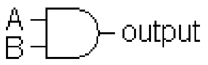
Chapter 1 : Digital Logic Circuits

Logic Gates

Binary logic deals with binary variables and with operations that assume a meaning. It is used to describe in algebraic or tabular form, the manipulation and processing of binary information. The manipulation of binary information is done by logic circuits called gates. Gates are blocks of hardware that produce singnals of binary 1 or 0 when input logic requirements are satisfied. A variety of logic gates are commonly used in digital computer system. Each gate has sa distinct graphic symbol and its operation can be described by means of an algebraic expression. The input – output relationship of the binary variables for each gate can be represented in tabular form by a truth table.

AND Gate

A simple AND gate has two inputs, A and B, and one output, Y. All inputs to an AND gate must be 1 for the output to be 1. If both Aand B are 1 then the output is 1. Otherwise, the output is 0.



A AND B = A•B = AB = A∧B = A∩B

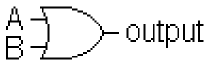
Output is TRUE only if both inputs are TRUE.

A	B	output
F	F	F
F	T	F
T	F	F
T	T	T

A	B	output
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

A basic OR gate has two inputs, A and B, and an output Y. The output is 1 if either A or B, or both, are 1, and 0 only when both A and B are 0.



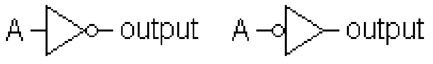
A OR B = A + B = A∨B = A∪B

Output is TRUE if either input (or both) is TRUE.

A	B	output
F	F	F
F	T	T
T	F	T
T	T	T

A	B	output
0	0	0
0	1	1
1	0	1
1	1	1

NOT or Inverter Gate
It produces the NOT, or complement function. If input is 0 output is 1, if input is 1 output is 0.

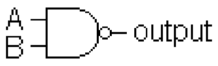


NOT A = $\overline{A}$ = A' = -A = *A = /A = A* = A/

A	output
F	T
T	F

A	output
0	1
1	0

NAND Gate
The NAND gate is derived from the abbreviation of NOT-AND. The NAND function is the complement of the AND function, is indicated by the symbol, which consists of an AND gate followed by small circle. Out put of NAND GATE is inverse of AND GATE.

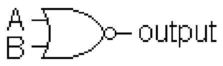


A NAND B = NOT A AND B = $\overline{(A \cdot B)}$
Output is FALSE only if both inputs are TRUE.

A	B	output
F	F	T
F	T	T
T	F	T
T	T	F

A	B	output
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate
The NOR gate is derived from the abbreviation of NOT-OR. The NOR function is the complement of the OR function, is indicated by the symbol, which consists of an OR gate followed by small circle. Out put of NOR GATE is inverse of OR GATE.



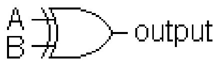
A NOR B = $\overline{(A + B)}$

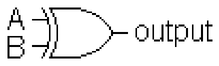
Output is FALSE if either input (or both) is TRUE.

A	B	output
F	F	T
F	T	F
T	F	F
T	T	F

A	B	output
0	0	1
0	1	0
1	0	0
1	1	0

XOR GATE
The exclusive –OR gate has a graphic symbol similar to the OR gate except for ath additional curved lined on the input side. The output of this gate is 1 if any input is 1 but excludes the combination when both inputs are 1.





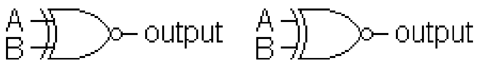
A XOR B = $\overline{A}\bullet B + A\bullet\overline{B} = (A + B)\bullet(\overline{A} + \overline{B}) = A\oplus B$
Output is TRUE if either input (but not both) is TRUE.

A	B	output
F	F	F
F	T	T
T	F	T
T	T	F

A	B	output
0	0	0
0	1	1
1	0	1
1	1	0

XNOR GATE

The exclusive –NOR is the complement of the exclusive –OR, as indicated by the small circle in the graphic symbol of XOR. The output of this gate is 1 only if both inputs are equal to 1 or 0.



A XNOR B = A•B + $\overline{A}$•$\overline{B}$ = $\overline{A \oplus B}$

Output is FALSE if either input (but not both) is TRUE.

A	B	output
F	F	T
F	T	F
T	F	F
T	T	T

A	B	output
0	0	1
0	1	0
1	0	0
1	1	1

Boolean Algebra

Opening up a computer, a terminal, or practically any other "computerized" item reveals boards containing little black rectangles. These little black rectangles are the integrated circuit (IC) chips that perform the logic of the computer. Each IC can be represented by a Boolean Algebra equation, and vice versa. The representation that is used depends on the context. Boolean Algebra provides a convenient representation and notation for simplifying and solving equations. Digital Electronics provides a layout that can then be implemented with IC chips.

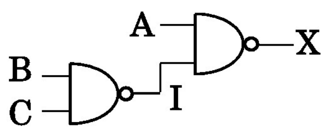
The operators used in these categories are listed in the description of the Digital Electronics category. The logic gates are usually used in Digital Electronics questions; the algebraic equations, symbols and truth tables, in Boolean Algebra. Of course, it is crucial to be able to translate between a digital electronics circuit and its Boolean Algebra notation. The order of operator precedence is NOT; AND and NAND; XOR and EQUIV; OR and NOR. Binary operators with the same level of precedence are evaluated from left to right.

Boolean Algebra Identities

1. $A+B=B+A$ $\quad A*B=B*A$ (Communicative Property)
2. $A+(B+C)=(A+B)+C$ $\quad A*(B*C)=(A*B)*C$ (Associative Property)
3. $A*(B+C)=A*B+A*C$ (Distributive Property)
4. $\overline{A+B}=\overline{A}*\overline{B}$ (DeMorgan's Law)
5. $\overline{A*B}=\overline{A}+\overline{B}$ (DeMorgan's Law)
6. $A+0=A$ $\quad A*0=0$
7. $A+1=1$ $\quad A*1=A$
8. $A+\overline{A}=1$ $\quad A*\overline{A}=0$
9. $A+A=A$ $\quad A*A=A$
10. $\overline{\overline{A}}=A$
11. $A+\overline{A}*B=A+B$
12. $(A+B)*(A+C)=A+B*C$
13. $(A+B)*(C+D)=A*C+A*D+B*C+B*D$
14. $A*(A+B)=A$
15. $A\oplus B=A*\overline{B}+\overline{A}*B$
16. $\overline{A\oplus B}=\overline{A}\oplus B=A\oplus\overline{B}$

Sample Programs

We can build circuits of any complexity out of these simple building blocks. We can look at their function in two different ways. Firstly we can write down a boolean equation by tracing through the circuit, and then try to simplify it. Secondly we can calculate a truth table which completely specifies its behaviour. This is most easily done by calculating the intermediate nodes. Consider a circuit made up by cascading two NAND gates.

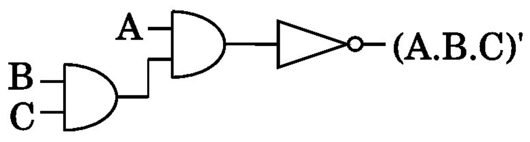


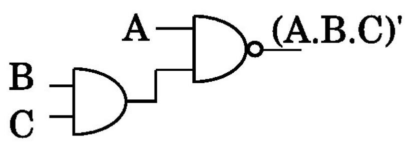
We mark the output as X and the intermediate wire as I. We can use Boolean algebra to write:
$I = (B \cdot C)$
$X = (A \cdot I)0 = (A \cdot (B \cdot C)\text{‘})\text{‘}$
$X = A0 + B \cdot C$ (by de Morgan's theorem)

We can build a truth table by systematically working through the circuit. We calculate I first, and then use it to find X.

