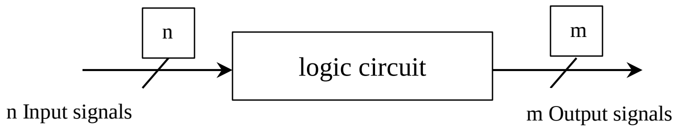
CHAPTER 1 DIGITAL LOGIC

Section 1. Introduction -- Logic Circuits

The electronics inside a modern computer are comprised of circuits in which only two voltage levels are of interest: a high voltage (typically in the range 3 – 5 volts) and a low voltage (typically 0 – 1.5 volts). Rather than refer to the voltage levels, however, we talk about *signals* that are (logically) true, or 1, or are *asserted*; or signals that are (logically) false, or 0, or are *deasserted*. Because of the association of digit values with these various voltage levels such circuits are known as *digital circuits*. The further adaptation of the terms "true" and "false" from logic has resulted in such circuits being referred to as ***logic circuits***.

When we associate the value *true/1/asserted* with a high voltage and the value *false/0/deasserted* with a low voltage we employ what is known as a ***positive logic***-based system. This is the system we shall assume in this course. The values *false/0/deasserted* and *true/1/asserted* are ***complements*** of one another.

In their most abstract form, we can regard a digital circuit as one that accepts one or more digital values as input values (or input signals) from an external source and produces one or more output values (or output signals). We can illustrate such a circuit abstractly as follows:



Logic circuits are classified as being either *combinational* circuits or *sequential* circuits. In ***combinational circuits*** the output values are uniquely determined by the input values. ***Sequential circuits***, on the other hand, contain memory elements that can store a 0 or 1, and which collectively comprise the ***state*** of the sequential circuits. Unlike combinational circuits the output values are uniquely determined by both the input values and the current state of the circuit. Moreover the input values and the current state together uniquely determine a, possibly different, *next state* for the circuit.

Section 2. Truth Tables, Logic Functions and Logical Expressions

Because the output values for a combinational circuit are uniquely determined by the input values, when there are a small number of input signals it is possible to represent the associations between input and output values using a ***truth table***, a structure once again adopted from logic.

Examples: Examples of truth tables for representing combinational circuits with one output signal, X, and 2, 3 and 4 inputs respectively.

Done in class

Example: A 1-bit, ***half adder*** is a combinational circuit with two input values A and B representing 1-bit values to be added, and two output values ***sum*** and ***carry*** representing the sum and carry values from doing binary addition on the values of A and B. Represent the input-output associations with a truth table.

Done in class

Example: A 1-bit, ***full adder*** is a combinational circuit with three input values A, B and carry-in (C_{in}) representing 1-bit values to be added together with a third value representing a value to be carried into the addition, and two output values ***sum*** and ***carry-out (C_{out})*** once again representing the sum and carry values from doing binary addition on the values of A, B and C_{in}. Represent the input-output associations with a truth table.

Done in class

Logic Functions

Truth tables work fine for describing combinational circuits when there are a small number of input signals, say 2, 3 or 4, but beyond this the truth tables have too many rows. An alternative approach for representing combinational circuits is to regard the various patterns of input signal associations as binary numbers and to give the binary numbers of only those input combinations that have the value 1 associated with them, the understanding being that the entries not listed have the value 0 associated with them

	A	B	f
0	0	0	**1**
1	0	1	0
2	1	0	0
3	1	1	**1**

Represented as `f(A,B)` $= \Sigma$`(0,3)`

Of course there is no particular reason for favoring those entries that have ones, so with a slight adjustment in notation, we can show those entries that have the value 0 associated with them. For the above case this would be

`f(A,B)` $= \Pi$`(1,2)`

It is imperative that one show the number of input variables since otherwise we have no way of knowing how many rows are in the table being represented.

