 9 NOT bubbles (which are NOT gates) and cost of the XOR version is 5 — 2 XOR gates, 2 AND gates, and 1 IOR gate (a decrease of approximately 74%). This is deceptive because it does not

reflect the number of transistors or the time for a change in the input signal to cause a stable output signal. We need to take into account how the individual gates are constructed from transistors in order to get a better handle on the cost. Let us redefine the cost as the total number of transistors. Using this definition, the cost is 1 for a NOT gate, 2 for a NOR or NAND gate, 3 for an IOR or AND gate, and 8 for an XOR gate. With this definition the cost of the SOP version of an FA is 39 and the cost of the XOR version is 25 (a decrease of approximately 36%).

However, the number of transistors is not the only consideration. The delay from the input of a circuit to its output determines the time it takes to perform its function. The delay for a circuit is the maximum of the delays along each of the distinct paths through the circuit, that is, the path through the transistors in the circuit that has the longest delay. This delay is equal to the switching time for one transistor multiplied by the number of transistors in the longest path. The delay for a NOT gate is Δt since there is only one transistor. The delay for a NOR or NAND gate is also Δt , even though there are two transistors since they both switch in parallel. The delay for an IOR or AND gate is $2\Delta t$ since they each have an additional transistor in their output. The delay for an XOR gate is $3\Delta t$, $1\Delta t$ the NOT gate in front of each of the two parallel NAND gates and $2\Delta t$ for the following sequence of two NAND gates. Computing the delay for both versions of the FA, we see that the delay for the SOP version is 5 (a NOT input to an AND followed by an IOR) for both the C_0 and S outputs; and the delay for the XOR version is 6 for the S output and 7 for the C_0 output. Thus, while the SOP version has a higher component cost, it has a lower delay cost.

In practice, extensive use is made of NAND and NOR gates because of their simple electronic implementation. However, logic design with them is less straightforward than with AND and IOR gates. One major difficulty is that NAND and NOR are not associative (AND and IOR are both associative). This makes it more difficult to cascade similar gates. Cascading is required in some situations because of the limitation on the number of separate lines that can be inputs to a single gate.

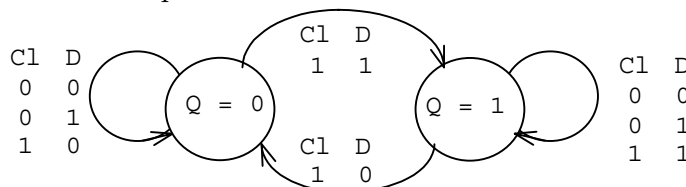
In addition to component and delay costs, other implementation aspects need to be considered. Minimizing the number of places a wire crosses over another wire may significantly simplify the chip fabrication process and reduce the cost of manufacturing the chip. There are also physical limits such as power consumption, fan-in, and fan-out. The higher the speed at which a circuit operates, the more power is consumed and thus more heat needs to be dissipated to prevent a chip melt-down. The fan-in is the maximum number of inputs that a gate can have and fan-out is the maximum number of other gates that can be connected to a gate's output.

4.6 Sequential Circuits and Finite State Machines

A *combinational circuit* has the property that its output values are uniquely determined for each set of input values, independent of its past input history. Each set of input values always causes the same output values whenever that set of input values is applied. The circuit is *time-sequence independent*, that is, it has *no memory*. A *sequential circuit* has the property that its outputs are a function of both the current input values and the previous sequence of input values. So the same set of input values, applied at different times, may cause different output values, depending on the past sequence of input values. The circuit is *time-sequence dependent* and thus has *memory*.

4.6.1 Finite State Machines

The behavior of a sequential circuit cannot be defined by a truth table. The output of a sequential circuit depends not only on its inputs but also on its current state, which reflects the past sequence of inputs. Each row in a truth table defines the output values corresponding to a set of input values, but does not include any information about the current state of the circuit. The behavior of a sequential circuit can be defined by a *finite state machine (FSM)*. An FSM consists of a set of nodes, each of which corresponds to a particular state of the circuit, and a set of labeled, directed arcs that connect some nodes to other nodes. The arcs emanating from a node are labeled with the input set(s) that transform the circuit from the state represented by that node to the state represented by the node at the end of the directed arc. For example, the FSM diagramed below has two inputs [C_1 , D] and two states [$Q=0$, $Q=1$]. The first number in each input set is the value of C_1 and the second number is the value of D .



