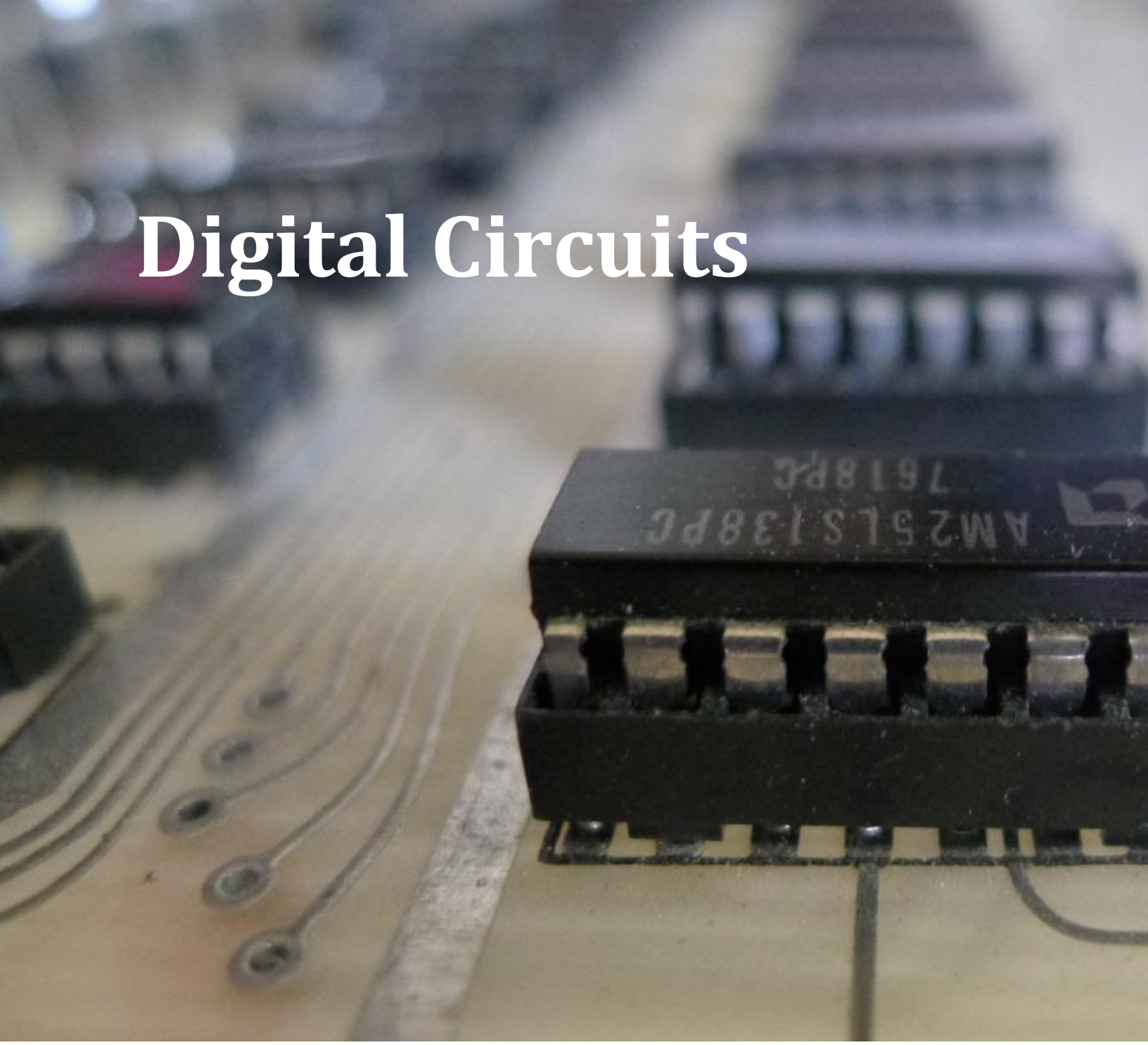
Digital Circuits



DALHOUSIE UNIVERSITY
Inspiring Minds

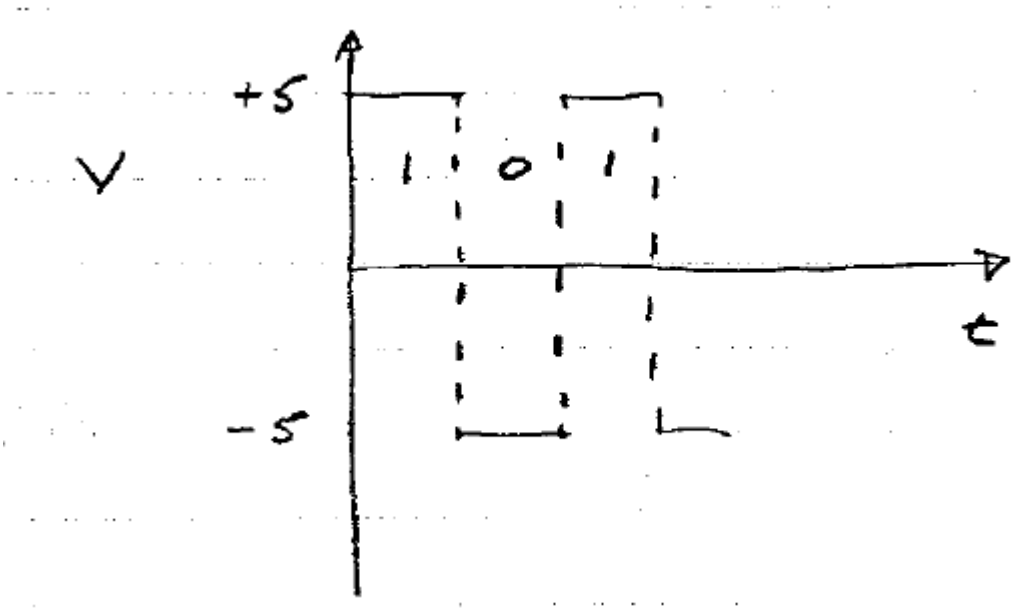
Electrical & Computer Engineering Department (ECED) Course Notes

ECED2200

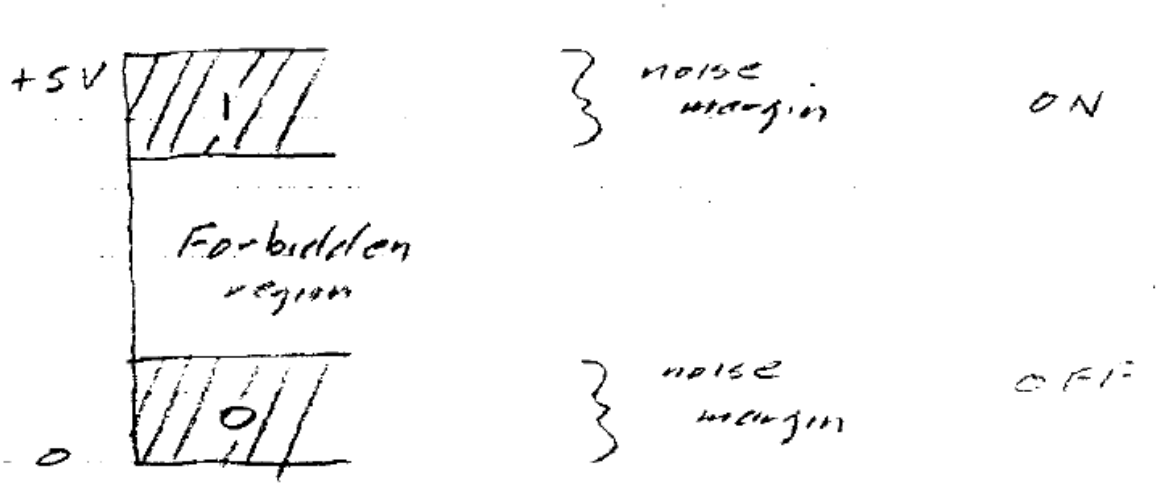
Table of Contents

Digital Circuits

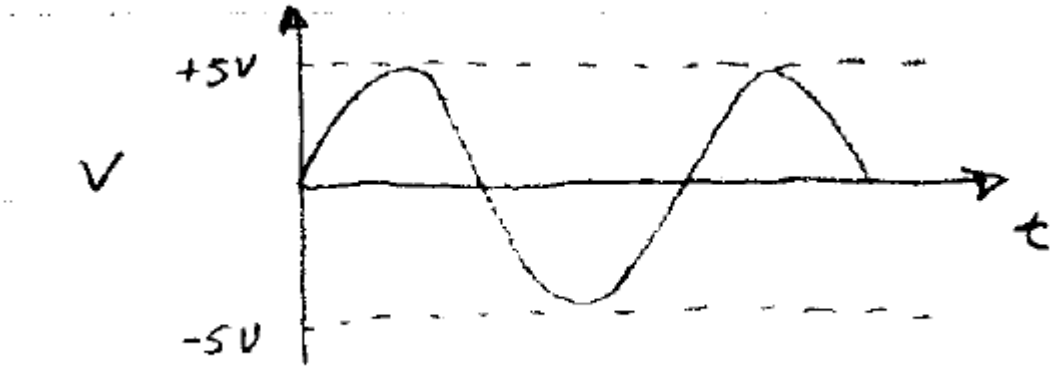
Digital Circuits have inputs and outputs that are represented by discrete values. The figure below shows a typical output for a digital circuit.



There are two possible output values, namely ±5 volts. Two distinct voltage levels separated by a forbidden region electronically represent the binary numbers 1 and 0.



In analog circuits the inputs and outputs have continuous values as show below:



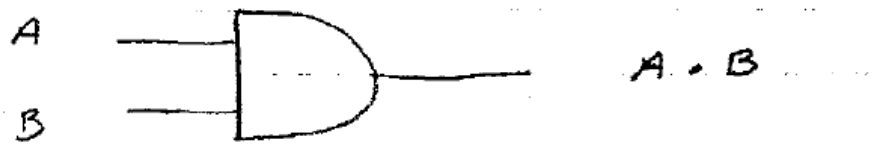
Analog waveform more realistically represents physical quantities such as sound and temperature. Digital waveforms only approximate real values if there are many discrete values. Digital waveforms, however, can best represent degraded signals.

Logic Gates

A *gate* is a device that controls the flow of information, usually in the form of pulses. Each logic operation will be indicated by a symbol whose function is defined by a truth table that shows all possible inputs and the corresponding outputs.

AND Gate

Symbol



A ● B is read “A and B”.

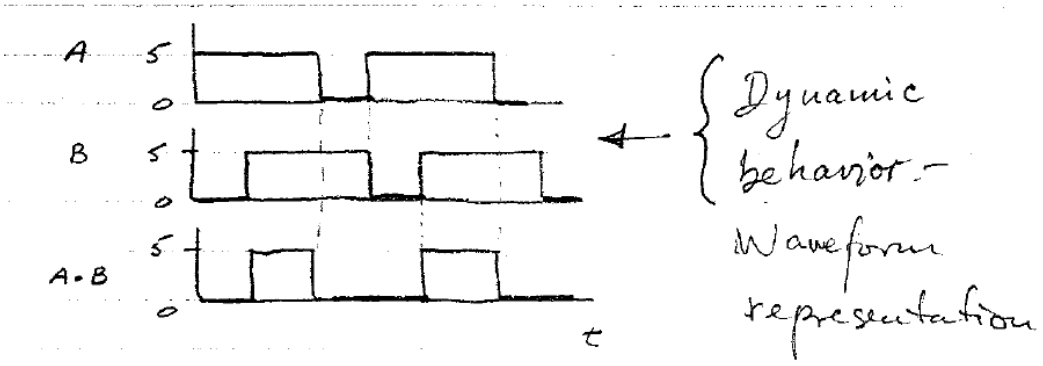
Truth Table

A	B	A • B
0	0	0
0	1	0
1	0	0
1	1	1

An output appears only when there are inputs at A and B. In general, there may be several input terminals.

Typical Response

A typical response for two inputs varying with time is shown below:



OR Gate

Symbol

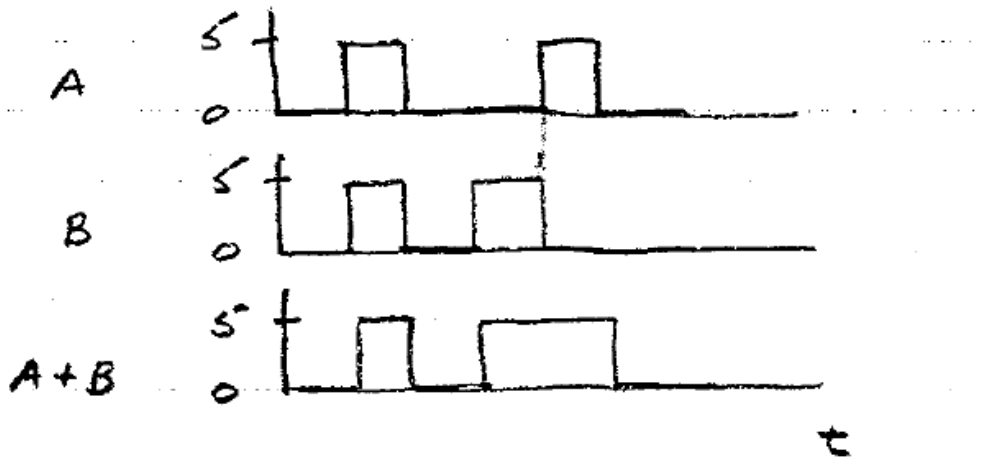
A + B is read “A or B”.

Truth Table

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

Typical Response

A typical response for two inputs varying with time is shown below:



NOT Gate

Signal inversion corresponds to a logic NOT.

Symbol

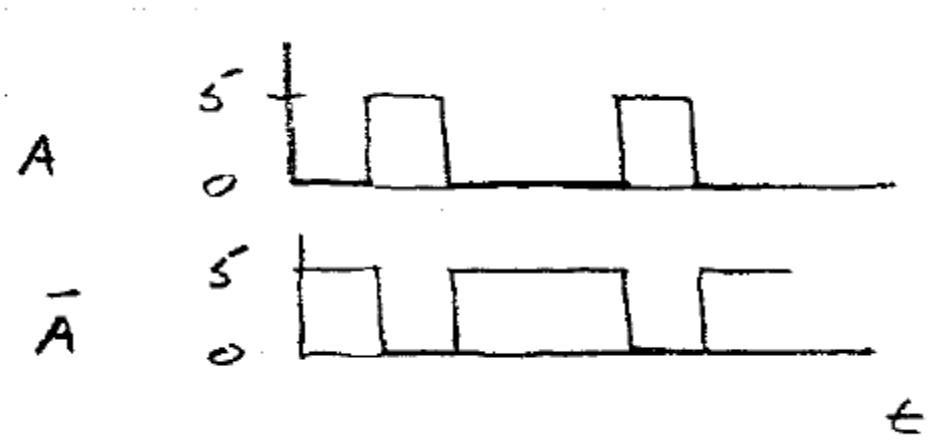
$\overline{A}$ is read “not A”.

Truth Table

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

The NOT element is an inverter; the output is the complement of the signal input.

Typical Response



NOR Gate

An inverted OR gate results in a NOT OR or NOR operation.

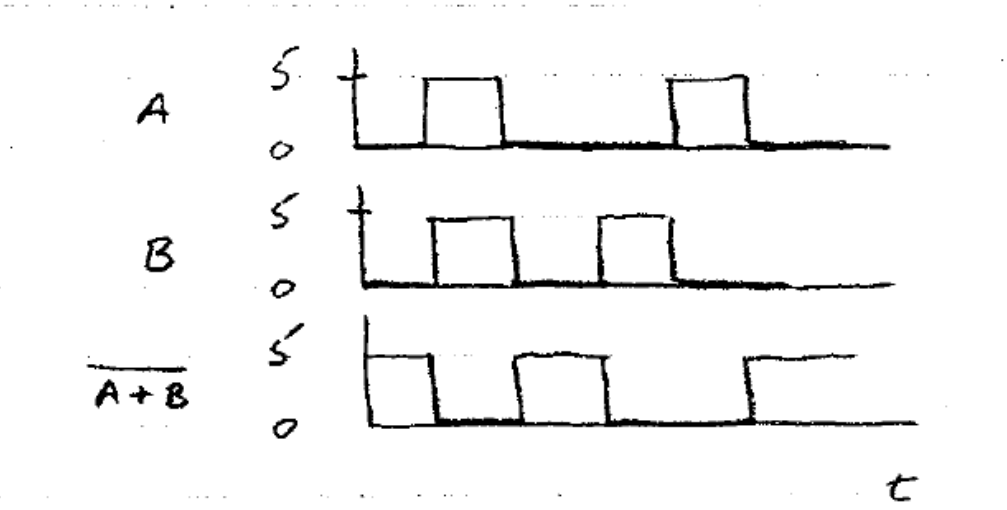
Symbol

The small circle at the output of the gate, and the line over A + B indicate the inversion process. Thus $\overline{A+B}$ is A+B inverted.

Truth Table

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

Typical Response

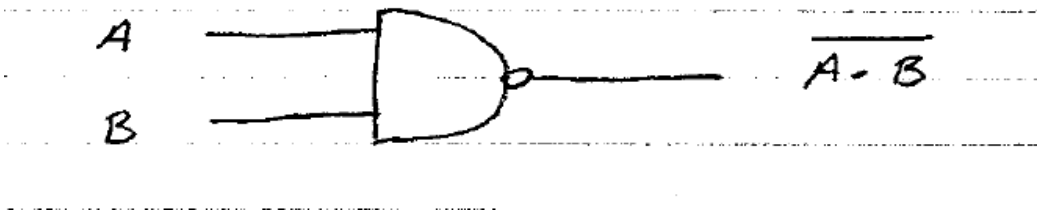


All basic logic operations can be achieved by using only NOR gates.

NAND Gates

An inverted AND gate results in a NOT AND or NAND operation. A NAND gate has all the advantages of a NOR gate and is very easy to fabricate. In a complex logic system, it is convenient to use one type of gate, even when simpler types would be satisfactory, so that gate characteristics are the same for the whole system.

Symbol

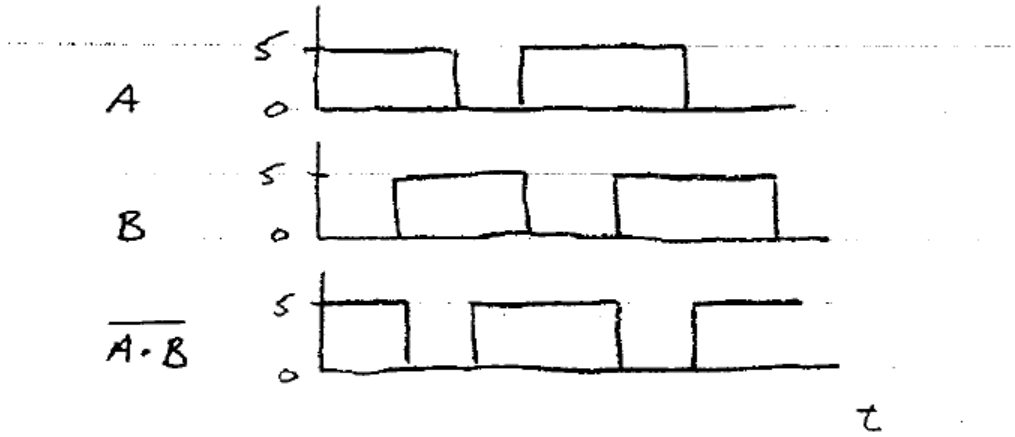


The small circle and the line over A ● B indicate inversion.

Truth Table

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

Typical Response



Three NAND gates can be used to replace an OR gate. The combination of NAND gates is equivalent to an OR gate in that it performs the same logic operation (see example below).

Example:

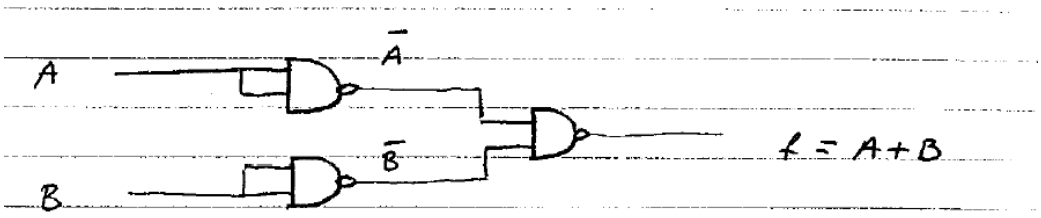
Use NAND gates to form a two-input OR gate.

The desired function is defined by the following truth table:

desired			implementation		
A	B	f	$\bar{A}$	$\bar{B}$	$\overline{\bar{A}\cdot\bar{B}}$
0	0	0	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	1	0	0	1

From the table we see that if each input were inverted (replaced by its complement) the NAND gate would produce the desired result as indicated in the table.

To obtain the inversion, tie both terminals of a NAND gate together as shown below.



In digital notation, the function f is defined by:

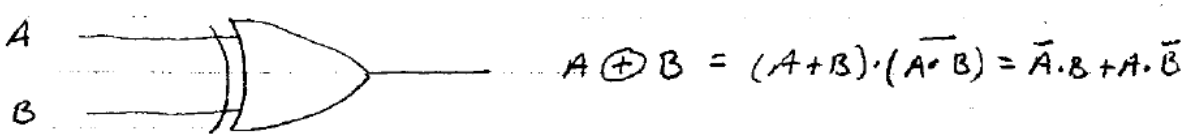
$$f = \overline{\overline{A}\bullet\overline{B}} = A+B$$

This relation was obtained by comparing the desired and available truth tables. A "digital algebra" for direct manipulation of such expressions will be considered later.

XOR Gate

As indicated by the truth tables, the Exclusive-OR operation can be expressed as $(A+B)\bullet(\overline{A\bullet B})$ which reads "(A or B) and not (A and B)". The alternate form $\overline{A}\bullet B + A\bullet\overline{B}$ is called an *inequality comparator* since it provides an output of one if A and B are not equal.

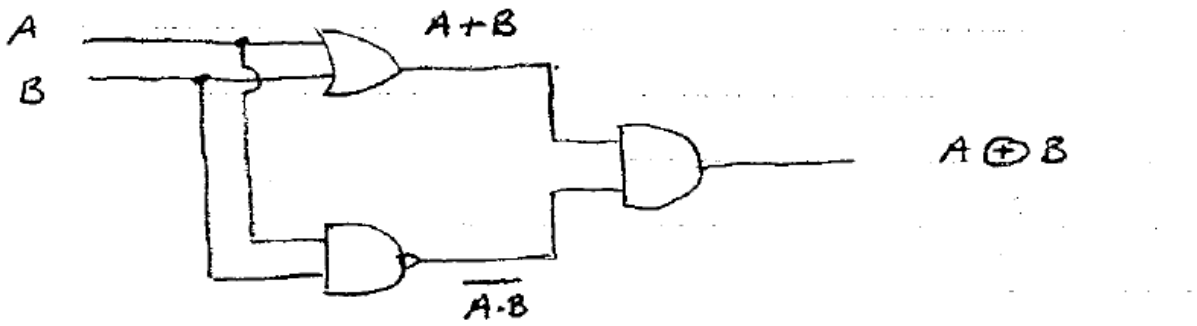
Symbol



Truth Table

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

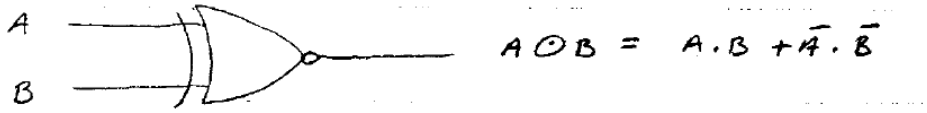
One realization of this gate is shown below:



X NOR Gate

The exclusive-NOR operator can be expressed as (see the truth tables) $A \cdot B + \overline{A} \cdot \overline{B}$. It is the inverse of the inequality comparator $\overline{\overline{A} \cdot B + A \cdot \overline{B}}$. This is an "equality comparator" since the output is 1 if A and B are equal.

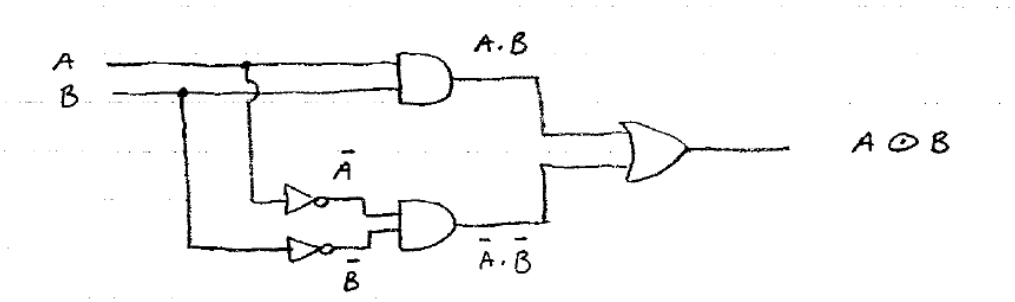
Symbol



Truth Table

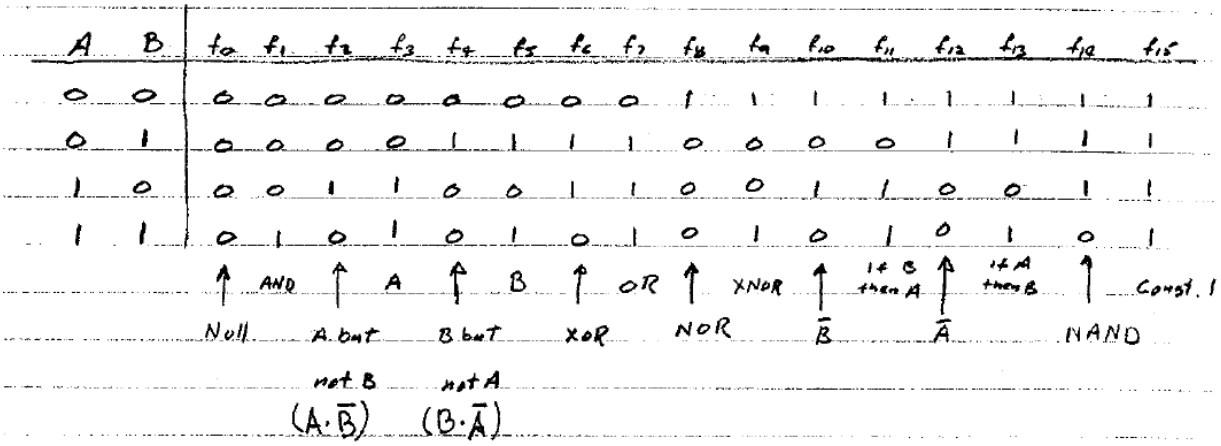
A	B	A ⊙ B
0	0	1
0	1	0
1	0	0
1	1	1

One realization of this gate is shown below:



Additional Gates

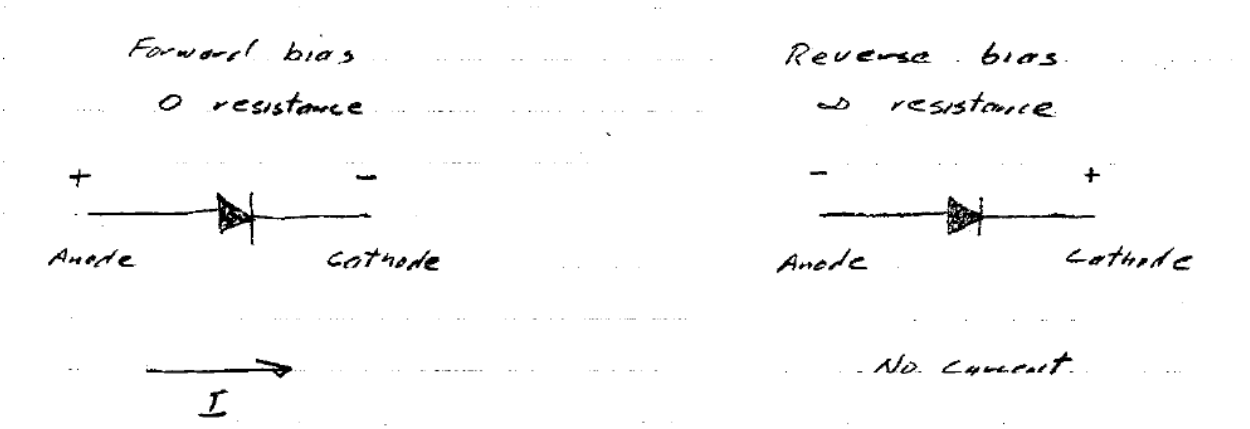
The following truth table shows all possible gates. This is based on writing out all possible variations of the truth table, with names for some of those gates given:



Electric Switches

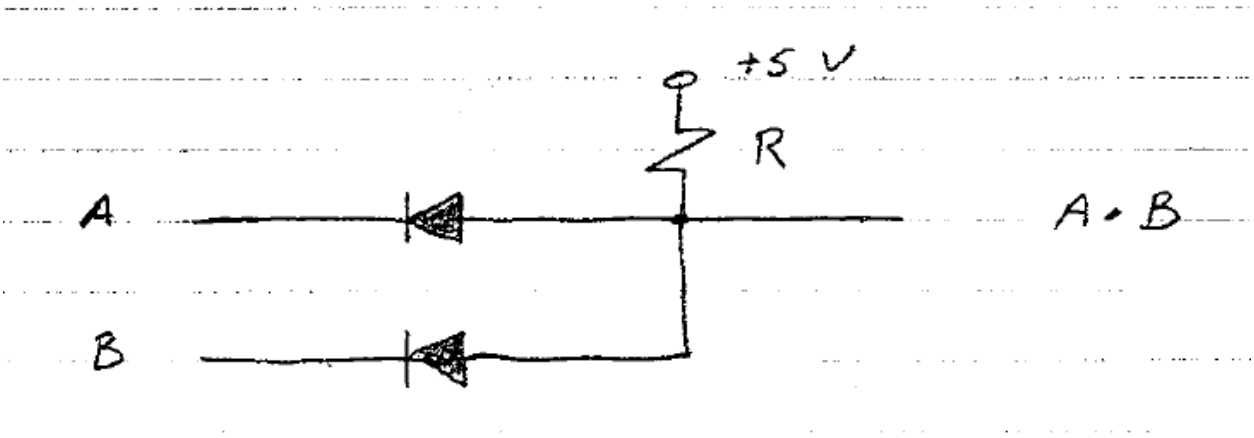
Diodes

A diode is a two-terminal electrical device that allows current to flow in one direction but not the other. A schematic diagram from a diode is shown below.



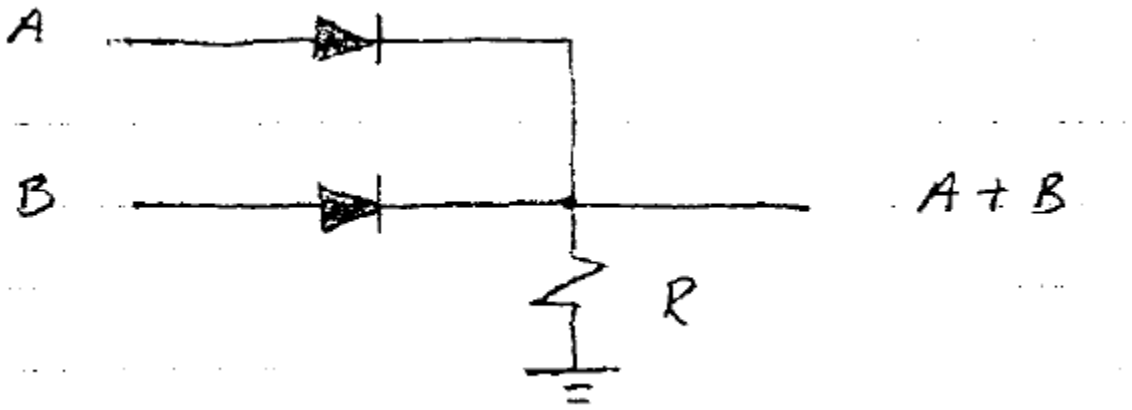
If the anode is at a higher voltage than the cathode, the diode is *forward biased*, its resistance is very low, and current flows. The diode has voltage drop of about 0.7V across it. If the anode is at a lower voltage than the cathode, the diode is *reverse biased*, its resistance is very high, and no current flows.

Simple gates can be constructed by using diodes and a resistor. An AND gate is shown below:



If the inputs are positive (> +5V) with respect to ground, inputs at A and B turn off both diodes, no current flows through R and there is a positive output (a 1). In general, there may be several input terminals. If any of those inputs are zero (0), current flows through the forward-biased diode, and the output is nearly zero (0).

An OR gate is shown below:



For no input (zero voltage) no current flows and the output is zero (0). An input of +5V (1) at either terminal A or B on both (or on any terminal in the general case) forward biases the corresponding diode, current flows through the resistor, and the output voltage rises to nearly 5V (1).

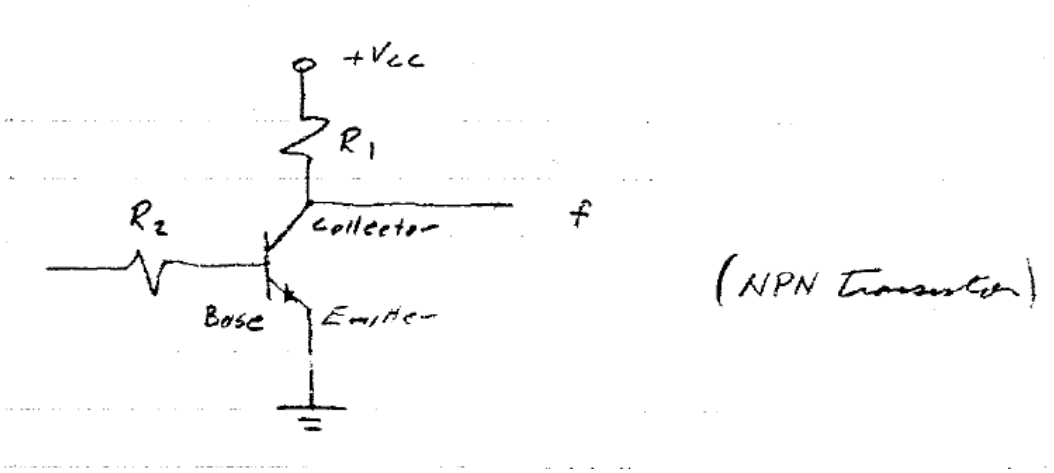
The voltage drops across the diodes add up when circuits of this type are cascaded in series and the voltage levels are degraded. Note that it is not possible to construct an inverter using only diodes and resistors. Transistors can be used to circumvent these problems.

Transistors

Bipolar

A bipolar transistor is a three-terminal semiconductor device. Under control of one of the terminals, called the *base*, current can below from the *collector* terminal to the *emitter* terminal.

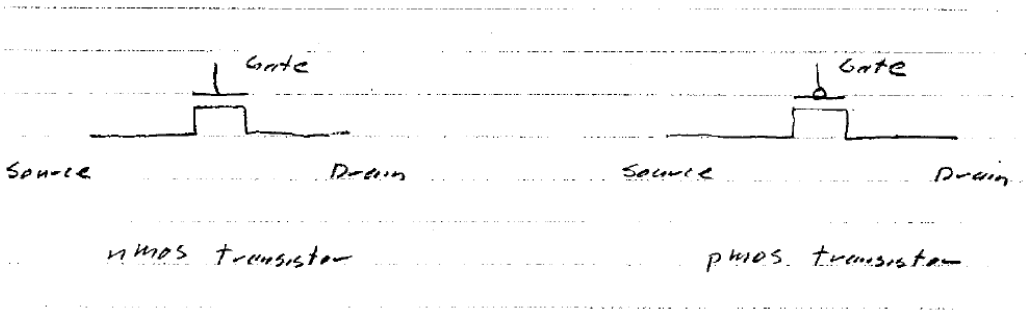
The basic inverter circuit is shown below.



A high voltage at the base turns on the transistor. The output f is discharged to ground, getting close to 0V (but never quite reaching it). When a low voltage is placed on the base, the transistor is turned off. The output node f voltage approaches the power supply voltage Vcc through the pull-up/load resistor R1.

Metal Oxide Semiconductor

A Metal Oxide Semiconductor (MOS) transistor is a voltage-controlled switch. It has three terminals: a source, a drain, and a gate. There are two different types of MOS transistors, called NMOS and PMOS. Their schematic symbols are shown below:



An NMOS transistor conducts when a high voltage (1) is placed on its gate, and is non-conducting when a zero voltage (0) is on the gate. The PMOS transistor is complementary. A PMOS transistor conducts when a logic 0 is placed on the gate, and is non-conducting when a logic 1 is on the gate.

Logic Classifications

Diodes, transistors, and resistors can be used to implement a wide range of gates. Electronic logic circuits are classified in terms of the components employed. Basic operations can be performed by:

1. Diode Logic (DL)
2. Resistor-Transistor Logic (RTL)
3. Diode-Transistor Logic (DTL)
4. Transistor-Transistor Logic (TTL)
5. Metal-Oxide Semiconductor (MOS)
6. Complementary MOS (CMOS)
7. Emitter-Coupled Logic (ECL)

Logic types vary in (a) signal degradation (b) fan-in (c) fan-out and (d) speed.

Signal Degradation:

As mentioned earlier, a disadvantage of diode logic is that the forward voltage drops is appreciable, and the output signal is degraded. The use of transistors minimizes degradation.

Fan-In:

The number of inputs that can be accepted is called fin-in. It is low (3 or 4) for DL and high (8 or 10) for TTL.

Fan-Out:

The number of outputs that can be supplied by a logic element is called the fan-out. Fan-out depends on the output current capacitor (and the input current requirement) and varies from 4 in DL to 10 or more in TTL.

Speed:

The speed of a logic operation depends on the time required to change the voltage levels, which is determined by the effective time constant of the element. In high speed diodes, the charge storage is so low that response is limited primarily by wiring and lead capacitance. In transistors in the ON stat, base current is high and the

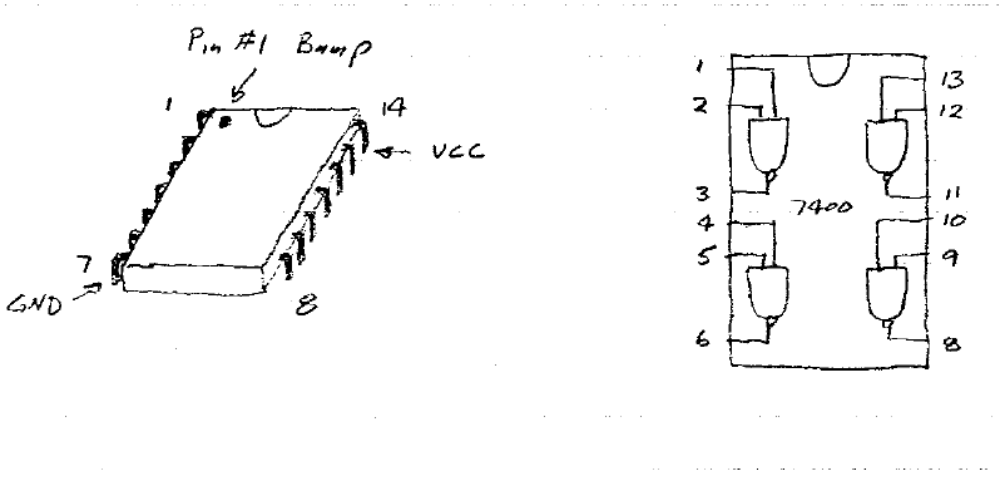
charge stored in the base region is high. This charge must be removed before the collector bias can reverse. Typically, 5 to 10 nS are required to process a signal. In ECL, the charge stored is minimal and ECL gates can operate at rates up to 200 MHz.

Noise Margin:

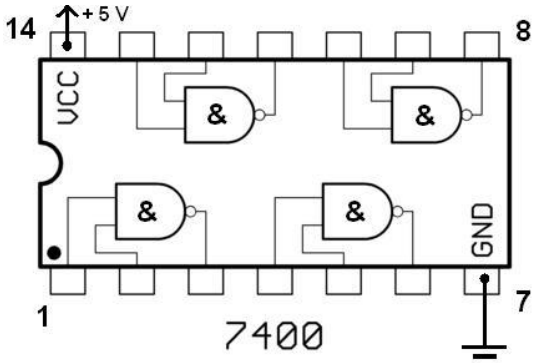
The difference between the operating input voltage and the threshold voltage is called the noise margin.

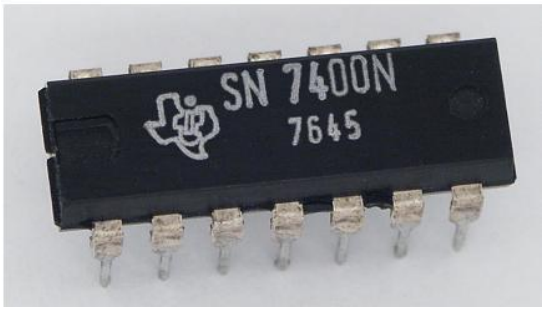
TTL Packaged Logic:

Integrated Circuits containing few than a dozen gates are small-scale integration (SSI); those with more than a hundred elements are large-scaled integration (LSI). In between are medium-scale integration (MSI) circuits. A TTL integrated circuit package typically contains several simple logic gates. The Texas Instruments (TI) 74-series components provide the standard number scheme used by the industry. For example, a package containing four 2-input NAND gates is a “7400” while a “7404” contains six inverters. A 14-pin package along with a diagram of its internal logic and pin connectivity is shown below.

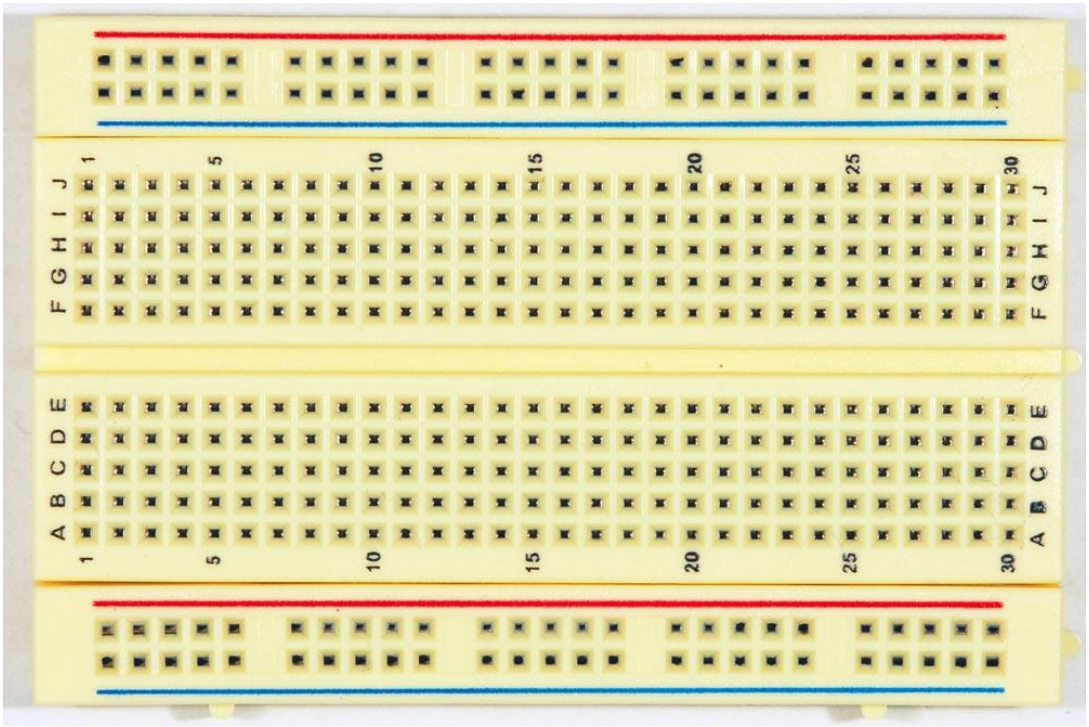


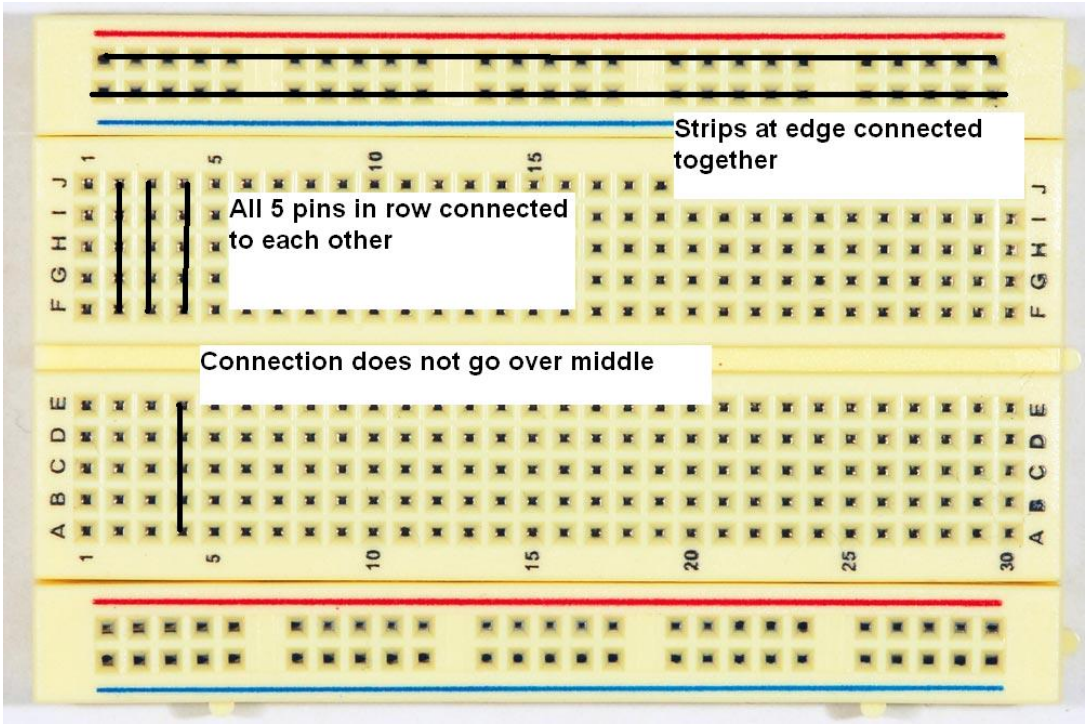
Another interpretation:





The Breadboard




Strips at edge connected together
All 5 pins in row connected to each other
Connection does not go over middle

Number Systems

To design efficient digital circuits, we need a special numbering system and a special algebra. We will now consider the binary number system and apply logic to binary relations.

Binary Numbers

A number N can be written as a polynomial of the form:

$$N = b_{n-1}r^{n-1} + b_{n-2}r^{n-2} + \ldots + b_1 r^1 + b_0 r^0 + b_{-1}r^{-1} + \ldots + b_{-m}r^{-m}$$

$$= \sum_{i=-m}^{n-1} b_i r^i$$

Where:

r = base or radius of the system

b_i = i^{th} bit (digit)

b_{n-1} = most significant bit (digit) MSB

b_{-m} = least significant bit (digit) LSB

n = number of integer bits (digits)

m = number of fraction bits (digits)

and

$$0 \le b_i \le r-1 \text{ for all i, } -m \le i \le n-1$$

In the decimal system a quantity is represented by the value and the position of a digit. For example, the number 503.14 can be written as:

$$500 + 0 + 3 + \frac{1}{10} + \frac{9}{100}$$

or

$$5 \times 10^{2} + 0 \times 10^{1} + 3 \times 10^{0} + 1 \times 10^{-1} + 9 \times 10^{-2}$$

We see that 10 is the base and each position to the left or right of the decimal point corresponds to a power of 10.

For data with only two possibilities such as the ON-OFF position of a switch which can be represented by the number 0 or 1, we use the binary system. In this system the base is 2. For example the number 10 can be written as:

$$1 \times 2^{3} + 0 \times 2^{2} + 1 \times 2^{1} + 0 \times 2^{0} = 1010$$

or

$$8 + 0 + 2 + 0 = 10$$

In electronics 1 and 0 usually correspond to the specified voltage levels e.g.: in TTL, 0 corresponds to a voltage near zero and 1 to a voltage near +5V.

Number Conversion

Binary to Decimal Conversion

In a binary number, each position to the right or left of the "binary point" corresponds to a power of 2, and each power of 2 has a decimal equivalent.

To convert a binary number to its decimal equivalent, add the decimal equivalents of each position occupied by a 1.

Example

Write in decimal the following numbers:

$110001 = 1\times2^5+1\times2^4+0\times2^3+0\times2^2+0\times2^1+1\times2^0 = 32+16+1 = 49$

$101.01 = 1\times2^2+0\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2} = 4+1+\frac{1}{4} = 5.25$

$01011 = 0\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0 = 8+2+1 = 11$

$0.0011 = 0\times2^0+0\times2^{-1}+0\times2^{-2}+1\times2^{-3}+1\times2^{-4} = \frac{1}{2^3}+\frac{1}{2^4} = \frac{1}{8}+\frac{1}{16} = 0.1875$

Decimal to Binary Conversion

A decimal number can be converted to its binary equivalent by expressing the decimal number as a sum of powers of 2. A more convenient method is the double-dabble method of handling integers and decimals separately.

To convert a decimal integer to its binary equivalent, progressively divide the decimal number by 2, noting the remainders; the remainder taken in reverse order forms the binary equivalent.

To convert a decimal fraction to its binary equivalent, progressively multiply the fraction by 2, removing and noting the carries; the carries taken in forward order from the binary equivalent.

Example

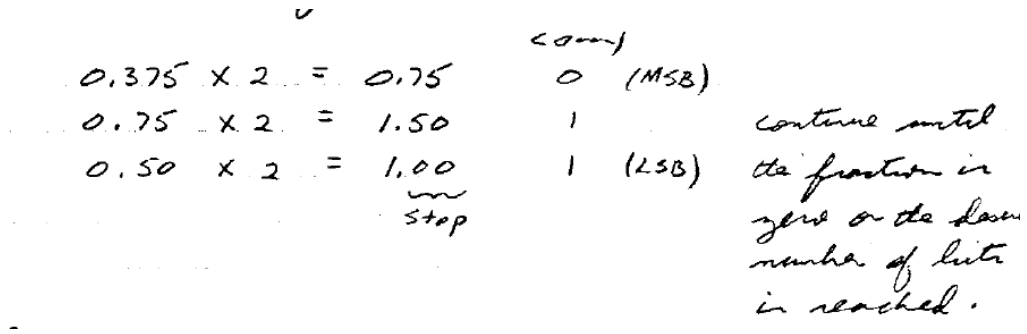
Convert decimal 28.375 and 0.625 to their binary equivalent.

A) Using the shorthand notation for the double-dabble method:

		remainder	
2	28		
	14	0	(LSB)
	7	0	continue until a zero quotient is obtained.
	3	1	
	1	1	
	0	1	(MSB)

The binary equivalent is 11100.

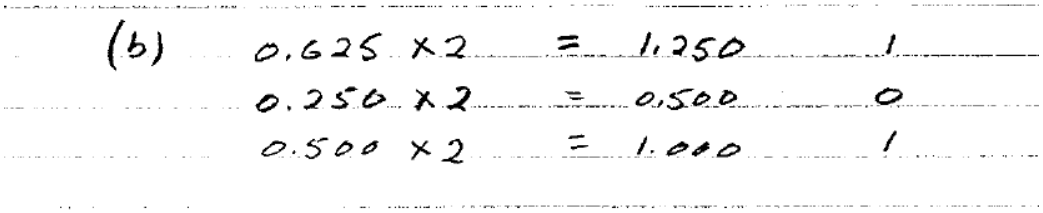
Then convert the fraction:



The binary equivalent is .011

Hence, 28.375 is equivalent to binary 11100.011:

B)



The binary equivalent is 0.101.

Binary Arithmetic

Binary Addition

Add column by column carrying where necessary into higher position columns.

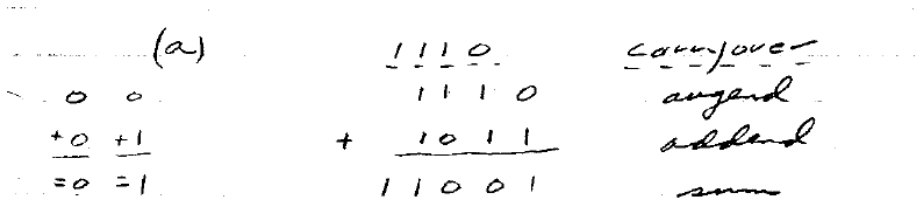
Examples

A) Perform 1110 + 1011

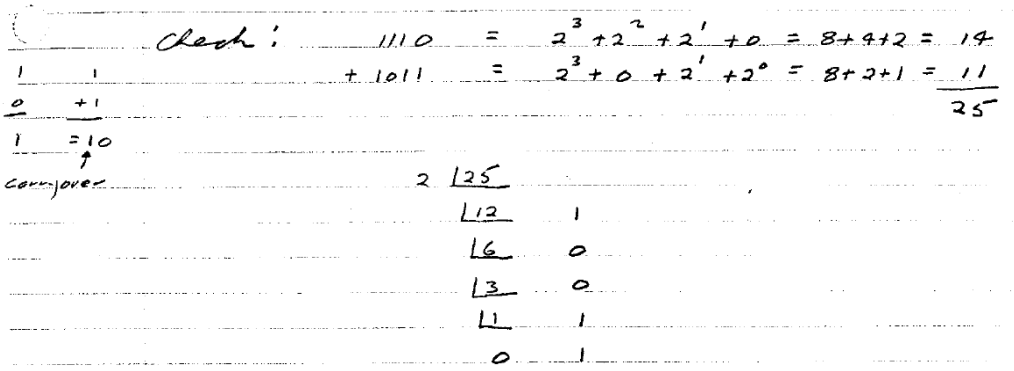
B) Perform 0110.110 + 0110.011

Results:

A)

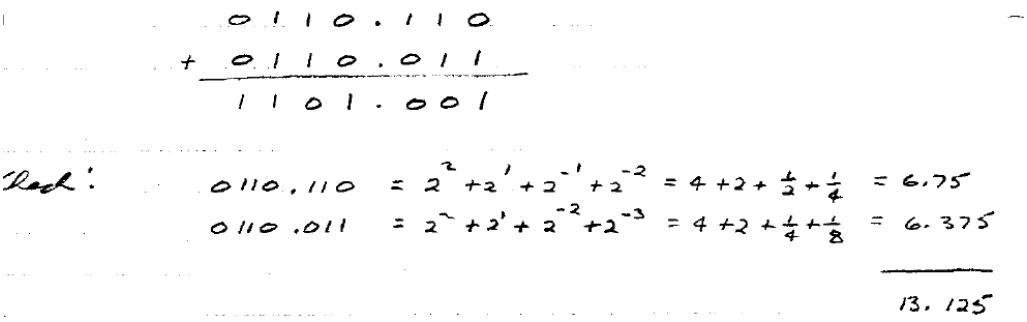


Check your results:

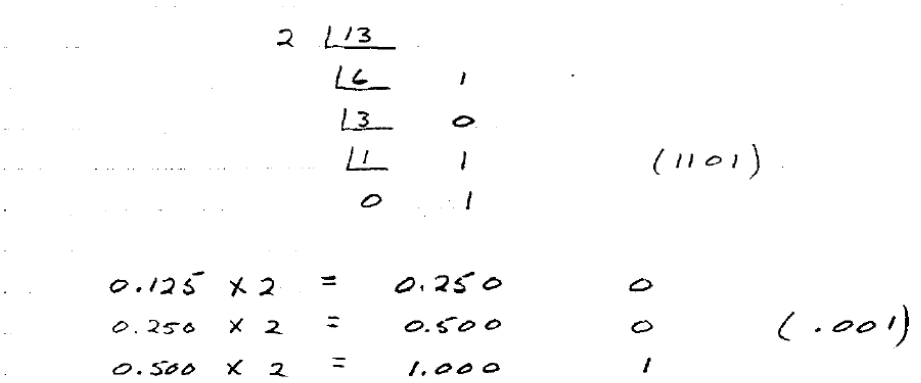


The binary equivalent is 11001 which checks OK.

B)



The binary equivalent is:



1101.001 is the binary equivalent of 13.125, which checks OK.

Binary Subtraction

Subtract column by column borrowing where necessary from higher position columns.

Example:

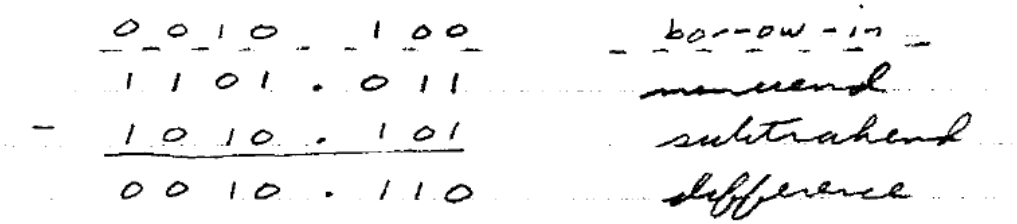
Perform the following binary subtractions:

A) 1101.011 – 1010.101

B) 1010 – 1101

Answers:

A)



Check:

$1101.011 = 2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = 8 + 4 + 1 + \frac{1}{4} + \frac{1}{8} = 13.375$

$-1010.101 = 2^3 + 2^1 + 2^{-1} + 2^{-3} = 8 + 2 + \frac{1}{2} + \frac{1}{8} = -10.625$

2.750

$2 \mid 2$

$1 \quad 0 \qquad (10)$

$0 \quad 1$

$0.750 \times 2 = 1.500 \qquad 1 \qquad (.11)$

$0.500 \times 2 = 1.000 \qquad 1$

Hence 10.110 is the binary equivalent to 2.75, so this answer checks out OK.

B)

$1010 = -1101$ subtract smaller

$-1101 \quad 1010$ from the larger

-0011 and change sign.

Check:

$-1101 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = -13$

$1010 = 2^3 + 2^1 = 8 + 2 = 10$

-3

$2 \mid -3$

$1 \quad 1 \qquad (-11)$

$0 \quad 1$

Hence, -11 is the binary equivalent of -3, so the answer checks out.

Binary Multiplication

Obtain partial products using the binary multiplication table:

0 x 0 = 0

0 x 1 = 0

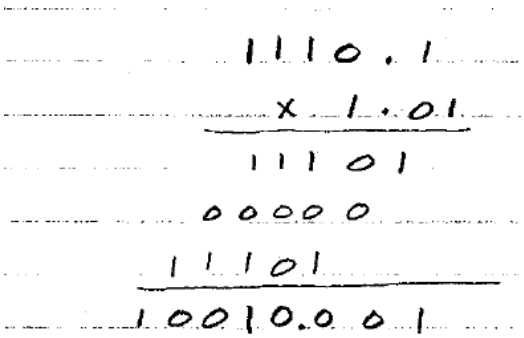
1 x 0 = 0

1 x 1 = 1

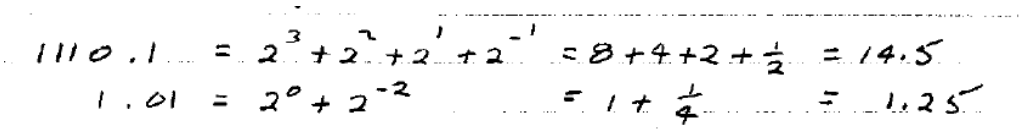
and then add the partial products. The binary point is handled in the same way a decimal point would be when multiplying.

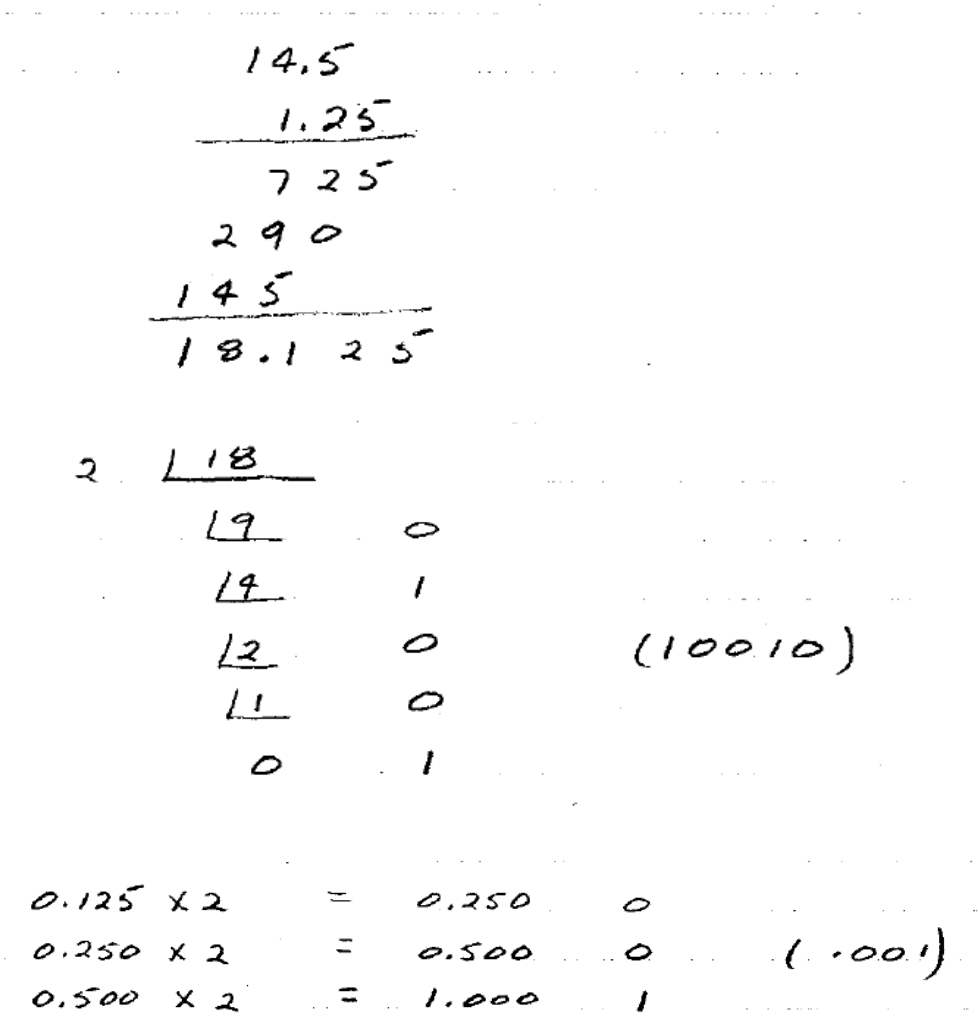
Example

Perform the following binary multiplication: 1110.1 x 1.01. Check by converting from binary to decimal and multiplying.



Check:





This shows that 10010.001 is the binary equivalent of 18.125, so the multiplication checks OK.

Binary Division

Perform repeated subtractions as in long division of decimals.

Example

Perform the following binary division: 10011.01 ÷ 11.1 . Check by converting from binary to decimal and then dividing.

$$11.1 \overline{)\,10011.01} \qquad \text{divisor} \overline{)\,\text{dividend}}^{\text{quotient}}$$

$$
\begin{array}{r}
101.1 \\
111 \overline{)\,100110.1} \\
\underline{111} \\
1010 \\
\underline{111} \\
111 \\
\underline{111} \\
0
\end{array}
$$

Check:

$$10011.01 = 2^4 + 2^1 + 2^0 + 2^{-2} = 16 + 2 + 1 + \tfrac{1}{4} = 19.25$$

$$11.1 = 2^1 + 2^0 + 2^{-1} = 2 + 1 + \tfrac{1}{2} = 3.5$$

$$3.5 \overline{)\,19.25}$$

$$
\begin{array}{r}
5.5 \\
35 \overline{)\,192.5} \\
\underline{175} \\
175 \\
\underline{175} \\
0
\end{array}
$$

2	5		
	2	1	
	1	0	(101)
	0	1	

$$.5 \times 2 = 1.0 \qquad (.1)$$

Hence, 101.1 is the binary equivalent of 5.5, which checks out OK.

Bits, Bytes and Words

A single binary digit is called a “bit”. All information in a digital system is represented by a bit.

4 bit sequence is a *nibble*

8 bit sequence is a *byte*

16 bit sequence is a *word*

The number of bits in the data sequence processed by a computer is an important characteristic. An 8 bit microprocessor can receive, store, and transmit data or instructions in the form of bytes. Eight bits can be arranged in $2^8 = 256$ different combinations, thus a byte can have 256 values.

Other Notations

The number of years in a century can be written as 100D or 100_{10} in the decimal system. In binary notation this would be written 01100100B or 01100100_2 ; the suffix B or subscript 2 is used wherever necessary to avoid confusion.

Octal Number System

The *octal* number system is a base 8 system and so has eight distinct digits {0, 1, 2, 3, 4, 5, 6, 7 }. It is expressed as a string of any combination of the eight digits. To convert from octal to decimal, we follow the same procedure for converting from binary to decimal; that is, express the octal number in its polynomial form and evaluate this polynomial by using decimal-system addition.

Example

Convert the number 367.240_8 to its decimal equivalent.

$$367.240_8 = 3\times8^2 + 6\times8^1 + 7\times8^0 + 2\times8^{-1} + 4\times8^{-2} + 0\times8^{-3}$$

$$= 192 + 48 + 7 + \frac{2}{8} + \frac{4}{64} + 0$$

$$= 247 + \frac{20}{64}$$

$$= 247.3125_{10}$$

To convert from decimal to octal we use the same procedure as converting from decimal to binary, but instead of diving by 2 for the integer part, divide by 8 to obtain the octal equivalent. Also, instead of multiplying by 2 for the fractional part, multiply by 8 to obtain the fractional octal equivalent of the decimal system. However, it is more common to convert from binary to octal and vice-versa.

The conversion from binary to octal is accomplished by grouping the binary numbers into groups of 3 bits each, starting from the binary point and proceeding to the right and to the left. Each group is then replaced by its octal equivalent.

Example

Convert 01100100_2 into its octal equivalent.

Grouping the bits into groups of 3 bits from the binary point we get:

001 100 100

Note that a leading zero was added to complete the first group. Each group is now replaced by its octal equivalent to get:

001 100 100

1 4 4

Thus,

$01100100_2 = 144\ O = 144_8$

The three-bit octal numbers are easier to work with than their 8-bit binary equivalents.

To convert from octal to binary replace each octal digital by its 3-bit binary equivalent.

Hexadecimal Numbering System

The hexadecimal numbering system is a base-16 system and has sixteen distinct digits {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} where A is the equivalent of decimal 10, B to 11, …, and F to 15. A hexadecimal number is expressed as a string of any combination of the 16 symbols. To convert from hexadecimal to decimal and vice versa we follow the same procedure for conversion between decimal and octal, except we now use 16 instead of 8.

Example

Convert the number $2AB.F8_{16}$ to its decimal equivalent.

$$2AB.F8 = 2\times16^{2} + A\times16^{1} + B\times16^{0} + F\times16^{-1} + 8\times16^{-2}$$

$$= 2\times16^{2} + 10\times16^{1} + 11\times16^{0} + 15\times16^{-1} + 8\times16^{-2}$$

$$= 512 + 160 + 11 + \frac{15}{16} + \frac{8}{256}$$

$$= 683 + \frac{248}{256}$$

$$= 683.96875_{10}$$

To convert from binary to hexadecimal group the binary numbers into 4 bits each; starting from the binary point and proceeding to the right and to the left and then replace each group by its hexadecimal equivalent.

Example

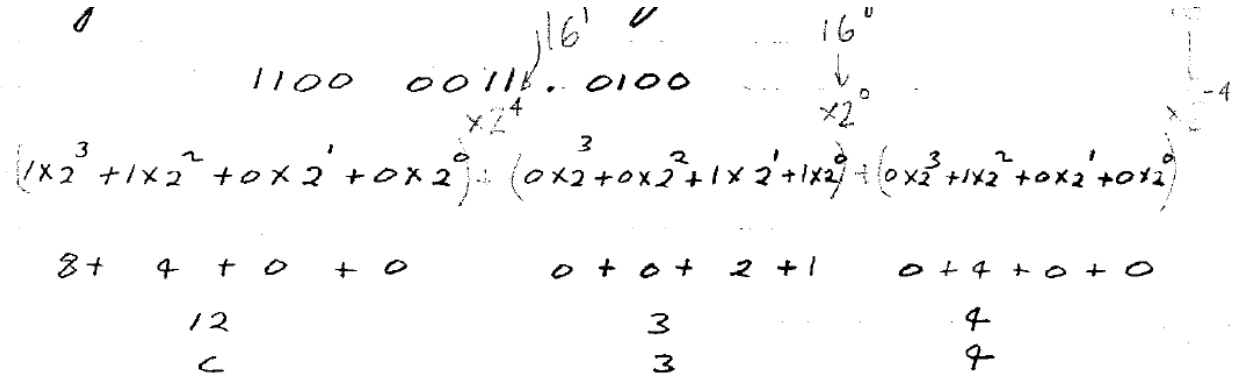
Convert the following into their hexadecimal equivalents.

