Boolean Algebra — Lab Manual, Chapter Two

A boolean function is an abstract mathematical representation of some computation or electronic circuit. Mathematically, a program and an electronic circuit are equivalent. In this laboratory you will explore this equivalence using *LOGICEV*, a program that evaluates logic functions and simulates an electronic circuit. You will also get the opportunity to input various equivalent logic functions and verify that they are truly equivalent.

2.1 Boolean Algebra

Boolean algebra is a mathematical system with two values (zero and one) and three operators: logical AND, logical OR, and logical NOT[1]. The boolean system is *closed* with respect to these three operations; that is, these operators always produce a boolean result given boolean operands. Logical AND and OR are *commutative*; you can switch the operands and get identical results. Logical AND and OR are *associative*; that is, A AND (B AND C) is equal to (A AND B) AND C. The same is true for logical OR. Logical AND and OR are also *distributive*; that is, A AND (B OR C) is equal to (A AND B) OR (A AND C). Similarly, A OR (B AND C) is equivalent to (A OR B) AND (A OR C). The value one is the identity element with respect to logical AND, the value zero is the identity element with respect to logical OR. For any boolean value A, there is one other value A' that is not equal to A (the inverse of A). A OR A' is one and A AND A' is zero. These statements form the basic *postulates* of the boolean algebra system. We can prove all other theorems and facts about the boolean system using this set of postulates.

2.1 Which rule (postulate) can we use to prove A AND (B AND C) is equal to (A AND B) AND (A AND C)? ________________________________

2.2 Which rule (postulate) can we use to show that A AND (B AND C) is equal to (A AND B) AND C? ________________________________

From the postulates above, we can easily prove several important boolean algebra theorems. Some important theorems you'll commonly use include:

```
Th1:  A + A = A         Th2:  A • A = A
Th3:  A + 0 = A         Th4:  A • 1 = A
Th5:  A • 0 = 0         Th6:  A + 1 = 1
Th7:  (A+B)′ = A′B′     Th8:  (AB)′ = A′ + B′
Th9:  A + AB = A        Th10: A • (A + B) = A
Th11: A + A′B = A + B   Th12: A′ • (A+B′) = A′B′
Th13: AB + AB′ = A      Th14  (A′ + B′) • (A′ + B) = A′
```

2.3 In Chapter One you learned that you could use the logical AND operation to force various bits in a bit string to zero. Which theorem above describes this operation? ____________

2.4 In Chapter One you learned that you could use the logical OR operation to force some bits in a bit string to one. Which theorem above describes this operation? ______________

2.2 Boolean Functions and Truth Tables

A boolean function, or expression, is a set of literals combined with the logical AND and OR operators. A literal is either the value zero or one or a primed or unprimed variable. The prime denotes logical negation (NOT). Examples of typical boolean functions include:

```
F = AB + C          G = A(B + C)        H = A′B′C + ABC′
```

Given values for A, B, and C, you can compute the values of the above function. For example, if A=0, B=1, and C=1, then F = 0•1+1 = 0 + 1 = 1, G = 0•(1+1) = 0•(1) = 0, and H = 1•0•1 + 0•1•0 = 0 + 0 = 0. However, evaluating boolean

1. There are other operators, but you can synthesize them all using logical AND, OR, and NOT.

functions by hand in this fashion is tedious and error prone. Since it is likely that you will confuse some values when doing mental calculations, a better solution is to look up the answer in a table rather than compute the result manually.

Since there are only two possible boolean values, it is possible to enumerate all the values that a boolean function can produce. For example, function F above uses three independent variables A, B, and C. Given *n* variables, each having two distinct values, there are 2^n different possible combinations of values for these variables. Therefore, there are eight (2^3) possible combinations of A, B, and C as inputs to function F. Therefore, it is easy to create a table listing all possible inputs to a given function (or functions) and supply the function results in that table. Such a table is a *truth table*. To construct a truth table, begin by listing out all possible combinations of the variables appearing in the function. The three functions in the current example all have three input variables. So begin by listing the eight three-bit binary values between zero and seven:

Table 7: Truth Table for F, G, and H. Step One.

C	B	A	F	G	H
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

By convention, we'll treat variable "A" as the L.O. bit and variable "C" as the H.O. bit of the binary number.

The next step is to compute the values for each function and insert them into the table. For function F, you compute the following:

```
F(C,B,A) = AB + C

F(0,0,0) = 0•0+0 = 0
F(0,0,1) = 1•0+0 = 0
F(0,1,0) = 0•1+0 = 0
F(0,1,1) = 1*1+0 = 1
F(1,0,0) = 0•0+1 = 1
F(1,0,1) = 1•0+1 = 1
F(1,1,0) = 0•1+1 = 1
F(1,1,1) = 1•1+1 = 1

G(C,B,A) = A(B + C)

G(0,0,0) = 0•(0+0) = 0
G(0,0,1) = 1•(0+0) = 0
G(0,1,0) = 0•(1+0) = 0
G(0,1,1) = 1•(1+0) = 1
G(1,0,0) = 0•(0+1) = 0
G(1,0,1) = 1•(0+1) = 1
G(1,1,0) = 0•(1+1) = 0
G(1,1,1) = 1•(1+1) = 1
```

The next step is to insert these values into the truth table at the appropriate place. For the F and G functions above, you wind up with the following entries in the truth table:

2.1 Distributive Law

Table 8: Truth Table for F, G, and H. Step Two.

C	B	A	F	G	H
0	0	0	0	0	
0	0	1	0	0	
0	1	0	0	0	
0	1	1	1	1	
1	0	0	1	0	
1	0	1	1	1	
1	1	0	1	0	
1	1	1	1	1	

2.2 Associative Law

2.5 What are the function values for H?

2.3 Th 5

Consider the function F = AB + C'D'. Since this function uses four input values, there are 2^4 (16) different combinations of inputs possible. Enumerating these values yields:

```
J(D, C, B, A) = AB + C'D'

J(0, 0, 0, 0) = 0•0 + 1•1 = 1
J(0, 0, 0, 1) = 0•1 + 1•1 = 1
J(0, 0, 1, 0) = 1•0 + 1•1 = 1
J(0, 0, 1, 1) = 1•1 + 1•1 = 1
J(0, 1, 0, 0) = 0•0 + 1•0 = 0
J(0, 1, 0, 1) = 0•1 + 1•0 = 0
J(0, 1, 1, 0) = 1•0 + 1•0 = 0
J(0, 1, 1, 1) = 1•1 + 1•0 = 1
J(1, 0, 0, 0) = 0•0 + 0•1 = 0
J(1, 0, 0, 1) = 0•1 + 0•1 = 0
J(1, 0, 1, 0) = 1•0 + 0•1 = 0
J(1, 0, 1, 1) = 1•1 + 0•1 = 1
J(1, 1, 0, 0) = 0•0 + 0•0 = 0
J(1, 1, 0, 1) = 0•1 + 0•0 = 0
J(1, 1, 1, 0) = 1•0 + 0•0 = 0
J(1, 1, 1, 1) = 1•1 + 0•0 = 1
```

2.4 Th 6

2.6 Draw the truth table for function J above:

Table 9: Truth Table for Function J

D	C	B	A	J

For complex boolean functions with lots of different terms and operations, you can write a simple program to automatically emit a truth table for you. The LOGICEV program you'll use in this lab is a good example of a program that will automatically generate a truth table directly from a logic function.

2.3 Algebraic Manipulation of Boolean Expressions

Since boolean algebra is an algebraic system, it should come as no surprise that you can manipulate boolean expressions algebraically. Using the theorems presented earlier, you can transform one boolean expression into a different, equivalent expression. There are two reasons why you would want to do this: to simplify a complex boolean expression or to transform an expression into a *canonical* form.

To demonstrate how you can use algebraic transformations to simplify an expression, the following example works *backwards*. It starts with a simple equation and makes it complex via algebraic transformations. The simplification process is this same sequence of transformations, only in reverse:

```
F     =     ab + c                        original function
      =     ab•1 + c                      by Th4
      =     ab(c + c’) + c                Inverse law (P5)
      =     abc + abc’ + c                Distributive law (P4)
      =     abc + abc’ + c + 0            by Th3
      =     abc + abc’ + c + bc•0         by Th5
      =     abc + abc’ + c + a’bca        Inverse law (P5)
      =     a(bc + bc’ + a’bc) + c        Distributive law (P4)
      =     a(bc + bc’ + a’bc) + c•1      by Th4
      =     a(bc + bc’ + a’bc) + c(b+b’)  Inverse law (P5)
      =     a(bc + bc’ + a’bc) + cb + cb’ Distributive law
```

Obviously, we can go on forever making this expression more complex with each step. The important thing to note is that you can undo this complexity by following these steps in reverse. If you have a complex expression that you can simplify, then there is *some* sequence of algebraic operations that produces the complex form from the simplified form.

2.5 0, 0, 0, 1, 1, 0, 0, 0, 0

Therefore, there is a corresponding sequence (working in reverse) that transforms the complex expression into the simplified one. Unfortunately, there is no simple algorithm that describes how to do this. It is a skill you improve via experience and a lot of trial and error.

Now let's take a complex expression and attempt to simplify it. Consider the logic equation for segment number four of a seven segment display:

$$S_4 = D'C'B'A' + D'C'BA + D'CBA' + DC'B'A'$$

When attempting to optimize an expression, the trick is to combine terms that contain a primed and unprimed version of the variable. In the expression above, we can use the distributive law to combine D'C'B'A' and DC'B'A', yielding (D+D')C'B'A'. Since the law of inverses says that D + D' is one, this reduces those two terms to the single term C'B'A'. Therefore, a simpler version of S_4 is

$$C'B'A' + D'C'BA + D'CBA'$$

We can use the distributive law on the last two terms to reduce this to

$$C'B'A' + D'B(C'A + CA')$$

This last step increased the number of terms from three to four, but reduced the total number of operations required to compute the function result. The previous version requires eight AND operations and two OR operations, the latter version requires six AND operations and two OR operations.

2.7 Provide the applicable rule for each of the following steps in the simplification of S_0:

S_0 = D'C'B'A' + D'C'BA' + D'C'BA +D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A
= D'C'B'A' + D'C'BA' + D'C'BA +D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A' + DC'B'A

____

2.8 = C'B'A'(D+D') + D'C'BA' + D'C'BA +D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A

____

2.9 = C'B'A' + D'C'BA' + D'C'BA +D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A

____

2.10 = C'B'A' + D'C'BA' + D'C'BA +D'CB'A + D'CBA' + D'CBA + DC'B'(A' + A)

____

2.11 = C'B'A' + D'C'BA' + D'C'BA +D'CB'A + D'CBA' + D'CBA + DC'B'

____

2.12 = C'B'A' + D'C'B(A' + A) +D'CB'A + D'CBA' + D'CBA + DC'B'

____

2.13 = C'B'A' + D'C'B + D'CB'A + D'CBA' + D'CBA + DC'B'

____

2.14 = C'B'A' + D'C'B + D'CB'A + D'CB(A' + A) + DC'B'

____

2.15 **= C’B’A’ + D’C’B + D’CB’A + D’CB + DC’B’**

2.16 **= C’B’A’ + D’B(C + C’) + D’CB’A + D’CB + DC’B’**

2.17 **= C’B’A’ + D’B+ D’CB’A + D’CB + DC’B’**

2.18 **= C’B’A’ + D’B+ D’C(B’A + B) + DC’B’**

2.19 **= C’B’A’ + D’B+ D’C(A + B) + DC’B’**

2.20 **= C’B’A’ + D’B+ D’CA + D’CB + DC’B’**

2.21 **= C’B’A’ + D’B•1+ D’CA + D’CB + DC’B’**

2.22 **= C’B’A’ + D’B(1 + C) + D’CA + DC’B’**

2.23 **= C’B’A’ + D’B(1) + D’CA + DC’B’**

2.24 **= C’B’A’ + D’B + D’CA + DC’B’**

2.25 **= C’B’(A’ + D) + D’B + D’CA**

2.26 **= C’B’(A’ + D) + D’(B + CA)**

Optimizing a circuit generally means reducing the number of operations or terms in a boolean expression. However, this is not always the case. Sometimes an optimization operation will produce *more* terms or operations. Consider an electrical engineer who is designing a circuit and needs to compute A+B. The engineer can use an OR gate, such as a 74LS32 integrated circuit (IC), to compute this function. A 74LS32 IC actually provides *four* OR gates; similarly, a 74LS08 chip provides four logical AND gates and a 74LS04 chip contains six inverter gates. Further suppose that the engineer already has a 74LS04 with three unused inverters and a 74LS08 with one unused AND gate, but there are no OR gates available in the current circuit. To implement this OR function, the engineer could add a 74LS32 part to the circuit. This, however, increases the cost of the circuit. If the engineer wants to reduce cost and parts count (i.e., optimize cost rather than the number of terms or operations), that engineer could synthesize the OR operation as follows:

```
A + B = (A′ • B′)′
```

That is, the engineer could use two inverters to invert the values of A and B, then use an AND gate, then use a third inverter to invert the output of the AND gate.