In the diagram, each node is drawn as a circle with the state written inside the circle and the directed arcs are drawn as arrows. This FSM diagram is actually the definition of a D latch, which is a component that stores one bit of data. If the state is $Q=0$, the stored value is 0, if the state is $Q=1$, the stored value is 1.

Simple sequential circuits can also be defined by a *state transition table* (STT) In the STT corresponding to the above FSM, there is one row for each possible combination of current state Q_n and input value set $[C1, D]$. The last column in each row shows the next state Q_{n+1} when the current state is Q_n and the new input set is $[C1, D]$.

Q_n	C1	D	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

For example, when the current state is $Q=0$ and the new input set is $[1, 0]$, the FSM remains in the same state, but if the new input set is $[1, 1]$, the next state is $Q=1$, that is, the FSM makes a state transition of from $Q=0$ to $Q=1$. A condensed form of STT is often used. In this form, all rows in which the next state is the same as the current state are collapsed into a single row and the column for the current state is deleted. Condensing the above STT, we get

C1	D	Q_{n+1}
0	0	Q_n
0	1	Q_n
1	0	0
1	1	1

This table shows that when $C1=0$, the next state is the same as the current state, for all values of D . When $C1=1$ and $D=0$, the next state is always 0, so if the current state is $Q=0$ then there is no state transition, that is, the FSM stays in the state $Q=0$. Actually this STT can be condensed even further.

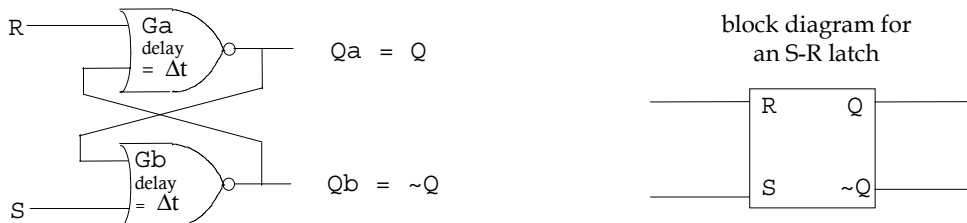
C1	D	Q_{n+1}
0	?	Q_n
1	0	0
1	1	1

Here the question mark means any value of D , that is, when $C1=0$, the next state is the same as the current state, for all values of D .

4.6.2 Flip-Flops

The atomic storage unit in a CPU is a latch or a flip-flop. The distinction between a latch and flip-flop will be discussed later. Some authors make no distinction and call every device of this type a flip-flop, Some authors are inconsistent, that is, no distinction is made between the behavior of a latch and a flip-flop, yet they use both terms. We will observe the distinction between latches and flip-flops, which will be explained later. We start our discussion with latches, which are simpler than flip-flops.

The simplest latch is the *S-R latch*. It is implemented with two cross-coupled NOR gates.



[Note: some authors use R-S flip-flop as the name for this circuit.]. The feedback from Q_a to the NOR gate G_b and from Q_b to the NOR gate G_a provides the memory. The latch has two stable states, the *reset state* stores 0 ($Q_a=0$) and the *set state* stores 1 ($Q_a=1$). The latch's normal input values are $R=0$ and $S=0$, which, as we will see, keeps the latch in a stable state. The Q_b output is always the complement of Q_a and the value stored in the latch is considered to be the output at Q .

Starting with $R=0$ and $S=0$, asserting R ($R=1$) and keeping $S=0$ will reset the latch, that is, store 0 in the latch (if 0 is already stored in the latch then this input has no effect). When the G_a NOR gate's R input becomes 1, after a Δt time delay, its Q_a output is forced to 0, regardless of the current value of the gate's other input (Q_b) since 1 NOR ?

always equals 0. Now both inputs to the G_b NOR gate are 0 (Q_a and S) so, after another Δt, its output (Q_b) will be 1. When R is returned to 0, the other input to G_a (Q_b) is 1, so G_a's output (Q_a) remains equal to 0.