A) 11000011.01_2

B) 01100100_2

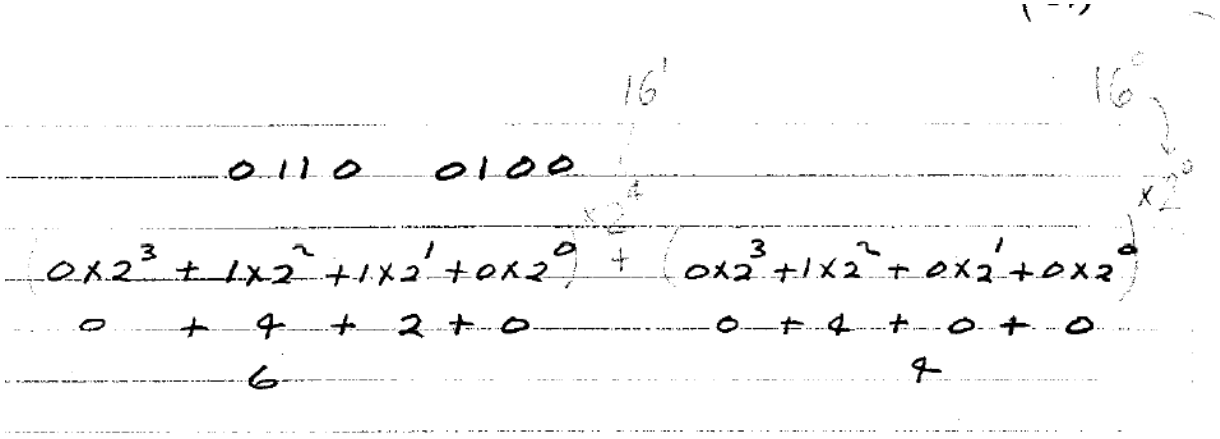
Results:

A) Group the bits into groups of 4 bits from the binary point and replace each group by its hexadecimal equivalent:



Thus $11000011.01_2 = C3.4_{16}$.

B)



Thus $01100100_2 = 64H = 64_{16}$.

To convert from hexadecimal to binary replace each hexadecimal digit by its 4-bit binary equivalent. A table of the four number systems is given below:

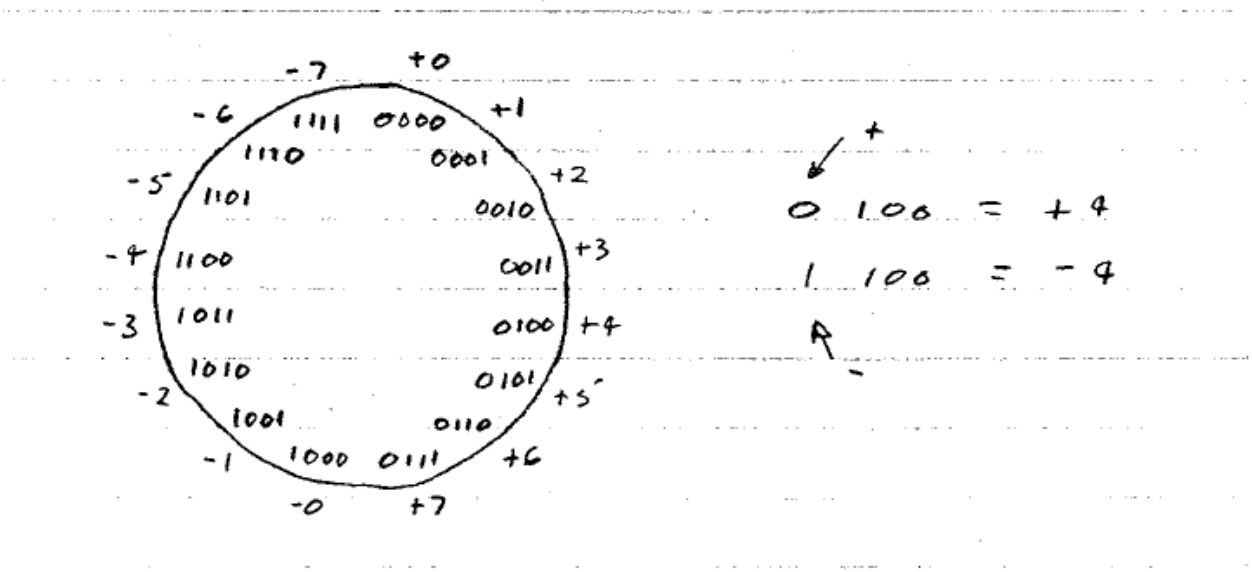
Decimal	Binary	Octal	Hexadecimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Signed Magnitudes

In binary notation, an n-bit data word can represent the first 2^n non-negative integers. To allow for both positive and negative numbers, the most significant bit (MSB) can be designated as the sign bit (0 for positive numbers, 1 for negative numbers). The lower order bits then represent the magnitude of the number in binary notation.

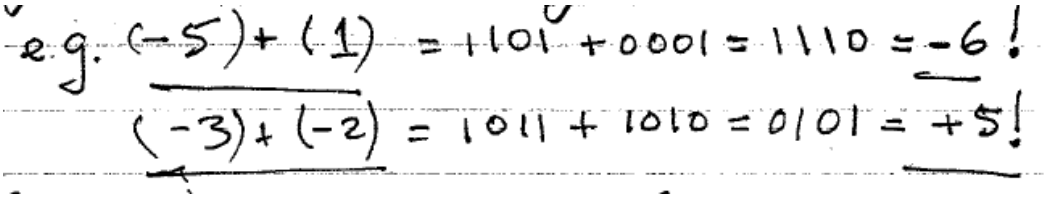
The figure below shoes a “number wheel” representation of a 4-bit number system. The figure shows the binary numbers and their decimal integer equivalents, assuming that the numbers are interpreted as sign and magnitude.

The largest positive number that can be represented in three data bits is +7 = 2^3 – 1. Similarly, the smallest negative number is -7.



This method has the following disadvantages:

- The number zero has two different representations
- Two different arithmetic circuits are required to process positive and negative numbers, see the following straight-binary example giving incorrect answers:



Complements

A better notation for computers is based on the fact that adding the complement of a number is equivalent to subtracting the number. Hence instead of performing A-B using a subtractor, we can perform A + (-B) to obtain the same result using an adder.

For each base r system, there are two types of complements, namely, the *radix* complement, also known as the r's complement, and the *diminished radix* complement, also known as the (r-1)'s complement.

Radix Complement

The radix complement, denoted by $[N]_r$, for a n-digital and r-base number $(N)_r$ is defined as follows:

$$[N]_r = \begin{cases} (r^n)_r - (N)_r & \text{for } N \neq 0 \\ 0 & \text{for } N = 0 \end{cases}$$

Example:

A) Obtain the 2-digit 10's complement of 15 and 24.

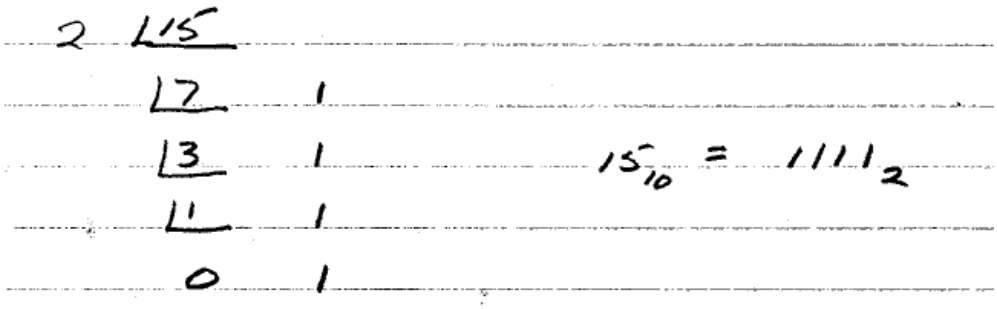
B) Represent -15 and -24 in 8-bit signed 2's complement notation.

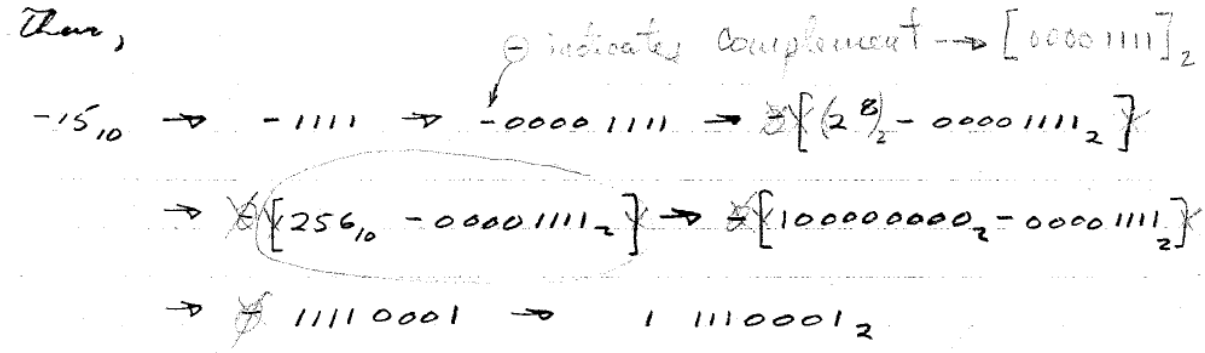
Answers:

A) $15_{10} = 10^2 - 15 = 100 - 15 = 85$

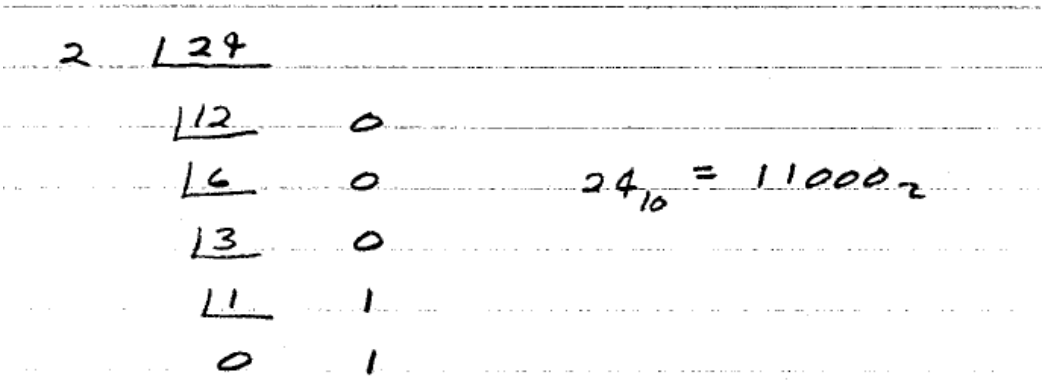
$24_{10} = 10^2 - 24 = 100 - 24 = 76$

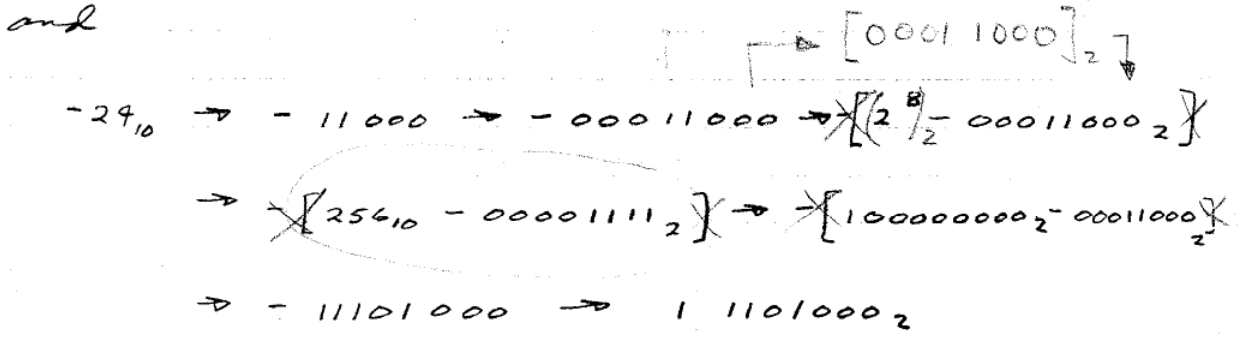
B) In binary:





and:



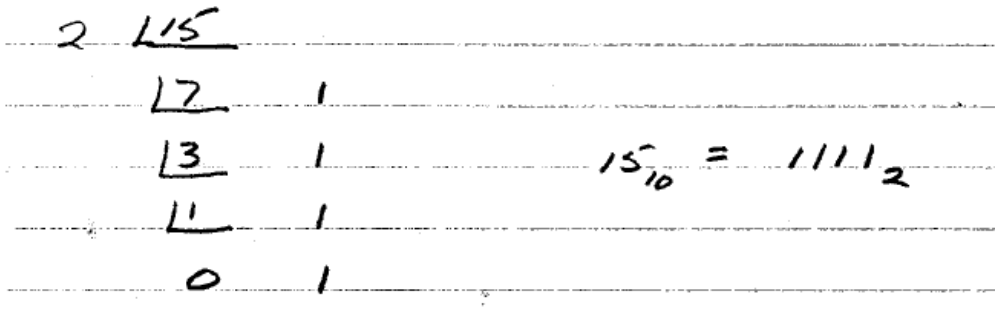


The 2's complement of a binary number can be obtained directly from the given number of copying each bit of the number, starting at the lest significant bit, and proceeding towards the most significant bit until the first 1 has been copied. After the first 1 has been complied, replaced each of the remaining 0's and 1's by 1's and 0's respectively.

Example

The 4-bit 2's complement number representation is shown below. Note there is only one representation for zero.

A) Represent -15 and -24 in 8-bit signed 2's complement notation.



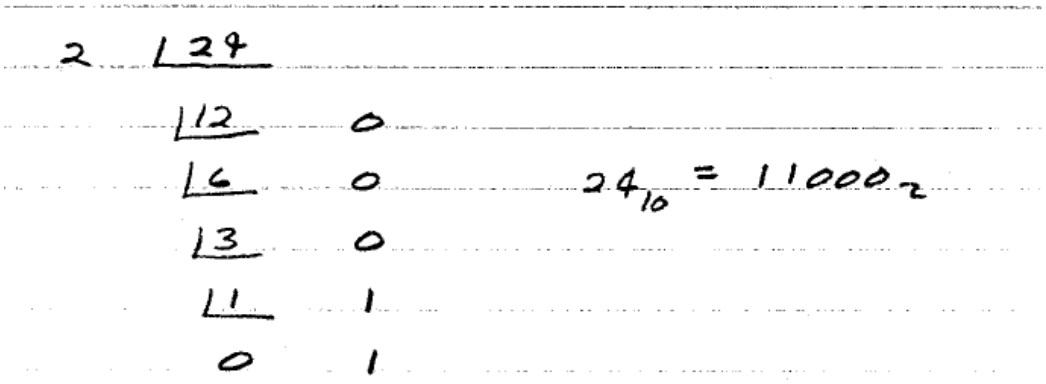
Convert 1111 to 8-bit number:

00001111

Starting from left-hand side, invert each bit until the last '1' is encountered:

11110001

Therefore -15 is 11110001 in signed 2's complement.



Again convert to 8-bit number:

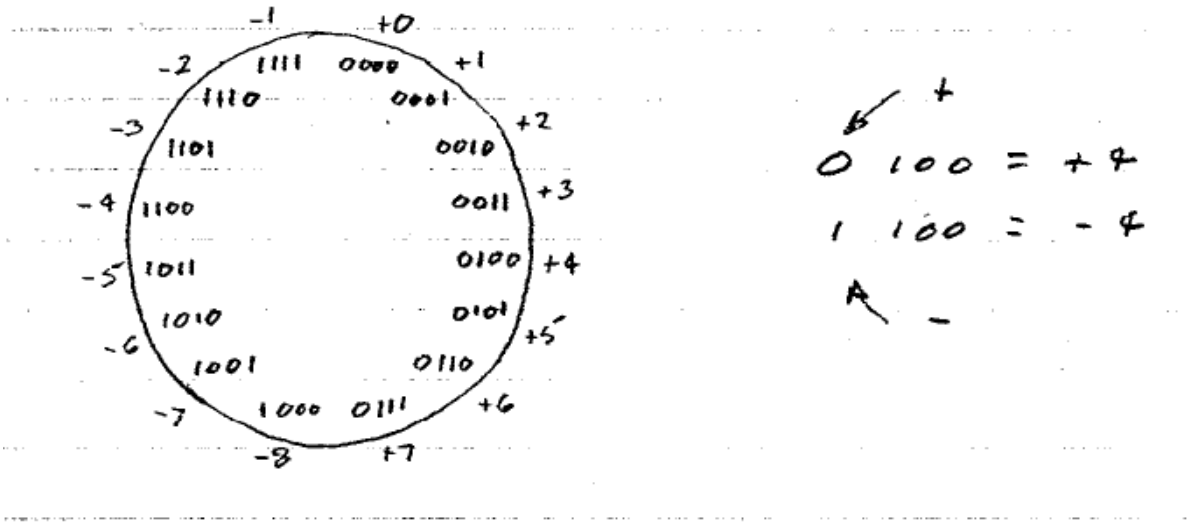
00011000

Starting from left-hand side, invert each bit until the last '1' is encountered:

11101000

Therefore -24 is 11101000 in signed 2's complement.

The 4-bit 2's complement number representation is shown below. Note there is only one representation for zero.



Two's Complement Arithmetic

Addition

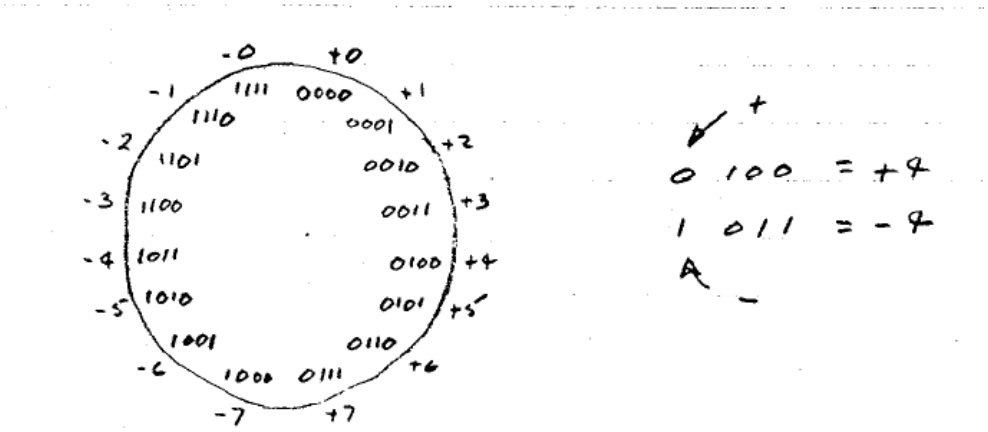
Two n-bit signed binary numbers in 2's complement format are added by performing a binary addition of the two numbers, including the sign bits. If a carryover bit results from the leftmost bit, it is discarded. The leftmost bit of the result will give the sign of the sum.

If the sign bit is a 1 we must take the 2's complement of the result to get the real magnitude of the final answer.

Subtraction

In 2's complement format subtraction of two signed numbers is performed by adding the 2's complement of the subtractand to the numerand. If a carryover results from the leftmost bit, it is discarded. Also the leftmost bit gives the sign of the difference.

Note that the 10's complement can be obtained by forming the 9's complement and adding 1. The 2's complement can be obtained by forming the 1's complement ad adding 1. The 1's complement is formed by changing 1's to 0's and 0's to 1's. The 1's complements representation is shown below. Note the two representations of zero:

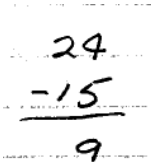


Example

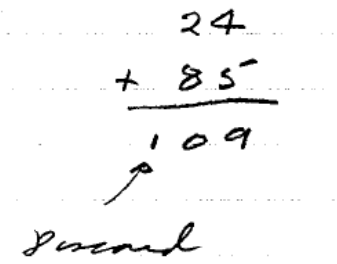
Perform (A) 24-15 and (B) 15-24 directly and by complement notation.

Answers:

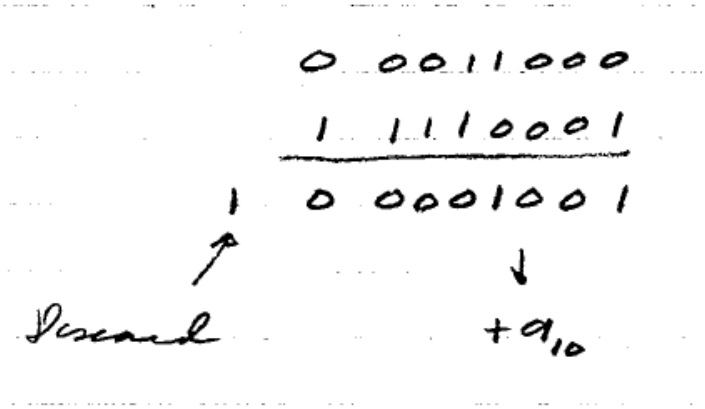
A) Direct



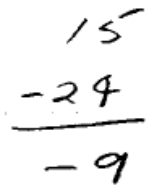
10's Complement



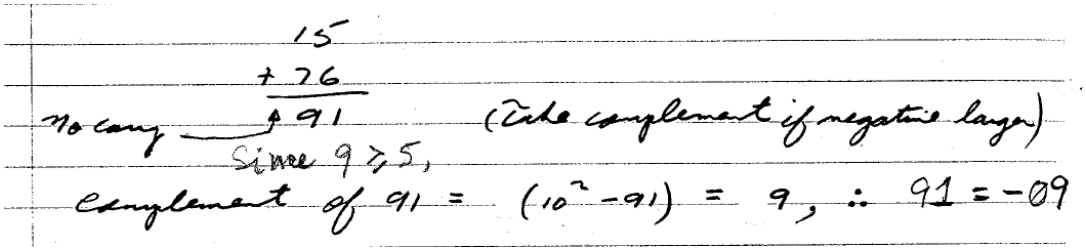
2's Complement



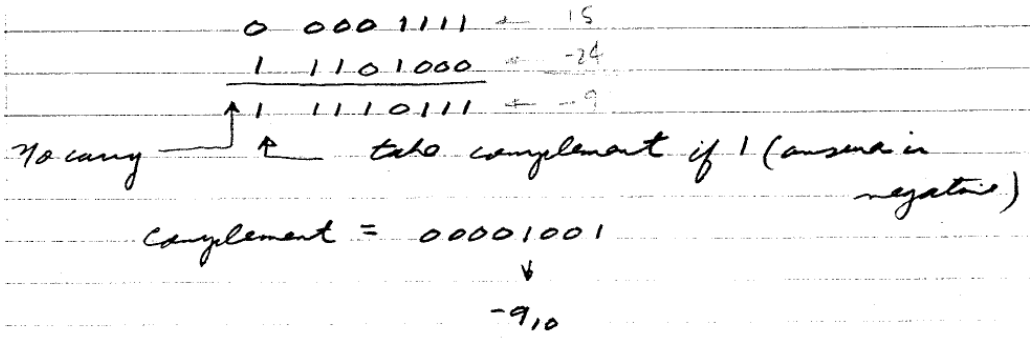
B) Direct



10's Complement



2's complement



Note that for the 10's complement the carryover is discarded and if the result is negative the complement must be taken to get the final result.

Binary Coded Decimal

For convenience, computer input/output devices may accept/provide decimals on the human side and binaries on the computer side. In a binary-coded decimal number each of the decimal digital is coded in binary, using 4 bits. For example in the 8421 code $6_{10} = 0110_2$, $3_{10} = 011_2$, and 363 = 0011 0110 0011 $_{BCD}$.

When a computer is to handle letters as well as numbers, the *alphanumeric* code is used. In the American Standard Code for Information Interchange (ASCII) seven bits are used to represent all the characters and punctuation marks on a teletypewriter keyboard plus some control signals. Note that $2^7 = 128$ combinations of 7 bits. An eighth bit, the MSB, is a *parity bit* used in error correction. In the *even parity* connection, the MSB is set so that the number of 1's in each ASCII character is even, the present of an odd number of 1's indicates an error.

Boolean Algebra

Boolean algebra is useful in manipulating binary variables (0,1) in OR, AND, or NOT relations and in the analysis and design of all types of digital systems.

Boolean Theorems

The basic postulates are given in the tables below. In general, the inputs and outputs are variables (either 1 or 0).

Boolean Postulates in 0 and 1

OR	AND	NOT
$0+0=0$	$0 \cdot 0 = 0$	$\bar{0} = 1$
$0+1=1$	$0 \cdot 1 = 0$	$\bar{1} = 0$
$1+0=1$	$1 \cdot 0 = 0$	
$1+1=1$	$1 \cdot 1 = 1$	

Basic Boolean Identities

No.	Identity	Comments
1	$A+0=A$	Operations with 0 and 1
2	$A+1=1$	Operations with 0 and 1
3	$A+A=A$	Idempotent
4	$A+\overline{A}=1$	Complements
5	$A \bullet 0=0$	Operations with 0 and 1
6	$A \bullet 1=A$	Operations with 0 and 1
7	$A \bullet A=A$	Idempotent
8	$A \cdot \bar{A} = 0$	Complements
9	$\overline{\overline{A}}=A$	Involution
10	$A+B=B+A$	Commutative
11	$A \bullet B=B \bullet A$	Commutative
12	$A+(B+C)=(A+B)+C=A+B+C$	Associative
13	$A \bullet (B \bullet C)=(A \bullet B) \bullet C=A \bullet B \bullet C$	Associative
14	$A \bullet (B+C)=(A \bullet B)+(A \bullet C)$	Distributive
15	$A+(B \bullet C)=(A+B) \bullet (A+C)$	Distributive
16	$A+(A \bullet B)=A$	Absorption
17	$A \bullet (A+B)=A$	Absorption
18	$(A \bullet B)+(\overline{A} \bullet C)+(B \bullet C)=(A \bullet B)+(\overline{A} \bullet C)$	Consensus
19	$\overline{A+B+C+\ldots}=\overline{A} \bullet \overline{B} \bullet \overline{C} \ldots$	De Morgan
20	$\overline{A \bullet B \bullet C \bullet \ldots}=\overline{A}+\overline{B}+\overline{C} \ldots$	De Morgan
21	$(A+\overline{B}) \bullet B=A \bullet B$	Simplification
22	$(A \bullet \overline{B})+B=A+B$	Simplification

The validity of the 22 rules can be verified by substituting all possible values for the Boolean variables and evaluating the left and right-hand sides of each identity. This is known as a proof by *perfect induction*.

Example:

Use proof by induction to verify the consensus identity:

$$(A \cdot B) + (\bar{A} \cdot C) + (B \cdot C) = (A \cdot B) + (\bar{A} \cdot C)$$

A	B	C	$\bar{A}$	$A \cdot B$	$\bar{A} \cdot C$	$B \cdot C$	LHS $(A \cdot B)+(\bar{A} \cdot C)+(B \cdot C)$	RHS $(A \cdot B)+(\bar{A} \cdot C)$
0	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	1	1
0	1	0	1	0	0	0	0	0
0	1	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
1	1	0	0	1	0	0	1	1
1	1	1	0	1	0	1	1	1

Note that when B●C=1, this means B=C=1. One of the remaining terms will always be 1 in that case, which is why B●C is redundant.

The first nine identities are the fundamental relations of Boolean algebra. Identities 10-14 are similar to the laws of ordinary algebra. Identities 10 and 11 are the commutative rules, 12 and 13 are the associative rules, and 14 and 16 are the distributive rules. Identities 16-18 do not apply to ordinary algebra but are very useful in Boolean Algebra. Identities 16 and 17 are the absorption identities; identity 18 is the consensus identity; identity 19 and 20 are De Morgan's rules. Formally identities 21 and 22 are simplification rules.

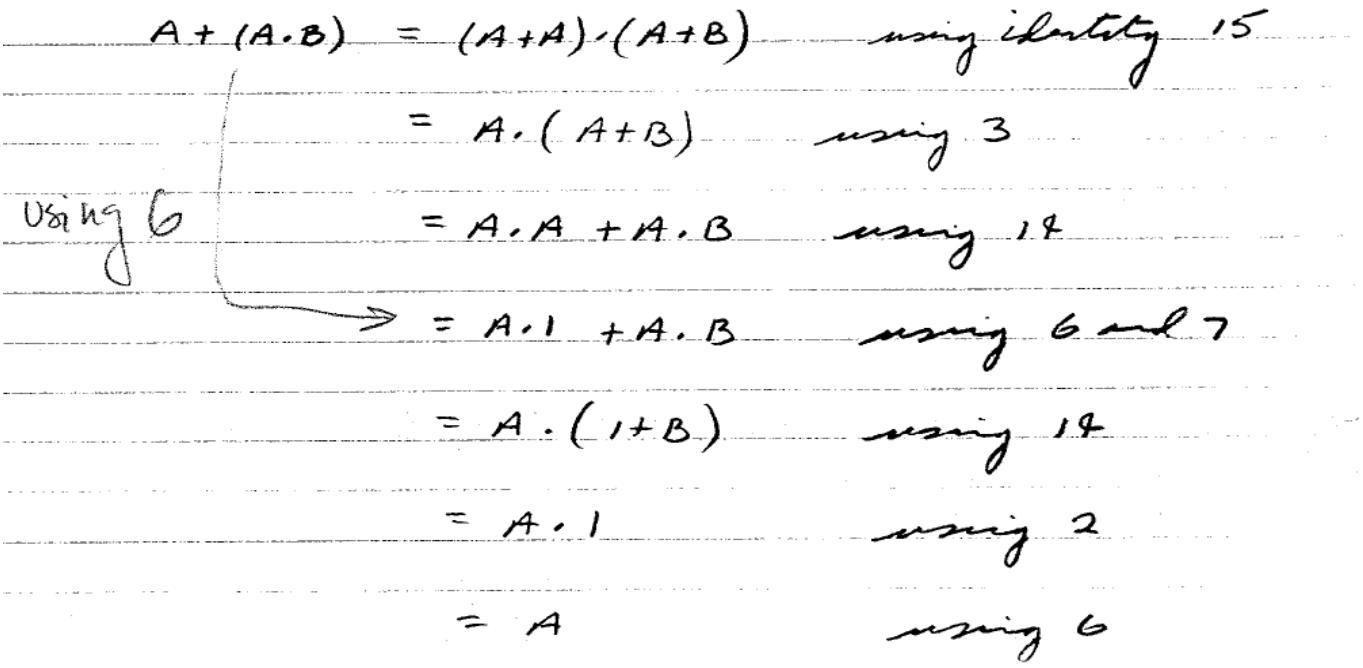
The basic identities can be used to simplify Boolean functions.

Example

Derive the absorption rule:

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

Using other basic theorems.



De Morgan's Theorems

De Morgan's theorems are easily interpreted in terms of logic circuits. The first says that a NOR gate is equivalent to an AND gate with NOT circuits in the inputs. The second says that a NAND gate is equivalent to an OR gate with NOT circuits in the inputs. As started by Shannon, De Morgan's theorem says:

> To obtain the inverse of any Boolean function, invert all variables and replace all OR's by AND's and all AND's by OR's.

Example

Use De Morgan's theorems to design a combination of NAND gates equivalent to a two-input OR gate.

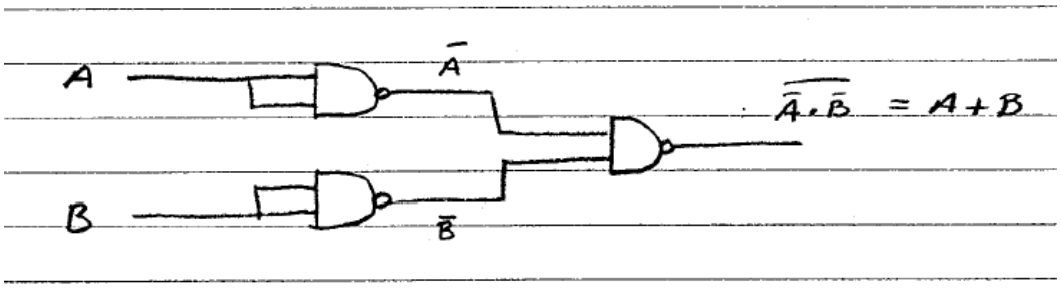
The desired function is:

$$f = A + B$$

Using De Morgan's theorem (Identity 19) we get:

$$f = A + B = \overline{\bar{A} \cdot \bar{B}}$$

Suggesting a NAND gate with NOT inputs because $\overline{A \cdot A} = \bar{A}$ by theorem 7, a NAND gate with the inputs tied together performs the NOT operation. The logic circuit is shown below:



Logic Circuit Analysis

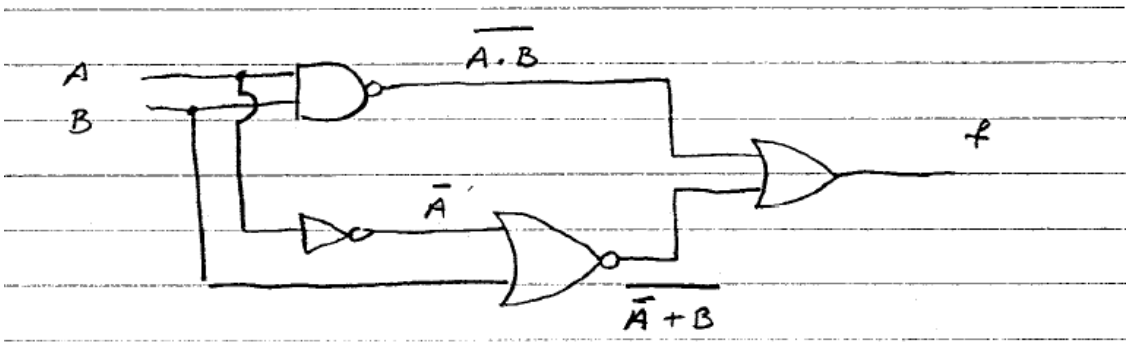
The Boolean identities permit us to manipulate logic statements or functions directly, without setting up truth tables. Also, the use of Boolean algebra can lead to simpler logic statements that are easier to implement. De Morgans theorems are useful in finding NAND operations that are equivalent to other operations.

The analysis of a logic circuit consists in writing a logic statement expression the overall operation of the circuit. This can be done by starting at the input and tracking through the circuit noting the function realized at each output. The resulting expressions can be simplified or put into an alternate form by using Boolean Algebra. A truth table can be constructed.

Note the symbol A●B can be simplified to AB or A(B).

Example

Analyze the given logic circuit:



Construct the truth table to demonstrate that this circuit could be replaced by a single NAND gate.

The suboutputs are as noted on the diagram. The overall function can be simplified as follows:

$$f = \overline{A \cdot B} + \overline{\bar{A} + B}$$

$$= (\bar{A} + \bar{B}) + A \cdot \bar{B} \quad \text{using 19 and 20}$$

$$= \bar{A} + (1 + A) \cdot \bar{B} \quad \text{using 14}$$

$$= \bar{A} + 1 \cdot \bar{B} \quad \text{using 2}$$

$$= \bar{A} + \bar{B} \quad \text{using 6}$$

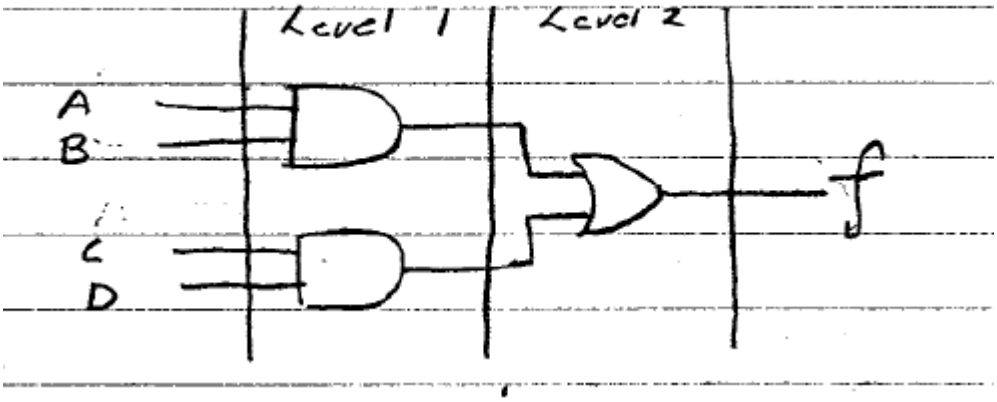
$$= \overline{A \cdot B} \quad \text{using 20}$$

The truth table is given below:

A	B	$A \cdot B$	$\overline{A \cdot B}$	$\bar{A} + B$	$\overline{\bar{A} + B}$	f
0	0	0	1	1	0	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	1	0	0

Two-Level Combinational Logic

A two-level implementation means that there are only two gates between input and output. A two-level implementation of $f = A \bullet B + C \bullet D$ is shown below:



Each appearance of a variable or its complement is an expression is called a *literal*. *Combinational* networks are those where the outputs depend only on the current input. They are circuits without a memory.

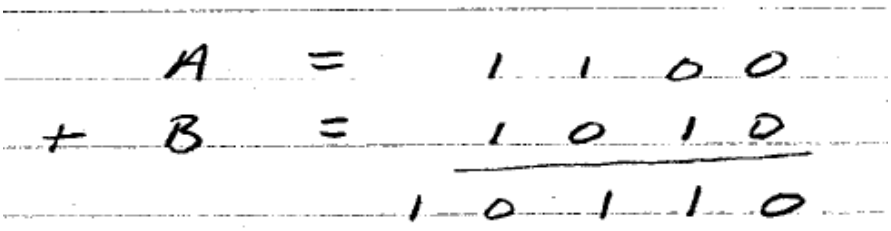
Logic Circuit Synthesis

The logic designer starts with a logic statement or truth table, converts the logic function into a convenient form and then realizes the desired functions by means of a standard or special logic Elements.

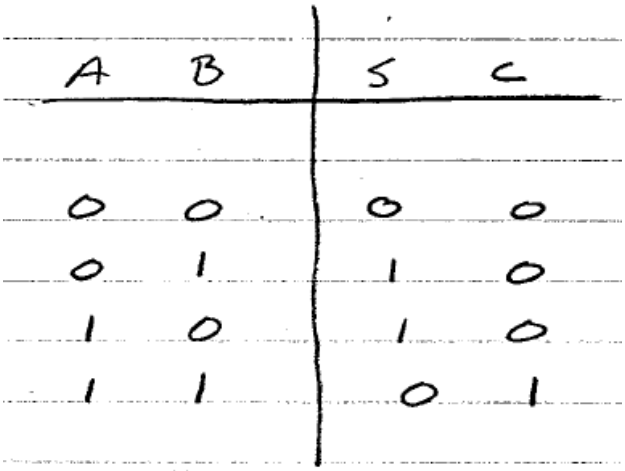
Adding

The Half Adder

Consider the process of addition. In adding two binary digits, the possible sums are shown below. Note that when A=1 and B=1, the sum in the first column is 0 and there is a carry of 1 to the next higher column.



As indicated in the truth table, the half-adder must perform as follows: “s is 1 if A is 0 AND B is 1, OR if A is 1 AND B is 0; c is 1 if A AND B are 1”. In logic nomenclature, this becomes:

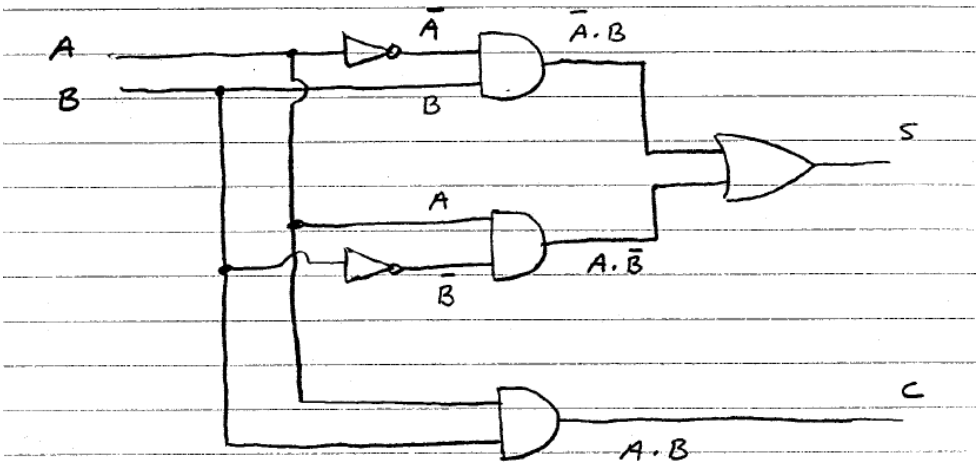


Which can be written as:

$$S = \bar{A} \cdot B + A \cdot \bar{B} \quad \text{and} \quad C = A \cdot B$$

Note that a full-adder can accept the carry from the adjacent column.

To synthesize a half-adder circuit, start with the output and work backwards. The above equation indicates that the sum s is the output of an OR gate; the inputs are obtained from AND gates; inversion of A and B is necessary. The above expression also indicates that the carry c is the output of an AND gate. The logic circuit is shown below.



Different Boolean expressional are possible for a given logic statement and some will lead to better circuit realizations than others. Consider the last expression:

$$\bar{A}\cdot B + A\cdot\bar{B} = \overline{\bar{A}+\bar{B}} + \overline{\bar{A}+B} \quad \text{using 19}$$

$$= \overline{(A+\bar{B})\cdot(\bar{A}+B)} \quad \text{using 20}$$

$$= \overline{(A\cdot\bar{A})+(A\cdot B)+(\bar{A}\cdot\bar{B})+(B\cdot\bar{B})} \quad \text{using 14}$$

$$= \overline{(\bar{A}\cdot\bar{B})+(A\cdot B)} \quad \text{using 8}$$

$$= \overline{(\overline{A+B})+(\overline{\bar{A}+\bar{B}})} \quad \text{using 19}$$

$$= (A+B)\cdot(\bar{A}+\bar{B}) \quad \text{using 20}$$

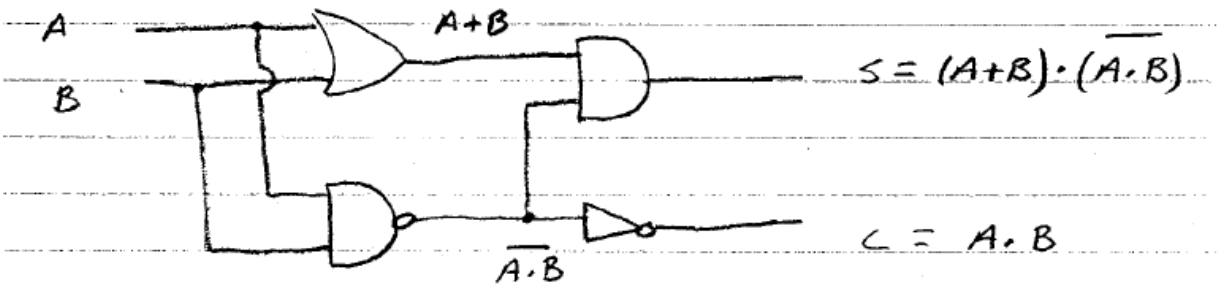
$$= (A+B)\cdot\overline{A\cdot B} \quad \text{using 20}$$

And referring to the truth table we see that another interpretation is:

"S is 1 if (A OR B) is 1 AND (A AND B) is NOT 1". The binary addition is:

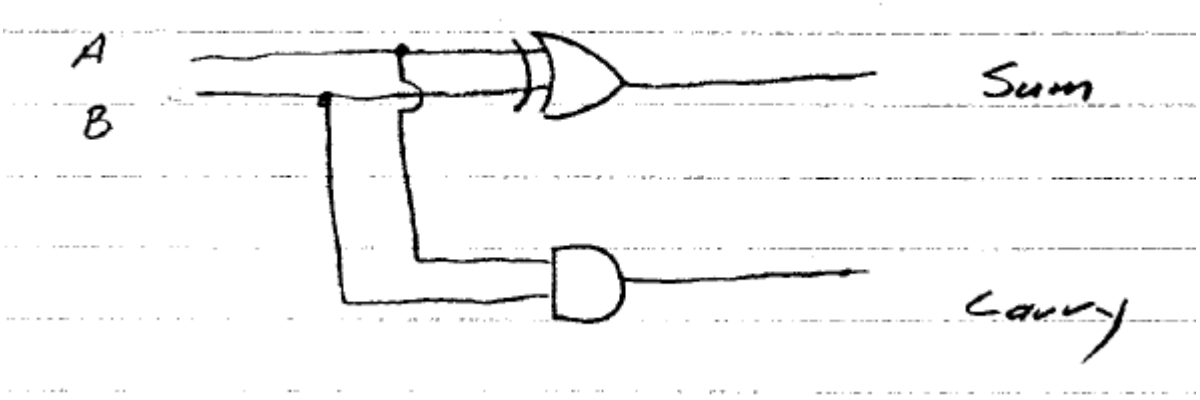
$$S = (A+B)\cdot(\overline{A\cdot B}) \quad \text{and} \quad C = A\cdot B$$

The synthesis of the circuit, working backwards from the output, is shown below:

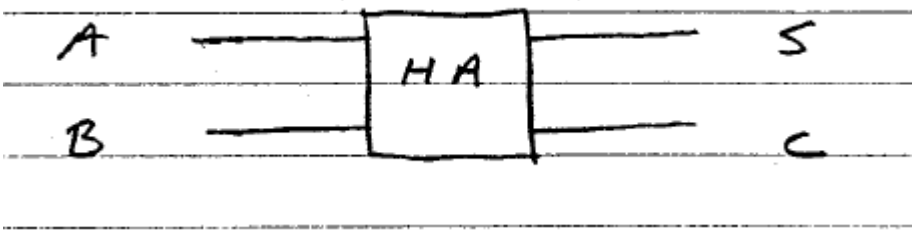


The circuit is better than the previous one in that fewer logic elements are used and the longest path from input to output passes through fewer levels.

In terms of the Exclusive-OR gate, the half-adder takes the simple form shown below:



The half adder can be treated as a discreet logic element and represented as shown below:



The Full Adder

To add two binary digits (bits) the half-adder performs the most elementary part. For a complete addition we need a fill-adder capable of handling the carry input as well. The addition process is illustrated below where ci is the carry from the proceeding column:

Inputs	C_i	1 1 1 0	(carry in)
	A	0 1 0 1 1	
	B	0 1 1 1 0	
Outputs	S	1 1 0 0 1	(Sum)
	C_o	0 1 1 1 0	(carry out)

Each carry of 1 must be added to the two digits in the next column, so the logic circuit must be able to combine three inputs. The truth table for the full-adder is shown below.

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

Note that both S and Co have four cases with 1's in the output columns. In logic notation we have:

"S is ONE when..." $S = A \cdot \bar{B} \cdot \bar{C}_i + \bar{A} \cdot B \cdot \bar{C}_i + \bar{A} \cdot \bar{B} \cdot C_i + A \cdot B \cdot C_i$

...OR...OR...OR...

"C_o is ONE when..." $C_o = \bar{A} \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$

~~OR...OR...OR...~~

The expression for Co can be simplified as follows:

$C_o = \bar{A} \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$

$= \bar{A} \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i + A \cdot B \cdot C_i$ using 3

$= \bar{A} \cdot B \cdot C_i + A \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$ (Comm). using 10

$= (\bar{A} + A) \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$ (Dist). using 14

$= 1 \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$ using 4

$= B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$ using 6

$= B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i + A \cdot B \cdot C_i$ using 3

$= B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$ using 10

$= B \cdot C_i + A \cdot (\bar{B} + B) \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$ using 4

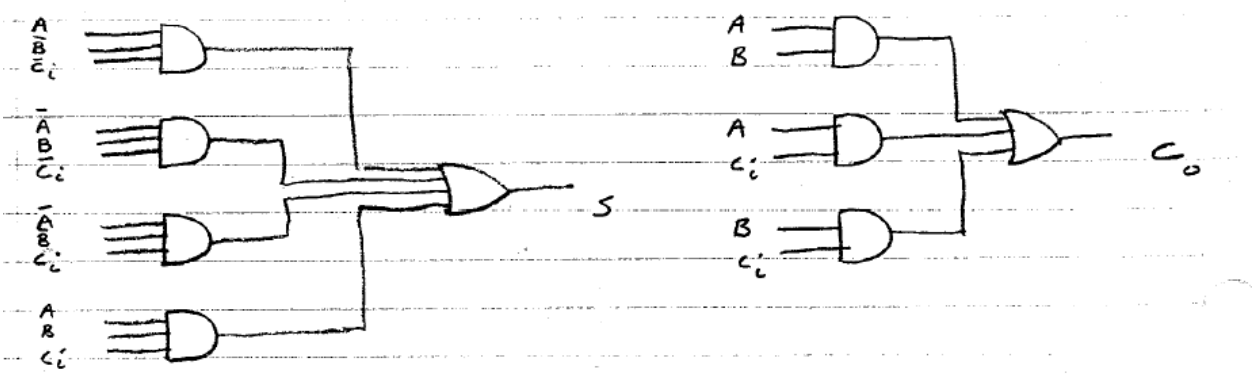
$= B \cdot C_i + A \cdot 1 \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$ using 9

$= B \cdot C_i + A \cdot C_i + A \cdot B \cdot (\bar{C}_i + C_i)$ using 6 and 14

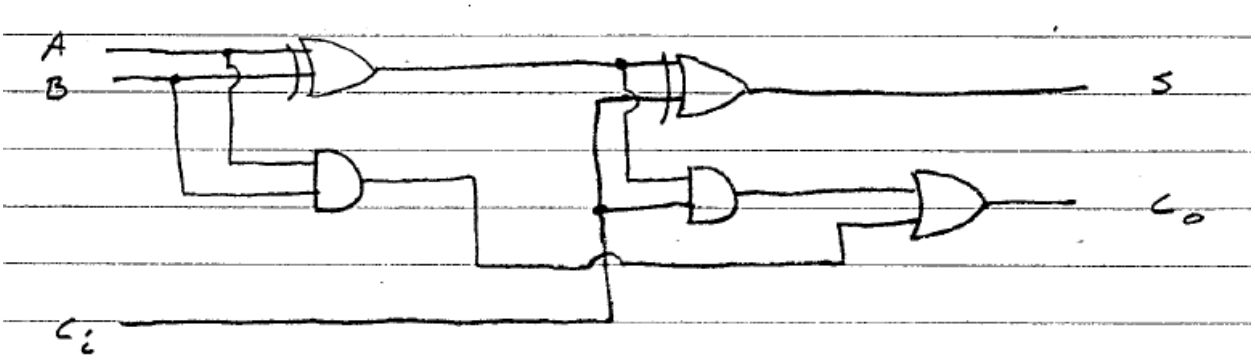
$C_o = B \cdot C_i + A \cdot C_i + A \cdot B$ using 4

Although this leads to a simpler expression, applying the rules of Boolean algebra in this situation does not guarantee the simplest expression. A more systematic approach will be discussed later.

Using the expression for S and Co the full adder can be implemented as shown below:



The full adder can also be implanted with two half-adders and one OR gate, as shown below:



For this case the S output from the second half-adder is the Exclusive-OR of Ci and the output of the first half-adder, giving:

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

$$= \bar{C}_i \cdot (\bar{A} \cdot B + A \cdot \bar{B}) + C_i \cdot \overline{(\bar{A} \cdot B + A \cdot \bar{B})}$$

$$= \bar{C}_i \cdot (\bar{A} \cdot B + A \cdot \bar{B}) + C_i \cdot (\overline{(\bar{A} \cdot B)} \cdot \overline{(A \cdot \bar{B})}) \quad \text{using 19}$$

$$= \bar{C}_i \cdot (\bar{A} \cdot B + A \cdot \bar{B}) + C_i \cdot ((A + \bar{B}) \cdot (\bar{A} + B)) \quad \text{using 20}$$

$$= \bar{C}_i \cdot (\bar{A} \cdot B + A \cdot \bar{B}) + C_i \cdot (\underbrace{A \cdot \bar{A}}_{=0 \text{ by } 5'} + A \cdot B + \bar{B} \cdot \bar{A} + \underbrace{\bar{B} \cdot B}_{=0 \text{ by } 5'})$$

$$= \bar{C}_i \cdot (\bar{A} \cdot B + A \cdot \bar{B}) + C_i \cdot (A \cdot B + \bar{A} \cdot \bar{B})$$

$$S = A \cdot \bar{B} \cdot \bar{C}_i + \bar{A} \cdot B \cdot \bar{C}_i + \bar{A} \cdot \bar{B} \cdot C_i + A \cdot B \cdot C_i$$

as before. The carry out is the (Exclusive-OR of A and B AND Ci) OR'ed with A AND B, or:

$$C_o = C_i \cdot (\bar{A} \cdot B + A \cdot \bar{B}) + A \cdot B$$

$$= \bar{A} \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B$$

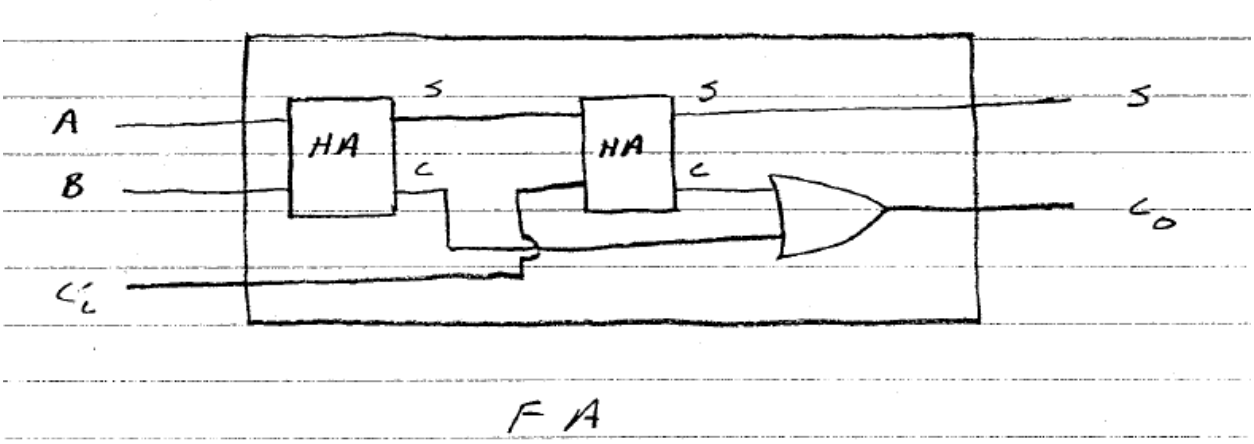
$$= \bar{A} \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot (\bar{C}_i + C_i) \quad \text{using 4}$$

or

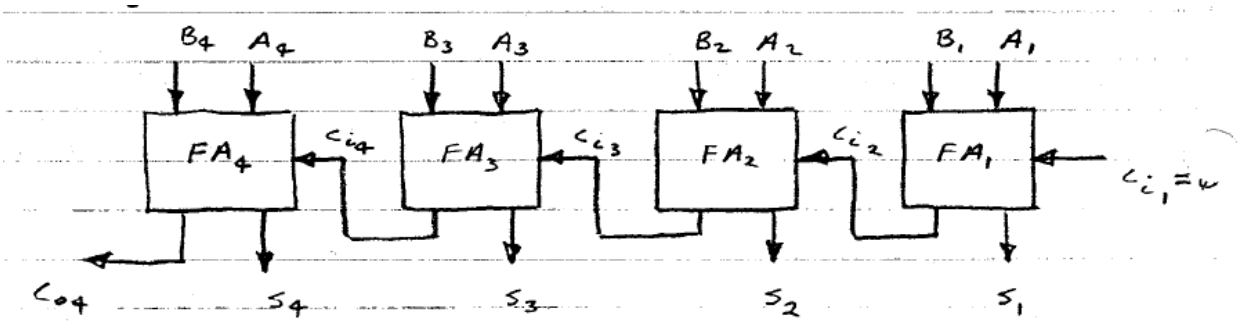
$$C_o = \bar{A} \cdot B \cdot C_i + A \cdot \bar{B} \cdot C_i + A \cdot B \cdot \bar{C}_i + A \cdot B \cdot C_i$$

as before.

The full adder can be treated as a discreet logic element and represent as shown below:



To obtain the binary addition of two n-bit binary numbers, we cascade n full-adder circuits together, with the carry in of a full-adder being connected to the carry out of the previous full adder. The interconnection of four full-adders to provide the addition of two 4-bit binary numbers is shown below:



Note that the initial adder need only be a half-adder since the initial Ci is 0.

MSI (Medium Scale Integration) packages are available that contain 4 and 8-bit binary adders.

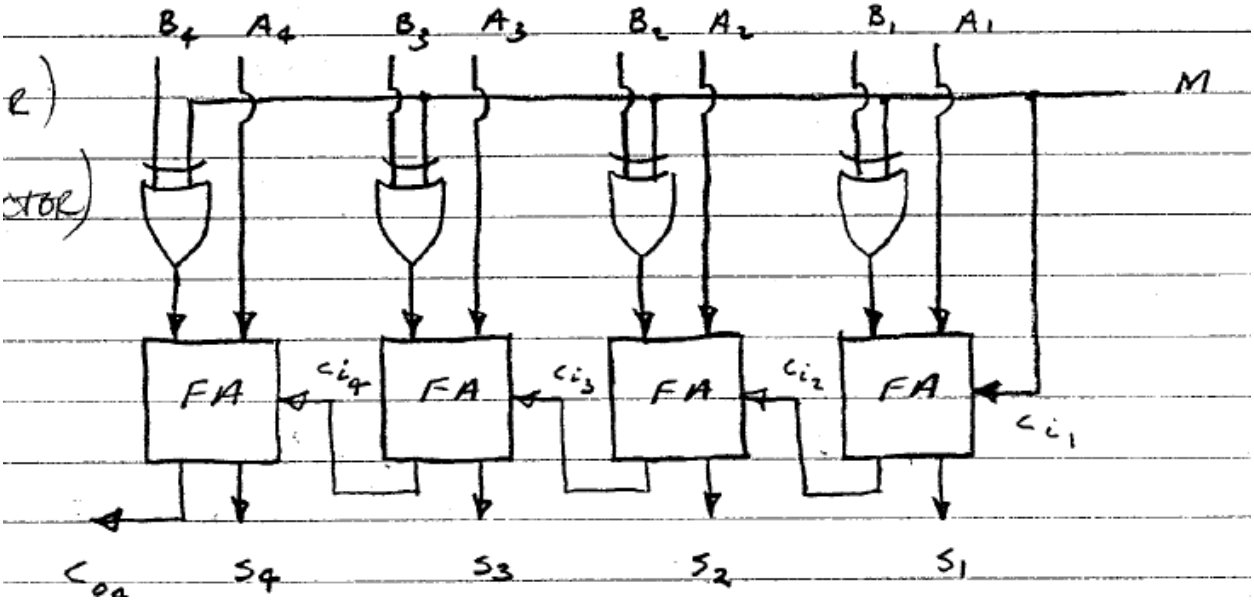
Subtraction

Direct Approach

Subtraction can be implemented with logic circuits in a direct manner as was done for adders. In this method the subtractend is subtracted from the numerend to form the difference. If the numerend is smaller than the subtractend, a 1 is borrowed from the next significant position. This borrow must be conveyed to the next stage. As in the case of adders, there are half- and full-subtractors.

Indirect Approach (Using Adders)

As discussed earlier, subtraction may be accomplished by taking the complement of the subtractend and adding it to the numerend. Subtraction then becomes addition requiring full-adders for machine implementation. The addition and subtraction operation can be combined into one circuit with the common binary adder. This is done by including an Exclusive-OR gate with each full adder as shown below. The mode input (M) controls the operation. When M=0, the circuits is an adder, and when M=1, the circuit becomes a subtractor. Each Exclusive-OR gate has input M and one of the inputs of B (Bi).

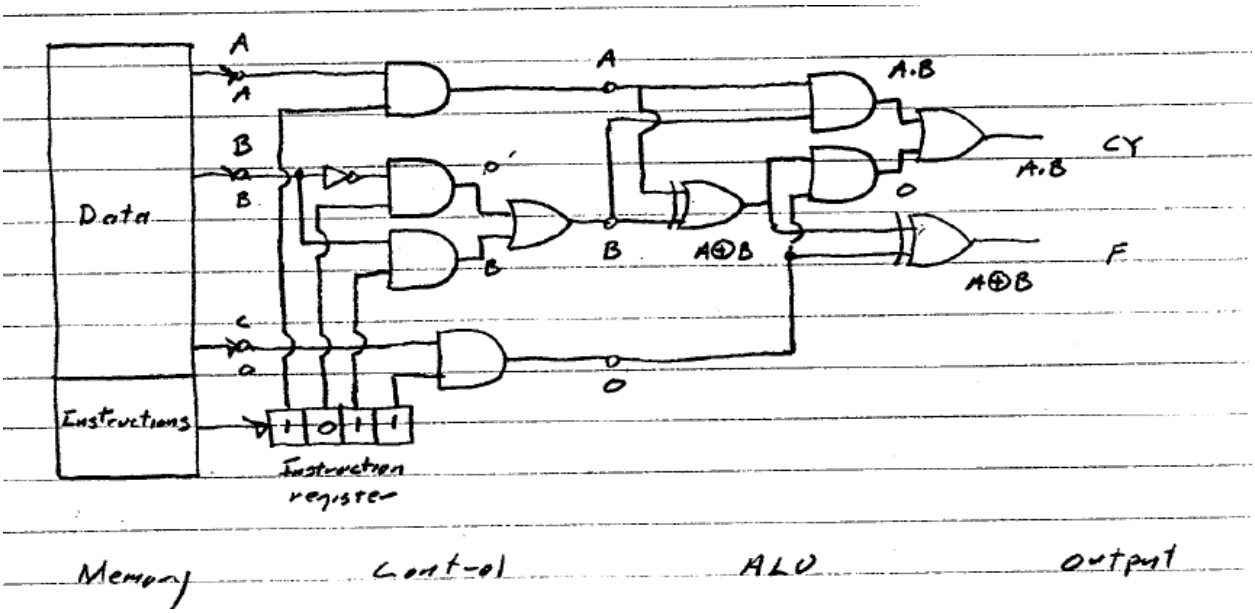


When M=0, we have Bi XOR 0 = Bi. The full-adders receive the value Bi, the input carry is 0, and the circuit performs A+B. When M-1, we have Bi XOR 1 = NOT Bi, and the input carry is 1. The Bi inputs are all complemented and a 1 is added through the input carry. The circuit performs (A + NOT(B) + 1) which is A plus the 2's complements of B. Note that NOT(B) is actually the 1's complement, but also called the "diminished 2's complement".

Arithmetic Logic Unit (ALU)

A arithmetic logic unit (ALU) is a combinational network of logic gates arranged to perform addition, complementing, incrementing, and the associated register for temporary storage of data or results. The ALU is governed by a control unit, which sets the various logic gates, feeds the numeric data, and provides the clock pulse

that regulates the speed of operation. An ALU and its control unit for an elementary example are shown below.



In this case the stored number A and B are operated according to the instruction in the form of a 4-bit word. The instruction is taken from memory and placed in a register. The instruction 1011 shown sets the logic gates so that A, B, and 0 are available for processing. Other instructions and the outputs are shown in the table below. There are 2 x 2^4 = 32 possibilities.

C	Instruction	Input to ALU	F output of ALU	C_Y output of ALU
0	1011	A, B, 0	$A \oplus B$	$A \cdot B$
0	1110	A, 1, 0	$\bar{A}$ (comp. of A)	A
0	1010	A, B, 0	$A \oplus B$	$A \cdot B$
1	1101	A, $\bar{B}$, 1	$\overline{A \oplus B}$	$(A \oplus B) + A \cdot \bar{B} = A \oplus B$
1	0101	0, $\bar{B}$, 1	$\bar{B}+1$ = 2's comp of B	$\bar{B}$
1	1011	A, B, 1	$\overline{A \oplus B}$	$A \cdot B + A \oplus B = A + B$

A Design Procedure

In logic design, gates must be combined to realize the desired function. The design proceeds according to the following steps.

1. Statement of function
2. Form a truth table
3. Obtain the Boolean expression of the function
4. Manipulate the Boolean expression to the simplest form
5. Realize in terms of AND, OR and NOT gates

Example

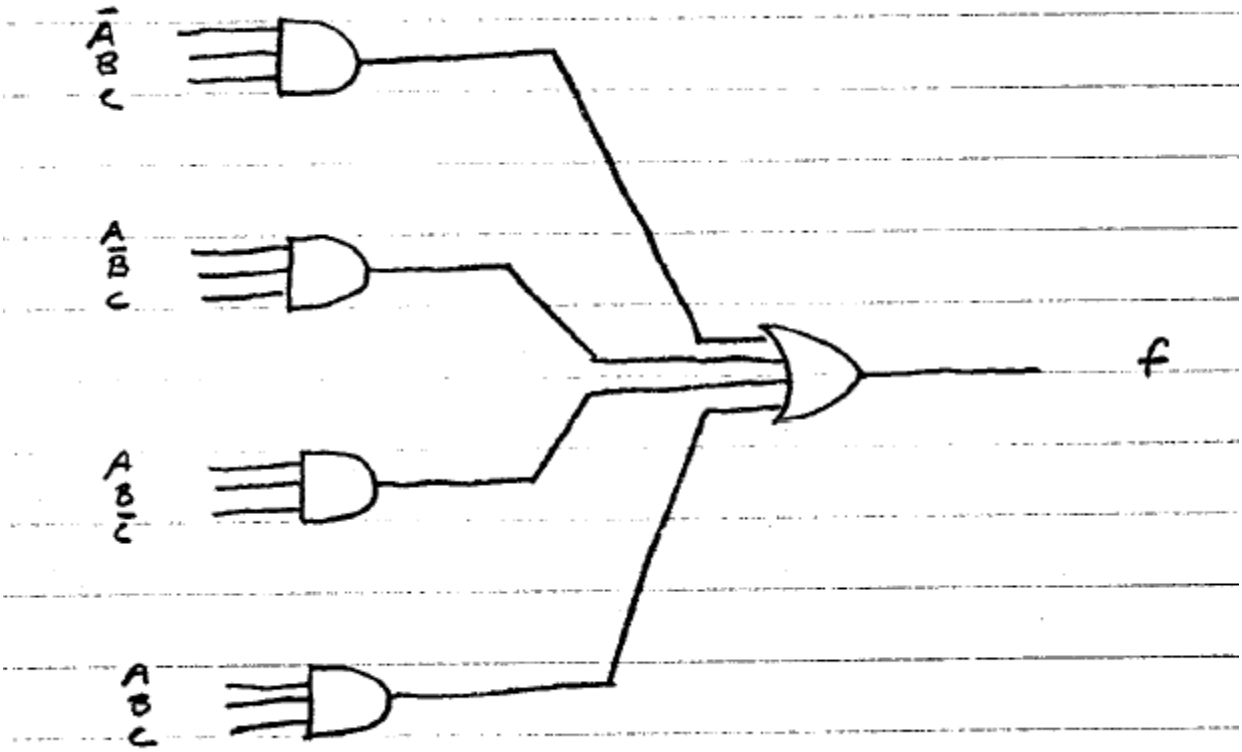
For increased reliability on a spacecraft triple sensing systems are used; no action is taken unless at least two of those systems call for action. The required system is known as a vote taker whose truth table is shown below:

A	B	C	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Because the function is YES(1) only when a majority of inputs are YES, the Boolean expression is:

$$f = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$$

If the complement of each variable is available (true in most computers), the realization is a combination of four AND gates feeding an OR gate:



If the complements are not available eight logic elements (three NOT elements) would be required, and simplification of the circuit is desirable. We proceed as follows:

$$f = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C + A \cdot B \cdot C$$

$$= A \cdot B \cdot (\bar{C} + C) + C \cdot (A \cdot B + A \cdot \bar{B} + \bar{A} \cdot B)$$

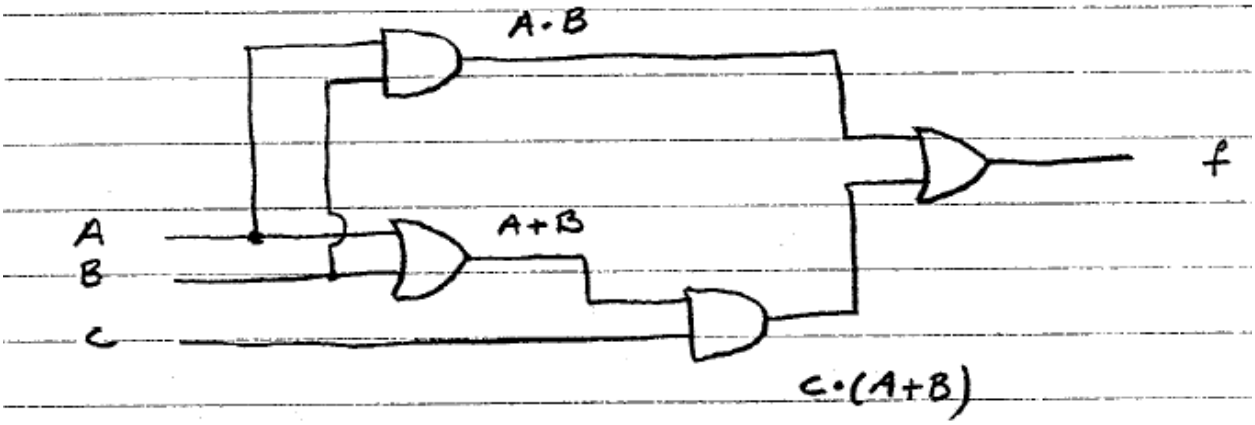
$$= A \cdot B + C \cdot [A \cdot \underbrace{(B + \bar{B})}_{=1} + \bar{A} \cdot B]$$

$$= A \cdot B + C \cdot (A + \bar{A} \cdot B)$$

$$= A \cdot B + C \cdot [\underbrace{(A + \bar{A})}_{=1} \cdot (A + B)] \quad \text{using 15}$$

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

This function requires only four logic elements as shown below:



Two-Level Canonical Forms

A Boolean function can be written in different forms. Certain forms, however, lead to more desirable combinational networks. These forms which are *canonical* forms are of two types: sum of products and product of sums.

