Using Logic Gates to Build Computer Component

Adel H A AlATIEH

Abstract—This paper Shows and introduce how to build a computer component from the basic logic gates

Index Terms -- Introduction, Logic gates & Truth Table, Combinational and Sequential Circuits, Boolean algebra, Building complicated circuits Using Logic Gates

I. INTRODUCTION

Digital signals

As we know that Digital signals have only a certain number of possible values and for the basic logic circuits that we will be discussing, there are only two possible values , a digital signal must still be represented by an analog voltage, it may be corrupted by noise and so on. Having two possible values then might work something like this, allowing for noise: let's suppose the two possible values for a voltage are zero and 1 volt. Then, if the voltage is less than 0.5 volts, assume its actual value is zero volts; if it is greater than 0.5 volts assume it's actually 1 volt. this could have some advantages : If the signal is supposed to be zero volts, it needs a lot of noise to make it appear as 1 volt, and vice versa. That is, the S/N ratio would have to be very poor for an error to occur. Although the digital signal doesn't contain much information, it's pretty robust. The circuits which use signals like these don't need to linear, it's actually best if they aren't, but tend to "stick" to voltage values of (in this example) zero or 1 volt.

Showing information in binary form

A (two-state) signal like this is called a binary signal, and can be used to represent any information which also has two possible states. These might be:

1. True/false
2. High/low
3. Yes/no
4. 1/0

- It turns out that this is not all that restrictive, because we can represent any amount of information by combining a number of such signals. The common game of "20 questions" is an example of this. Here, one of a (fairly large) number of possibilities is identified by successive "yes/no" answers. With each question, the number of possibilities potentially halves. Any sequence of answers can be written down as a string of symbols, each of which is "yes/no", or, if you like, "0/1" (the latter takes less paper..). Each one of these smallest units of information is called a bit (for binary digit) . and we can specify the information content of a document, message, or whatever by specifying the number of bits needed to represent it. Very often the information is inherently numerical, and the "0/1" interpretation of our two signal states is the most appropriate.

Revised Version Manuscript Received on May 22, 2015.
Adel H A Alateyah, Kuwait City, Kuwait, Kuwait 18502. Kuwait.

We are used to seeing numbers in decimal form; that is, base ten, where the digits 0 to 9 are used. As you are probably aware, we can also represent numbers in binary form, using only the digits 0 and -- The decimal integers 0 to 10 are represented in binary form as
Follows:

Decimal	*Binary*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010

Binary representation of integers 0 to 10.

With 4 bits we can count from 0 to 15 , we can represent 16 different numbers, or 16 different *states* (or "possibilities"). The more bits we have available, the more numbers we can represent. With n bits, 2n different states can be represented. Although it is more tedious for humans to write numbers in this way, it is much more convenient for electronic circuits to store and manipulate them, and the binary representation is used in computers. Most personal computers these days basically represent numbers using 32 or 64 bits. This is referred to as the *word length* of the computer. With 32 bits, we can count from 0 to 4,294,967,295 (= 232 or 4,294,967,296 possible states). The table below shows the number of possible states for some common word lengths encountered in digital circuits or computers. Note that a *byte* is a group of 8 bits.

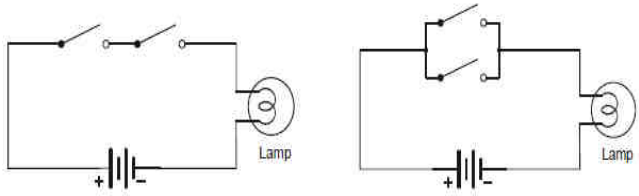
Word length in bits	*Possible number of states*
4	16
8 (1 byte)	256
16 (2 bytes)	65,536
32 (4 bytes)	4,294,967,296
64 (8 bytes)	18,446,744,073,709,551,616

- Number of states which can be represented by a given number of bits.

A Simple logic circuits:

- the left hand circuit below, which

makes use of two states. we call the two states ON and OFF, as it makes most sense.



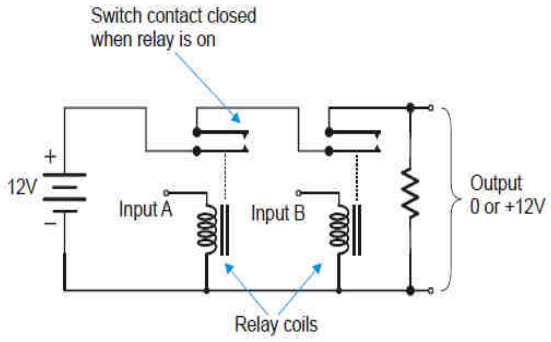
- **Circuits using switches to implement the AND & OR logic**

functions. This circuit has two "inputs" - the positions of the two switches (either ON or OFF), and one "output" -

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

whether the lamp is ON or OFF. Since the switches and the lamp are all in series, the lamp will light only if both switches are ON. Thus the lamp is ON only if switch 1 is ON **and** switch 2 is ON, but at no other time. We would say that this circuit performs an *AND* function. In the right hand circuit - Here the lamp is ON if switch 1 is ON **or** switch 2 is ON (and this includes the case when both switches are ON). We say that this circuit thus performs an OR function.

Now it wouldn't be much use if we had to rely on flipping switches to get results for calculations involving large numbers of binary signals. What we need is a circuit which takes binary signals say, as voltages, and generates outputs which are also voltages. One relatively primitive way of doing this is with *relays*, which can be thought of as switches activated by a voltage. Most cars have quite a few relays to switch relatively heavy currents for starter motors, headlights, rear window demisters and the like. In a relay, the voltage across a coil of wire wound around an iron rod causes a current to flow and a magnetic field to be generated (note that the coil is also an inductor, but it's not the inductance that's of interest to us here). The relatively strong magnetic field pulls a switch contact closed, and this can be used to control a large current. Here is a relay version of an AND circuit, with voltage inputs and output:



- AND logic function is implemented by A circuit uses relays

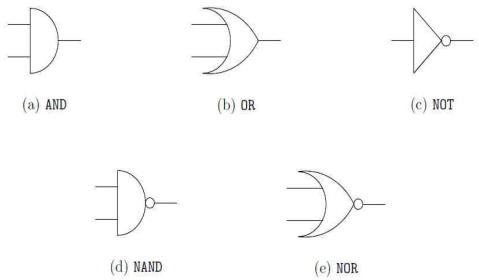
The advantage of this circuit is that the output voltage can now be used as an input to other circuits, so we could build quite complicated *logic* circuits. For example, we might implement something like this (assuming we have appropriate input signals from sensors): *"If reactor getting too hot AND ((NOT backup safety circuit active) OR , Homer Simpson is at control panel) then sound immediate evacuation alarm".* relays aren't really a practical proposition for complicated, highspeed or compact systems. Fortunately, we can use transistors, which can be coaxed to act in a somewhat similar way to relays, to build fast, cheap, compact and incredibly reliable logic circuits. It is now commonplace to buy integrated circuits (*ICs*, or "silicon chips") containing millions of such circuits.