Examples of More Logic Functions:

Done in class

Logical (Boolean) Expressions

Another approach is to express the logic function with logic expressions. This is done with the use of notation and language borrowed *Boolean algebra* (named after George Boole, a 19th-century mathematician), and hence the resulting expressions are sometimes known as ***Boolean expressions.***

In Boolean algebra, all the variables have the values 0 or 1 and, in typical formulations, there are three operators:

- The OR operator, written as +, as in $A + B$, yields a result of 1 if the value of either (or both) of the variables is 1; otherwise the result is 0. The OR operation is also called a *logical sum*. We can represent the OR operations via the truth table

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

- The AND operator, written as × as in $A \times B$ has result of 1 only if both inputs are 1; otherwise the result is 0. The AND operator is also called *logical product*. We can represent the AND operation via the truth table

A	B	A × B
0	0	0
0	1	0
1	0	0
1	1	1

Alternatively a dot is sometimes used for logical AND as in A·B, and even simple adjacency positioning when there is no danger of confusion and thus not using an operator symbol, whence instead of A × B or A·B one would simply write AB.

- The unary operator NOT, written as $\bar{A}$ for a value A, returns the complement of the value of A; that is it returns 1 if A is 0 and 0 if A is 1. For completeness we give its truth table, even though it is simple.

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

We note that while we have defined OR (+) and AND (·) for just two operands, we can easily extend it to n operands as follows:

- The OR of operands x_1, x_2, …, x_n is 1 only if at least one the values of an operand x_1,…,x_n is 1; otherwise the value is 0. We represent the OR of x_1,x_2 …,x_n as $x_1+x_{2+} \ldots +x_n$

- The AND of operands x_1, x_2, …, x_n is 1 only if the values of all the operands x_1,…,x_n are 1; otherwise the value is 0. We represent the AND of x_1,x_2 …,x_n as $x_1 \times x_{2\cdot} \times \ldots \times x_n$ or $x_1 \cdot x_{2\cdot} \ldots \cdot x_n$ or even $x_1 x_{2\cdot} \ldots x_n$

The reason we can use such representations when there are more than two operands is because both the OR and the AND operators have the associativity property.

Starting with our above definitions and notations for NOT, AND, OR, we can combine them into more complex Boolean expressions using parenthesis for grouping, if necessary for clarity, in a manner familiar to you from standard algebra. An example of such an expression is

$$(x1 \cdot x2) + \left(x1 + \overline{(x2 \cdot x3)}\right) + x3$$

Canonical Expressions

We now show how one can associate Boolean expressions with a truth table so that the function defined by the Boolean expression has the given truth table as its associated truth table. In particular we show how to derive two such Boolean expressions known as the *canonical Boolean expressions* for the truth table. There are two forms of canonical expressions – *canonical sum-of-products* form and *canonical product-of-sums* form.

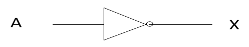
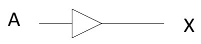
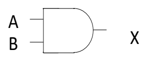
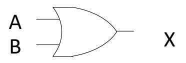
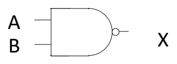
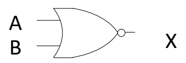
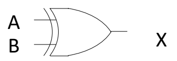
1. ***sum of products form*** (SOP):
   a. Each row where the value of the table is 1 is a product term (or *minterm*); all the product terms are to be ORed (although in deference to the use of the symbol +, we use the word "summed" even though there is no addition going on).
   b. For each product term, leave the individual variables uncomplemented if the value of the variable in that row is 1, otherwise complement it.

2. ***product of sums form*** (POS):
   a. Each row where the value of the table is 0 is a sum term (or *maxterm*); all the product terms are to be ANDed (although again in deference to the notation used we sometimes say "multiplied," even though no arithmetic multiplication is going on).
   b. For each sum term, leave the individual variables uncomplemented if the value of the variable in that row is 0, otherwise complement it.