Sum of Products

We have used the sum of products form in our earlier work. A sum of products expression is formed as follows. Each row of the truth table in which the function takes on the value 1 contributes an ANDed term. These are called *minterms*. A minterm is defined as an ANDed product of literals in which each variable appears exactly once in either normal or complemented form, but not both. The minterms are then ORed to form the expression for the function. The minterm expression is equivalent since it is derived from the truth table.

The figure below shows a truth table for an arbitrary function *f* and its complement. The minterms and maxterms for each row are also shown. The minterm expressions for *f* and *f* NOT are:

$$f = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$$

$$\bar{f} = \bar{A} \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot \bar{C}$$

Truth table which above is based on:

A	B	C	f	$\bar{f}$	Minterms	Maxterms
0	0	0	0	1	$\bar{A}\cdot\bar{B}\cdot\bar{C} = m_0$	$A+B+C = M_0$
0	0	1	0	1	$\bar{A}\cdot\bar{B}\cdot C = m_1$	$A+B+\bar{C} = M_1$
0	1	0	0	1	$\bar{A}\cdot B\cdot\bar{C} = m_2$	$A+\bar{B}+C = M_2$
0	1	1	1	0	$\bar{A}\cdot B\cdot C = m_3$	$A+\bar{B}+\bar{C} = M_3$
1	0	0	1	0	$A\cdot\bar{B}\cdot\bar{C} = m_4$	$\bar{A}+B+C = M_4$
1	0	1	1	0	$A\cdot\bar{B}\cdot C = m_5$	$\bar{A}+B+\bar{C} = M_5$
1	1	0	1	0	$A\cdot B\cdot\bar{C} = m_6$	$\bar{A}+\bar{B}+C = M_6$
1	1	1	1	0	$A\cdot B\cdot C = m_7$	$\bar{A}+\bar{B}+\bar{C} = M_7$

The above expression can be written in a shorthand notation. Note that the indexing of the Boolean variables is important in deriving the minterm and maxterm. In shorthand notation we have:

$$f(A,B,C) = \sum m_i(3,4,5,6,7) = m_3 + m_4 + m_5 + m_6 + m_7$$

$$\bar{f}(A,B,C) = \sum m_i(0,1,2) = m_0 + m_1 + m_2$$

Where $\sum m_i(\quad)$ means the sum of all the minterms whose subscript i is given inside the parentheses.

The minterm expression is not likely to be the simplest form of the function. The expression for f can be reduced by using Boolean algebra.

$$f = \bar{A}\cdot B\cdot C + A\cdot\bar{B}\cdot\bar{C} + A\cdot\bar{B}\cdot C + A\cdot B\cdot\bar{C} + A\cdot B\cdot C$$

$$= A\cdot\bar{B}\cdot\underbrace{(\bar{C}+C)}_{=1} + \bar{A}\cdot B\cdot C + A\cdot B\cdot\underbrace{(\bar{C}+C)}_{=1}$$

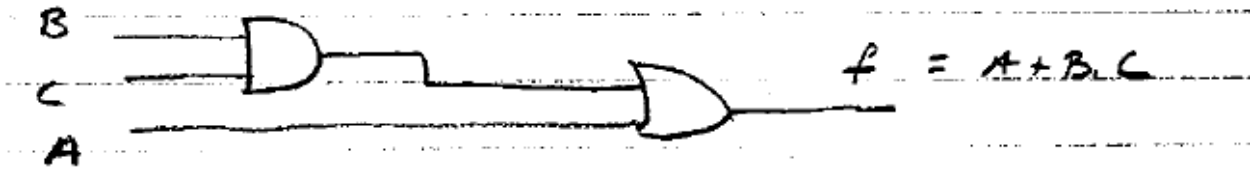
$$= A \cdot \bar{B} + \bar{A} \cdot B \cdot C + A \cdot B$$

$$= A \cdot (\underbrace{\bar{B} + B}_{=1}) + \bar{A} \cdot B \cdot C$$

$$= A + \bar{A} \cdot B \cdot C$$

$$f = A + B \cdot C \quad \text{using 22}$$

The minimized gate-level implementation of *f* is shown below:



The expression *f NOT* can also be reduced:

$$\bar{f} = \bar{A} \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot \bar{C}$$

$$= \bar{A} \cdot \bar{B} \cdot (\underbrace{\bar{C} + C}_{=1}) + \bar{A} \cdot B \cdot \bar{C}$$

$$= \bar{A} \cdot \bar{B} + \bar{A} \cdot B \cdot \bar{C}$$

$$= \bar{A} \cdot (\bar{B} + B \cdot \bar{C})$$

$$= \bar{A} \cdot (\bar{B} + \bar{C}) \quad \text{using 22}$$

$$= \bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C}$$

Product of Sums

A product of sums expressions is formed as follows. Each row of the truth table in which the function takes on the value 0 contributes an ORed therm. These are called *maxterms*. A maxterm is defined as an ORed sum of literals in which each variable appears exactly once in either true or completed form, but not both. The maxterms are then ANDed to form the expression for the function. This is opposite to the way we formed minterms.

The products of sum of functions *f* and *f NOT* is obtained from the truth table as:

$$f = (A+B+C)\cdot(A+B+\bar{C})\cdot(A+\bar{B}+C)$$

$$\bar{f} = (A+\bar{B}+\bar{C})\cdot(\bar{A}+B+C)\cdot(\bar{A}+B+\bar{C})\cdot(\bar{A}+\bar{B}+C)\cdot(\bar{A}+\bar{B}+\bar{C})$$

Using a shorthand notation we can write *f* and *f NOT* as:

$$f(A,B,C) = \pi M_i(0,1,2) = M_0 \cdot M_1 \cdot M_2$$

$$\bar{f}(A,B,C) = \pi M_i(3,4,5,6,7) = M_3 \cdot M_4 \cdot M_5 \cdot M_6 \cdot M_7$$

Where $\pi M_i(\quad)$ means the product of all the maxterms whose subscript *I* is given inside the parentheses.

Conversion Between Canonical Forms

One canonical form can be mapped into the other by applying De Morgan's Theorem. For example if we apply DeMorgan's Theorem to the minterm expansion of *f NOT* we get:

$$\bar{f} = \bar{A}\cdot\bar{B}\cdot\bar{C} + \bar{A}\cdot\bar{B}\cdot C + \bar{A}\cdot B\cdot\bar{C}$$

$$\bar{\bar{f}} = \overline{\bar{A}\cdot\bar{B}\cdot\bar{C} + \bar{A}\cdot\bar{B}\cdot C + \bar{A}\cdot B\cdot\bar{C}}$$

$$= \left(\overline{\bar{A}\cdot\bar{B}\cdot\bar{C}}\right)\cdot\left(\overline{\bar{A}\cdot\bar{B}\cdot C}\right)\cdot\left(\overline{\bar{A}\cdot B\cdot\bar{C}}\right) \quad \text{using 19}$$

Or:

$$f = \left(A+B+C\right)\cdot\left(A+B+\bar{C}\right)\cdot\left(A+\bar{B}+C\right)$$

Which is the maxterm expansion of *f*. Similarly applying DeMorgan's Theorem to them maxterm expansion of *f NOT* gives:

$$\bar{f} = \left(A+\bar{B}+\bar{C}\right)\cdot\left(\bar{A}+B+C\right)\cdot\left(\bar{A}+B+\bar{C}\right)\cdot\left(\bar{A}+\bar{B}+C\right)\cdot\left(\bar{A}+\bar{B}+\bar{C}\right)$$

$$\bar{\bar{f}} = \overline{\left(A+\bar{B}+\bar{C}\right)\cdot\left(\bar{A}+B+C\right)\cdot\left(\bar{A}+B+\bar{C}\right)\cdot\left(\bar{A}+\bar{B}+C\right)\cdot\left(\bar{A}+\bar{B}+\bar{C}\right)}$$

$$= \overline{A+\bar{B}+\bar{C}} + \overline{\bar{A}+B+C} + \overline{\bar{A}+B+\bar{C}} + \overline{\bar{A}+\bar{B}+C} + \overline{\bar{A}+\bar{B}+\bar{C}}$$

Or using 19:

$$f = \bar{A}\cdot B\cdot C + A\cdot\bar{B}\cdot\bar{C} + A\cdot\bar{B}\cdot C + A\cdot B\cdot\bar{C} + A\cdot B\cdot C$$

Which is the maxterm expansion of *f*.

The minimized product of sums form can be found by starting with the minimized sum of products form of *f NOT* and using DeMorgan's Theorem.

$$\bar{f} = \bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C}$$

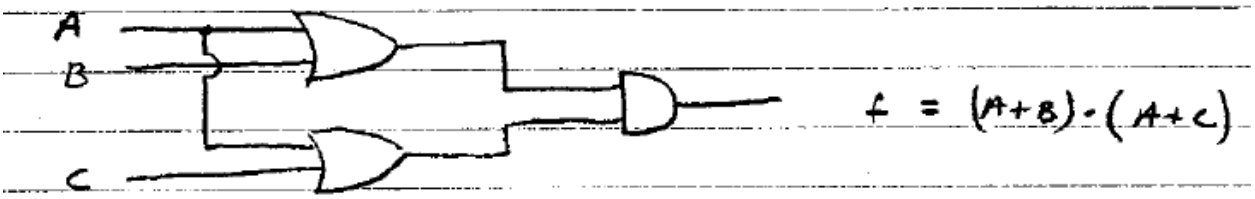
$$\overline{\bar{f}} = \overline{\bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C}}$$

$$= \left(\overline{\bar{A} \cdot \bar{B}}\right) \cdot \left(\overline{\bar{A} \cdot \bar{C}}\right)$$

Or using 20:

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

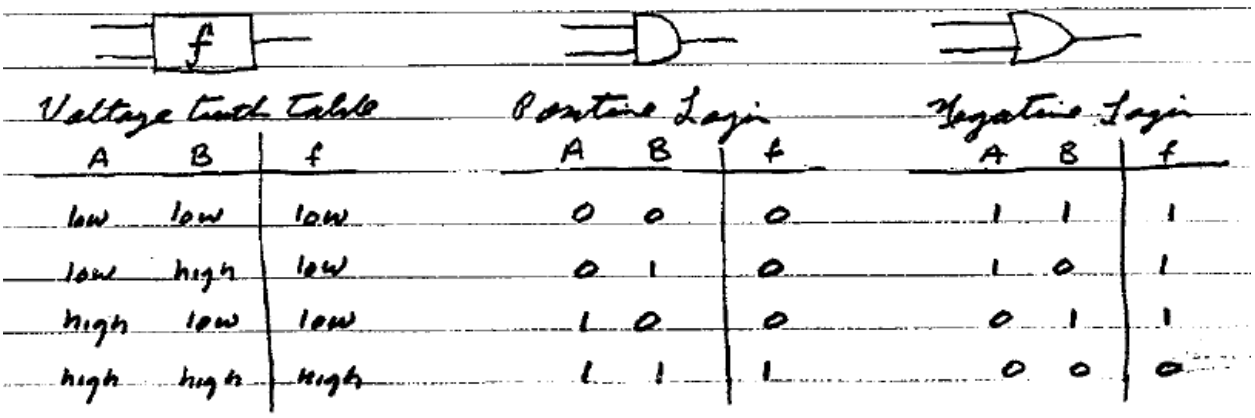
The minimized gate-level implementation is shown below:



Positive Versus Negative Logic

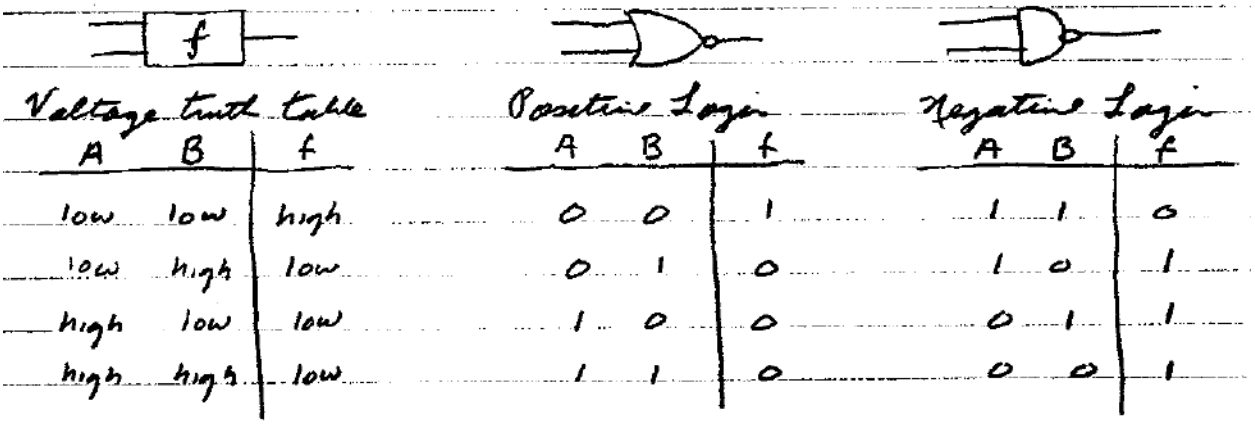
So far, we have assumed that logic 1 is represented by a higher voltage than logic 0. This is known as *positive* logic. If we use the low voltage to represent the asserted signal and the high voltage to represent the unasserted signal we have *negative* logic.

Consider a truth table given in terms of high and low voltages:



In the positive logic case the truth table describes an AND function, whereas, in the negative logic case we obtain an OR function. This is to be expected since an AND function and an OR function are duals, by replacing 0's in one truth table with 1's in the other, and vice versa.

Given a function is positive logic, the equivalent negative logic can be found by applying duality. For example the dual of the NOR function is the NAND function.



Minimization by Mapping

The optimum form of a logic circuit is desired. The criteria is often ONE of the following:

a) Maximum speed – fastest logic implementation

Or:

b) Minimum cost – fewest gate levels because the number of levels determines the cost of manufacturing and the cost of assembly.

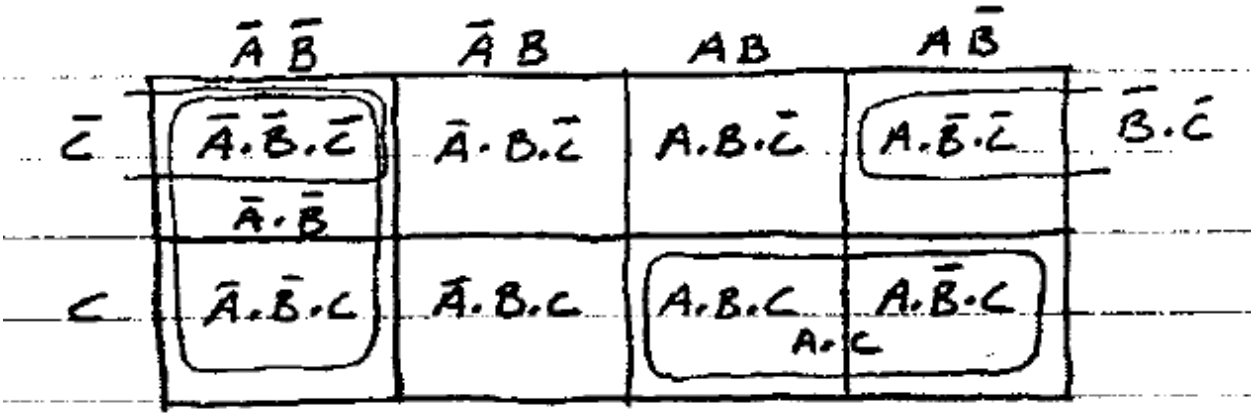
Or:

c) Minimum design time – if only a few circuits are required

Boolean algebra can be used to devise simpler logic expressions. If the truth table is available or if the logic function is expressed as a sum of products we can go directly to a minimum expression by a mapping technique from Maurice Karnaugh.

Karnaugh Maps (K-Maps)

The K-map of the general logic function of three variables is shown below. Each square in the map corresponds to one of the eight possible combinations of the three variables. The order of the columns is such that <u>combinations in adjacent squares different only in the value of one variable</u>.



We see that 2-square groups are independent of one variable. E.g.: for the groups circled:

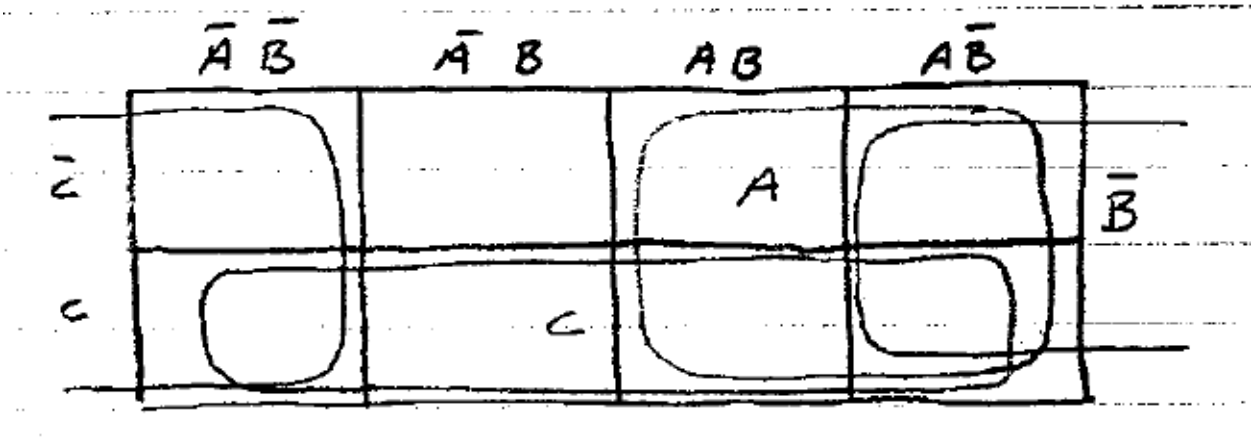
$$f_1 = \bar{A}\cdot\bar{B}\cdot\bar{C} + \bar{A}\cdot\bar{B}\cdot C = (\bar{A}\cdot\bar{B})\cdot\underbrace{(\bar{C}+C)}_{=1} = \bar{A}\cdot\bar{B}$$

$$f_2 = A\cdot B\cdot C + A\cdot\bar{B}\cdot C = A\cdot(B+\bar{B})\cdot C = A\cdot C$$

$$f_3 = \bar{A}\cdot\bar{B}\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} = (\bar{A}+A)\cdot(\bar{B}\cdot\bar{C}) = \bar{B}\cdot\bar{C}$$

As shown above those relations are easily determined using Boolean algebra, but they are *obvious by inspection* of the K-maps.

We can extend the groupings from adjacent squares as shown below where the labels are omitted from the squares.



We see that 4-square groups are independent of the variables, e.g.:

$$f_4 = \bar{A}\cdot\bar{B}\cdot C + \bar{A}\cdot B\cdot C + A\cdot B\cdot C + A\cdot\bar{B}\cdot C$$

$$= \bar{A}\cdot(\bar{B}+B)\cdot C + A\cdot(B+\bar{B})\cdot C = \bar{A}\cdot C + A\cdot C$$

$$= (\bar{A}+A)\cdot C = C$$

$$f_5 = A\cdot B\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C + A\cdot\bar{B}\cdot C$$

$$= A\cdot(B+\bar{B})\cdot\bar{C} + A\cdot(B+\bar{B})\cdot C = A\cdot\bar{C} + A\cdot C$$

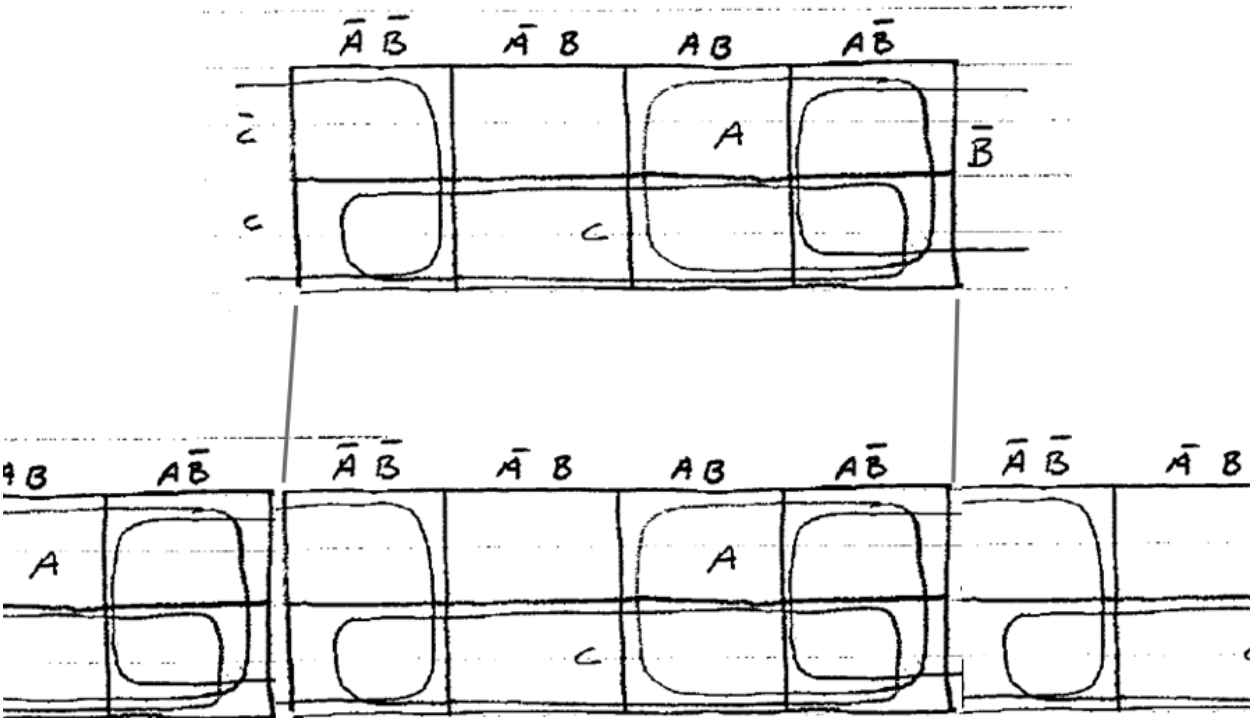
$$= A\cdot(\bar{C}+C) = A$$

$$f_6 = A\cdot\bar{B}\cdot\bar{C} + A\cdot\bar{B}\cdot C + \bar{A}\cdot\bar{B}\cdot\bar{C} + \bar{A}\cdot\bar{B}\cdot C$$

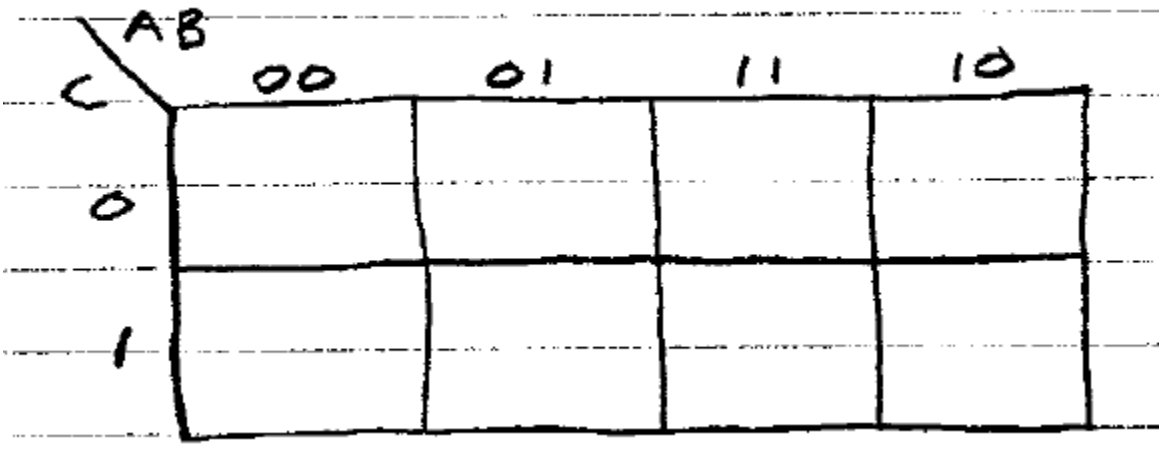
$$= A\cdot\bar{B}\cdot(\bar{C}+C) + \bar{A}\cdot\bar{B}\cdot(\bar{C}+C) = A\cdot\bar{B} + \bar{A}\cdot\bar{B}$$

$$= (A+\bar{A})\cdot\bar{B} = \bar{B}$$

Enlarging groups by overlapping simplifies the table. Note that the map is continuous, in that the last column on the right is “adjacent” to the first column in the left:



The standard labeling for K-maps (shown below) is convenient for mapping from the truth table.



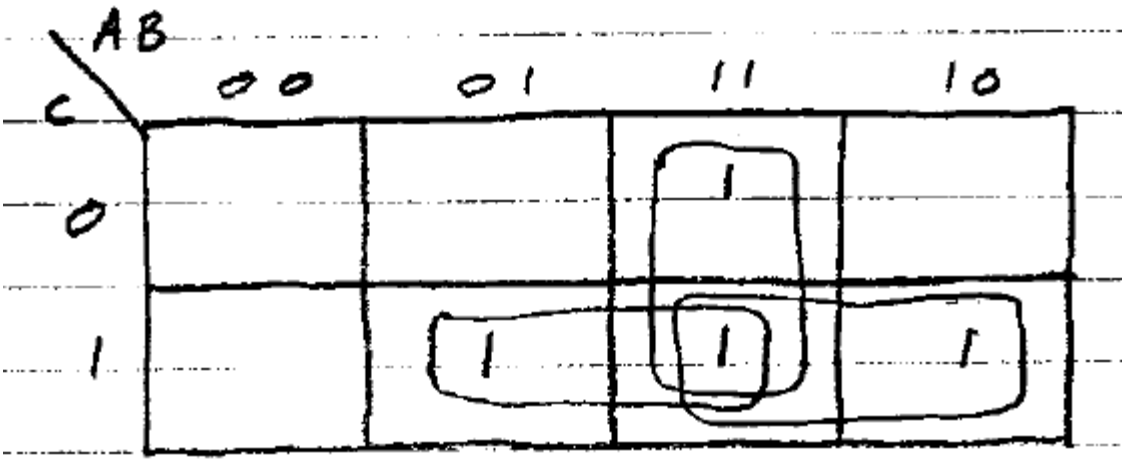
Each square in the map corresponds to a row in the truth table. A specific logic function is mapping by placing a 1 in each square for which the function is 1. Possible simplifications are then easily recognized.

Example

Map the vote-taker function and simplify the circuit realization, if possible. From the truth table of the vote taker function:

A	B	C	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

We first place 1's in the squares corresponding to the tows in the truth table for which the result of the function is 1, resulting:



$$f = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$$

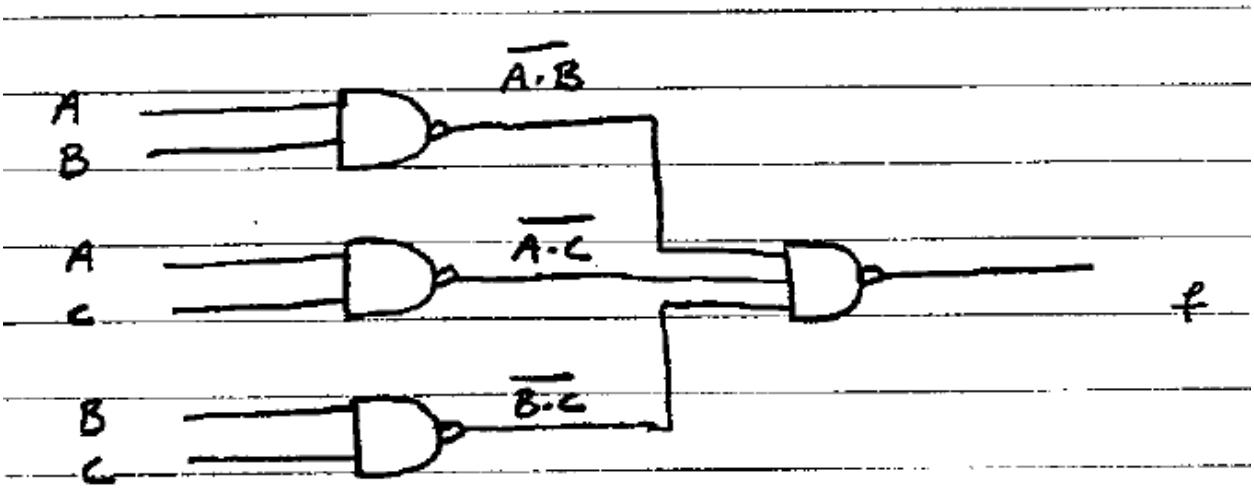
All the 1's can be included in three overlapping 2-square groups. The complete function can be represented by:

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

This is the simplest expression for the function. By using DeMorgan's Theorem, any "sum of products" can be converted into a "NANDed product of NAND's". In this case:

$$f = \overline{(\overline{A \cdot B}) \cdot (\overline{A \cdot C}) \cdot (\overline{B \cdot C})}$$

Which can be synthesized using NAND gates only:

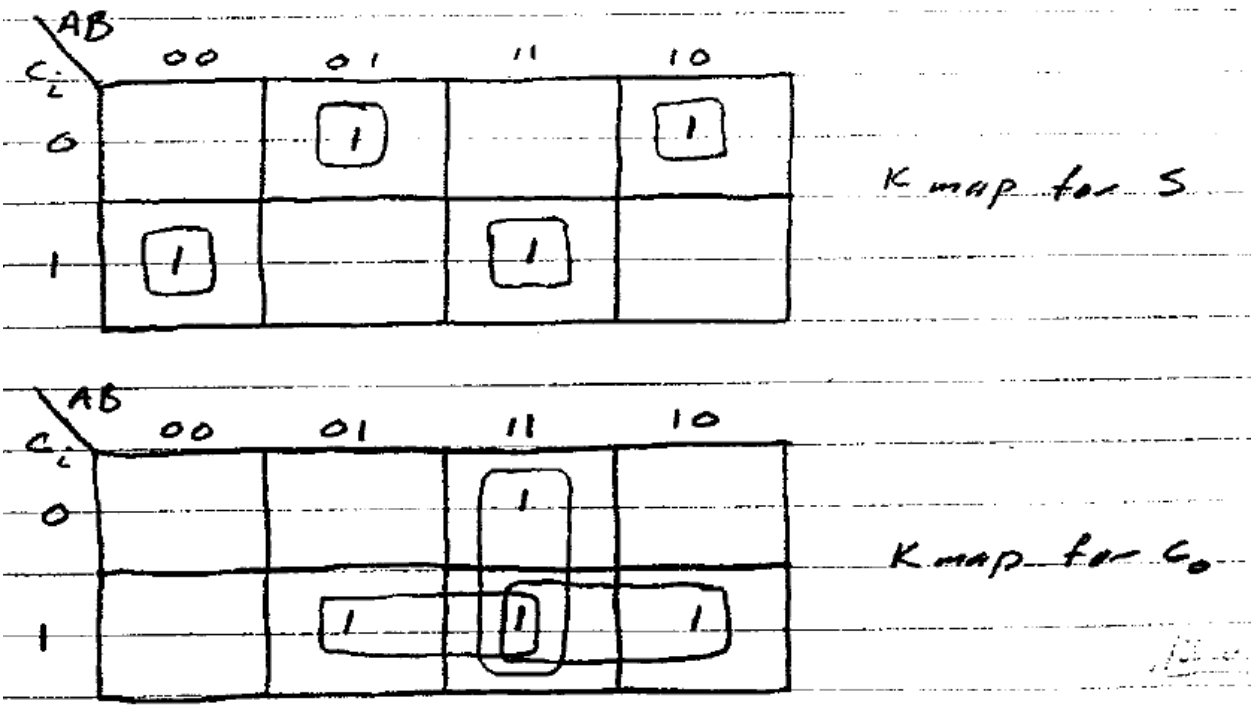


Example

Map the full-adder sum and conjugate functions. Obtain the simplest forms of the function. The truth table is as follows:

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

Because there is two output functions, we have two separate K-maps:



From the K-maps we see that the simplest form for the sum is given by:

$$S = \bar{A} \cdot \bar{B} \cdot C_i + \bar{A} \cdot B + \bar{C}_i + A \cdot B \cdot C_i + A \cdot \bar{B} \cdot \bar{C}_i$$

And the simplest form for the conjugate is given by:

$$C_o = A \cdot B + B \cdot C_i + A \cdot C_i$$

Both of these results agree with the earlier results. Note that the conjugate function is the same as the vote-taker function of the last example.

Mapping in Four Variables

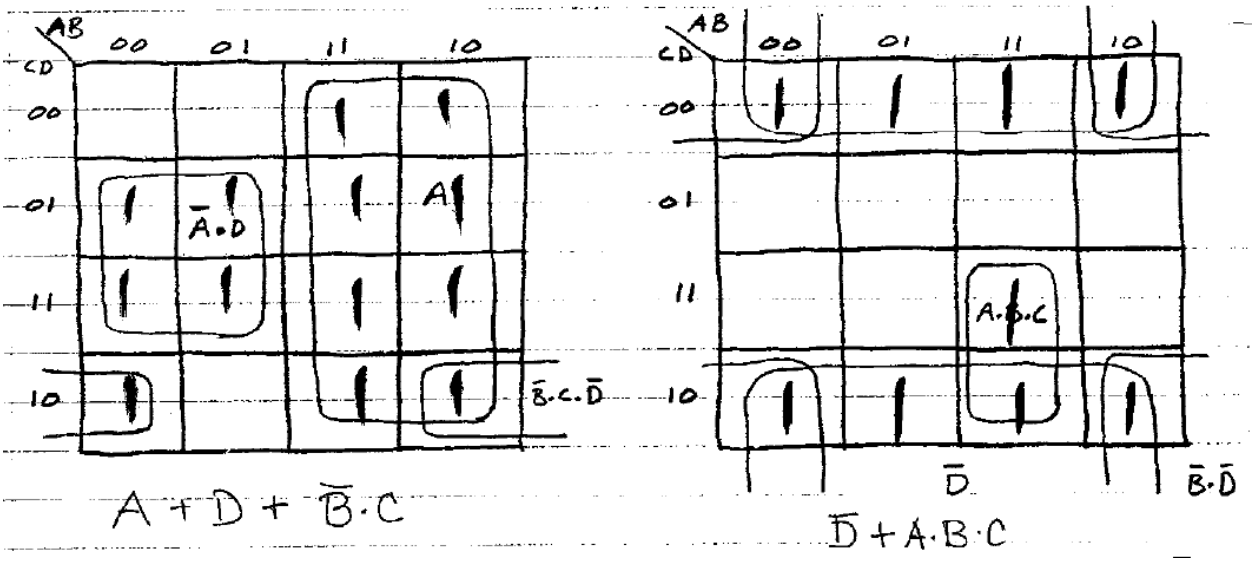
K-maps are useful in simplification functions involving four variables. Typically once more than four variables are involved it becomes easier to use other techniques. A four-variable K-map is shown below:

	$\bar{A}\,\bar{B}$	$\bar{A}\,B$	AB	$A\,\bar{B}$
$\bar{C}\,\bar{D}$	$\bar{A}\cdot\bar{B}\cdot\bar{C}\cdot\bar{D}$	$\bar{A}\cdot B\cdot\bar{C}\cdot\bar{D}$	$A\cdot B\cdot\bar{C}\cdot\bar{D}$	$A\cdot\bar{B}\cdot\bar{C}\cdot\bar{D}$
$\bar{C}\,D$	$\bar{A}\cdot\bar{B}\cdot\bar{C}\cdot D$	$\bar{A}\cdot B\cdot\bar{C}\cdot D$	$A\cdot B\cdot\bar{C}\cdot D$	$A\cdot\bar{B}\cdot\bar{C}\cdot D$
$C\,D$	$\bar{A}\cdot\bar{B}\cdot C\cdot D$	$\bar{A}\cdot B\cdot C\cdot D$	$A\cdot B\cdot C\cdot D$	$A\cdot\bar{B}\cdot C\cdot D$
$C\,\bar{D}$	$\bar{A}\cdot\bar{B}\cdot C\cdot\bar{D}$	$\bar{A}\cdot B\cdot C\cdot\bar{D}$	$A\cdot B\cdot C\cdot\bar{D}$	$A\cdot\bar{B}\cdot C\cdot\bar{D}$

As indicated in the figures below (where standard labeling for K-maps is used):

- 2-square groups are independent of one variable
- 4-square groups are independent of two variables
- 8-square groups are independent of three variables

Note that adjacent rows different by only one complement bar, and the bottom row is adjacent to the top row with the left column adjacent to the right.



The four corner squares for the groups $\bar{B} \cdot \bar{D}$.

Some general guidelines for finding the minimal expression for a K-map are:

a) Include all 1's in groups of eight, four, two, or one.
b) Groups may overlap; larger groups result in simpler terms
c) Of the possible selection of terms, select the simplest

Example

Map the function:

$$f = \overline{(A \cdot B)} \cdot \overline{(C + D)} + \bar{A} \cdot C \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D$$

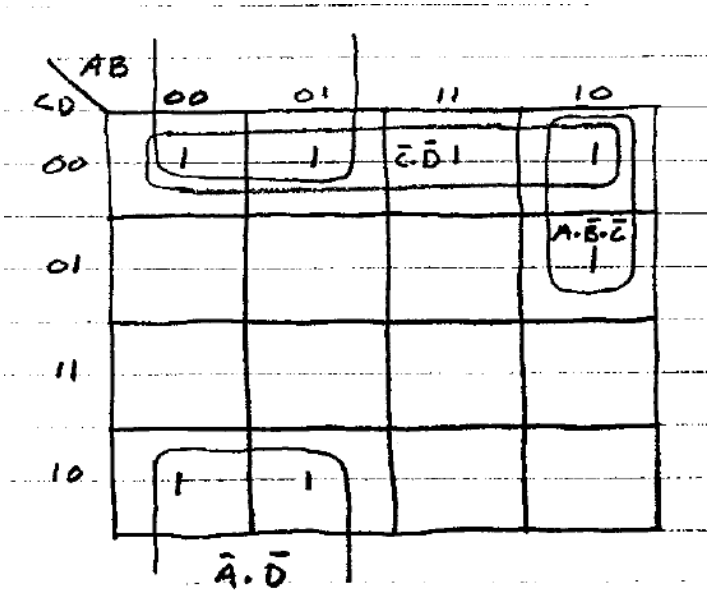
And obtain a minimum sum of products expression.

Using DeMorgan's theorem the given expression can be written as:

$$f = (\bar{A} + \bar{B}) \cdot (\bar{C} \cdot \bar{D}) + \bar{A} \cdot C \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D$$

$$= \bar{A} \cdot \bar{C} \cdot \bar{D} + \bar{B} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot C \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D$$

The K-map is given below:



All the 1's can be included in two 4-square and one 2-square groups. Thus:

$$f = \bar{A} \cdot \bar{D} + \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C}$$

Note that the other expressions are possible but none with fewer, simpler terms.

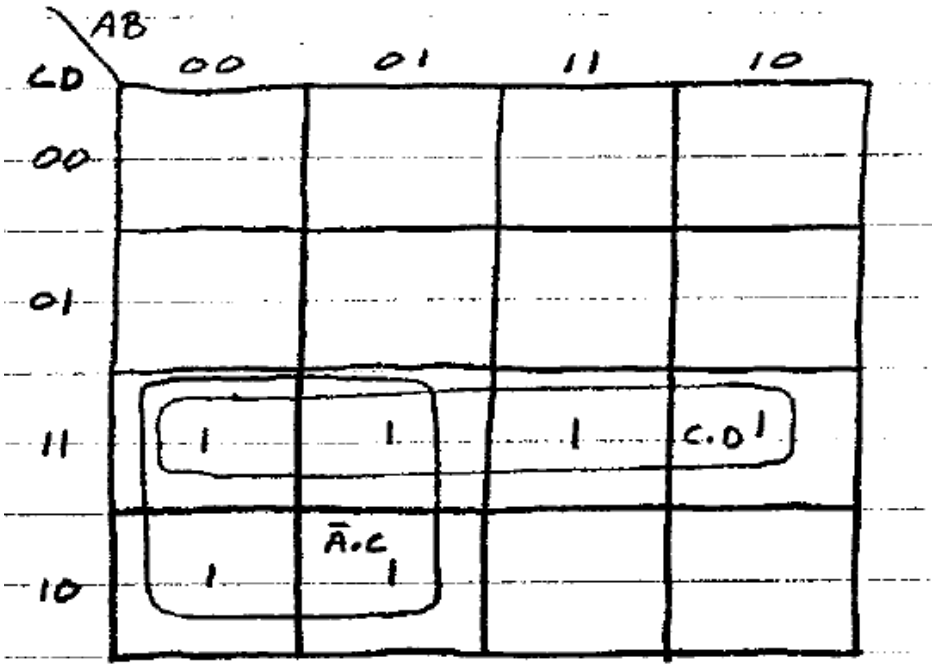
Example

Map the function:

$$f = A \cdot \bar{B} \cdot C \cdot D + \bar{A} \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot C + B \cdot C \cdot D$$

And find the minimal sum of products form

The K-Map is shown below:



From the map we see that the minimum f is:

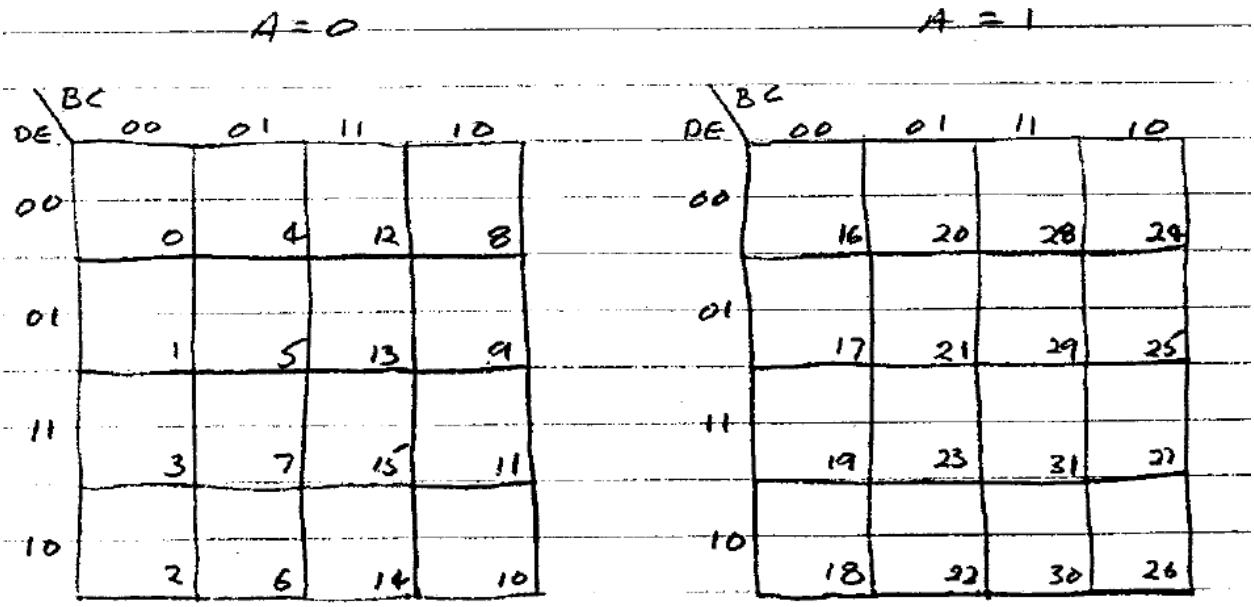
$$f = \bar{A} \cdot C + C \cdot D$$

Again other arrangements are possible, but not minimal.

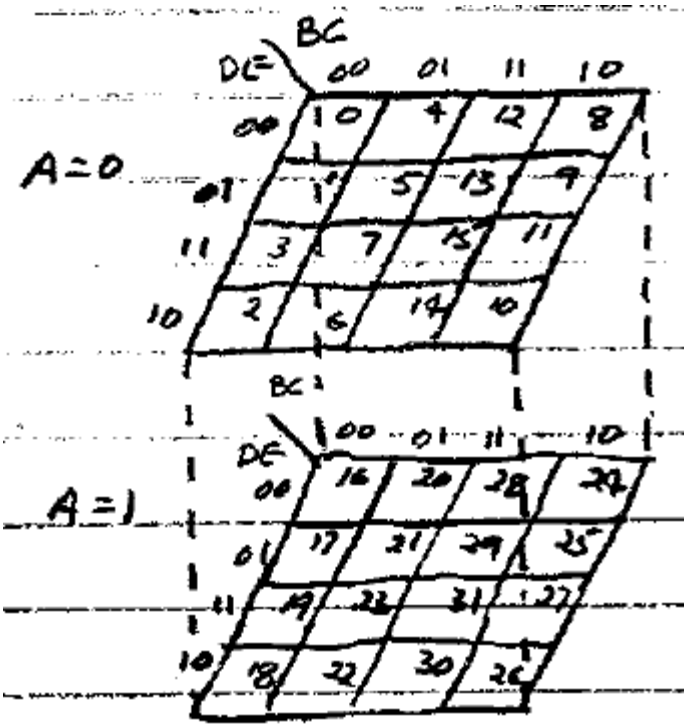
Five-Variable Maps

Maps for more than four variables are not as simple to use. A five-variable map needs 32 squares and a six-variable map needs 64 squares. With a large number of variables the number of squares is large and the geometry for combining adjacent squares is made convoluted.

The five-variable map shown below consist of 2 four-variable maps with variables, A,B,C,D, and E. Variable A distinguishes between the two maps. The left-hand four-variable map represents the 16 squares where A=0, and the other four-variable map represents the squares where A=1. Minterms 0 through 15 belong with A=0, and minterms 16 through 31 with A=1. Note that the numbering of the minterms is important.



Each four-variable map retains the previously defined adjacency when taken separately. In addition, each square in the A-0 map is adjacent to the corresponding square in the A=1 maps. For example, minterm 4 is adjacent to minterm 20 and minterm 15 to 31. The best way to visualize this new adjacency rile is to cascade the two half maps as being one on top of the other. Any two squares that fall one over the other are considered adjacent:

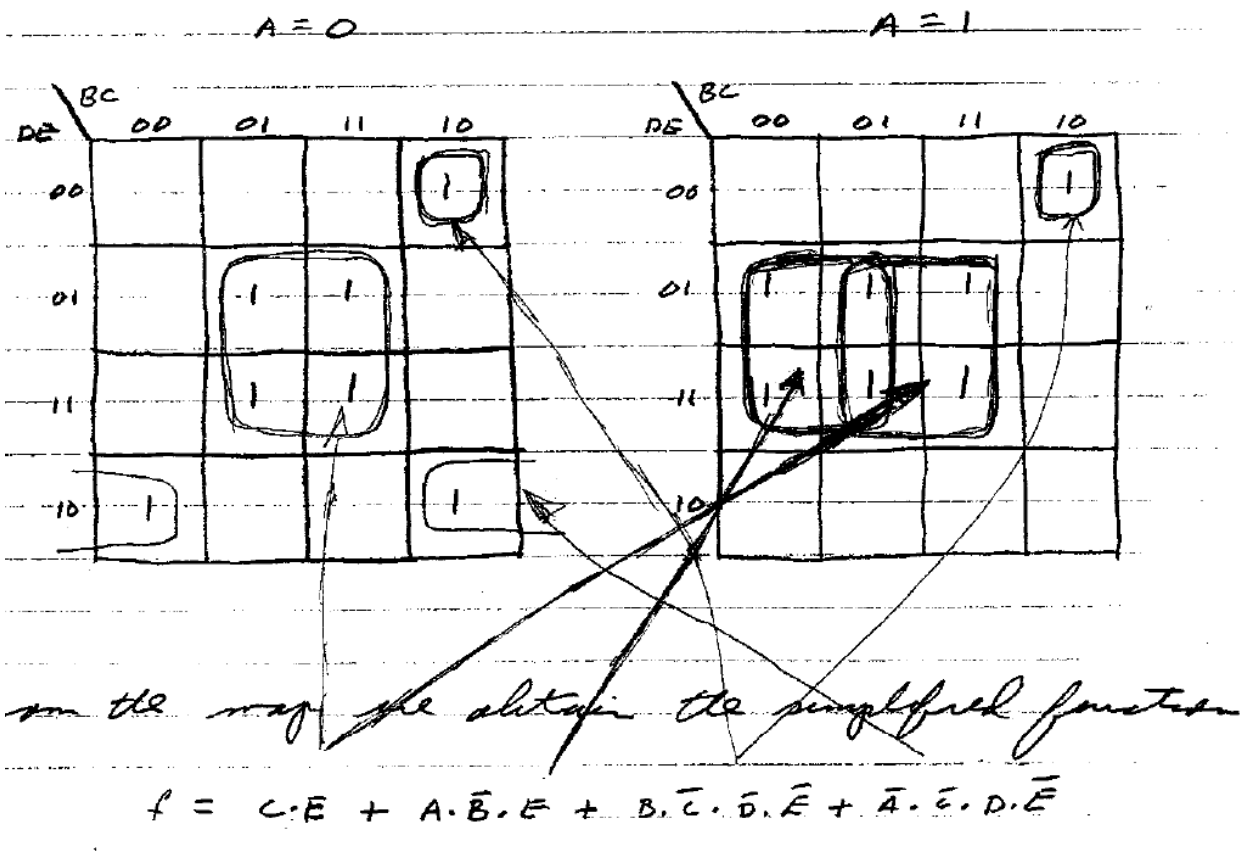


Example

Simplify the Boolean function:

$$f(A,B,C,D,E) = \sum_{m} (2,5,7,8,10,13,15,17,19,21,23,24,29,31)$$

The filled in K-map is shown below, along with the simplified function:

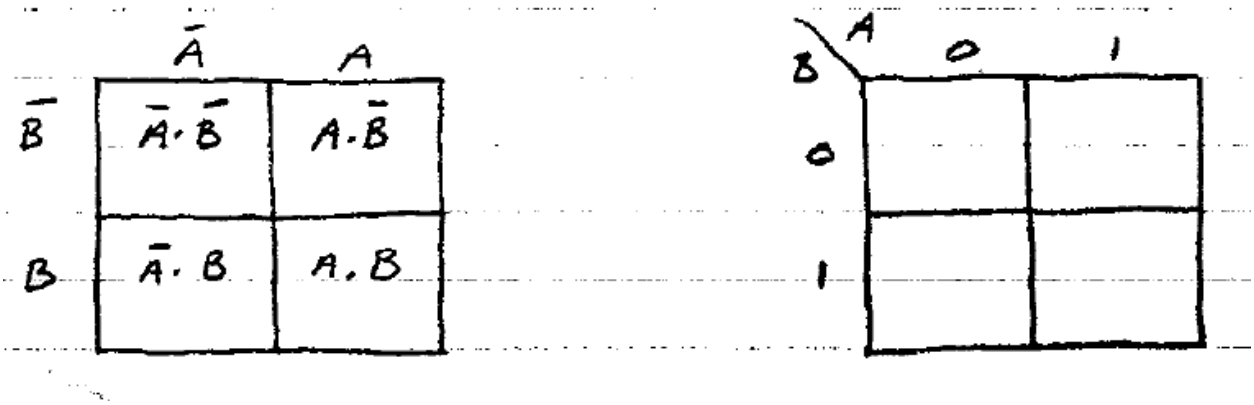


Note the redundant agency $\bar{A} \cdot B \cdot \bar{C} \cdot \bar{E}$ has been omitted.

By following the procedure used for the five variable maps, it is possible to construct a six-variable map using four of the 4-variable maps to obtain the required 64 squares. For maps with N variables one must check for adjacencies in N directions. Maps with six or more variables need too many squares and are impractical to use. It is simpler to use computer programs written to simplify Boolean functions with a large number of variables.

Comments on Maps

1. Two variable K-maps are shown below:



2. An N-Variable K-map has 2^N cells.

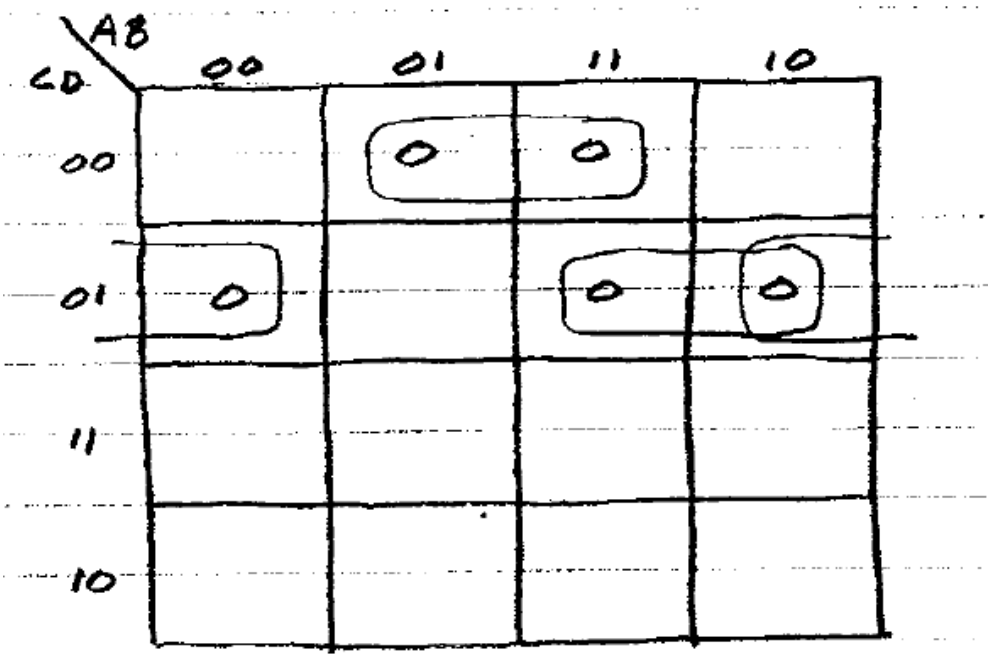
3. In some circuits, certain combinations of inputs never occur. These *don't care* combinations map be mapped as X's and considered as either 0's or 1's, whichever provides the greatest simplification.

4. In some circuits the simplest realization results from finding *f NOT* as the sum of products and then inverting the result to obtain *f*.

5. The K-map can be used to find the minimum product of sums expression. In this case we collect the maximal adjacent group of 0's and write the functions complement in the sum of product forms. Applying DeMorgan's Theorem we get the product of sums form.

Example of Points 4&5:

Given the following K-Map:



Find *f* in the minimum product of sums form.

From the map we see that:

$$\bar{f} = B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{C} \cdot D + \bar{B} \cdot \bar{C} \cdot D$$

Therefor:

$$\bar{\bar{f}} = \overline{B \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{C} \cdot D + \bar{B} \cdot \bar{C} \cdot D}$$

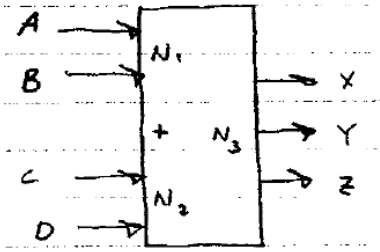
Using DeMorgan's Theorem we get:

$$f = \left(\overline{B \cdot \bar{C} \cdot \bar{D}}\right) \cdot \left(\overline{A \cdot \bar{C} \cdot D}\right) \cdot \left(\overline{\bar{B} \cdot \bar{C} \cdot D}\right)$$

$$= \left(\bar{B} + C + D\right) \cdot \left(\bar{A} + C + \bar{D}\right) \cdot \left(B + C + \bar{D}\right)$$

Example

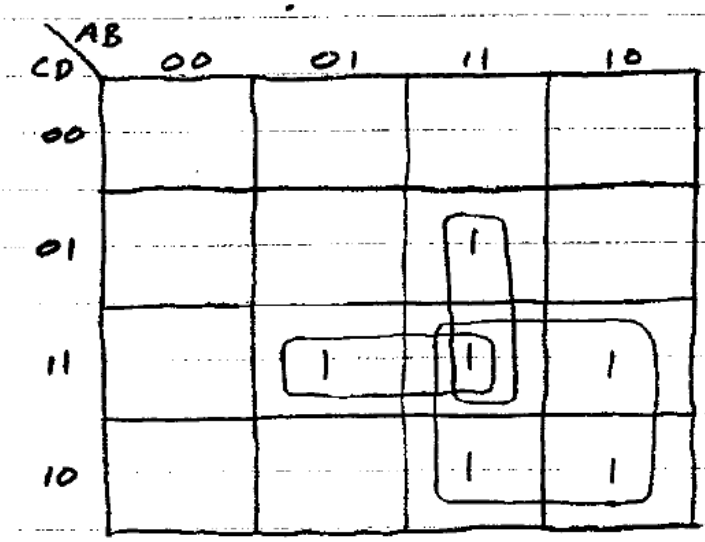
Use K-maps to synthesize a 2-bit binary adder whose diagram and truth table are given below:



A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

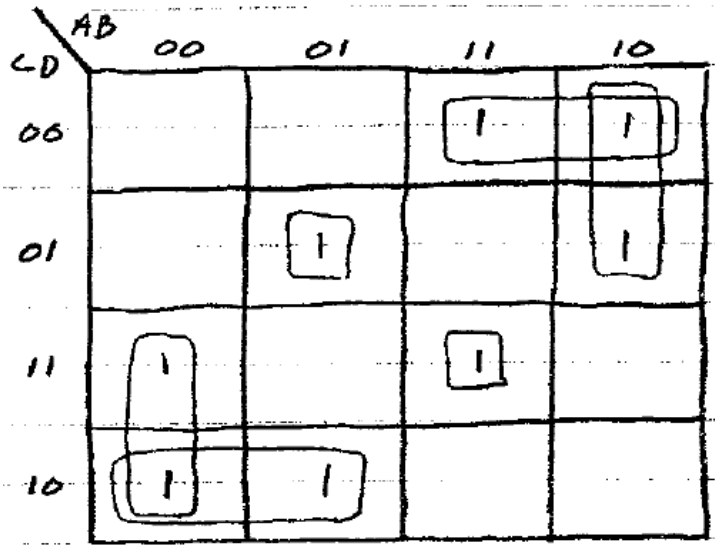
The adder has two 2-bit binary numbers N1 and N2 as inputs and produces a 3-bit number, N3, as an output. In the truth table N1 is represented by the inputs A&B, and N2 by C&D. The output is represented by the Boolean function X,Y, and Z.

The K-maps for the outputs are shown below. From the maps we can write the function X, Y, and Z:



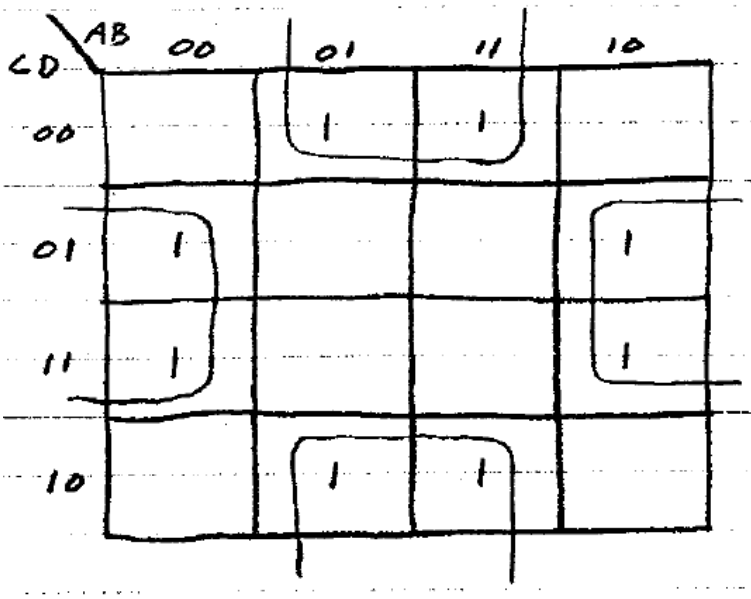
K-map for X

$$X = A \cdot B \cdot D + B \cdot C \cdot D + A \cdot C$$



K-map for Y

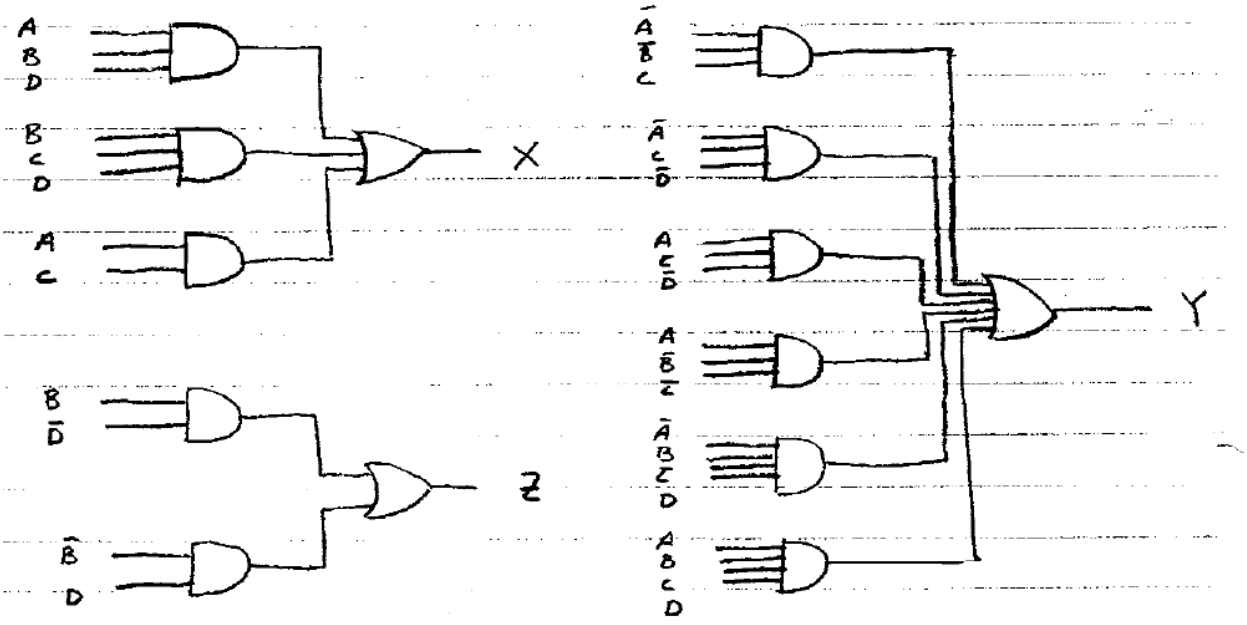
$$Y = \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot C \cdot \bar{D} + A \cdot \bar{C} \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} \cdot D + A B C \cdot D$$



K-map for Z

$$Z = B.\bar{D} + \bar{B}.D = B \oplus D$$

The functions can be synthesized as shown below:

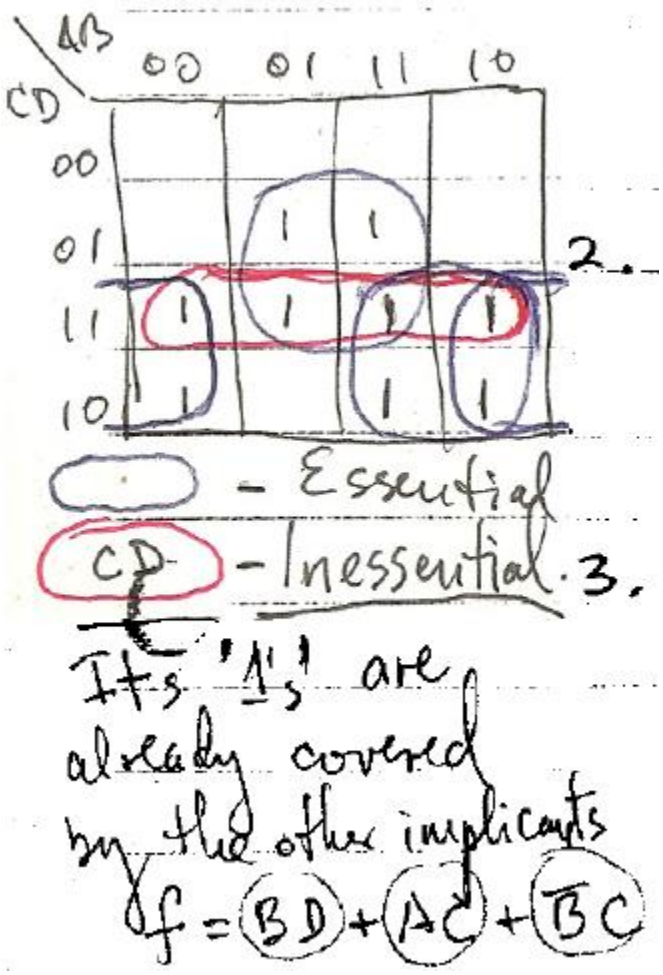


Some More Notes: Implicants

1) An *implicant* of a function *f* is a single or group of elements that can be combined together in a K-map.

2) A *prime implicant* is an implicant that cannot be combined with another one to eliminate a literal.

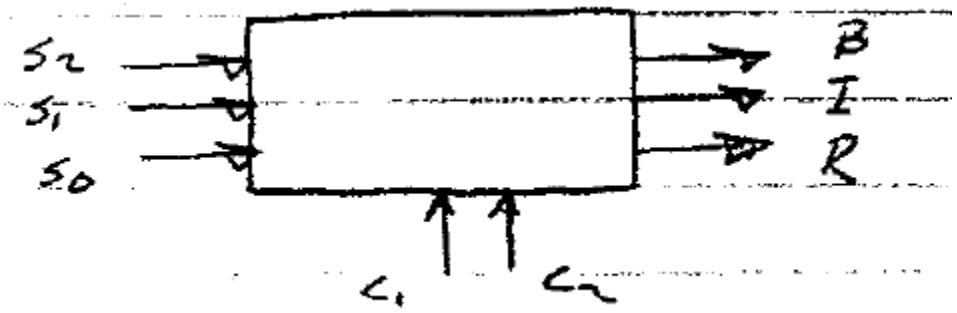
3) If a particular element is covered by a single prime implicant, it is called an *essential prime implicant*.