II. The Logic gates &The Truth tables

Gates :

A gate is an electronic device with one or more inputs, each of which can assume , either the value 0 or the value 1. As mentioned earlier, the logical values 0 and 1 are generally represented electronically by two different voltage levels, but the

physical method of representation need not concern us. A gate usually has one output, which is a function of its inputs, and which is also either 0 or 1.

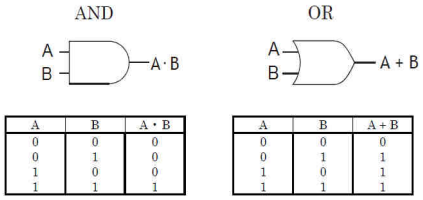


Symbols for gates.

A basic circuit which performs logical operations such as AND or OR is referred to as a *gate*. In the simple examples we have looked at so far,, there were only two inputs, but AND and OR (and some other) gates may have many inputs. The most common types of gates use voltages as logic signals, with, say, +5 volts representing a *high* "logic level" (or 1, or TRUE, ..etc…) and zero volts representing a *low* logic level (or 0, or FALSE, etc..).

- the truth tables for AND and OR gates, together with the

  symbols commonly used for them:



A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

Logic symbols for 2-input AND & OR gates and truth tables

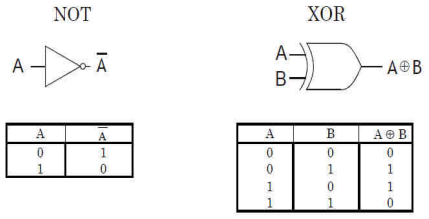
There are some other types of gates, the *NOT* logical function, and this gate is called an *inverter.*

Inverters

gates compute some particular Boolean function , AND and OR-gates are usually easy to build, as are NOT-gates, which are called inverters. AND & OR gates can have any number of inputs, although, as we discuss in Section 13.5, there is usually a practical limitation on how many inputs a gate can have. The output of an AND-gate is 1 if all its inputs are 1, and its output is 0 if any one or more of its inputs are 0. Likewise, the output of an OR-gate is 1 if one or more of its inputs are 1, and the output is 0 if all inputs are 0. The inverter (NOT-gate) has one input; its output is 1 if its input is 0 and 0 if its input is We also find it easy to implement NAND- and NOR-gates in most technologies. The NAND-gate produces the output 1 unless all its inputs are 1, in which case it produces the output 0. The NOR-gate produces the output 1 when all inputs are 0 and produces 0 otherwise. An example of a logical function that is harder to implement electronically is equivalence, which takes two inputs x and y and produces a 1 output if x and y are both 1 or both 0, and a 0 output when exactly one of x and y is 1. However, we can build equivalence circuits out of AND, OR , and NOT gates by implementing a circuit that realizes the logical function $xy + \bar{x}\bar{y}$. The symbols for thegates we have mentioned are shown in Fig. 13.1. In each case except for the inverter (NOT-gate), we have shown the gate with two inputs. However, we could easily show more than two inputs, by adding additional lines. A one-input AND- or OR-gate is possible, but doesn't really do anything; it just passes its input to the output. A one-input NAND- or NOR-

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

gate is really an inverter. This gate has only one input and one output, and changes a 0 at the input to a 1 at the output, and vice versa. The inverter is also said to *complement* or *negate* the input signal – that is, the complement of 0 is 1, and vice versa. The logic symbol and truth table for the inverter are shown at the left of the diagram below:

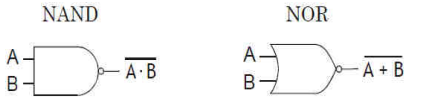


A	$\bar{A}$
0	1
1	0

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Logic symbols for the inverter (NOT gate) and XOR gate and truth tables for these functions.

The right-hand side of this diagram shows another common function – the *exclusive OR*, or *XOR* function. This is the same as an OR function, except when both inputs are high, and in this case the output is low. Another way of looking at the 2-input XOR function is that it gives a high output if the two inputs are different. The AND and OR functions may be combined with an inverter (the NOT function) to give NAND and NOR functions. These are the same as AND & OR except that the outputs are inverted. Real NAND and NOR gates are more common than AND & OR because they can be made more simply (and can operate a little faster). The symbols and truth tables are shown below. Notice the little "bubbles" on the gate outputs which indicate negation (inversion)



A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Logic symbols for 2-input NAND (NOT AND) & NOR (NOT OR) gates and truth tables.

III. Combinational and Sequential Circuits

There is a very close relationship between the logical expressions we can write using a collection of logical operators, such as AND, OR, and NOT, on one hand, and the circuits built from gates that perform the same set of operators, on the other hand. Before proceeding, we must focus our attention on an important class of circuits called combinational circuits. These circuits are acyclic, in the sense that the output of a gate cannot reach its input, even through a series of intermediate gates. We can use our knowledge of graphs to define precisely what we mean by a combinational circuit. First, draw a directed graph whose nodes correspond to the gates of the circuit. Add an arc u → v if the output of gate u is connected directly to any input of gate v. If the circuit's graph has no cycles, then the circuit is combinational; otherwise, it is sequential.

- **Sequential Circuits**

There is a very close relationship between the deterministic finite automata that we discussed in Chapter 10 and sequential circuits. While the subject is beyond the scope of this book, given any deterministic automaton, we can design a sequential circuit whose output is 1 exactly when the sequence of inputs of the automaton is accepted. To be more precise, the inputs of the automaton, which may be from any set of characters, must be encoded by the appropriate number of logical inputs (which each take the value 0 or 1); k logical inputs to the circuit can code up to 2k characters.

We shall discuss sequential circuits briefly at the end of this chapter. As we just saw in Example 13.2, sequential circuits have the ability to remember important things about the sequence of inputs seen so far, and thus they are needed for key components of computers, such as main memory and registers. Combinational circuits . on the other hand, can compute the values of logical functions, but they must work from a single setting for their inputs, and cannot remember what the inputs were set to previously. Nevertheless, combinational circuits are also vital components of computers. They are needed to add numbers, decode instructions into the electronic signals that cause the computer to perform those instructions, and many other tasks. In the following sections, we shall devote most of our attention to the design of combinational circuits.