2.27 What theorems state that the two logic forms above are equivalent?

When optimizing logic equations for an electronic circuit implementation, simply reducing the number of terms and operators may not provide the most cost-effective solution. On the other hand, when you implement a logic function in software, the optimal form will probably the one that uses the least number of operations.

2.28 Optimize the function F=AB+A' to minimize the number of operations

2.4 Canonical Forms

Because there are an infinite number of equivalent boolean functions, we will use *sum of minterms* canonical form to specify logic functions. For any given logic function there is a unique sum of minterms form. The sum of minterms form is convenient because it is easy to create a truth table from a logic equation in this form, furthermore, it is easy to construct the sum of minterms logic equation from a truth table.

A *minterm* is a term in which all variables in a boolean expression are present in either primed or unprimed form. If you have a boolean function of four variables, then a minterm will always contain four literals (remember, a literal is a primed or unprimed variable). Because each variable can take one of two states (primed or unprimed) there are 2^n different possible minterms for a function of *n* variables. This corresponds to the number of entries in a truth table.

Each minterm in the canonical form corresponds to an entry in the truth table that is one. The sum of minterms form is the logical OR of all the minterms present in a truth table (that is, the "address" of each entry in the truth table containing a one). For example, consider the following truth table:

	B'A'	B'A	BA'	BA
C'	0	1	1	1
C	0	1	1	0

Sample Truth Table

This particular truth table contains ones in the squares addressed by C'B'A, C'BA', C'BA, CB'A, and CBA'. These "addresses" correspond to the minterms. The sum of minterms form is the logical OR of these minterms, hence, the function for the truth table above is C'B'A + C'BA' + C'BA + CB'A + CBA'.

2.29 What is the canonical (sum of minterms) form for the following truth table?

	B'A'	B'A	BA'	BA
C'	1	0	0	0
C	1	0	0	1

2.6

Table 1: J

D	C	B	A	J
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

2.7 Th 1 (D C' B' A' = D C' B' A' + D C' B' A')

2.8 Dist Law (D C' B' A' + D C' B' A' = (D + D') C' B' A')

2.9 Inv Law ((D + D') = 1)

2.10 Dist Law ((DC'B'A + DC'B'A' = DC'B'(A'+A))

2.11 Inv Law (A + A' = 1)

2.12 Dist Law (D'C'BA' + D'C'BA = D'C'B(A + A'))

2.13 Inv Law (A' + A = 1)

2.14 Dist Law (D'CBA + D'CBA' = D'CB(A+A'))

2.30 What is the canonical form for the following truth table?

	B'A'	B'A	BA'	BA
C'	1	1	0	1
C	1	1	0	1

2.31 What is the canonical form for the following truth table?

	B'A'	B'A	BA'	BA
C'	1	0	0	0
C	1	1	1	1

Building a truth table given the canonical form is just as easy. All you need to do is fill each entry in the truth table whose address corresponds to the minterms. If you have the logic equation F=C’B’A’ + C’BA + CBA + CB’A’ you would place ones in the entries corresponding to each of these minterms:

	B'A'	B'A	BA'	BA
C'	C'B'A'			C'BA
C	CB'A'			CBA

	B'A'	B'A	BA'	BA
C'	1	0	0	1
C	1	0	0	1

2.32 Provide the truth table for F = CBA + C’BA’ + CBA’ + C’BA

	B'A'	B'A	BA'	BA
C'				
C				

2.33 Provide the truth table for F=C’ B’A’ + C’B’A + C’BA’ + C’BA + CBA

	B'A'	B'A	BA'	BA
C'				
C				

2.34 Provide the truth table for F= DC’BA + DC’B’A’ + DCBA + D’CBA

	B'A'	B'A	BA'	BA
D'C'				
D'C				
DC'				
DC				

2.5 Simplifying Boolean Functions Using the Map Method

If you want to simplify a boolean expression to reduce the number of *terms* in the expression, there is an easier technique than using algebraic transformations to accomplish this – the mapping optimization method. The first step is to build a *truth map* (also known as a *Veitch Diagram* or *Carnot Map*). Truth maps are a minor variation on the truth table. The only difference between a truth map and a truth table is the layout of the rows and columns. A two variable truth map is identical to a two variable truth table. A three variable truth map is similar to a three variable truth table, except you swap the last two columns. A four variable truth map is similar to a four variable truth table except you swap the last two columns and last two rows.

	A'	A
B'	B'A'	B'A
B	BA'	BA

Two Variable Truth Map

	B'A'	B'A	BA	BA'
C'	C'B'A'	C'B'A	C'AB	C'BA'
C	CB'A'	CB'A	CAB	CBA'

Three Variable Truth Map

	B'A'	B'A	BA	BA'
D'C'	D'C'B'A'	D'C'B'A	D'C'AB	D'C'BA'
D'C	D'CB'A'	D'CB'A	D'CAB	D'CBA'
DC	DCB'A'	DCB'A	DCAB	DCBA'
DC'	DC'B'A'	DC'B'A	DC'AB	DC'BA'

Four Variable Truth Map

2.15 Inv Law (A + A’ = 1)

2.16 Dist Law (D’BC + D’B’C’ = D’B(C+C’))

2.17 Inv Law (C + C’ = 1)

2.18 Dist Law (D’CB’A + D’CB = D’C(B’A + B))

2.19 Th 11 (B + B’A = A + B)

2.20 Dist Law (D’C(A+B) = D’CA + D’CB)

2.21 Th 4 (D’B = D’B • 1)

2.22 Dist Law (D’B•1 + D’CB = D’B(1+C))

2.23 Th 6 (1+C = 1)

2.24 Th 4 (D’B•1 = D’B)

2.25 Dist Law (C’B’A’ + DC’B’ = C’B’(A’+D))

2.26 Dist Law (D’B + D’CA = D’(B+CA)

2.27 Demorgan’s theorems (Th 7 & Th 8)

2.28 F=A+B
2.29 F = C’B’A’ + CB’A’ + CBA

For the boolean function F=C’B’A + C’BA +CBA + CB’A’ the three variable truth map is

	B'A'	B'A	BA	BA'
C'	1	0	1	0
C	1	0	1	0

2.35 Build the truth map for F = CBA + C’BA’ + CBA’ + C’BA

2.36 Build the truth map for F=C’ B’A’ + C’B’A + C’BA’ + C’BA + CBA

2.37 Build the truth map for F= DC’BA + DC’B’A’ + DCBA + D’CBA

Once you construct a truth map, the next step is to visually identify groups of minterms that form 1x1, 1x2, 2x1, 2x2, 1x4, 4x1, 4x2, 2x4, and 4x4 clusters. Remember, the edges of a truth table wrap around to the opposite side. That is, the top of the truth map is adjacent to the bottom of the truth map and the left and right sides are adjacent. Given the following truth map for F=C’B’A’ + C’BA + CBA + CB’A’, we see there are two 1x2 clusters:

	B'A'	B'A	BA	BA'
C'	1	0	1	0
C	1	0	1	0

One thing you will notice about both of these clusters is that they contain squares in the C’ row and the C row. These clusters correspond to two minterms that differ only in that one minterm in the cluster contains a C’ literal and the other contains a C literal. For example, the leftmost cluster above corresponds to the two minterms C’B’A’ and CB’A’.

Algebraically, we can reduce these two terms to B’A’(C + C’). Since the parenthetical term is equal to one, we can combine these two terms to obtain the single term B’A’.

2.38 Why is (C+C’) equal to one (what rule applies here)?

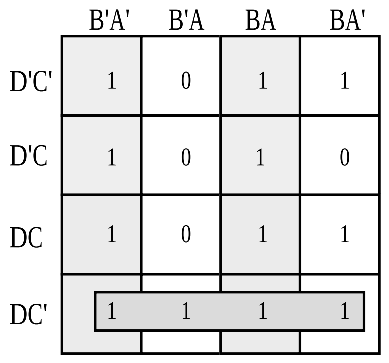
____

We can repeat this process for the second cluster in the truth map. The rightmost cluster is equal to the two minterms C’BA and CBA. We can combine these using the distributive law to obtain BA(C’ + C) which is equal to BA. Hence, F = B’A’ + BA is a simplified version of F=C’B’A’ + C’BA + CBA + CB’A’.

When grouping the clusters, you must surround all groups of squares containing ones that fit into the above patterns (1x1, 1x2, 2x1, etc.), however, these clusters must not contain any zeros. Clusters may overlap as long as at least one minterm (square containing a one) in the cluster is not enclosed inside another cluster. The object is to choose the set of largest possible rectangles in the truth map. Consider the following truth map:

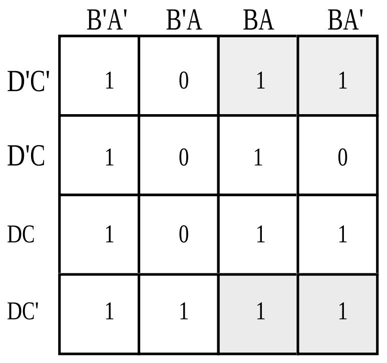
	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1

There are three 1x4 / 4x1 clusters that are very easy to spot. They are:

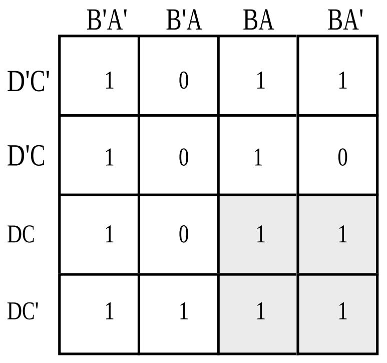


	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1

This leaves only two minterms to group into a cluster. These ones are members of two 2x2 clusters:



	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1



	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1

2.30 F=C’B’A’ + C’B’A + C’BA + CB’A’ + CB’A + CBA

2.31 F= C’B’A’ + CB’A’ + CB’A + CBA’ + CBA

2.32

Table 2: F

	B’A’	B’A	BA’	BA
C’	0	0	1	1
C	0	0	1	1

2.33

Table 3: F

	B’A’	B’A	BA’	BA
C’	1	1	1	1
C	0	0	0	1

2.34

Table 4: F

	B’A’	B’A	BA’	BA
D’C’	0	0	0	0
D’C	0	0	0	1
DC’	1	0	0	1
DC	0	0	0	1

Note that this is not the only groups of clusters we can generate for this particular logic function. For example, we could capture the D'C'BA' minterm (the one in the upper right hand corner) in the cluster formed by the four corners and we can capture the DCBA' using the bottom two on the left and right hand edges of the truth map:

	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1

	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1

It does not matter which groups we use, all choices generate an equation with a minimal number of terms. From the point of view of reducing terms, they are equivalent. We will use the former versions in our example.

Once you group the adjacent blocks of minterms into appropriate clusters, you can convert them into terms by eliminating literals that appear in both primed and unprimed forms in a cluster. For example, consider the first cluster:

	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1

This cluster encompasses minterms that have C / C' and D / D' literals. These four squares share the B'A' term. Therefore, we can create a single term by throwing out the C, C', D, and D' literals producing the simplified term B'A'.

2.39 Consider the following cluster. What literals appear in primed and unprimed form in the selected cluster?

___

	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	1	1

2.40 What is the resulting simplified expression for this cluster?

___

2.41 What variables appear in primed and unprimed form in the following cluster?

	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	1	1
DC'	**1**	**1**	**1**	**1**

2.42 What simplified logic expression corresponds to this cluster?

2.43 What variables appear in primed and unprimed form in the following cluster?

D'C'	1	0	1	1
D'C	1	0	1	0
DC	1	0	**1**	**1**
DC'	1	1	**1**	**1**

2.44 What simplified logic expression corresponds to this cluster?

2.45 What variables appear in primed and unprimed form in the following cluster?

	B'A'	B'A	BA	BA'
D'C'	1	0	**1**	**1**
D'C	1	0	1	0
DC	1	0	1	1
DC'	1	1	**1**	**1**

2.35

	B'A'	B'A	BA	BA'
C'	0	0	1	1
C	0	0	1	1

2.36

	B'A'	B'A	BA	BA'
C'	1	1	1	1
C	0	0	0	1

2.37

	B'A'	B'A	BA	BA'
D'C'	0	0	0	0
D'C	0	0	1	0
DC	0	0	1	0
DC'	1	0	1	0

2.38 Inverse Law

2.46 What simplified logic expression corresponds to this cluster?

Once you have created simplified expressions for each of the clusters, you can create a simplified expression that is equivalent to the original boolean function by logically ORing these simplified expressions. Since there were five clusters, there will be five terms in the final, simplified, boolean expression. Compare this against the 12 minterms in the original canonical expression.