Example

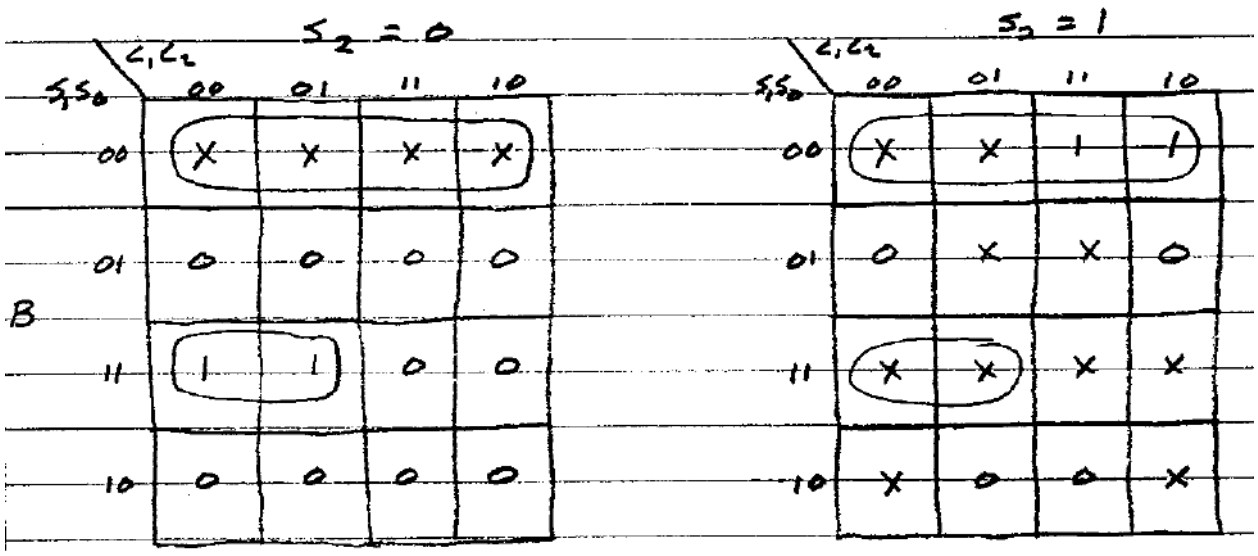
The logic box for a controller with inputs S2, S1, S0, C1, and C2 has to be designed using combinational logic gates. For the truth tables

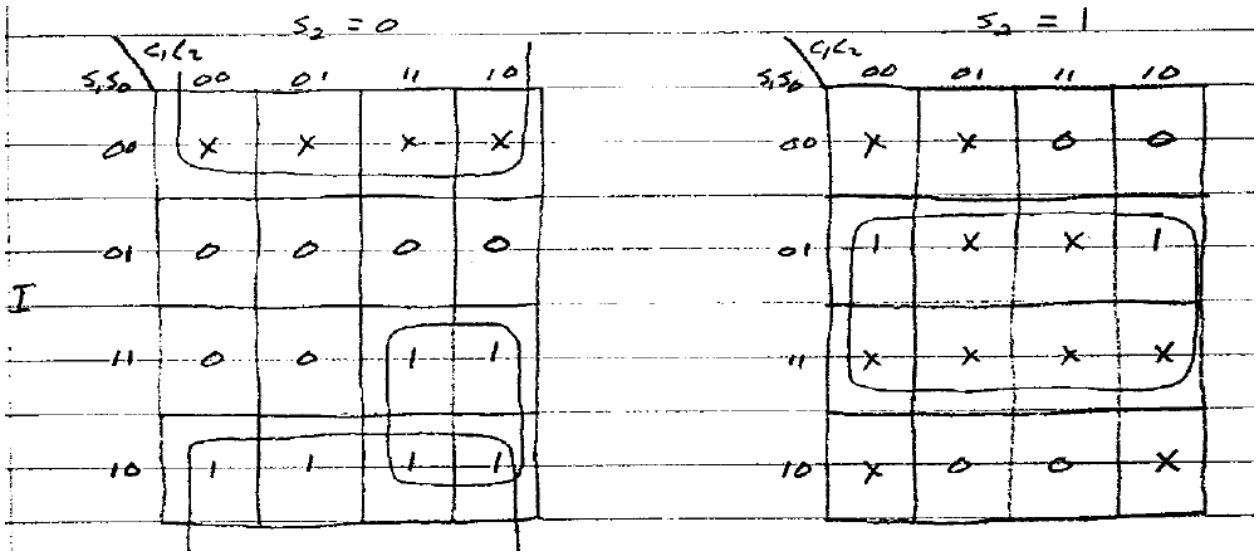
given below where B, I and R are outputs. Use the five variable K map procedure to draw the minimum circuit necessary to complete the table.

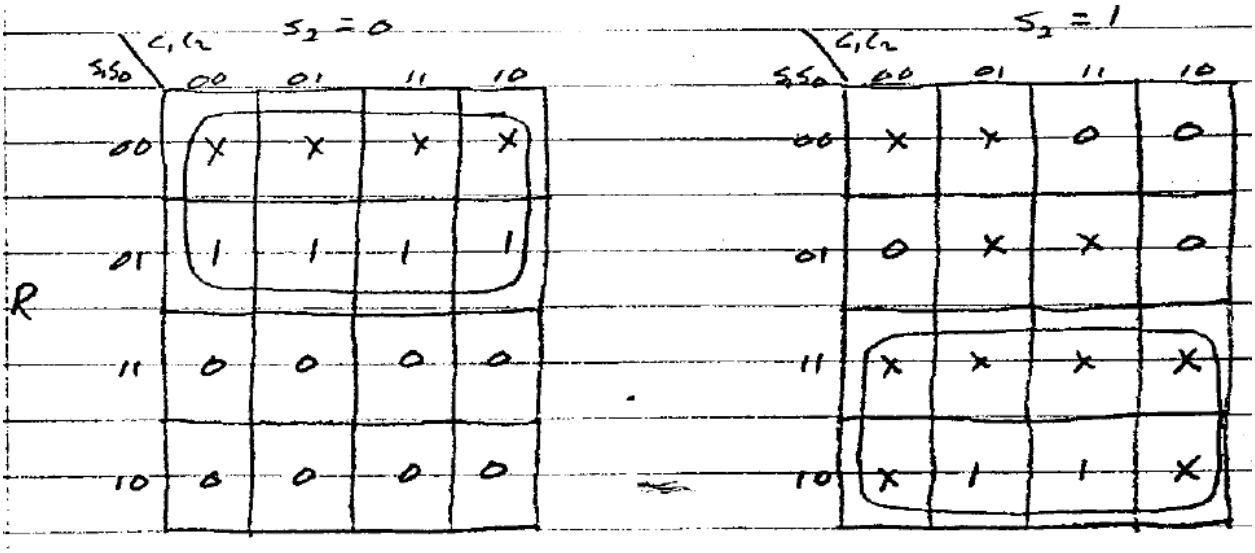


S_2	S_1	S_0	C_1	C_2	B	I	R
0	0	1	X	X	0	0	1
0	1	0	X	X	0	1	0
0	1	1	0	X	1	0	0
1	0	0	1	X	1	0	0
1	0	1	X	0	0	1	0
1	1	0	X	1	0	0	1
0	1	1	1	X	0	1	0

The resulting K-Maps are shown below:


$S_2 = 0$
C_1C_2
S_1S_0
00 01 11 10
00 X X X X
01 0 0 0 0
B
11 1 1 0 0
10 0 0 0 0
$S_2 = 1$
C_1C_2
S_1S_0
00 01 11 10
00 X X 1 1
01 0 X X 0
11 X X X X
10 X 0 0 X


$S_2 = 0$
C_1C_2
S_1S_0
00 01 11 10
00 X X X X
01 0 0 0 0
I
11 0 0 1 1
10 1 1 1 1
$S_2 = 1$
C_1C_2
S_1S_0
00 01 11 10
00 X X 0 0
01 1 X X 1
11 X X X X
10 X 0 0 X



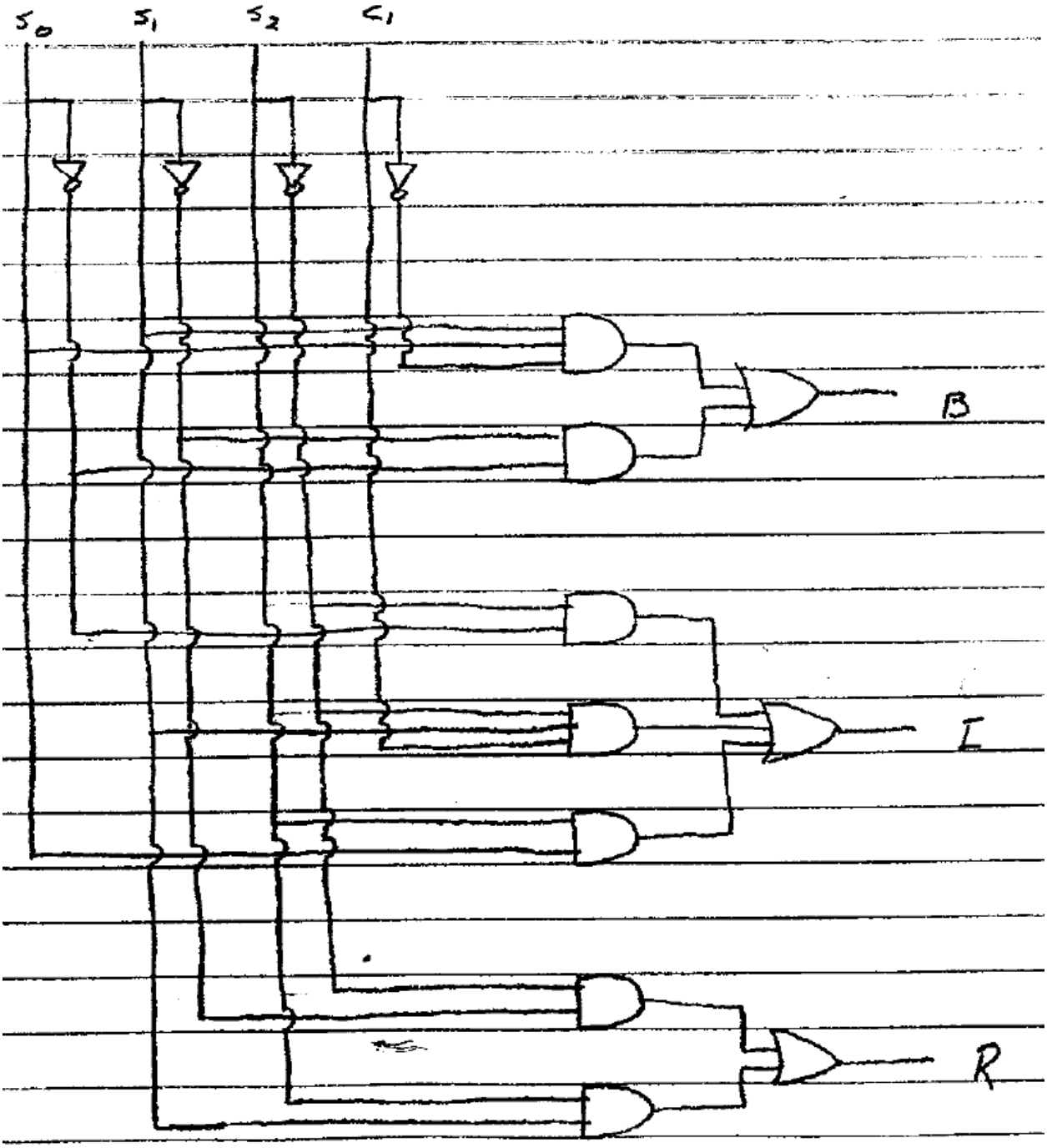
From the K-maps we see that:

$$B = S_1 \cdot S_0 \cdot \bar{C}_1 + \bar{S}_1 \cdot \bar{S}_0$$

$$L = \bar{S}_2 \cdot \bar{S}_0 + \bar{S}_2 \cdot S_1 \cdot C_1 + S_0 \cdot S_0$$

$$R = \bar{S}_0 \cdot \bar{S}_1 + S_0 \cdot S_1 = S_0 \odot S_1$$

The minimum circuit is shown below:

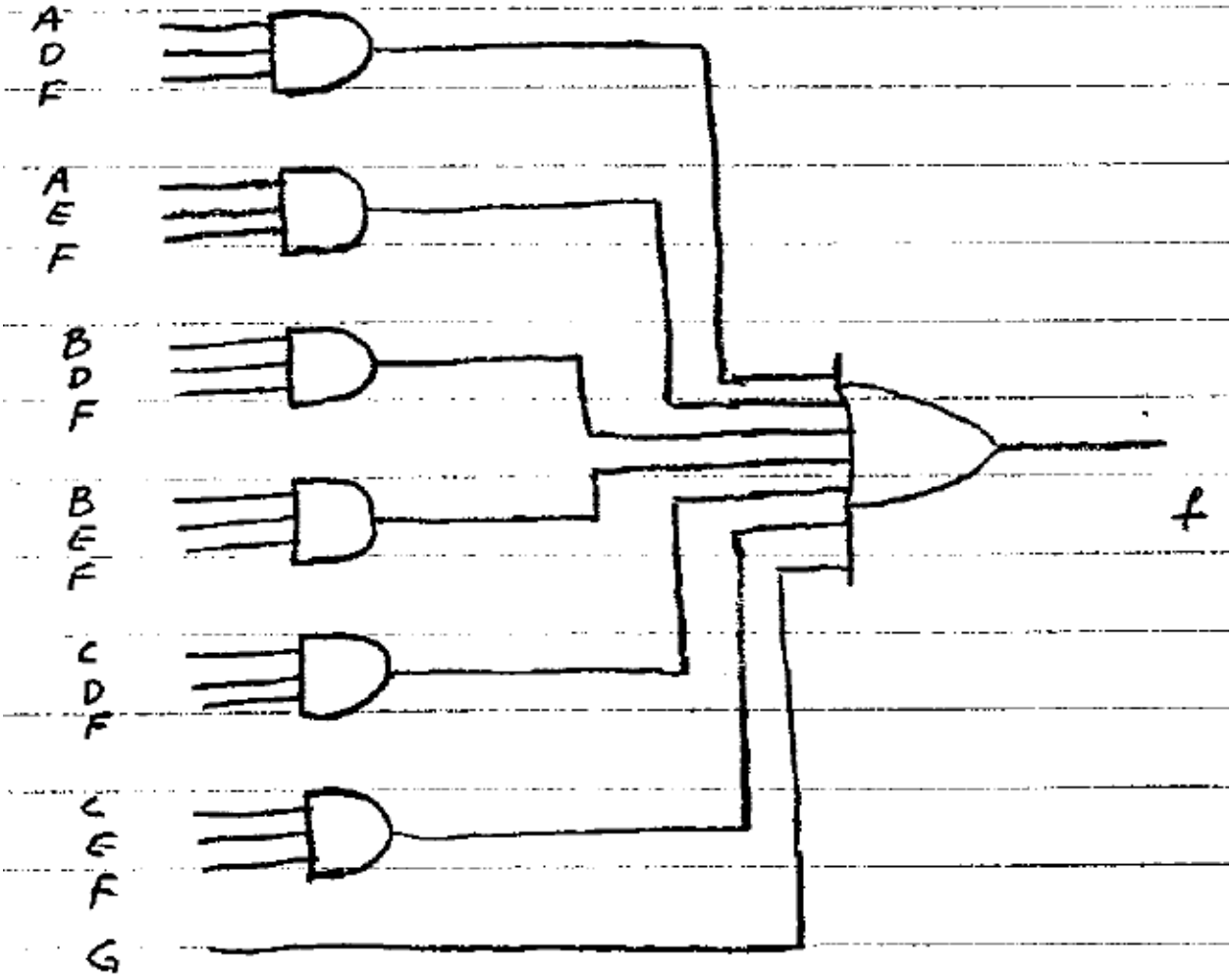

S_0
S_1
S_2
C_1
B
I
R

Multilevel Combinational Logic

Consider the function:

$$f = A \cdot D \cdot F + A \cdot E \cdot F + B \cdot D \cdot F + B \cdot E \cdot F + C \cdot D \cdot F + C \cdot E \cdot F + G$$

Which is in its numerical sum of products form. The corresponding logic circuit is shown below:



We see that as a two-level network of AND and OR gates it requires six 3-input AND gates and one 7-input OR gate for a total of seven gates and 19 literals.

We can replace the two-level form with a factored form as follows:

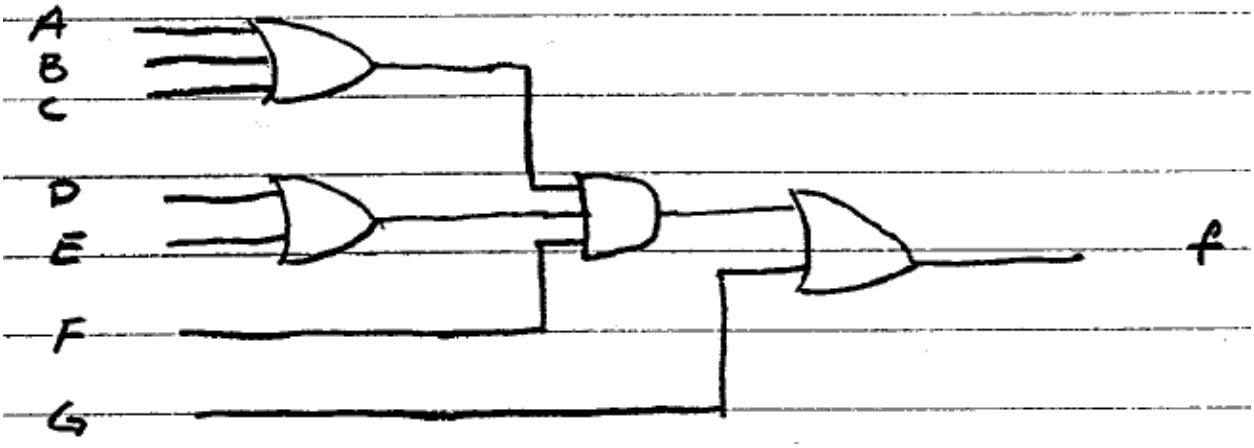
$$f = (A \cdot D + A \cdot E + B \cdot D + B \cdot E + C \cdot D + C \cdot E) \cdot F + G$$

$$= [(A+B+C) \cdot D + (A+B+C) \cdot E] \cdot F + G$$

Or:

$$f = (A+B+C) \cdot (D+E) \cdot F + G$$

The corresponding circuit is shown below:



The result is a three (3) level network which requires one 3-input OR gate, two 2-input OR gates, and a 3-input AND gate for a total of four gates and seven literals.

We have reduced the number of wires and gates required but this implementation probably has more delay because of the increased levels of logic. In general, multilevel circuits are more gate efficient than the corresponding two-level circuits but have worse propagation delay.

Conversion to NAND and NOR Networks

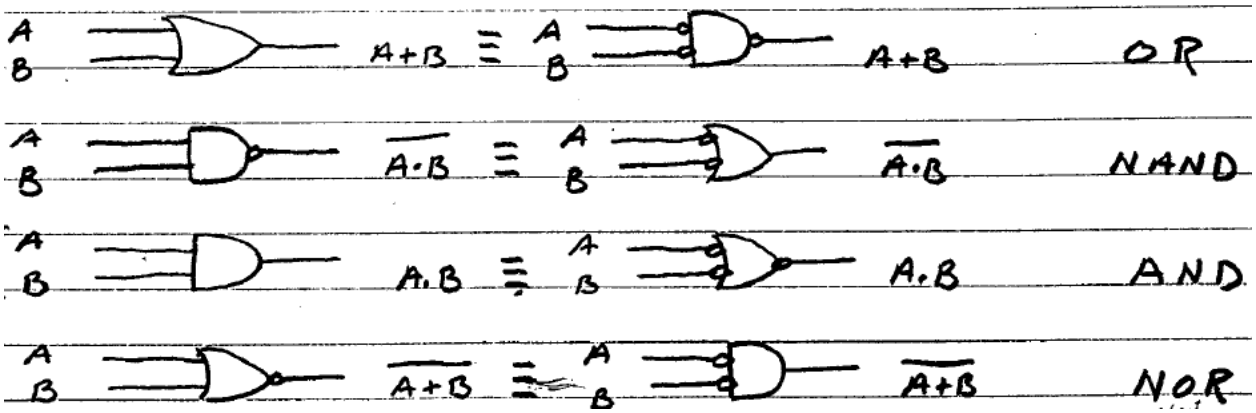
The canonical forms studied so far are expressed in terms of AND and OR gates. In practice it is more efficient to use NAND and NOR gates. We will now see how to map a network with AND and OR gates into that consisting only of NAND or NOR gates.

As can be seen from the truth tables below:

i) An OR gate is logically equivalent to a NAND gate with its inputs inverted
ii) A NAND gate is equivalent to an OR gate with its inputs inverted
iii) An AND gate is equivalent to a NOR gate with its inputs inverted
iv) A NOR gate is equivalent to an AND gate with its inputs inverted

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

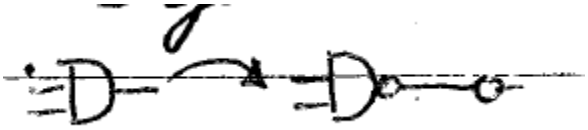
The graphic symbols for each gate are shown below:



To obtain a multilevel NAND circuit from a Boolean expression, proceed as follows:

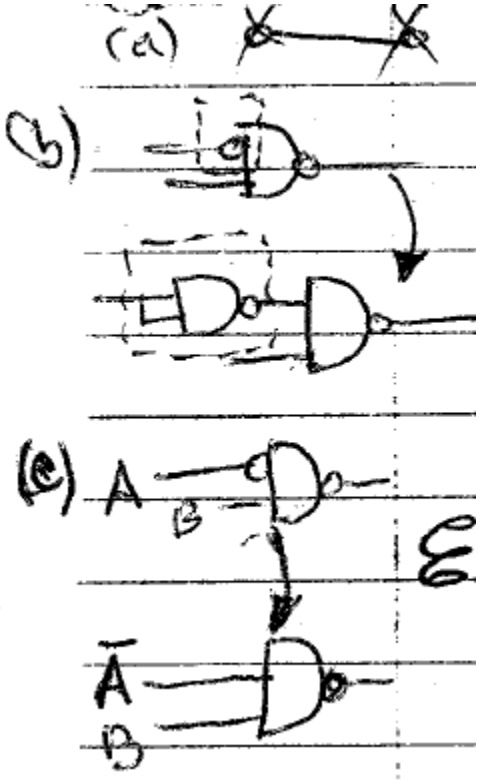
1. From the given Boolean expression, draw the logic diagram with AND, OR, and NOT inverter gates. Assume that both the normal and complement inputs are available.

2. Convert all AND gates to NAND gates with AND-invert graphic symbols:



3. Convert all OR gates to NAND gates with invert-OR graphic symbols:

4. Check all small circles in the diagram. For every small circle that is not compensated by another small circle along the same long, insert an inerter (one-input NAND gate) or complement the input variable:



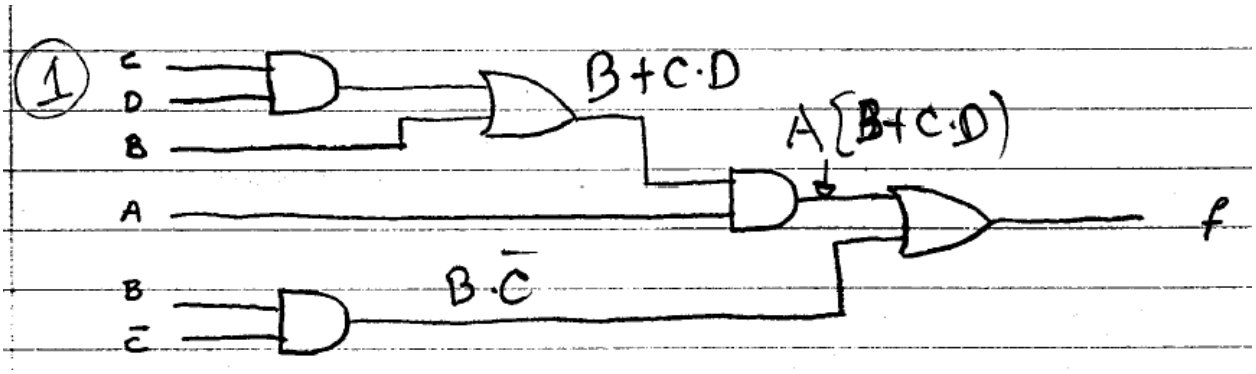
Example

Given the function:

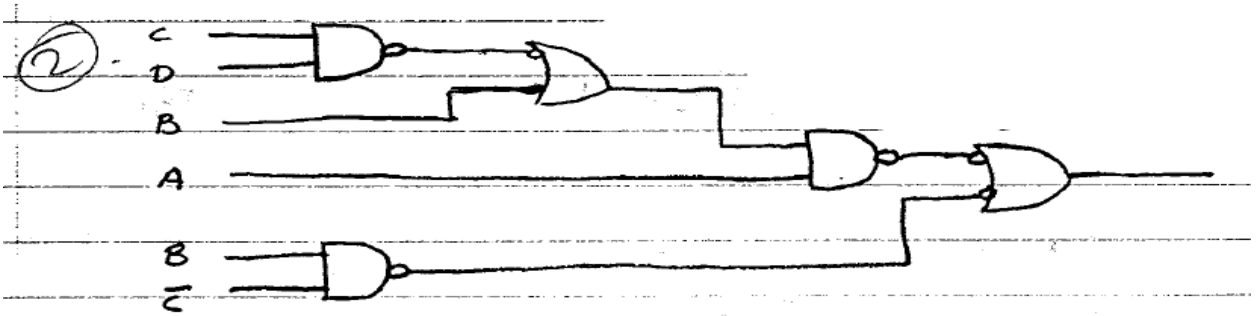
$$f = A \cdot [B + C \cdot D] + B \cdot \bar{C}$$

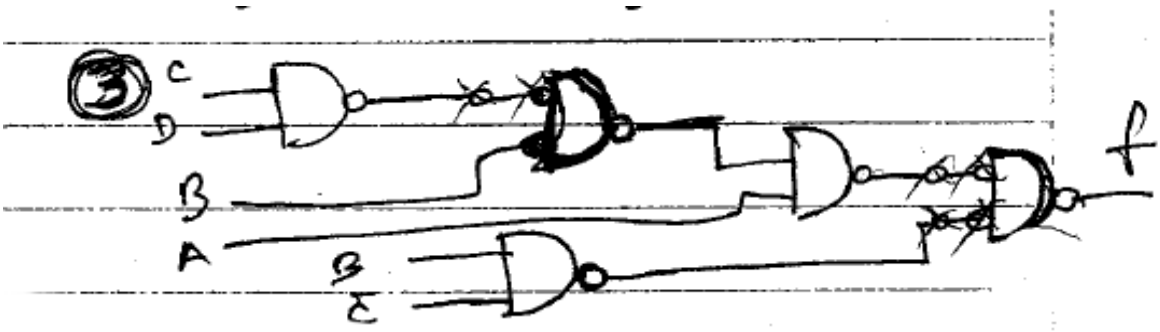
Draw the logic diagram in the AND/OR form and convert to NAND logic.

The AND/OR form is shown below:

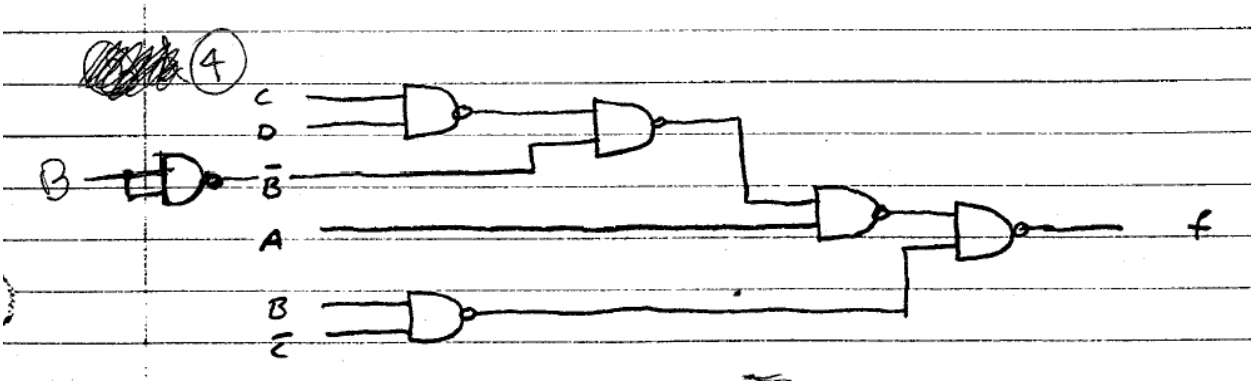


There are four levels of gates in the circuit. Using the procedure given earlier obtain the NAND diagram using two symbols:





Note that the literal B input to the second level NAND gate must be inverted to preserve the original sense of the signal. Since it does not matter whether we use AND-invert or the invert-OR symbols to represent a NAND gate, the diagram below is identical to the one above:

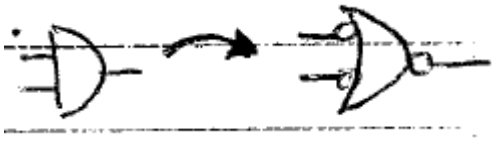


To obtain a multilevel NOR circuit from a Boolean expression, proceed as follows:

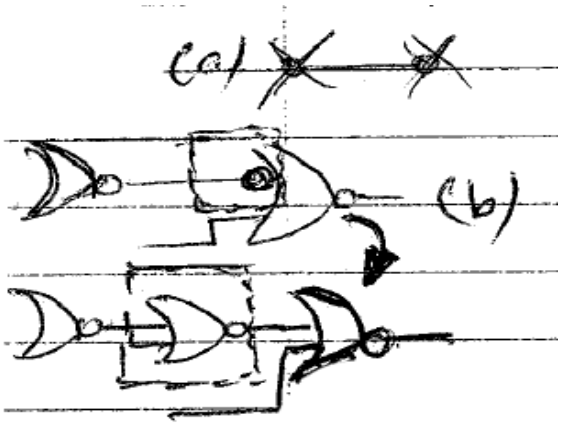
1. Draw the AND-OR logic diagram from the given algebraic expression. Assume that both the normal and complement inputs are available.

2. Convert all OR gates to NOR gates with OR-invert graphic symbols:

3. Convert all AND gates to NOR gates with invert-AND logic symbols:



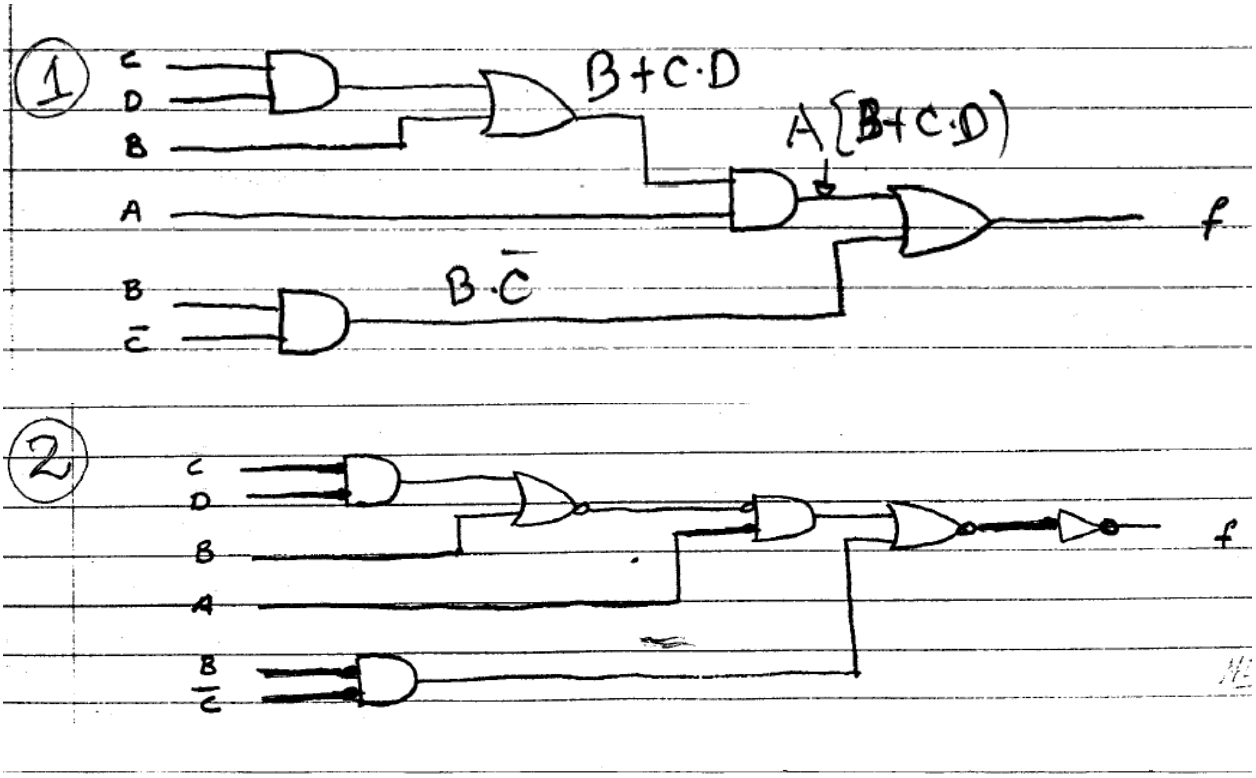
4. Any small circle that is not complemented by another small circle along the same line needs and inverter or the complementation of the input variables.

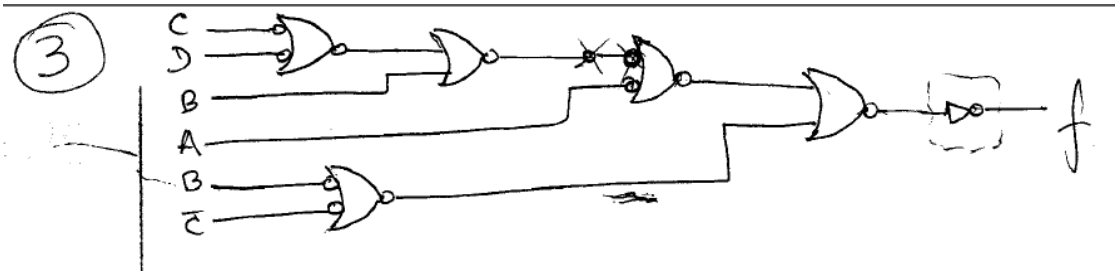


Example

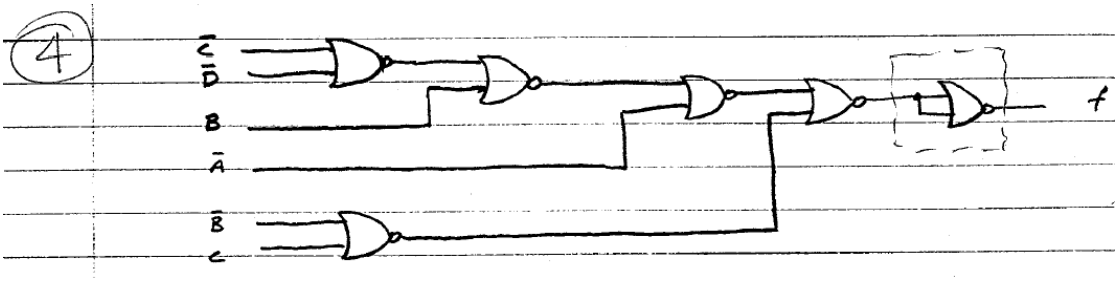
Convert the function f of the last example to NOR logic.

Using the above procedure the AND/OR form is convert to the NOR diagram below:





Note the extra inversion required at the output. The final NOR only circuit is shown below:



The inversion at the output has been implemented by a NOR gate with both inputs tied to the same signal.

Computer Aided Design Tools

Computer-Aided Design (CAD) is used to speed up the high level design process. Besides allowing the exploration of design alternatives, design tools can improve the quality of the design by simulating an implementation before physical construction. Packages such as MIS-II developed at the University of California at Berkley are available for this purpose.

NOTE: Since these notes were written a huge variety of newer tools are available. One of the more popular is Eagle (www.cadsoftusa.com) although it has more limited simulation support. Autotrax (www.kov.com) has fairly good simulation support and an easier user interface.

Time Response in Combination Networks

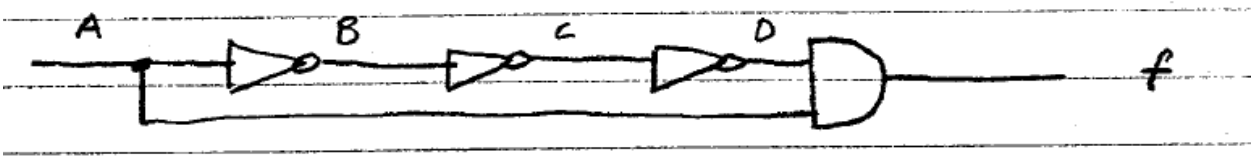
The propagation of signals through a network is not instantaneous. These delays may lead to logical errors at the outputs. Delays come from several sources:

Gate Delays

A gate delay is the amount of time it takes for a change at the gate input to cause a change at the output. Various families of TTL have trade-offs between delay and power. The faster a component, the more power it consumes. Propagation delays often depend on whether the output is going from a low to high (t_{LH}) or from high to low (t_{HL}). For example for the 7400 gate a typical t_{HL} = 7nS and t_{LH} = 11 nS.

Timing Waveforms

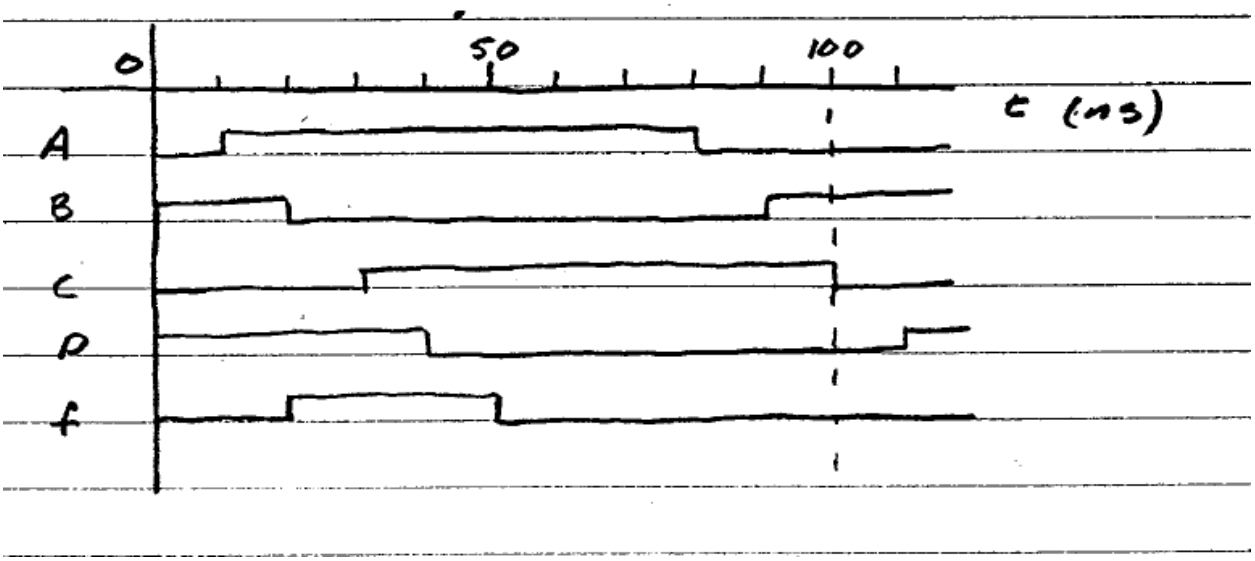
As an example of a timing waveform consider the circuit shown below:



An input signal A passes through three inverters and is then ANDed with the original signal. This implements the function:

$$A \cdot \bar{A} = 0$$

This appears to be a useless function. However, examining the timing diagram below shows that after the input A goes high, the output goes high for a short time before going low. This circuit is called a *pulse shaper*.

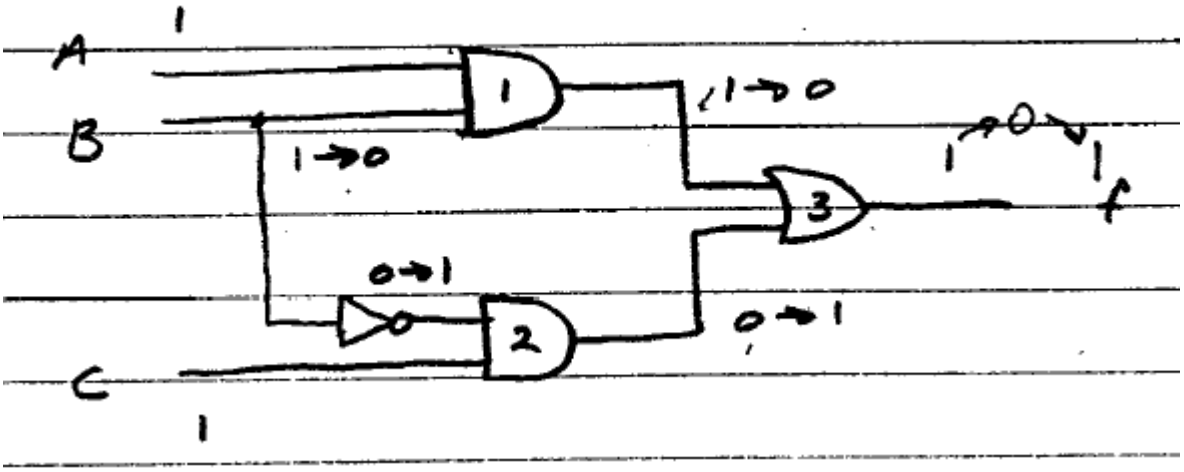


To see how the circuit operations, assume that the initial state has A=0, B=1, C=0, D=1, and f=0 as shown at t=0. Further, assume that each gate has a propagation delay of 10 time units. When A changes from 0 to 1 at time 10, B does not change until time step 20, C at time step 30, and D at time step 40. We see that between time 10 and 40, both A and D are at logic 1. If the AND gate also has a 10-unit gate delay, the output f will be high between time stops 20 and 50. The pulse f is three inverter delays wide. To change the width, use a different number of inverters.

Hazards and Glitches

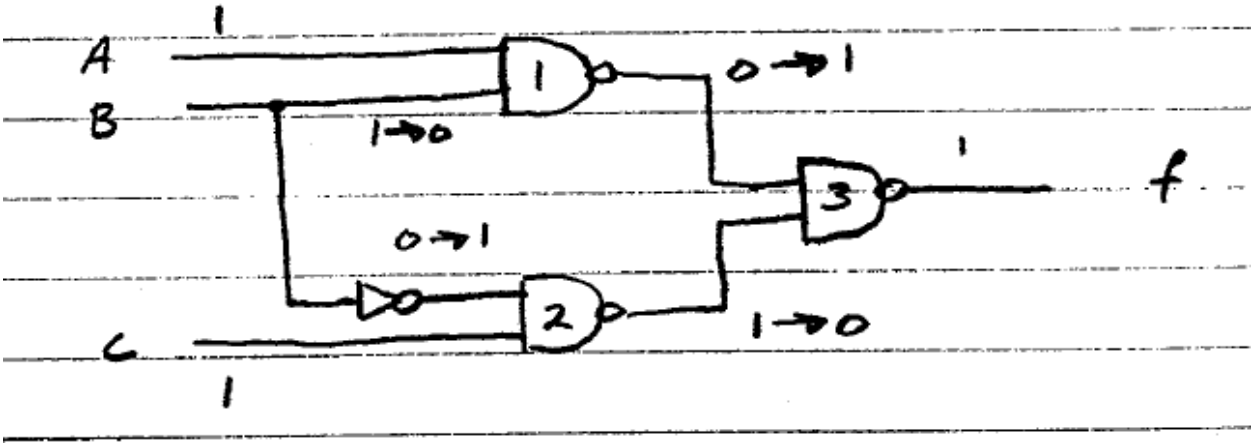
A *glitch* is an unintended pulse at the output of a combinational logic network. A circuit with the potential for a glitch is said to have a *hazard*.

The circuit below demonstrates the occurrence of a hazard. Assume that all inputs are initially 1. The output of gate 1 will then be 1, that of gate 2 will be 0, and the output of the circuit will be 1.



Let B change from 1 to 0. The output of gate 1 changes to 0 and that of gate 2 changes to 1, leaving the output at 1. The output may momentarily go to 0 if the delay through the inverter is large enough. The delay may cause the output of gate 1 to change to 0 before the output of gate 2 changes to 1. In this case both inputs to gate 3 are momentarily equal to 0, causing the output to go to 0 for a short time.

The figure below is a NAND implementation of the same Boolean function. It has a hazard for the same reason:



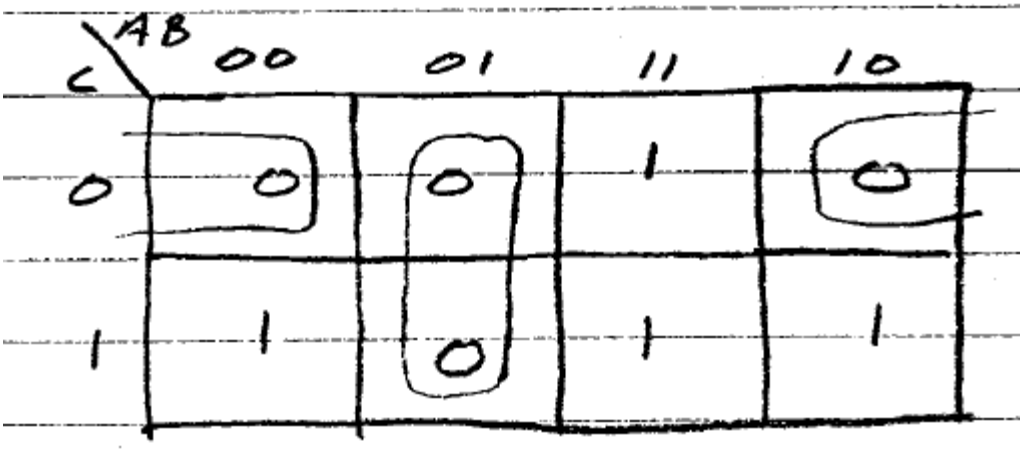
When B changes from 1 to 0, both inputs of gate 3 may equal 1, causing a momentary change to 0 in the output.

The circuits above implement the Boolean function in the sum of products:

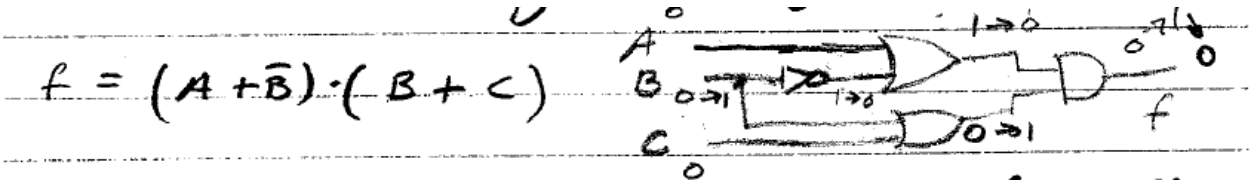
$$f = A \cdot B + \bar{B} \cdot C$$

$$= \overline{(\overline{A \cdot B}) \cdot (\overline{\bar{B} \cdot C})}$$

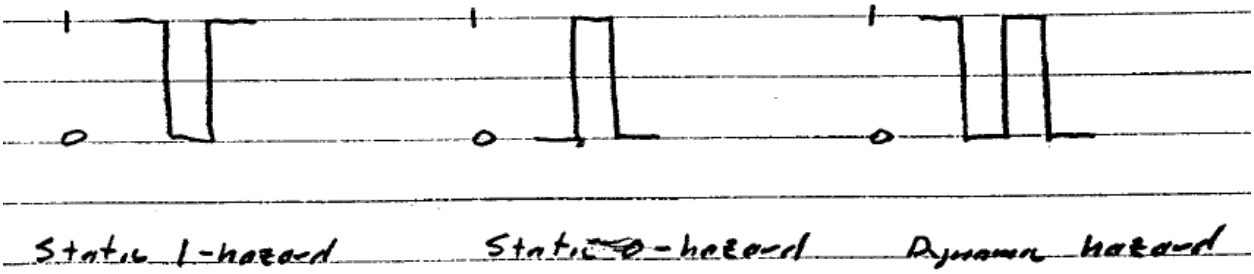
For this circuit the output may go to 0 when it should remain at 1. The K-map for the above circuit is shown below:



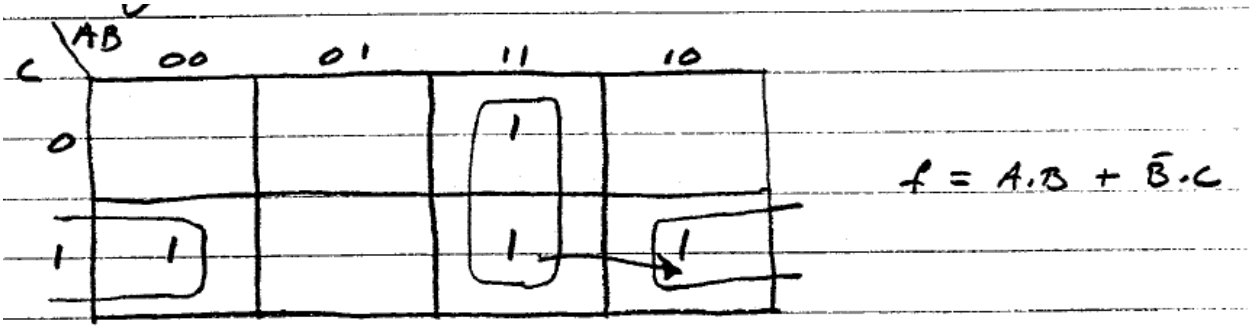
From the zero location the circuit can be implemented in product of sum form:



In this form the output may momentarily go to 1 when it should remain 0. The first case is a *static 1-hazard* and the second case is a *static 0-hazard*. A third type of hazard known as a *dynamic* hazard causes the output to change three or more times when it should change from a 1 to 0 or from a 0 to 1. The figure below shows the three types of hazards:



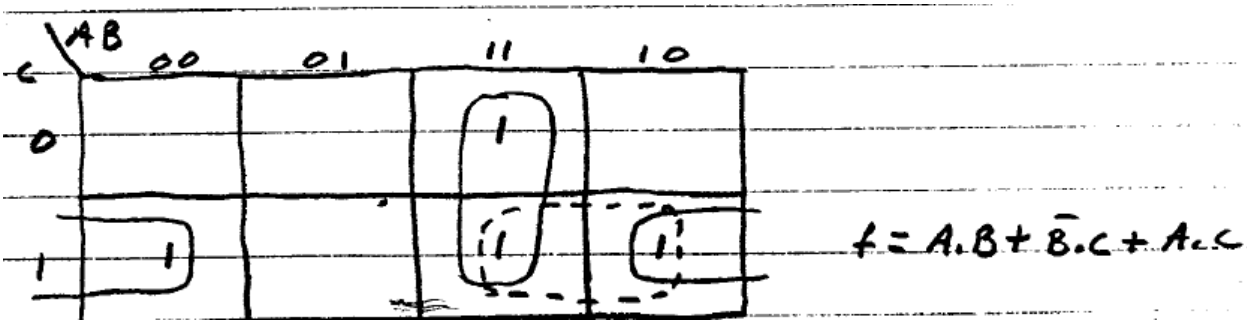
The occurrence of a hazard can be detected by inspecting the K-maps of the particular circuit. For example consider the K-map of the above AND-OR circuit:



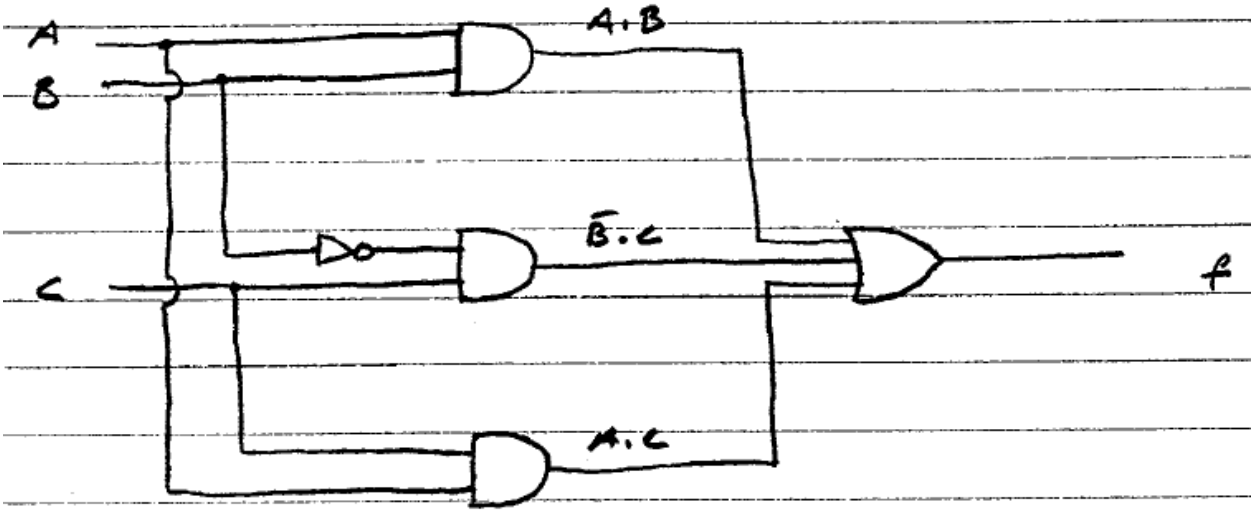
The change in B from 1 to 0 moves the circuit from minterm 111 to minterm 101. The hazard exists because the change of input results in a different product term implicant covering the two minterms. Minterm 111 is covered by the product term implemented in gate 1, and minterm 101 is covered by the product term implemented in gate 2. Whenever the circuit moves from one product term to

another, there is a possibility of a momentary interval when neither term is equal to 1, giving rise to an undesirable 0 output.

Hazards can be eliminated by enclosing the two minterms in a function with another product term that covers both groupings. This is shown in the K-map below:



The hazard-free circuit is shown below. The extra gate in the circuit generates the product term A●C. The removal of the hazard requires the addition of redundant gates to the circuit.



Notes

1. In two-level networks when a circuit is synthesized in sum of products with AND-OR gates or with NAND gates, the removal of static 1-hazard guarantees that no static 0-hazards or dynamic hazards will occur.

2. Methods for eliminating hazards always depend on the assumption that the unexpected changes in the output are in response to *single-bit* changes in the inputs.

Hazards in Multilevel Networks

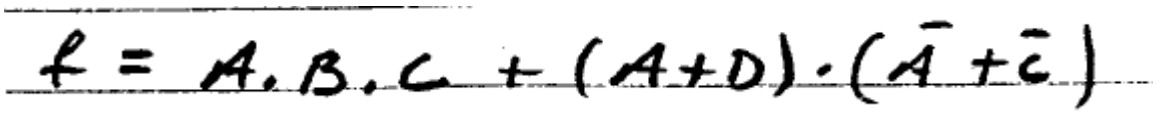
Begin by mapping the multilevel function into a two-level form called the *transient output function*. In forming this function the variable and its complement are treated as independent variables. This means that one cannot use the Boolean laws $A \bullet \overline{A}=0$ and $A+\overline{A}=1$, since the former introduces static 0-hazards, and the latter leads to static 1-hazards. In addition we cannot use any of the simplification theorems derived from these Boolean laws. Since the distributive laws can *never* introduce a hazard, it can be used freely to simplify a function.

A static hazard-free network is assured if the function is put in such a form that the transient output function guarantees that every set of adjacent 1's in the K-map are covered by a term, and that no terms contain both a variable and its complement. The first condition eliminates 1-hazards and the second eliminates 0-hazards.

Dynamic hazards occur because of the multiple paths in the multilevel network, each with different time delays. Since it is difficult to eliminate dynamic hazards in multilevel networks it is best to implement the network as a hazard-free two-level network.

Example

Consider the multilevel function:

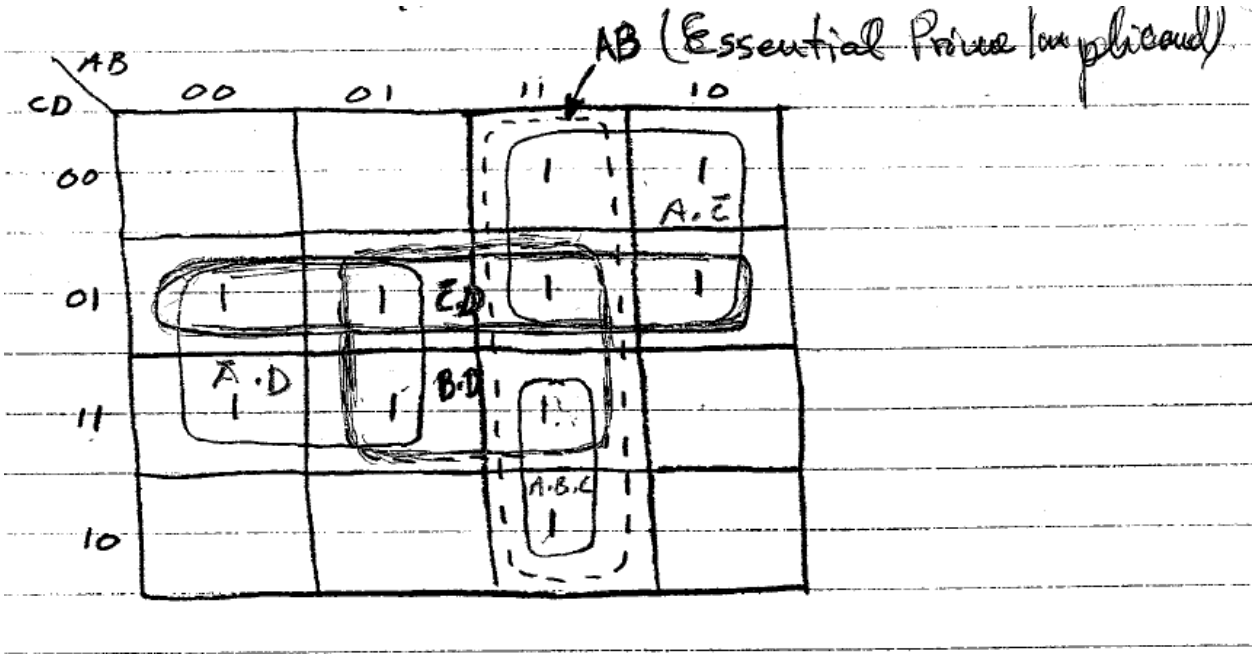


Design and implement a static hazard free network.

Now:

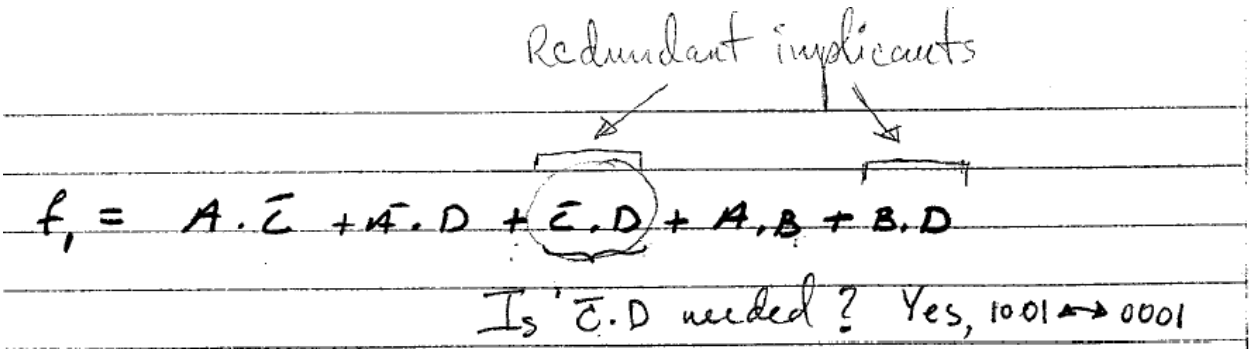
$$f = A \cdot B \cdot C + (A + D) \cdot (\bar{A} + \bar{C})$$
$$= A \cdot B \cdot C + A \cdot \bar{A} + A \cdot \bar{C} + \bar{A} \cdot D + \bar{C} \cdot D$$

This is the transient output function in sum of products form. Note that since A and its complement are treated as independent variables all the terms must be kept. Note the function is in two-level form. To check for static 1-hazards we draw the k-maps as shown below:



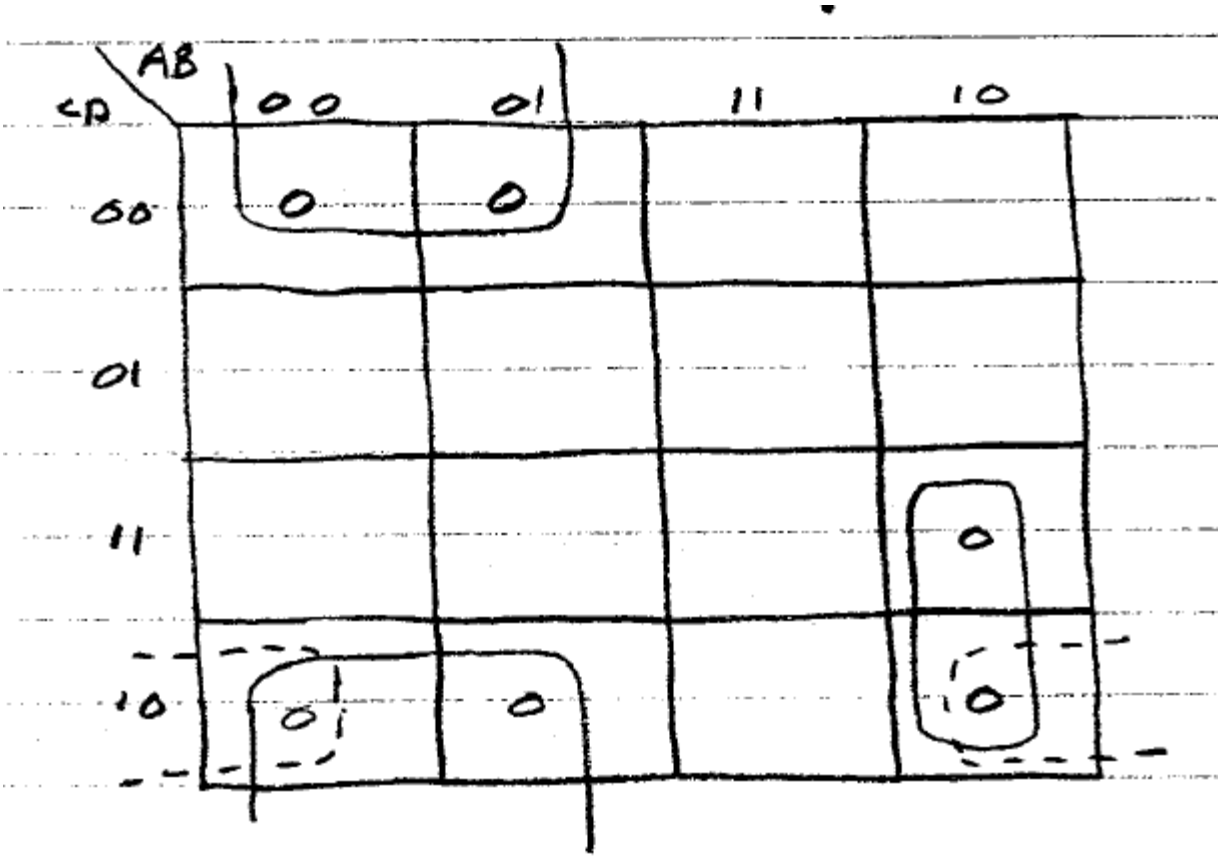
Note that the term $A \bullet \overline{A}$ can never cause a 1-hazard. With the groupings as shown the function contains static 1-hazards, such as the transition from ABCD = 1111 to 0111, or 1111 to 1101.

To eliminate these hazards add *redundant prime implicants* AB and BD as shown. The function then becomes:



Note that since AB completely covers the term ABC it does not appear in f1.

To verify that f1 is free of static 0-hazards, we proceed as follows: From the circled 0's in the K-map we see that:



The function has a 0-hazard on the transition from 1010 to 0010. The problem can be corrected by multiplying f2 by the implicant $(B+\bar{C}+D)$ as indicated by the K-map. The resulting function is now:

$$f_3 = (A+D)\cdot(\bar{A}+B+\bar{C})\cdot(B+\bar{C}+D)$$

$$= A\cdot\bar{A} + (A\cdot B + A\cdot\bar{C} + \bar{A}\cdot D + B\cdot D + \bar{C}\cdot D)\cdot(B+\bar{C}+D)$$

$$= A\cdot B + A\cdot B\cdot\bar{C} + A\cdot B\cdot D + A\cdot B\cdot\bar{C} + A\cdot\bar{C} + A\cdot\bar{C}\cdot D + \bar{A}\cdot B\cdot D + \bar{A}\cdot\bar{C}\cdot D + \bar{A}\cdot D + B\cdot D + B\cdot\bar{C}\cdot D + B\cdot D + B\cdot\bar{C}\cdot D + \bar{C}\cdot D + A\cdot\bar{A}\cdot B + A\cdot\bar{A}\cdot\bar{C} + A\cdot\bar{A}\cdot D$$

$$= A\cdot B\cdot(1+\bar{C}+D) + A\cdot\bar{C}\cdot(1+D) + \bar{A}\cdot D(1+B+\bar{C}) + B\cdot D + \bar{C}\cdot D(1+B)$$

($1+\bar{C}+D = 1$ by $+B$; $1+D = 1$ by $+\bar{A}$; $1+B+\bar{C} = 1$ by $+A$; $1+B = 1$)

$$= A\cdot\bar{C} + \bar{A}\cdot D + \bar{C}\cdot D + A\cdot B + B\cdot D$$

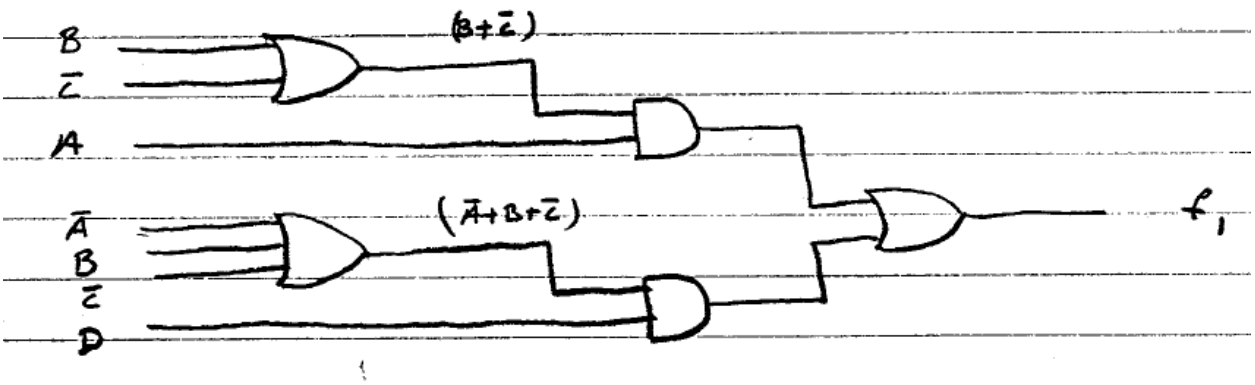
$$= f_1$$

Both expressions are simultaneously free of static 0- and 1-hazards.

To implement consider the expression for f1 and factor to obtain a multilevel static-hazard free expression:

$$f_1 = A\cdot\bar{C} + \bar{A}\cdot D + \bar{C}\cdot D + A\cdot B + B\cdot D$$

$$= A\cdot(B+\bar{C}) + (\bar{A}+B+\bar{C})\cdot D$$



This is a three-level circuit requiring five gates.

Programmable and Steering Logic

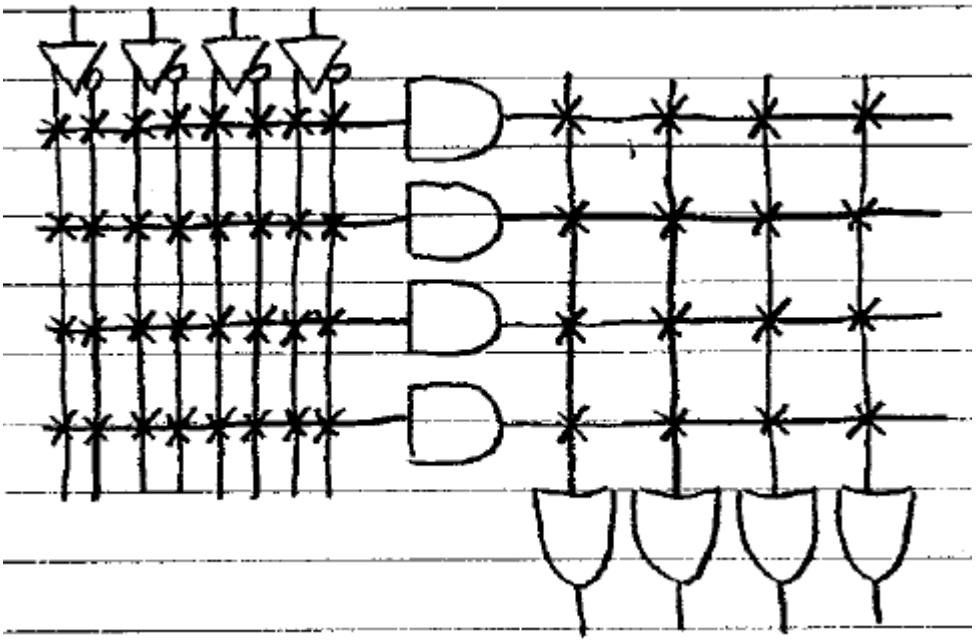
AND and OR gates (or NOR and NAND gates) can be arranged into a generalized array structure whose connections can be programmed to implement a specific function. Such general-purpose logic building blocks are called PAL's (programmable array logic) or PLA's (programmable logic arrays).