IV. Boolean algebra

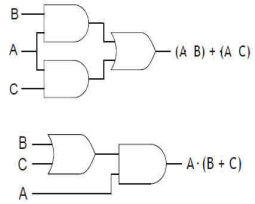
it is One of the important tools that digital designers use is *boolean algebra* (named after the nineteenth-century English mathematician, George Boole). It's a way of representing and manipulating logic signals and functions, and enables us, for example, to select the most economical combination of gates to carry out some function. The various logic functions correspond to *operators* (analogous to addition, multiplication etc.) which act on logic signals. They are shown in the following table, where A and B represent two logic signals (or *variables*, just like normal algebra), each of which may have the value 0 or 1:

Operation	*Symbol*	*Example*	*Meaning*
NOT	¯ (bar)	$\bar{A}$	NOT A (complement, inversion or negation)
AND	• (dot)	A • B	A AND B
OR	+ (plus sign)	A + B	A OR B
XOR	⊕	A ⊕ B	A XOR B

logical operators

Although the symbols for AND and OR might look like multiplication and addition (and they do have similarities) they operate differently . The result of any one of these operations is always just 0 or 1 (that is, one bit's worth , just like any input) For example, we find that:

(A • B) + (A • C) = A • (B + C) so that the following two logic circuits are equivalent, although the second uses one less gate:



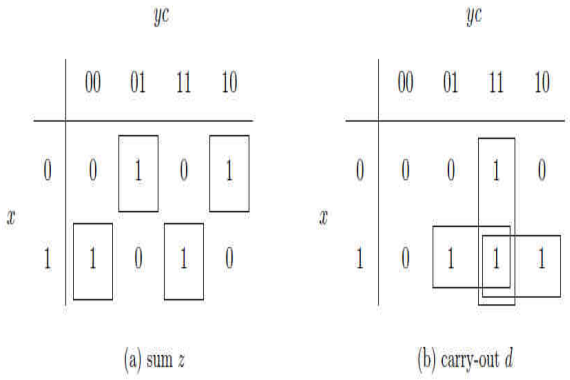
Two logic circuits which implement the same function.

Notice that this is like a similar rule in normal algebra, and there are other rules which are also similar. Although we won't take the idea of Boolean algebra any further than this, it at least gives you a taste of the possibilities.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

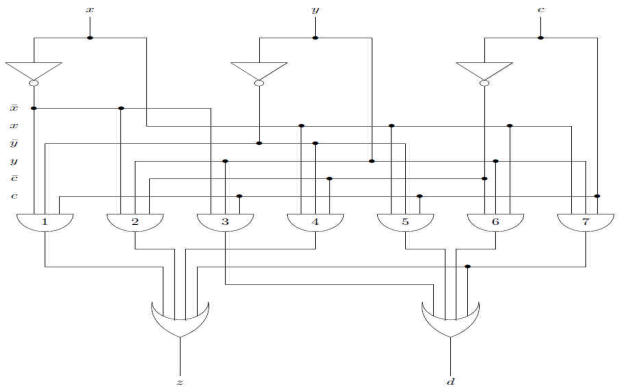
Circuit Diagram Convention

When circuits are complicated, as is the circuit in Fig. 13.10, there is a useful convention that helps simplify the drawing. Often, we need to have "wires" (the lines between an output and the input(s) to which it is connected) cross, without implying that they are part of the same wire. Thus, the standard convention for circuits says that wires are not connected unless, at the point of intersection, we place a dot. For example, the vertical line from the circuit input y is not connected to the horizontal lines labeled x or $\bar{x}$, even though it crosses those lines. It is connected to the horizontal line labeled y, because there is a dot at the point of intersection.



Karnaugh maps of the sum & carry-out functions.

In the above Fig. we see Karnaugh maps for z and d, the sum and carry-out functions of the one-bit adder. Of the eight possible minterms, seven appear in the functions for z or d, and only one, xyc, appears in both. and A systematically designed circuit for the one-bit adder is shown in the Fig , We begin by taking the circuit inputs and inverting them, using the three inverters at the top. Then we create AND-gates for each of the minterms that we need in one or more outputs. These gates are numbered 1 through 7, and each integer tells us which of its inputs are "true" circuit inputs, x, y, or c, and which are "complemented" inputs, $\bar{x}$, $\bar{y}$, or $\bar{c}$. That is, write the integer as a 3-bit binary number, and regard the bits as representing x, y, and c, in that order. For example, gate 4, or (100)2, has input x true and inputs y and c complemented; that is, it produces the output expression $x\bar{y}\bar{c}$. Notice that there is no gate 0 here, because the minterm $\bar{x}\bar{y}\bar{c}$ is not needed for either output. Finally, the circuit outputs, z and d, are assembled with OR-gates at the bottom. The OR-gate for z has inputs from the output of each AND-gate whose minterm makes z true, and the inputs to the OR-gate for d are selected similarly. Let us compute the output expressions for the circuit of Fig. 13.10. The topological order we shall use is the inverters first, then the AND-gates 1, 2, . . . , 7, and finally the OR-gates for z and d. First, the three inverters obviously have output expressions $\bar{x}$, $\bar{y}$, and $\bar{c}$. Then we already mentioned how the inputs to the AND-gates were selected and how the expression for the output of each is associated with the



One-bit-adder circuit.

binary representation of the number of the gate. Thus, gate 1 has output expression $\bar{x}\bar{y}c$. Finally, the output ofthe OR-gate z is the OR of the output expressions for gates 1, 2, 4, and 7, that is $\bar{x}\bar{y}c + \bar{x}y\bar{c} + x\bar{y}\bar{c} + xyc$ Similarly, the output of the OR-gate for d is the OR of the output expressions for gates 3, 5, 6, and 7, which is $\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc$ We leave it as an exercise to show that this expression is equivalent to the expression $yc + xc + xy$

- **Chips**

Chips generally have several "layers" of material that can be used, in combination . to build gates. Wires can run in any layer, to interconnect the gates , wires on different layers usually can cross without interacting. The feature size," roughly the minimum width of a wire, is in 1994 usually below half a micron (a micron is 0.00l millimeter, or about 0.00004 inches). Gates can be built in an area several microns on a side. The process by which chips are fabricated is complex. For example, one step might deposit a thin layer of a certain substance, called a photoresist, all over a chip. Then a photographic negative of the features desired on a certain layer is used. By shining light or a beam of electrons through the negative, the top layer can be etched away in places where the beam shines through, leaving only the desired circuit pieces.