A	B	I = (B · C)0	X = (A · I)0
0	0	1	1
	0	1	1
0	0	1	1
	1	0	1
0	1	1	0
	0	1	0
0	1	1	0
	1	0	1

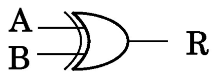
As another exercise we can consider how to build a three input NAND gate. In Boolean algebra we write this as $X = (A \cdot B \cdot C)0 = (A \cdot (B \cdot C))0)$. Notice that we are using the associative rule of Boolean algebra to build a three input AND gate from two two input AND gates. The final circuit can be simplified by using a NAND gate to replace the AND and inverter combination.





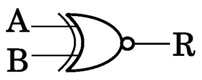
We introduce two very commonly used gates which have their own individual symbols in Boolean circuit design.

Exclusive Or (XOR)



R = A.B' + A'.B

Exclusive Nor (XNOR)



R = A'.B' + A.B

From the boolean equations we see that it is possible to build these out of other gates, but in practice they are so useful that they deserve to be considered Boolean functions in their own right. The truth tables are as follows:

A	XOR
0	0
0	1
1	1
1	0

A	B	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

We have now seen one one input gate and six two-input gates, but are these the only useful ones? One may ask the question “how many different gates one can build?” and “what do they all do?” We can answer the first question by noting that with one input, there are two possible outputs, and therefore 22 = 4 different gates. With two inputs, there are four possible outputs, and therefore 24 = 16 different gates. If we consider the casef single input circuits, we can enumerate the four possibilities:

A	G0	G1	G2	G3
0	0	0	1	1
1	0	1	0	1

We see that G2 is the inverter gate, and is the only useful one of the four. G0 always produces the output 0, G1 simply returns the input value of A, and G3 always produces the output 1. The function of each gate can be summarised by the following table:

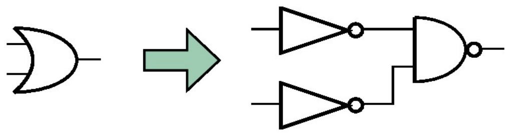
	G0	G1	G2	G3
R =	0	A	A’	1

By a similar method we can enumerate all possible two input gates and their Boolean functions:

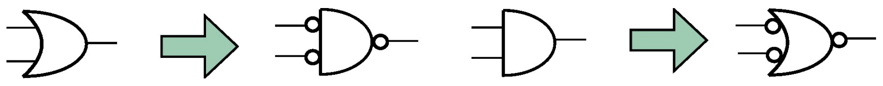
AB	G0	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	G13	G14	G15
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
R =	0	AND		A		B	XOR	OR	NOR	XNOR	B’		A’		NAND	1

Again we see that the six gates we have already described are the ones that provide useful functions of the two inputs. We may find a use for the others in specific design problems, but they are not sufficiently useful to merita special symbol or function name.
Graphical manipulation of circuit diagrams can often be done to simplify a design. Consider **DeMorgan’s** theorem which we can use to write the equation: A + B = (A’ · B’)’. This

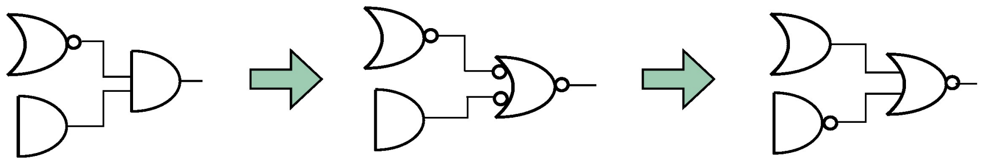
could be the basis of a circuit manipulation that we can make simply by looking at the circuit diagram:



Often it is convenient to use just the circle for inversion rather than the whole inverter gate symbol. Doing this we represent de Morgan's theorem graphically as follows:



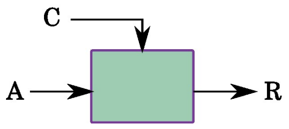
And we can now perform the same manipulations that we did using Boolean algebra directly to circuits as we design them:



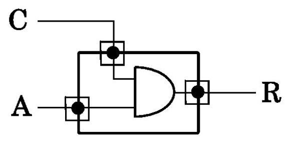
Apply de Morgan's theorem

Re-group and cancel the inverters

To finish up this lecture we introduce a new way of looking at Boolean circuits. So far we have treated gates as implementing Boolean functions of their input variables, and have not considered the possibility that the inputs may be of different types. We now introduce the idea that Boolean variables in circuits may belong to one of two types: data and control. Control variables are used to determine the flow of data. Here is a simple example. The block diagram shows a circuit with one control variable, one data variable and one output.



We now specify the action of the circuit informally as: If the control variable is 1 then the output is the same as A. If it is zero the output is zero. In other words the circuit either allows the data to pass from input to output or blocks it, depending on the control variable. The implementation of this circuit is trivial: it is just the AND gate, We now see why the name gate is used.

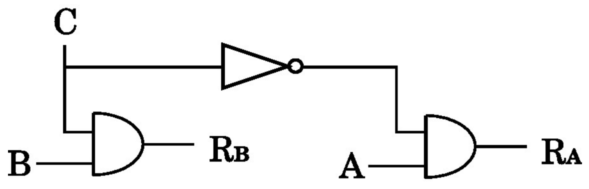


Thinking about a circuit operation in this way will prove invaluable as we begin to design more complex digital circuits.

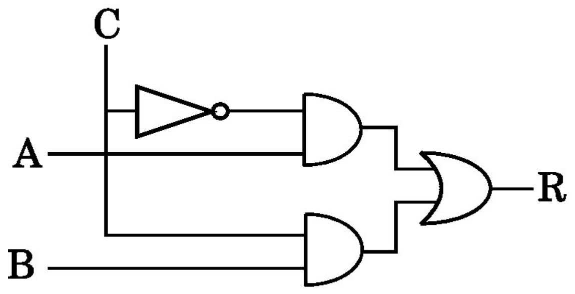
We can extend this concept to design a very useful circuit. It has two data inputs, A and B, one control in-put and one output. The specification is: "If the control input is zero then the output is the same as A. If it is 1, then the output is the same as B. In other words it is a switch that choses between data line A and data line B. Starting with this specification we can express the circuit's function as a truth table:

C	A	B	R
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

We will introduce a systematic method for designing circuits from truth tables next lecture, but for the moment we will do the design by reasoning. We have seen that the AND gate can be used to block a data path, so we can apply this idea to the data inputs A and B, but invert the control input so that when C is 0 A is unblocked but B is blocked and vice versa. We can do this with just two AND gates and one inverter.



All we need to do is to combine the two outputs into one. We note that in Boolean algebra A + 0 = A, and so all we need to do is to combine RA and RB with an OR gate.



Combinatorial Circuits and its Minimisation

This lecture introduces a methodology for designing combinatorial digital circuits. These are circuits whose outputs are defined as functions of their inputs only, and in particular not in terms of their outputs. That is to say the circuits are defined by a set of equations of the form:

R1 = f1 (A, B, C, D)R2 = f2 (A, B, C, D)R3 = f3(A, B, C, D) where R1 , R2 and R3 are outputs, A, B, C and D are inputs (not equal to R1, R2 or R3) and f1 , f2 and f3 are Boolean functions. Circuits where the outputs can appear on the right hand side of the equation as well as the left are called sequential circuits, and we will deal with them later in the course.