PAL's and PLA's

Array logic components are multi-input/multi-output devices, typically organized into an AND subarray and an OR subarray. The AND subarray maps the inputs into particular product terms, depending on the programmed connections. The OR subarray takes those terms and OR's them together to produce the final sum of products expression.

The details of the programming process depend on the particular integrated circuit. One technique places fuses between all possible inputs to a gate and the gate itself. By place a high current through selected fuses they are blown and the selected paths are disconnected.

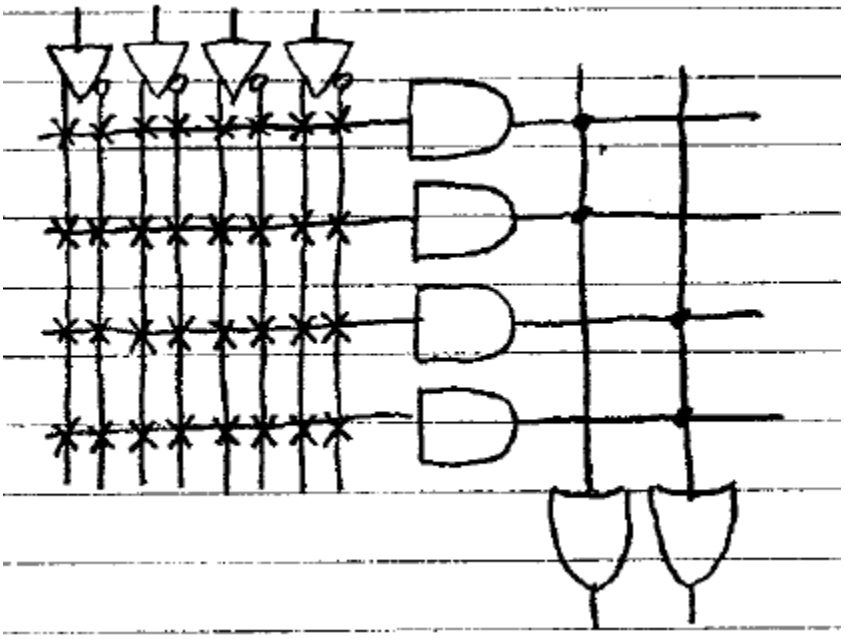
A commonly used notation for representing the technology of array logic is shown below. The single wires entering the AND and OR gates represent multiple inputs. The X's represent the fuse locations.



The Difference Between PLA's and PAL's

The above figure implies that both the AND and OR subarrays can be personalized in any way the designer wants. Devices with this generality are called Programmable Logic Arrays (PLA's). However, not all programmable logic is fully programmable. Some devices have a programmable AND array but the connections between product terms and specific OR gates are hardwired. The number of product term inputs to an OR gate is internally limited to 2,4,8, or 16. Such devices are called programmable array logic (PAL). The figure below shows a 4 input / 4 product-term / 2 output PAL organized with a particular fixed OR array. The OR gates for this case are limited to the product terms each.

The main difference between PLA's and PAL's is that the former can take advantage of shared product terms and the latter cannot.



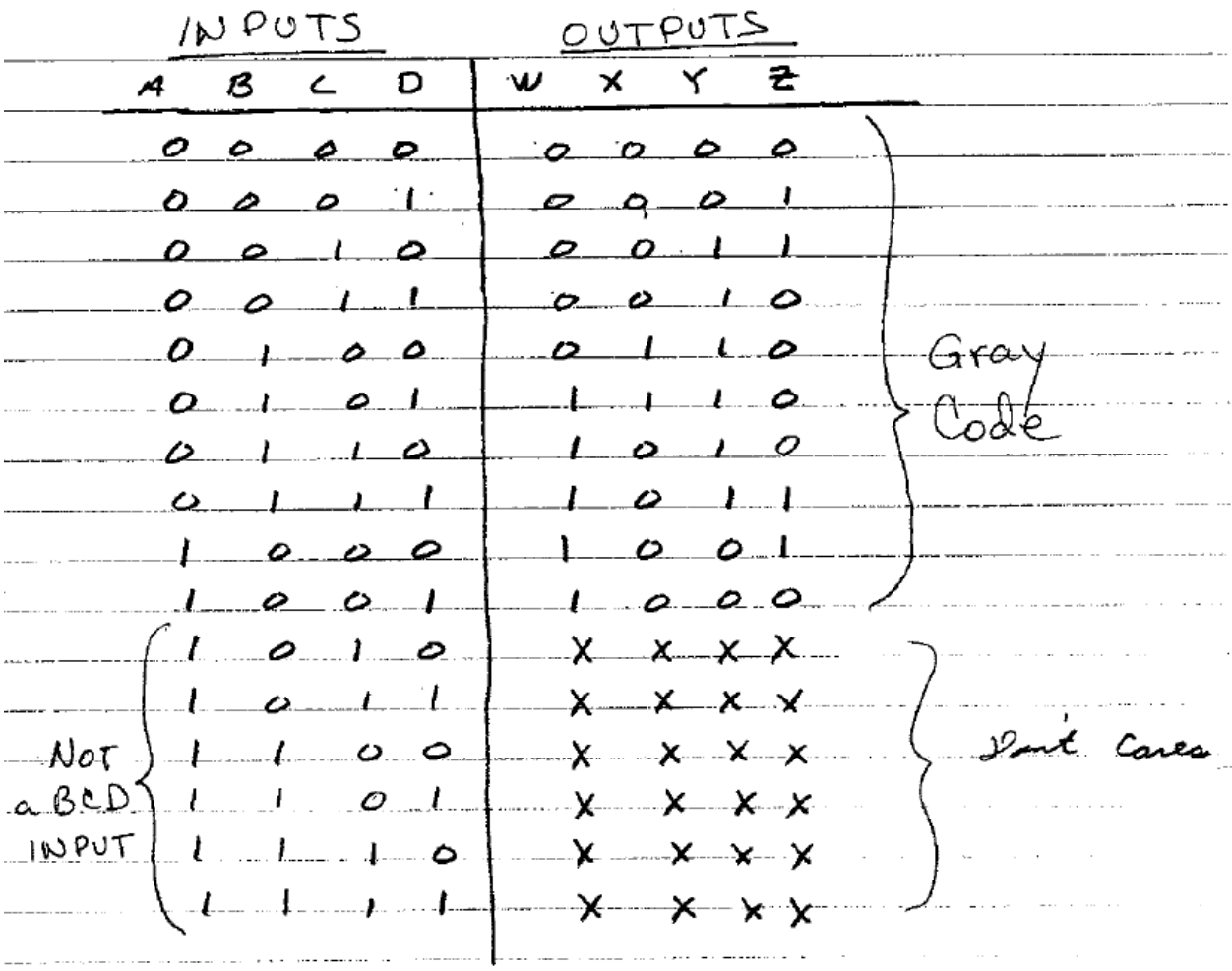
For devices with an equivalent internal capability, a PLA is able to implement a more complex collection of functions than a PAL if many product terms are shared. A PLA will, however, be slower because of the relatively higher resistance of fuse-based connections than standard wire connections.

Example: BCD-to-Gray-Code Converter

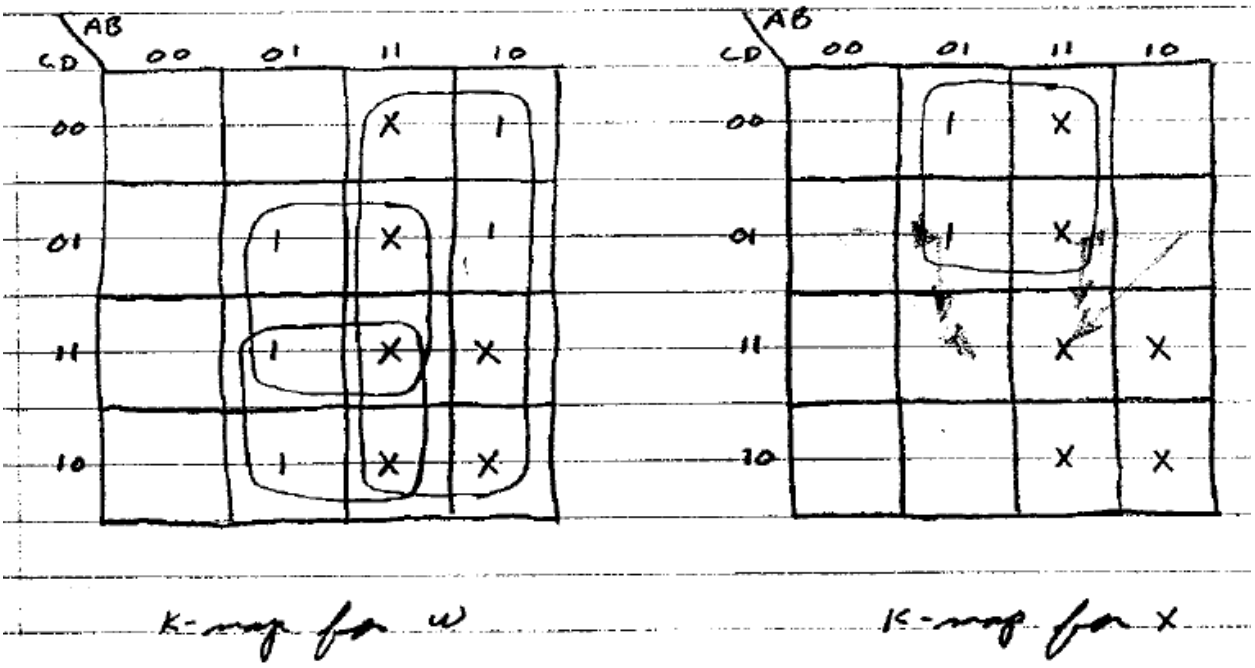
Design a code converter that maps a 4-bit Binary Coded Decimal (BCD) number into a 4-bit Gray code number.

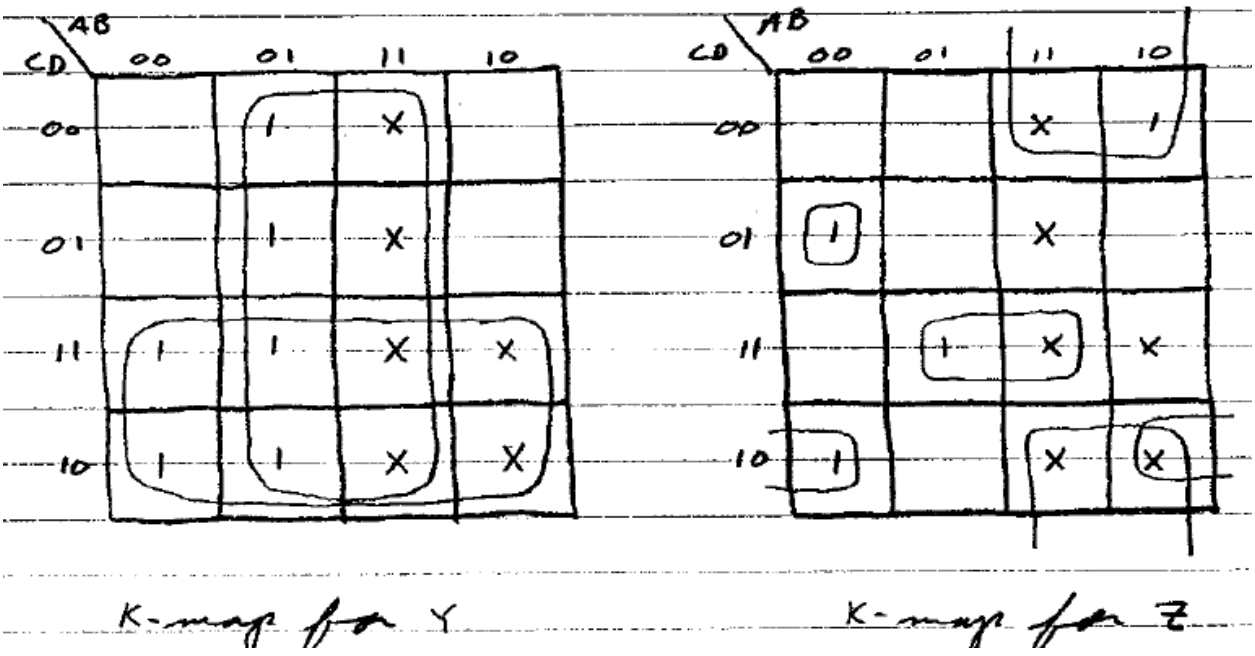
Each number in a Gray code sequence differs from its predecessor by 1 bit. The circuit has four inputs A,B,C,D which represent the BCD number, and four outputs W,X,Y,Z which represent the 4-bit Gray code word.

The truth table is shown below:



The K-maps are shown below with the prime implicants circled:





The reduced equations are:

$$W = A + B \cdot D + B \cdot C$$

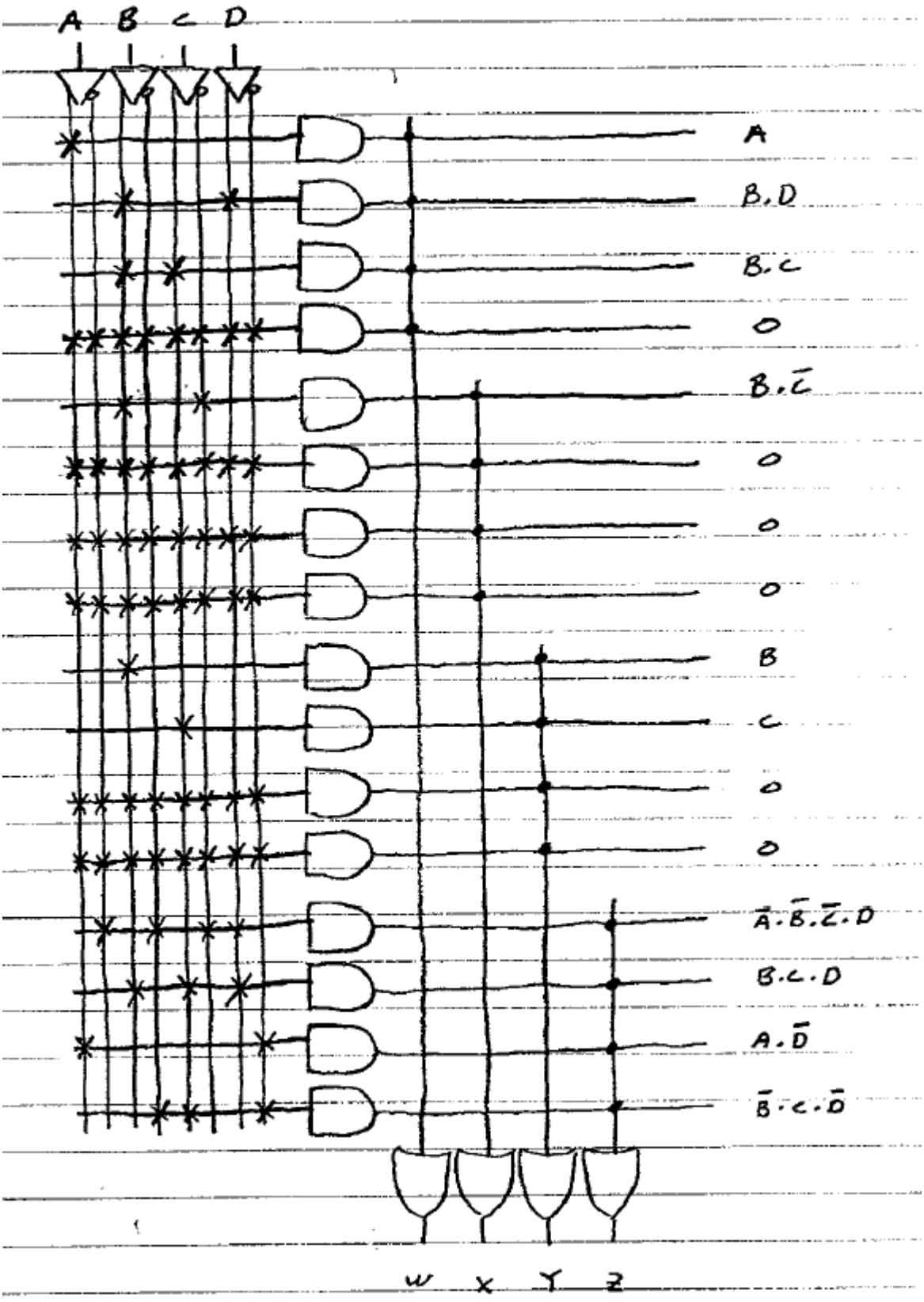
$$X = B \cdot \bar{C}$$

$$Y = B + C$$

$$Z = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot D + \bar{B} \cdot C \cdot \bar{D} + B \cdot C \cdot D + A \cdot \bar{D}$$

Since there are no shared product terms, a PAL will be used to implement the functions. Note that hazards are of no concern here since the only possible adjacency is in the K-maps for Z and this occurs in a don't care situation.

The PAL as shown below contains four 4-input OR gates. Many AND gates are being waster. A PLA could be used to implement the function but would be slower. The programmable logic approach implements two functions in a single integrated circuit package.


A B C D
A
B.D
B.C
0
B.C̄
0
0
0
B
C
0
0
Ā.B̄.C̄.D
B.C.D
A.D̄
B̄.C.D̄
W X Y Z

Design Procedure

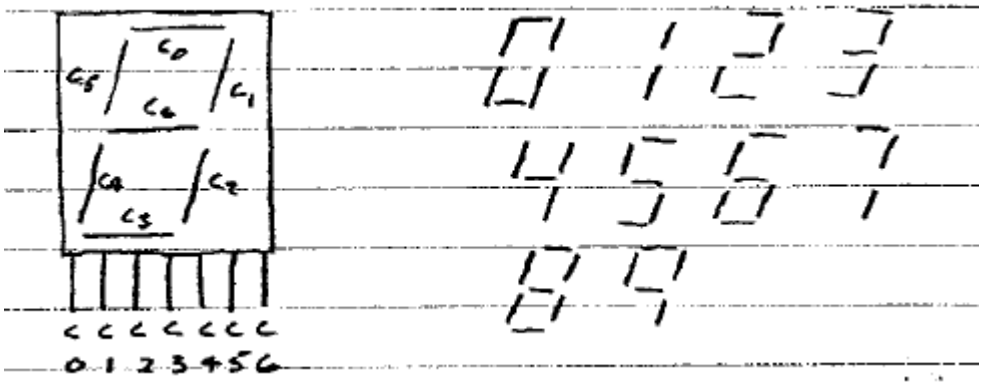
A design procedure consist of the following steps:

1. Understand the problem.
2. Formulate the problem in terms of a truth table or other suitable design representation.
3. Follow implementation procedure. Synthesize minimized expressions for a two-level sum of products combinational network.
4. Choose implementation technology (PLA/PAL).

Example: BCD-to-7-Segment Display Converter

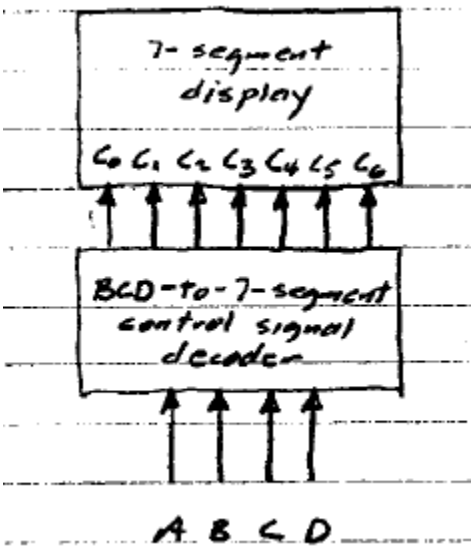
Design a combinational circuit that maps a 4-bit BCD digit to the segments that control a seven-segment display.

The display element contains seven light-emitting diodes (LEDs). When the appropriate LED control line is asserted, the associated LED segment lights. We will assume that the LED driver inputs are active high (most of the actual LED driver components are really active low). Otherwise, the LED segment is off. The seven segments are controlled independently; there is no limit to the number of segments that could be illuminated at the same time. The figure below shows the seven-segment display and its configuration displaying each of the 10 possible BCD digits:



Step 2: Understand the Problem

What is the circuit supposed to do? What are the inputs and outputs? There are four input signals, representing the 4-bit BCD digits. There should be seven outputs, one for each of the LED segments that must be controlled. A block diagram is shown below:



Step 2: Formulate in Terms of a Truth Table

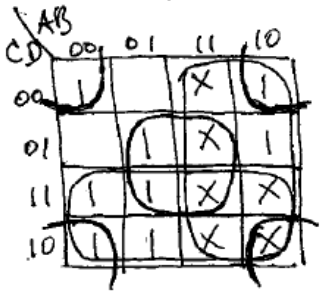
It is best to tabulate the input values with the desired outputs. For example, the BCD representation for the digit 0 should cause the LED segment 0,1,2,3,4, and 5 to illuminate. Hence, for the output 0000 the control signals C0 to C5 would be asserted, with C6 unasserted. For the input 0001, segments 1 & 2 are turned on, while segments 0 and 3-6 are left off. In the table entry for 0001, C1 and C2 are asserted, while the remainder are unasserted. The whole truth table is shown below, where the only valid entries are for decimal 0-9, corresponding to binary 0000-1001.

A	B	C	D	C_0	C_1	C_2	C_3	C_4	C_5	C_6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

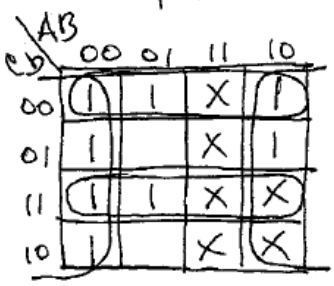
Step 3: Implementation Procedure

Since we desire a two-level network we will use K-map techniques. Note that seven 4-variable maps are required. The K-maps with circled prime implicants are shown below:

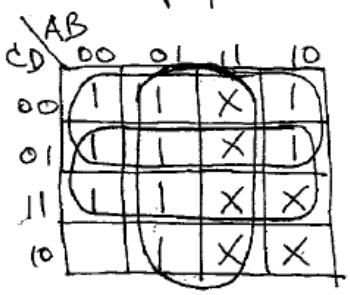
K-map for C_0



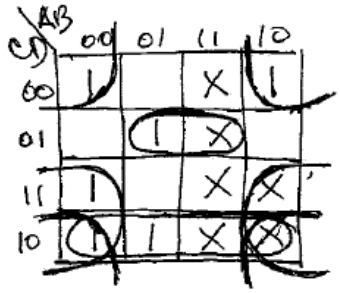
K-map for C_1



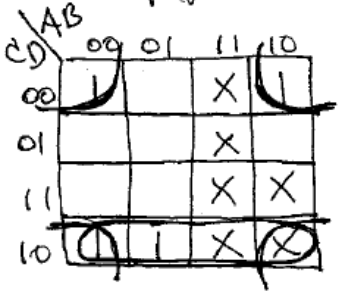
K-map for C_2



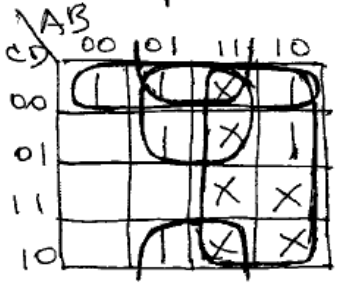
K-map for C_3



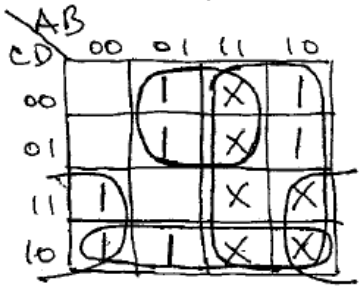
K-map for C_4



K-map for C_5



K-map for C_6



From the K-maps we can write the following equations for the LED segment control outputs:

$$C_0 = A + B.D + C + \bar{B}.\bar{D}$$

$$C_1 = \cancel{A} + \bar{C}.\bar{D} + C.D + \bar{B}$$

$$C_2 = \cancel{A} + B + \bar{C} + D$$

$$C_3 = \bar{B}.\bar{D} + C.\bar{D} + B.\bar{C}.D + \bar{B}.C$$

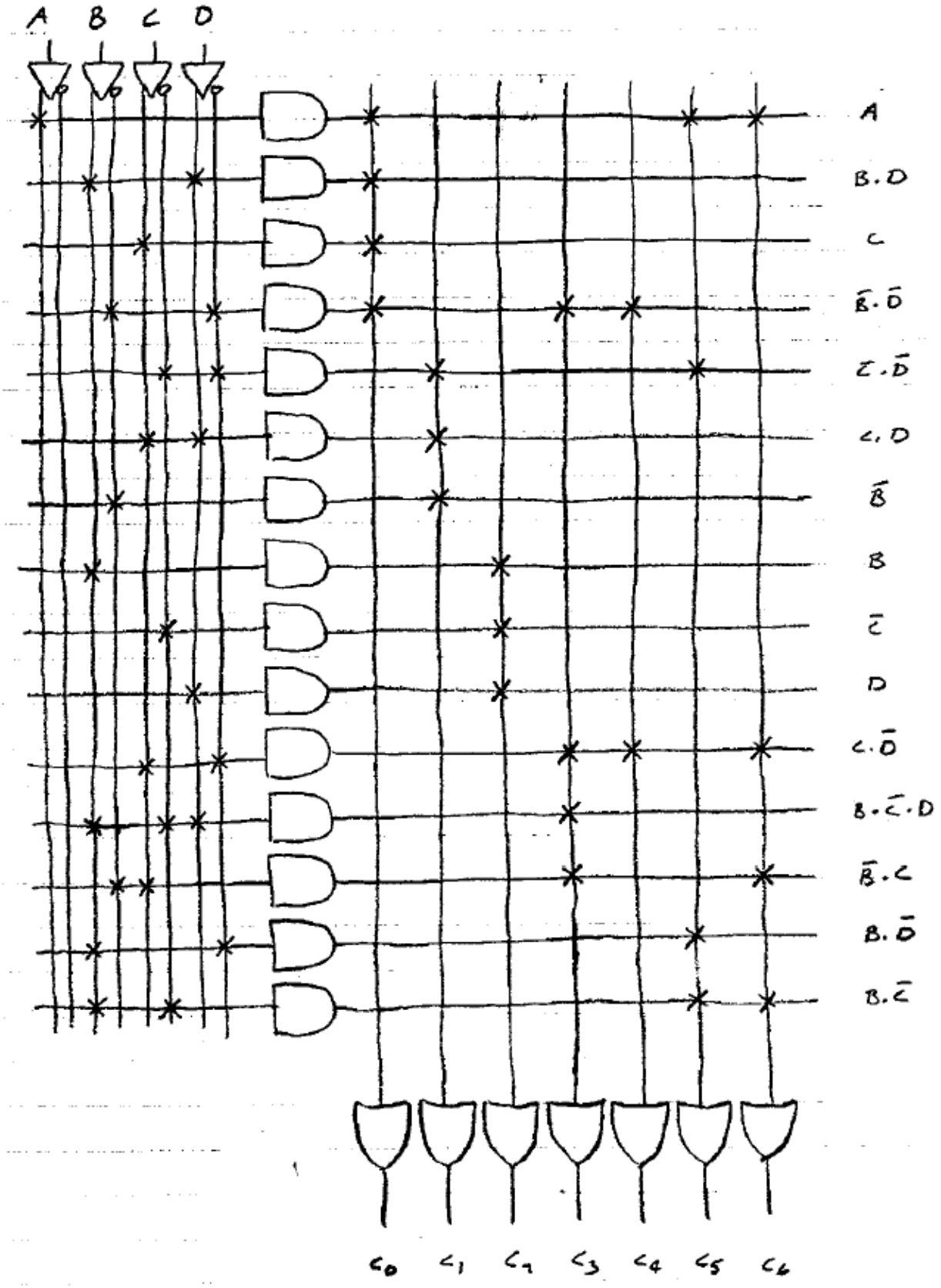
$$C_4 = \bar{B}.\bar{D} + C.\bar{D}$$

$$C_5 = A + \bar{C}.\bar{D} + B.\bar{D} + B.\bar{C}$$

$$C_6 = A + C.\bar{D} + B.\bar{C} + \bar{B}.C$$

Step 4: Implementation

The PLA implementation is shown below. The limiting factor in a PLA is the number of unique product terms to implement the outputs. There are fifteen required product terms in the above set of equations. A typical PLA component can handle sixteen inputs, eight outputs, and forty-eight product terms. From the K-maps one can see that the hazards are not of concern in this problem. CAD methods can be used to find a multi-level solution.

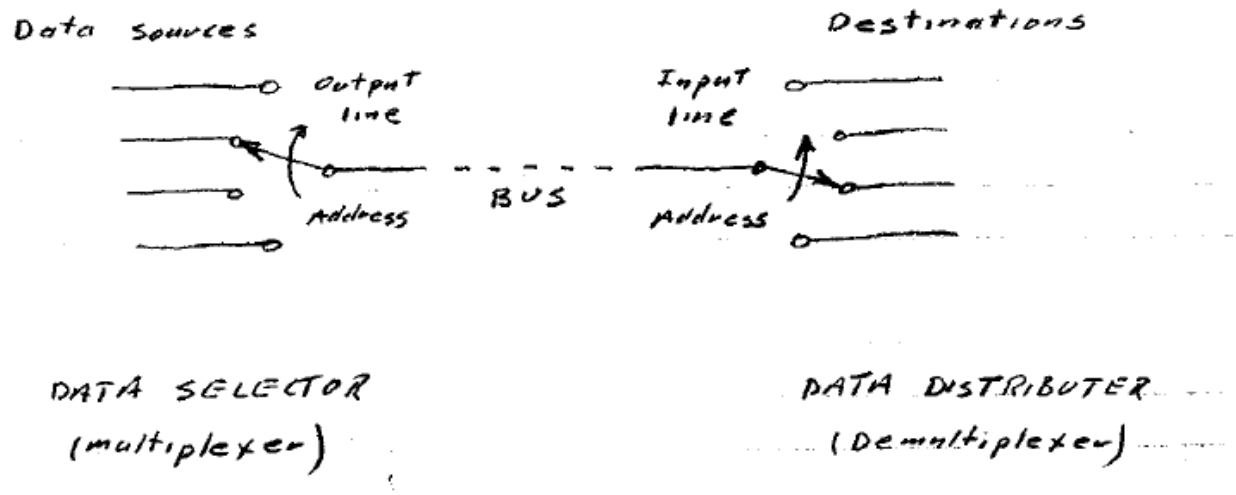

A
B
C
D
A
B.D
C
$\bar{B}.\bar{D}$
$\bar{C}.\bar{D}$
C.D
$\bar{B}$
B
$\bar{C}$
D
$C.\bar{D}$
$B.\bar{C}.D$
$\bar{B}.C$
$B.\bar{D}$
$B.\bar{C}$
C_0
C_1
C_2
C_3
C_4
C_5
C_6

Beyond Simple Logic Gates

Switching Logic

Switching networks provide an alternative to discrete gates for constructing digital systems. They operate by steering or directing inputs to outputs through a network of switching paths rather than by computing a Boolean function.

A typical digital system has several sources of information and several destinations. In practice the desired source is connected to a common path or *bus* and the bus is then connected to the desired destination. This is called *multiplexing* and is shown below:

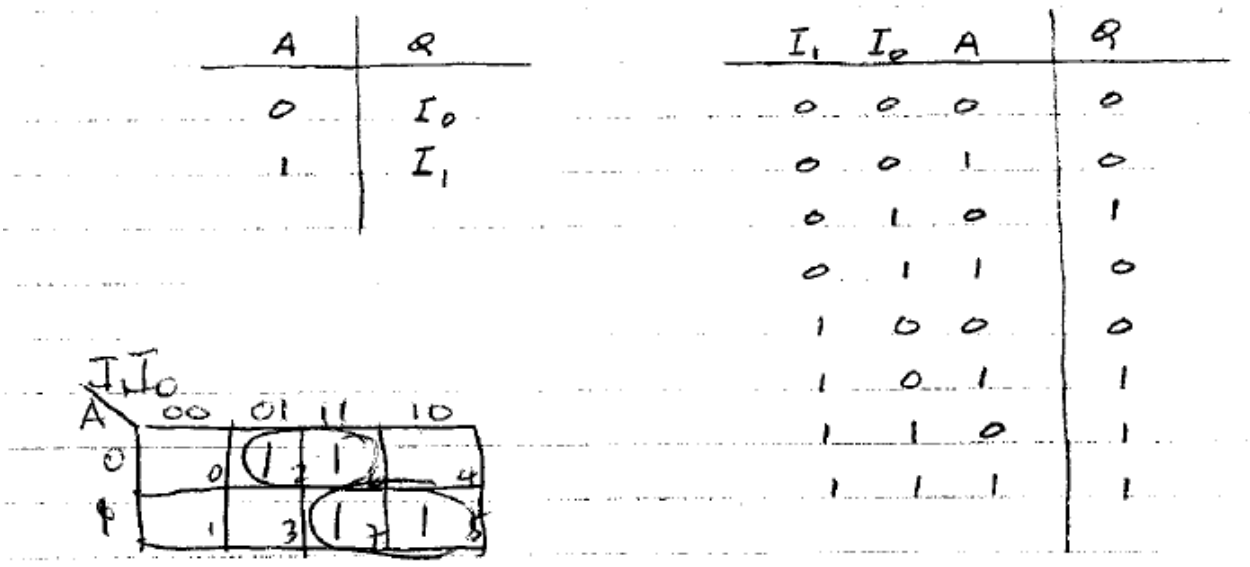


A multiplexor or data selector selects the desired source and places its information on the bus; a demultiplexer or data distributor transfers information on the bus to the selected destination.

Multiplexer/Data Selector

A multiplexer, or MUX, is a combinational logic network with 2^n data inputs, n control inputs, and one data output. Depending on the settings of the control inputs, a single data input is selected and steered to the outputs. Since a multiplexer selects an input for connection to the output is often referred to as a data selector.

The figure below gives a functional truth table in the left and a conventional truth table on the right for a multiplexer with two data inputs, I0 and I1, and one control input A:



A	Q
0	I_0
1	I_1

I_1	I_0	A	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The functional truth table indicates that we are passing a selected output to the output. Using a Boolean equation, the two-input multiplexer can be described as:

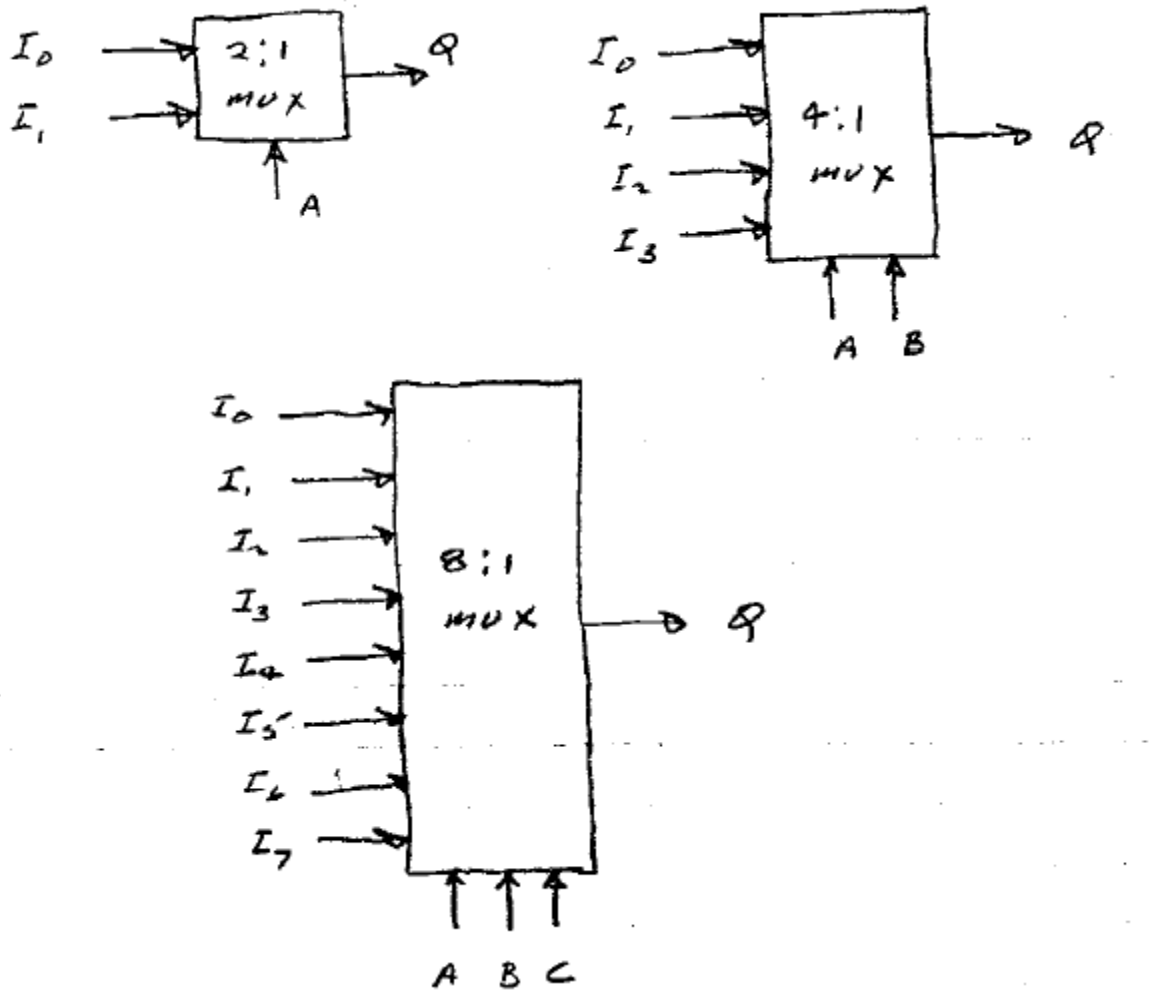
$$Q = \bar{A} \cdot I_0 + A \cdot I_1$$

OR

$$Q = \overline{(\overline{\bar{A} \cdot I_0})(\overline{A \cdot I_1})}$$

If A=0, the output is given by I_0. If A=1, the output is given by I1.

Multiplexors are described by the number of data inputs, since the number of control inputs can be inferred from this. Hence, a 2:1 multiplexor has two data inputs, one data output, and one control input. A 4:1 multiplexor has four data inputs, one data output, and two control inputs. The figure below shows the block diagrams for 2:1, 4:1, and 8:1 multiplexer:



The Boolean equation for the 4:1 and 8:1 multiplexers can be generalized from the 2:1 multiplexer:

$$Q = \bar{A}\cdot\bar{B}\cdot I_0 + \bar{A}\cdot B\cdot I_1 + A\cdot\bar{B}\cdot I_2 + A\cdot B\cdot I_3$$

$$Q = \bar{A}\cdot\bar{B}\cdot\bar{C}\cdot I_0 + \bar{A}\cdot\bar{B}\cdot C\cdot I_1 + \bar{A}\cdot B\cdot\bar{C}\cdot I_2 + \bar{A}\cdot B\cdot C\cdot I_3 + A\cdot\bar{B}\cdot\bar{C}\cdot I_4$$
$$+ A\cdot\bar{B}\cdot C\cdot I_5 + A\cdot B\cdot\bar{C}\cdot I_6 + A\cdot B\cdot C\cdot I_7$$

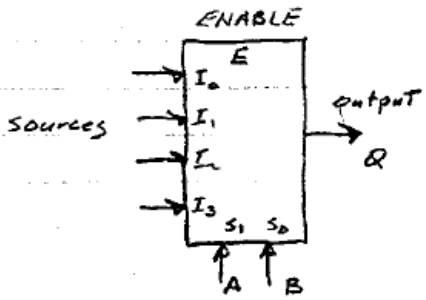
Or in general:

$$Q = \sum_{k=0}^{2^n-1} m_k \cdot I_k$$

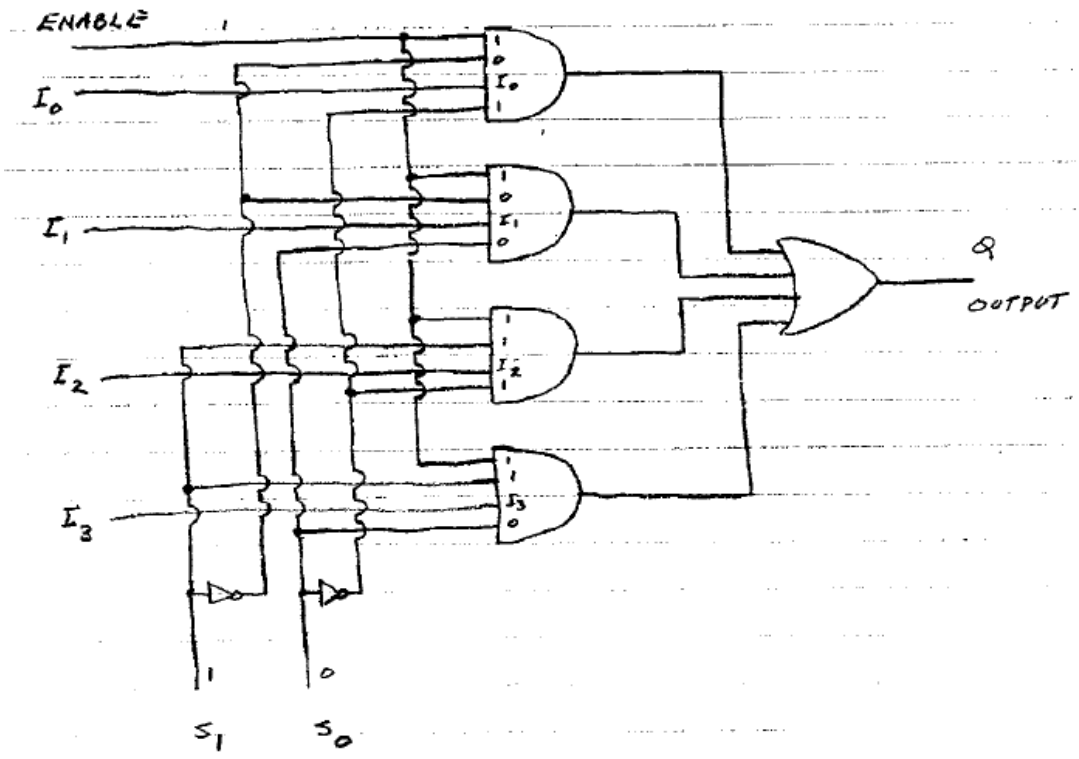
Example:

Design a 4:1 multiplexer. Show its block diagram, functional truth table, and logic diagrams. Show how the multiplexer works by considering the case where S_1=1, S_0=0. Assume the device is enabled, i.e.: E=1.

The block diagram, functional truth table, and logic diagram are shown below:



Selection Lines		Output
S_1	S_0	Q
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



Each of the four inputs I_0 through I_3 are selected by S_0 and S_1, and directed to the output when the device is enabled. The equation describing the above device is:

$$Q = \bar{S}_1 \cdot \bar{S}_0 \cdot I_0 + \bar{S}_1 \cdot S_0 \cdot I_1 + S_1 \cdot \bar{S}_0 \cdot I_2 + S_1 \cdot S_0 \cdot I_3$$

To see how the above device works consider the case where E=1, S_0=0, S_1=1. Tracing the input signals I_0 through I_3, we get Q=I_2 so only the input whose address equals 2 is seen at the output.

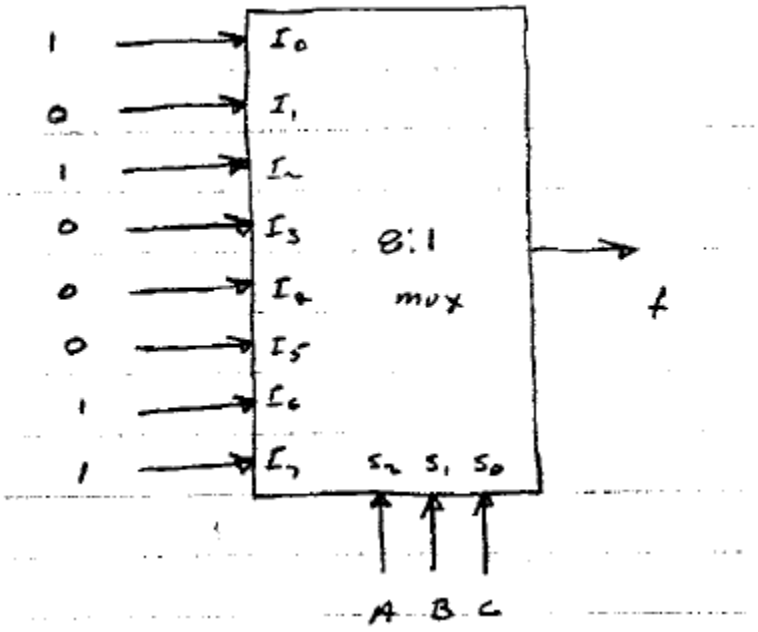
Multiplexer as a Logic Building Block

A multiplexer can implement a general-purpose logic building block. A truth table can be implemented directly into hardware by using a multiplexor. Consider the function:

$$f(A,B,C) = m_0 + m_2 + m_6 + m_7$$

$$= \bar{A}\cdot\bar{B}\cdot\bar{C} + \bar{A}\cdot B\cdot\bar{C} + A\cdot B\cdot\bar{C} + A\cdot B\cdot C$$

The function can be implemented by an 8:1 multiplexer as shown below. The input variables A,B, and C are connected to the multiplexer selection inputs. The input I_i is set to 1 if the function includes minterm m_i. All other inputs are set to 0. In this case I_0, I_2, I_6, and I_7 are all set to 1, while I_1, I_3, I_4, I_5 are set to 0.

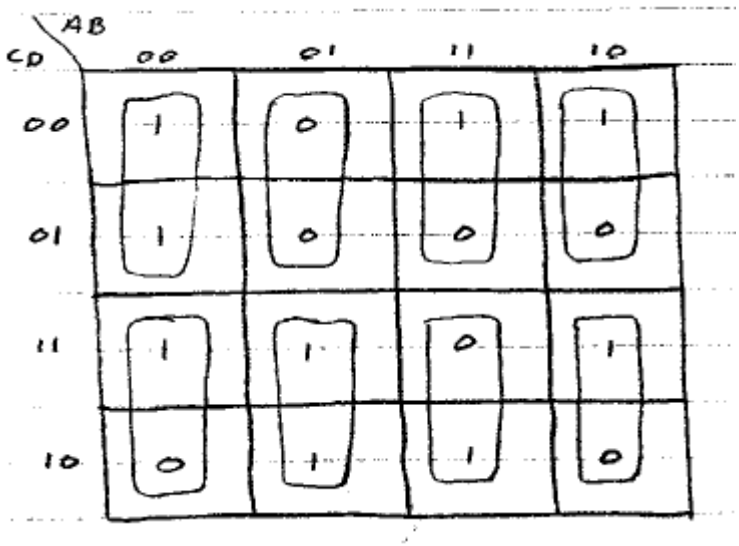


To illustrate, consider the case where A=B=C=0. This corresponds to minterm m_0. With these inputs the multiplexor will select I_0 and set f=1. If A=B=0 and C=1, then I_1 is selected and f is set to 0, and so on.

In general, we see that by selecting n-1 variables as control inputs to a 2^{n-1} input multiplexor, we can implement any Boolean function of a variable.

Example

Use a multiplexer to implement the function f(A,B,C,D) whose k-map is given below.

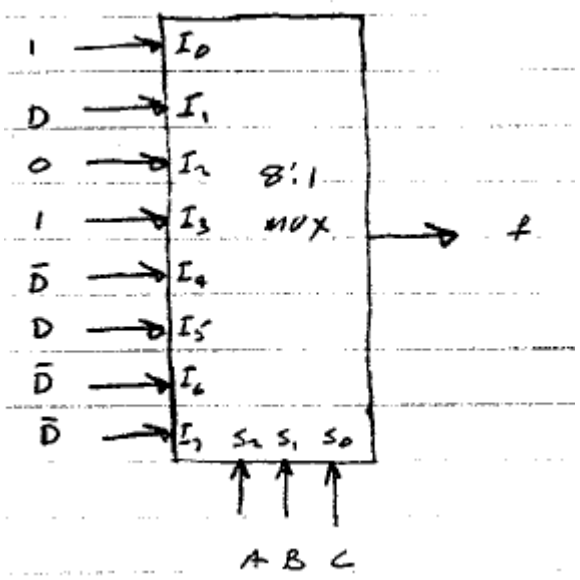


Since f is a function of four variables it can be implemented by an eight-input multiplexer. Select A,B, and C as the control inputs. The k-map is then partitioned into eight pars of k-map entries, each sharing common values for the three control inputs. Each pair can be replaced by either 0,1,D, or NOT D. f can be represented by the equation:

$$f = \bar{A}\cdot\bar{B}\cdot\bar{C}\cdot(1) + \bar{A}\cdot\bar{B}\cdot C\cdot(D) + \bar{A}\cdot B\cdot\bar{C}\cdot(0) + \bar{A}\cdot B\cdot C\cdot(1)$$

$$+ A\cdot\bar{B}\cdot\bar{C}\cdot(\bar{D}) + A\cdot\bar{B}\cdot C\cdot(D) + A\cdot B\cdot\bar{C}\cdot(\bar{D}) + A\cdot B\cdot C\cdot(\bar{D})$$

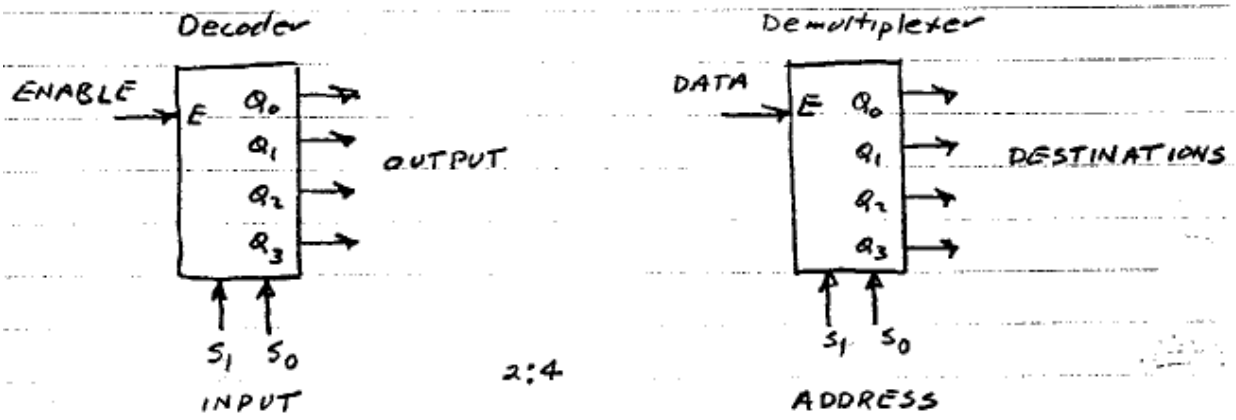
The multiplexer is shown below:



Two-, 4-, 8- and 16-to-1 multiplexers are commercially available as MSI packages.

Decoders/Demultiplexer/Data Distribution

Decoders convert binary information from one coded form to another. As shown below, the same unit can serve as a decoder or as a demultiplexer (data distributor), depending on how the terminals are interpreted.



When enabled by E going high, the decoder places a 1 on the OUTPUT line corresponding to the INPUT code; all other output lines remain LOW. When used

as a demultiplexer DATA from the bus is applied to the E terminal and appears on the DESTINATION line selected by the ADDRESS.

A decoder/demultiplexer takes as input a single data input (an enable signal) and n control signals, and uses the latter to assert one of 2^n output lines. For example, a 1:2 decoder/demultiplexer has two inputs, E(enable) and S(select), and two outputs, Q0 and Q1. The Boolean equations for the outputs are as follows:

$$Q_0 = E \cdot \bar{S}_0$$

$$Q_1 = E \cdot S_0$$

If E=0 both outputs are at 0. When E=1 the value of S0 determines which of the two outputs will be driven high. The equations for the 2:3 demultiplexer are:

$$Q_0 = E \cdot \bar{S}_1 \cdot \bar{S}_0 \qquad Q_2 = E \cdot S_1 \cdot \bar{S}_0$$

$$Q_1 = E \cdot \bar{S}_1 \cdot S_0 \qquad Q_3 = E \cdot S_1 \cdot S_0$$

And for the 3:8 demultiplexer:

$$Q_0 = E \cdot \bar{S}_2 \cdot \bar{S}_1 \cdot \bar{S}_0 \qquad Q_4 = E \cdot S_2 \cdot \bar{S}_1 \cdot \bar{S}_0$$

$$Q_1 = E \cdot \bar{S}_2 \cdot \bar{S}_1 \cdot S_0 \qquad Q_5 = E \cdot S_2 \cdot \bar{S}_1 \cdot S_0$$

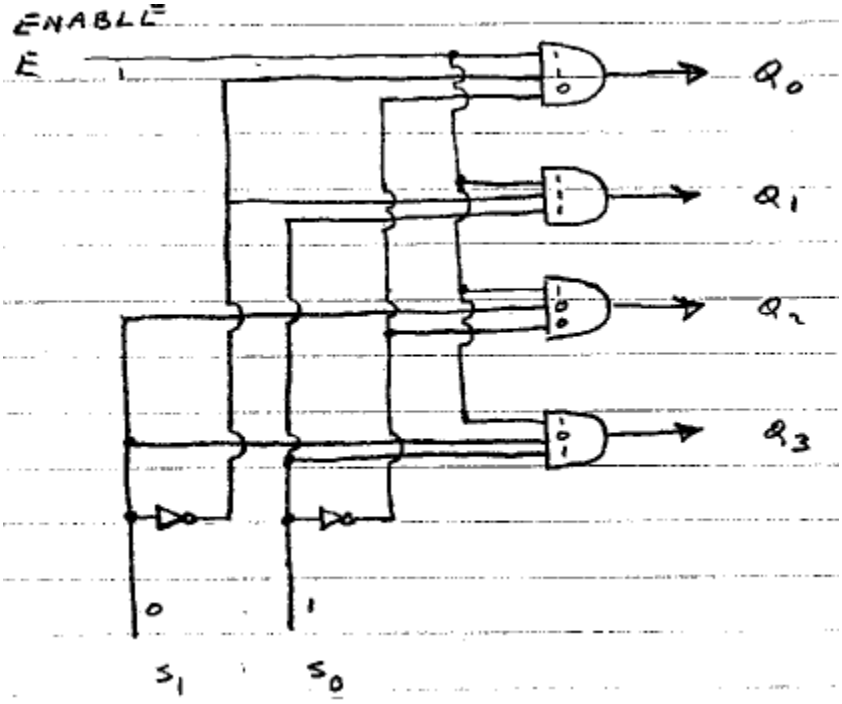
$$Q_2 = E \cdot \bar{S}_2 \cdot S_1 \cdot \bar{S}_0 \qquad Q_6 = E \cdot S_2 \cdot S_1 \cdot \bar{S}_0$$

$$Q_3 = E \cdot \bar{S}_2 \cdot S_1 \cdot S_0 \qquad Q_7 = E \cdot S_2 \cdot S_1 \cdot S_0$$

A decoder/demultiplexer is typically named by the number of control signals and the number of output signals (e.g. 1:2, 2:4, 3:8). Compare with the multiplexer naming: the number of data inputs and the number of data outputs (e.g. 2:1, 4:1, 8:1).

The truth table and logic diagram for a 2:4 demultiplexer are shown below:

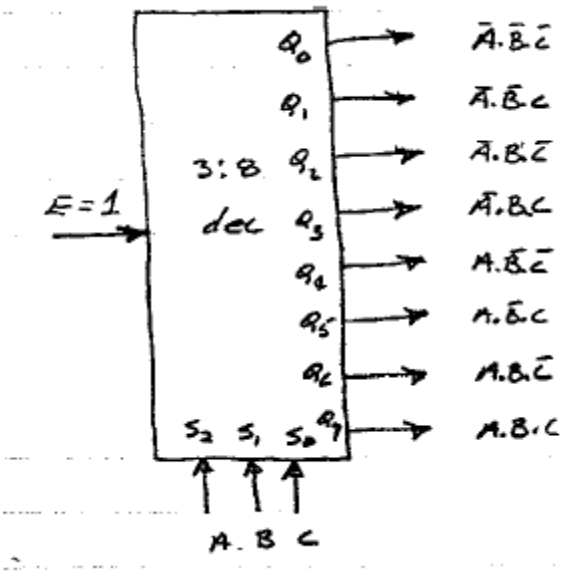
S_1	S_0	E	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	1	1	0	0	0



To see how the input works consider the case where E=1, S1=0, S0=1. Tracing the signals through we see that Q0 Q1 Q2 Q3 = 0100 so that only Q1 receives the data.

Decoder/Demultiplexer as a Logic Building Block

A decoder can also be used as a “minterm generator”. The figure below shows a 3:8 decoder where the select lines have signals A,B,C. Each output is labeled with the select line combination that causes that output to be asserted.



As an example, suppose the control signals A,B, and C are set to 0,1 and 0, respectively. This corresponds to minterm $\bar{A}.B.\bar{C}$ and output Q2 is enabled.

The decoder can also be used as a general-purpose combinational logic building block. Any function expressed in sum of products form over n variables can be implemented by an n:2^n decoder in conjunction with OR gates.

To illustrate consider the following three functions of the Boolean variables A,B,C,D:

$$f_1 = \bar{A}.B.\bar{C}.D + \bar{A}.\bar{B}.C.D + A.B.C.D$$

$$f_2 = A.B.\bar{C}.\bar{D} + A.B.C$$

$$f_3 = \bar{A} + \bar{B} + \bar{C} + \bar{D}$$

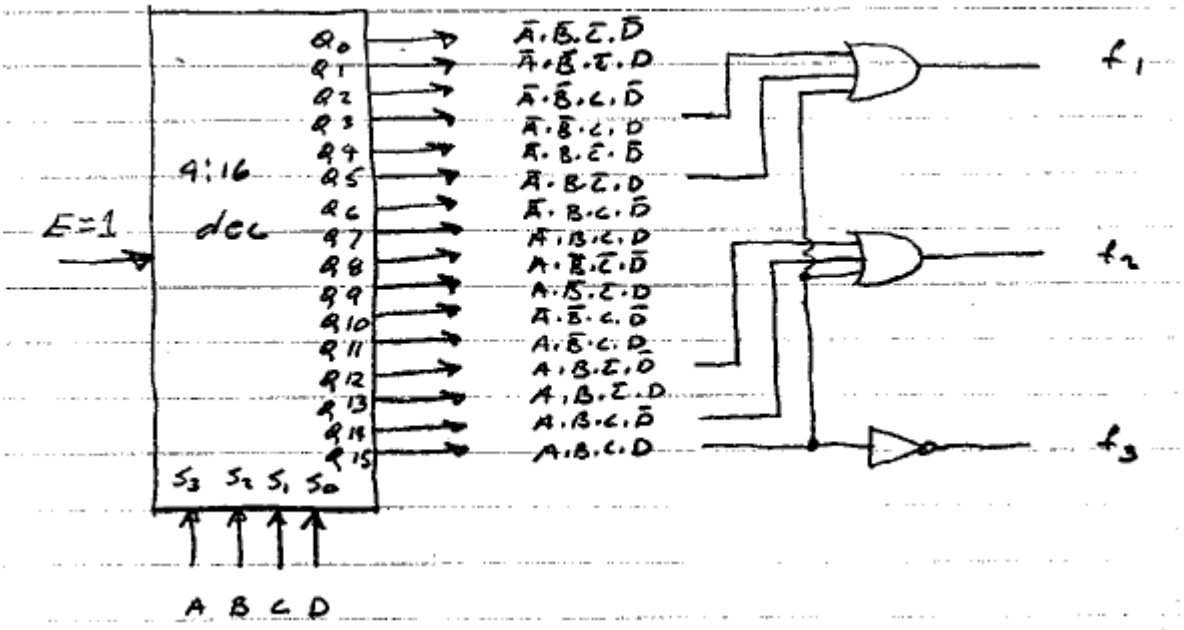
It is more convenient to express as the sum of 4-input minterms:

$$f_1 = \bar{A} \cdot B \cdot \bar{C} \cdot D + \bar{A} \cdot \bar{B} \cdot C \cdot D + A \cdot B \cdot C \cdot D$$

$$f_2 = A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot C \cdot (\bar{D} + D) = A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot C \cdot \bar{D} + A \cdot B \cdot C \cdot D$$

$$f_3 = \overline{A \cdot B \cdot C \cdot D} \quad \text{using 19}$$

The figure below uses a 4:16 decoder to implement these functions:



f1 is asserted whenever any of its three minterms are asserted. By connection A,B,C, and D to the decoder select lines, the output Q5, Q3, or Q15 will be asserted if the inputs corresponds to the desired minterm. f1 is then implemented by an OR gate connected to these decoder outputs.

In a similar manner, f2 is implemented by a three-input OR gate connected to decoder outputs Q12, Q14, and Q15. f3is obtained by an inverter driven by the Q15 decoder output.

This approach to implementing logic is useful for functions of a relatively small number of variables, as decoders with more than four select inputs are not as readily available, and a small number of minterms per function.

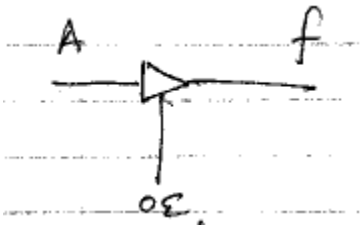
Tri-State Gates

Besides 0 and 1, there is a third signal value in digital circuits: the *high-impedance state*, denoted by Z. When a gates output is in a high-impedance state it is as though the gate were disconnected from the output. Gates that can be placed in such a state are called *tri-state* gates with outputs 0,1, and Z. In addition to its normal inputs, a tri-state has another input called *output enable*. When this input is 0, the output is Z. When the output enable is 1, the gates output is determined by its data inputs.

The truth table of a tri-state buffer gate is shown below. When output enable (OE) equals 0, the output is Z, no matter what the input A is. When OE=1 the buffer passes its input to the output.

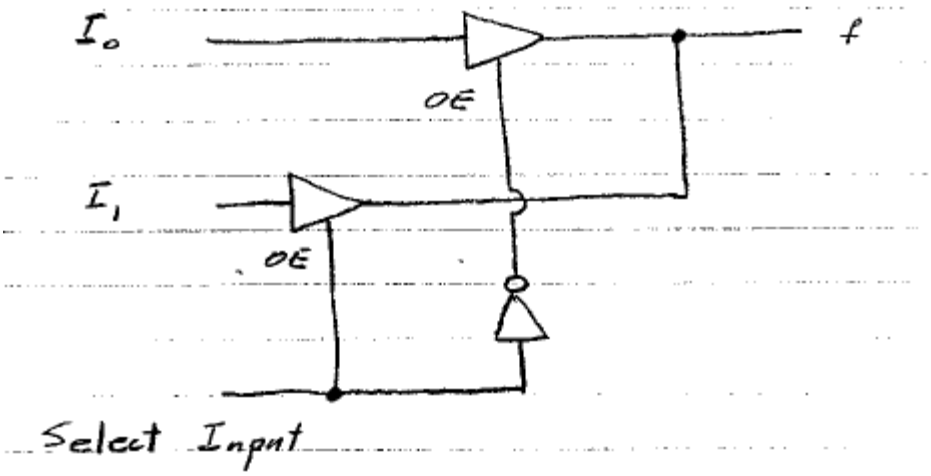
A	OE	f
X	0	Z
0	1	0
1	1	1

The symbol for the buffer gate is shown below:



Tri-state buffer gates are useful for situations such as a bidirectional data bus, where two drivers are both connected to the same wire. One side must always have their driver in the high-impedance state so the other side can drive the wire.

To see how tri-state gates work, consider the circuit below which consists of two tri-state buffers (with active-high enables) and an inverter:



If the *Select Input* is 0 then I_0 steers to f (the output of the I_1 buffer is open), and if it is 1 then I_1 steers to f (the output of the I_0 buffer is open).

Sequential Logic Design

The basic logic gates are connected to form *combinational* circuits that make decisions in response to the present inputs. In addition to these decision components we need *memory* components to store instructions and results. The outputs of these *sequential* circuits are affected by past inputs as well as present inputs. A memory unit must have the following characteristics:

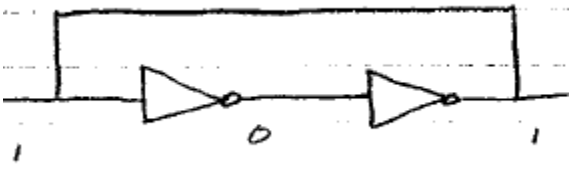
1. A binary storage device must have two (2) distinct states.
2. It must remain in one state until instructed to change.
3. It must change rapidly from one state to another, and the state value (0 or 1) must be clearly evident.

A simple memory component can be implemented from cascaded inverters. This is the basic circuit structure using in static RAM (Random Access Memory) designs. Alternatively, simple memory structures can be build using cross-coupled NOR or NAND gates. These elements for the basic building blocks of the *latch* and the *flip-flop* (bistable multivibrator) memory elements which are used in many types of data processing systems.

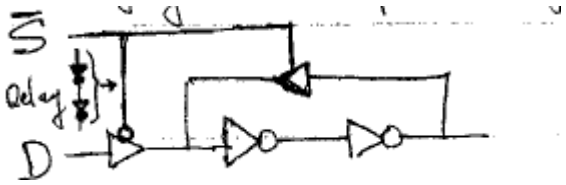
Logic Gate Memory Units

Inverter Chains

Consider the circuit shown below:

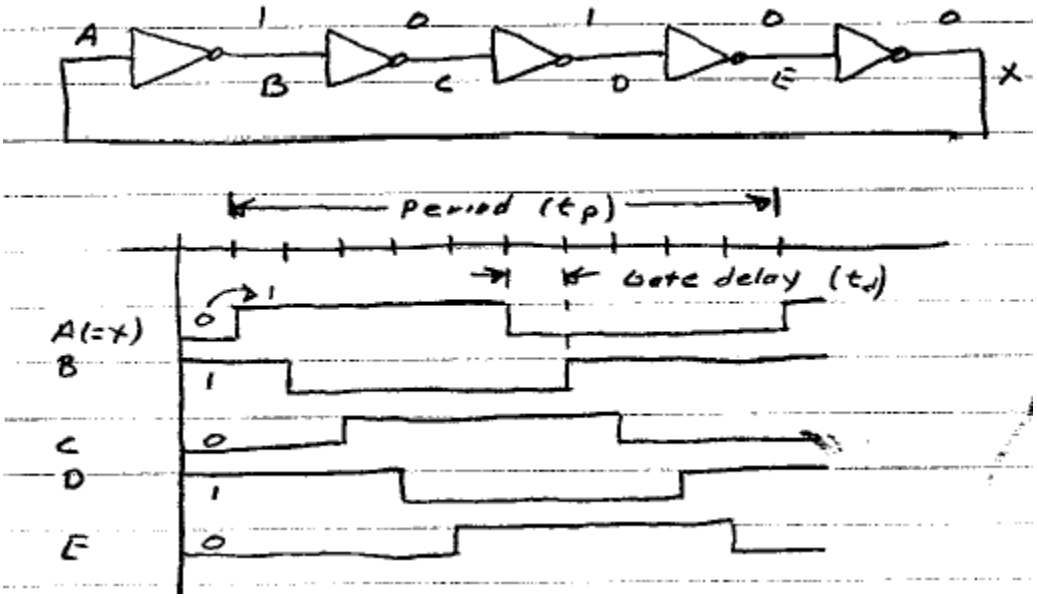


A 1 at the input to the first inverter becomes a 0 at the input to the second which reinforces the value at the first inverters input. Similar a 0 at the input is also reinforced. The circuit is a storage element. Some extra logic is required to open the feedback path what the input is changed:



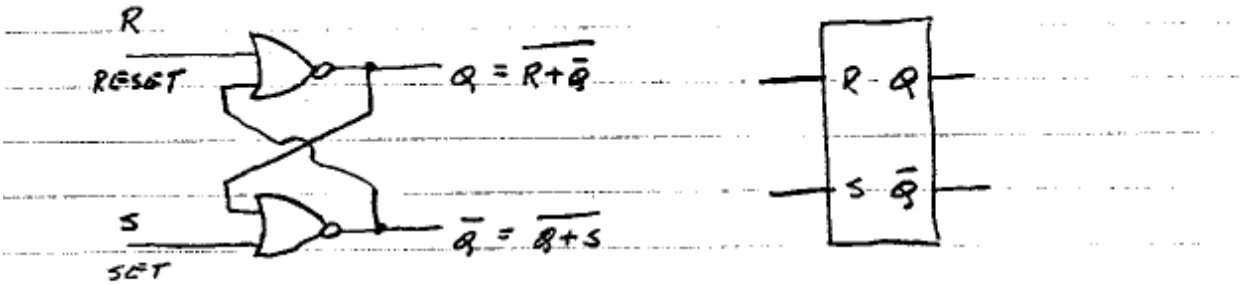
Cascaded inverts can also be used to build circuits whose outputs oscillate between low and high voltages. Such circuits are called *ring oscillators*. The figure below shows an inverter chain and the associated timing waveform. The waveform begins with A (=X). The odd number of inverters (five) results in a *period* tp = 10 time units. Duty cycle is defined as the percentage of time a signal is high during its period. In this case the signal has a 50% duty cycle.