Starting with R=0 and S=0, asserting S (S=1) and keeping R=0 will set the latch, that is, store 1 in the latch (if 1 is already stored in the latch then this input has no effect). When the G_b NOR gate's S input becomes 1, after a Δt time delay, its Q_b output is forced to 0, regardless of the current value of the gate's other input (Q_a). Now both inputs to the G_a NOR gate are 0 (Q_b and R) so, after another Δt, its output (Q_a) will be 1. When S is returned to 0, the other input to G_b (Q_a) is 1, so G_b's output (Q_b) remains equal to 0.

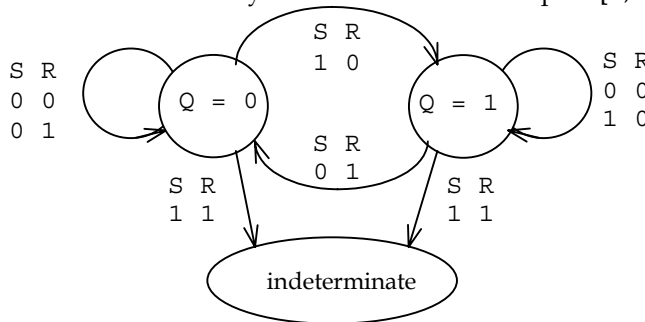
We see from the above discussion that

- a) when R changes from 0 to 1 (and S stays 0), Q will be 0 after Δt and ~Q will be 1 after 2Δt and
- b) when S changes from 0 to 1 (and R stays 0), ~Q will be 0 after Δt and Q will be 1 after 2Δt.

Therefore the maximum delay for an S-R latch is 2Δt, where the maximum delay is defined to be the interval between the time when the input changes and when all the output values reach there final states, no matter what values were input. The following diagram shows the timing, starting with Q_a=0 and Q_b=1, of the various changes in the outputs in response to the various changes in the input values.



The behavior of an S-R latch can be defined by an FSM that has two inputs [S, R] and two states [Q=0, Q=1].



The corresponding condensed STT is

S	R	Q _{n+1}
0	0	Q _n
0	1	0
1	0	1
1	1	?

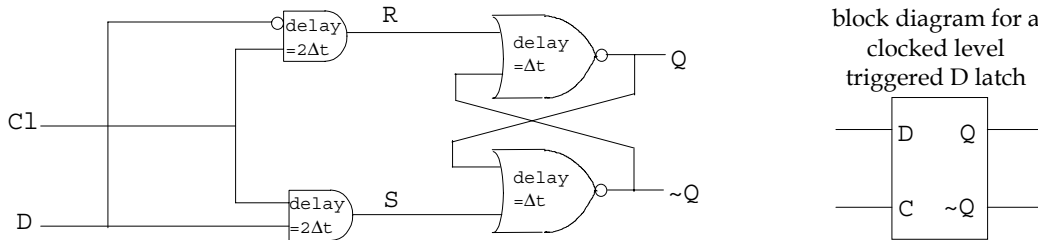
Here the question mark means that the next state is indeterminate, that is, it can be either Q=0 or Q=1.

In an S-R latch, the input set [R=1, S=1] is disallowed because the result is indeterminate. One input to each NOR gate is 1, so the each will output 0. When R and S both return to 0, both inputs to each NOR gate will be 0, so each gate's output will change to 1. Now one input to each gate is 1, so their outputs will change to 0. This oscillation could continue forever, but more likely, since the Δt of the two gates may not be exactly the same, one of the gates will switch before the other and the latch will settle into one of the two stable states, but which one is unpredictable. When the final state of any circuit is sensitive to the relative arrival time of signals, the result may be a *glitch*, which is an unwanted state or output. A circuit that can produce a glitch is said to have a *hazard*, but this may or may not cause a glitch, depending on the operating conditions of the circuit at a particular time.