There are two main criteria for judging the quality of any digital circuit that is designed. Firstly it should be fast and secondly it should be small. We will now discuss a design methodology for digital circuits that aims to make them as small as possible. We have already seen that circuits can be represented by Boolean equations, and their behaviour can be described by a truth table. Boolean equations are a good representation for formalising what a circuit should do. It is always possible to generate a truth table from a set of combinatorial Boolean equations, and we will use the truth table as the starting point for a design,

As a preliminary we need to define minterms and maxterms. Suppose a circuit has n inputs designated A1, A2 , ..., An . A minterm is simply a Boolean product term in which for each input, say Ak, either Ak or its complement A0 appears exactly once. For example, if we have a three input circuit with inputs A, B and C then: A · B' · C ' and A · B · C ' are both minterms, but A · B' and B' · C ' are not minterms, since they do not contain all the input variables.A maxterm is the similarly defined as a Boolean sum term in which for each input, say Ak , either Ak or

its complement A'appears exactly once. If again we have a three input circuit with inputs A, B and C then: (A + B' + C ') and (A + B + C ') are both maxterms, but (A + B') and (B' + C ') are not.

Any combinational circuit can be expressed in one of two canonical forms. (A canonical form is a standard and agreed upon way of writing a mathematical equation.) The minterm canonical form is a set of Boolean equations, one for each output, in which each equation is a Boolean disjunction (sum) of minterms. The maxterm canonical form is a set of Boolean equations, one for each output, in which each equation is a Boolean product of maxterms. As a simple example consider the three input majority circuit which has output 1 when two or more of its inputs are 1. Let the inputs be A, B, and C and the output be R. The truth table is easily seen to be:

A	B	C	R	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	Minterm A' · B · C
1	0	0	0	
1	0	1	1	Minterm A · B' · C
1	1	0	1	Minterm A · B · C
1	1	1	1	Minterm A · B · C

For each input state where R = 1 a corresponding minterm is shown. For that given input only it will evaluate to 1. The sum of all these minterms will clearly produce the correct output, and the canonical equation is:

R = A' · B · C + A · B' · C + A · B · C ' + A · B · C

The canonical maxterm form can be found by determining the max terms that define all the zeros in the output, and then multiplying them together. If none of the maxterms evaluates to zero then the result will be 1.

$R = (A + B + C) \cdot (A + B + C') \cdot (A + B' + C) \cdot (A' + B + C)$

For most of the course we will use the minterm canonical form as the starting point for our design methodlogy.
It is also known as the sum-of-products or the disjunction-of-conjunctions form. In practice we can think of it as a more formal specification of a truth table. However, we can derive the canonical form directly from equations, or a circuit, without drawing up a truth table. For example consider the following badly designed circuit:

Working backwards from the output R we can determine its equation which is:

$R = A' \cdot (B' + C \cdot (A + B))$

For this, as for any equation we can multiply out the brackets. This yields:

$R = A' \cdot B' + A \cdot A' \cdot C + A' \cdot B \cdot C$

and since $A \cdot A'$ is always 0 this simplifies to:

$R = A' \cdot B' + A' \cdot B \cdot C$

We have one minterm $A' \cdot B \cdot C$, but the first term does not contain input C . We solve this problem using augmentation. Since, for any Boolean variable (C+C') = 1, we know that:

$A' \cdot B' = A' \cdot B' \cdot (C + C') = A' \cdot B' \cdot C + A' \cdot B' \cdot C'$

and so the canonical for is: $R = A' \cdot B' \cdot C + A' \cdot B' \cdot C' + A' \cdot B \cdot C$

Karnaugh Map

The strength (and purpose) of the canonical form is that it allows us to devise a simple algorithm for designing any circuit automatically, starting from a set of Boolean equations and finishing with an integrated circuit. We will see next lecture the structure of an integrated circuit, called a programmable logic array, which supports this design methodology. However, we also note that the canonical form is not the smallest representation of most circuits, and consequently not the best implementation according to the size criterion. We therefore now consider how to transform the canonical form into the smallest circuit that implements it. This, theoretically is done by factorisation and simplification (essentially the reverse of the augmentation process that we used in deriving the canonical form). For example if we consider the majority circuit defined by the truth table above:

$$R = A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B \cdot C' + A \cdot B \cdot C$$

we can see that we can factorise $A \cdot B$ out of the last two terms giving

$$R = A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B \cdot (C' + C) = A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B$$

This factorisation has already reduced the number of gates that we will need in the implementation. However, this is not the minimum form of the circuit, which can be derived by first expanding the expression to:

$$R = A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B \cdot C' + A \cdot B \cdot C + A \cdot B \cdot C + A \cdot B \cdot C$$

and then applying the three possible factorisations to get the minimal form:

$$R = B \cdot C + A \cdot C + A \cdot B$$

Factorisations of this kind are in general not easy to derive; one practical visual aid to find them is called the Karnaugh Map. The Karnaugh Map is simply the truth table written out as a two dimensional array. The format for two, three and four variables for the majority circuit are as follows.

		B	
		0	1
A	0	0	1
	1	1	1

		BC			
		00	01	11	10
A	0	0	0	1	0
	1	0	1	1	1

		CD			
		00	01	11	10
AB	00	0	0	1	0
	01	0	1	1	1
	11	1	1	1	1
	10	0	1	1	1

Karnaugh maps for majority circuits

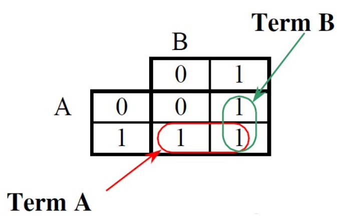
Notice that the order in which the input values are written preserves the rule that adjacent values change in only one variable. Each non-zero square represents a minterm which will appear in the canonical equation. Factorisations can be found by identifying adjacent ones in either the horizontal or the vertical directions but not the diagonal direction.

Consider the two input map first. This corresponds to the canonical equation:

X = A’ · B + A · B’ + A · B

Looking at the map we can see two possible simplifications, or factorisations of the expression which are:
X = A · (B’ + B) + A’ · B = A + A’ · B
and X = B · (A’ + A) + A · B’ = B + A · B’

In practice we would apply both of these to obtain the minimal form which is X = A + B. To do this we indicate simplifications by circling the adjacent ones as shown in Figure 1. Each circle will represent one term in the expression (NB we use the term circle loosely here to mean a closed boundary around adjacent squares on the Karnaugh map). The fact that a 1 in the Karnaugh map may appear in more than one circle is irrelevant.
In Figure 1 we see that the last row circled together is independent of B, and corresponds to the term A, and similarly the last column corresponds to the term B in the simplest form.



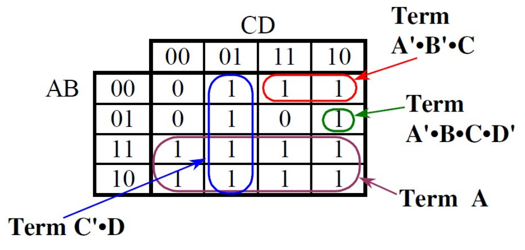
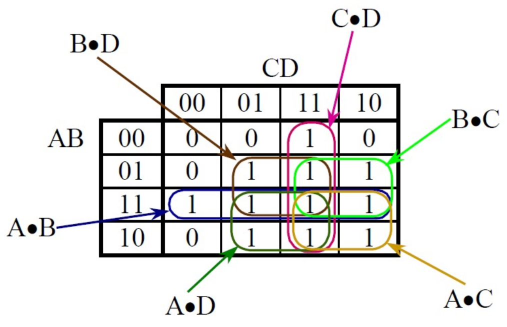


Figure 1: Karnaugh Map of the OR gate Figure 2: Karnaugh Map Terms

In general a circle on the Karnaugh map of a group of ones represents a factorisation applied to the canonical form. In the Canonical form every one on the Karnaugh map has a corresponding term in the equation. How- ever, if we can circle adjacent ones then only one term, corresponding to the area covered by the circle, need appear in the equation. The areas circled must have side lengths of 1, 2 or 4 - never 3.
Figure 2 shows a four by four Karnaugh map, and indicates some possible circles on it. To find the term that corresponds to a circle we find the inputs that do not vary in that circle. The circle covering the second column is defined by C = 0, D = 1. The inputs A and B can be either 1 or 0 within the circle. To make sure that the output is a 1 at all four input values within hat circle we simply need ensure that C ‘ · D = 1. Notice that we always try to find the largest possible circle since this will correspond to

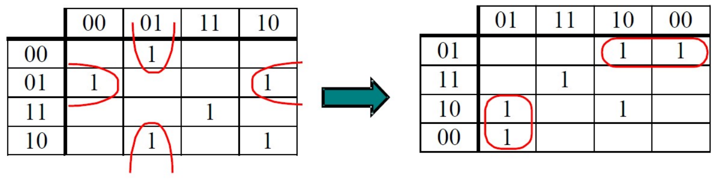
the greatest simplification. In Figure 2 we could find a greater simplification by circling together all the ones in the fourth column rather than
the single one corresponding to input 0110.
We can now consider the four input majority voter again.