Examples: Done in class

Section 3. Logic Gates and Digital Circuits

One way to implement digital circuits is to build them from simpler logic circuits known as ***logic gates***. On the following page we list some of the fundamental logic gates that we will use in this chapter, and indeed in the rest of the course. All are combinational circuits. Rather than describe the input-output relationship with cumbersome truth-tables, however, we use Boolean expressions instead.

Name	**Symbol**	**Expression**
inverter		$X = \overline{A}$
buffer		$X = A$
AND		$X = A\bullet B$ or $X = AB$
OR		$X = A + B$
NAND		$X = \overline{A \cdot B}$
NOR		$X = \overline{A + B}$
XOR		$X = A\overline{B} + \overline{A}B$ (also denoted $A\oplus B$)

Although we don't show them here, one can visualize AND, OR, NAND and NOR gates that have three or more inputs, and indeed we shall assume these gates exist for any desired number of inputs, even if no such gates are actually manufactured.

Pages 148-149 of your textbook have a discussion of how the inverter, NAND, and NOR gates might be implemented using transistors and resistors.

Examples of Implementations of Logic Circuits

Done in Class

Section 4. Circuit Equivalence, Boolean Algebra

In the previous section we found that both the canonical sum of products form and the canonical product of sums form can be used to express the same function, as evidenced by the fact that both expressions have the same truth table. Two logical expressions are said to be ***logically equivalent*** if they both generate the same truth table, or define the same logic function.

Logical equivalence is particularly important when used in connection with logic circuits as simpler logical expressions generally result in simpler circuit which may involve fewer a gates overall and fewer inputs in some of these gates.

Example: Consider the logic function `f(x,y,z) = Σ(0,2,4,6,7 ).`

The canonical expressions associated with this function are:

Canonical SOP: $f(x,y,z) = \bar{x}\bar{y}\bar{z} + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z} + xyz$

Canonical POS: $f(x,y,z) = (x + y + \bar{z})(x + \bar{y} + \bar{z})(\bar{x} + y + \bar{z})$

Note, however, that the Boolean expression $\bar{z} + xy$ is logically equivalent to both of the above expressions but is structurally much simpler than either f the above, and hence yields a much simpler implementation as a logic circuit.

Boolean Algebra

Determining that two logical expressions are logically equivalent can be tedious and time-consuming, even for expressions with a small number of inputs, if the expressions have many or complex terms. Even more challenging, however, is to find an expression that is indeed simpler, but logically equivalent to, a given expression. Fortunately there is an abstract mathematical system – *Boolean algebra* – that assists us greatly, if not invaluably, in this effort. Although originally based on formal logic, Boolean algebras can be generalized beyond this particular case, and it is this general system that we define now.

Definition of a Boolean Algebra: A ***Boolean algebra*** is a mathematical system $< B, +, \bullet, \overline{\ } >$, where B is a set and

$+ : B \times B \rightarrow B$

$\bullet : B \times B \rightarrow B$

$\overline{\ } : B \rightarrow B$

Are operators such that for all x, y, z in B

1. $x + y = y + x$; $x \bullet y = y \bullet x$ *commutativity* of + and •
2. $(x + y) + z = x + (y + z)$; $(x \bullet y) \bullet z = x \bullet (y \bullet z)$ *associativity* of + and •
3. $x + (y \bullet z) = (x + y) \bullet (x + z)$; $x \bullet (y + z) = x \bullet y + x \bullet z$ *distribution* of + over • and • over +

In addition there exist distinguished elements 0 and 1 in B such that for all x in B

4. $x + 0 = x$; $x \bullet 1 = x$
5. $x + \bar{x} = 1$; $x \bullet \bar{x} = 0$

As we've noted earlier, often adjacency of operands is used instead of the dot notation, whence we write xy instead of $x \bullet y$.