Some Physical Constraints on Circuits

Today, most circuits are built as "chips," or integrated circuits. Large numbers of gates, perhaps as many as millions of gates, and the wires interconnecting them, are constructed out of semiconductor and metallic materials in an area about a centimeter (0.4 inches) on a side. The various "technologies," or methods of constructing integrated circuits, impose a number of constraints on the way efficient circuits can be designed. For example, we mentioned earlier that certain types of gates, such as AND, OR, and NOT, are easier to construct than other kinds

Circuit Speed

Associated with each gate is a delay, between the time that the inputs become active and the time that the output becomes available. This delay might be only a few nanoseconds (a nanosecond is 10^{-9} seconds), but in a complex circuit, such as the central processing unit of a computer, information propagates through many levels of gates, even during the execution of a single instruction. As modern computers perform instructions in much less than a microsecond (which is 10^{-6} seconds), it is evidently imperative that the number of gates through which a value must propagate be kept to a minimum. Thus, for a combinational circuit, the maximum number of gates that lie along

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

any path from an input to an output is analogous to the running time of a program as a figure of merit. That is, if we want our circuits to compute their outputs fast, we must minimize the longest path length in the graph of the circuit. The delay of a circuit is the number of gates on the longest path — that is, one plus the length of the path equals the delay. For example, the adder of Fig. 13.10 has delay 3, since the longest paths from input to output go through one of the inverters, then one of the AND-gates, and finally, through one of the OR-gates; there are many paths of length 3. Notice that, like running time, circuit delay only makes sense as an "order of magnitude" quantity. Different technologies will give us different values of the time that it takes an input of one gate to affect the output of that gate. Thus, if we have two circuits, of delay 10 and 20, respectively, we know that if implemented in the same technology, with all other factors being equal, the first will take half the time of the second. However, if we implement the second circuit in a faster technology, it could beat the first circuit implemented in the original technology.

Size Limitations

The cost of building a circuit is roughly proportional to the number of gates in the circuit, and so we would like to reduce the number of gates. Moreover, the size of a circuit also influences its speed, and small circuits tend to run faster. In general , the more gates a circuit has, the greater the area on a chip that it will consume. There are at least two negative effects of using a large area.

1. If the area is large, long wires are needed to connect gates that are located far apart. The longer a wire is, the longer it takes a signal to travel from one end to the other. This propagation delay is another source of delay in the circuit, in addition to the time it takes a gate to "compute" its output .
2. There is a limit to how large chips can be, because the larger they are, the more likely it is that there will be an imperfection that causes the chip to fail.

If we have to divide a circuit across several chips, then wires connecting the chips will introduce a severe propagation delay.

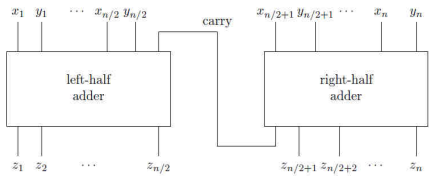
Our conclusion is that there is a significant benefit to keeping the number of gates in a circuit low.

Fan-In and Fan-Out Limitations

it is a constraint on the design of circuits comes from physical realities. We pay a penalty for gates that have too many inputs or that have their outputs connected to too many other inputs. The number of inputs of a gate is called its fan-in, and the number of inputs to which the output of a gate is connected is that gate's fanout. While, in principle, there is no limit on fan-in or fan-out, in practice, gates with large fan-in and/or fan-out will be slower than gates with smaller fan-in and fan-out. Thus, we shall try to design our circuits with limited fan-in and fan-out.

A Divide-and-Conquer Addition Circuit

One of the key parts of a computer is a circuit that adds two numbers. While actual microprocessor circuits do more, we shall study the essence of the problem by designing a circuit to add two nonnegative integers. This problem is quite instructive as an example of divide-and-conquer circuit design. We can build an adder for n-bit numbers from n one-bit adders, connected in one of several ways. use the circuit as a one-bit-adder circuit. This circuit has a delay of 3, which is close to the best we can do , The simplest approach to building an adder circuit is the ripple-carry adder . In this circuit, an output of each one-bit adder becomes an input of the next one-bit adder, so that adding two n-bit numbers incurs a delay of 3n. For example, in the case where n = 32, the circuit delay is 96. A Recursive Addition Circuit We can design an adder circuit with significantly less delay if we use the divide-and-conquer strategy of designing a circuit for n/2 bits and using two of them, together with some additional circuitry, to make an n-bit adder. In Example 13.6, we spoke of a divide-and-conquer circuit for taking the OR of many bits, using 2-input OR- gates. That was a particularly simple example of the divide-and-conquer technique, since each of the smaller circuits performed exactly the desired function (OR), and the combination of outputs of the subcircuits was very simple (they were fed to an OR-gate). The two half-size circuits did their work at the same time (in parallel), so their delays did not add. For the adder, we need to do something more subtle. A naive way to start is to add the left half of the bits (high-order bits) and add the right half of the bits (low-order bits), using identical half-size adder circuits. However, unlike the n-bit OR example, where we could work on the left and right halves independently, it seems that for the adder, the addition for the left half cannot begin until the right half is finished and passes its carry to the rightmost bit in the left half, as suggested If so, we shall find that the "divide-and-conquer" circuit is actually identical to the ripple-carry adder, and we have not improved the delay at all. The additional "trick" we need is to realize that we can begin the computation of the left half without knowing the carry out of the right half, provided we compute more than just the sum. We need to answer two questions. First, what would the sum be if there is no carry into the rightmost place in the left half, and econd, what would the sum be if there is a carry-in?3 We can then allow the circuits for the left and right halves to compute their two answers at the same time. Once both have been completed, we can tell whether or not there is a carry from the right half to the left. That tells us which answer is correct, and with three more levels of delay, we can select the correct answer for the left side. Thus, the delay to add n bits will be just three more than the delay to add n/2 bits, leading to a circuit of delay 3(1 + log2 n). That compares very well with the ripple-carry adder for n = 32; the divide-and-conquer adder will have delay 3(1+ log2 32) = 3(1+ 5) = 18, compared with 96 for the ripple-carry adder.



-An inefficient divide-and-conquer design for an adder.

More precisely, we define an n-adder to be a circuit with inputs x1, x2, . . . , xn & y1, y2, . . . , yn, representing two n-bit integers, and outputs

1. s1, s2, . . . , sn, the n-bit sum (excluding a carry out of the leftmost place, i.e., out of the place belonging to x1 and y1) of the inputs, assuming that there is no carry into the rightmost place (the place of xn and yn).
2. t1, t2, . . . , tn, the n-bit sum of the inputs, assuming that there is a carry into the rightmost place.
3. p, the carry-propagate bit, which is 1 if there is a carry out of the leftmost place, on the assumption that there is a carry into the rightmost place.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

4. g, the carry-generate bit, which is 1 if there is a carry out of the leftmost place, even if there is no carry into the rightmost place.

Note that g → p; that is, if g is 1, then p must be 1. However, g can be 0, and p still be 1. it is clear , if the x's are 1010 · · · , and the y's are 0101 · · · , then g = 0, because when there is no carry in, the sum is all 1's and there is no carry out of the

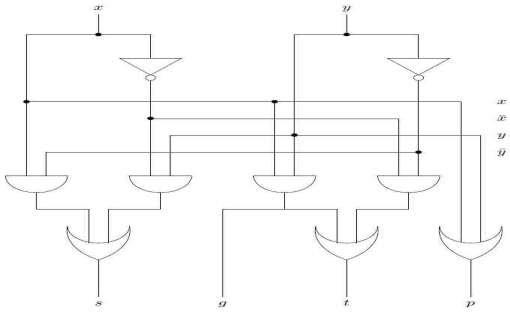
leftmost place. On the other hand, if there is a carry into the rightmost position, then the last n bits of the sum are all 0's, and there is a carry out of the leftmost place; thus p = 1.