We can see that there are many possible factorisations which can be applied. Consider in particular the square block of four ones in the bottom right hand corner. The top pair corresponds to the minterms
A · B · C · D +A · B · C · D’
which simplifies to A · B · C , and the bottom pair corresponds to
A · B’ · C · D + A · B’ · C · D’,
which simplifies to A · B’ · C . There is a further factorisation of these two simplified terms to A · C , and this will always be the case for any square block of four, or any row or column of the Karnaugh Map where all the entries are ones. Clearly the bigger the block that we can mark the fewer the variables in each term, and the simpler the expression. Hence we can derive the simplest expression using the six groups of four ones shown above. Inspection tells us which variables appear in the terms. These are just the ones that are always constant
in the circled block. From the Karnaugh map we read the simplified expression for the four input majority voter as:

X = A · B + A · C + A · D + B · C + B · D + C · D

The Karnaugh map must be treated as cyclic, so that the last row and column should be considered to be adjacent to the first. We can make this explicit by moving the top row to the bottom, and then the first column to the right hand side end. Consider the four input map shown in Figure 3. Without cycling, there are no apparent simplifications, but when the cycle is made explicit the two possible simplifications become clear.



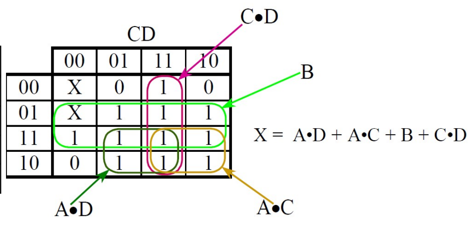


Figure 3: Cyclic Karnaugh Map   Figure 4: Karnaugh Map with don't cares

In many applications we may know that some possible input states may never occur, and therefore we do not care about the outputs of the circuit. We can make these explicit in the truth table by placing a cross in the output column rather than a zero or one. This has the advantage in minimisation problems that the corresponding points in the Karnaugh map may be treated as either a 1 or a 0 for the purpose of simplification. For example, considering again the four input majority circuit, if we happen to know the input states 0000 and 0100 never occur we can treat them as "don't cares" in the Karnaugh map as shown in Figure 4. Clearly it is advantageous to treat 0100 as a 1 then we can obtain a major simplification with the central block of eight which corresponds to the term B in the second row.

We can apply the principal of duality to all of the preceding material by designing with maxterms. The characteristic of the dual canonical form is that if any maxterms is zero the output is zero. The maxterms are therefore derived from the zeros in the truth table. Karnaugh maps can again be used, but this time it is adjacent zeros that represent possible simplifications. These simplifications follow rules of the form:

$(A' + B) \cdot (A + B) = B$

When circling zeros, don't cares can be applied in the same way, but remembering that simplifications will arise by treating them as zeros not ones. Lastly, it should be noted that the choice of canonical form will often lead to a dramatic simplification. A trivial, but telling example is the design of the OR function.

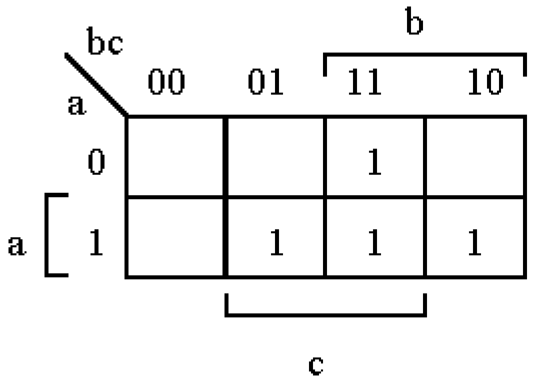
A	B	C	
0	0	0	Maxterm
0	1	1	Minterm
1	0	1	Minterm
1	1	1	Minterm

The minterm expression which we noted above is $X = A \cdot B + A \cdot B + A \cdot B$ whereas the maxterm expression is $X = (A + B)$.

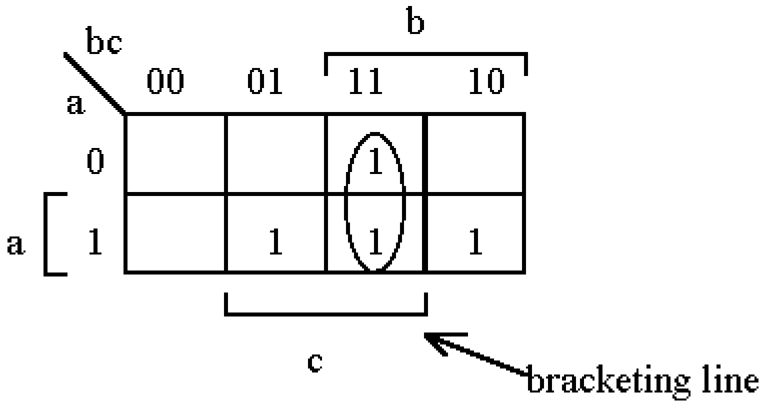
Minimization of Boolean expressions using Karnaugh maps.
Given the following truth table for the function m:

abc	m
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

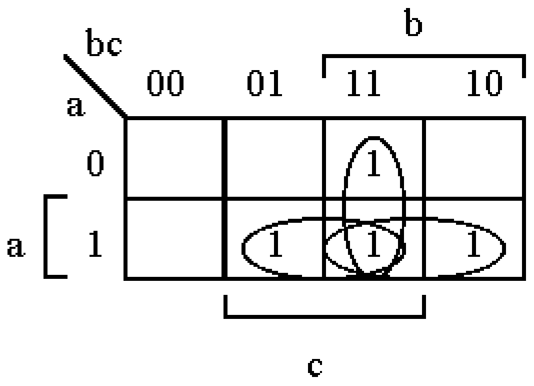
The Boolean algebraic expression is
m = a'bc + ab'c + abc' + abc.
We have seen that the minimization is done as follows.
m = a'bc + abc + ab'c + abc + abc' + abc
= (a' + a)bc + a(b' + b)c + ab(c' + c) $[NB : a' = \bar{a}]$
= bc + ac + ab
The abc term was replicated and combined with the other terms.
To use a Karnaugh map we draw the following map which has a position (square) corresponding to each of the 8 possible combinations of the 3 Boolean variables. The upper left position corresponds to the 000 row of the truth table, the lower right position corresponds to 110. Each square has two coordinates, the vertical coordinate corresponds to the value of variable a and the horizontal corresponds to the values of b and c.



The 1s are in the same places as they were in the original truth table. The 1 in the first row is at position 011 (a = 0, b = 1, c = 1). The vertical coordinate, variable a, has the value 0. The horizontal coordinates, the variables b and c, have the values 1 and 1.
The minimization is done by drawing circles around sets of adjacent 1s. Adjacency is horizontal, vertical, or both. The circles must always contain 2n 1s where n is an integer.



We have circled two 1s. The fact that the circle spans the two possible values of a (0 and 1) means that the a term is eliminated from the Boolean expression corresponding to this circle. The bracketing lines shown above correspond to the positions on the map for which the given variable has the value 1. The bracket delimits the set of squares for which the variable has the value 1. We see that the two circled 1s are at the intersection of sets b and c, this means that the Boolean expression for this set is bc.