The CU in a processor is governed by a *clock*, whose output signal synchronizes every state dependent circuit such as a latch or flip-flop. The clock's output is essentially a *fixed frequency* square wave (e.g., 500 MHz). One *cycle* of this output is called a *clock period* or often just a *clock*. One clock is equal to the inverse of the frequency (e.g., 1/500,000,000 = 0.000000002 = 2 ns).

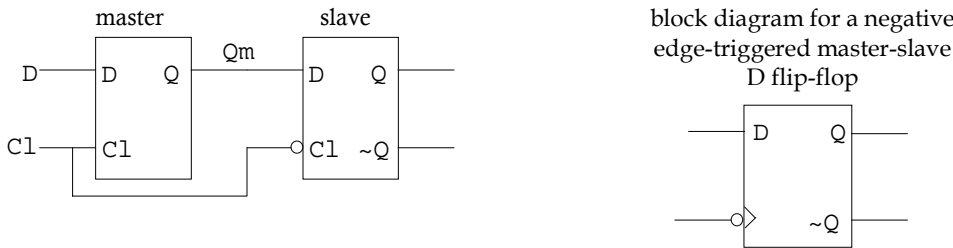
A *clocked latch* has an input called the *clock input* and the latch's state changes only while the clock input is 1; when the clock input is 0, its state does not change even if its other inputs (e.g., R and S) change. Note that the clock input of a latch is not directly connected to the clock's output, but to some other component such as a decoder. A component whose state can only change while the clock input is high, or only when it is low, is called *level-triggered*. In the basic (unclocked) S-R latch, there can be a change in the outputs within one or two Δt of any change in the input throughout the entire clock period. This can be a problem in some circuits.

A *D latch* is a clocked latch that, other than the clock input C1, has only a single input D. The example in Section 4.6.1 above is the FSM definition of a D latch.



Since the R input is always the complement of the S input, the disallowed $R=S=1$ input can never occur. When the C1 input is 1, the current value of D is stored in the latch (i.e., if $D=1$ then Q will become 1 and if $D=0$ then Q will become 0). When $D=0$ (and $C1=1$), Q will be 0 after $4\Delta t$ and $\sim Q$ will be 1 after $5\Delta t$ (don't forget the Δt delay in the negation bubble, which is shorthand for a NOT gate). When $D=1$ (and $C1=1$), $\sim Q$ will be 0 after $3\Delta t$ and Q will be 1 after $4\Delta t$. Thus the maximum delay for a D latch is $5\Delta t$. However, if only the Q output is used, the maximum delay is only $4\Delta t$.

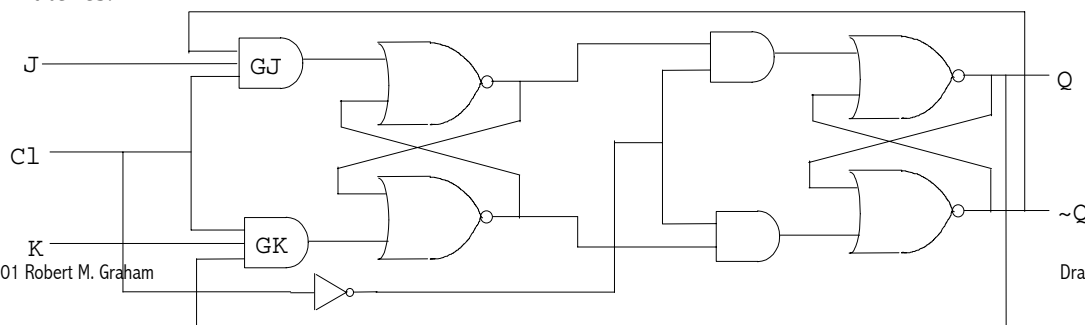
A *master-slave flip-flop* consists of two clocked D latches (or S-R latches) connected in series. The clock input to the second latch, called the *slave*, is the complement of the clock input to the first latch, called the *master*.