2.47 Since we've never seen the "original" boolean expression for this function, how do we know there were 12 minterms in the canonical expression?

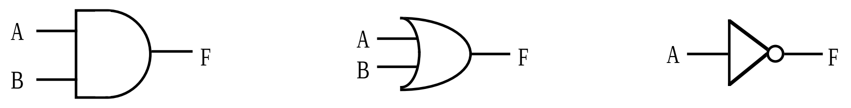
2.48 What is the simplified boolean expression from the above truth map optimization?

2.6 Electronic Circuits and Boolean Expressions

At one time electrical engineers thought in terms of gates and circuits. In modern digital design, electrical engineers supply a series of logic equations to a program that converts the logic equations to a series of bits that the engineer can program into a PLA, PAL, or FPGA. Logic languages and silicon compilers have turned much of digital circuit design into a programming exercise. Nevertheless, electrical engineers still use small scale integrated circuits like AND, OR, and inverter gates. Even if you consider yourself a "pure" programmer, you plan to avoid hardware design at all costs, you will still need to be able to read simple digital circuit schematics since you will probably have to write a program that interfaces with the hardware at one point or another. This is one of the reasons a digital logic course is typically a core course in the Computer Science program at most major universities today.

Even if you never touch an AND, OR, or inverter integrated circuit in your life, it's still a good idea to learn how to read and write simple circuit diagrams. As it turns out, an electronic schematic provides a visual representation of a logic function; it is often easier to read and understand a schematic than it is to read and understand a truth table or a boolean expression.

Although there are *thousands* of electronic schematic symbols, we'll only need to use three since we can design any logic circuit using only logical AND, OR, and NOT. The corresponding schematic symbols are

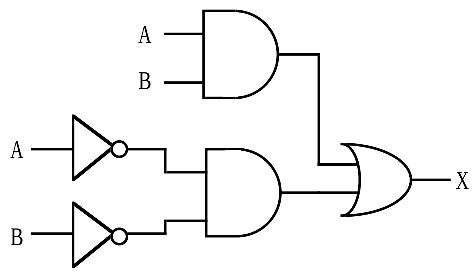


F = AB F = A + B F = A'

The lines on the left hand side of each symbol represent inputs to the function. The symbol itself denotes the function. The line on the right of each symbol is the output from the function.

Generally, symbols like A, B, C, D, and other upper case letters denote circuit inputs or outputs. Intermediate outputs that are fed into inputs elsewhere in the circuit generally are not given symbolic names. For example, the function X = AB + A'B' would look like the following:

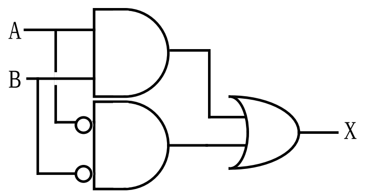
2.39 C/C’ and D/D’



2.40 BA

F = AB + A'B'

Since inverters commonly appear in a boolean expression, we’ll adopt the convention of using a little bubble on and input or output line to denote logical negation. We can redraw the circuit above as:



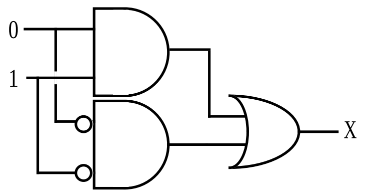
2.41 A/A’ and B/B’

F = AB + A'B'

This logic function, by the way, is the *exclusive-NOR* or *equality* function.

To trace through a logic circuit to determine its output, you begin by labelling the input lines with the values of the input variables. For example, if A=0 and B=1 you would begin by placing a zero on every line labelled by A and a one on every line labelled with B:

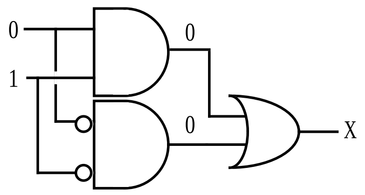
2.42 DC’



2.43 A/A’ and C/C’

F = AB + A'B'

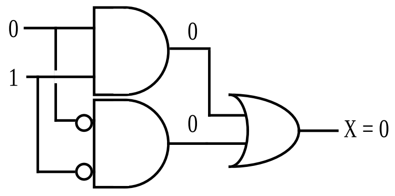
The next step is to compute the outputs for every gate (function) those input lines feed:



2.44 DB

F = AB + A'B'

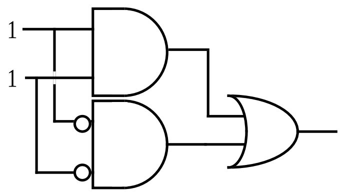
You keep repeating this process until you compute the final output of the function:



2.45 A/A’ and D/D’

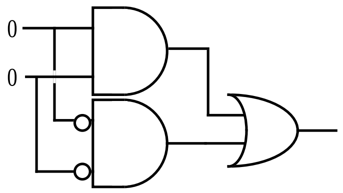
F = AB + A'B'

2.49 Given the inputs A=1 and B=1, provide all the function results for the F=AB+A’B’ schematic:



F = AB + A'B'

2.50 Given the inputs A=0 and B=0, provide intermediate and final function results for

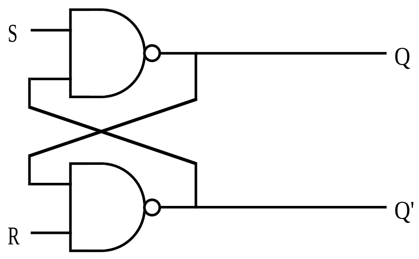


F = AB + A'B'

2.51 Provide a schematic for the exclusive-OR (not equal) function:

2.7 Combinatorial Circuits

A combinatorial circuit is one whose outputs depend only on its current inputs. The outputs immediately reflect changes to the inputs[2]. The AND, OR, and inverter gates are examples of simple combinatorial circuits. One important restriction on a combinatorial circuit is that it must not have any *feedback*. That is, an input to a gate cannot be a function result that depends on the output of that gate; that is, you cannot have any *loops* in a combinatorial circuit. For example, the following circuit is not a combinatorial circuit because the inputs to the two NAND gates depend on the outputs of each other:

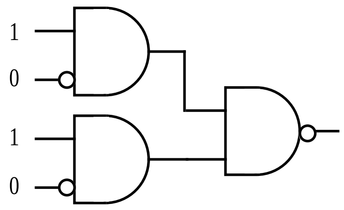


A Sequential (non-combinatorial) Circuit

2. Real electronic circuits require a small amount of time between the application of the inputs and the output of the corresponding function result. However, we will ignore these *propagation* delays.

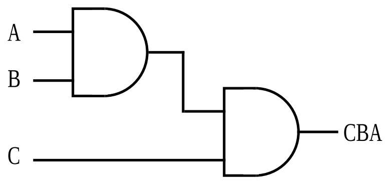
A *decoder* is a good example of a combinatorial circuit. A decoder is a circuit that produces a specific value (typically a zero[3]) when a specific value or a specific set of values appears on its inputs. For example, a simple decoder circuit that produces a zero output when four input lines contain 1010 would be the following:

2.46 C'B



Decoder for input value 1010

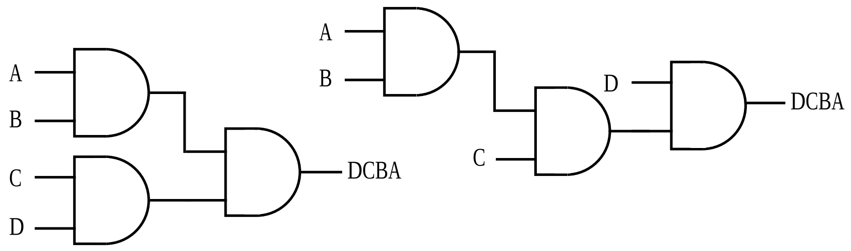
Another set of important combinatorial circuits are the *n-input AND gates* and the *n-input OR gates.* For example, if you want to compute the logical AND of three inputs, you can *cascade* two AND gates as follows:



2.47 Because 12 entries in the truth map contain ones.

Three Input AND Function

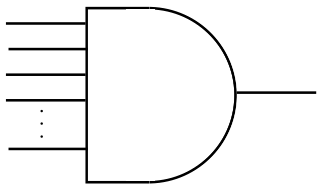
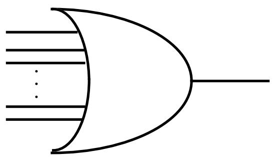
Similarly, if you want to construct a four input AND gate, you can combine three two-input AND gates as either of the following:



Two Variations of a Four Input AND Gate

You can construct an *n-input OR gate* in a similar fashion. Since logical AND and OR gates with more than two inputs are common (consider the minterms of four variables), we will use a single symbol to denote *n-input* gates:

2.48 F=C'B+DB+DC'+BA or F = B(C'+D+A) + DC'

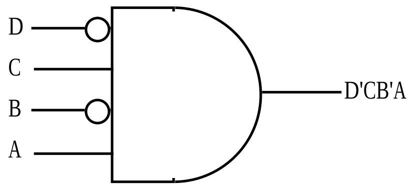



n-Input AND Gate *n*-Input OR Gate

To convert a logic equation to an electronic schematic, begin by constructing an *n*-input AND gate for each term in the expression. If an unprimed literal appears in the term, then supply

3. Intuitively, one would expect a decoder to produce a logic one when it detects its trigger value. However, electronic decoder circuits typically produce a zero when they trigger, so we will follow that convention.

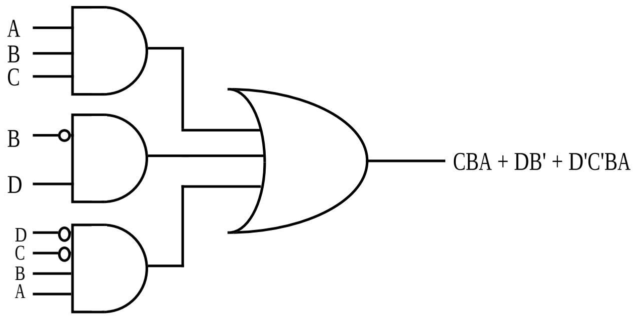
that input directly to the AND gate; if a primed variable appears in the term, invert that variable before supplying it to the AND gate (i.e., put a little bubble on that input line). For example, the minterm D'CB'A has the following schematic symbol:



Schematic for D'CB'A

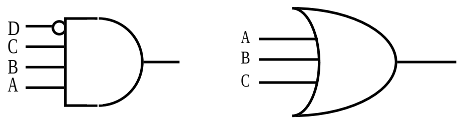
2.52 Provide the schematic diagrams for CBA' (use a three-input AND gate) and DCB'A' (use a four-input AND gate):

Once you design the circuits for the individual terms in a boolean expression, you can complete the circuit for that expression by ORing together the outputs from the AND gates for each of the individual terms. The function F= CBA + B'D + D'C'BA becomes:



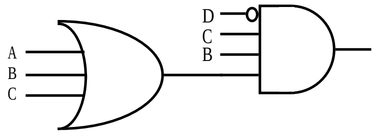
2.53 Provide the schematic diagram for the function F= DCB'A' + CBA'

Going in the other direction, converting a schematic diagram to a logic equation, is just as easy. All the inputs to a logical AND gate become a single term. Inputs to an OR gate become sums in an expression. If there is a bubble on an input, you negate (invert) that input by adding a primed to the input variable:



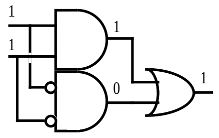
D'CBA (C+B+A)

If the output of a particular gate is fed into the input of some other gate, simply surround that gate's expression with parentheses and use that parenthetical expression as the input "variable" for the second gate:



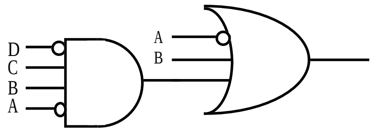
(A+B+C) • B • C • D'

2.49

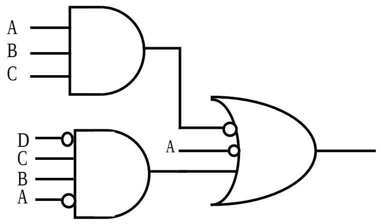


F = AB + A'B'

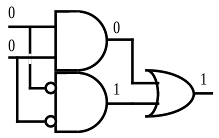
2.54 What is the boolean expression that corresponds to the following circuit?



2.55 What is the boolean expression that corresponds to the following circuit?



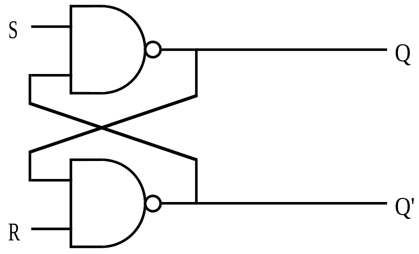
2.50



F = AB + A'B'

2.8 Sequential Circuits

A sequential circuit is a combinatorial circuit with the addition of *feedback*. That is, one or more outputs that are fed back as inputs to the circuit. This provides the circuit with a memory, or history, of previous computations. Consider the *set/reset flip-flop* given as an example of a non-combinatorial circuit in the previous section:

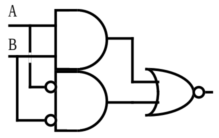


Set / Reset Flip-Flop

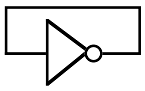
The output of the top NAND gate is an input to the bottom NAND gate and the output of the bottom NAND gate is an input to the top NAND gate.