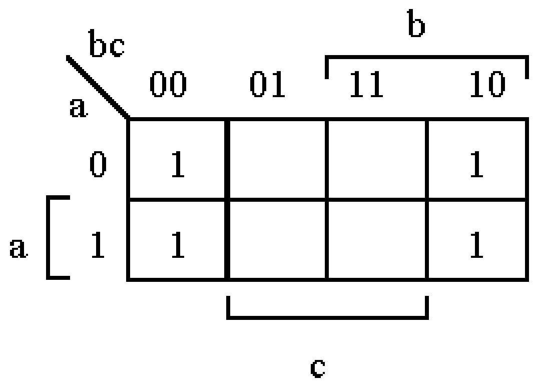
Now we have drawn circles around all the 1s. The left bottom circle is the term ac. Note that the circle spans the two possible values of b, thus eliminating the b term. Another way to think of it is that the set of squares in the circle contains the same squares as the set a intersected with the set c. The other circle (lower right) corresponds to the term ab. Thus the expression reduces to bc + ac + ab as we saw before.

What is happening? What does adjacency and grouping the 1s together have to do with minimization? Notice that the 1 at position 111 was used by all 3 circles. This 1 corresponds to the abc term that was replicated in the original algebraic minimization. Adjacency of 2 1s means that the terms corresponding to those 1s differ in one variable only. In one case that variable is negated and in the other it is not.

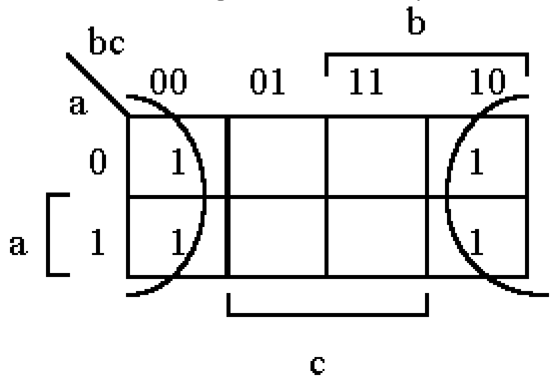
For example, in the first map above, the one with only 1 circle. The upper 1 is the term a'bc and the lower is abc. Obviously they combine to form bc (a'bc + abc = (a' + a)bc = bc). That is exactly what we got using the map.

The map is easier than algebraic minimization because we just have to recognize patterns of 1s in the map instead of using the algebraic manipulations. Adjacency also applies to the edges of the map.

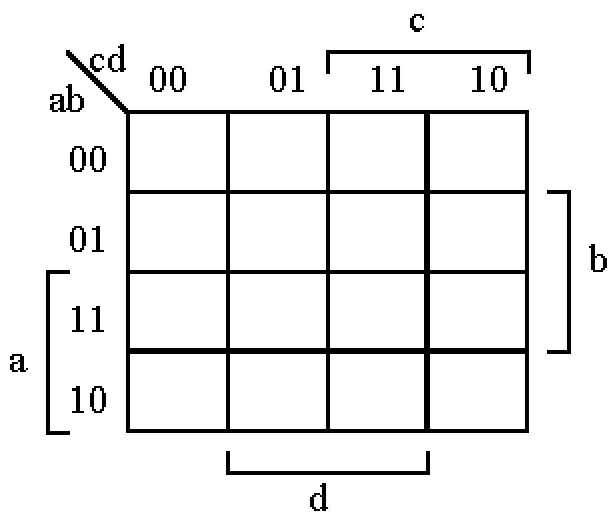
Let's try another 3 variable map.



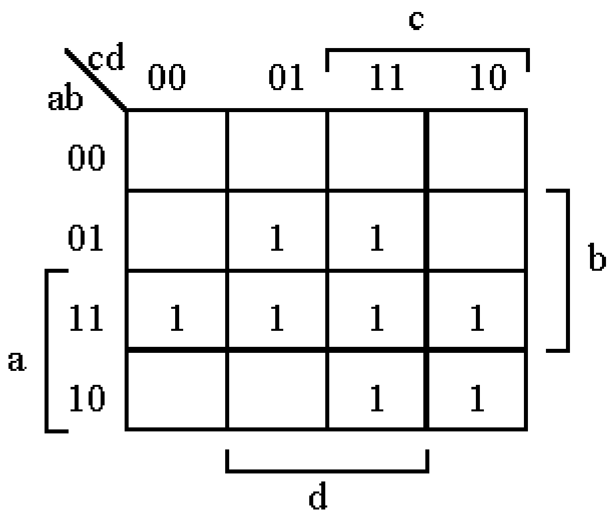
At first it may seem that we have two sets, one on the left of the map and the other on the right. Actually there is only 1 set because the left and right are adjacent as are the top and bottom. The expression for all 4 1s is c'. Notice that the 4 1s span both values of a (0 and 1) and both values of b (0 and 1). Thus, only the c value is left. The variable c is 0 for all the 1s, thus we have c'. The other way to look at it is that the 1's overlap the horizontal b line and the short vertical a line, but they all lay outside the horizontal c line, so they correspond to c'. (The horizontal c line delimits the c set. The c' set consists of all squares outside the c set. Since the circle includes all the squares in c', they are defined by c'. Again, notice that both values of a and b are spanned, thus eliminating those terms.)



Now for 4 Boolean variables. The Karnaugh map is drawn as shown below.

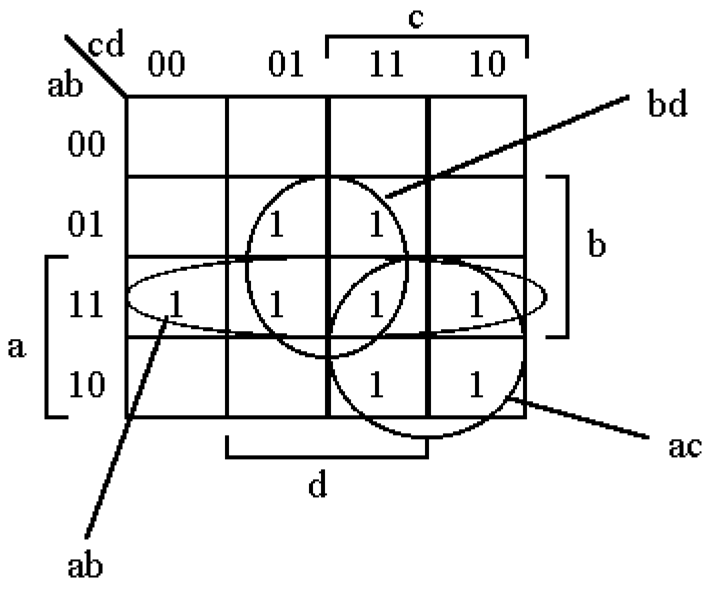


The following corresponds to the Boolean expression q = a'bc'd + a'bcd + abc'd' + abc'd + abcd + abcd' + ab'cd + ab'cd'

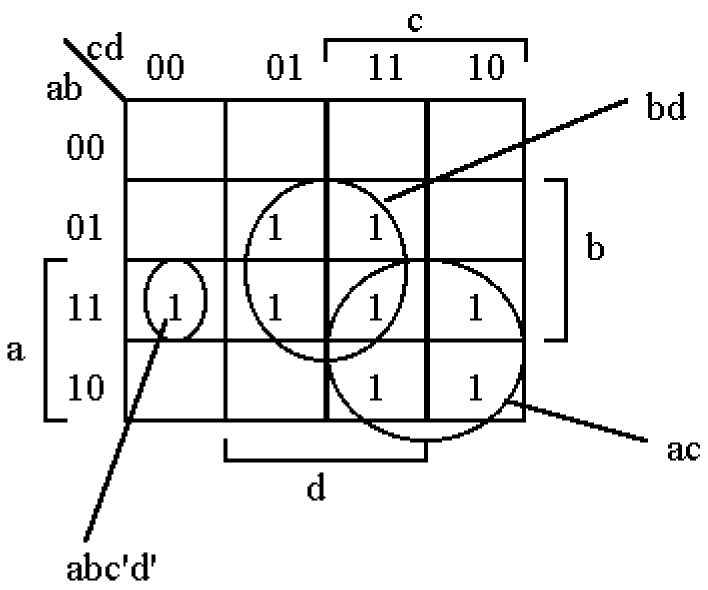


RULE: Minimization is achieved by drawing the smallest possible number of circles, each containing the largest possible number of 1s.