We shall construct an n-adder recursively, for n a power of 2. BASIS. Consider the case n = 1. Here we have two inputs, x and y, and we need to compute four outputs, s, t, p, and g, given by the logical expressions

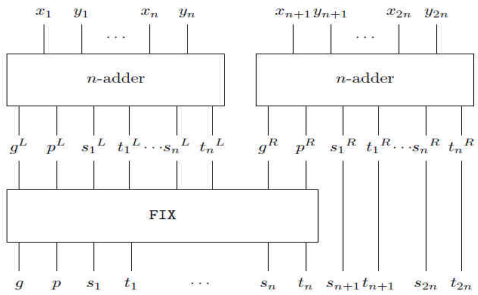
$s = x\bar{}\,y + \bar{}\,xy$
$t = xy + \bar{}\,x\bar{}\,y$
$g = xy$
$p = x + y$

To see why these expressions are correct, first assume there is no carry into the one place in question. Then the sum bit, which is 1 if an odd number of x, y, and the carry-in are 1, will be 1 if exactly one of x and y is 1. The expression for s above clearly has that

property. Further, with no carry-in, there can only be a



Adder Circuit



The Divide-and-Conquer Adder

Delay of the Divide-and-Conquer Adder

Let D(n) be the delay of the n-adder we just designed. We can write a recurrence relation for D as follows. For the basis, n = 1, examine the basis circuit and conclude that the delay is 3. Thus, D(1) = 3. Now examine the inductive construction of the circuit . of the circuit is the delay of the n-adders plus the delay of the FIX circuitry. Then adders have delay D(n). Each of the expressions developed for the FIX circuitry yields a simple circuit with at most three levels.

Thus, D(2n) is three more than D(n). The recurrence relation for D(n) is thus

D(1) = 3
D(2n) = D(n) + 3

The solution, for numbers of bits that are powers of 2, begins D(1) = 3, D(2) = 6, D(4) = 9, D(8) = 12, D(16) = 15, D(32) = 18, and so on. The solution to the recurrence is

D(n) = 3(1 + log2 n)

particular, note that for a 32-bit adder, the delay of 18 is much less than the delay of 96 for the 32-bit ripple-carry adder. Number of Gates Used by the Divide-and-Conquer Adder We should also check that the number of gates is reasonable. Let G(n) be the number of gates used in an n-adder circuit. The basis is G(1) = 9, T, the inductive case, has 2G(n) gates in the two n-adder sub circuits. To this amount, we must add the number of gates in the FIX circuitry. As we may invert gR and pR once, each of the n si's and ti's can be computed with three gates each (two AND's and an OR), or 6n gates total. To this quantity we add the two inverters for gR and pR, and we must add the two gates each that we need to compute g and p. The total number of gates in the FIX circuitry is thus 6n + 6. The recurrence for G is hence ,G(1) = 9 , G(2n) = 2G(n) + 6n + 6 Again, our function is defined only when n is a power of 2. The first six values The closed-form expression for G(n) is 3n log2 n + 15n − 6, for n a power of 2 , Actually, we can do with somewhat fewer gates, if all we want is a 32-bit adder . For then, we know that there is no carry-in at the right of the 32nd bit, and so the value of p, and the values of t1, t2, ... , t32 need not be computed at the last stage of the circuit. Similarly, the right-half 16-adder does not need to compute its carry propagate bit or its 16 t-values; the right-half 8-adder in the right 16-adder does not need to compute its p or t's and so on. It is interesting to compare the number of gates used by the divide-and-conquer adder with the number of gates used by the ripple-carry adder. The circuit for a full adder that we designed in Fig. 13.10 uses 12 gates. Thus, an n-bit ripple-carry

n	$G(n)$
1	9
2	30
4	78
8	186
16	426
32	954

Numbers of gates can be used by n-adders.

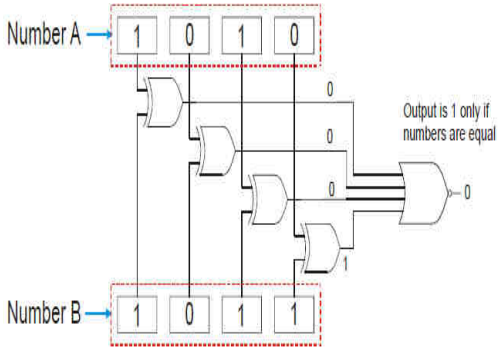
adder uses 12n gates, and for n = 32, this number is 384 (we can save a few gates if we remember that the carry into the rightmost bit is 0). We see that for the interesting case, n = 32, the ripple-carry adder, while much slower, does use fewer than half as many gates as the divide-and-conquer adder. Moreover, the latter's growth rate, O(n log n), is higher than the growth rate of the ripple-carry adder, O(n), so that the difference in the number of gates gets larger as n grows. However, the ratio is only O(log n), so that the difference in the number of gates used is not severe. As the difference in the time required by the two classes of circuits is much more significant [O(n) vs. O(log n)], some sort of divide-and-conquer adder is used in essentially all modern computers.

V. Building complicated circuits Using Logic Gates

By combining a number of gates, we can start building more practically useful circuits. For example, the circuit shown

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

below compares two 4-bit binary numbers, giving an output of 1 if they are equal, or 0 if they are different.



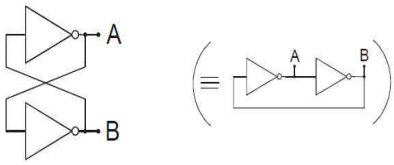
A circuit which compares two 4-bit binary numbers.

In order to see how it works, you will need to remember the behaviour of the exclusive-OR (XOR) and NOR gates. Recall that the XOR gate gives an output of 0 only if its inputs are the **same**. The NOR gate here has four inputs rather than two, but remember that it is a "NOT (OR)", so that it will give an output of 1 **only** if all of its inputs are 0. Here's how it works: a separate XOR gate compares the two bits at each bit position of the two numbers. If the bits are the same, then the XOR will give an output of 0. A 4-input NOR gate then combines the results from all the XOR gates. If the two numbers are identical, then all the XOR outputs will be 0, and the output of the NOR will be 1. On the other hand, if the numbers are different, then at least one of the XOR outputs will be 1, forcing the output of the NOR gate to zero. For the two binary numbers being compared in the diagram above (1010 and 1011), the first three bits are identical, so that the outputs of the first three XOR gates are 0. The last bit is different, but it only takes one input of the NOR gate to be 1 to make its output zero. The circuit can easily be expanded to handle larger numbers of bits, and might serve as a useful building block in a larger digital system, where, for example, a notification might be required when a counter reaches a certain value.