Some sequential circuits are *unstable*. That is, you can feed the inputs back into a circuit and that feed back will change the output, the new output fed back into the circuit will change it again, and so on forever. A simple example of an unstable sequential circuit is a simple inverter with its output connected back to its input:

2.51



F = A xor B



An Unstable Circuit

In this circuit, an initial input of one produces an output of zero. The circuit feeds this zero back into the input and it produces a one as the output. This output feeds back into the input to produce a zero, and so on...

In the mathematical sense, an unstable circuit like this one is *inconsistent*. This is comparable to a division by zero or taking the logarithm of a negative number, you just don't do it. In the electronic world, building an unstable circuit produces an *oscillation*. That is, the output continually changes between zero and one at a frequency related to the propagation delay of the circuit.

A stable circuit, on the other hand, settles on a particular output and stays in that state until the inputs change. The set/reset flip-flop is an example of a stable circuit. As long as the inputs do not change, the outputs remain constant. Normally, the S and R inputs are a logic one. Setting the S input to zero forces the Q output to zero (and Q' to one). The outputs remain in this state even after you return the S input to a logic one. Similarly, setting the R input to zero forces the Q' output to one (and the Q input to zero). Returning the R input to one leaves the outputs in this state. This S/R flip-flop is a good example of a *memory circuit*. It remembers the input that was last set to zero.

2.56 What will the Q and Q' outputs contain if you set *both* inputs to zero?

____

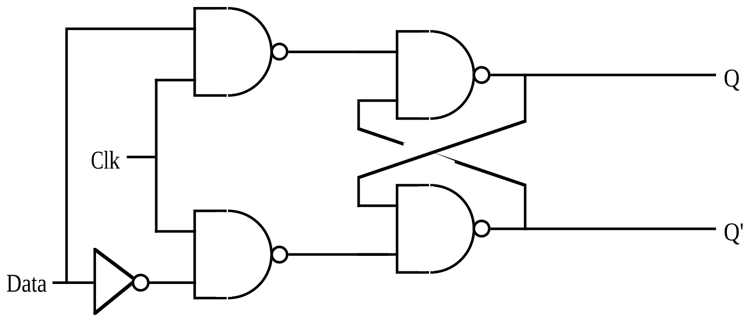
2.57 What will the Q and Q' outputs contain if you set both inputs to zero and then set R to one while leaving S at zero?

____

2.58 What will the Q and Q' outputs contain if you set both inputs to zero and then set S to one while leaving R at zero?

____

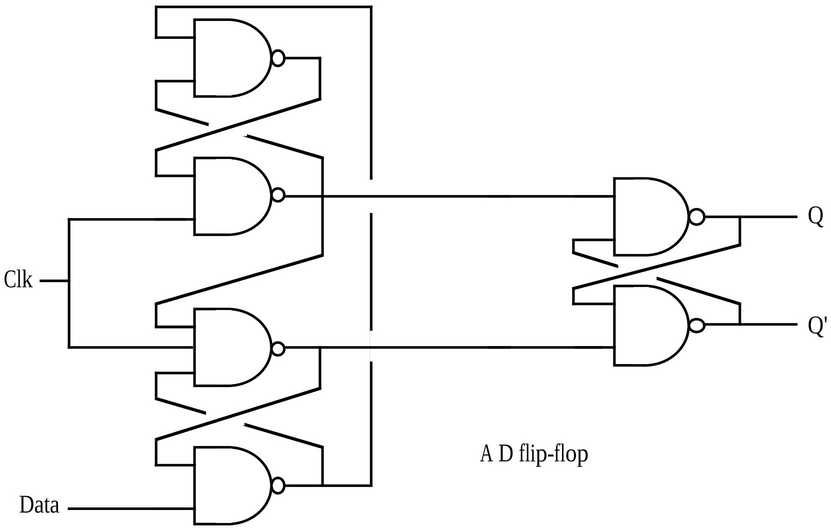
The textbook presents the following schematic for a *D flip-flop*:



A Transparent Latch (D flip-flop)

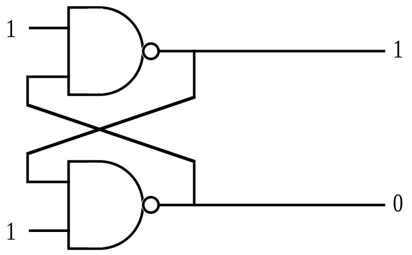
Technically, this is a *one bit transparent latch*, not a D flip-flop. A D flip-flop captures the data on the Data input when the clock changes from low to high (the *rising edge* of the clock). The outputs do not change while the clock is in a low (zero) or high (one) state, only when there is a transition from low to high. The circuit above does not work this way. As long as the clock line is high, whatever value you place on the Data input is sent to the Q output (and the inverse value to the Q' output). That is why this is a *transparent* latch. As long as the clock line is high, the circuit is transparent – the data immediately flows to the output. When you set the clock line low, the last value on the Data input is latched on the outputs.

A true D flip-flop only captures the Data input value when the clock line switches from low to high. The Data line needs to be stable (maintain the same value) during this transition, but it can contain any other value at any time without affecting the outputs. The circuit for a D flip-flop is



A D flip-flop

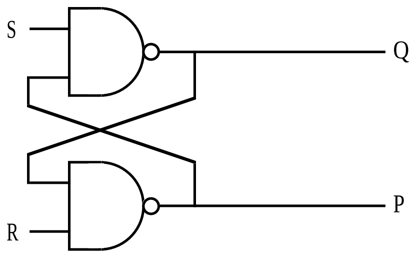
Like combinatorial circuits, we can construct boolean expressions for sequential circuits. We will use the "#" symbol to denote the clock input. For values we feed back into the circuit, we will assign an intermediate variable name (or use the function output name) and use the name as though it were an input variable. When computing the output for a sequential circuit, you use the *previous* output values as the feedback values in the sequential circuit. For example, suppose you have the following state of a set / reset flip-flop:



Set / Reset Flip-Flop

If you change the R input from one to zero, this forces the Q' output to one because (RQ)' = (0•1)' = 1. The Q' output is fed back into the Q top NAND gate so its inputs are one (Q') and one (S). This forces the Q output to zero.

To generate the boolean expressions for a sequential circuit, you start at each of the outputs and work backwards. We will assign the names P and Q to the outputs of the NAND gate[4]. Here is the circuit and the corresponding equations:

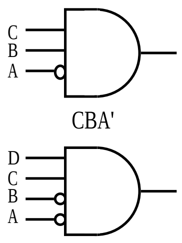


Q = (SP)'
P = (RQ)'

Set / Reset Flip-Flop

4. Although electrical engineers label these outputs Q and Q', the Q' output is really a different function; it is *not* the complement of the Q output. For example, if you set the S and R inputs to zero, then the outputs are not the complements of one another. To properly describe this circuit we must use different boolean expressions for the two outputs.

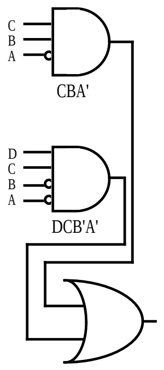
2.52



CBA'

DCB'A'

2.53

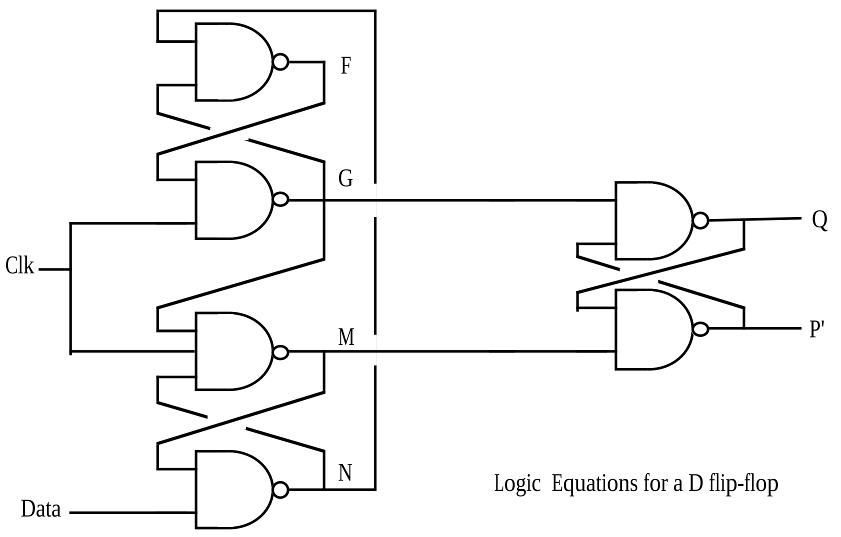


DCB'A' + CBA'

2.54 (D'CBA') + A' + B

2.55 (ABC)' + (D'CBA') + A'

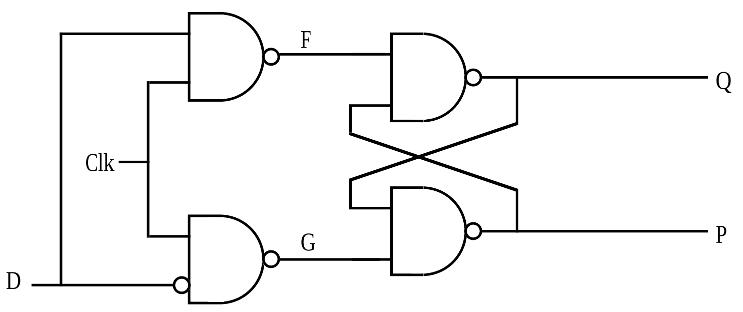
For a more complex circuit that has several intermediate outputs that it uses as other inputs, you will have to invent some variable names to represent those outputs. Consider the D flip-flop and its logic functions:



F = (GN)'
G = (F#)'
M = (GN#)'
N = (DM)'
Q = (GP)'
P = (MQ)'

Logic Equations for a D flip-flop

2.59 What is the set of logic equations for the one bit transparent latch?



One Bit Transparent Latch

2.9 Simulating Logic with Software

Since there is a one-to-one relationship between boolean expressions and electronic circuits and a one-to-one relationship between boolean expressions and computer programs, clearly there is a one-to-one relationship between electronic circuits and computer programs. A course on digital design will teach you how to design a circuit that implements some algorithm; since this text covers computer programming, we will concentrate on converting electronic circuits into software.

Combinatorial circuits are very easy to implement in software. All you need do is convert the circuit to a boolean expression. Converting a boolean expression to a function in a high level language like C++ or Pascal is a trivial exercise. Consider the boolean expression for segment four of a seven segment display:

$$S_4 = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'$$

The Pascal function that implements this expression is:

```
function S4(D, C, B, A:boolean):boolean;
begin

      S4 := ((not D) and (not C) and (not B) and (not A))
or
            ((not D) and (not C) and B and (not (A)) or
            ((not D) and C and B and (not A)) or
            (D and (not C) and (not B) and (not A));
end;
```

2.56 Q=0, Q'=0

The corresponding C++ code is

```
int S4(int D, int C, int B, int a)
{
      return      (!D && !C && !B && !A)  ||
                  (!D && !C && B && !A)||
                  (!D && C && B || !A) ||
                  (D && !C && !B && !A);
}
```

Code like this demonstrates why you might want to optimize a boolean expression before you convert it to a function. As you may recall from the section on optimization in this chapter (see "Algebraic Manipulation of Boolean Expressions" on page 48), a simplified version of this boolean expression is

```
C'B'A' + D'B(C'A + CA')
```

The corresponding C++ code is

```
int S4(int D, int C, int B, int A)
{
      return      (!C && !B && !A) ||
                  (!D && !B && (!C && A || C && !A));
}
```

Clearly, this last version will be shorter and faster unless you have a *really* good compiler that can optimize the previous code. Even if you have such a compiler, the latter version of the code is still better since it is shorter and therefore easier to read and understand.