Grouping the 1s together results in the following.



The expression for the groupings above is
q = bd + ac + ab
This expression requires 3 2-input and gates and 1 3-input or gate.
We could have accounted for all the 1s in the map as shown below, but that results in a more complex expression requiring a more complex gate.



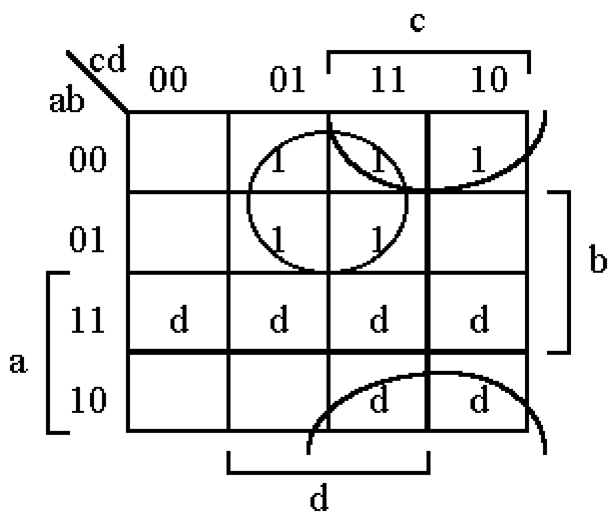
The expression for the above is bd + ac + abc'd'. This requires 2 2-input and gates, a 4-input and gate, and a 3 input or gate. Thus, one of the and gates is more complex (has two additional inputs) than required above. Two inverters are also needed.

Don't Cares

Sometimes we do not care whether a 1 or 0 occurs for a certain set of inputs. It may be that those inputs will never occur so it makes no difference what the output is. For example, we might have a bcd (binary coded decimal) code which consists of 4 bits to encode the digits 0 (0000) through 9 (1001). The remaining codes (1010 through 1111) are not used. If we had a truth table for the prime numbers 0 through 9, it would be

abcd	p
0000	0
0001	1
0010	1
0011	1
0100	0
0101	1
0110	0
0111	1
1000	0
1001	0
1010	d
1011	d
1100	d
1101	d
1110	d
1111	d

The ds in the above stand for "don't care", we don't care whether a 1 or 0 is the value for that combination of inputs because (in this case) the inputs will never occur.



The circle made entirely of 1s corresponds to the expression a'd and the combined 1 and d circle (actually a combination of arcs) is b'c. Thus, if the disallowed input 1011 did occur, the output would be 1 but if the disallowed input 1100 occurs, its output would be 0. The minimized expression is
p = a'd + b'c
Notice that if we had ignored the ds and only made a circle around the 2 1s, the resulting expression would have been more complex, a'b'c instead of b'c.
The term ”combinational” comes to us from mathematics. In mathematics a combination is an unordered set, which is a formal way to say that nobody cares which order the items came in. Most games work this way, if you rolled dice one at a time and get a 2 followed by a 3 it is the same as if you had rolled a 3 followed by a 2. With combinational logic, the circuit produces the same output regardless of the order the inputs are changed.
There are circuits which depend on the when the inputs change, these circuits are called sequential logic. Even though you will not find the term ”sequential logic” in the chapter titles, the next several chapters will discuss sequential logic.
Practical circuits will have a mix of combinational and sequential logic, with sequential logic making sure everything happens in order and combinational logic performing functions like arithmetic, logic, or conversion.

Combinational Circuit

Flip Flops

The circuit of Figure 5, which we introduced in the last lecture, is called a flip-flop or bi-stable. If we try to write down its truth table, we immediately find that we have a problem. We can construct the table for the three values of SR (00, 01 and 10) where at least one input is zero. This is because if any input to a NAND gate is zero, the output will be one regardless of the other input, and we can therefore work around the loop to calculate all the values. However when R and S are both 1 we cannot immediately calculate P and Q. The only way we can analyse what happens is to look at the possible values that P and Q could hold at the time when the inputs became 11. Thus we need to expand the truth table to include P_{t-1} and Q_{t-1} where the subscript indicates the time at which the value was present.

S	R	P_{t-}	Q_{t-}	P_t	Q_t	
1	1	0	0	1	1	Unstable
1	1	0	1	0	1	Stable
1	1	1	0	1	0	Stable
1	1	1	1	0	0	Unstable

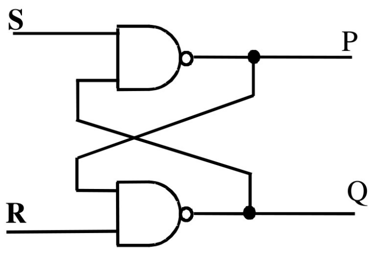


Figure 5: The Set-Reset Flip Flop

Here we see that there are two states where $P_t = P_{t-1}$ and $Q_t = Q_{t-1}$ (1101 and 1110 respectively). These are therefore stable states. The other two states (1100 and 1111) are unstable, and using the simple model with time delay t_d will oscillate with a period of $2t_d$. In practice, the circuit will fall into one of the two stable states rather than oscillate. This is because the time delays of the two NAND gates will not be precisely the same, and depending on which changes fastest the circuit will fall into one of the two stable states. Which state if will finish in depends on the manufacturing process. In practice we are not interested in the unstable states, only in the stable ones.

This circuit can be considered to be a one bit memory circuit since Q can be set to one or zero. To see this we need to look at a sequence of inputs as shown in Figure 6. At the third time step we have the input 10 which puts the circuit into a known state and the output Q to 1. That value of Q is memorised and remains as long as the input is kept at 11. At the sixth time step the input 01 forces the output Q to be a 0, and as long as the input is held at 11 this 0 remains. This way of looking at the circuit gives rise to the names of the inputs S for Set and R for Reset, and so this flip flop is often given the name R-S. The following three points should be noted. We are

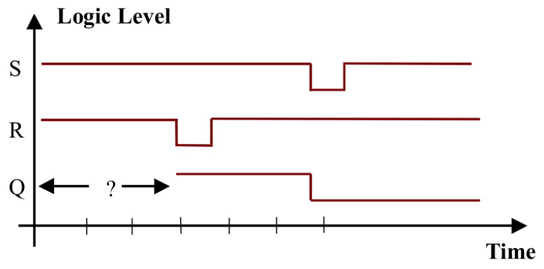


Figure 6: The Set-Reset Flip Flop Timing

now describing the behaviour by means of a sequence of inputs, and for this reason, these circuits are referred to a sequential. Secondly, in all the cases of interest for this circuit P = Q'. Thirdly, an R-S flip flop can equivalently be built out of NOR gates.
The R-S flip flop is a building block from which other circuits can be constructed. The first refinement that we need to make is some form of gate onto the R and S inputs, so that we can control the times at which the set or reset inputs are presented. This circuit is shown in Figure 7. When the latch is 1, we have that S = D and R = D' , and the value of Q will therefore be set to the value of D. When the L (latch) input is zero, the values of S and R are both 1, in other words the R-S flip flop will hold its previous value. This circuit now forms a practical controllable memory. The L input can be thought of as a write line in that whenever it is set to 1 the value of D is written to the flip flop i.e. the output Q = D. When L = 0 the value of Q is held constant no matter what the value of D takes. A truth table for this circuit can be written using the previous subscript notation, and the common symbol for this D-type flip flop is shown in Figure 8.
Notice that the flip flop is now nicely encapsulated in that it is not possible for the user to evoke any unstable