VI. FLIP FLOPS - REGISTERS- COUNTERS

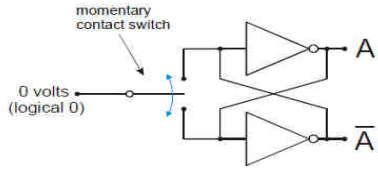
1. Flip-flops as memory elements

The figure below shows a strange-looking circuit – two inverters, with outputs A and B, talking only to each other (note that you could redraw this circuit as a simple closed loop with a "chain" of two inverters).



Cross-coupled inverters form a basic flip-flop.

suppose for a start that outputs A and B can only have values 0 or 1.Then, let's assume that A = 1. Is this OK? The output A goes into the lower inverter and is complemented, so that B must be 0. Output B is fed to the upper inverter, producing an output of 1 at A. This is the same as our original assumption, and tells us that everything is consistent; the circuit can have this state. Furthermore, there is no reason for the state to change. It is *stable*. What if we had chosen A = 0 instead? Following the same chain of reasoning, we could argue that this is **also** a stable state (but obviously the only other one). Thus the circuit has two stable states; it is referred to as a *bistable* circuit, or a *flip-flop*. Note also that B = A. Another way of looking at it is to say that the circuit has a "memory". If it is somehow forced into one state, it will stay there. So, in principle, we have a simple way of "saving" binary information (at least as long as the power is not turned off!). The only problem we have is that it's not clear how to make the flip-flop "flip" or "flop" on demand. A "brute-force" method is shown in the diagram below, where a switch can connect the input of either flip-flop (and hence the output of the other) directly to a voltage source representing logical 0, or just hang loose in the middle position. Connecting one gate input to logical 0 also forces the input of the other gate to 1, and hence its output to 0, and so the circuit can be "set" or "reset".

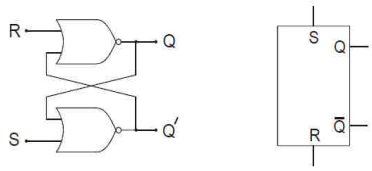


Adding a switch enables the flip-flop to be forced into one state or the other.

In practice we might not simply connect outputs to 0 volts like this, as the gates might be damaged; we would include resistors in appropriate places. A version of this technique is often used to *"de-bounce"* switches, such as pushbuttons on a panel connected to logic circuits. It is normal for switch contacts to "chatter" as they close, causing multiple changes of signal level. The de-bouncing scheme makes sure that only one change occurs, at the time the switch first makes contact.

2. The R-S flip-flop

Using switches to set values in flip-flops clearly isn't going to be much practical use, except in some fairly simple applications. We need to be able to set and reset flip-flops using logic signals. The circuit below shows such a possibility. It is called an *R-S flip-flop* (where the R and S refer to the *reset* and *set* functions), and this particular one is made by cross-coupling two NOR gates, rather than simple inverters. This leaves two spare gate inputs, which are used to set and reset the circuit. The symbol for this R-S flip-flop is also shown on the right of the diagram.



A flip-flop formed from two NOR gates makes an R-S flip-flop which may be set and reset via the two inputs R and S.

Here's how it works:

• If R = S = 0 then the two NOR gates just function as inverters for the signals fed between the two gates. (To see this, look at the truth table for a NOR gate with one input set to 0. The output is then just always the complement of the other input.) Thus the circuit behaves exactly as the simple pair of inverters – that is, it just stays in whichever state it finds itself, and acts as a one-bit memory.

• If R = 1 and S = 0 (that is, we make the reset input high), then output Q is forced to 0. (To see why, look at the NOR truth table again.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

Making **any** input 1 forces the output to 0). Q′ (= Q) is thus forced to 1, and the flip-flop is **reset**. It doesn't matter how many times this is then repeated; the circuit will still stay reset.

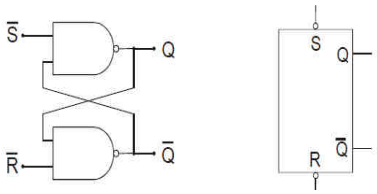
Note: Q is taken to be the "normal" output of the flip-flop (Q is always just its complement). Making Q = 1 is said to be *setting* the flip-flop, while making Q = 0 is re*setting* it. *Set* and *reset* are often also called *preset* and *clear*.

• If R = 0 and S = 1 (that is, we make the *set* input high), then output Q´ is forced to 0 (it's just the reverse of the previous case – the circuit is symmetric). Q is thus forced to 1, and the flip-flop is **set**.

• If R = 1 and S = 1 at the same time, we have a problem. Why? Because the outputs of both gates will be forced low. Then, when R and S are both returned to 0, which state will the circuit choose? (Note that Q = Q´ = 0 is **not** a stable state if R = S = 0). The answer is – we don't know! It all depends on whether R or S went back to 0 first. ***For this reason, the combination R = S = 1 is not allowed***.

So, to store a 0 or 1 in this R-S flip-flop, we momentarily set R **or** S respectively to 1.

It is also possible to build an R-S flip-flop using NAND gates, as shown inthe diagram below



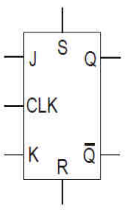
Another R-S flip-flop - this time using NAND gates.

We won't work through the logic of it, but the R-S input signals used here have to be the **complement** of those used in the previous (NOR) case. That is, the set and reset signals (labelled S and R) are normally 1 in "remember" mode, and are taken to 0 to set or reset the circuit. Here the state S = R = 0 is not allowed.

Note that these "inverted sense" input signals are denoted by bars over the S and R (hence becoming S and R) and by the little bubbles at the S and R inputs on the flip-flop symbol.

3. The J-K flip-flop

Although the R-S flip-flop is occasionally used, the *J-K flip-flop* is far more commonly found in digital systems. In a sense it is an enhancement of the R-S type, being rather more versatile, but it does has some fundamental differences. It has two inputs, *J* and *K*, which function somewhat like the S and R inputs on an R-S flip-flop, as well as a third *clock* input. The symbol for it is shown in the diagram below. The particular variety of J-K flip-flop shown in the diagram also has R and S inputs, which function in the same way as in a plain R-S flip-flop (to force it into one state or other). However, we won't be concerned with them here; they are not a necessary feature of J-K flip-flops.



J	K	Clock	Q^{n+1}
0	0	↓	Q^n
0	1	↓	0
1	0	↓	1
1	1	↓	$\overline{Q^n}$

The J-K flip-flop and its truth table. (Note that the R and S inputs are not provided on all J-K flip-flops.)