In the ring oscillator, the duration of the period depends on the number of inverters in the chain. That that in the example here each inverter has a unit delay.



Cross-Coupled NOR Gates

In the NOR gate latch shown below the output of each NOR gate is fed back into the input of the other gate:



NOR gate RS latch RS latch symbol

The operation is summarized in the table below where, to start, we assume the present state of the output Q+ is 0 and the inputs to the *set* terminal S and the *reset* terminal R are both 0.

Action	Q	S	R	$\bar{Q}$	Q^+	Conclusion
Assume	0	0	0	1	0	Stable state
Apply 1 to S	0	1	0	0	1	Unstable; Q^+ changes
(Q becomes 1)	1	1	0	0	1	Stable
Remove 1 from S	1	0	0	0	1	Stable state after SET
Apply 1 to S again	1	1	0	0	1	No change in Q^+
Remove 1 from S	1	0	0	0	1	Stable state after SET
Apply 1 to R	1	0	1	0	0	Unstable; Q^+ changes
(Q becomes 0)	0	0	1	1	0	Stable
Remove 1 from R	0	0	0	1	0	Stable state after RESET
Apply 1 to S and R	0	1	1	0	0	Unacceptable; $Q^+ \neq \bar{Q}$

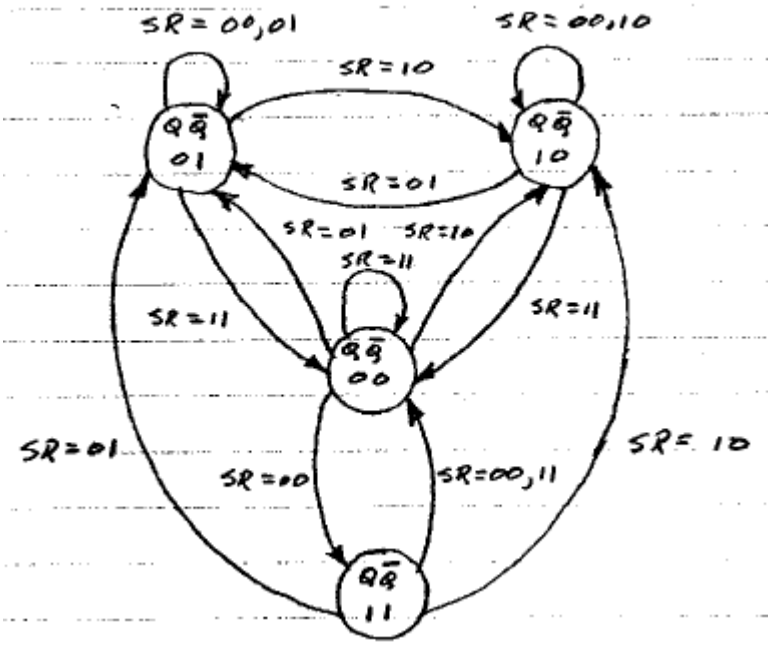
To SET the latch, a 1 is applied to S only. For $Q = 0$, $\bar{Q} = \overline{Q+S} = 0$ and $Q^+ = \overline{R+\bar{Q}} = 1$ the present state of the output is inconsistent with the input, the

systems is *unstable*, and Q must *flip*. After Q changes, the present sate changes to 1, and $\bar{Q}$ becomes 0, hence Q^+=1, a stable state. Note that if either input to a NOR gate is 1, the output is 0. Removing the input from S causes no chance. Hence this is a stable state after being SET. Applying another input to S causes no change.

To RESET the latch, a 1 is applied to R only. This results in an unstable system and Q^+ must *flop* to 0. A change in Q to 0 results in a stable output Q^+ = 0. Removing the input to R or applying another input to R produces no change. Hence Q^+ = 0 and $\bar{Q}$ =1 is the stable state after being RESET.

Only very short pulses are needed for triggering. Attempting SET and RESET simultaneously would create an ambiguous state with both Q^+ and $\bar{Q}$ = 0. This is unacceptable in a bistable unit and circuits are designed to avoid this condition.

Another way to represent the behavior of a cross-coupled NOR gates is called the *state diagram* as shown below.



The circuits state depends on the value of Q and NOT Q ($\bar{Q}$), so there are four possible states. Since there are two inputs, S and R, there are four transitions for each state.

The states 01 and 10 are the normal ones for the circuit. When S=1, we enter state 10 (Q=1, NOT Q = 0). When R=1, the state changes to 01 (Q=0, NOT Q = 1). When S = R = 0 the current state is held.

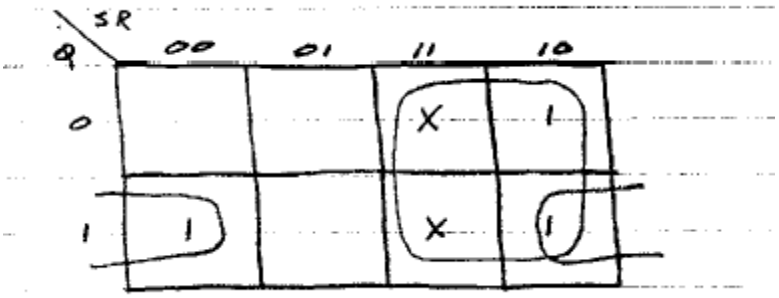
When S=R=1 the circuit enters the forbidden state 00. It stays as long as those inputs are held. As soon as one input returns to 0, the circuit returns to state 01 or 10. If the current state is 00 and S=R=0, the circuit enters the forbidden state 11. It does not stay very long before returning to state 00 if S and R remain 0. If the delays are match the circuit can oscillate between these states forever. This is known as a *race condition*. The circuit should never be put in state 00.

From the circuit for the RS latch or the table we can deduce the detailed truth table for the latch as shown below:

S	R	Q	Q^+	
0	0	0	0	HOLD
0	0	1	1	
0	1	0	0	RESET
0	1	1	0	
1	0	0	1	SET
1	0	1	1	
1	1	0	X	Not allowed
1	1	1	X	

Where Q+ is the next state output based on the current state Q and inputs S & R.

The K-map for the truth table is given below:



From the K-map we get the characteristic equation:

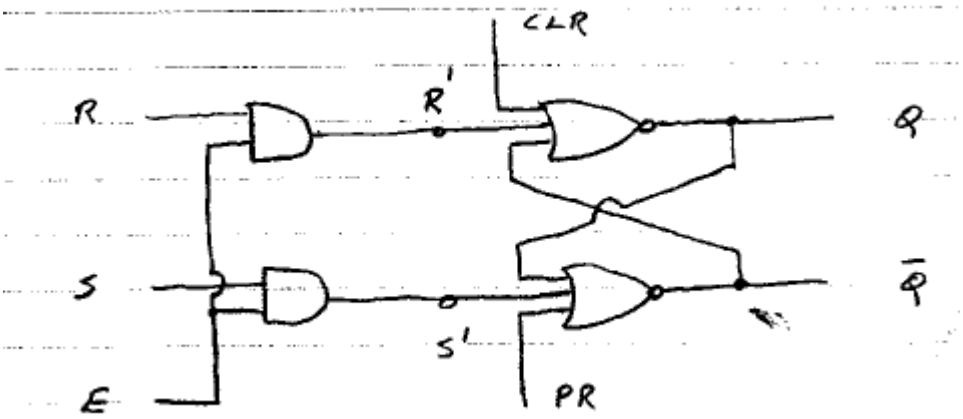
$$Q^+ = S + \bar{R} \cdot Q$$

This equation summarizes the behavior of the RS latch. For example, if S=1 and R=0, the next state Q+ becomes 1 independent of the current state. When S=0 and R=1, the next state is forced to 0, independent of the current state.

Timing Waveforms

In the RS latch a 1 input at S will SET the output Q to 1. To RESET the latch, a 1 is applied to input R. The duration of the input (it must exceed a certain minimum time) and the time at which the input signal is applied are not significant. Such a latch responds to the *asynchronous* inputs.

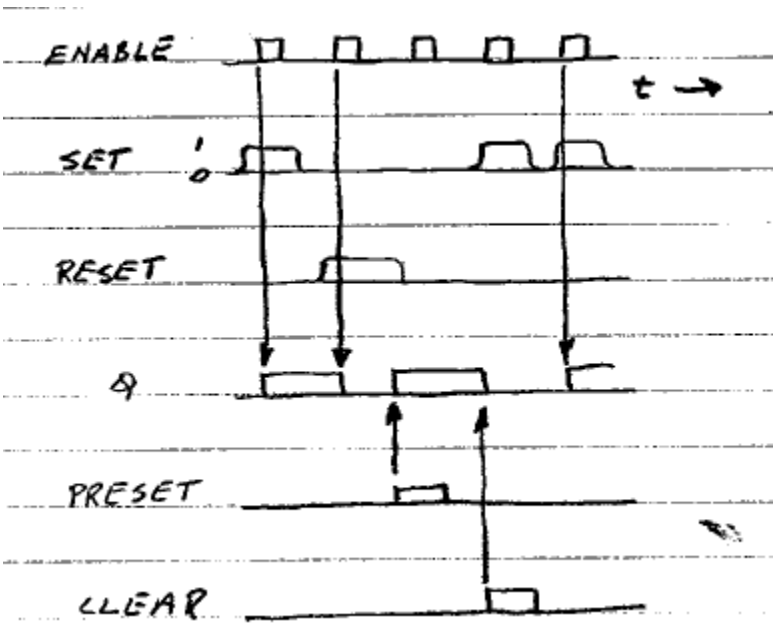
A more sophisticated latch using two AND gates is shown below.



Here an input is effective only when *enabled* by a 1 input at terminal E. In digital systems composed of many elements, it is usually necessary for the outputs of all elements to be synchronized. The synchronizing signal may come from a *clock.* The enabling terminal is frequently designed CLOCK (CK). In a clocked system, transactions cannot happen at random by occur in an order one-step-at-a-time fashion. In addition to the synchronous inputs R and S, there may be asynchronous inputs to *clear* or *preset* the *flip-flops*.

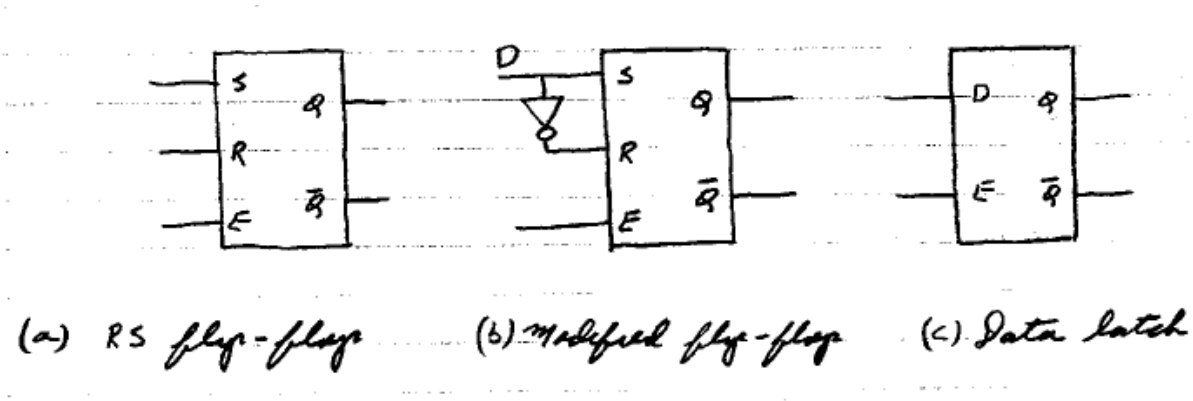
Flip-flips different from latches in that their outputs change only with respect to the clock, whereas latches change outputs when their inputs change.

The operation of a clocked RS latch (or flip flop) is shown below. Initially, output Q=0. If a 1 appears at SET, when ENABLE goes to 1 the flip-flop is set with Q=1. At the next clock pulse, the presence of a 1 at RESET forces the output to 0. At any time, a 1 at PRESET forces the output to 1; a 1 input at the CLEAR terminal overrides other inputs and forces Q to 0.



The Data Latch

The symbol for a simple RS flip-flop (without PRESET or CLEAR) is shown below. The ambiguous state which results when R=1 and S=1 simultaneously can be avoided by modifying the circuit as shown in (B) below.

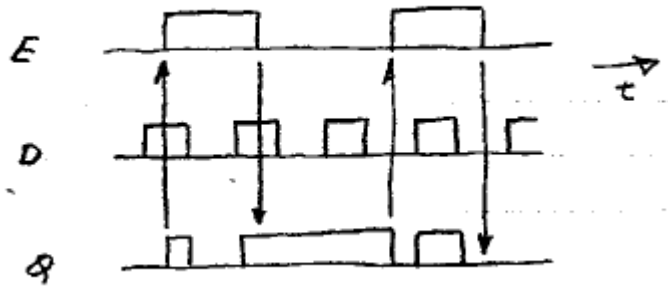


By connecting an inverter between the R and S terminal and using only one input signal, the ambiguity is avoided and the number of terminals are decreased. When ENABLE is HIGH, the output Q follows the input D, when ENABLE goes LOW, no change in Q is possible, and the output is *latched* at the previous data value.

This data latch is widely used as an element in digital systems.

Example

The enable and data inputs to a data latch are shown below. The product the waveform of the output.

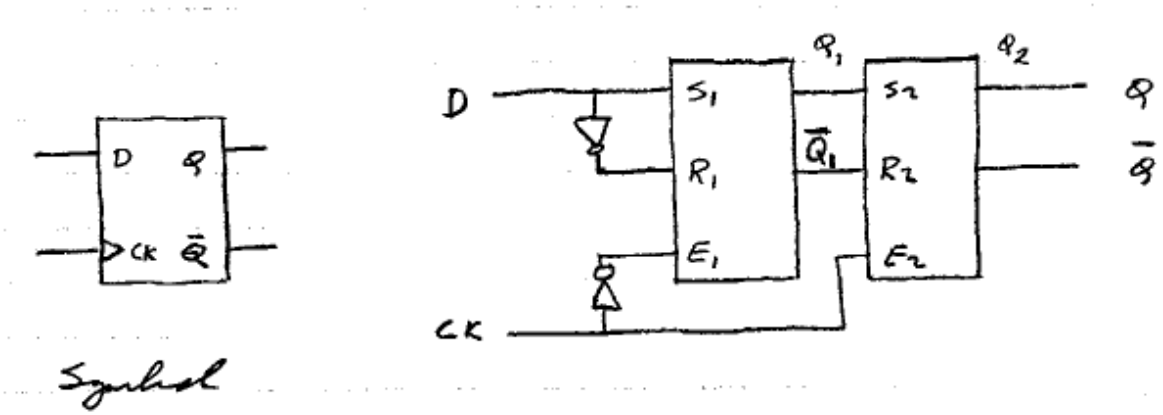


The output Q follows input D whenever enabled (E=1). When E goes to 0, the output remains latched in the previous condition.

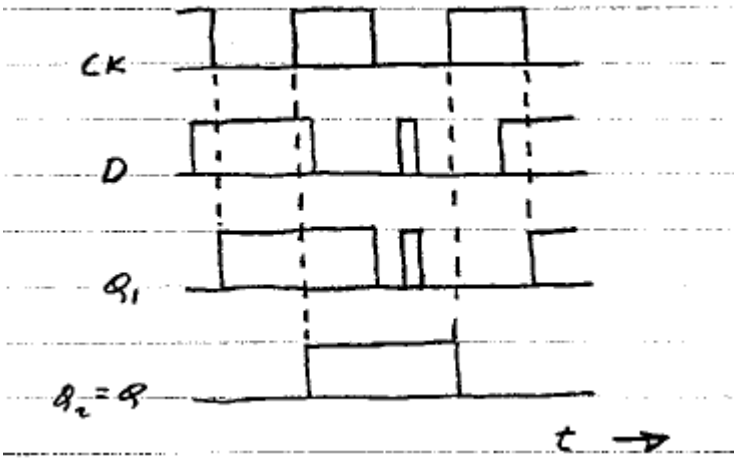
In the figure, When E goes HIGH, D=1 so Q follows D and becomes 1. As long as E is HIGH, Q follows any change in D. When E goes LOW, Q=D=1 and remains so. The Q waveform is as shown.

The D Flip-Flops

Sometimes it is desirable to delay the transfer of data from input to output. For example, we may wish to maintain the present state Q, while a new state is being read that will be transferred later. The D (delay) flip-flop is shown below:



It is a data latch with a second RS flip-flop. The data latch is enabled when the clock signal goes LOW, but the following RS flip-flop is enabled when the clock signal goes HIGH. We see that Q1 follows D whenever CK is LOW, but any change in the output Q=Q2 is delayed until the next upward transition of CK. This is an *edge-triggered* flip-flop; Q1 follows D while CK is LOW, then on the leading edge of the clock pulse, the value of D is transferred to output Q. On the logic symbol, the small triangle indicates an edge-triggered device.

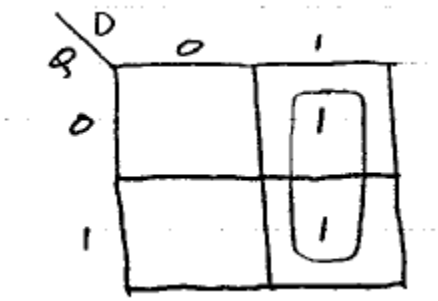


Because the output change only at the instant the clock goes HIGH, the output can be synchronized without the outputs of other elements. In addition, a sudden spurious change in D, like the one shown above, will not affect the output.

The truth table for the D flip-flip is shown below:

D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

The k-maps as obtained from the above table is given below:



From the K-maps we get the characteristic equation:

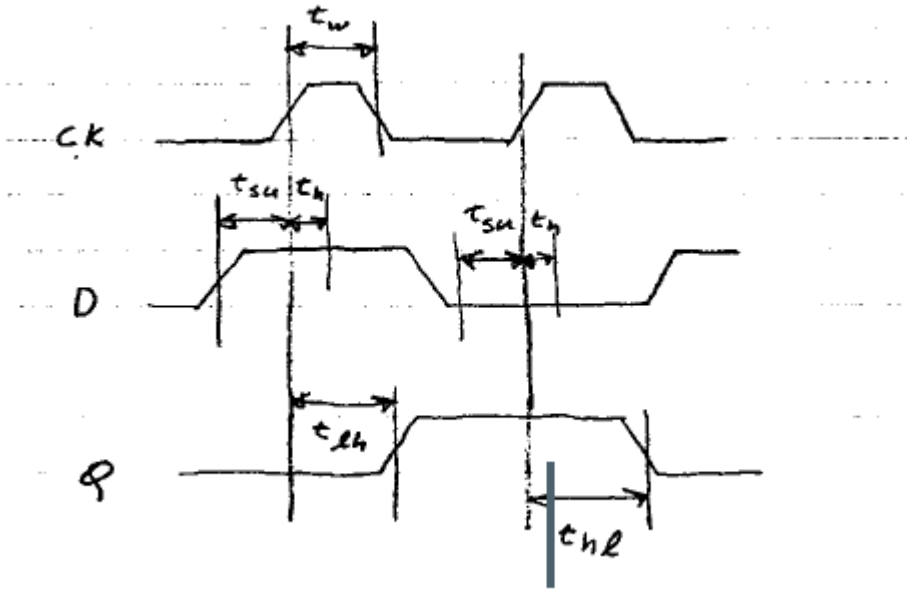
$$Q^+ = D$$

Timing

Timing is more complicated in sequential circuits than in combination circuits where glitches are the only concern. Sequential logic, on the other hand, must examine both the current input & current state to determine the outputs and the next state. In addition, outputs can change in response to clocking changes as well as input changes.

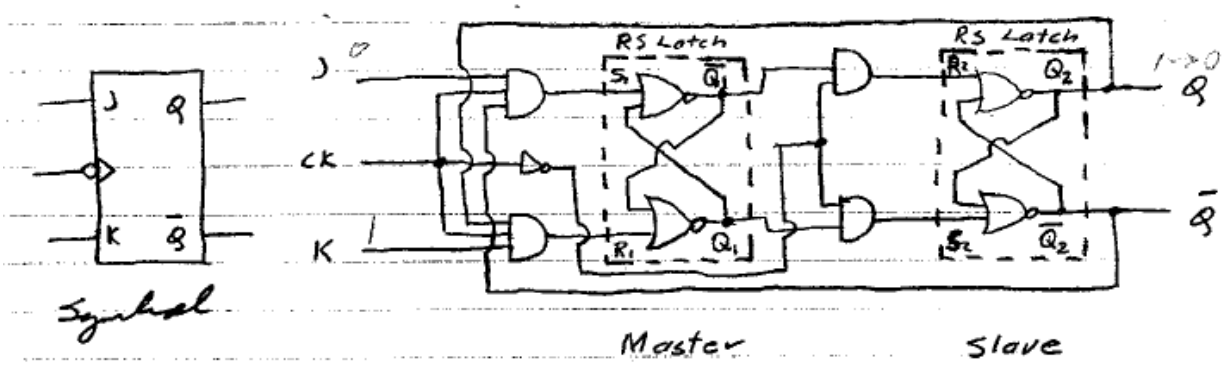
For proper operation the data must be stable for a few nanoseconds before the device is clocked (called the *setup time* or t_{su}) and must remain stable for a few

nanoseconds after the clocking is initiated (the *hold time*, t_h). The figure below shows the timing constraints for a typical edge-triggered flip-flop. In the diagram t_w is the clock signal minimum duration, t_{hl} is the propagation delay from high to low, and t_{lh} the propagation delay from low to high. Typical numbers are: t_w = 25 nS, t_{su} = 20 nS, t_h = 5 nS, t_{hl} = 25 nS, t_{lh} = 13 nS.



The JK-Flip Flop

A very popular memory unit is the JK flip-flop shown below. In its most common form, the output changes state on downward transitions of the clock pulses. The small circle on the symbol identifies this as a *falling-edge-triggered* flip-flop. The operation is improved by using a *master* flip-flop that is enabled on the upward transition of the clock pulse while the *slave* flip-flop is inactive. The slave is enabled on the downward transition and follows its master, i.e. it takes on the state of the master.



Because of the feedback connections from the output to the input, the output of the JK flip-flop depends on the state of the inputs and outputs at the instant the clock goes LOW. In addition the ambiguity (R=S=1) is avoided. A truth table showing S1, R1, Q+ (the next output state) for all values of J, K, and Q (the present output state) is shown below:

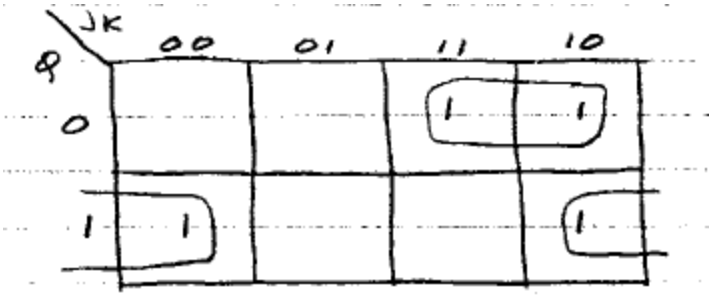
J	K	Q	S_1	R_1	Q^+	
0	0	0	0	0	0	HOLD
0	0	1	0	0	1	
0	1	0	0	0	0	RESET
0	1	1	0	1	0	
1	0	0	1	0	1	SET
1	0	1	0	0	1	
1	1	0	1	0	1	Toggle
1	1	1	0	1	0	

From the truth table we see the following modes of response are possible:

1. With inputs J=K=0 the clock as no effect, and the flip-flop remains in its present state Q.
2. When J and K unequal, the unit behaves like an RS flip-flop where J=S and K=R. That is J=1 K=0 SETs the output on falling clock edge, and J=0 K=1 RESETS the output on falling clock edge.

3. With inputs J=K=1 the flip-flop toggles; that is the output changes each time the clock goes low.

The K-map for the truth table is given below:



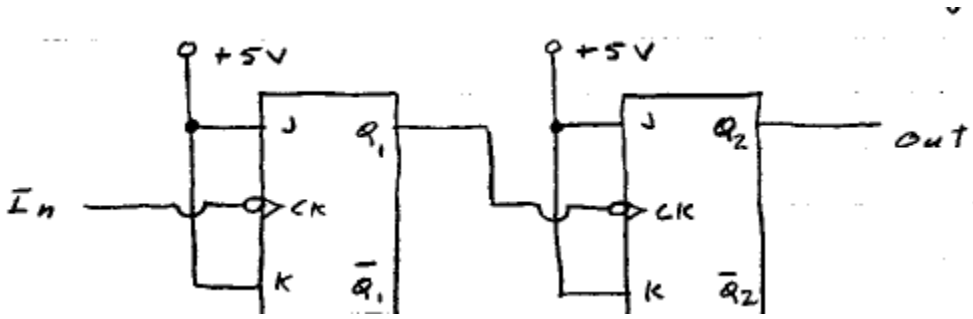
From the K-map we get the characteristic equation:

$$Q^+ = J \cdot \bar{Q} + \bar{K} \cdot Q$$

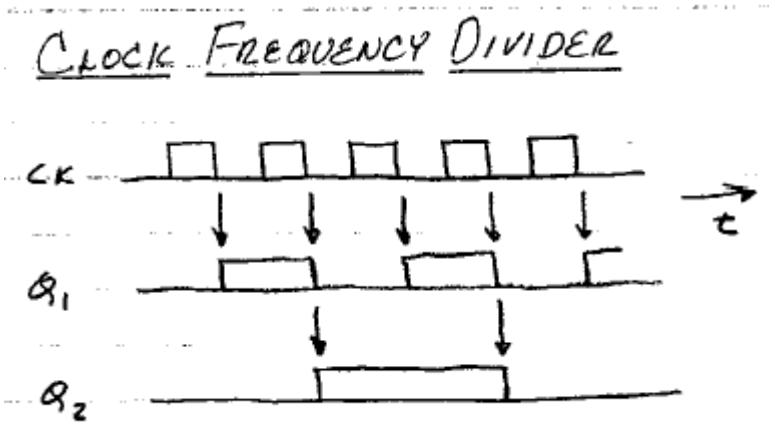
This equation summarizes the behavior of the JK flip-flop.

Example:

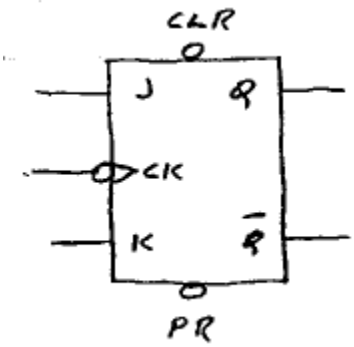
Two JK flip-flops that respond to downward transitions are connected in tandem as shown. For a 2 KHz square wave input, determine the output:



Since we have 1 inputs at both J and K, the output will change each time the clock goes LOW. For a 2 KHz square wave input the output of the first flip-flop will be a 1 KHz square wave. Since this is the input to the second flip-flop, its output will be 500 Hz.



The JK flip-flop is used in a number of digital computer applications such as counters, arithmetic units, and registers. For greater flexibility, some versions include PRESET and CLEAR capabilities as shown below. In the unit shown PR and CLR are normally held HIGH.



The small circles (inversion, or “active low”) indicate that if PR goes LOW, Q is forced to 1; where is CLR goes LOW, Q is forced to 0.

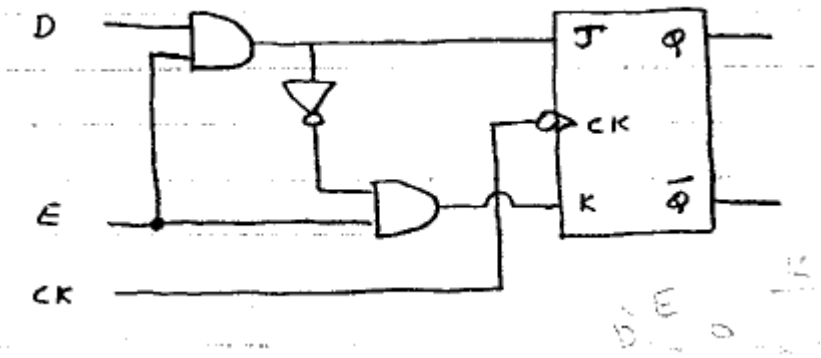
Example

Connect a JK flip-flop to function as a data latch. That is, when it is ENABLED, the DATA is to be transferred to Q when clock goes LOW.

Logically this means that: when ENABLED, Q should follow J=D, which requires K is not equal to J. When DISABLED, Q should remain “latched” in its present state, which requires K=J=0. Thus:

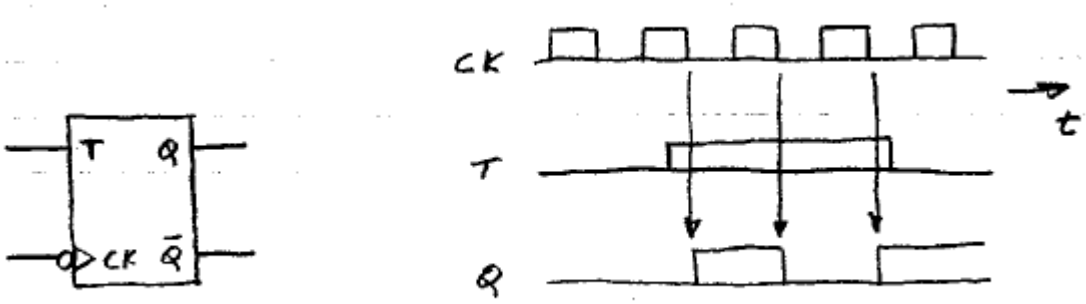
E	J	K
0	0	0
1	D	$\bar{J}$

Which results in the following logic synthesis:



The T Flip-Flop

With the J & K inputs tied together (resulting in one terminal), the JK unit becomes the T or *toggle* flip-flop as shown below:

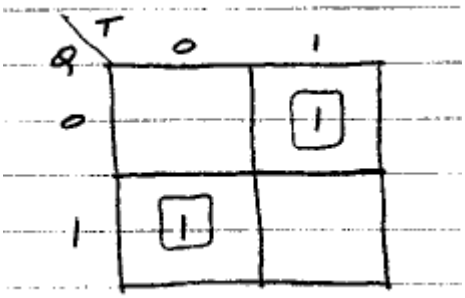


For T=0 (J=K=0) the clock pulse has no effect on output Q. For T=1 (J=K=1), the flip-flop toggles each time CK goes to LOW. The waveforms shows that for T held HIGH, the output is a square wave of half the frequency of the clock; the device is a frequency divider.

The truth table for a T flip-flop is shown below:

T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

The K-map is obtained from the above table and shown below:



Finally from the K-Map we can get the characteristic equation:

$$Q^+ = T \cdot \bar{Q} + \bar{T} \cdot Q$$

Conversion of One Flip-Flop Type to Another

Any flip-flop can be implemented as combinational logic for the next state function in conjunction with a flip-flop of another type.

A general procedure to map amount the different kinds of flip-flops is based on the concept of an excitation table. This table lists all possible state transitions and the values of the flip-flop inputs that cause a given transition to take place.

The figure below gives excitation tables for RS, D, JK, and T flip-flops.

Q	Q^+	R	S	D	J	K	T
0	0	X	0	0	0	X	0
0	1	0	1	1	1	X	1
1	0	1	0	0	X	1	1
1	1	0	X	1	X	0	0

If the current state is 0 and the next state is to be 0, then the first row of the table describes the flip-flop inputs to cause that state transition to take place. For the RS latch it doesn't matter the value on R provided that S=0. If using a D flip-flop the input is set to the next discrete state, which is 0 in this case.

If a JK flip-flop us being using, the transition from 0 to 0 occurs when J=0. The value of K does not matter. If using a T flip-flop, the transition does not change the current state, so the input should be 0. The same kind of analysis can be applied to complete the excitation table for the three other cases.

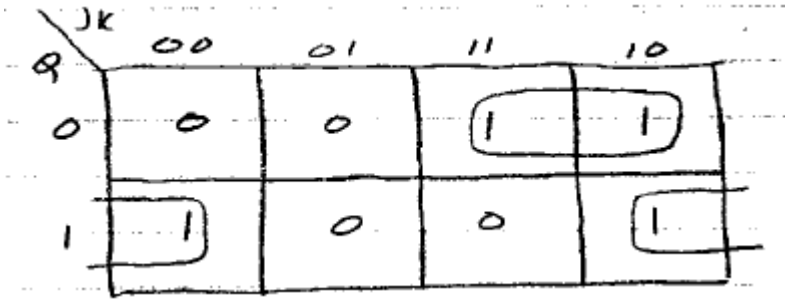
The procedure is to use the excitation table for the flip-flops in question to form a K-map. The K-map layout is for the desired flip-flop and the values entered are for the flip-flops being used. The method is elaborated in the following examples.

Example: JK with D

Show how to implement a JK flip-flop starting with a D flip flop.

The excitation table for the JK and D flip-flops are shown below. The K-map is formed for the JK flip-flops with the values for the D flip-flops entered in the map.

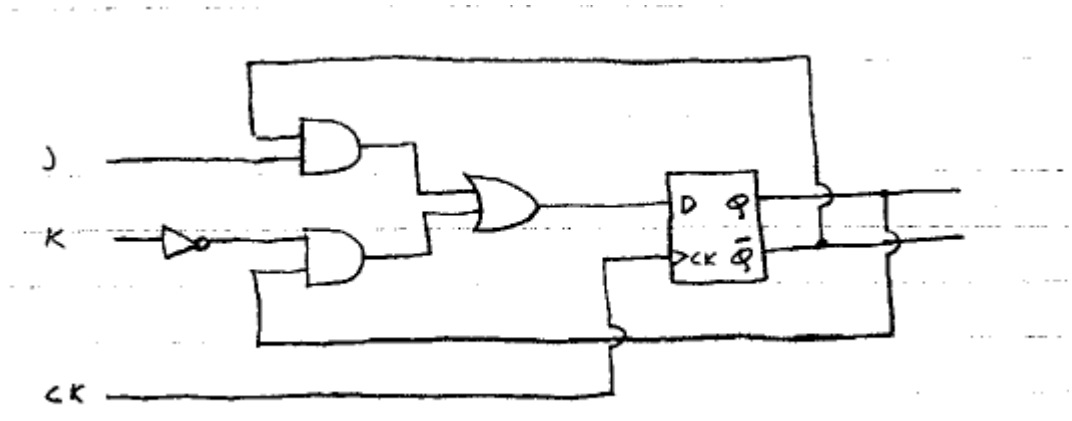
Q	Q^+	D	J	K
0	0	0	0	X
0	1	1	1	X
1	0	0	X	1
1	1	1	X	0



From the map we see that:

$$D = J \cdot \bar{Q} + \bar{K} \cdot Q$$

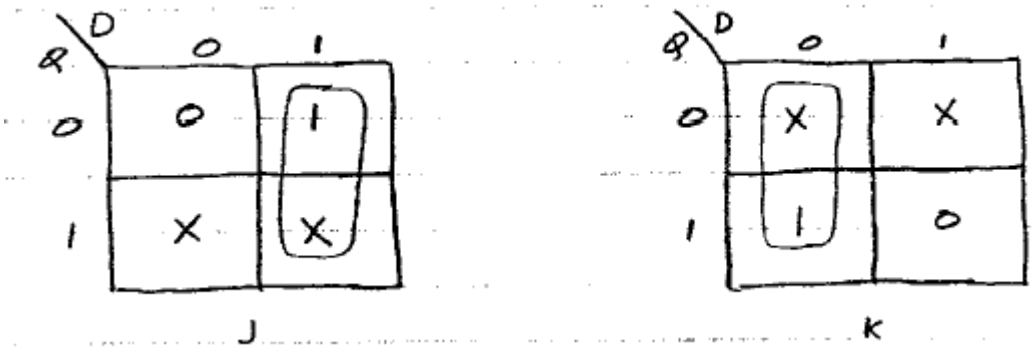
Note that since the transition from 100 to 101 does not change the state of the flip-flops the hazard is of no concern. The implementation is shown below:



Example: D with JK

Show how to implement a D flip-flip starting with a JK flip-flop.

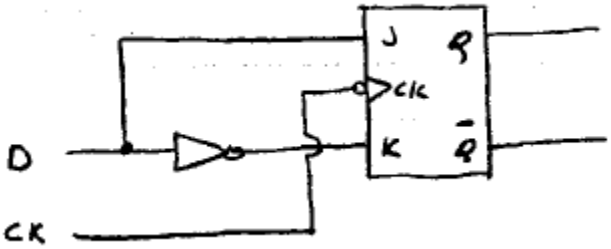
The excitation table is as in the last example. The K-maps are formed for the D flip-flops with values for J and K entries in the maps:



From the maps we see that:

$$J = D \quad \text{and} \quad K = \bar{D}$$

Thus the implementation becomes:



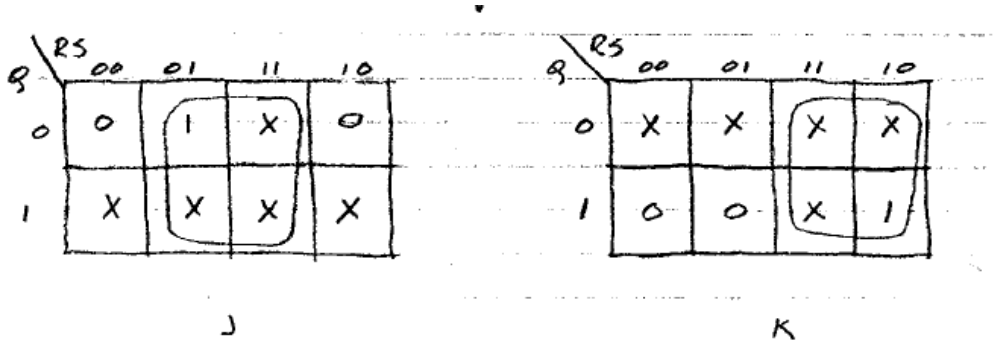
Example: RS from JK

Implement an RS flip-flop starting with a JK flip-flop.

The excitation table for the RS and JK flip-flops is shown below:

Q	Q^+	R	S	J	K
0	0	X	0	0	X
0	1	0	1	1	X
1	0	1	0	X	1
1	1	0	X	X	0

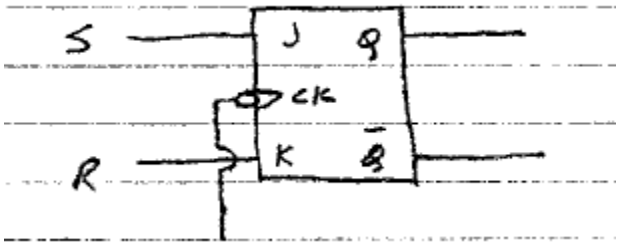
The K-maps for the RS are formed from with JK values entered in the maps:



From the K-maps we see that:

$$J = S \quad \text{and} \quad K = R$$

Thus:



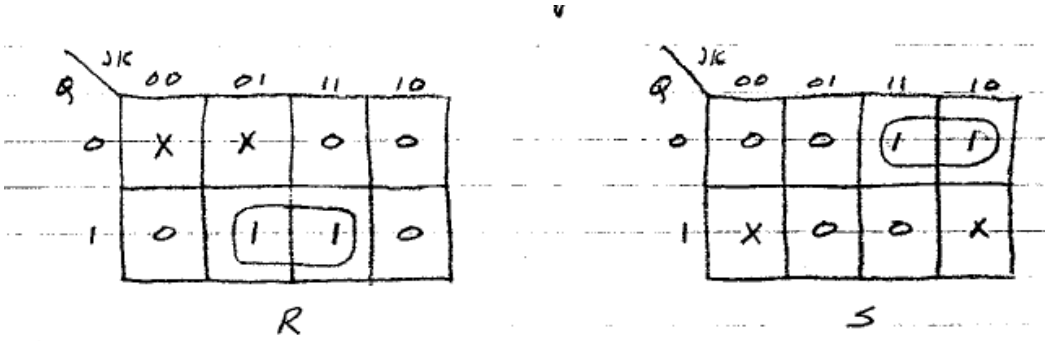
Example: JK from RS

Implement a JK flip-flop starting with a RS flip-flop.

The excitation table for the RS and JK flip-flops is shown below:

Q	Q^+	R	S	J	K
0	0	X	0	0	X
0	1	0	1	1	X
1	0	1	0	X	1
1	1	0	X	X	0

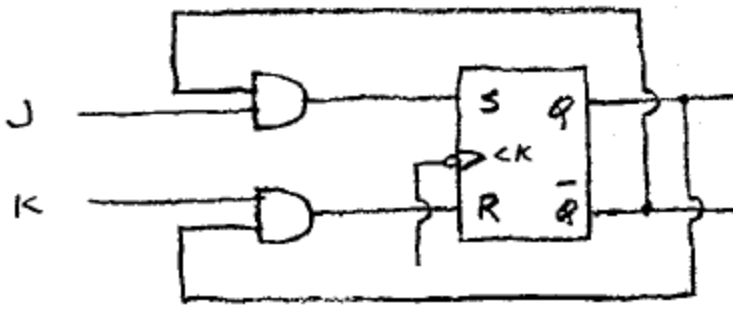
The K-maps for the JK are formed from with RS values entered in the maps:



We see that:

$$R = K \cdot Q \quad \text{and} \quad S = J \cdot \bar{Q}$$

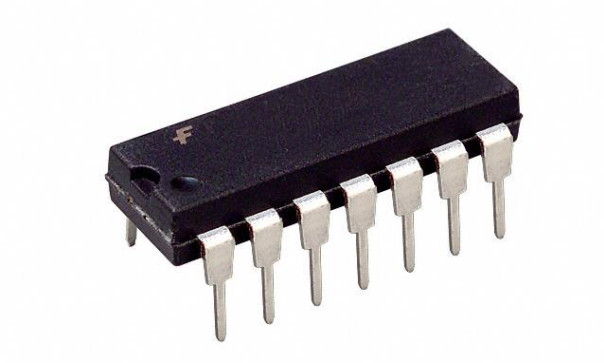
Thus:



Sets of flip-flops can be used to represent *binary numbers* in which each digit corresponds to a value of Q (0 or 1) of a flip-flop. A *register* is a set of flip-flops in which binary data can be stored. Flip-flops can be connected to serve as a *counter* in which the number stored is the number of events being counted.

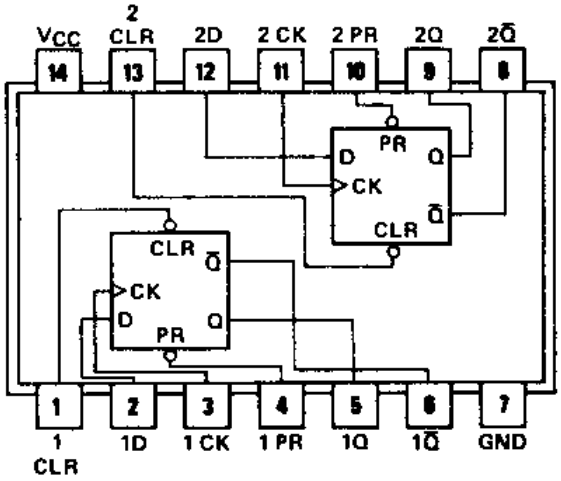
Practical Matters

Logic gates and memory elements are available in IC form which are small in size, have low power consumption, and have low cost. Typically they are used in DIP (Dual In-line Package), with 14 or 16 pins. A photo of a 14-pin DIP is shown below:

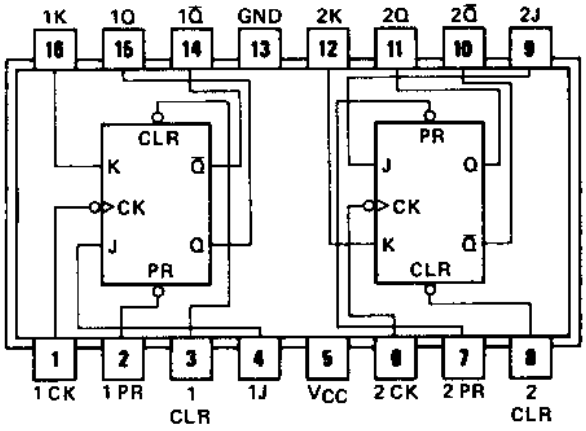


Diagrams for TTL dual D and JK flip-flops are shown below, where the pin numbers correspond to pins on the above package.

7474

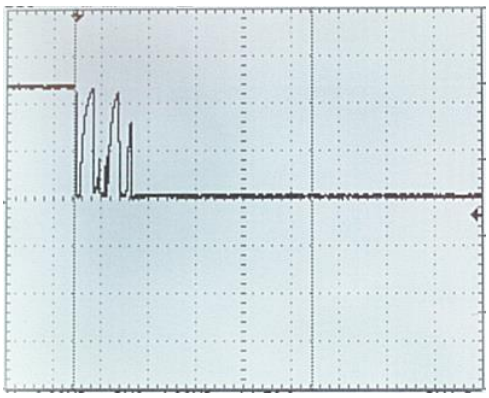


7476



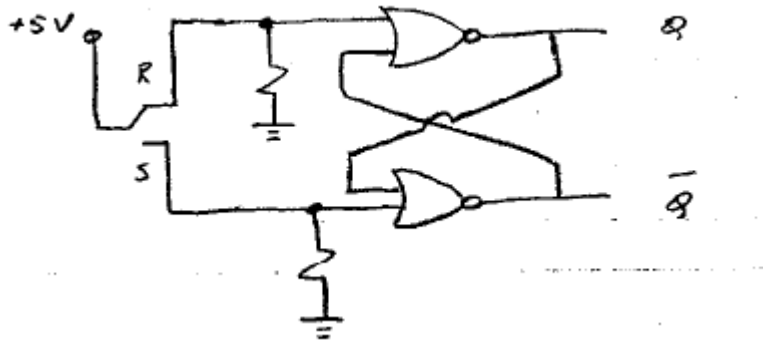
Debouncing Switches

When a switch is flipped from one terminal to another, it does not make a clean, solid contact with the new terminal. Instead, it bounces several times before coming to rest. Because of this and the fact that TTL chips treat floating inputs as 1's, there are several transitions from 1 to 0. This cause errors in reading the switches output. The following diagram shows an example of switch bounce:



If the switch was for example counting votes, that single push would be read as several quick pushes.

The problem is solved by using an RS latch. The figure below shows an RS latch, a single pole double throw (SPDT) switch, and two resistors connected to ground:



When the switch is in the reset position, R is high and Q is low. If the switch is moved so that it is in transition towards S, the grounded resistors pulls the latch low. The latch is in its holding state since both inputs are 0.

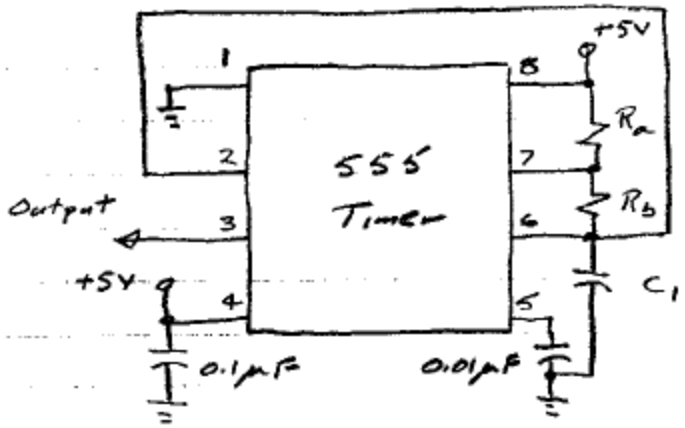
When the switch first touches S the latch goes high and Q=1. If the switch bounces, temporarily breaking the connection, the latch input returns to 0, leaving the latch in a holding state. If the switch bounces back, remaking the S connection, the latch is set again and so no state change occurs.

As long as the switch does not bounce enough to remake R, the Q output will remain high as long as the switch is bouncing into its final position.

A similar analysis applies for a switch moving from S to R.

The 555 Timer

The 555 timer is a programmable timer chip. The circuit diagram is shown below:



The period an duty cycle are determined by placing the appropriate resistors and capacitors between the pins. The following formula are used to calculate the clocks characteristics:

Clock high time = $0.7\ (R_a + R_b)\ C_1$

Clock low time = $0.7\ (R_b)\ C_1$

Clock Period = high time + low time = $0.7\ (R_a + 2R_b)\ C_1$

Clock Frequency = 1 / (Clock Period)

Duty Cycle = $(R_a + R_b) / (R_a + 2R_b)$

Note that the 555 timer draws large currents for short periods of time when the output changes state. To minimize the resulting spikes that can upset the rest of the circuit it is important to put a 0.1 uF bypass capacitor from the 5V pin, pin 4, to ground.

Example:

Design a clock signal with a period of 500 uS and a 75% duty cycle.

Clock frequency = 1/period = 1/500E-6 = 2000 Hz = 2 KHz

$$\text{Duty cycle} = \frac{R_a + R_b}{R_a + 2R_b} = 0.75$$

or

$$R_a + R_b = 0.75(R_a + 2R_b)$$

$$(1 - 0.75)R_a = (1.5 - 1)R_b$$

$$0.25 R_a = 0.5 R_b$$

$$R_a = 2 R_b$$

Let

$$R_a = 3600\ \Omega \quad \text{then} \quad R_b = 1800\ \Omega$$

$$\text{clock high time} = 0.75(500)10^{-6} = 375\ \mu S$$

$$\text{clock low time} = 0.25(500)10^{-6} = 125\ \mu S$$

$$= 0.7(R_b)C_1$$

so

$$C_1 = \frac{125(10^{-6})}{(0.7)(1800)} = 0.099\ \mu F$$

$$\simeq 0.1\ \mu F$$

Sequential Logic Applications

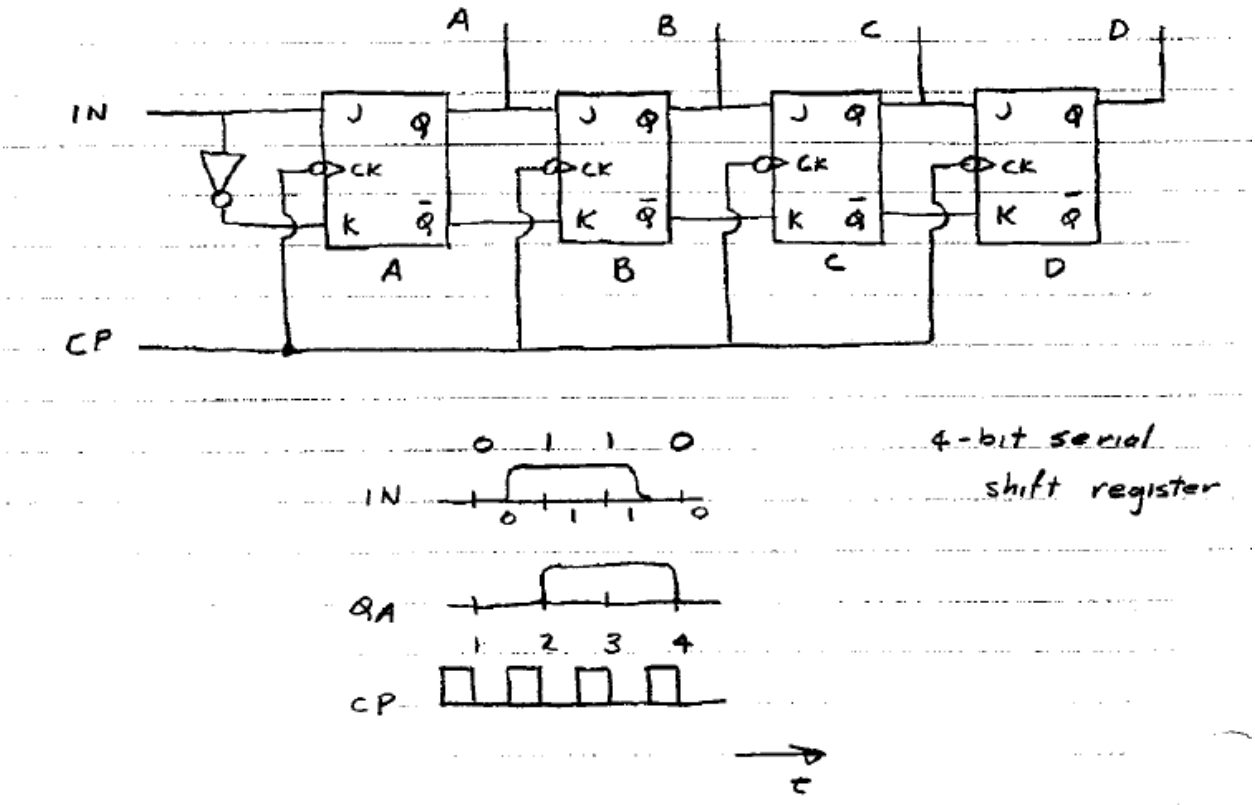
In this section we will examine three useful sequential logic components: registers, counters, and memories.

Registers

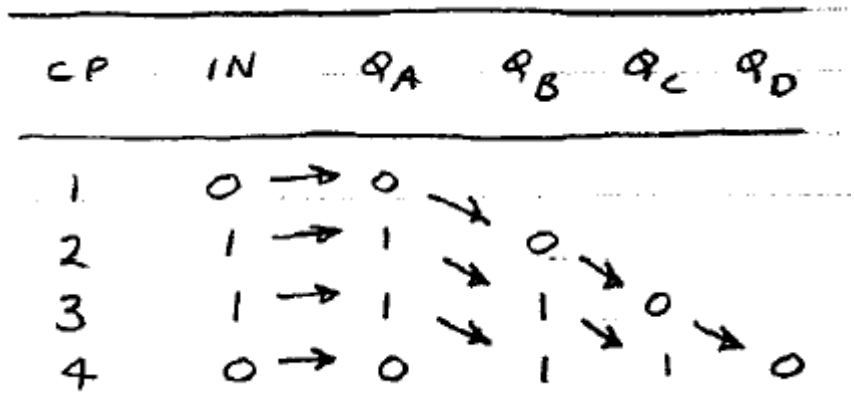
In addition to logic circuits that process data, digital systems must include memory devices to store data and results. A flip-flop can store or "remember" one digit of a binary number, one bit. A *register* is an array of flip-flops that can temporarily store data or information in digital form. A great variety of registers are available in IC form.

Shift Registers

The serial shift register below consists of four trailing-edge-triggered JK flip-flops connected so that J ≠ K. At the trailing edge of each clock pulse Q follows J in each flip-flop of the 4-bit register. The data are entered serially, that is, one bit at a time, and shifted right through the register at each clock pulse.

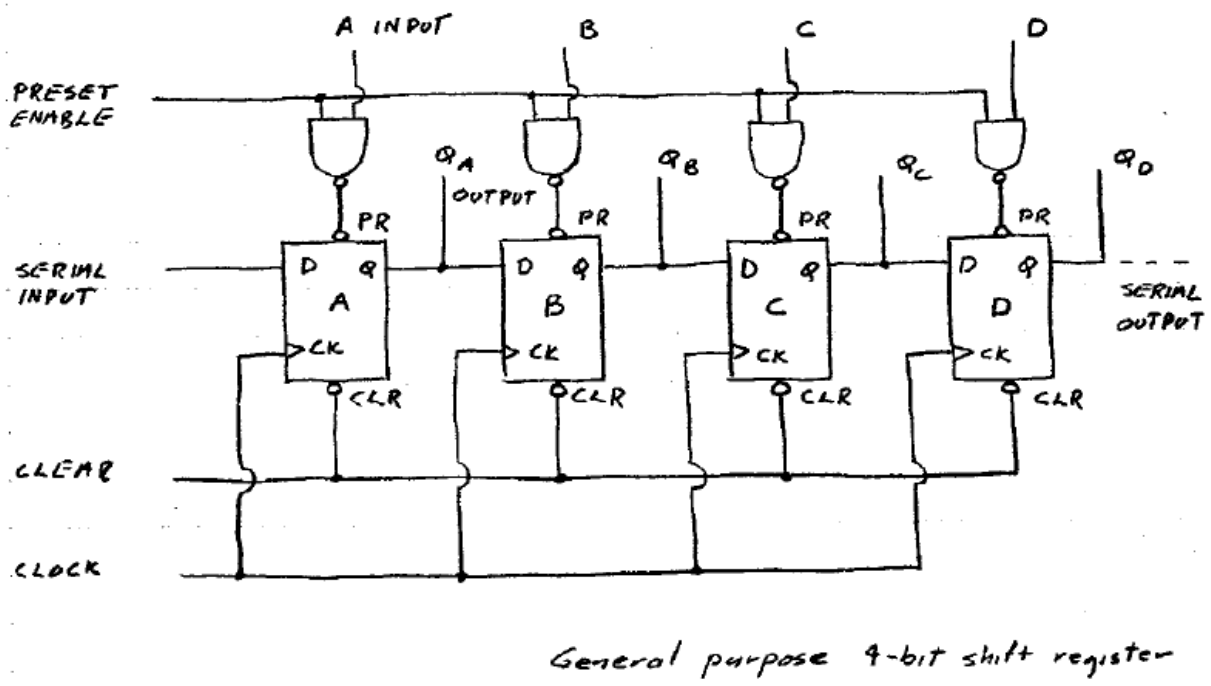


The table below shows how 0100 would be placed in the register:



Begin with the least significant bit. With a 0 at IN=J_A and K ≠ J, at the trailing edge of the first clock pulse (CP1), Q_A follows J_A and the LSB is transferred to the output of flip-flop A. During the next clock cycle, B=Q_A=0 and the second bit, a 1, is applied at IN = J_A. At CP2, the 0 is transferred to Q_B (i.e., shifted one position to the right) and the 1 is transferred to Q_A. After four clock pulses, the 4-bit number is stored in the register and 0110 is available at parallel outputs ABCD. An application of this register is an serial-to-parallel converter. A single input line and four output lines are required.

The shift register shown below consists of D flip-flops with CLEAR and PRESET. The symbols indicate that the flip-flops are cleared to 0 if CLR goes LOW while PR is inactive (HIGH) (clearing is independent of the clock level). On the positive-going edge of the clock signal, the input at D is transferred to Q.



General purpose 4-bit shift register

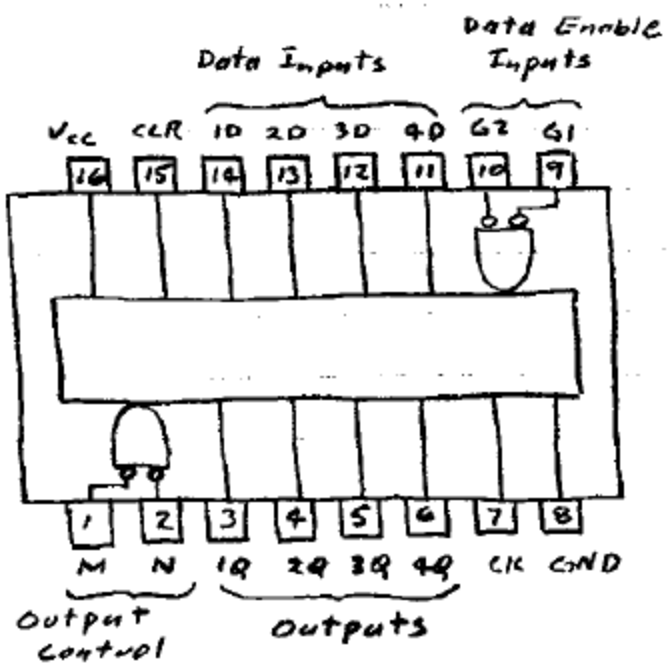
Since both inputs and outputs are accessible this unit can function as:

i. A 4-bit storage register (serial or parallel)
ii. As a serial-to-parallel converter
iii. As a parallel-to-serial converter

For function (iii), all stages are cleared to 0, and the input data are applied to the A,B,C,D INPUTS. A HIGH signal to PRESET ENABLE is NANDed with any 1 input to send PR LOW setting Q to 1 in that stage (independent of the clock level). At the next upward transition of the clock pulse (unless the clock is inhibited), the data are shifted to the right; the value of Q_D is output and the valued of Q_C is transferred to Q_D, etc.

A Practical Register

The 74173 TTL is a 4-bit register incorporating D flip-flops. The pin diagram is shown below:



For M+N=0, normal logic states are available; for M+N=1, the outputs are disconnected. When both G1 and G2 are LOW, data at the D inputs are loaded on the next positive transition of the CLOCK.

Counters

Flip-flops can be connected as *counters* to count random events, or to divide a frequency or to measure a parameter (e.g.: time, distance, speed).

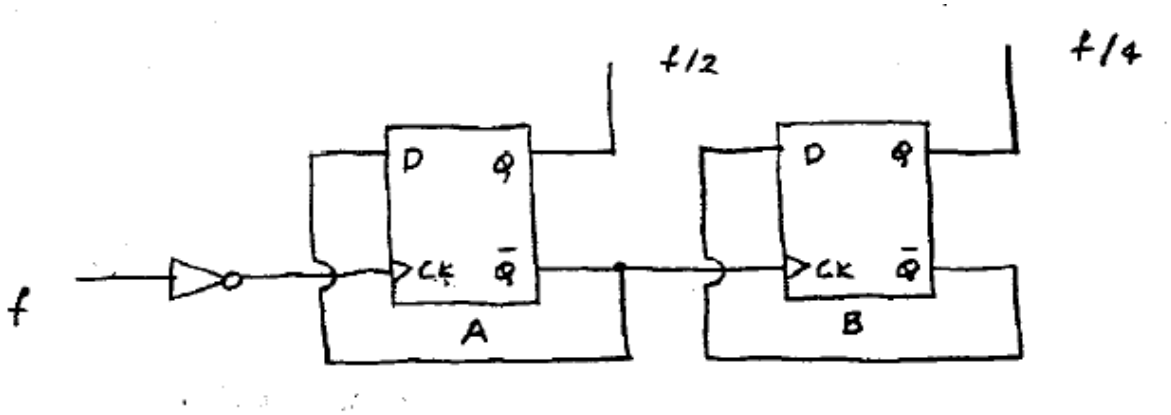
Counters are used to keep track of operations in digital computers and in instrumentation. JK, T and D flip-flops are used in counter design.

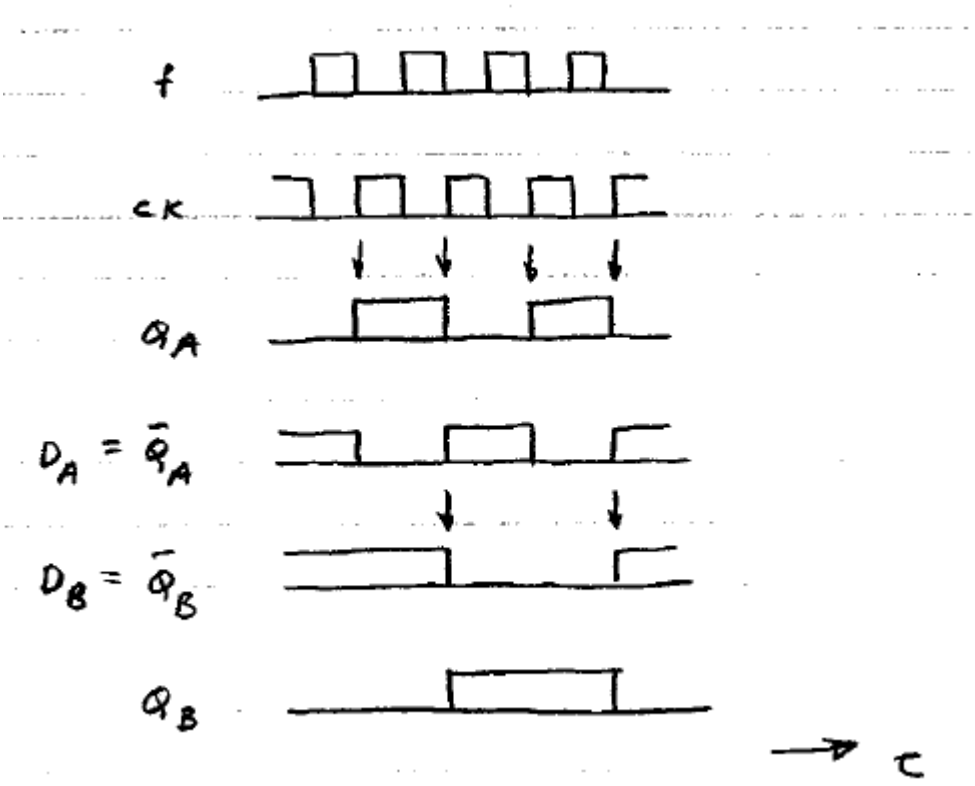
Types of Counters

Divide-by-n Circuits

A divide-by-n counter produces one output pulse for n input pulses: n is called the "modulo" of the counter. As we see earlier a J-K flip-flop divides by two or four for J&K help HIGH, the output Q is the number of CK inputs divided by two.

Two D flip-flops can be used to obtain a divide-by-four circuit as shown below:

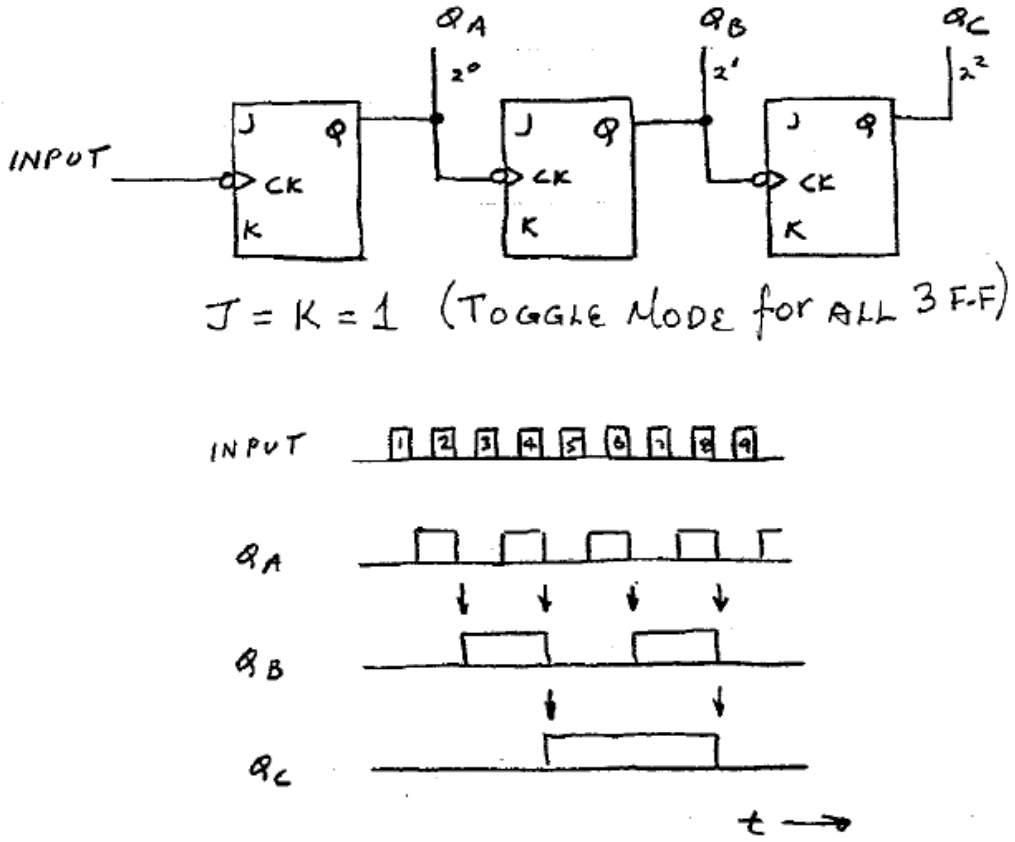




With Q_A and Q_B cleared (0), when clock sign f goes LOW, CK goes HIGH and $D_A = \overline{Q}_A = 1$ is transferred to Q_A. The next time f goes LOW, Q_A changes to low; time cycles at f complete one cycle of Q_A, or Q_A=f/2. When Q_A goes low, $\overline{Q}_A = CK_B$ goes HIGH and $D_B = \overline{Q}_B = 1$ is transferred to Q_B; from cycles at f complete one cycle of Q_B or Q_B = f/4.