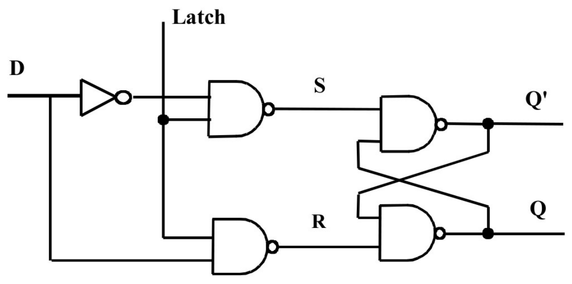


Figure 7: The D-type Latch

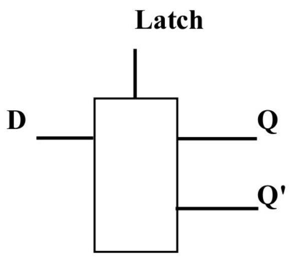


Figure 8: D-Type Latch Symbol

states. The only period that the circuit could be undetermined is from the time that the power is switched on until the latch input is set to 1. In practical designs we would need to ensure that the latch goes to 1 momentarilyas the circuit is switched on to ensure that the behaviour is correct.
The simple D type flip flop does however have drawbacks which will be seen if we consider a timing diagram shown in Figure 9. Here the value of D to be stored arrives a little later than the clock pulse, and in the case where the previous value of Q is the same as the new value, a spike (momentary wrong state) is caused which may make the circuit malfunction. In order to avoid this happening we need to build a circuit that will change its output only at one instance of time. Clearly, in Figure 9, if we could arrange the circuit so that the output Q is set only when the latch C changes from one to zero, then we would have a reliable circuit. Such devices are termed edge triggered.

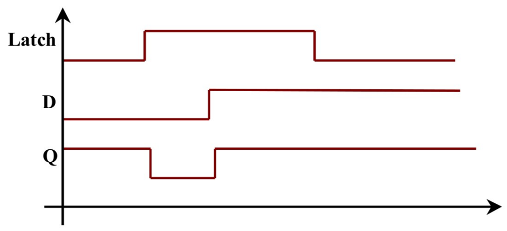


Figure 9: A Spike in the D-Type Latch flop

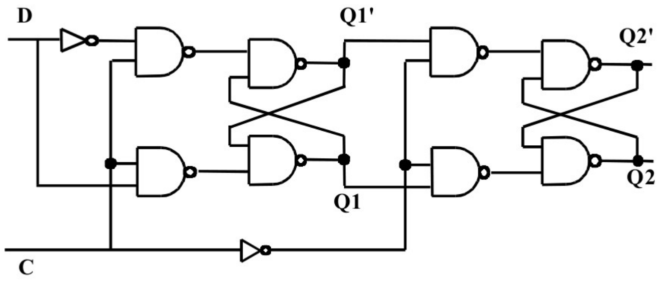
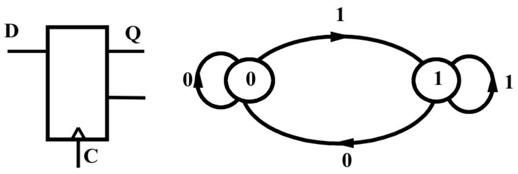
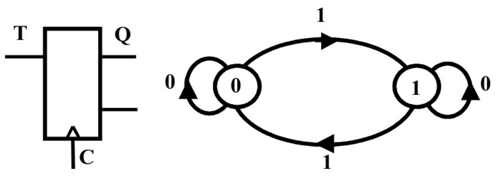


Figure 10: The master-slave D-Type flip

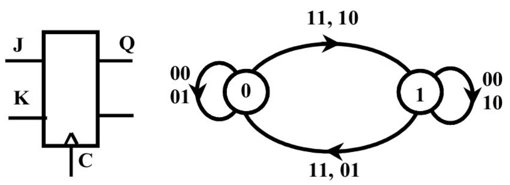
The first type of edge triggered flip flop we will look at is called the master-slave. It is built from two gatedR-S flip flops as shown in Figure 10. The trick is that the gates of the two flip flops can never be open at the same time. Thus, if the clock is at 1, the output of the first flip flop, Q1 follows the input D, but the gate of the second stage is closed, and the output Q2 cannot change state. When the clock goes from 1 to 0, the first flip flop is now blocked and the value of Q1 is held at the value of D when the transition occurred, but the gate to the second R-S flip flop is now open, and the state of Q1 is transmitted to Q2. It is easy to see that this output state cannot change until the clock goes first to 1 and then to zero again. It should be noted that there still remains the possibility of a spike being created with this circuit if the gate of the second stage opens momentarily before the gate of the first stage closes. However, the implementation in Figure 10 is quite safe since the invertor ensures that the signal will change on stage 1 before it changes on stage 2. The master slave arrangement is convenient to use and easy to understand, however, a smaller arrangement of the edge triggered D type flip flop can be made, as we will see later.
The edge triggering is important in that it allows us to build circuits where state transitions occur at a set time. This is important in that it allows us to build synchronous circuits, i.e. those where we ensure that all signals change at the same time. It is useful to think of synchronous sequential circuits as having a finite state machine representation. For example, if we take the edge triggered D-type flip flop, there are two states which are defined by the value of the output Q. The finite state machine is shown in Figure 11a. Here the states are shown in the circle, and the possible inputs (D) are shown on the arcs. Transitions can only occur when a negative edge appears on the clock signal, but this is not shown in the finite state machine model. In practical circuits, the same clock signal will be fed to all the flip flop clock inputs to ensure synchronous behaviour.



(a) D-Type Flip Flop



(b) T-Type Flip Flop



(c) J-K Flip Flop

Figure 11: Different Flip Flops and their Symbols

Flip flops can be made with other characteristics, and one important one is illustrated by the T-type, or toggle, flip flop, shown in Figure 11b. Here the circuit is best considered to work in two modes. If T = 1 then when the clock changes from 1 to zero, the output will change its state, but if T is set to 0 the output will not change. This may be represented functionally by the equation:

$$Q_t = T\,Q'_{t-1} + T'\,Q_{t-1}$$

The most flexible flip flop is the JK device shown in Figure 11c. This can be set up to have the characteristics of either a SR or a toggle flip flop. The four modes of operation are defined by the values of J and K as follows as shown in the diagram. The transitions are only made when a negative edge appears on the clock. By connecting J and K together and using them as one input we create a T-type flip flop. By inverting J and connecting it to K we create a D -type flip flop (with the J terminal equivalent to D) and in cases where we want a simple R-S flip flop we use J as S and K as R.
Figure 12 shows a very ingenious design for the edge triggered filp flop. This circuit actually latches whenthe clock changes from 0 to 1, rather than 1 to 0 in our previous circuits. Both types of edge triggering are readily available in practical circuits. Note that when the clock is 0 in Figure 12, the input to the final R-S flip flop is forced to be 1 1 and the output will remain unchanged. To understand the circuit further it is necessaryto trace the effect of different inputs around the circuit.

To finish up we will look at one further feature required in flip flops, namely the ability to preset a particular output, regardless of the clock. Such inputs are referred to as CLEAR which sets the output to zero, and PRESET which sets it to 1. Such inputs are used to set a circuit into a known state, for example when it is switched on or perhaps when a reset button is pressed. A simple implementation of the edge-triggered D type flip flop with preset and clear is given in Figure 13. Notice that if the PRESET and CLEAR inputs are 1, the circuit behaves normally. If one of them goes to zero the output will be forced to the corresponding state. Notice also that if both are set to zero the output and its complement are both forced to 1. This is of course an unwanted effect, and should be eliminated by the external circuitry. You should satisfy yourself that with PRESET and CLEAR held at 1 the circuit behaves as a D type.