Of special interest to us is the Boolean algebra with B = {0,1} and where $+ \leftrightarrow$ OR, $\bullet \leftrightarrow$ AND, and $\overline{\ } \leftrightarrow$ NOT. This particular Boolean algebra is more properly called the *logic algebra* or *switching algebra*, and was in fact the prototype Boolean algebra. It is often referred to as **the** Boolean algebra. However **it is not the only Boolean algebra**.

Example: Let S be an arbitrary set and let B be the power set of S; that is, the set of all subsets of S. If we now make the following associations $0 \leftrightarrow \phi$; $1 \leftrightarrow S$; $+ \leftrightarrow \cup$; $\bullet \leftrightarrow \cap$; and $\overline{\ } \leftrightarrow$ set complement, then B is a Boolean algebra. Moreover, if S has n elements, then B has 2^n elements, thus we can generate infinitely many examples of Boolean algebras with more than 2 elements.

The following results can be derived about all Boolean algebras (not just the logic algebra).

a. (idempotency) $x + x = x$; $x \bullet x = x$
b. $x + 1 = 1$; $x \bullet 0 = 0$
c. (absorption 1) $x + x\,y = x$; $x\,(x + y) = x$
d. (absorption 2) $x + \bar{x}y = x + y$; $x\,(\bar{x} + y) = xy+$
e. (DeMorgan's laws) $\overline{x + y} = \bar{x} \bullet \bar{y}$; $\overline{x \bullet y}) = \bar{x} + \bar{y}$

Boolean algebras are also ***dual*** with respect to + and •, i.e. if one relationship is true in the Boolean algebra then the relationship derived by interchanging + and • and 0 and 1 is also true in the algebra.

Section 5. Circuit Simplification and Gate Homogeneity

The real power of the relationship between Boolean algebra and digital circuits manifests itself in the following way: As we have already seen, any combinational circuit can be represented by a Boolean expression and vice-versa. Using Boolean algebra one can transform this Boolean expression (and hence the combinational circuit) into one which is different but logically equivalent. Two particular types of transformations are particularly desirable.

1. Transform a given combinational circuit into one which requires a smaller number of gates (*circuit simplification*).
2. Transform a combinational circuit into one which uses gates chosen from a restricted set of gate types. We have already shown that any combinational circuit can be implemented using only AND gates, OR gates, and inverters. We shall show later that in fact any circuit can be implemented using just NAND gates or just NOR gates (*gate homogeneity*).

We now consider each of these issues.

Circuit Simplification

As we noted earlier, the complexity of a logic circuit that implements a Boolean function is directly related to the complexity of the Boolean expression that defines the function. It is to the advantage of a circuit designer therefore to strive for a design which simplifies a combinational circuit in the following ways:

1. The length of the chain of gates input signals must traverse to produce the desired output signal should be as short as possible, since each gate through which signals are propagated causes a delay between the arrival of the signals and the appearance of the appropriate output signal. The existence of canonical POS and SOP expressions for a truth table show us that any combinational circuit can be realized as a two level circuit if input signal are available both in their true form and in their inverted form; if both of these forms are not available, an additional level of inverters may be needed to generate them.
2. The number of gates and gate inputs should be as small as possible since these influence the cost of implementing the circuit.

In the following discussion we first show how Boolean expressions can be simplified using relations from Boolean algebra. Unfortunately, for complex expressions it is usually difficult to tell whether a simplified expression meets criterion 2 above. We follow this with a gate minimization technique based on a structure known as a *Karnaugh map* which allows one to create a SOP or POS expression for a truth table which uses a minimum number of terms and variables. Further simplification may still be possible, but usually the expression derived with by Karnaugh maps is sufficient.

A. **Simplification Using Boolean Algebra Relations:** We use examples to illustrate how Boolean algebra can be used to simplify an expression defining a Boolean function.

   Examples: Done in class

B. **Simplification Using Karnaugh Maps:** A *Karnaugh map* is an arrangement of cells, each one representing a combination of variables in a truth table.