Here's how the J-K flip-flop works:

• A change can **only** occur when the clock **rises** (that is, changes from 0 to 1). Nothing can happen at any other time (for example, when the clock **falls**). The flip-flop is said to be *rising-edge-triggered*. (It is also possible to get J-K flip-flops which are *falling-edge-triggered*.)

• **What** happens depends on the values of J and K at the instant the clock rises. Now, the truth table:

• The third column of the truth table is just to remind you that the values of J and K are only important at the moment the clock rises.

• The fourth column indicates the state which Q goes to after the clock rises, for a particular set of values of J and K. The notation looks a bit strange, but here's what it means: Qn+1 means the state **after** the clock rises, while Qn means the state **before**. So the four entries in column 4 for Qn+1 mean:

Q^n: (if J = K = 0) Q doesn't change ($Q^{n+1} = Q^n$)

0: (if J = 0, K = 1) Q is set to 0 (regardless of previous state)

1: (if J = 1, K = 0) Q is set to 1 (regardless of previous state)

$\overline{Q^n}$: (if J = K = 1) Q changes to the **complement** of what it was. That is, if Q was 0, it goes to 1, and vice versa. This is known as *toggling*. Note that this behaviour did not occur with the R-S flip-flop. The J-K truth table is actually not too difficult to remember. The rules are:

(1) *Things only happen when the clock rises.*
(2) *If J = K = 0 then nothing happens.*
(3) *If J = K = 1 then the flip-flop toggles*
(4) *If J ≠ K then Q goes to the value that J has.*

The J-K flip-flop has some advantages over the R-S type. First, because changes are controlled by the clock, the operation of many circuits can be synchronised. Second, the behaviour is defined for all values of J and K (remember that R = S = 1 was not allowed with the R-S flip-flop). Third, the toggling behaviour when J = K = 1 allows us to design circuits which do some neat things.

There are many useful circuits we can build by connecting J-K flip-flops together. Two of the most important are *shift registers* and *counters*. Let's look at some of these.

4. Shift registers

In digital parlance, a *register* is simply a fancy name for a memory circuit to hold one "chunk" of information consisting of a number of bits. A *shift register* has the ability to "shift" all the bits of the binary number contained in it one place to the left (or right, or either, depending on its design), for each "tick" of a clock. It's exactly like having a full row of seats at the cinema, each seating a girl (1) or boy (0). Some newcomer wants to sit at the end, so everybody shifts along by one at the same time to accommodate them. Of course, somebody also gets dropped off the other end, since there are only a fixed number of seats. So the situation might look like this: *Before shift:* 1 → 1010101100110100 --- (new girl) *After shift:* 1101010110011010 → 0 --- (boy lost off end) The figure below shows a single stage of a shift

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

register using a J-K flipflop. J is set to the new value to be *shifted in*, while K is set to its complement (this is easy to arrange with an inverter if necessary). When the clock rises, the J and K values are transferred to Q and Q respectively (refer back to the J-K truth table to see why). The old values of Q & Q are lost from this flip-flop, but are shifted to the following stage at the same time

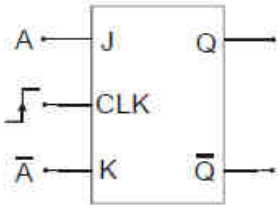
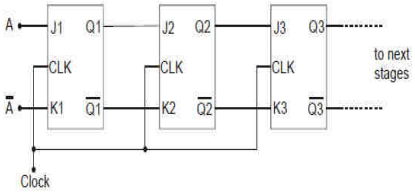
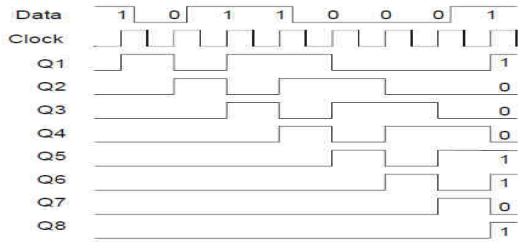


Figure 16.6 A basic shift register stage. With complementary J and K inputs, the data at J and K are transferred to Q and Q on rising clock edges. We can string together as many basic stages as we like, each feeding the next, as shown in the 3-stage example below. Note that the same clock signal connects to all stages. After the clock rises each time, a new value must be present at the input (A) to get shifted into the register.



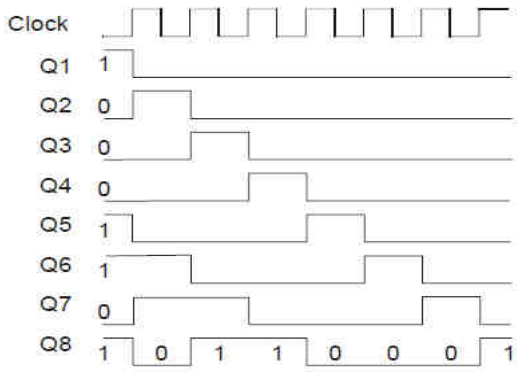
A three-stage shift register.

A *timing diagram* is shown below for an 8-stage shift register, with outputs named Q1 to Q8 (flip-flops 1 to 8). Initially the shift register contains all zeroes (see down left-hand side). Notice that the bits comprising the data (input) signal (going to flip-flop 1) are presented one bit at a time (*serially)*, and are changed just after the clock rises each time. Notice also that Q1 follows the input signal (but delayed by just a little; it doesn't change until the clock rises). Q2 follows Q1, but one clock cycle later, and so on. Eventually, after 8 clock cycles, the 8 bits of the input signal are shifted along to appear at Q1 through Q8.



A Timing diagram for serial-to-parallel conversion

using an 8-bit shift register. Q1 to Q8 are all initially 0, but after 8 clock cycles are set to the values of the incoming serial data. At this point all the bits of the signal are available simultaneously at the flip-flop outputs – that is, in *parallel*, rather than one at a time. Since the *serial* data stream is converted into *parallel* form, the shift register thus acts as a *serial-to-parallel* converter. (In practice, a little more circuitry would be required to inform the eventual receiver of the data that a new output was available after every 8 bits.) This might be useful, for example, if you wanted to test whether consecutive bits of the serial data stream were equal to some particular pattern (you would also need a little extra circuitry), or if they had to be sent to another device which required them all at the same time. For example, printers used with personal computers used to accept their data this way, one byte (8 bits) at a time, via the computer's *parallel port*. A shift register can also be used to perform *parallel-to-serial* conversion, as illustrated in the next timing diagram. Here the idea is that the shift register is "loaded" with the new data, 8 bits at a time. These 8 bits are then shifted to the right one bit at a time, appearing at Q8. After 8 clock cycles, the next 8 bits are loaded, and so on.



A Timing diagram for parallel-to-serial conversion Using an 8-bit shift register.