2.57 Q=1, Q'=0

2.60 Provide a short C++ or Pascal function that implements the boolean expression F= A+BC'

Converting a sequential circuit to a program is complicated by the fact that the output of a sequential circuit depends on the previous outputs of the circuit. Another complication with the conversion of a sequential circuit to a high level language function is *propagation*. To understand the difficulty with propagation of signals, consider the logic equations for the set / reset flip-flop appearing earlier in this chapter:

```
Q = (SP)'
P = (RQ)'
```

2.58 Q=0, Q'=1

The corresponding C++ functions look like the following:

```
int Q(int S, int P)
{
      return !(S && P);
}

int P(int R, int Q)
{
      return !(R && Q);
}
```

Given four values, S, R, Qval, and Pval, you might think you can compute the output values (Qval and Pval) using two calls like the following:

```
Qval = Q(S, Pval);
Pval = P(R, Qval);
```

The problem with this simplistic approach is that Qval's computation uses the old version of Pval while the computation of Pval uses the new value of Qval. Since the two outputs depend on one another, we would need to recompute Qval's value since it depends on Pval's new value:

```
Qval = Q(S, Pval);
Pval = P(R, Qval);
Qval = Q(S, Pval);
```

However, this can produce a new value for Qval that affects Pval. So we have to repeat the computation for Pval:

```
Qval = Q(S, Pval);
Pval = P(R, Qval);
Qval = Q(S, Pval);
Pval = P(R, Qval);
```

Obviously we can repeat this process forever, the question is "when do we stop?" As it turns out, if a circuit is stable, then the outputs will stabilize after you repeat the sequence of operations *n* times where *n* is the number of functions (AND and OR gates) in the circuit (worst case). Since the set / reset flip-flop has only two NAND gates, we need only execute the output functions twice, so the last example above will produce the correct output.

2.61 How many iterations will the one bit transparent latch circuit require (worst case)?

____

2.10 The LOGICEV Laboratory Exercises

In this laboratory you will construct a simple circuit containing LEDs and switches. You will then run the LOGICEV.EXE program to simulate various logic functions. The LOGICEV.EXE program reads its inputs from the switches and writes its outputs to the LEDs.

Warning: This laboratory uses two different resistor values, some value between 2.2K and 3.3K for the LEDs and 10K for the switches. If you use the 10K resistor on the LEDs no harm will occur but the LEDs will probably be too dim to see. On the other hand, there is a small chance that you can damage your computer if you connect the 2-3K resistors to the switches. Carefully double check your work after you assemble the circuit to verify it's correctness.

2.10.1 Before Coming to the Laboratory

Your pre-lab report should contain the following:

- A copy of this lab guide with all the questions answered (note: there are more questions after this point in this chapter!).
- A write-up of the LOGICEV.EXE program explaining, in your own words, how it operates.
- Several logic equations and schematics for simple combinatorial circuits you've designed.

In addition to the pre-lab report, you should construct the circuit (described below) for this lab before coming to the laboratory.

2.59 F=(D#)'
G=(D'#)'
Q=(FP)'
P=(GQ)'

See Chapter Two of this laboratory manual for an example pre-lab report.

Note: your Lab Instructor may elect to give a quiz before the lab begins on the material covered in the laboratory. You will do quite well on that quiz if you've properly prepared for the lab and studied up on the stuff prior to attending the lab. If you simply copy the material from someone else you will do poorly on the quiz and you will probably not finish the lab. Do not take this pre-lab exercise lightly.

2.10.2 Constructing the Circuitry Required for this Lab

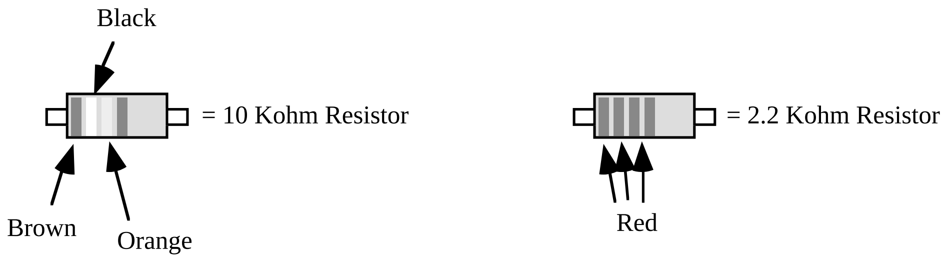
For this laboratory exercise you will need the solderless breadboard (Radio Shack part number 276-175), eight low-current LEDs (RS part number 276-044), eight 2.2K resistors (RS part number 271-1325) or eight resistors in the range 2.5-3.3 Kohms, four 10 Kohm resistors (RS part number 271-1335), one four position DIP switch (RS part number 275-1304c), the DB25F connector you constructed (see the Forward to this lab manual), 22 gauge (22 AWG) solid wire, preferably in multiple colors, and a pair of wire strippers.

You must exercise some caution during the construction of the circuitry for this laboratory. You will be using two different resistor sizes – 2.2 Kohms for the LEDs and 10 Kohms for the switches. If you swap these and connect the 10 Kohm resistors to the LEDs, they will not light (or they will be very dim). That will not harm the LEDs or your computer. On the other hand, there is a small chance that connecting the 2.2 Kohm resistors to the switches could damage the parallel port on your computer system. Therefore, exercise caution when constructing the circuitry. Be sure to connect the resistors properly.

2.60 C++:
```
int f(int C, int B, int A)
{
   return A || B && !C;
}
```

Pascal:
```
function     f(C,B,A:boolean):
boolean;
begin
F := A or (B and not C);
end;
```

You identify the resistance of a resistor by a *color code* on the resistor. The 2.2 Kohm resistors have three red bands on them (generally a fourth band will be gold or silver, you can ignore this band). The 10 Kohm resistors have a brown, black, and orange band on them:

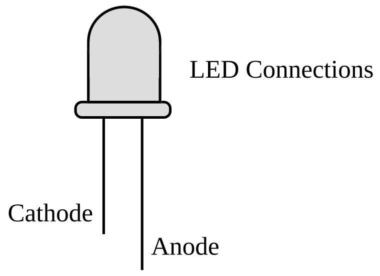


Identifying the Resistors in Your Parts Kit

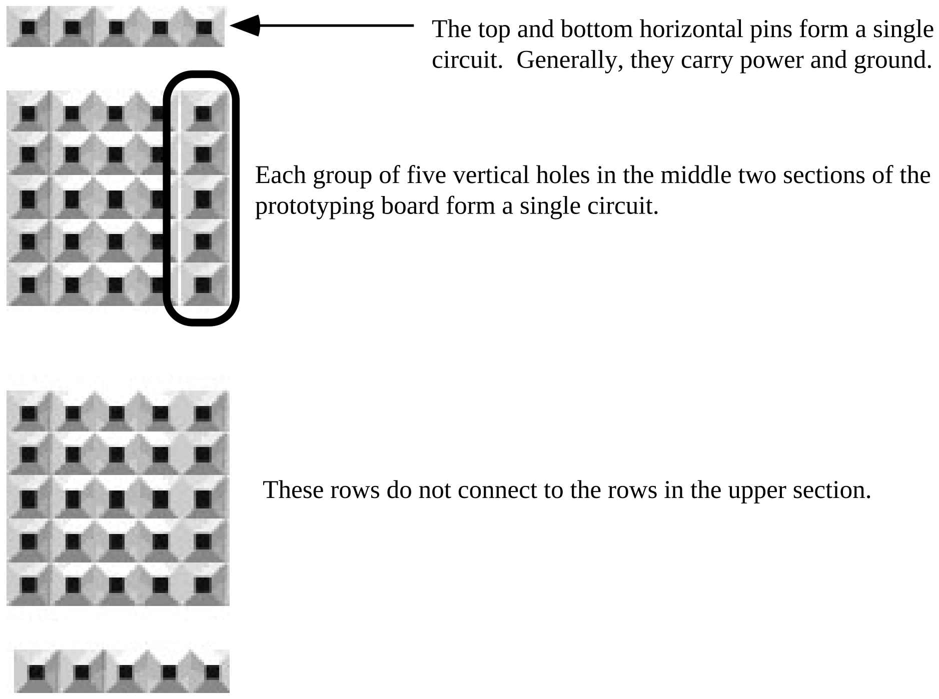
The orientation of a resistor is immaterial. You can connect the resistor into a circuit with the bands on the positive side of the circuit or on the negative side; resistors have no polarity.

LEDs, on the other hand, do have a polarity. If you connect them backwards into a system they will not work properly. The names of the leads (wires) on LEDs are anode (positive) and cathode (negative). If you connect a positive voltage source to the anode and ground to the cathode, the LED will emit light. If you connect the LED backwards, it will not emit any light. This will not, however, damage the LED.

If you take a look at the LEDs, you will notice that the two leads are not the same length. This is how you identify the anode and cathode leads. The anode is longer than the cathode. Since you will probably be trimming the leads on the LEDs, it wouldn't be a bad idea to put a small scratch or some other permanent mark on the LED near the anode.



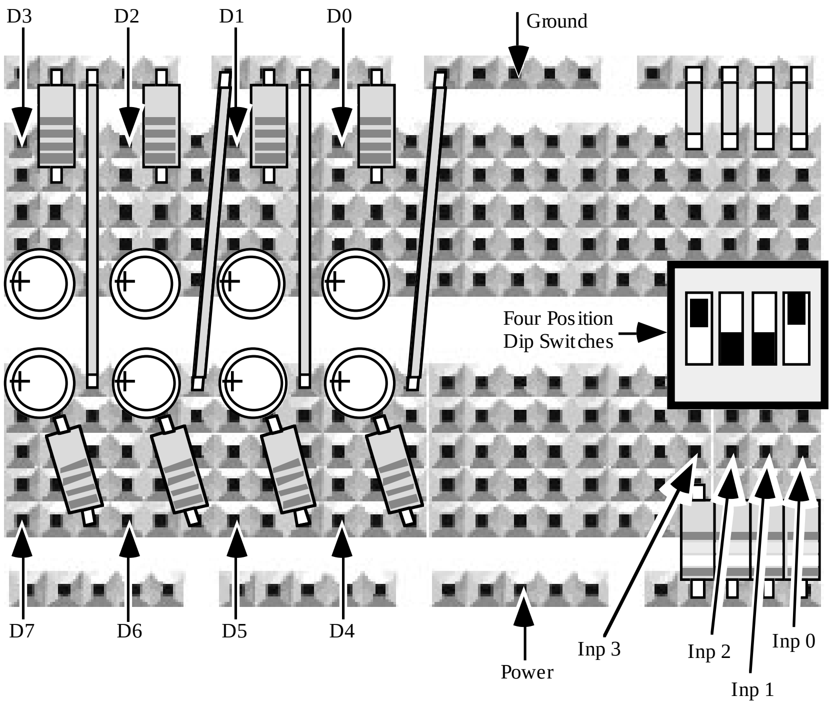
If you have never used a solderless breadboard before, the concept is quite simple. A typical working area consists of two sets of five rows of holes and one row above and below the main breadboarding area. The holes in the top row form one circuit and the holes in the bottom row form one circuit. These two rows generally carry the power and ground signals. Note that some breadboard brands do not have the power and ground *busses* or *tie points* at the top and bottom of the breadboard. You will have to use one (or more) of the other rows for this purpose on such breadboards.

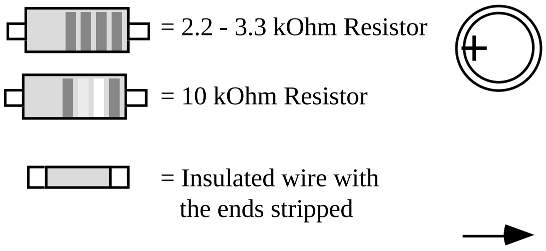

The top and bottom horizontal pins form a single circuit. Generally, they carry power and ground.
Each group of five vertical holes in the middle two sections of the prototyping board form a single circuit.
These rows do not connect to the rows in the upper section.

If you insert two 22 gauge (22 AWG) wires into a pair of connected holes, the breadboard electrically connects those two wires. It is very easy to prototype a circuit by inserting resistors, LEDs, switches, and wires into one of these solderless breadboards.