2 input Karnaugh map

AB	
00	
01	
11	
10	

3 input Karnaugh map

A \ BC	00	01	11	10
0				
1				

4 input Karnaugh map

AB \ CD	00	01	11	10
00				
01				
11				
10				

To minimize a SOP expression the output values from the truth table are inserted in the table in squares determined by the input combination which produced the output value and the input combination represented by each square. One then seeks out and circles groups of "logically adjacent" 1's of the largest possible sizes that are powers of two. We define *logically adjacent* as follows:

1. Two squares of 1's are logically adjacent if the product terms they represent are the same except in one variable, where they are complementary. Such a pair of cells can be described by a product term comprised of the variables where they agree.

Examples:

2. Four squares of 1's are logically adjacent if they can be decomposed into two sets of logically adjacent pairs so that the product terms describing these pairs are the same except in one variable, where they are complementary. Such a quadruple of cells can be described by a product term comprised of the variables that agree in the product terms of the pairs into which the quadruple was decomposed.

Examples:

3. In general 2^n squares of 1's ($n > 1$) are logically adjacent if they can be decomposed into two (disjoint) sets of 2^{n-1} logically adjacent squares of 1's such that the product terms which describe each of these sets are the same except in one variable, where they are complementary.

Such a group of 2^n cells can be described by a product term comprised of the variables that agree in the product terms of the sets of 2^{n-1} cells into which the given group was decomposed.

Examples:

C. **Quine-McKluskey Tables**: Karnaugh maps are effective for generating minimimal SOP expression involving up to four variables, and with some additional effort can be made to work for expressions requiring up to six variables. For expressions involving more than 6 variables, it is impractical to use Karnaugh methods. Fortunately there is an expression minimization method that is practical for more than 6 variables; in fact this method can be described algorithmically and hence implemeted by a computer. The method involves the use of Quine-McKluskey tables. Since we shall have no need for the method in this course, however, we do not discuss it here, but rather leave it to the student to explore this technique.

Gate Homogeneity in Logic Circuits

In the following discussion we show how any Boolean expression can be implemented using only NAND gates or only NOR gates. The ability to employ such gate homogeneity in a circuit is attractive from an engineering viewpoint since NAND gates and NOR gates have simple electronic realizations. Unfortunately logic design with NAND and NOR gates is not as straightforward as design with AND and OR gates and inverters (primarily because associativity does not hold for NOR and NAND), whence achieving gate minimization and gate homogeneity is more difficult. For our purposes we concentrate on showing how (minimal) POS and SOP expressions can be implemented using only NAND gates or only NOR gates. In logic diagrams a small circle is used to denote complementation as is observable in the graphic symbols for inverters, NOR gates and NAND gates.

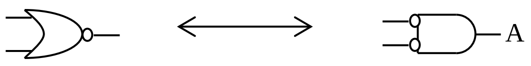
Consider now the Boolean expressions

$\overline{x + y}$ and $\overline{x \bullet y}$

which can be implemented in a digital circuit by a NOR gate and a NAND gate respectively. If we apply DeMorgan's laws to these expressions however we get

$$\overline{x + y} = \bar{x} \bullet \bar{y} \quad \text{and} \quad \overline{x \bullet y} = \bar{x} + \bar{y}$$

This suggests that in addition to representing a NOR gate with an OR symbol followed by a circle, we can represent it by an AND symbol preceded by circles in all inputs.

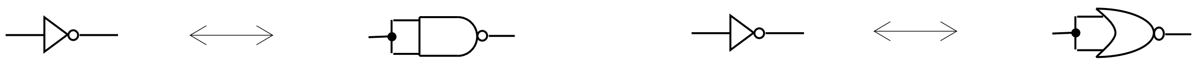


Similarly a NAND gate can be represented by an OR symbol preceded by circles in all inputs.

Finally we note that by idempotency

$\overline{x + x} = \overline{x}$ and $\overline{x \bullet x} = \overline{x}$

suggesting that inverters can be implemented with a NAND gate or a NOR gate as follows:



We now show by examples how these observations can be used to convert any digital circuit to one which uses only NAND gates or only NOR gates.