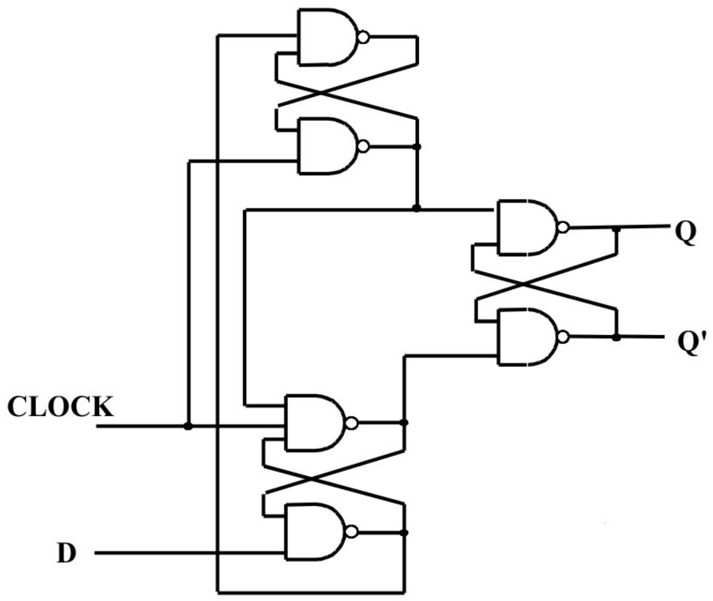


Figure 12: Edge Triggered Flip Flop

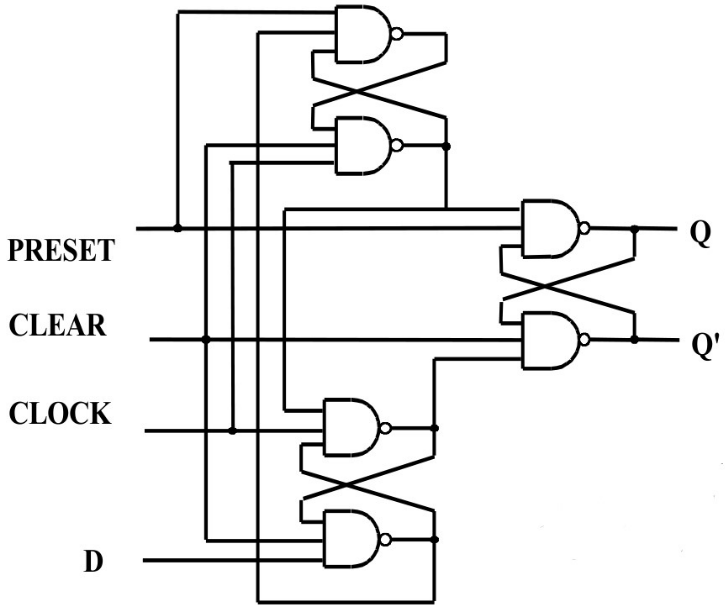
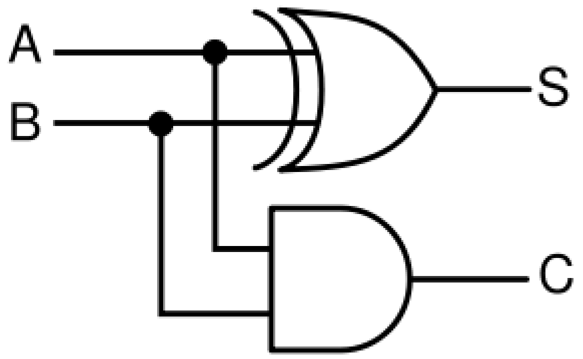


Figure 13: Edge Triggered Flip Flop with Preset and Clear

Half adder

A half adder has two inputs, generally labeled A and B, and two outputs, the sum S and carry C. S is the two-bit XOR of A and B, and C is the AND of A and B. Essentially the output of a half adder is the sum of two one-bit numbers, with C being the most significant of these two outputs.

The second type of single bit adder is the full adder. The full adder takes into account a carry input such that multiple adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labelled Ci or Cin while the carry out is labelled Co or Cout.



Half adder circuit diagram

A half adder is a logical circuit that performs an addition operation on two binary digits. The half adder produces a sum and a carry value which are both binary digits.

$S = A \oplus B$

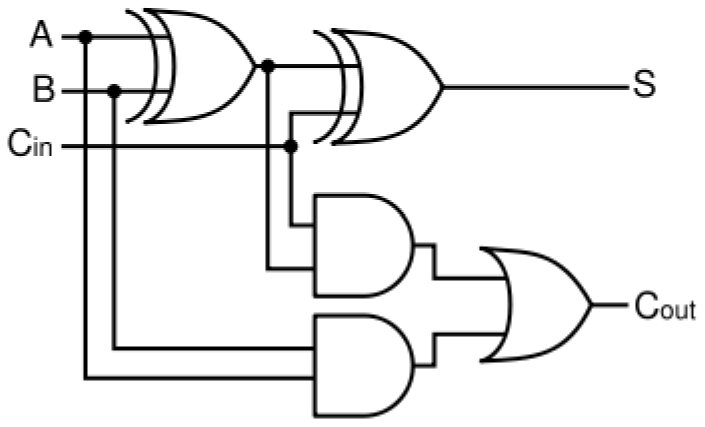
$C = A \cdot B$

Following is the logic table for a half adder:

Input Output

A	**B**	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full adder



Full adder circuit diagram
Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}

Schematic symbol for a 1-bit full adder

A full adder is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carry value, which are both binary digits. It can be combined with other full adders (see below) or work on its own.

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

$$C_o = (A \cdot B) + (C_i \cdot (A \oplus B)) = (A \cdot B) + (B \cdot C_i) + (C_i \cdot A)$$

Input Output

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

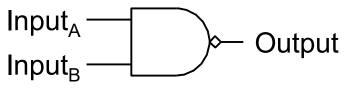
Note that the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. This is because the only discrepancy between OR and XOR gates occurs when both inputs are 1; for the adder shown here, one can check this is never possible. Using only two types of gates is convenient if one desires to implement the adder directly using common IC chips.

A full adder can be constructed from two half adders by connecting A and B to the input of one half adder, connecting the sum from that to an input to the second adder, connecting C_i to the other input and or the two carry outputs. Equivalently, S could be made the three-bit xor of A, B, and C_i and C_o could be made the three-bit majority function of A, B, and C_i. The output of the full adder is the two-bit arithmetic sum of three one-bit numbers.

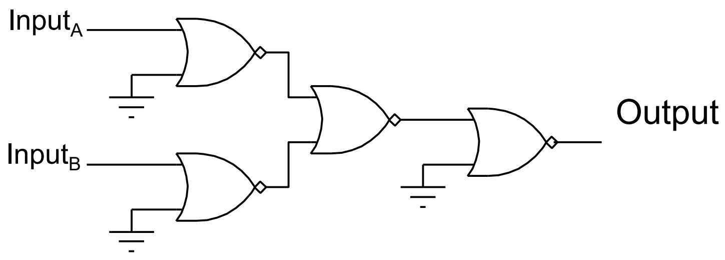
NAND function

It would be pointless to show you how to ”construct” the NAND function using a NAND gate, since there is nothing to do. To make a NOR gate perform the NAND function, we must invert all inputs to the NOR gate as well as the NOR gate’s output. For a two-input gate, this requires three more NOR gates connected as inverters.

2-input NAND gate



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0


Input$_A$
Input$_B$
Output