During the first half of a clock period ($C1=1$), Q_m will change in response to any change in the D input $4\Delta t$ time units after the input change. However, since the clock input to the slave is 0 during this time, there will be no change in the slave's output. When C1 falls to 0, the clock input to the slave becomes 1, so the value of Q_m is transferred to the slave. During this time the value of Q_m will not change even if the value of D changes. Thus a master-slave flip-flop will maintain a stable output value during the first half of a clock period and change only at the beginning of the second half of the period. This is known as *edge triggering* and the master-slave flip-flop is edge triggered, which makes it a true flip-flop. The \triangleright symbol replaces C1 at the clock input and indicates edge-triggering. The negation bubble outside the clock input means the flip-flop is negative edge-triggered; if there is no bubble, it is positive edge-triggered.

The distinction that many authors make between a latch and a flip-flop is that a *latch* is level triggered and a *flip-flop* is edge triggered. A *level-triggered*, clocked latch changes state continuously, with only a slight delay, in response to changes in the input throughout the time the clock input is high. An *edge-triggered* flip-flop changes state only on a low to high (*positive edge*) or high to low (*negative edge*) transition in the clock input. All input signals must be held constant for a short time before (*setup time*) and a short time after (*hold time*) the edge in order for an edge-triggered flip-flop to function properly.

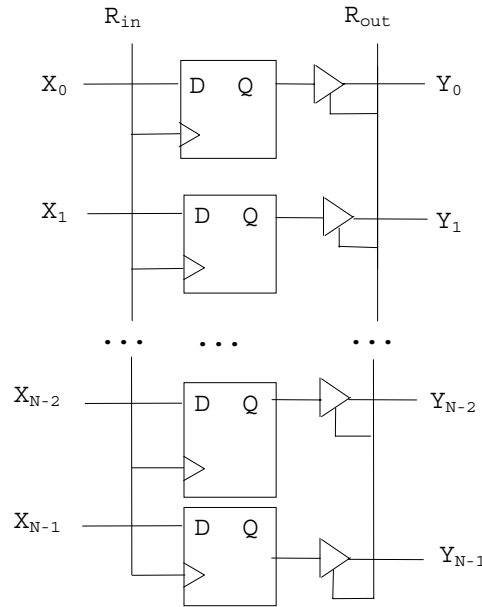
A *J-K flip-flop* has a structure similar to a master-slave flip-flop, but is made with two clocked S-R latches rather than two D latches.



In addition to the three inputs J , K , and Cl , its Q output is fed back into the GK AND gate and its $\sim Q$ output is fed back into the GJ AND gate. It is similar to an S-R latch in that the normal inputs [$J=K=0$] leave its state unchanged, the input [$J=0, K=1$] resets the flip-flop, and the input [$J=1, K=0$] sets the flip-flop. However, the input [$J=1, K=1$] is not disallowed, instead, because of the feedback, it complements the present state of the flip-flop, that is, the Q output becomes $\sim Q$ and the $\sim Q$ output becomes Q . This is a very useful property that will be required in a later section.

4.7 Storage and Shift Registers

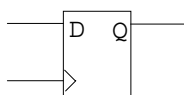
An N -bit *storage register* consists of N flip-flops, one for each bit, along with additional logic to control input and output to and from the register.



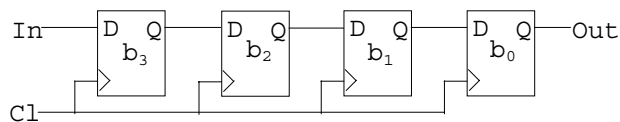
A storage register is normally connected to a bus. Since the value transmitted on the bus is often not a value that is to be stored into a register, we must control when a value on the bus is actually loaded into the register. The R_{in} line in the diagram controls the loading of a value into the register. It is connected to the clock input of each flip-flop. As long as R_{in} is zero, the values in the flip-flops do not change. When R_{in} is asserted, the values on X_{N-1}, \dots, X_0 are loaded into the flip-flops. The value stored in the register is always available on the Q outputs of the flip-flops (Y_{N-1}, \dots, Y_0) and can be read at any time. However, we do not want to connect the outputs of a register directly to the bus since the value in the register would be continuously put onto the bus and thus interfere with the use of the bus to transfer a value between other registers. We use a *tristate gate* in each flip-flop's output line to isolate its output from the bus. When the input at the lower side of the tristate gate is zero, its output is completely disconnected from the bus; when the input at the lower side of the tristate gate is one, the input on the left is passed on to the bus unchanged. The R_{out} line controls when the value in the register is actually put onto the bus. As long as R_{out} is zero, all the flip-flops in the register are effectively disconnected from the bus. When R_{out} is asserted, the flip-flops are connected to the bus and the value in the register goes onto the bus.