Examples:

Done in class

Section 6. Basic Logic Circuits I - Combinational Circuits

Integrated Circuits

Logic gates are available commercially in integrated circuit (IC) form. Here the gates are manufactured on a silicon wafer and sealed in a protective package that has pins protruding from it in order to make connections to the gates. Depending on the number of gates they contain, ICs are roughly classified as follows:

small scale integrated circuit (SSI)	1 to a few 10s of gates
medium scale integrated circuit (MSI)	a few 10s to a few 100s of gates
large scale integrated circuit (SLSI)	a few 100s to a few 100,000 gates
very large scale integrated circuit (VLSI)	more than a few 100,000 gates

Combinational Circuits

For the remainder of this chapter we shall discuss some fundamental design principles of digital circuits, emphasizing those that relate directly to computer organization. In keeping with our hierarchical approach, we shall use gates as the primitive components for our components, and use these to develop other SSI and MSI circuits (most of which are themselves available as integrated circuits). These will in turn become primitive components of sub-layers within the digital logic level.

As we have seen, combinational circuits can be described by a truth table showing the binary relations between n inputs and m outputs or by m Boolean functions of n input variables, thus allowing many of the techniques of the previous section to come into play. Our first three examples below illustrate this approach to combinational circuit design, though it works well primarily for relatively simple circuits (SSI and some MSI) and is generally not widely used with more complex circuits (MSI and above).

Examples of combinational circuits

1. The *full adder*: A full adder accepts three bits as input - 2 addend bits, A and B, and a carry-in bit , C_{in}, and produces as output a sum bit, S, and a carry-out bit, C_{out}. This circuit is not new, as we discussed it earlier. What we do here, however, is provide an alternative implementation of the circuit.

If we represent a one-bit adder by the following symbol

then we can represent a two-bit adder by the following symbol

Moreover we can implement this two-bit adder using two one-bit adders as follows

This implementation is known as a ***ripple-carry adder*** for the manner in which it determines the final carry-out. The carry-out of the first full adder must feed (or "ripple") into the second full adder as its carry-in. As you will see in an assignment, when incorporated into higher bit adders it can result in a significant delay in calculating the adder's final sum and carry-out over other approaches such as carry-select (discussed in your textbook on pages 165-166) and carry-lookahead (Exercise 14 at the end of Chapter 3).

2. A 2-bit left/right ***shifter*** is a circuit that has two inputs D0 and D1, a control line C that determines the direction of the shift (0 for left and 1 for right), and two output lines S0 and S1. On a left shift, S0s gets the value 0 and S1 gets the value of S0; on a right shift S1 gets the value 0 and S0 gets the value of S1. A truth table for the shifter is shown below, following by (simplified) expressions for S0 and S1 in terms of C, D0, and D1.

C	D1	D0	S1	S0
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

$S1 = \bar{C}D0$

$S0 = CD1$

Similarly we can specify a 3-bit left-right shifter with the following truth table and expressions for S0, S1 and S2 in terms of C, D0, D1 and D2

.

C	D2	D1	D0	S2	S1	S0
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	1	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	1
1	1	1	1	0	1	1

$S2 = \bar{C}D1$

$S1 = \bar{C}D0 + CD2$

$S0 = CD1$

Finally, analyzing what we have seen in the expressions for the S's above in terms of C and the D's, we extrapolate that in an 8-bit left-right shifter, we get the following expressions for the S's in terms of C and the D's

$\mathrm{S7} = \bar{\mathrm{C}}\mathrm{D6}$

$\mathrm{Si} = \bar{\mathrm{C}}\mathrm{D}_{\mathrm{i}-1} + \mathrm{CD}_{\mathrm{i}+1}$ for i = 1,2,3,4,5,6

$\mathrm{S0} = \mathrm{CD1}$

A circuit that implements the above expressions is shown on page 164 of your textbook. Recall that shifting provide a simple means for carrying out multiplication by 2 (shift left) or division by 2 (shift right to get the quotient).