When you construct the circuit for this laboratory experiment, your breadboard should look like the following:

2.61 Four





= 2.2 - 3.3 kOhm Resistor

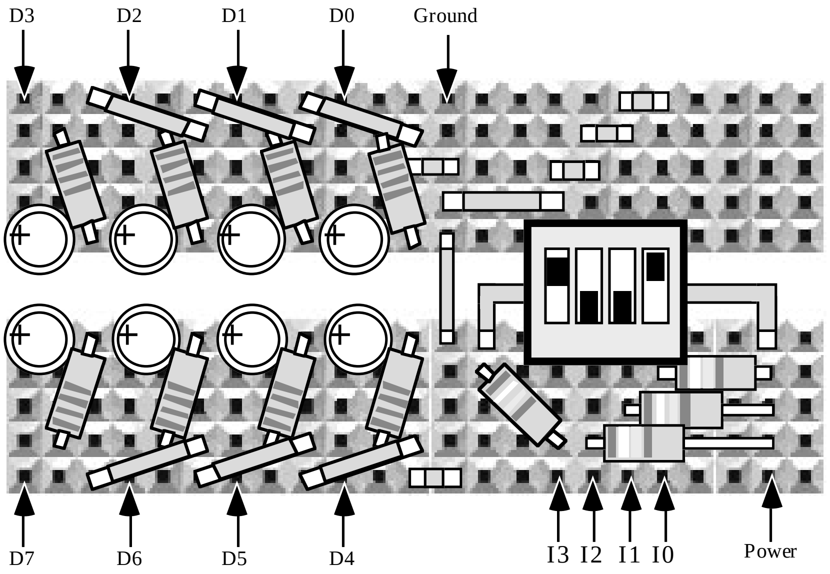
= 10 kOhm Resistor

= Insulated wire with the ends stripped

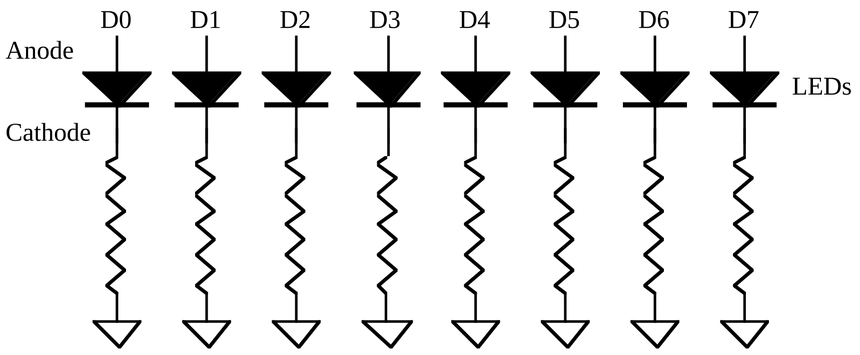
Low Current LED. Anode (positive) is the longer lead on the LED. The cathode (negative) lead is horizontally opposite the anode.

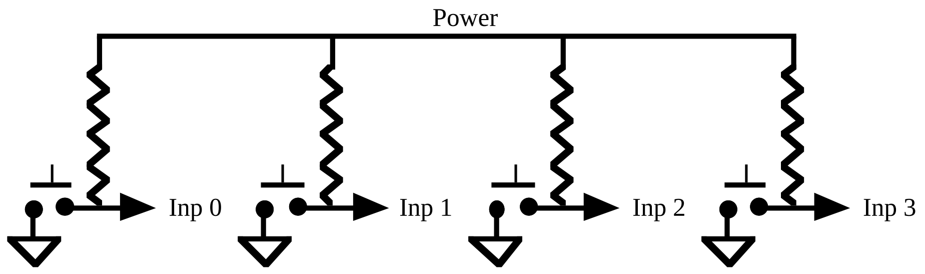
Arrows represent wires from the parallel port connector.

Suggested layout if you do not have power and ground busses on your breadboard:

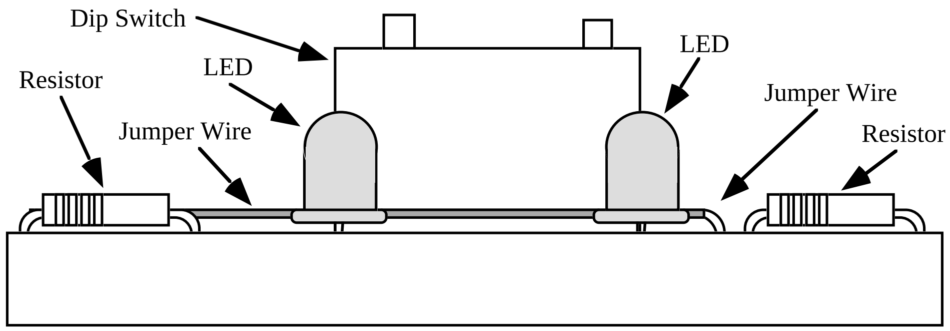


The electronic schematic for the circuit for this lab is very simple:





You will be using this circuit for several lab experiments in this manual. Therefore, you should take care and assemble it neatly. You should trim the leads on the resistors so that the resistors lie flush with the breadboard. The same is true for the jumper wires (the wires between the resistors and ground and the wires between the switches and ground). You can also trim the leads on the LEDS, but don't forget to mark the anode on the LEDs so you know which end is positive.



Side View of Circuit for this Laboratory

The most important feature of this circuit is that a resistor sits in-line between the LEDs and the circuit from power to ground. Similarly, this circuit contains a resistor in the circuit between the power line and the switches. Double check your circuit before connecting it to a computer to verify that you have inserted the resistors in their proper places in the circuit. Also, have your lab instructor check this circuit before you connect it to the computer. Remember, if the resistors are not present or are installed incorrectly you could damage the LEDs, the computer, or both.

2.10.3 Testing and Debugging the Circuit

Like beginning programmers have trouble understanding why their programs don't work right off the bat, beginning circuit designers can't believe their circuits don't always work the first time. However, very simple wiring errors can prevent the entire circuit from working properly. Soon, you will be designing boolean functions that read their inputs from the switches in your circuit and write their outputs to the LEDs in your circuit. If your circuit doesn't work properly, the software that drives the circuit will fail. If you just plug your circuit into the computer and start supplying logic functions to the software, an incorrect result could result from a bad logic function *or* a bad wiring job. You can spend a lot of wasted time trying to figure out why your logic functions are incorrect, only to find that the logic functions *are* correct and you've just wired the board wrong.

On the diskette accompanying this lab manual there is a program (TESTCIR1.EXE) that lets you test your wiring job. After you manually check over your wiring and have your lab instructor double check it, plug your connector into a six foot DB25 male to male cable connected to the printer port and run the TESTCIR1.EXE program. This program is very simple. It will light the LEDs one at a time and ask you if the specified LED is on. If so, you press the Enter key to try out the next LED.

The two most common problems with the LEDs are (1) the specified LED does not come on or (2) the wrong LED lights. If the specified LED does not illuminate, the problem is likely to be one of the following:

- The LED is installed backwards. You must connect the anode to one of the data lines (D0..D7).
- The appropriate line from the DB25F connector you built is not plugged into the same column of holes as the anode of the LED.
- The cathode of the LED is not connected to the 2.2K resistor.
- You have connected the wrong wire from the DB25-F connector to the LED.
- The other side of the resistor is not connected to ground (possibly through a jumper wire).
- If an incorrect LED comes on, you've connected the wrong data line to the anode of that LED.
- You've installed your circuit in the wrong printer port (see comment below).

If you discover a problem with your LEDs, correct the wiring error and retest the circuit from the beginning.

The second part of the test program checks out the wiring on your switches. It will ask you to flip the switches on and off (individually). If the program discovers a problem, it will report an error message. If you get such an error message, the problem is likely to be one of the following:

- There is no resistor going from power to one side of the DIP switch.
- The other side of each DIP switch does not go to ground.
- You have connected the wrong wires on your DB25 connector to the switches.
- You connected the wires from the DB25F connector to the wrong side of the DIP switches (the input lines go between the resistor and the DIP switch, not on the side of the switch that goes to ground).
- You've installed your circuit in the wrong printer port (see comment below).

If you discover a problem with your switches, correct the wiring error and retest the circuit from the beginning.

By default, the TESTCIR1.EXE program assumes that you've connected your circuit to the LPT1: printer port. Most PCs have only one printer port and it is LPT1. However, if your machine has more than one printer port, you may want to connect your circuitry to LPT2: or LPT3: rather than LPT1:. To test out your circuitry in a different parallel port, simply specify the parallel port's name as a command line parameter to TESTCIR1:

```
TESTCIR1                Uses LPT1 by default.
TESTCIR1 LPT1:          Force the use of LPT1:.
TESTCIR1 LPT2:          Use LPT2:.
TESTCIR1 LPT3:          Use LPT3:.
```

2.10.4 The LOGICEV Program

In this laboratory you will be running the LOGICEV.EXE (Logic Evaluation) program provided on the diskette accompanying this lab manual. If you run this program from MS-DOS, it will assume you've connected your circuitry to the LPT1: printer port. If you want to use a different printer port, specify the port's name as a command line parameter (just like the TESTCIR1 program, above):

```
LOGICEV
LOGICEV LPT1:           Force the use of LPT1:.
LOGICEV LPT2:           Use the circuitry connected to LPT2:.
LOGICEV LPT3:           Use the circuitry connected to LPT3:.
```

After an initial greeting screen, the LOGICEV program expects you to enter a sequence of logic formulae. A logic formula consists of a function name, following by an equal sign, and then a boolean expression for that formula:

name = expression

All LOGICEV variable and function names are a single alphabetic character. The variable names A, B, C, and D are reserved for use as inputs. On each iteration of the simulation program LOGICEV reads the values for A, B, C, and D from the switches (open=0, closed=1). Likewise, LOGICEV writes the values for W, X, Y, and Z to LEDs D3, D2, D1, and D0. It also copies the input values from the switches to LEDs D7, D6, D5, and D4 (D, C, B, and A, respectively).

Logic expressions may contain primed and unprimed variables, the "+" operation, the "*" operator, and parentheses. The "*" is optional; you can compute the logical AND of two values by concatenating the variable names. Logical AND takes precedence over logical OR. Logical NOT (') has the highest precedence. Examples of legal functions:

```
F=AB+C
G=D'+C(A+B)'
X=F+G
Y=F*G
Z=(A+B)(C+D)
```

In addition to the variable names A-Z, you can also specify a clock signal for sequential circuits. Use the "#" symbol for the clock signal. The LOGICEV program generates a positive pulse for the clock signal. That is, the clock signal is normally low, goes high briefly, and then goes back low. If you need a negative pulse (high to low and back to high) simply negate the clock symbol by placing a prime on it:

```
#'          Generates a negative going clock.
```

LOGICEV limits boolean expressions to four or fewer variables plus an optional clock symbol. If you use more than four variables in an expression (five variables counting the clock symbol), LOGICEV will generate an error. Since there are only four switches on your circuit, four input variables to any given function should be sufficient. If, however, you absolutely need five or more variables in an expression, you can always break a single expression into two separate expressions. For example, if you want to compute W=ABCDEF+G, you could use the following statements:

```
H = ABCD
W= HEF+G
```

This will compute the value you desire. Note that LOGICEV limits you to four separate variables in an expression, it does not limit the number of terms or literals in an expression[5]. The following is perfectly legal:

```
Z = A+B(CA'+ DA) + A'B'C'D' + D(A+B')
```

After you enter a logic expression, LOGICEV checks the syntax to see if there are any errors in the expression. Entering too many different variables is a good example of an illegal expression. There are other errors as well. For example, LOGICEV will complain about a statement that looks like this:

```
X = A + + B
```

Another common error is reusing a function name. Function names must be unique (it doesn't make any sense to have two different functions for the variable F in a system). If LOGICEV prints an error message, correct the error and reenter the function.

5. Other than a limitation of 128 characters per line.

After you enter your boolean function, LOGICEV will ask you if you want to see the truth table for that function. If your answer is affirmative, LOGICEV will display the truth table for you. If you use the clock symbol in your boolean function, LOGICEV will display two truth tables, one for #=0 and one for #=1. If you did not use the clock symbol in your boolean function, LOGICEV will display the canonical form of the equation.

After displaying the truth table (or after you answer "N" if you didn't want to see the truth table), LOGICEV will ask you to enter a second function. You can enter as many logic functions as you desire, although there is a practical limitation of 22 functions since function names have to be unique alphabetic characters and cannot be A-D (the inputs).

LOGICEV does not let you enter the literal values zero and one, but you can easily synthesize these values using the boolean expressions (A+A') or (AA').

2.62 Which of the above expressions is equal to one? Why?

2.63 Which of the above expressions is equal to zero? Why?

Another way to generate zero or one in an expression is to initialize a variable to zero or one and use that variable within your expressions. You'll see how to initialize variables in just a moment.

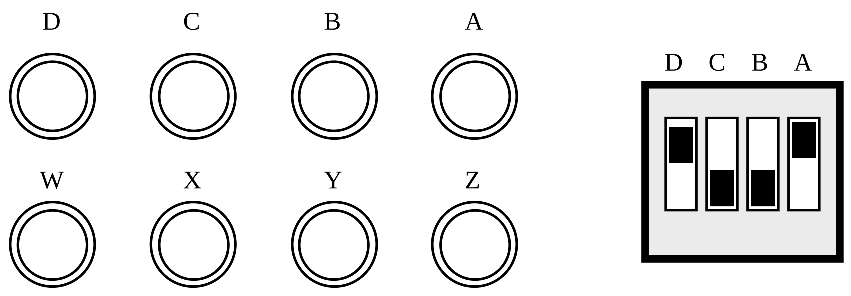
When you are done entering boolean functions, simply hit the Enter← key to complete the entry of boolean values. The LOGICEV program will process and store away truth tables for all the functions you've entered so it can simulate those logic functions later.

After you enter the logic functions, LOGICEV will allow you to initialize any variables. By default, variables E-Z start with the value zero. In the initialization screen you can toggle any values you like (except A-D) by pressing the corresponding key on the keyboard. For example, to initialize E to one all you need do is press the E key. To set it back to zero, press it a second time. When you are done initializing the variables, press the Enter← key to complete the operation. Since you'll rarely need to initialize any variables, you will usually press Enter← without bothering to change any values.