A shift register is a set of flip-flops connected in series. The following uses positive edge-triggered flip-flops.

positive edge-triggered
D flip-flop block diagram



4-bit, right shift register built with
positive edge-triggered D flip-flops

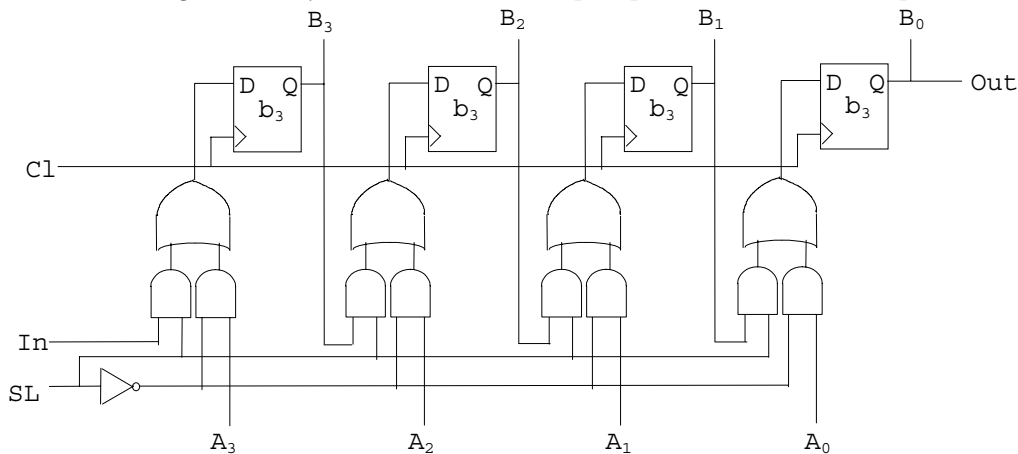


To perform a shift, we want the contents of each flip-flop to be transferred to the flip-flop immediately to its right when the Cl line is pulsed. In this case, the value in b_0 goes to Out , b_1 goes to b_0 , b_2 goes to b_1 , b_3 goes to b_2 , and In goes to b_3 . This shift register will shift the contents $b_3b_2b_1b_0$ one bit to the right, with the value on the In line going into b_3 , each time a positive edge appears on the Cl line. For example, if $b_3b_2b_1b_0=1010$ and $In=0$ then

the new value in the shift register will be 0101. Recall that the output of a level-triggered flip-flop will change in response to any change in its input throughout the high part of a clock period. Since the duration of the high part of a clock (half a clock period in duration) is much longer than the maximum delay through a flip-flop, which in the D flip-flops above is $4\Delta t$, using level-triggered flip-flops in a shift register would allow the value in b_3 to propagate through an indeterminate number of flip-flops to the right. We must use edge-triggered flip-flops, which change their state only when an edge arrives at the clock input. The length of the interval during which the value in an edge triggered flip-flop can change is smaller than the time it takes to transfer the value from one flip-flop to its neighbor. Thus none of the values can propagate more than one place to the right.