Binary Ripple Counter

The counter shown below shows a 3-bit ripple counter using JK flip-flops in cascade. With J and K held HIGH (the +5V connections are not shown), the flip-flops toggle at each downward transition of the pulse at CK. The bit Q_A changes state after each input pulse goes low. The next bit QB changes state whenever Q_A goes LOW since Q_A supplies CK_B. Similarly, Q_C changes state whenever Q_B goes LOW.



The table below shows the outputs QA, QB and QC for the first 8 clock pulses. Note that this counter is counting from 000 to 111. After the count reaches 111, counting begins again from 000. Thus, a 3-bit counter cycles through 8 states 000 through 111. Similarly, a 4-bit counter will cycle through 16 states, 0000 through 1111. In general, an n-bit ripple counter will cycle through 2^n states, 0 through 2^n-1.

Count	Q_C	Q_B	Q_A
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

Counters that cycle through 2^n states, from 0 to 2^n-1, are known as *up-counters*. In some cases it is desirable to counter down, and the circuits are known as *down-counters*. An n-bit ripple counter that cycles through the 2n states is known as a divide-by-2^n counter or as a modulo-2^n binary counter.

The present unit is an asynchronous binary, modulo-8, ripple counter; asynchronous because all flip-flops do not change at the same time; binary because it follows the binary number sequence with bit values 2^0, 2^1, and 2^2; modulo-8 because it counts through 8 distinct states; ripple because the changes in state ripple through the stages.

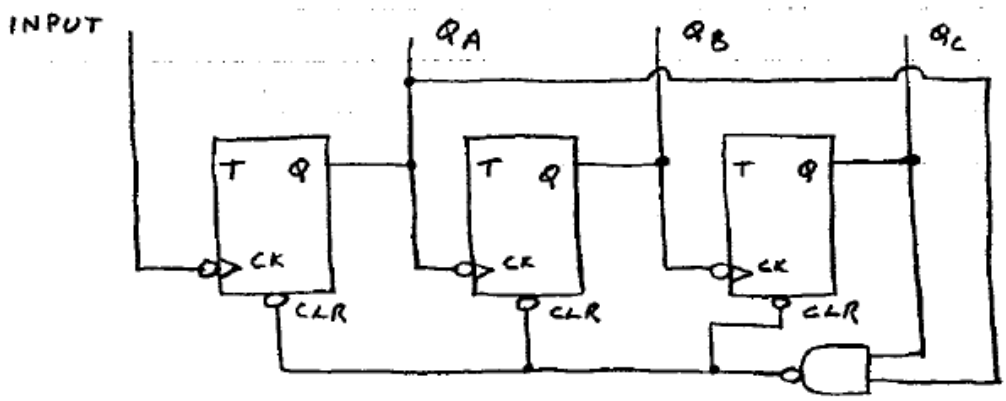
Example

Design a modulo-5 binary counter using T flip-flops with CLEAR capability.

Three stages are required to counter beyond 4. A modulo-5 counter must counter up to 4 and then, on the fifth pulse, clear all flip-flops to 0. The truth table is shown below:

count	Q_C	Q_B	Q_A	
0	0	0	0	
1	0	0	1	
2	0	1	0	
3	0	1	1	
4	1	0	0	
5	1	0	1	(unstable)
	0	0	0	(stable)

At the count of 5, the 1's at Q_A and Q_C can be NANDed. To generate a CLEAR signal as shown below:



Decade Counters

Communication with humans is more convenient in the decimal system. Flip-flops count in binary so binary numbers must be coded in decimal. To counter to 10 in the 8-4-2-1 code, four flip-flops are required. Ten distinct states can be obtained by modifying a 4-bit binary counter so that it skips the last six states. At counts in a normal manner from 0 to 9, then feedback logic resets the next count to zero.

Example

Design an 8-4-2-1 BCD ripple counter as follows:

A. Draw a block diagram of a 4-stage binary ripple counter using T flip-flops ABCD (T held HIGH).

B. Show the truth table of flip-flops for a decade counter that counts normally to decimal 9 and then resets to 0000. How

does it differ from the truth table for a binary counter in the tenth row?

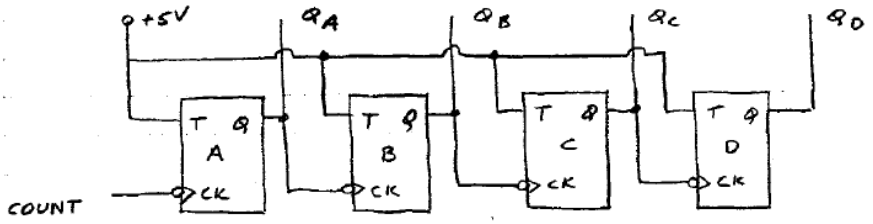
C. Modifying the circuit to accomplish the following: On the eighth count, the change in state of the D (MSB) flip-flop is to disable the input to the B flip-flop so it will not change or counter to 10. (see part (D)).

D. Modify the circuit so that on the tenth counter flip-flop D will be reset to 0 by the output of A flip-flop, without affecting the use of Q_C to toggle D.

E. Check the operation of the decade counter by drawing CK and flip-flop waveforms.

The results are as follows:

1. The block diagram is shown below:

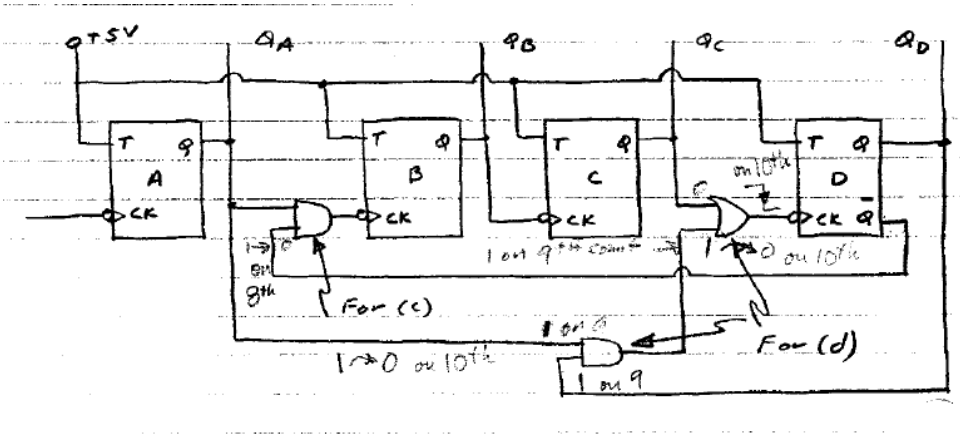


2. The truth table is shown below. Note that in the tenth row the counter is set to 0000, whereas the binary counter would have 1010 in that row.

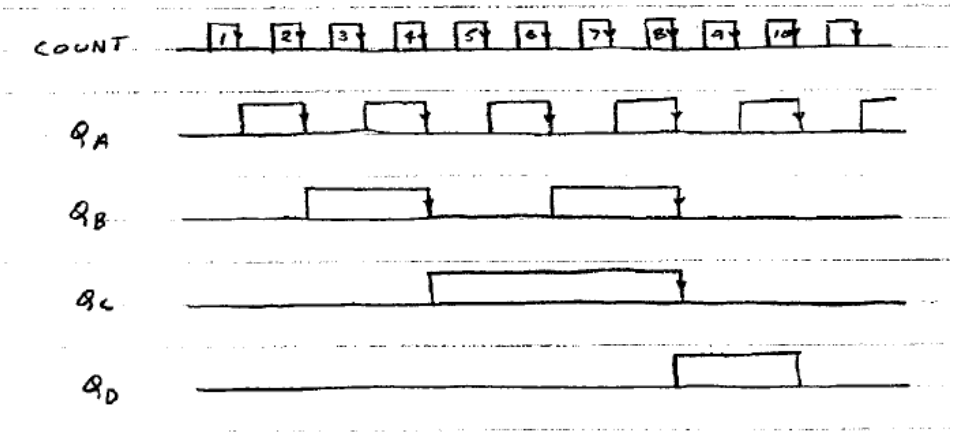
Count	Q_D	Q_C	Q_B	Q_A	State
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	0	0	0	0	0
	1	0	1	0	

3.

4.



5. The waveforms are shown below:

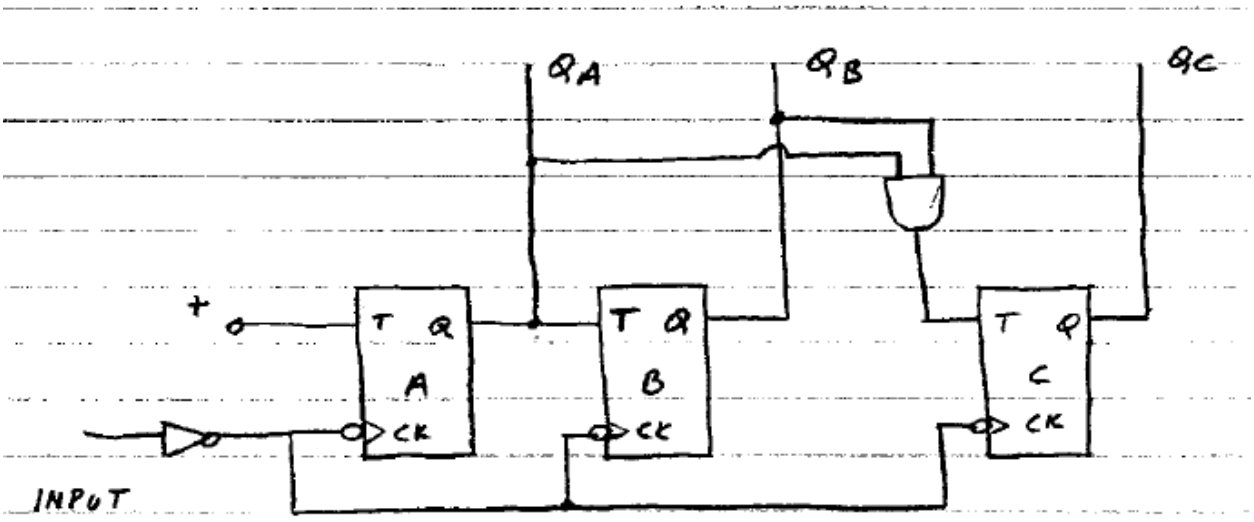


Synchronous Counters

One disadvantage of ripple counters is the slow speed of operation caused by the long time required for changes in state to ripple through the flip-flops. In addition these time delays can cause temporary state combinations (and voltage spikes called glitches) that result in false *synchronous* counters in which all flip-flops change state at the same instant.

In the synchronous counter shown below, T flip-flop A toggles, and the other flip-flops are clocked. Flip-flop B toggles on the next counter after Q_A becomes 1, as shown in the truth table. The AND gate causes flip-flop C to toggle on the next counter after Q_A and Q_B = 1 as called for in the truth table.

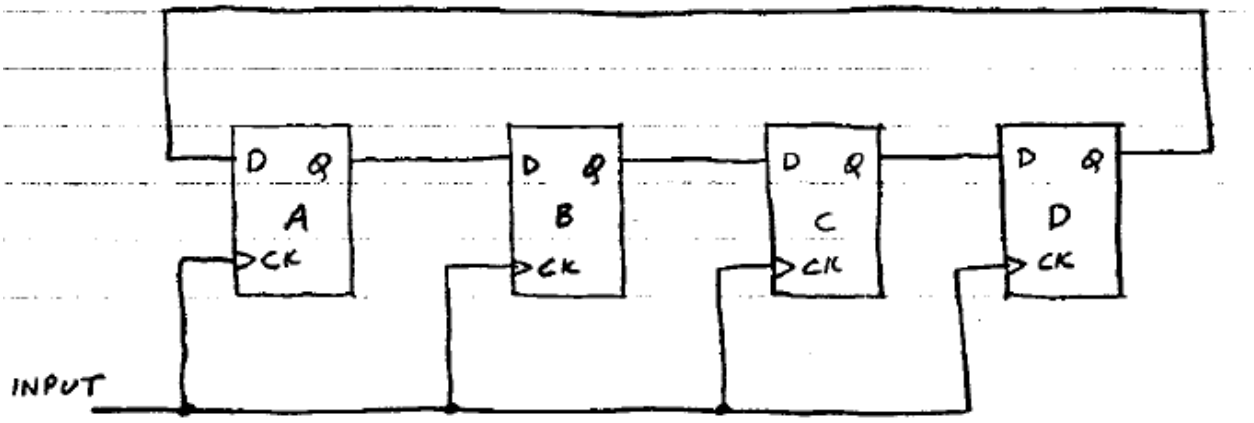
Count	Q_C	Q_B	Q_A
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
	0	0	0

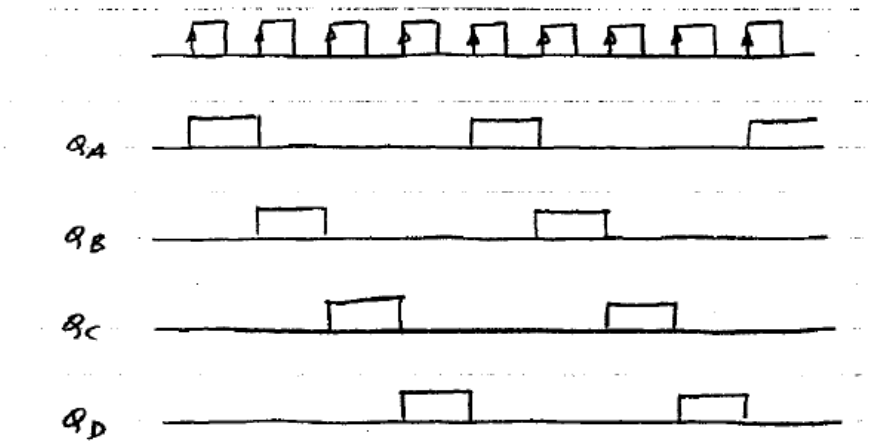


In general, synchronous counters are fast and trouble free.

Ring Counters

The figure below shows a 4-bit ring counter using D flip-flops. As in a synchronous counter, all flip-flops are triggered simultaneously; however, the output of each flip-flop drives only the adjacent flip-flops. In a ring counter a single pulse propagates through the ring, while all remaining flip-flops are at the zero-state.





The truth table is shown below:

Count	Q_D	Q_C	Q_B	Q_A
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0
4	0	0	0	1

A modulo-N ring counter requires N flip-flops and no other gates.

Counter Design Procedure

Counters are the simplest possible *finite-state machines*. The typically have only a single input instructing them to counter (after just the clock), and their outputs are just the current state.

A generalizing design process consists of the following four steps:

1. From the written specifications of the counter draw a state transition diagram that shows the counters desired sequence.

2. Design the state transition table from the state diagram, tabulating the current stat with the next state in the count sequence. Each state-bit is implemented by its own flip-flop.

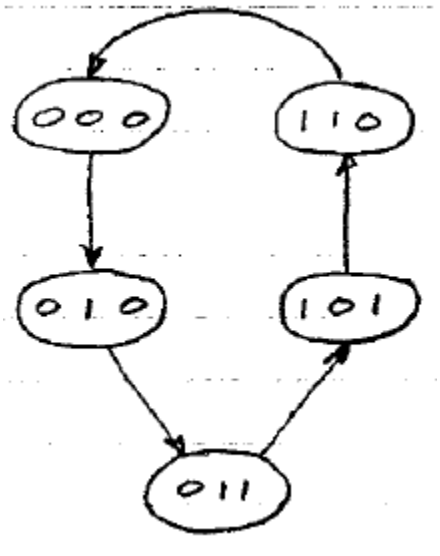
3. Express each next-state bit as a combinational logic function of the current state bits.

4. Choose a flip-flop for implementation of "remap" the next-state mapping (K-maps) determined in step 3 to obtain the desired behavior from the selected flip-flop.

Example: Generalized Counter Design

Design a 3-bit counter that advances through the sequence 000,010,011,101,110,000 and repeats. Not all the possible combinations of the 3 bits represent a valid state. The unused states 001,100, and 111, can be used as don't care conditions to simplify the logic.

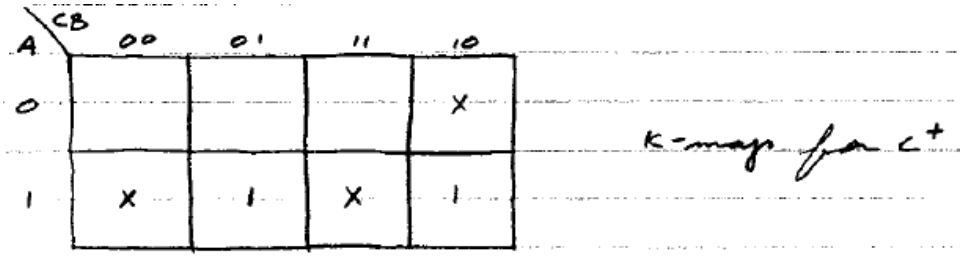
A. The state transition diagram is shown below:

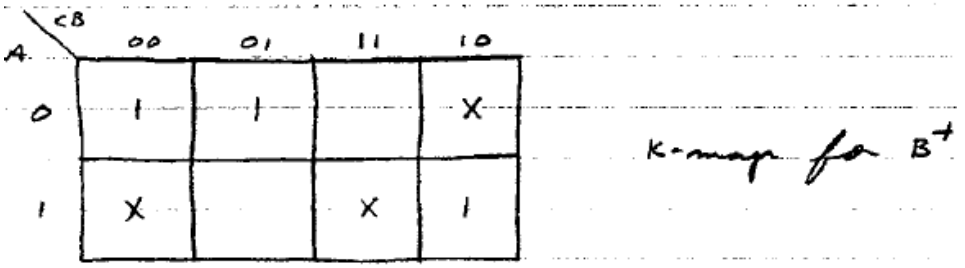


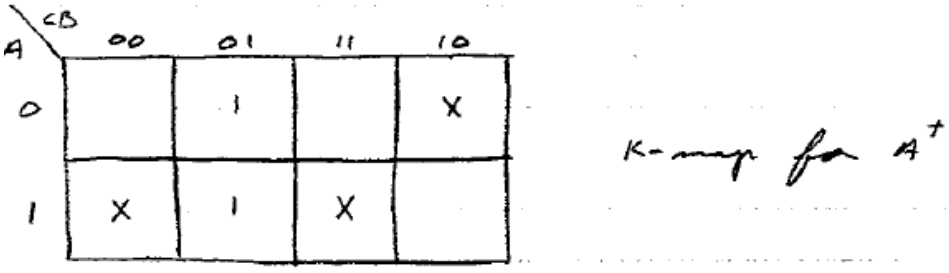
B. The state transition table is shown below. Note, the storage elements are named CBA.

Present state			Next state		
C	B	A	C^+	B^+	A^+
0	0	0	0	1	0
0	0	1	X	X	X
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	X	X	X
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	X	X	X

C. To express each next-state bit as a combinational logic function of the three current-state bits we draw the K-maps as shown below.







D. Since this is almost a straight binary sequence we will use a T flip-flops. The T flip-flops excitation table shown below will be used to derive new next state K-maps:

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

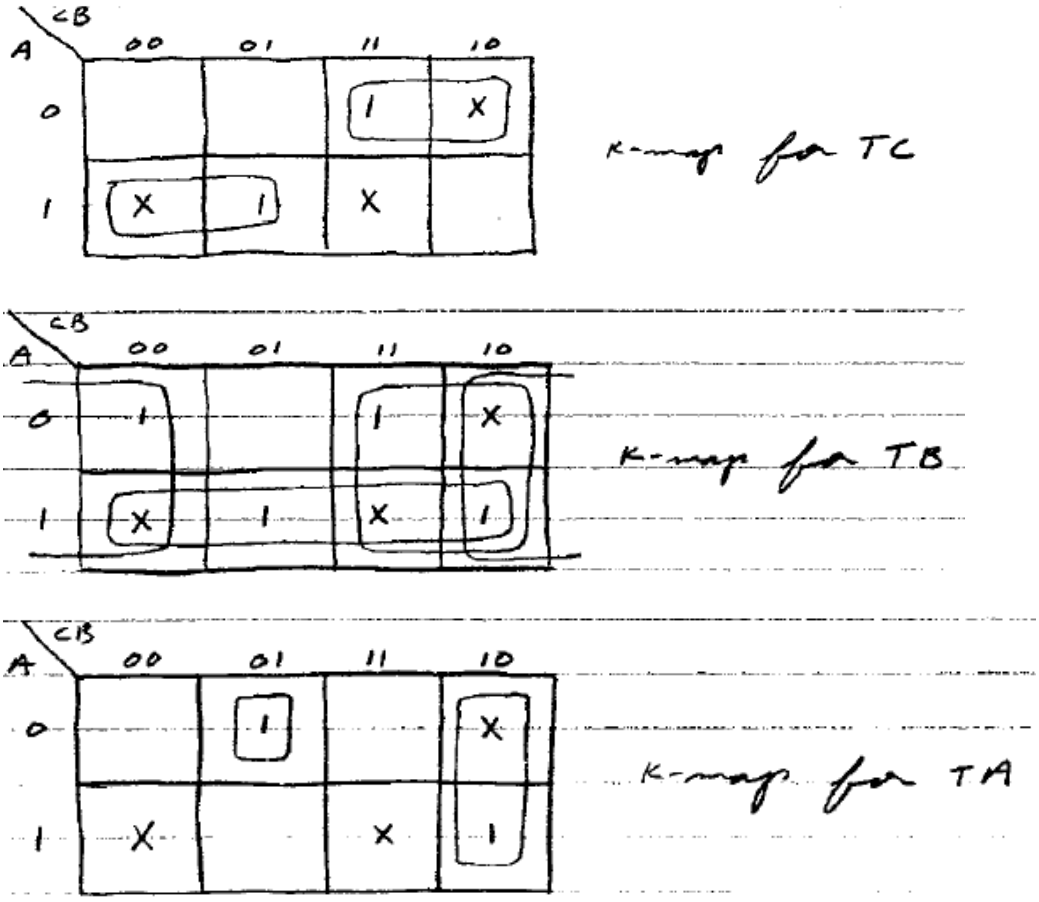
The figure below shows the toggle inputs needed to implement the desired state transitions:

Present State			Next State			T Inputs		
C	B	A	C^+	B^+	A^+	TC	TB	TA
0	0	0	0	1	0	0	1	0
0	0	1	X	X	X	X	X	X
0	1	0	0	1	1	0	0	1
0	1	1	1	0	1	1	1	0
1	0	0	X	X	X	X	X	X
1	0	1	1	1	0	0	1	1
1	1	0	0	0	0	1	1	0
1	1	1	X	X	X	X	X	X

For example, counter state 000 advances to 010, so the T inputs should be 0 (don't toggle) for C, 1 (toggle) for B, and 0 (don't toggle) for A. Similarly, state 110 returns to 000. In this

case, the control for C, B, A is toggle, toggle, don't toggle respectively, or 110.

The remapped K-maps for toggle implementation are given below:



Using the K-maps we obtain the minimalized functions:

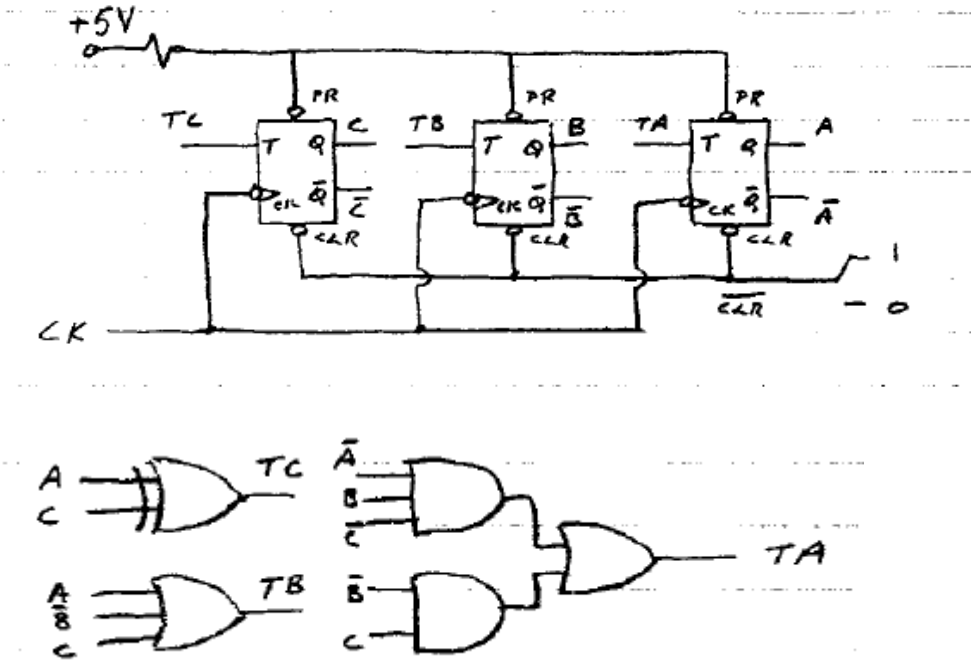
$$TC = \bar{A} \cdot C + A \cdot \bar{C} = A \oplus C$$

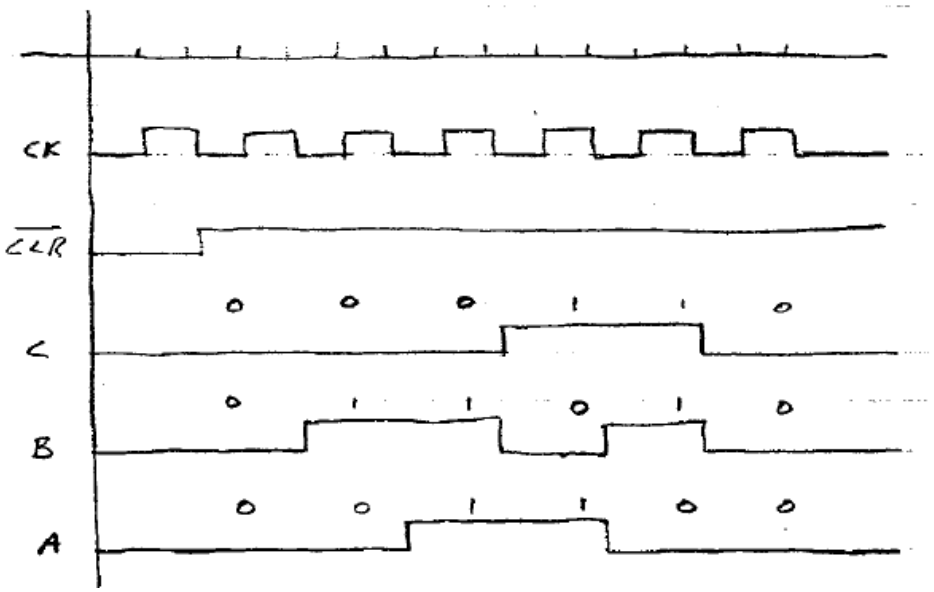
$$TB = A + \bar{B} + C$$

$$TA = \bar{A} \cdot B \cdot \bar{C} + \bar{B} \cdot C$$

The implementation is shown in the figure below. To reduce wiring complexity, the input and output networks are labeled rather than drawn as wires. Two networks with the same label are understood to be connected.

The timing waveform is also shown below. The proper sequencing through the states 000, 010, 011, 101, 110, 000 is clear from the waveform.





Self-Starting Counters

The counter should never be assumed to start in a particular state unless it is designed to do so. At power up the states are undefined; they could be 0 or 1 at random.

This leads to a problem for counters that do not use all state combinations of the storage elements. What happens if a counter enters one of the unused states at start-up depends on how don't cares have been mapped into 0's and 1's by the implementation procedure. The counter could sequence through the non-counter states and never enter the sequence it was designed for.

Verifying if a Counter is Self-Starting

A self-starting counter is one in which every possible state, even those not in the desired counter sequence, has a sequence of transitions that eventually leads to a valid counter state. Therefore, no matter how the counter starts up, it will eventually enter the proper counter sequence.

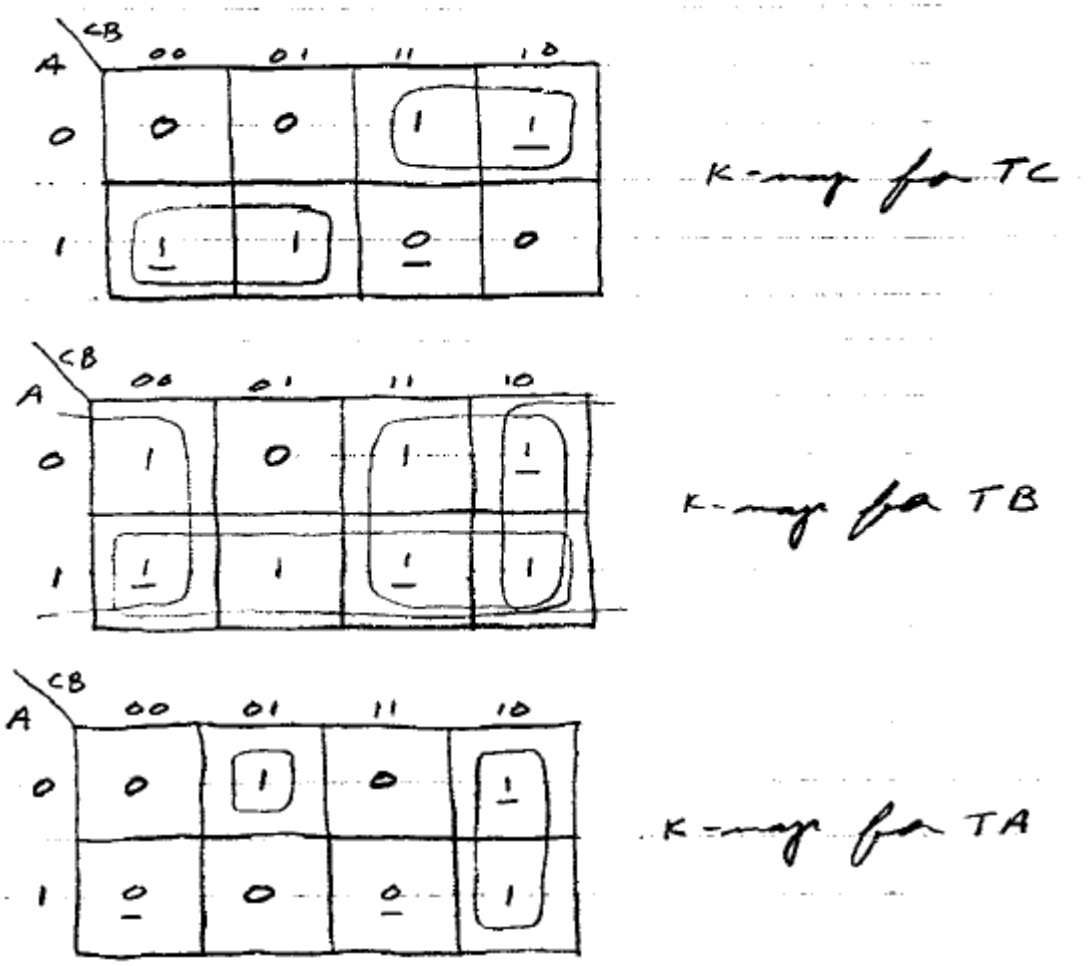
In general, it is desirable to enter the counter sequence is as few transitions as possible. However, it may be an advantage to depart from this rule if the sequence of noncounter states leads to reduced hardware.

A procedure to check if the counter is self-starting is illustrated in the following example.

Example

Analyze the solution of the last example to check whether or not it is self-starting.

First, replace the don't cares in the K-maps for the toggle implementation of the last example with the actually assigned 1's and 0's (they are underlined in the figure below):



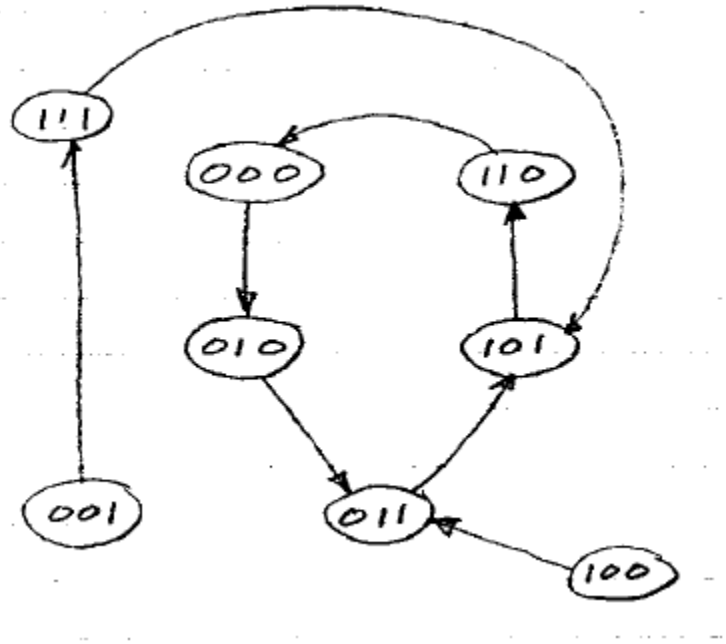
Since the K-maps represent the inputs to the toggle flip-flops they will be used to determine the flip-flops next state as shown in the diagram below. The next state is determined from the present state and the toggle inputs as required by the toggle excitation table.

Present state			Toggle Inputs			Next state		
C	B	A	TC	TB	TA	C^+	B^+	A^+
0	0	0	0	1	0	0	1	0
0	0	1	1	1	0	1	1	1
0	1	0	0	0	1	0	1	1
0	1	1	1	1	0	1	0	1
1	0	0	1	1	1	0	1	1
1	0	1	0	1	1	1	1	0
1	1	0	1	1	0	0	0	0
1	1	1	0	1	0	1	0	1

Toggle Excitation

Q	T	Q^+
0	0	0
0	1	1
1	1	0
1	0	1

The complete state transition diagram as obtained from the table is shown below. Note the counter is self-starting. It may however require two transitions before it is in the correct sequence.



Counter Reset

In the last example the particular starting state did not matter. It is more usual to have a fixed starting state for the counter or finite-state machine.

Flip-flops typically have preset and clear inputs. By use of those inputs any state can be chosen as the starting state.

Implementation with Different Kinds of Flip-Flops

Toggle flip-flops are a natural choice for implementing binary counters, but other flip-flop types may need less hardware for implementation.

We have shown how to implement the finite-state up-counter using toggle flip-flops. We will now implement this counter using RS, JK, and D flip-flops.

Example Implementation with RS Flip-Flops

Implement the five-state up-counter of the earlier example using RS flip-flops.

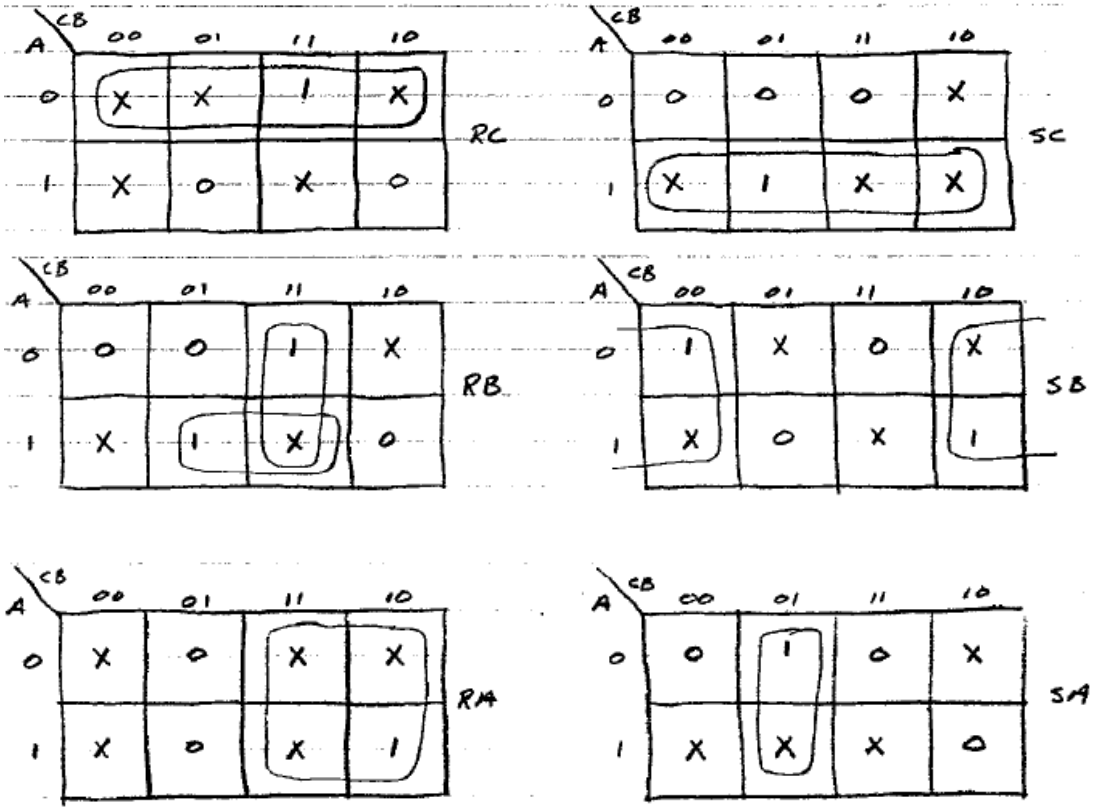
The first two steps of the counter design procedure – the state transition diagram, the state transition table, and the next-state K-map have already been performed. The next step starts with the RS flip-flop excitation table as shown below:

Q	Q^+	R	S
0	0	X	0
0	1	0	1
1	0	1	0
1	1	0	X

This table will be used to determine the RS inputs needed to implement the desired state transitions, as shown in the figure below.

Present state			Next State			RS Inputs					
C	B	A	C^+	B^+	A^+	RC	SC	RB	SB	RA	SA
0	0	0	0	1	0	X	0	0	1	X	0
0	0	1	X	X	X	X	X	X	X	X	X
0	1	0	0	1	1	X	0	0	X	0	1
0	1	1	1	0	1	0	1	1	0	0	X
1	0	0	X	X	X	X	X	X	X	X	X
1	0	1	1	1	0	0	X	0	1	1	0
1	1	0	0	0	0	1	0	1	0	X	0
1	1	1	X	X	X	X	X	X	X	X	X

The re-mapped K-maps for the RS implementation are given below.



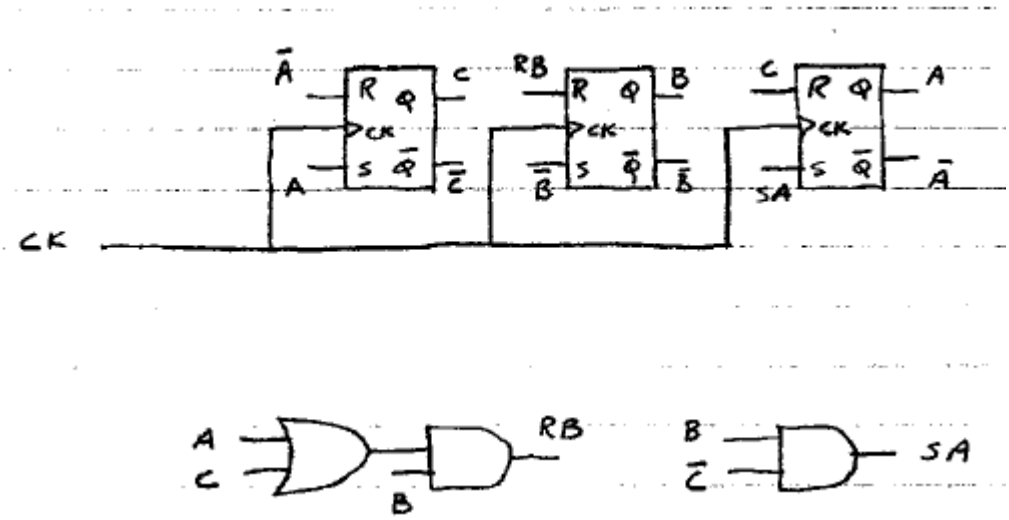
Using the K-maps we obtain the minimized functions:

$$R_C = \bar{A}, \qquad S_C = A$$

$$R_B = A \cdot B + B \cdot C = B \cdot (A + C), \qquad S_B = \bar{B}$$

$$R_A = C, \qquad S_A = B \cdot \bar{C}$$

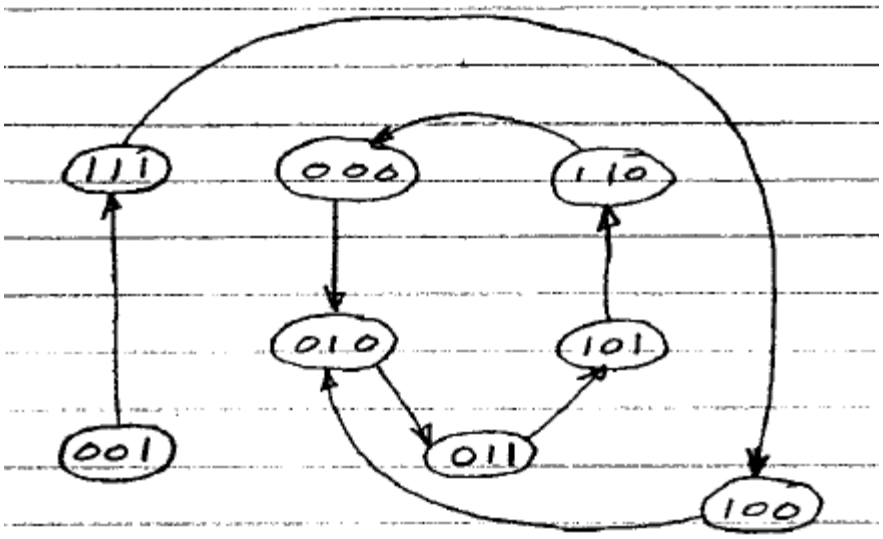
The implementation is shown in the diagram below. The figure doesn't show the reset logic.



To check if the system is self-starting we replace the don't care in the K maps with the actually assigned 1's and 0's (see table below):

Present State			Next State			RS Inputs					
C	B	A	C^+	B^+	A^+	RC	SC	RB	SB	RA	SA
0	0	0	0	1	0	1	0	0	1	0	0
0	0	1	1	1	1	0	1	0	1	0	0
0	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	0	1	0	1	1	0	0	1
1	0	0	0	1	0	1	0	0	1	1	0
1	0	1	1	1	0	0	1	0	1	1	0
1	1	0	0	0	0	1	0	1	0	1	0
1	1	1	1	0	0	0	1	1	0	1	0

The complete state transition diagram as obtained from the table is shown below. Note that the counter is self-starting. It may take three transitions before reaching the correct sequence.



Example Implementation with JK Flip-Flops

Implement the five-state up-counter of the earlier example using JK flip-flops.

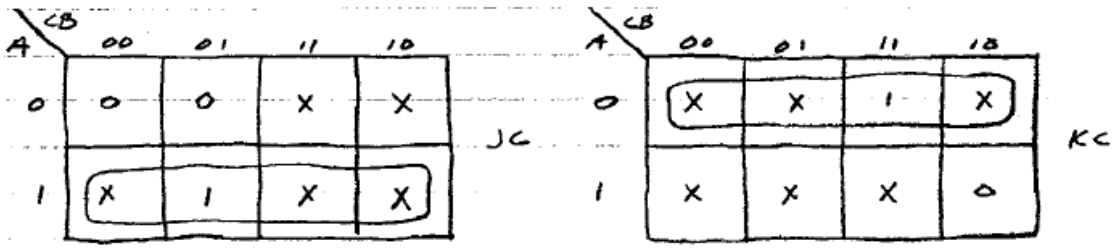
As in the last example the first three steps of the design procedure have been performed. The last step starts with the JK flip-flops excitation table as shown below.

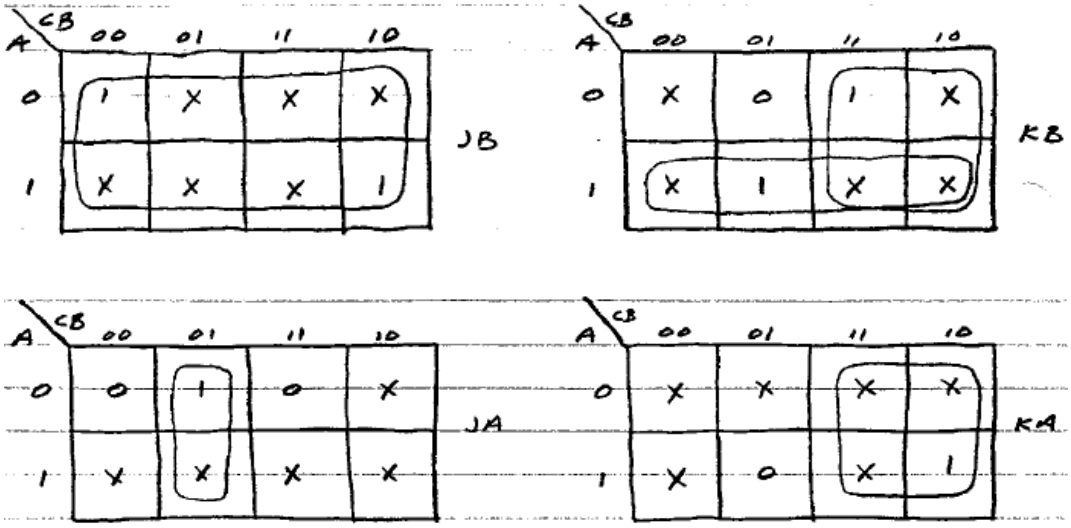
Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

From this table the JK inputs needed to implement the desired state transition will be determined (see table below).

Present state			Next state			JK Inputs					
C	B	A	C^+	B^+	A^+	JC	KC	JB	KB	JA	KA
0	0	0	0	1	0	0	X	1	X	0	X
0	0	1	X	X	X	X	X	X	X	X	X
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	1	1	X	X	1	X	0
1	0	0	X	X	X	X	X	X	X	X	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X
1	1	1	X	X	X	X	X	X	X	X	X

The remapped K-maps for the JK implementation are given below.





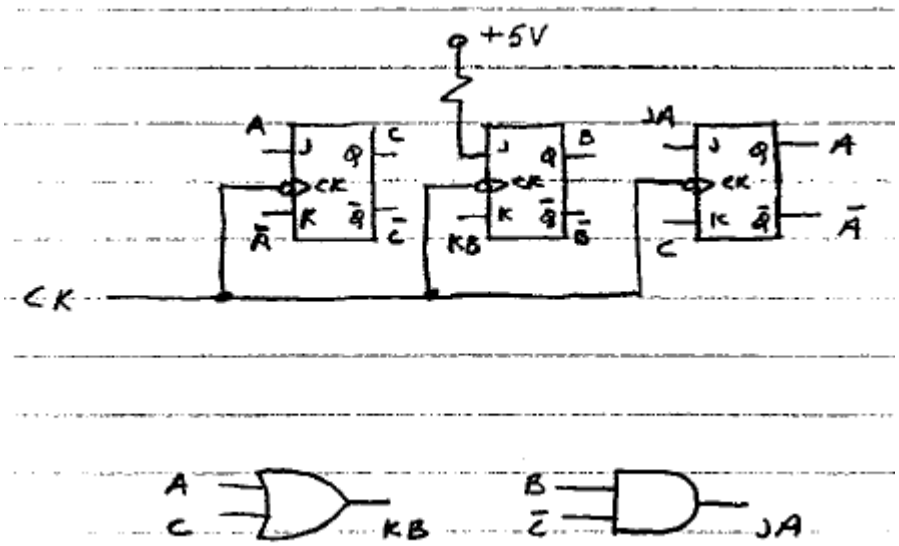
Using the K-maps we obtain the minimized functions:

$JC = A$, $KC = \bar{A}$

$JB = 1$, $KB = A + C$

$JA = B \cdot \bar{C}$, $KA = C$

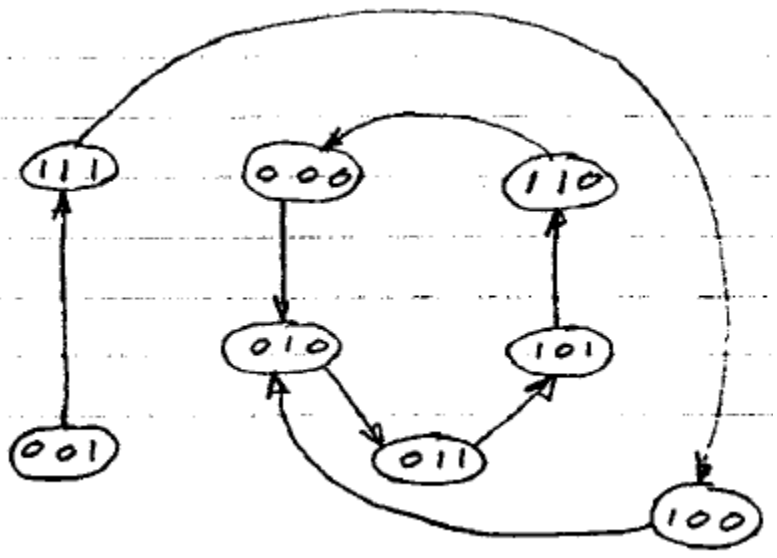
The implementation is shown in the diagram below. Again the reset logic is not shown.



To check if the system is self-starting we replace the don't cares in the K-maps with the actually assigned 1's and 0's (see table below).

Present state			Next state			JK Inputs					
C	B	A	C^+	B^+	A^+	JC	KC	JB	KB	JA	KA
0	0	0	0	1	0	0	1	1	0	0	0
0	0	1	1	1	1	1	0	1	1	0	0
0	1	0	0	1	1	0	1	1	0	1	0
0	1	1	1	0	1	1	0	1	1	1	0
1	0	0	0	1	0	0	1	1	1	0	1
1	0	1	1	1	0	1	0	1	1	0	1
1	1	0	0	0	0	0	1	1	1	0	1
1	1	1	1	0	0	1	0	1	1	0	1

The complete state transition diagram as obtained from the table is shown below. Note that the counter is self-starting. If may take three transitions before reaching the correct sequence.



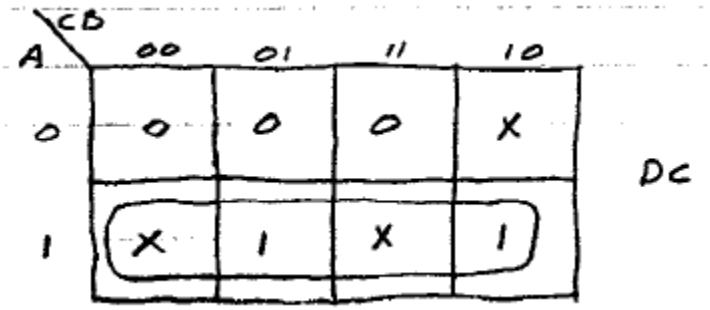
Example Implementation with D Flip-Flops

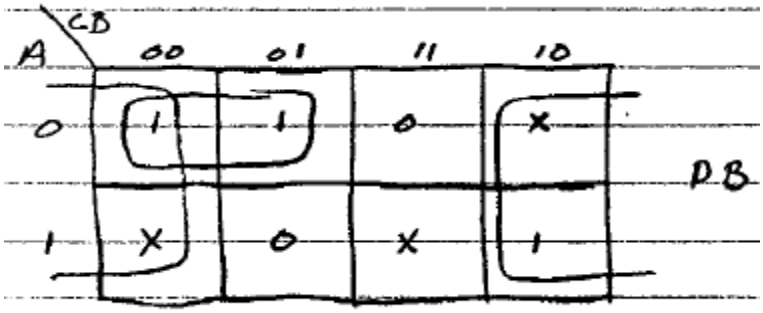
Implement the five-state up-counter of the earlier example using D flip-flops.

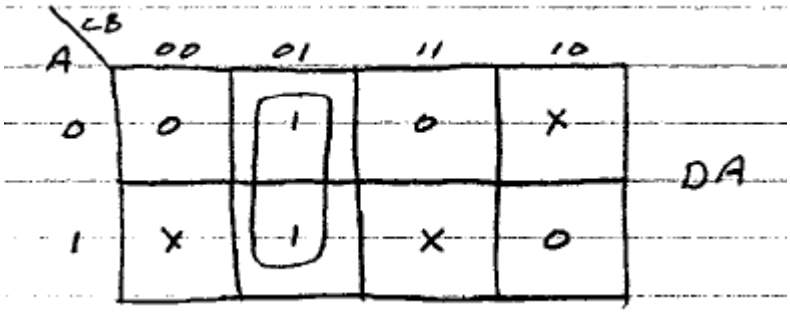
Again as in the last example the first three steps of the design procedure have been performed. From the excitation table for the D flip-flops shown below it is seen that the D inputs are identical to the next-state outputs.

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

These are already tabled in the state transition table. We place the next state outputs into K-maps and find the minimized functions. The K-maps are identical to those obtained earlier in the original example.







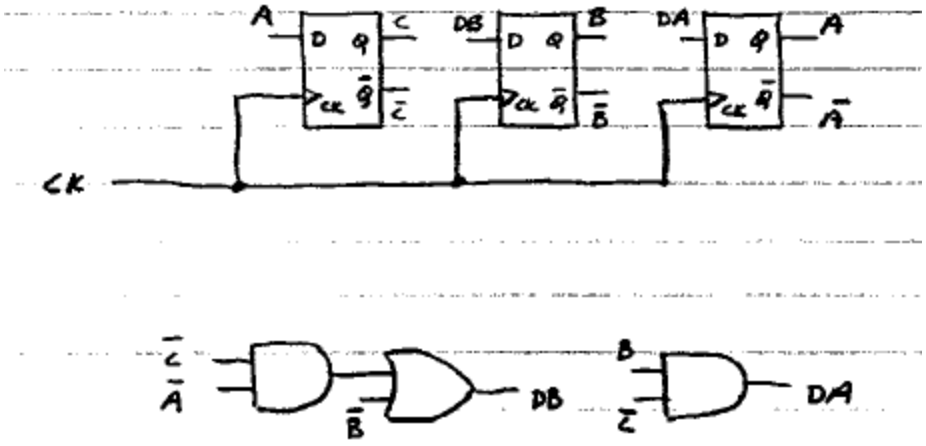
Using the K-maps we obtain the minimized functions:

$$DC = A$$

$$DB = \bar{A} \cdot \bar{C} + \bar{B}$$

$$DA = B \cdot \bar{C}$$

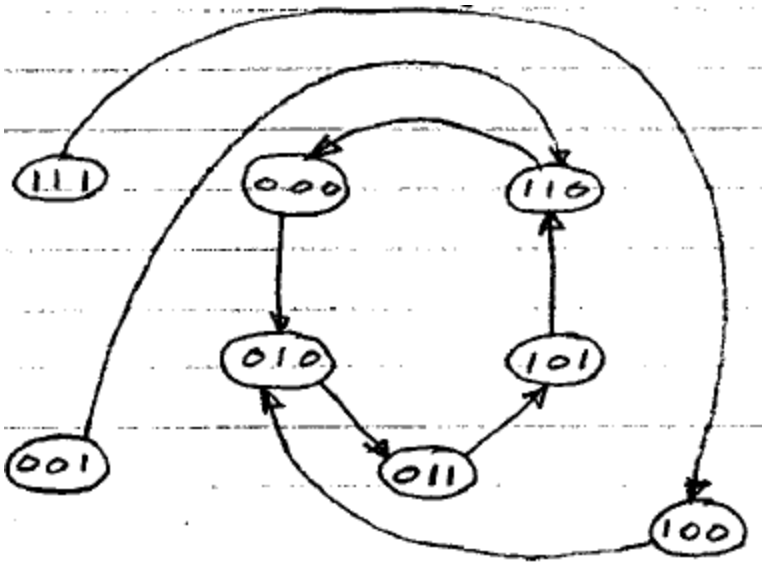
The implementation is shown in the diagram below. The reset logic is not shown.



To check if the system is self-starting we replace the don't cares in the K-maps with the actually assigned 1's and 0's:

C	B	A	C^+	B^+	A^+	DC	DB	DA
0	0	0	0	1	0	0	1	0
0	0	1	1	1	0	1	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	1	0	1
1	0	0	0	1	0	0	1	0
1	0	1	1	1	0	1	1	0
1	1	0	0	1	0	0	0	0
1	1	1	1	0	0	1	0	0

The complete state transition diagram as obtained from the table is shown below. Note that the counter is self-starting. It may take two transitions before reaching the correct sequence.



Comparison & Summary of Different Implementations

The same state diagram led to the different implementation costs:

T Flip-Flops	5	10	15
RS Flip-Flops	3	5	12
JK Flip-Flops	2	4	10
D Flip Flips	3	5	9

In general JK flip-flops usually result in the most gate and literal efficient implementations. Since the RS flip-flops behavior is a subset of the JK, there is no advantage in using RS devices.

T flip-flops are sorted for implementing straight-forward binary counters, but their advantage is lost when the counter follows a sequence is not in direct binary order. In the example considered the T flip-flop was the worst.

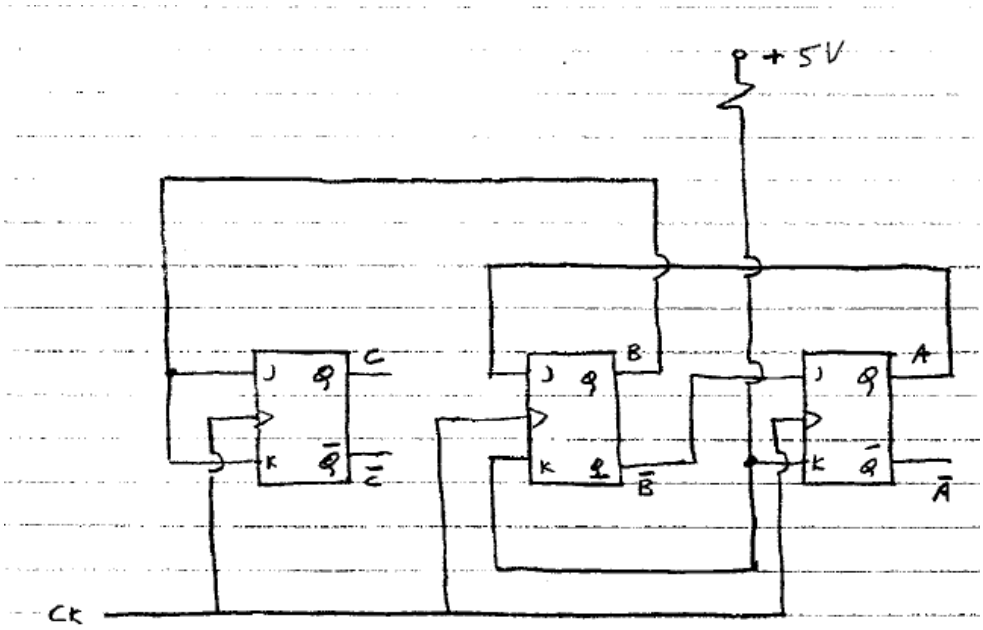
D flip-flops, although not the most gate efficient have same advantages. First, they simplify the design procedure where the next-state remapping steps can be skipped. Second, the wiring is not as complex. Wiring complexity is especially important when using programmable logic technologies. Finally, D storage elements are transistor efficient in MOS VLSI technologies.

In summary, for conventional packaged MSI/SSI TTL designs, JK flip-flops are usually preferred, especially when the design criterion is minimum gate and literal count. D-type devices are preferred when designing with programmable logic or in more highly integrated technologies than TTL, where minimum wire count or simplified design procedure is the objective.

The technique we have been discussing can be used in a reverse order to obtain the state transition table and diagram when the final circuit implementation is given.

Example

Find the state transition table and counter state diagram for the implementation given below.



From the above diagram we see that:

$J_C = B \qquad K_C = B$

$J_B = A \qquad K_B = 1$

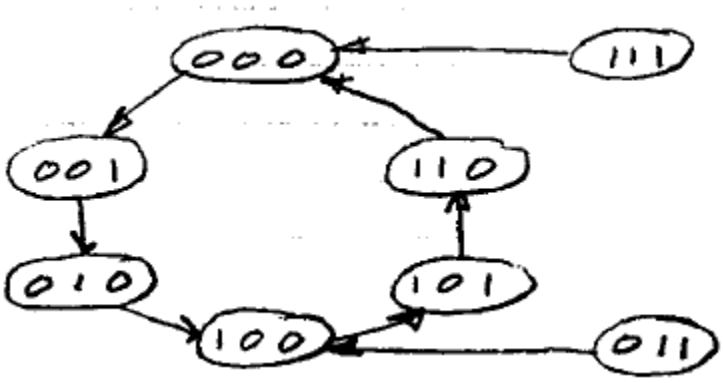
$J_A = \bar{B} \qquad K_A = 1$

We can now use the above expression along with the excitation table for the JK flip-flop to obtain the state transition table.

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

C	B	A	JC	KC	JB	KB	JA	KA	C^+	B^+	A^+
0	0	0	0	0	0	1	1	1	0	0	1
0	0	1	0	0	1	1	1	1	0	1	0
0	1	0	1	1	0	1	0	1	1	0	0
0	1	1	1	1	1	1	0	1	1	0	0
1	0	0	0	0	0	1	1	1	1	0	1
1	0	1	0	0	1	1	1	1	1	1	0
1	1	0	1	1	0	1	0	1	0	0	0
1	1	1	1	1	1	1	0	1	0	0	0

The state transition diagram is given below:



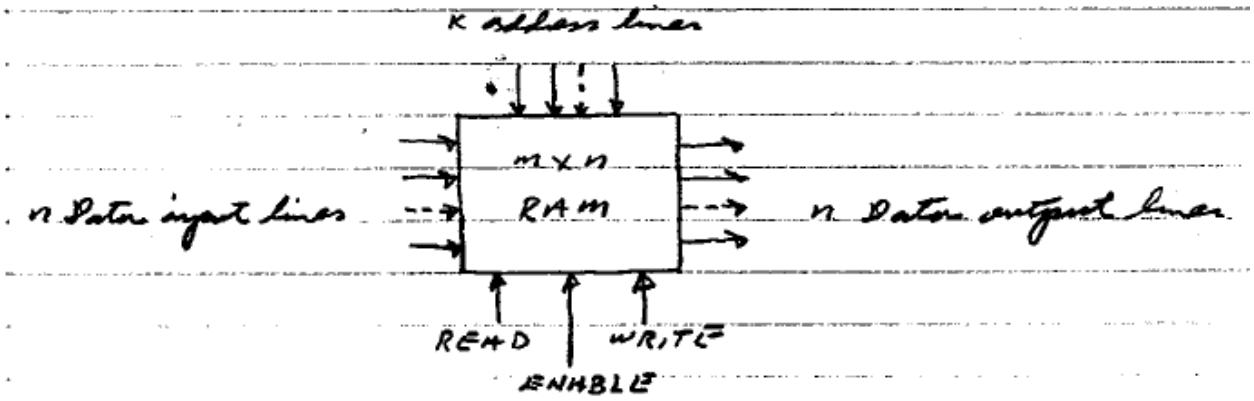
From the above diagram we see that two of the eight counts are not part of the counter sequence. However, since 111 -> 000 and 011 -> 100 the counter is self-starting.

Memory

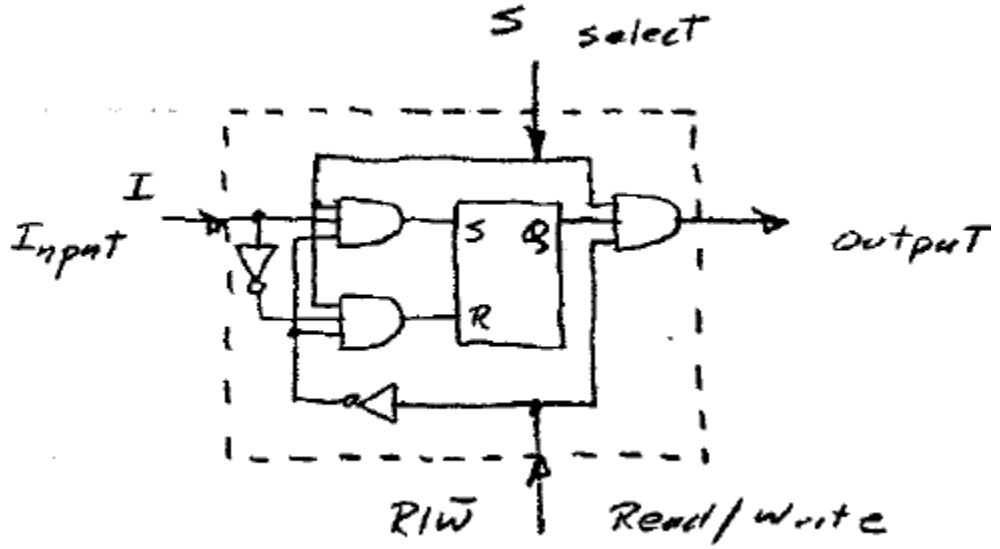
In digital computers instructions and numbers are stored in the *memory*. The memory is an organized arrangement of elements called *memory cells* each of which can store ("write") data at any selected location ("address") and retrieve ("read") the data at any later time. In Read-Only Memory (ROM), data we initially and permanently stored by the manufacture or the user. The computer can read the data at any address but cannot alter the stored data.

RAM

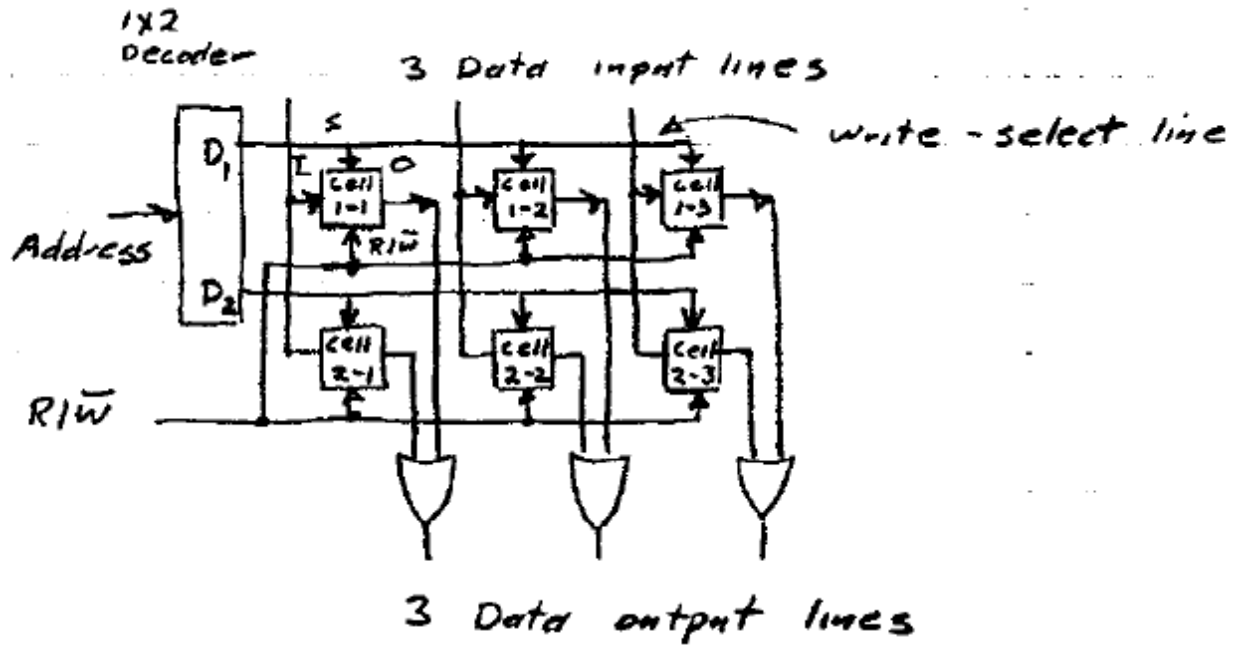
In the Read and Write Memory shown below, the k address lines can designed 2^k = m words whose n bits are carried on the n parallel input and output lines.



The memory includes an m x n matrix of memory cells. Each cell consists of a binary storage element and the associated control logic. In the simple cell shown below, the cell is selected when S goes HIGH. With $R/\overline{W}$ HIGH, the output is enabled and the value of Q is read out on the output data line. With $R/\overline{W}$ LOW, the input is enabled to units in the input data value.



An example of an elementary RAM with a capacitor of two 3-bit words is shown below:



The address causes the decoder to activate one of the two word-select lines. In the WRITE operation ($R/\overline{W}$ LOW), 3 bits of data are transformed from the input lines to the selected word. In the READ operation the 3 bits of the selected word

are transferred to the output lines with the OR gates. The outputs of the unselected words are all LOW.

Because the words in memory can be accessed in any order, we have a *Random-Access Memory* (RAM). In the 2x3 RAM above, address selection is *linear*, since one word-select line is activated. In large memories, selection is *coincident since* each cell is accessed by addressing an X select line to select the row and a Y select line to select the column. The intersection of the X and Y lines gives one cell in a two-dimensional matrix.