Before actually beginning the simulation, LOGICEV will ask you if you want to emulate the hardware circuitry using the keyboard. This lets you try out logic functions without having built the circuitry. Since you've built the circuitry for this laboratory, select "Hardware" operation rather than software operation. Once you select the hardware mode, LOGICEV enters the simulation mode.

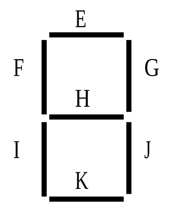
In the simulation mode, LOGICEV responds to A-D, ENTER, and ESC. Pressing A-D toggles the input variables in the software mode; in the hardware mode LOGICEV reads the inputs from the switches so pressing A-D has little effect. Pressing the Enter← key simulates the circuit based on the current input and variable values. This also triggers one pulse on the clock. Pressing ESC terminates the program.

Once the software circuit simulation begins, you will normally set the switches on your bread board, press the Enter← key, and then observe the results on your LEDs. LOGICEV reads A-D from the switches and writes A-D and W-Z to the LEDs as follows:

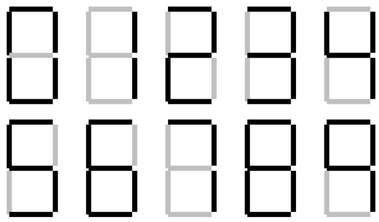


2.10.5 Quesitons Associated with the Laboratory Exercises

The LOGICEV program simulates a seven segment display on the PC's video screen. Variables E-K light the individual segments as follows:



Here are the segments to light for the binary values DCBA = 0000 - 1001:



2.64 What is the boolean expression for function E?

2.65 What is the boolean expression for function F?

2.66 What is the boolean expression for function G?

2.67 What is the boolean expression for function H?

2.68 What is the boolean expression for function I?

2.69 What is the boolean expression for function J?

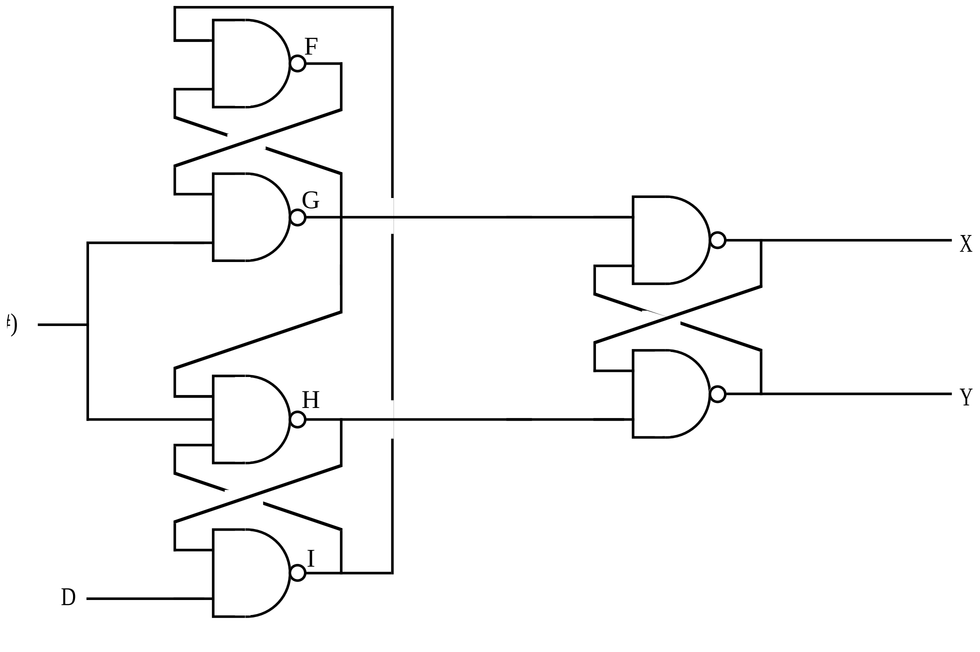
2.70 What is the boolean expression for function K?

2.62 A + A’ by the inverse law

In the laboratory, you will demonstrate the equivalence of two boolean functions. The function you will use is X=A(B+C)’ + D’B(A+C) + D’C’BA + D’C’BA’ + DB + C’B’A’ and its optimized version.

2.71 Using the mapping method, optimize this equation and produce a simplified version (Y) of it.

A transparent latch will not work properly for sequential circuits like shift registers and counters. A real D flip-flop will do the job. A true D flip-flop only latches the data on the D input during a clock transition from low to high. In this exercise you will simulate a D flip-flop. The circuit diagram for a true D flip-flop is



2.63 AA’, Also by the inverse law.

A True D flip-flop

2.72 What is the boolean formula for F?

2.73 What is the boolean formula for G?

2.74 What is the boolean formula for H?

2.75 What is the boolean formula for I?

2.76 What is the boolean formula for X?

2.77 What is the boolean formula for Y?

2.10.6 Laboratory Exercises

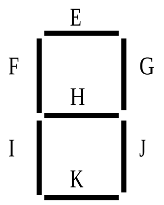
In this laboratory you will perform the following activities:

- Demonstrate the use of the LOGICEV.EXE program.
- Input and evaluate several logic functions, generate their truth tables, and verify their correctness
- Build logic equations for combinatorial and sequential circuits and then simulate those circuits.

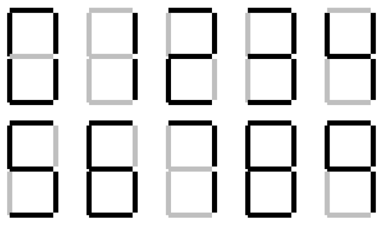
❏ Exercise 1: Sample LOGICEV session. Connect your breadboard circuit to the PC's parallel port and run the LOGICEV program. Press Enter← to get past the opening screen. Enter the boolean function Z=AB when LOGICEV asks you to enter a function. Press "Y" to see the truth table and canonical form for this function. After verifying the truth table, press Enter← again to return to the function entry mode. Enter the function Y=ABC and verify the truth table. Next, enter the function X=ABCD and look at its truth table and canonical form. Finally, enter the function W=A+B+C+D#' and look at its truth table. When LOGICEV asks for the next function, just press the Enter← key to end the function entry mode. You do not need to initialize any variables, so just hit the Enter← key again when you reach the initialization screen. The press "H" for hardware mode. Now LOGICEV should be ready to simulate these logic functions. Cycle D, C, B, and A through the 16 possible combinations of these variables and press the Enter← key for each value. Observe the results on the LEDs and compare these results against the truth tables for the W, X, Y, and Z functions. Press the ESC key after cycling through the 16 switch combinations.

For your lab report: Be sure to include the truth tables and canonical forms for each of the functions. Include a drawing of the LEDs' states for each of the 16 switch settings.

❏ Exercise 2: A Seven-Segment Decoder. The LOGICEV program simulates a seven segment display on the PC's video screen. Variables F-K light the individual segments as follows:



Here are the segments to light for the binary values DCBA = 0000 - 1001:



Enter the seven equations for these segments into LOGICEV and try out each of the patterns (0000 through 1111).

For your lab report: copy the seven segments for each value between 0000 and 1111 into your report. Describe how this display works. **Optional, for additional credit:**

2.64 D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A

2.65 D'C'B'A'+ D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A

2.66 D'C'B'A' + D'C'B'A + D'C'BA' + D'C'BA + D'CB'A' + D'CBA + DC'B'A' + DC'B'A

2.67 D'C'BA' + D'C'BA + D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A

2.68 D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'

2.69 D'C'B'A' + D'C'B'A + D'C'BA + D'CB'A' + D'CB'A +D'CBA' + D'CBA + DC'B'A' + DC'B'A

2.70 D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + DC'B'A'

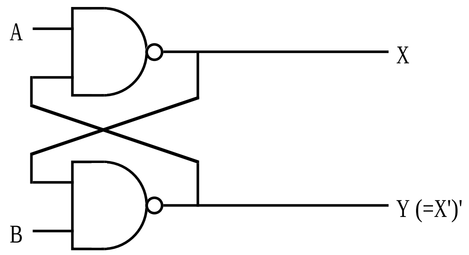
2.71 Y = B + C'

compute optimized forms for each of the above equations using the mapping method. Include these optimized forms in your lab report and use them as inputs to the LOGICEV program.

❏ Exercise 3: Equivalence of boolean functions. In this exercise you will enter two (different) boolean expressions that are equivalent to one another. You will then provide all 16 possible inputs to these two functions and verify that they both produce identical outputs. The function you will use is X=A(B+C)’ + D’B(A+C) + D’C’BA + D’C’BA’ + DB + C’B’A’ and its optimized version (see the questions immediately before the lab exercises).

Run the LOGICEV program and enter these two equations (X and Y). In the simulation mode, cycle the A-D inputs through the values 0000-1111. Verify that the LEDs corresponding to the X and Y outputs are always in the same state. **For your lab report**: Create a truth table for X and Y and provide the states of the LEDs for each input set. Using LOGICEV, determine the canonical form of these two equations and provide that in your lab manual. **For additional credit:** Create some additional pairs of equivalent functions and try them out.

❏ Exercise 4: A simple sequential circuit. For this exercise you will enter the logic equations for a simple set / reset flip-flop. The circuit diagram is



A Set/Reset Flip-Flop

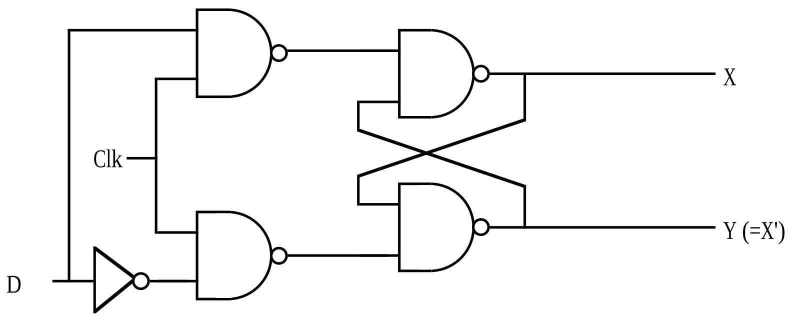
Since there are two outputs, this circuit has two corresponding logic equations. They are

```
X = (AY)’
Y = (BX)’
```

These two equations form a *sequential circuit* since they both use variables that are function outputs. In particular, Y uses the previous value for X and X uses the previous value for Y when computing new values for X and Y.

Enter these two equations into LOGICEV. Set the A and B inputs to one (the normal or *quiescent* state) and run the logic simulation. Try setting the A switch to zero and press the Enter← key to determine what happens. Press the Enter← key several more times with A still at zero to see what happens. Then switch A back to one and repeat this process. Now try this experiment again, this time setting B to zero. Finally, try setting *both* A and B to zero and then press the Enter← key several times while they are zero. Then set A back to one and press Enter← again. Try setting both to zero and then set B back to one and press the Enter← key. **For your lab report:** provide diagrams for the switch settings and resultant LED values for each time you press the Enter← key.

❏ Exercise 5: A transparent latch. The circuit for a transparent latch takes the following form:



A Transparent Latch

Once again there are two outputs, so we will need two logic functions. Those logic functions are

```
X = (Y * (DC)’)’       Note the use of “C” rather than “#”.
Y = (X * (D’C)’)’
```

Although these logic equations are perfectly fine, it is a good idea to generate a separate logic function for every output (except inverters) that appears within a function. In the circuit above, there are four NAND gates, so you should really use four logic functions rather than two. This will make your logic systems easier to read and understand. Furthermore, it is easier to convert circuits to equations when you use a separate logic equation for every gate. We will use functions E and F for the two NAND gates on the left side of the diagram above. The four logic equations are

```
E = (DC)'
F = (D'C)'
X = (EY)'
Y = (FX)'
```

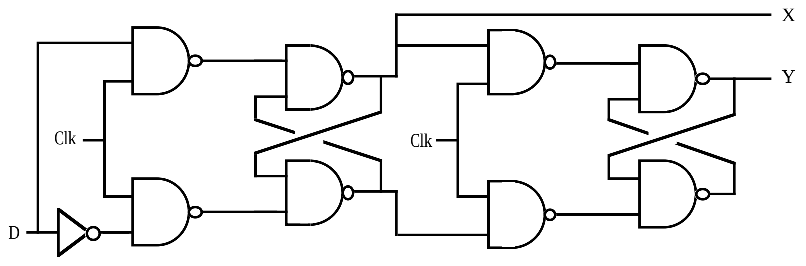
Another advantage to building the logic equations in this form is that LOGICEV will display the values of the intermediate computations on the PC's video display. Since you can see the intermediate values, it is easier to determine how the circuit is operating.