4.8 Serial-Parallel Conversion

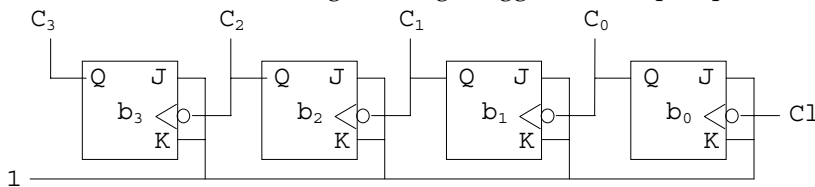
In a *parallel transfer* all the bits in a unit such as a byte or a word move at the same time on multiple wires (e.g., 8 wires for a byte). Internal buses in the CPU and the system bus use parallel transfers. In a *serial transfer* only one bit at a time moves on a single wire. Telephone modems uses serial transfers. Since parallel transfers are used internally in computers, *serial-to-parallel* and *parallel-to-serial* conversions are required in order to interface with a modem or other serial device. A shift register with *parallel load* of all its flip-flops is ideal for parallel-to-serial conversion and a shift register with *parallel read* of all its flip-flops is ideal for serial-to-parallel conversion.



The parallel output is available at all times, but the parallel loading is controlled by the 1-bit SL control input that selects either right shifting ($SL=1$) or parallel loading ($SL=0$). To do serial-to-parallel conversion, each bit of the serial input is presented on the serial input line In , one bit each clock, in lo to hi bit order, with the SL line held high during each clock. After all the serial bits have been shifted into the shift register, the value can be read on the parallel output lines $B_3B_2B_1B_0$. To do parallel-to-serial conversion, the value is loaded into the shift register on the parallel input lines $A_3A_2A_1A_0$, with the SL line held low. Then the contents of the shift register are shifted right, once for each bit in the value, with the SL line held high during each clock. The bits in the value will appear on the serial output line Out , in lo bit to hi bit order, one bit each clock.

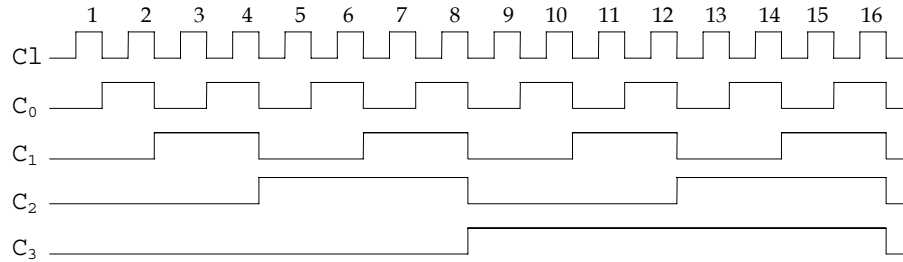
4.9 Binary Counters and Frequency Dividers

A 4-bit *binary counter* is built from four, negative edge-triggered J-K flip-flops.



Here both the J and K inputs of every flip-flop are always 1, thus every time a negative edge appears at the clock input of a flip-flop, its value will be complemented. The Q output of each flip-flop, except from the b_3 flip-flop, is connected to the clock input of the flip-flop to its immediate left. Every time the $C1$ input of the b_0 flip-flop is pulsed, its value is complemented. Assuming that initially every flip-flop is storing 0, the first pulse on $C1$ will change the value in the b_0 flip-flop from 0 to 1, which results in a positive edge on its Q output and in turn causes a positive edge on the clock input of the b_1 flip-flop, but its value will not change since it is negative edge-triggered. The second pulse on $C1$ will change the value in the b_0 flip-flop from 1 to 0, which results in a negative edge on its Q output and in turn causes a negative edge on the clock input of the b_1 flip-flop, so its value will be complemented. Thus the value in the b_0 flip-flop is changed every time $C1$ is pulsed and the value in the

b_1 flip-flop is changed every second time C_1 is pulsed. By induction, we see that the value in the b_2 flip-flop is changed every fourth time C_1 is pulsed and the value in the b_3 flip-flop is changed every eighth time C_1 is pulsed. At any instant in time, the binary number stored in the counter ($b_3b_2b_1b_0$) will be the total number of times C_1 was pulsed. A binary counter can also be used as a *frequency divider*. The waveforms of the clock input and the outputs of a binary counter



show this behavior. The C_1 input is pulsed at a fixed frequency f , so the frequency of the C_0 output will be $f/2$, the frequency of the C_1 output will be $f/4$, the frequency of the C_2 output will be $f/8$, and the frequency of the C_3 output will be $f/16$.