5. Counters

One particularly useful class of circuits which can be constructed with J-K , flip-flops is *counters*. Just like a digital clock, a counter steps through a specified sequence of numbers with each "tick" of a clock. In fact, let's just consider this example for a moment to tune in to how counters operate. A digital clock is probably best designed as, say, 3 counters, two of which count through 0 to 59 (seconds and minutes), and one which counts through 0 to 11 (or 0 to 23). When the seconds counter goes past 59, it resets to zero, as well as sending a "minute tick" to the minutes counter (this is a "carry"). And so on. It wouldn't be difficult to design such a clock with flip-flops and gates, but we'll do something a little bit easier. Here we're just going to consider *binary* counters. That is, circuits which count through a series of states which represent binary numbers, as in the following table (similar to the one in the previous chapter).

Decimal	*Binary*	*Decimal*	*Binary*
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

The Successive states of a 4-bit binary counter counting from 0 to 15 (decimal).

As a start, let's look a bit closer at a J-K flip-flop which is set up to *toggle* - (that is when J = K = 1), as shown in the diagram below. We know that for every clock cycle (when the clock rises), its state **changes**, from 0→1 or 1→0. If we look at the timing diagram on the right, we see that Q changes only **half as often** as the clock. (That is, the frequency of the signal at Q is half that of the clock. We could make immediate use of this if we had need for a circuit to divide the frequency of a (digital) signal by two.)

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

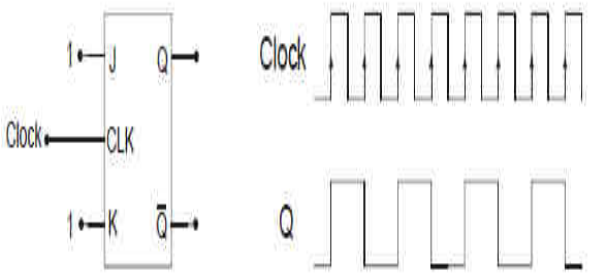
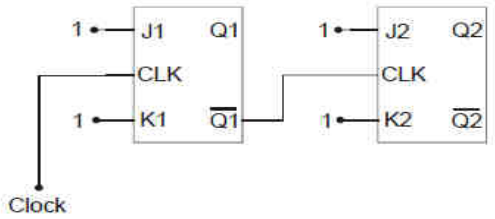
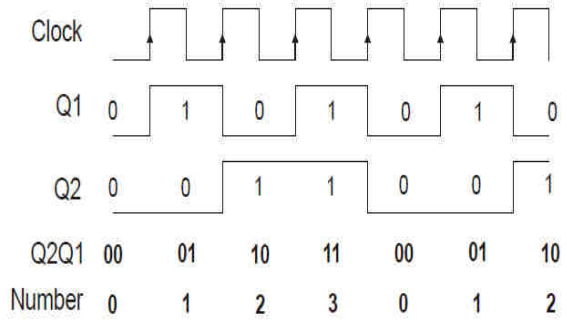


Figure 16.10 A J-K flip-flop with J = K = 1 toggles (changes state) on rising clock edges to give an output which changes at one-half the clock rate. If you look at the right-hand bit (usually referred to as *least-significant bit* or *LSB*) of each binary number in the table above, and compare it with the next bit (one from the right), you will notice that the LSB changes exactly twice as often as its left neighbour as the count proceeds. This bears a striking similarity to the relationship between the clock and Q signals in the timing diagram above. Similarly, the next bit to the left changes half as often again, and so on. Now suppose we connect two J-K flip-flops together as shown below. That is, the output Q of the first flip-flop is used as the clock input for the next flip-flop. The output of the second flip-flop will now change at one-quarter the rate of the clock.



The Cascading two stages gives a final output (Q2 or Q2) which is one-quarter the clock rate. The circuit also functions as a two-bit binary counter.

Assume that initially Q1 = 0 and Q2 = 0. Let's look at what happens on successive clock cycles. The timing diagram is shown below.



The Timing diagram for the two-bit binary counter.

Notice that:
-Every time the clock rises, Q1 (and Q1 of course) changes.
-Every time Q1 falls (that is, Q1 rises), Q2 changes. -After 4 clock cycles, everything is back to where it started, with Q1 = Q2 = 0. This is probably not surprising – there are only 4 different possibilities for the values of Q1 and Q2 taken together. That is, the whole circuit has only 4 different possible states. It certainly can't take longer than 4 clock cycles to return!

• Taken together as a binary number, the pair {Q2Q1} cycles through the values 00, 01, 10, 11, …(decimal 0, 1, 2, 3,…). That is, we have constructed a 2-bit binary counter.

VII. Conclusion

important computer components like Flip Flops , Registers and counters have been built by using the basic logic gates .

References

1. R. Landauer, "Irreversibility and Heat Generation in the Computational Process", IBM Journal of Research and Development, 1961.
2. D. P. Vasudevan, P.K. Lala , J. Di and J.P Parkerson, "Reversible–Logic Design with Online Testability", IEEE Trans. on Intrumentation and Measurement, April 2006.
3. C. H. Bennett, "Logical Reversibililty of Computation", IBM J. Research and Development, November 1973.
4. R. Feynman, "Quantum Mechanical Computers" , Optics News, 1985.
5. T. Toffoli, "Reversible Computing", Tech memo MIT/LCS/ TM-151, MIT Lab for Computer Science, 1980.
6. H. Thapliyal and N. Ranganathan, "Design of Reversible Sequential Circuits Optimizing Quantum Cost, Delay and Garbage Outputs," ACM Journal of Emerging Technologies in Computing Systems, Dec. 2010.
7. Abu Sadat Md. Sayem and Masashi Ueda, "Optimization of Reversible Sequential Circuits," Journal of Computing, 2010.
8. E. Fredkin and T. Toffoli, "Conservative Logic", Int˝l J. Theoretical Physics, 1982.
9. Peres, "Rversible Logic and Quantum Computers", Physical review A, 1985.
10. M.P Frank, "Introduction to Reversible Computing: Motivation, Progress and Challenges", Proceedings of the 2nd Conference on Computing Frontiers, 2005.
11. Diganta Sengupta, Mahamuda Sultana, Atal Chaudhuri, "Realization of a Novel Reversible SCG Gate and its Application for Designing Parallel Adder/Subtractor and Match Logic", International Journal of Computer Applications (0975-8887), October 2011.
12. B. Raghu kanth, B. Murali Krishna, M. Sridhar, V. G. Santhi Swaroop, "A Distinguish between Reversible and Conventional Logic Gates", International Journal of Engineering Research and Applications (IJERA), Mar-Apr 2012.
13. H. P. Sinha, Nidhi Syal, "Design of Fault Tolerant Reversible Multiplier", International Journal of Soft Computing and Engineering (IJSCE), January 2012.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.