Note that these logic equations use the C variable for the clock rather than the "#" symbol (LOGICEV's clock). By specifying the clock in this manner, you get to control when the clock is up and when it is down. If you use LOGICEV's built-in clock, LOGICEV will only generate a pulse each time you press the Enter key. For this exercise, you want to keep the clock high or low across several circuit evaluations.

2.72 E = (DC)'
F = (D'C)'
X = (EG)'
G = (XF)'
H = (XC)'
I = (GC)'
Y = (HJ)'
J = (IY)'

To try out the transparent latch, set C to zero and D to any value. Press the Enter key several times. Record the X and Y outputs in your lab report. Next, set C to one and press the Enter key. Once again, record the outputs in your lab report. Now, set the C input to zero and flip the value of your D input. Press the Enter key. Record the results in your lab report and explain them. Now set the C input to one and press the Enter key again. Describe the result in your lab report. Set C back to zero and press Enter. Try setting D to different values and press the Enter key without changing C's value; explain the results in your lab report. Now, set the C input to one and press the Enter key several times, toggling D each time. **For your lab report:** Explain the result in your lab report. Provide diagrams of the switch settings and LED outputs for each evaluation of this circuit.

❏ Exercise 6: A simple shift register. **Note: this circuit will not work properly.** In this exercise you will build a simple two-bit shift register using two transparent latches. One purpose of this exercise is to demonstrate that you cannot build a shift register using only transparent latches. The circuit for this (broken) shift register is



An Attempt at a Shift Register using Two Transparent Latches

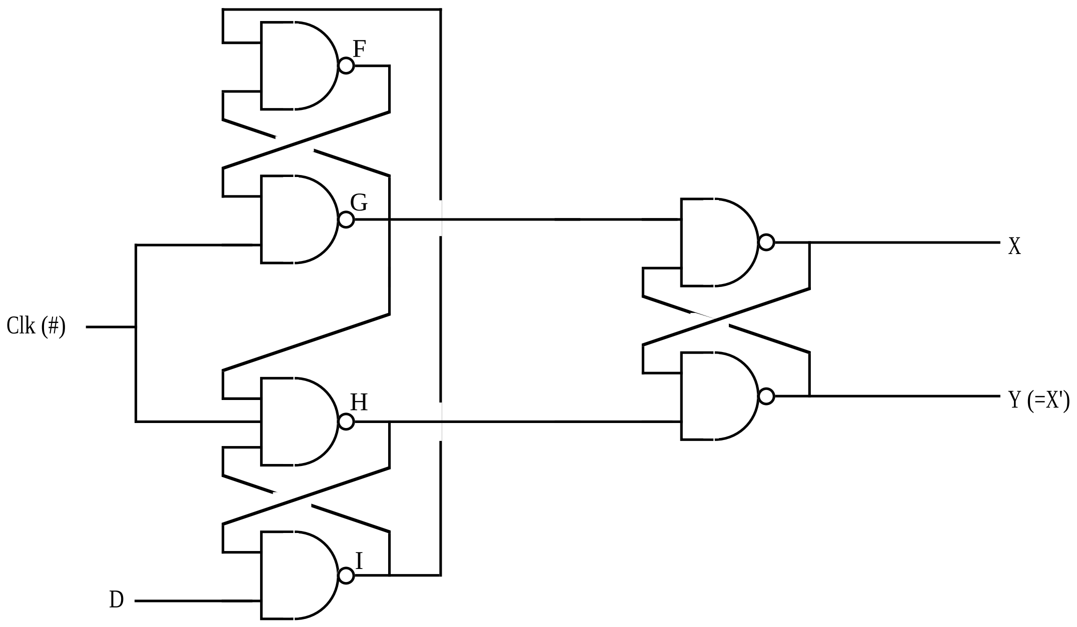
In theory, toggling the clock should shift X's value into Y and D's value into X. To see what really happens, enter the following equations for this circuit into LOGICEV for simulation.

2.73 F = (IG)'

```
E = (DC)'          F = (D'C)'
X = (EG)'          G = (XF)'
H = (XC)'          I = (GC)'
Y = (HJ)'          J = (IY)'
```

Enter these equations into LOGICEV using the C input for the clock in the above diagram. Set the D input to one, the C input to zero, and then press the Enter key. Next, set the C input to one and press the key. Finally set the C input back to zero and press the Enter key. Describe what happens in your lab report. Repeat this simulation with D equal to zero. Describe what went wrong. **Optional, for additional credit:** describe *why* it went wrong.

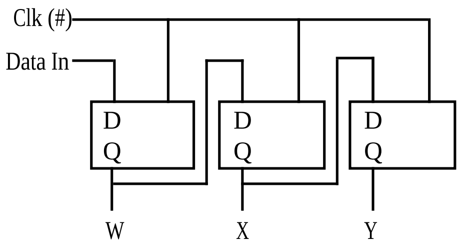
❑ Exercise 7: A true D Flip-Flop. A transparent latch will not work properly for sequential circuits like shift registers and counters. A real D flip-flop will do the job. A true D flip-flop only latches the data on the D input during a clock transition from low to high. In this exercise you will simulate a D flip-flop. The circuit diagram for a true D flip-flop is



A True D flip-flop

Repeat the exercises from exercise 5 for this true flip-flop (the equations for this flip-flop appear in the the prior section).

❑ Exercise 8: A true three-bit shift register. In this exercise you will build a three-bit shift register using the logic equations for a true D flip-flop. To construct a shift register, you connect the outputs from each flip-flop to the input of the next flip-flop. The data input line provides the input to the first flip-flop, the last output line is the "carry out" of the circuit. Using a simple rectangle to represent a flip-flop and ignoring the Q' output (since we don't use it), the schematic for a four-bit shift register looks something like the following:



A Three-bit Shift Register Built from D Flip-flops

In exercise seven, you used six boolean expressions to define the D flip-flop. Therefore, we will need a total of 18 boolean expressions to implement a three-bit flip-flop. These expressions are

```
Flip-Flop #1:

      W = (GR)’
      F = (IG)’
      G = (F#)’
      H = (G#I)’
      I = (DH)’
      R = (HW)’
Flip-Flop #2:

      X = (KS)’
      J = (MK)’
      K = (J#)’
      L = (K#M)’
      M = (WL)’
      S = (LX)’
Flip-Flop #3:

      Y = (OT)’
      N = (QO)’
      O = (N#)’
      P = (O#Q)’
      Q = (XP)’
      T = (PY)’
```

2.74 G = (#F)’

2.75 H = (G#I)’

2.76 I = (DH)’

Enter these equations into LOGICEV. Initialize W, X, AND Y to zero. Set D to one and press the Enter key once to shift a one into W. Now set D to zero and press the Enter key several times to shift that single bit through each of the output bits. **For your lab report**: try shifting several bit patterns through the shift register. Describe the step-by-step operation in your lab report.

2.77 X = (GY)’

For additional credit: Describe how to create a *recirculating shift register*. One whose output from bit four feeds back into bit zero. What would be the logic equations for such a shift register? How could you initialize it (since you cannot use the D input) when using LOGICEV?

2.78 Y = (HX)’

❏ Exercise 9: (**Optional, for additional credit**): Design a three-bit counter. Test the counter using LOGICEV. Provide the logic equations and describe the operation of your counter in your lab report.

❏ Exercise 10: (**Optional, for additional credit**): Design some combinatorial or sequential circuits of your own. Simulate the circuit using LOGICEV. Provide the schematic(s), logic equation(s), and a description of the circuit operation (under LOGICEV) in your lab report.

2.11 Programming Projects

- ❏ Program #1: Using a high level language like C++, Pascal, Ada, etc., write a short program that reads four boolean values from the user (input can be zero or one that you convert to true/false, if necessary). Evaluate several equivalent (though different) boolean expressions using these input values. Your program should verify that you get the same result for each input function.
- ❏ Program #2: Write a program to evaluate the following functions of three variables using the generic logic function given in the textbook.

  F = CB'A' + CB'A + C'B'A + CB + CA

  G = C'B'A' + CBA + C'BA' + CB'A

  H = CBA + C'B'A' + C'BA + CB'A'

  As in program #1, read the input values for A, B, and C from the user and display the result for each function.
- ❏ Program #3: Write a short high level language program to demonstrate each of the boolean logic theorems presented in this chapter.
- ❏ Program #4: Write a short program that inputs a value between zero and nine from the user and computes the functions S_0 through S_6 for the seven segment display. Print the results (zero or one) for each of these functions.
- ❏ Program #5: Write a short program that simulates the operation of a transparent latch using logic expressions. Your program should read an input value from the user (zero or one) for the data input. After reading (and checking the validity) of each input, your program should "toggle" the clock. Don't forget to evaluate each logic function for the transparent latch *n* times, where *n* represents the number of gates in the circuit, on each transition of the clock (i.e., evaluate the function *n* times when you set the clock high, evaluate them *n* times when you bring it back low.
- ❏ Program #6: Write a short program that simulates a four-bit shift register using a logic function implementation. Your program should read an input value from the user and then toggle the clock line high, then low. Don't forget to evaluate the functions *n* times, *n* being the number of gates in the circuit. Also, build your shift register using a true D flip-flop, not a transparent latch.
- ❏ Program #7: Write a short program that simulates a four-bit counter using logic expressions. Build your counter using the equations for a true D flip-flop. Each time you toggle the clock signal, the counter should increment by one. Don't forget to evaluate the logic expressions *n* times (*n* being the number of gates in the circuit) on each transition of the clock.
- ❏ Program #8: Write a short program that uses a complex boolean expression (perhaps to control a WHILE loop) that you can simplify using one of DeMorgan's theorems. Time the execution of the program (be sure that you execute this expression in a FOR loop so you can control how long it takes to execute so you can easily time it). Reduce the expression using DeMorgan's theorem and time the execution again. Compare the two times. **Note: if your compiler provides optimization control, be sure to turn all optimizations off.**

2.12 Answers to Selected Exercises

1b) 0 1d) none

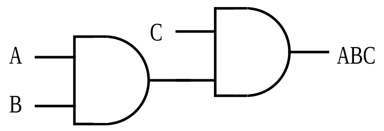
2a)

AND	0	1
0	0	0
1	0	1

3a)

CBA	B'A'	B'A	BA'	BA
C'	0	0	0	0
C	0	0	0	1

4a) A) ABC =



8) (F_x function) $F_x = 0EA_{16}$ or 234_{10}.

9a) Four possible functions of one input (0, 1, A, A').

10a) F	=	AB + AB')	
	=	A(B+B')	Distributive law
	=	A(1)	Inverse Law
	=	A	Th 4

11a)

	A'	A
B'	0	1
B	0	1

Note the cluster of ones on the right hand side of the truth map. They cross B/B', so this leaves us with F=A.

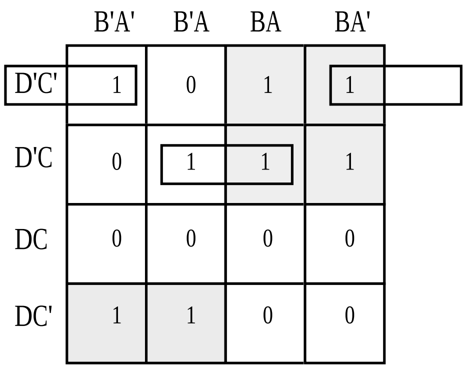
12) S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A

13) S_0:

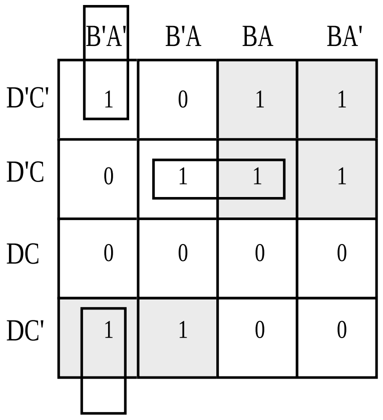
S_0	B'A'	B'A	BA'	BA
D'C'	1	0	1	1
D'C	0	1	1	1
DC'	1	1	0	0
DC	0	0	0	0

14) S_0 :

Two Possible Solutions:



	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	0	1	1	1
DC	0	0	0	0
DC'	1	1	0	0



	B'A'	B'A	BA	BA'
D'C'	1	0	1	1
D'C	0	1	1	1
DC	0	0	0	0
DC'	1	1	0	0

S_0 = D'B + D'C'A' + D'CA + DC'B'

S_0 = D'B + D'C'A' + C'B'A + DC'B'

18a) if (x or (not x and y)) then write('1');
becomes
if (x or y) then write('1');

19a) F(A,B,C) = A'BC + AB + BC
= CBA' + CBA + C'BA + CBA + CBA'
= CBA' + CBA + C'BA

20a) F(A,B,C) = (C' + B' + A')(C' + B' + A)(C' + B + A')(C + B' + A')(C + B' + A)
(Conversion by building truth table for 19a above and using entries containing zeros.