There are two types of RAM, *static* and *dynamic*. SRAM retains its data as long as power is applied without any further action from the computer. Each cell of a static RAM is a flip-flop. DRAM requires continuous actions from the computer to maintain its contents. Each cell in a dynamic RAM is a capacitor, which leaks charge and therefor requires continuous recharging to maintain its value. SRAM is used in microprocessor based systems that require small memory; DRAM is used in large memory systems because of lower cost and greater density.

MOS devices are widely used because of their high packing density and low power consumption. Bipolar RAM's are very fast but are less compact and less energy efficient than MOS RAM's. They are used as a "scratch-pad" memory for data being processed.

Example

An array of eight memory cells is arrange in two rows and four columns. Design he addressing system consisting of a row decoder and a column decoder. Assume a row or column is selected when driven HIGH (logic 1).

A) Row 0 and 1 are selected by row-select (RS) values 0 and 1 and columns 0 to 3 are selected similarly. Compare the truth tables for RS and for CS_1 and CS_2.

B) Show the eight memory cells (lettered A through D and E through H). Design the decoders using AND and NOT gates only and show the two decoder circuits on your diagram.

C) Specify the address values that will select cell F.

D) How many row-select and column-select lines would be required to address 1024 cells arranged in 16 rows and 64 columns.

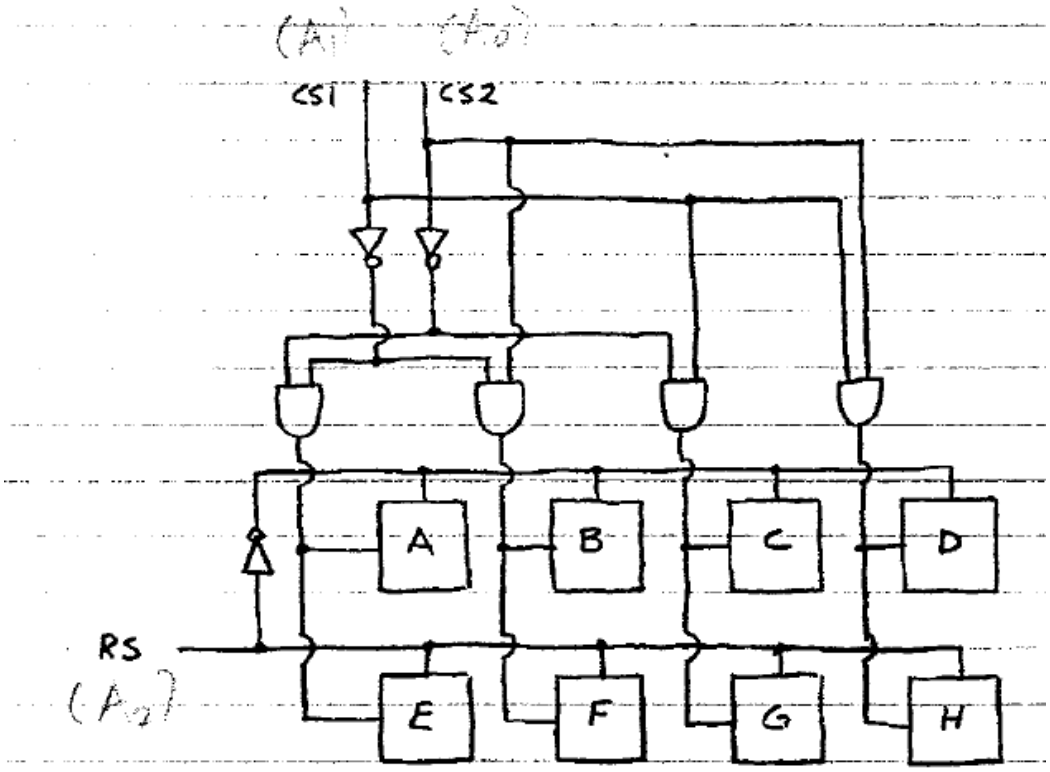
Answers:

A) The truth tables are shown below:

RS	Row 0	Row 1
0	1	0
1	0	1

CS1	CS2	Col 0	Col 1	Col 2	Col 3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

B)



C) Cell F is selected when:

RS = 1; CS_1 = 0; CS_2 = 1

D) The number of rows/columns is given by:

Number of rows/columns = 2^k

Where k = number of rows/columns select lines

So 16 rows = 2^4, which means 4 row select lines are required.

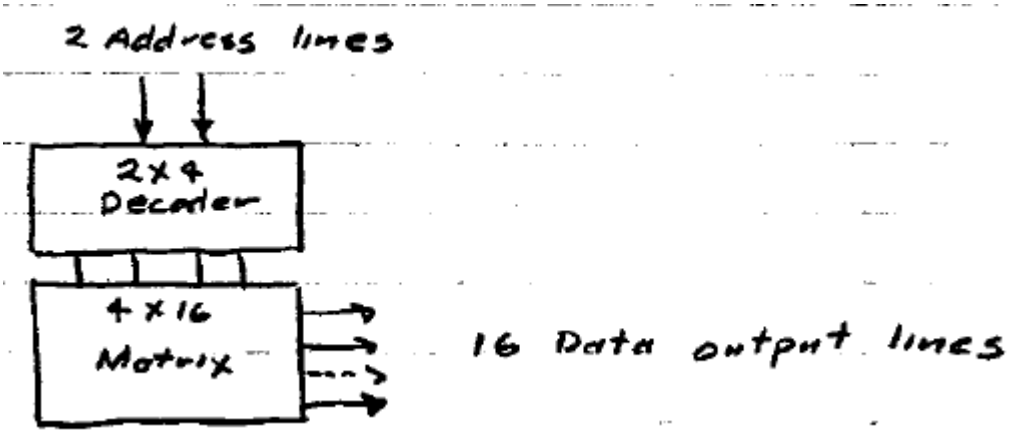
So 64 columns = 2^6, which means 6 column select lines are required.

ROM

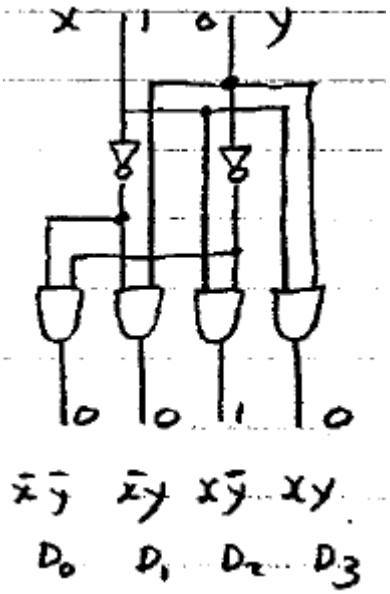
In Read-Only Memory (ROM), binary data is physically and permanently stored by deforming the state of the memory cells. A set of input signals on the address

lines is decoded to access a given set of cells whose states then appear on the output lines.

A typical 4x16 ROM is shown below:



The 64 bits of the 4x16 ROM are stored in 64 memory cells arranged in $2^k = 2^2 = 4$ words of 16 bits each available at the 16 output lines. The address is coded as a K-bit binary number; a decoder translates the coded address and specifies one of the 2^k words. On the 2x4 decoder shown below, the address xy = 10 yields a 1 at D_2 ($x\bar{y}$) and specifies that word 2 is to be read, that is, connected to the 16 output lines.



The function performed can be described as follows: (1) If the input is an address and the output is a word (data or instructions) stored, we have a “memory”. (2) If the input is data coded in one form and the output is the same data coded in another form we have a “code converter”. (3) If the input is a set of binary variables (a binary function) and the output is a related binary function we have a “combinational logic circuit” that can replace a network of logic gates.

There are three types of read only memory; mask-programmable (ROM), programmable (PROM), and erasable programmable (EPROM). The masked-programmed ROM’s are programmed during manufacturing. A PROM device initially contains all 0’s; the user programs the unit by electrically changing appropriate 0s and 1s. This is an irreversible process. EPROMs can be programmed and erased repeatedly. EPROMs can be erased by shining and ultraviolet light into a window at the top of the device.

The same bipolar and MOS technologies are used in IC ROM as in RAM. In general ROM is simpler since fewer control elements are necessary and no provision is made for changing cell states.

Example

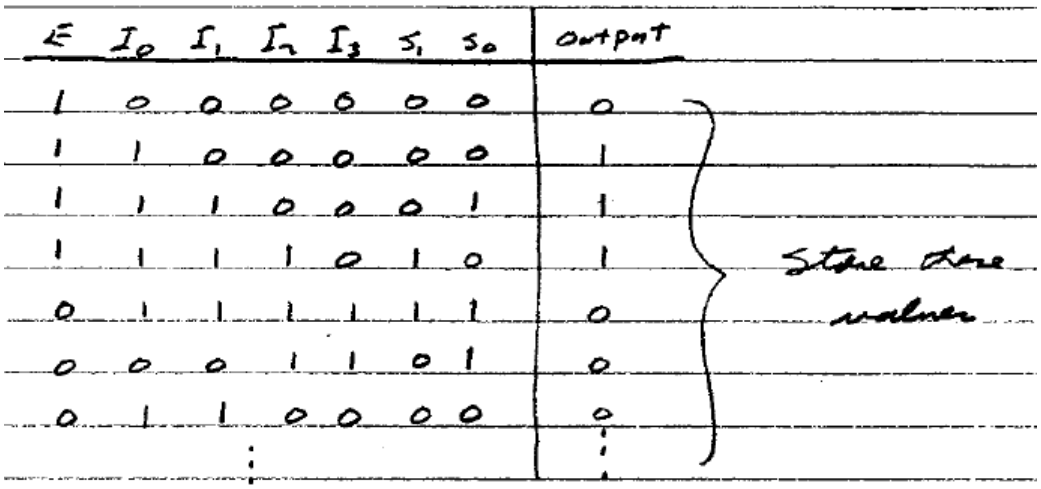
Show how a ROM can be used to realize:

A) The multiplexer discussed earlier, 4:1 MUX

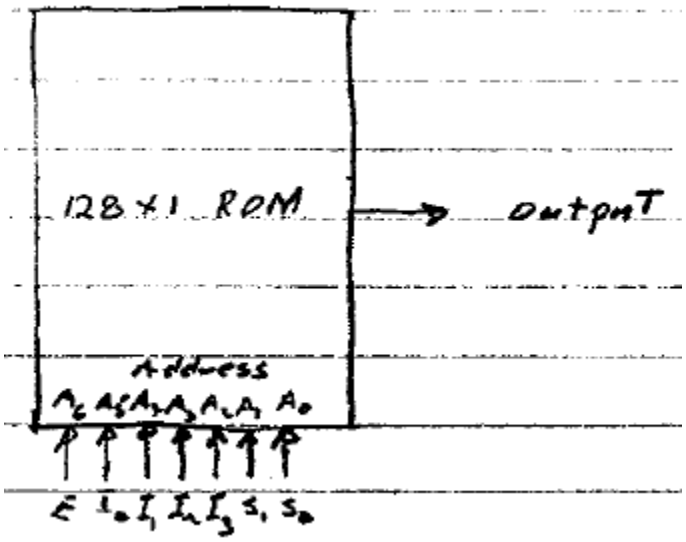
B) The decoder discussed earlier, 2:4 DEC

Answers:

A) For each combination of values at the E, I_0, E_1, I_2, I_3, S_1, and S_0 inputs, the output is to be 0 or 1. If each input combination is considered as an “address”, a ROM can store all possible outputs. For these 7 inputs, a 2^7 = 128x1 ROM is needed. A few representative lines of the truth table are given below.



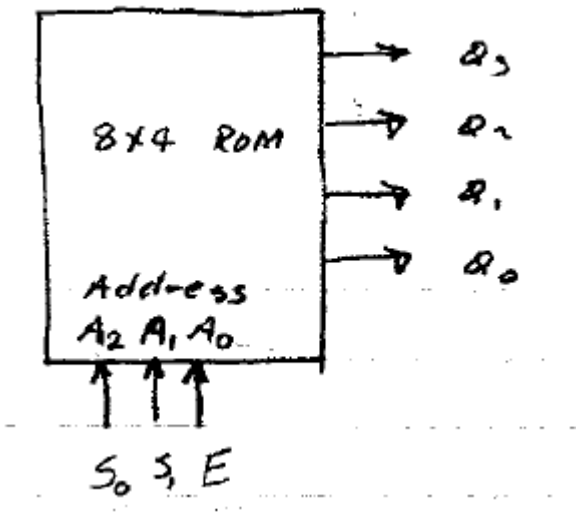
Connect E, I_0, E_1, I_2, I_3, S_1, and S_0 to the address inputs as shown below.



B) For this case 32 bits of ROM (8x4) are needed. Connect E, S_1 and S_0 to the address lines, and the 4 output lines to the destination (Q_0, Q_1, Q_2, Q_3). The outputs will all be 0 whenever E=0. The truth table for the decoder as given before is reproduced here:

S_0	S_1	E	Q_3	Q_2	Q_1	Q_0	
0	0	0	0	0	0	0	Store these values
0	0	1	0	0	0	1	
0	1	0	0	0	0	0	
0	1	1	0	0	1	0	
1	0	0	0	0	0	0	
1	0	1	0	1	0	0	
1	1	0	0	0	0	0	
1	1	1	1	0	0	0	

Connect E, S_1 and S_0 to the address lines as shown below:



Finite State Machines

Finite state machine are so named because the sequential logic that implements them can be in only a finite number of possible states. The counters discussed earlier are simple finite state machines. Their outputs and states are identified, and there is no choice of the sequence of counting.

In the more generalized case, the outputs and next state of the finite state machine are combinational logic functions of their inputs and present state. Finite state machines are essential for realizing the control and decision-making logic in digital systems.

In designing finite state machines a rigorous *synchronous design methodology* will be followed. This means that the state changes will be toggled with a global reference signal, the clock. *State time* is defined as the time between related *clocking events*.

Finite State Machine Design Procedure

A general design procedure for arbitrary finite state machines is given below.

1) Understand the problem. A finite state machine is often described by a written specification of its behavior. Some input sequences should be tried to help understand the conditions under which the various outputs are generated.
   Outcome: Descriptive block diagram

2) Obtain an abstract representation of the FSM. Put the problem in a form that is easy to manipulate by known procedures (e.g.: draw a state diagram).
   Outcome: Initial State Diagram

3) Perform state minimization. The abstract representation after has too many states. Some states may be eliminated to simplify the problem.
   Outcome: Simplified State Diagram & Symbolic State Transition Table

4) Perform state assignment. Outputs are derived from present and past states and a good choice of how to encode the state after leads to a simple implementation.
   Outcome: Encoded State & Transition Table

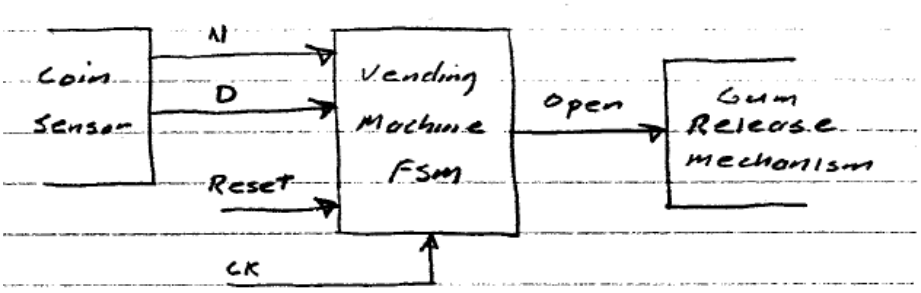
5) Choose flip-flop types for implementing the FSM's state. JK flip-flops tend to reduce gate count but have more connections. D flip-flops simplify the connection process.
6) Implement the finite state machine. Using Boolean equations or K-maps for the next state and output combinational functions produce the minimized two-level or multilevel implementations.
   Outcome: Remapped state transition Table (using excitation table of flip-flops), K-maps for flip-flop inputs & FSM output, Circuit Synthesis

Example: A Simple Vending Machine

Implement a simple finite state machine that controls a vending machine.

The control works as follows: The vending machine delivers a package of gum after it has received 15 cents in coins. The machine has a single coin slot that accepts nickels and dimes, one coin at a time. A mechanical sensor indicates to the control whether a dime or a nickel has been inserted into the coin slot. The controller's output causes a single package of gum to be released down a chute to the machine is to be designed so it does not give change. A customer who uses two dimes losses 5 cents.

1) Draw a block diagram to understand the inputs and outputs. In the figure below N is asserted for one clock period when a nickel is inserted into the coin slot:

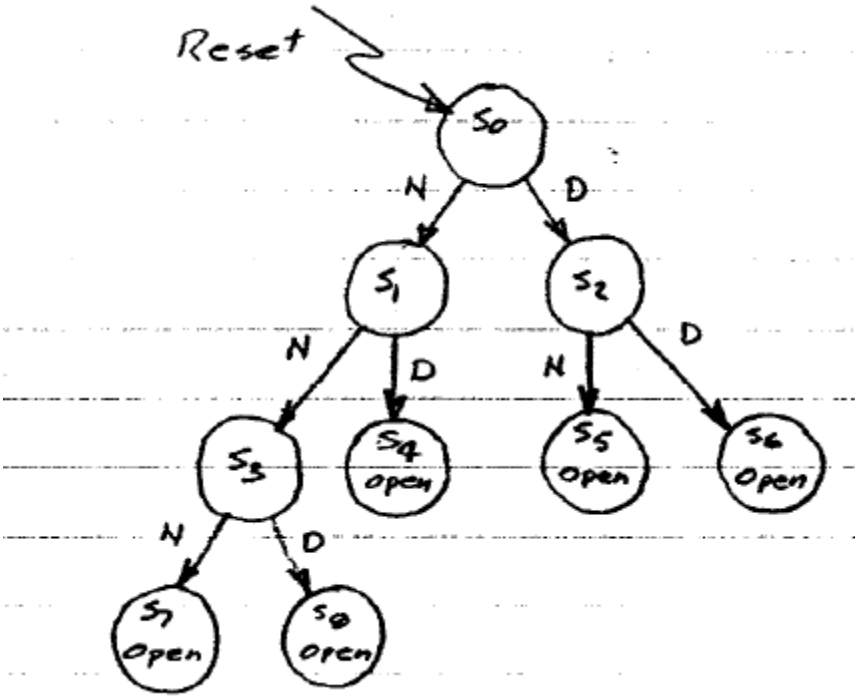


D is asserted when a dime has been deposited. The machine asserts Open for one clock period when 15 cents (or more) has been deposited since the last reset. It will be assumed that the coin sensor returns any coin not recognized, leaving N and D unasserted, and that extreme logic resets the machine after the gum is delivered.

2) A more suitable abstract representation is obtained by enumerating the possible unique sequences of inputs or configurations of the system. For this case gum is released for the input sequences:

    - Three nickels in sequence: N, N, N
    - Two nickels followed by a dime: N, N, D
    - A nickeled followed by a dime: N, D
    - A dime followed by a nickel: D, N
    - Two dimes in sequence: D, D

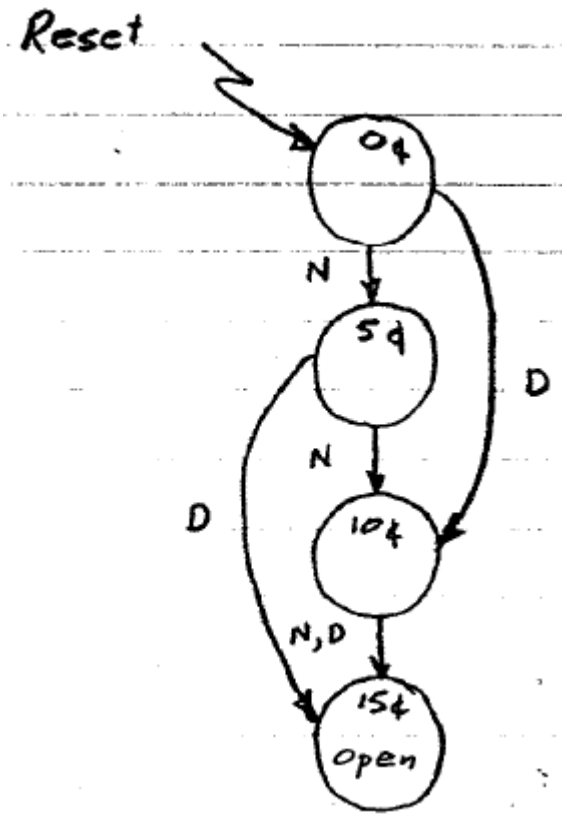
    This can be represented as a state diagram as shown below.



To keep the state diagram simple only transitions that explicitly cause a state change are concluded. Also Open is shown only in states where it is asserted.

3) This nine-state description is not the best. Since states S4, S5, S6, S6, and S8 have identical behavior they can be combined into a single state.

   A further reduction can be obtained if we think of each state as representing the amount of money received so far. The state diagram derived in this way is shown below. Note that only four states are required:



Techniques are available for minimizing the number of states. We now have a finite state machine with a minimum number of states. The symbolic state transition table is shown below. Note we assume that N and D are never asserted at the same time.

4) Next the states must be encoded. A natural state assignment would encode the states in 2 bits: state 0¢ as 00, state 5¢ as 01, state 10¢ as 10 and state 15¢ as 11. The encoded state transition table is shown below. A number of computer-based tools are available for funding an effective state encoding.

Symbolic Table

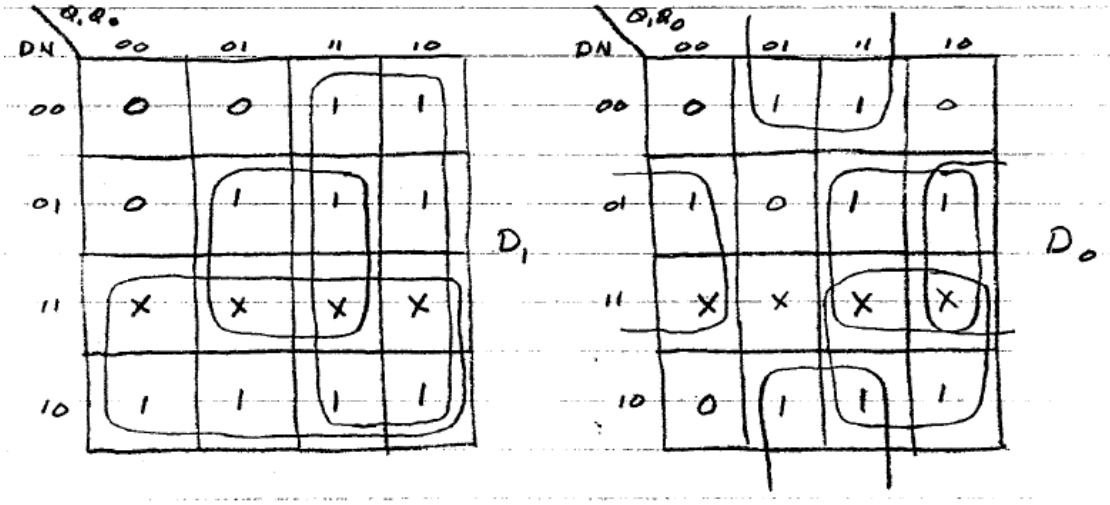
Present state	Inputs D	Inputs N	Next state	output open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

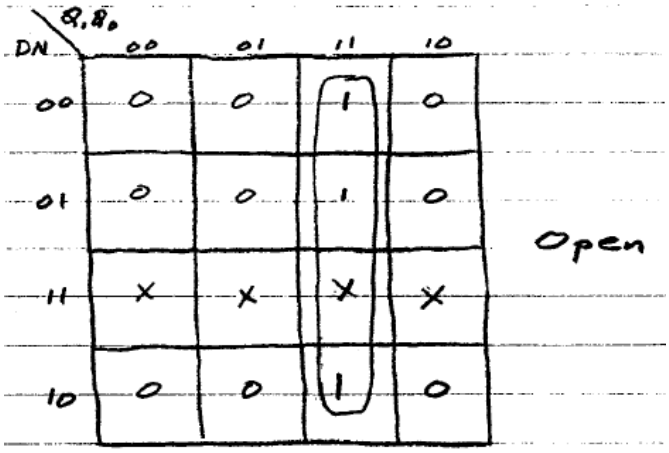
Encoded Table

Present state Q_1 Q_0	Inputs D N	Next state Q_1^+ Q_0^+	Output open
0 0	0 0	0 0	0
	0 1	0 1	0
	1 0	1 0	0
	1 1	X X	X
0 1	0 0	0 1	0
	0 1	1 0	0
	1 0	1 1	0
	1 1	X X	X
1 0	0 0	1 0	0
	0 1	1 1	0
	1 0	1 1	0
	1 1	X X	X
1 1	0 0	1 1	1
	0 1	1 1	1
	1 0	1 1	1
	1 1	X X	X

5) Implementation based on both D and JK flip-flops will be considered.

6) The K-maps for the D flip-flops implementation are shown below. These maps are obtained directly from the encoded state transition table.





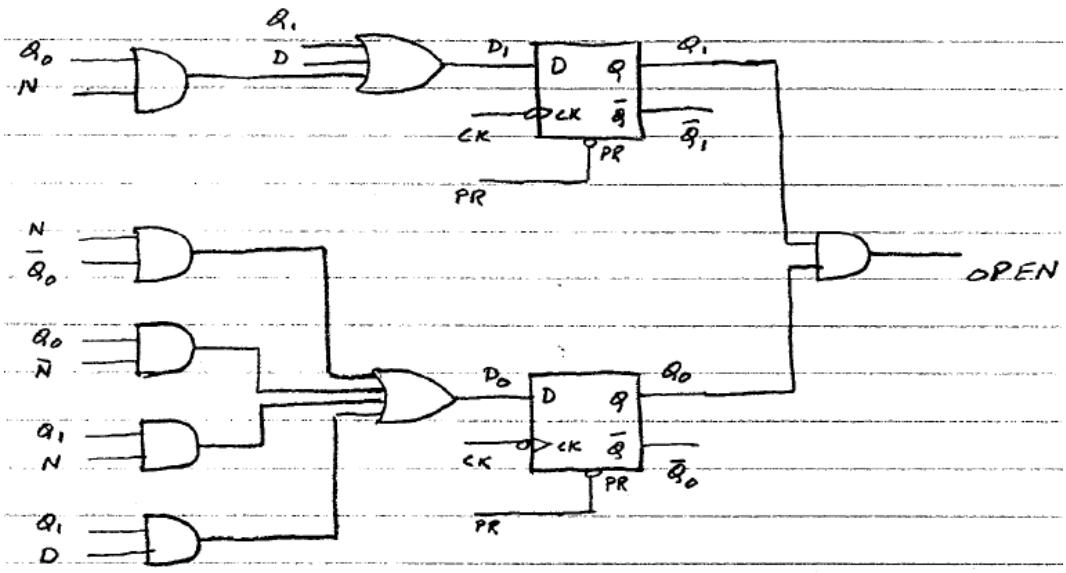
From the K-maps we obtain the following minimized equations:

$$D_1 = Q_1 + D + Q_0 \cdot N$$

$$D_0 = Q_0 \cdot \bar{N} + \bar{Q}_0 \cdot N + Q_1 \cdot N + Q_1 \cdot D$$

$$OPEN = Q_1 \cdot Q_0$$

The logic circuit is shown below. It uses eight gates and two flip-flops.



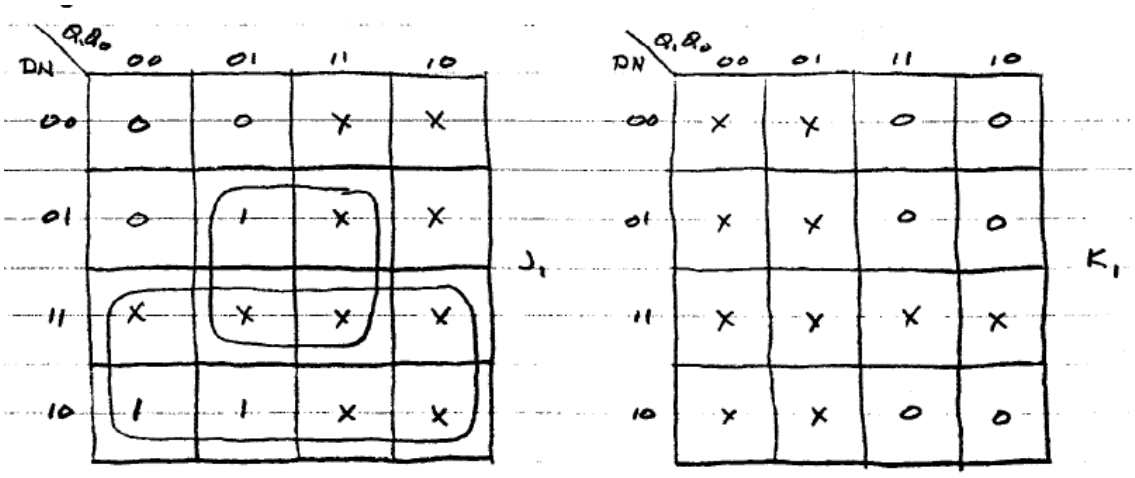
To implement using JK flip-flops the next-state functions must be remapped. We start with the JK excitation table as shown below:

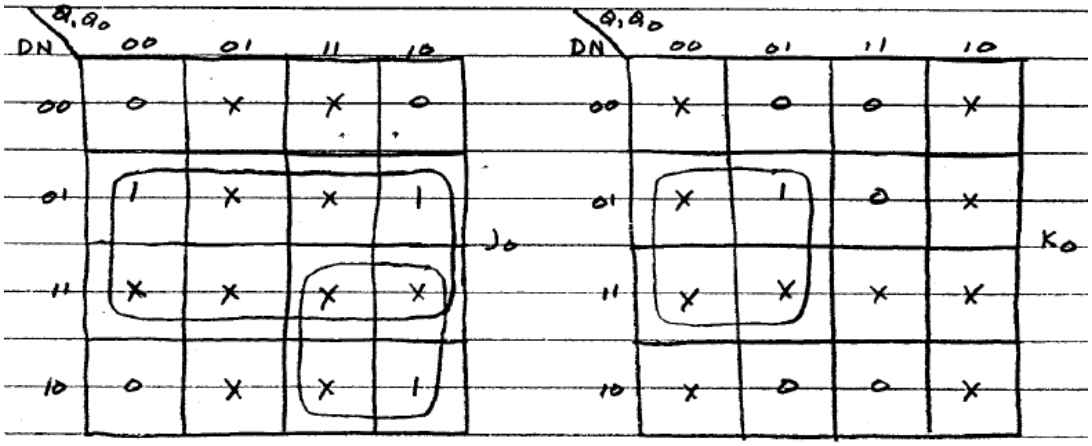
Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

The remapped state transition table is shown below.

Present State Q_1	Q_0	Inputs D	N	Next State Q_1^+	Q_0^+	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	X	0	X
		0	1	0	1	0	X	1	X
		1	0	1	0	1	X	0	X
		1	1	X	X	X	X	X	X
0	1	0	0	0	1	0	X	X	0
		0	1	1	0	1	X	X	1
		1	0	1	1	1	X	X	0
		1	1	X	X	X	X	X	X
1	0	0	0	1	0	X	0	0	X
		0	1	1	1	X	0	1	X
		1	0	1	1	X	0	1	X
		1	1	X	X	X	X	X	X
1	1	0	0	1	1	X	0	X	0
		0	1	1	1	X	0	X	0
		1	0	1	1	X	0	X	0
		1	1	X	X	X	X	X	X

The remapped K-maps for the JK implementation are given below.



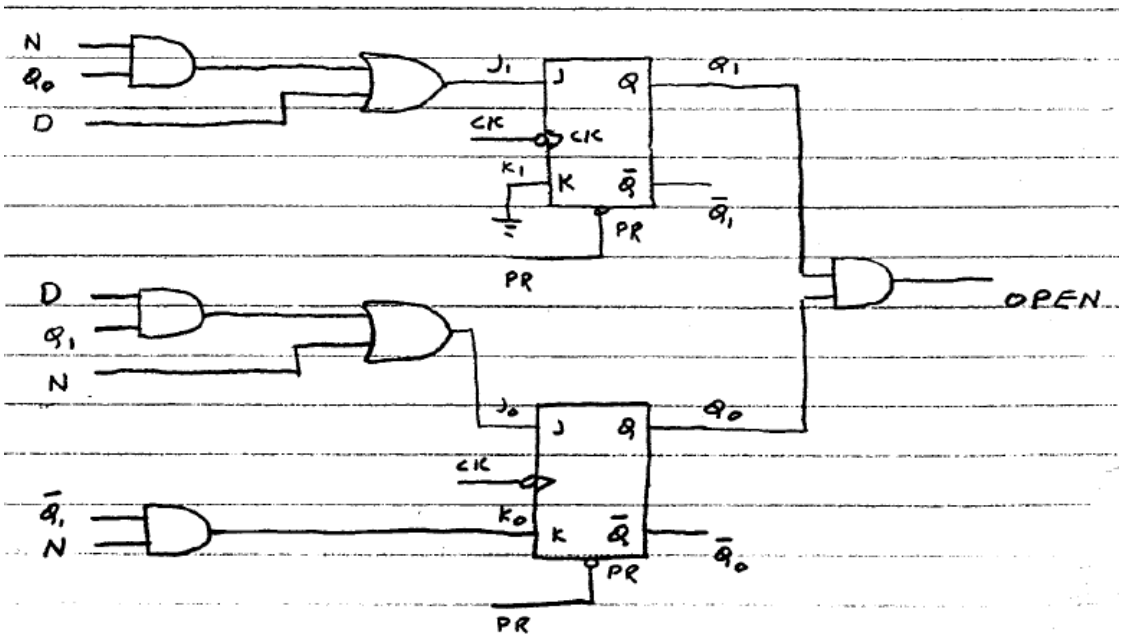


Using these K-maps we obtain the minimized functions:

$$J_1 = D + Q_0 \cdot N \quad , \quad K_1 = 0$$

$$J_0 = N + Q_1 \cdot D \quad , \quad K_0 = \bar{Q}_1 \cdot N$$

The implementation is shown below, it requires six gates and two flip-flops. This is a slight improvement over the D flip-flop case.

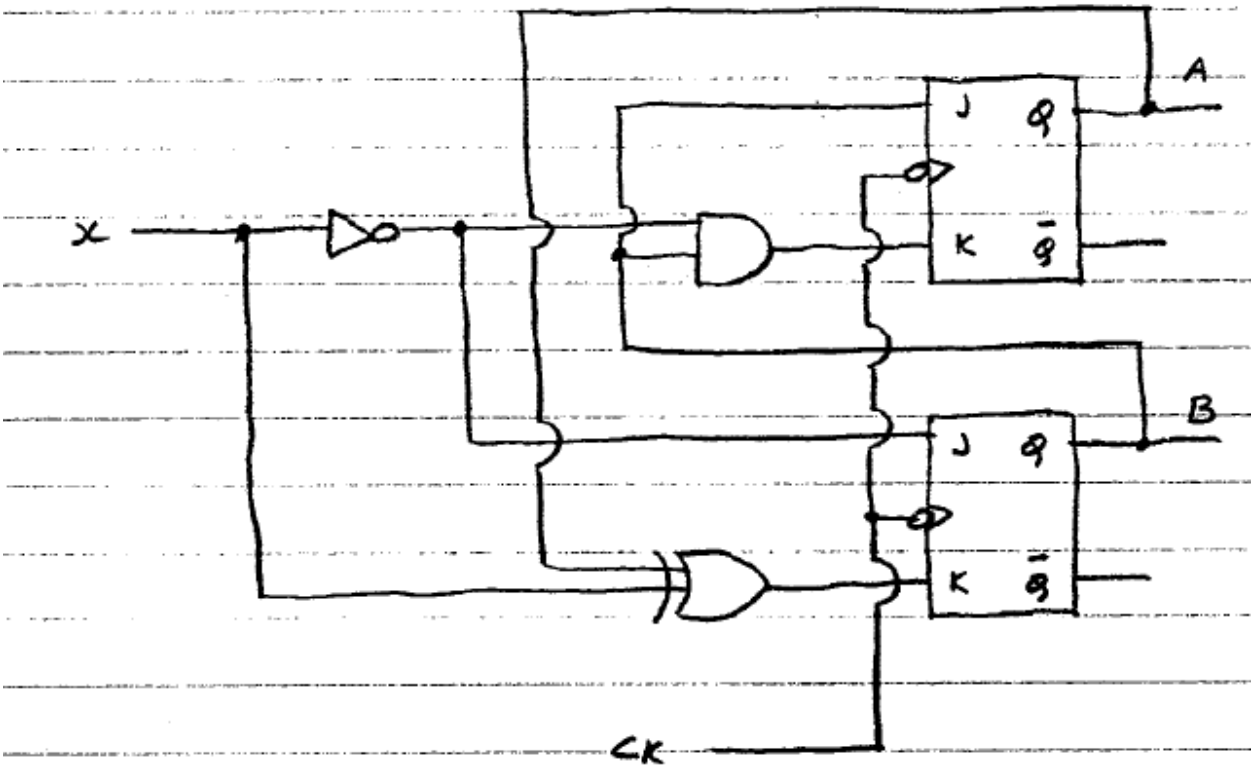


Moore and Mealy Machines

The most general form of a sequential circuit has inputs, outputs, and internal states. It is customary to distinguish between two types of sequential circuits: the Moore machine and the Mealy machine.

The Moor Machine

In the Moore machine, the outputs are a function of the present state only. An example of a Moore machine is shown below.



The circuit can be specified by the following input functions:

$$JA = B \quad , \quad KA = B \cdot \bar{x}$$

$$JB = \bar{x} \quad , \quad KB = \bar{A} \cdot x + A \cdot \bar{x} = A \oplus x$$

We see that the outputs are taken from the flip-flops and are a function of the present state only. The outputs change *synchronously* with the state transition and the clock edge. The finite state machine we have considered so far are Moore machines.

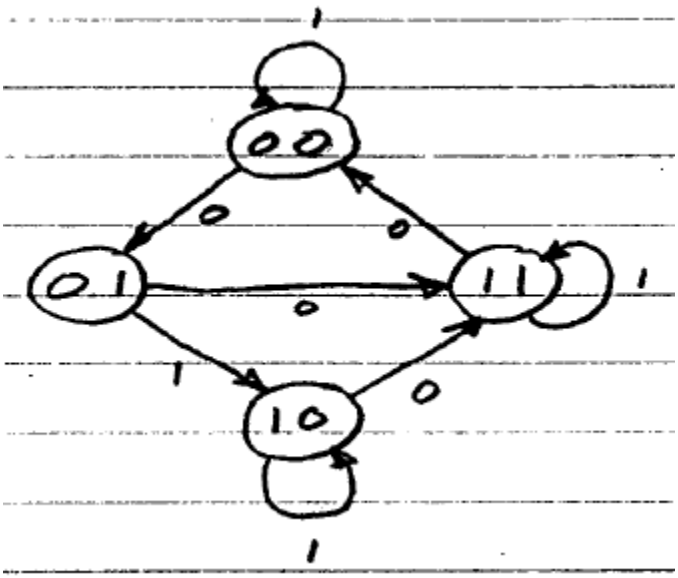
For the given system we can use the JK excitation table to find the state table and state transition diagram.

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

The state transition table is given below:

		Input							
A	B	x	J_A	K_A	J_B	K_B	A^+	B^+	
0	0	0	0	0	1	0	0	1	
0	0	1	0	0	0	1	0	0	
0	1	0	1	1	1	0	1	1	
0	1	1	1	0	0	1	1	0	
1	0	0	0	0	1	1	1	1	
1	0	1	0	0	0	0	1	0	
1	1	0	1	1	1	1	0	0	
1	1	1	1	0	0	0	1	1	

The state transition diagram is shown below:



Note that since the directed lines are marked with a single binary digit without a slash, there is one input variable and no output variables. The state of the flip-flops may be considered the outputs of the circuit.

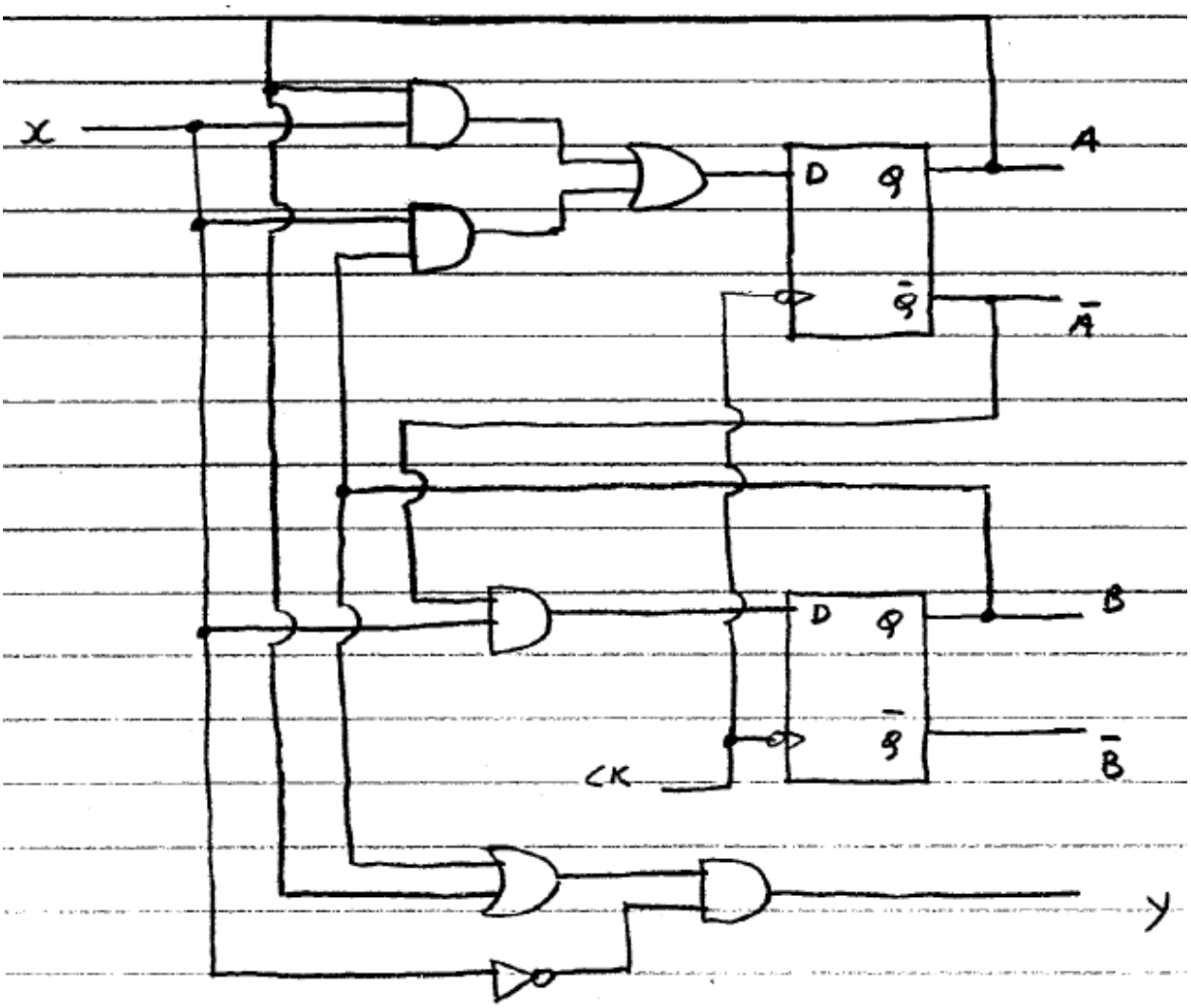
The Mealy Machine

In the Mealy machine, the outputs are functions of both the present state and inputs. An example of a Mealy machine is shown below. The circuit has one input x, one output y, and two D flip-flops A and B. The logic diagram can be expressed algebraically with two flip-flop input functions and one output circuit function:

$$DA = A \cdot x + B \cdot x \qquad = A^+$$

$$DB = \bar{A} \cdot x \qquad = B^+$$

$$y = (A + B) \cdot \bar{x}$$



We see that the output Y is a function of both input X and the present state of A and B. The outputs can change immediately after a change at the inputs, independent of the clock. A Mealy machine constructed in this fashion has *asynchronous* outputs.

For the given system we can use the D excitation table to find the state table and the state transition diagram.

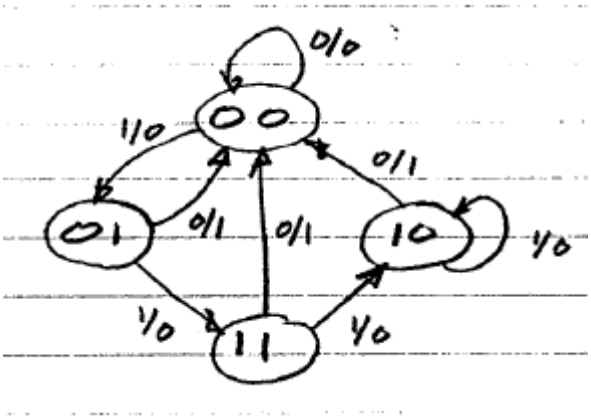
Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

Note that $D=Q^+$

The state transition table is given below:

		Input	DA	DB	Output
A	B	x	A^+	B^+	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The state transition diagram is shown below:

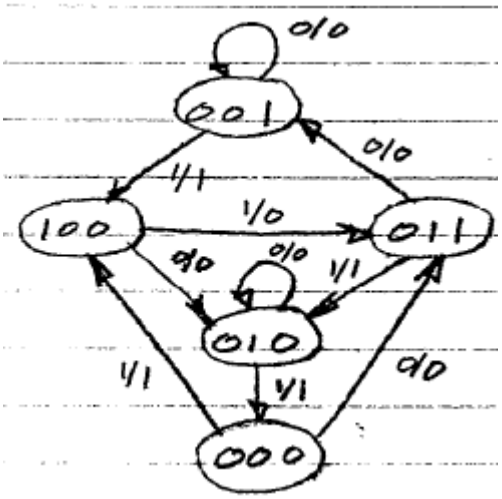


Note that the labeling on the directed lines indicate the input and output variables (i.e.: i/o = input/output).

An example where we obtain the circuit diagram gives the state transition diagram, inputs and outputs is given below.

Example:

A sequential circuit has three flip-flops, A, B, C; one input, x; and one output, y. The state diagram is shown below. The circuit is to be designed by treating the unused states as don't care conditions. Check the final circuit to ensure that it is self-starting. Use JK flip-flops in the design.

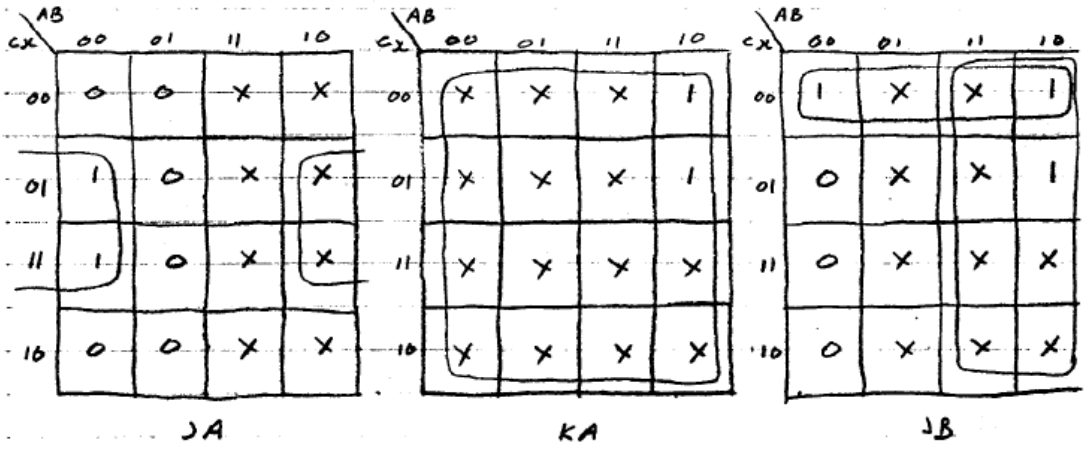


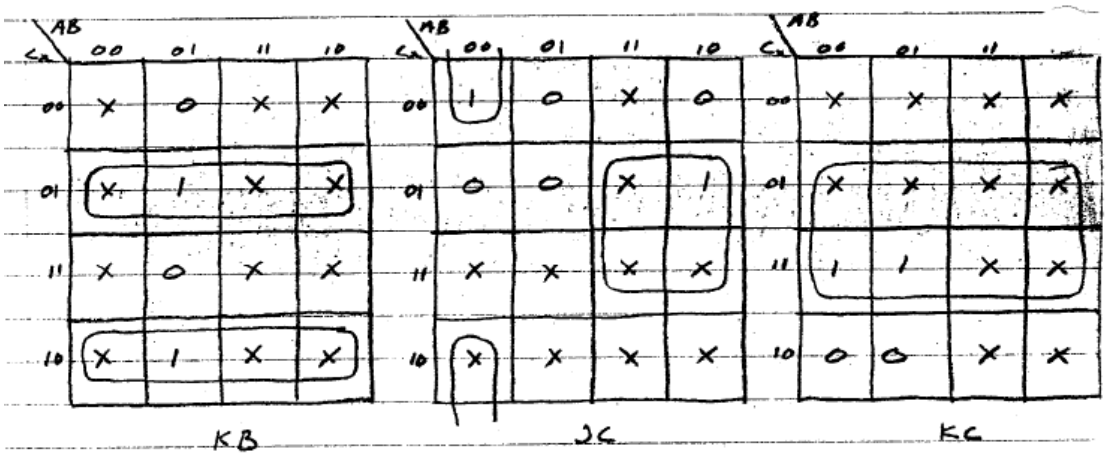
From the given diagram and the excitation table for JK flip-flops we can obtain the state table as given below.

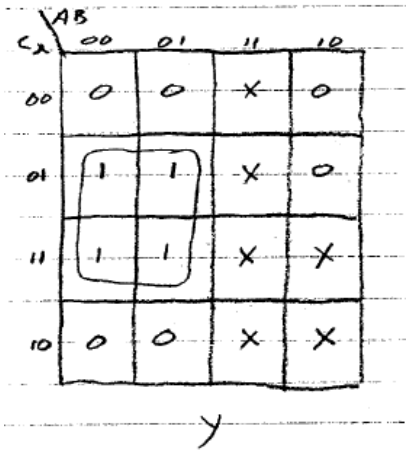
Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

	Input								output
A B C	x	A^+ B^+ C^+			JA KA	JB KB	JC KC		y
0 0 0	0	0 1 1			0 X	1 X	1 X		0
0 0 0	1	1 0 0			1 X	0 X	0 X		1
0 0 1	0	0 0 1			0 X	0 X	X 0		0
0 0 1	1	1 0 0			1 X	0 X	X 1		1
0 1 0	0	0 1 0			0 X	X 0	0 X		0
0 1 0	1	0 0 0			0 X	X 1	0 X		1
0 1 1	0	0 0 1			0 X	X 1	X 0		0
0 1 1	1	0 1 0			0 X	X 0	X 1		1
1 0 0	0	0 1 0			X 1	1 X	0 X		0
1 0 0	1	0 1 1			X 1	1 X	1 X		0
1 0 1	0	X^0 X^1 X^1			X^0 X^1	X^1 X^1	X^0 X^0		X^0
1 0 1	1	X^0 X^1 X^0			X^1 X^1	X^1 X^0	X^1 X^1		X^0
1 1 0	0	X^0 X^1 X^0			X^0 X^1	X^1 X^0	X^0 X^0		X^0
1 1 0	1	X^0 X^0 X^1			X^0 X^1	X^1 X^1	X^1 X^1		X^0
1 1 1	0	X^0 X^0 X^1			X^0 X^1	X^1 X^0	X^1 X^0		X^0
1 1 1	1	X^0 X^1 X^0			X^0 X^1	X^1 X^0	X^1 X^1		X^0

From the above table we can obtain the K-maps:







Using the K-maps we obtain the minimized functions:

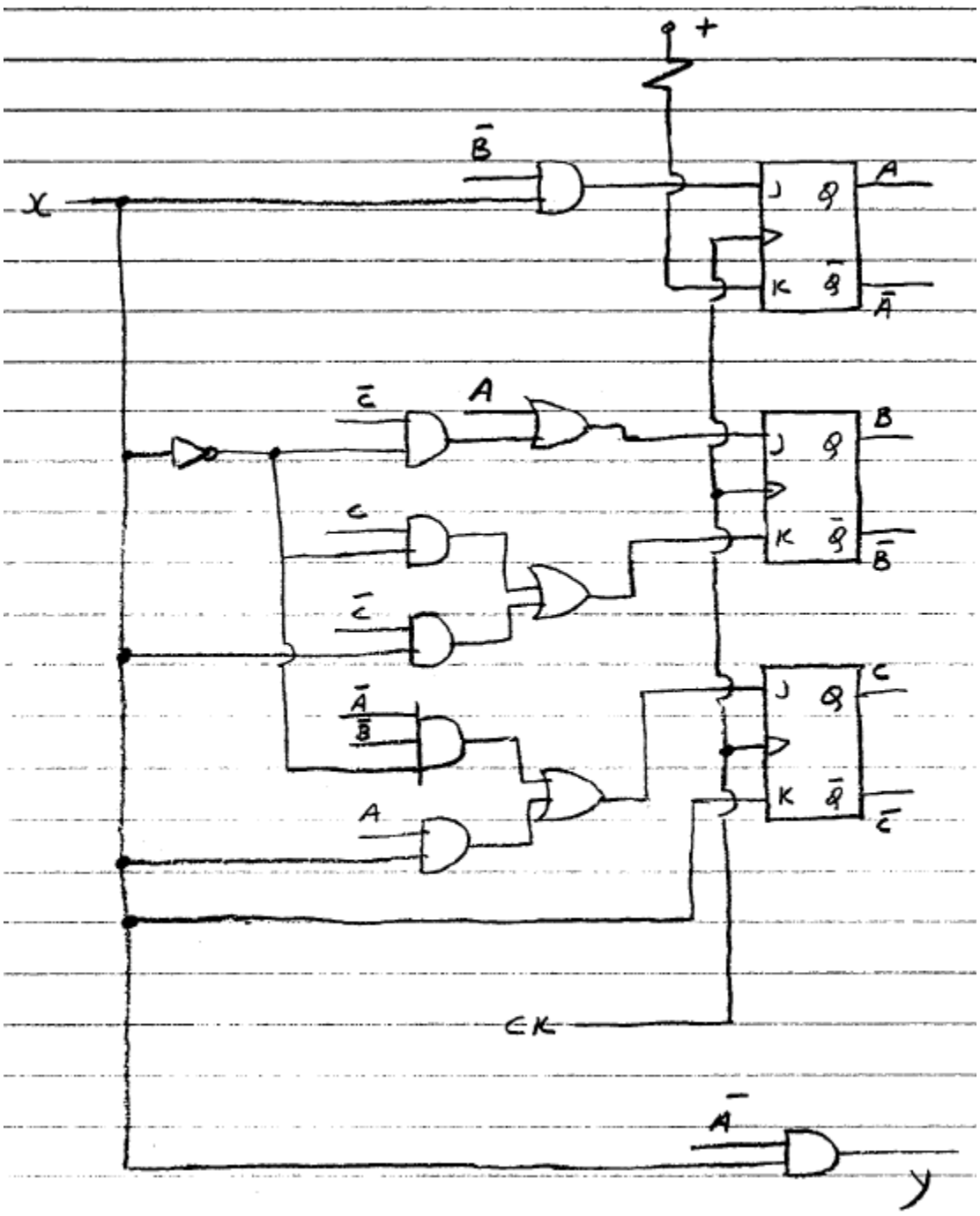
$$J_A = \bar{B} \cdot x \quad , \quad K_A = 1$$

$$J_B = \bar{C} \cdot \bar{x} + A \quad , \quad K_B = C \cdot \bar{x} + \bar{C} \cdot x$$

$$J_C = \bar{A} \cdot \bar{B} \cdot \bar{x} + A \cdot x \quad , \quad K_C = x$$

$$y = \bar{A} \cdot x$$

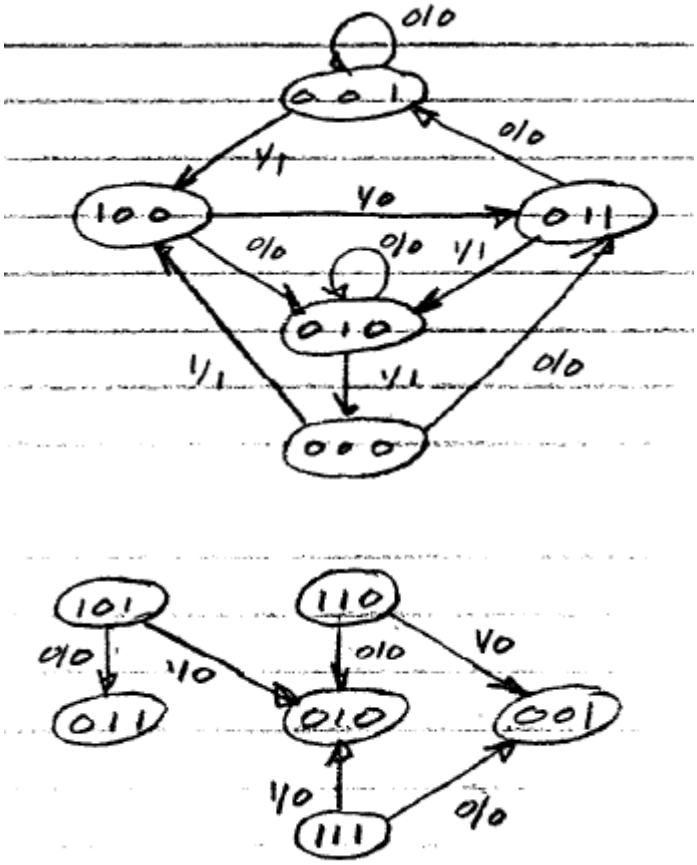
The implementation is shown below:



Note that the shorthand notation is used in the above diagram so that leads with the same label are connected together.

Since the state 101, 110, and 111 are not used the circuit must be checked to ensure that it is self-starting. To check we see what the don't cares become in the K-maps. The results are shown by the

numbers next to the X's and the state transition table. The complete state transition diagram is shown below.



We see from the last diagram that the system is self-starting.

Alternative State Machine Representations

State diagrams do not adequately capture the notion of an algorithm and are ineffective at capturing the structure behind complex sequencing. As a result hardware designers have shifted toward using alternative representations of FSM behavior that resemble software descriptions. The following alternate representations are being used:

A) Algorithmic state machine (ASM) notation, which is similar to program flowcharts but has a more rigorous concept of timing

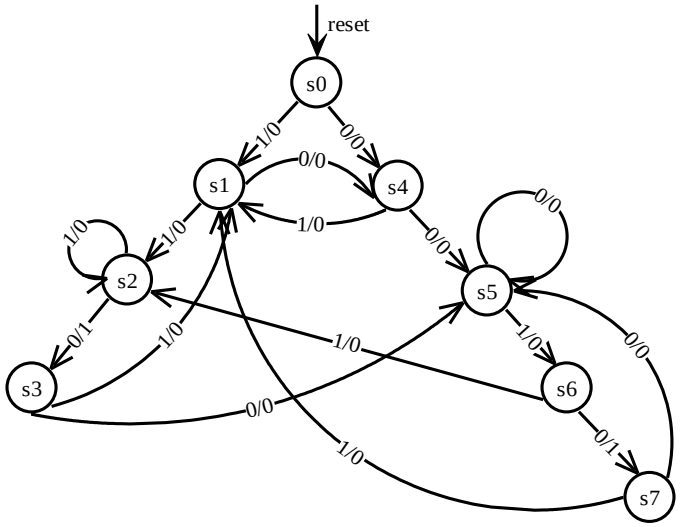
B) Hardware Description Languages (HDLs), which look like programming languages, but they explicitly support parallel computations.

Design Example: A Pattern Detector

Design a Mealy Machine that has one input X and one output Z:

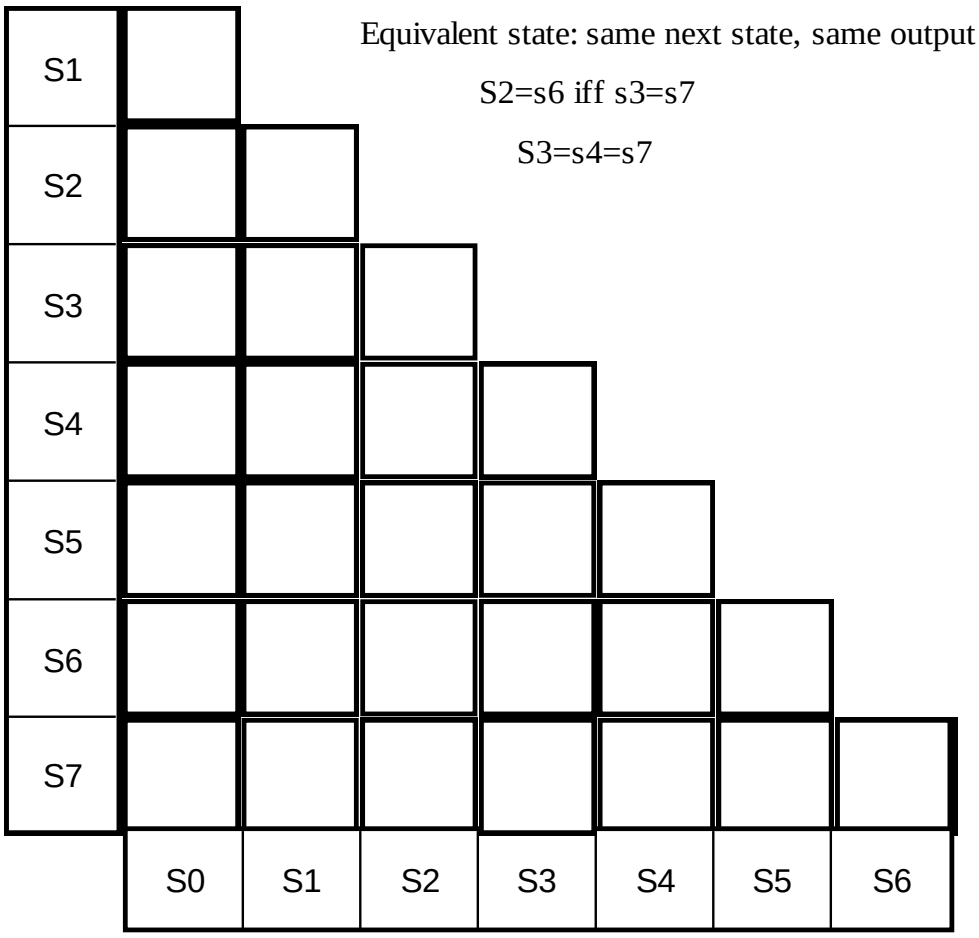
- Output Z=1 whenever either of the sequences 110 or 0010 has just been observed. At all other times Z is set to 0. Note that the machine does not reset once one of the two sequences is detected.

Step1: Derive the Symbolic State Diagram and State Transition Table:



Present state	Next state		Output, Z	
State	X=0	X=1	X=0	X=1
S0	s4	S1	0	0
S1	S4	S2	0	0
S2	S3	S2	1	0
S3	S5	S1	0	0
S4	S5	S1	0	0
S5	S5	S6	0	0
S6	S7	S2	1	0
S7	s5	s1	0	0

Step2: Reduce the State Table



Reduced State Table:

Present state	Next state		Output, Z	
State	X=0	X=1	X=0	X=1
S0	S3	S1	0	0
S1	S3	S2	0	0
S2	S3	S2	1	0
S3	S5	S1	0	0
S5	S5	S2	0	0

Step3: Find a Promising State Assignment

Heuristic rules--states which satisfy the following constraints should be given adjacent binary assignments:

1. States that have the same next state for a given input
2. States that are the next states of the same state
3. States that have the same output for the same input.

BC \ A	0	1
00	s0	s2
01	s1	s5
11		
10		s3

BC \ A	0	1
00	s0	
01	s1	
11	s2	
10	s5	s3

Final State Assignment (not unique)

Step4: Construct the Binary State Table

Present	Next State		output	
State,ABC	X=0	X=1	X=0	X=1
000	110	001	0	0
001	110	100	0	0
100	110	100	1	0
110	101	001	0	0
101	101	100	0	0

Final state assignment			
State	A	B	C
S0	0	0	0
S1	0	0	1
S2	1	0	0
S3	1	1	0
S5	1	0	1

Step5: Find Minimum Flip-flop Input & Output Functions

BC \ XA	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	Φ	Φ	Φ	Φ
10	Φ	1	0	Φ

A+

BC \ XA	00	01	11	10
00	1	1	0	0
01	1	0	0	0
11	Φ	Φ	Φ	Φ
10	Φ	0	0	Φ

B+

BC \ XA	00	01	11	10
00	0	0	0	1
01	0	1	0	0
11	Φ	Φ	Φ	Φ
10	Φ	1	1	Φ

C+

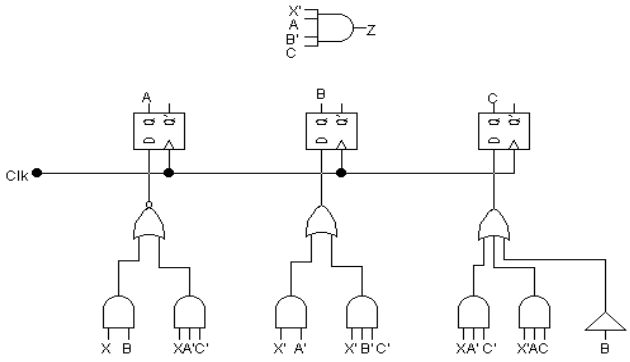
BC \ XA	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	Φ	Φ	Φ	Φ
10	Φ	0	0	Φ

Z

$$\overline{A^+} = X\,\overline{A}\,\overline{C} + B\,X \qquad C^+ = B + \overline{X}\,A\,C + X\,\overline{A}\,\overline{C}$$

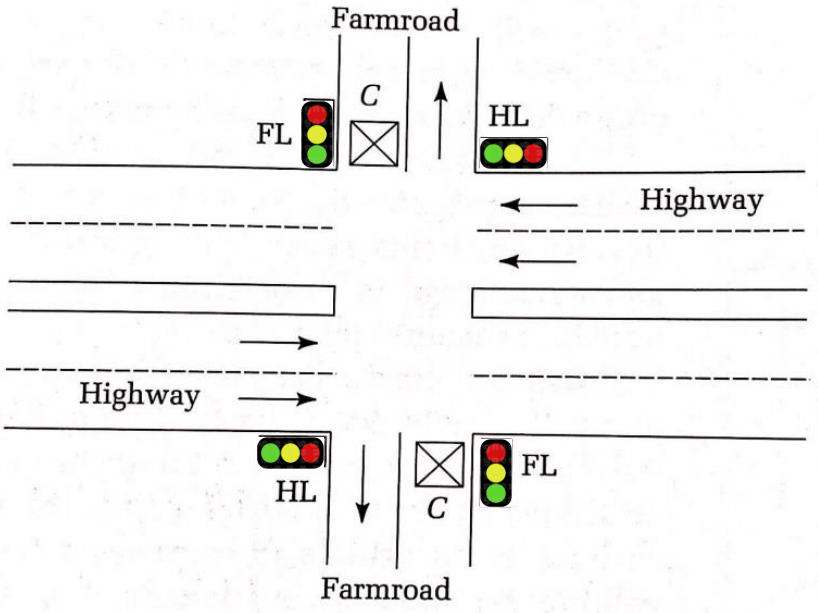
$$B^+ = \overline{X}\,\overline{A} + \overline{X}\,\overline{B}\,\overline{C} \qquad Z = \overline{X}\,A\,\overline{B}\,\overline{C}$$

Step6: Flip-Flop Implementations



Design Example: Traffic Light Controller

A busy highway is intersected by a little-used farmroad. Detectors are placed along the farmroad to raise the signal C when a vehicle is waiting to cross the highway. The lights should remain green in the highway direction if no vehicle is detected, and should remain green for a set interval even if vehicles are waiting to cross the highway. If a vehicle is detected on the farmroad, the highway lights should change from green to yellow to red, allowing the farmroad lights to become green. The farmroad lights stay green no longer than a set interval to allow the traffic to flow along the highway.



You may assume there is an external timer that, once set via the control signal "ST" (set timer), will assert the signal TS after a short time interval has expired (used for timing yellow lights) and "TL" after a long time interval (for green lights). The timer is automatically reset when ST is asserted.

Step 0: Simplify the problem by using external timer for timing intervals and using decoders to reduce the total number of outputs.

Step 1: Derive Algorithmic State Machine Notation (ASM)
Step 2: Derive Symbolic State Diagram
Step 3: Binary State Assignment
Step 4: Derive Next State Logic and Output Logic
Step 5: Circuit Synthesis