Designing the circuits for the next set of examples using truth tables and minimization techniques such as Karnaugh maps would be impractical. Instead we use more intuitive approach, similar to what we did for the shifter, based on what we know about how the circuit should work.

3. *Multiplexers*: A $2^n \times 1$ *multiplexer* receives input on 2^n input lines and transfers the value of one of these input lines to a single output line. The input line to be selected is determined from the bit combination of n additional input lines known as the *selection lines*.

Example: On page 161, Figure 3-12 (a) we have a graphic symbol for an 8-input multiplexer with 8 input lines D0 , D1 ,..., D7, one output line F, and 3 select lines A , B , C (although I would prefer S2, S1, S0 respectively).

- Multiplexers are available as integrated circuits.

A truth table for a 2 × 1 multiplexer, with associated expression for F would be the following

A	D1	D0	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$F = \bar{A}D0 + AD1$$

Representing a 4 × 1 multiplexer requires a 6-input truth table, which we do not want to work with, , but if we want to have the following associations between select values (A,B) and output F

```
(0,0) → F=D0, (0,1) → F=D1, (1,0) → F=D2, (1,1) → F=D3
```

A moment's refection should convince one of the validity of the following expression for representing F

$$F = \bar{A}\bar{B}D0 + \bar{A}BD1 + A\bar{B}D2 + ABD3$$

Similarly we would get the following expression for the output F of an 8 × 1 multiplexer

$$F = \bar{A}\bar{B}\bar{C}D0 + \bar{A}\bar{B}CD1 + \bar{A}B\bar{C}D2 + \bar{A}BCD3 + A\bar{B}\bar{C}D4 + A\bar{B}CD5 + AB\bar{C}D6 + ABCD7$$

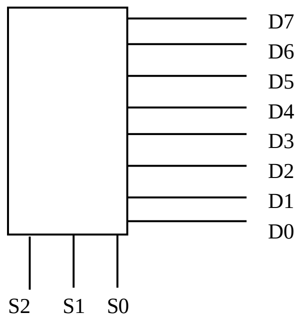
An implementation of this expression is given in Figure 3-11 of your textbook.

In the following example we show how Boolean functions can be implemented using multiplexers.

Example:

4. *Decoders*: An n × 2^n *decoder* receives input on n input lines A_0 ,..., A_{n-1} and by treating these signals as a binary number, sets the signal on exactly one of 2^n output lines to 1 (known as *selecting the line*), and makes the rest 0. We label the output lines as D_0 ,..., D_{2^n-1}.

Example: In a 3 × 8 decoder such as the one below, input signals of 1 0 1 might select line D_5 (generally one must use a data sheet to find the correspondence between given input signals and the line selected).



- Often an *enable input* is included on a decoder to control its operation. Here all of the outputs will be 0 (be *disabled*) if the enable input has one value, say 0, while if the enable input has the other value, in this case 1, the decoder will operate as expected (be *enabled*).

Examples of 3-to-8 decoders with enable.

Decoders are available as integrated circuits.

In the following example we show how to use two 2×4 decoders with enables to construct a 3×8 decoder.

Example: Done in Class

5. *Arithmetic logic units*: In this example we show how some of the previous circuits can be used to design a simple 1-bit arithmetic/logic unit (ALU). This ALU will be capable of performing the following operations with operands A and B:

$$A \text{ AND } B,\ A \text{ OR } B,\ \overline{B},\ A + B + \text{carry-in}$$

An operation is selected according to the following input signals on the select lines F0 and F1:

F1	F0	operation
0	0	A AND B
0	1	$\overline{B}$
1	0	A OR B
1	1	A + B + carry-in (produces carry-out)

If we represent this 1bit alu by the graphic symbol

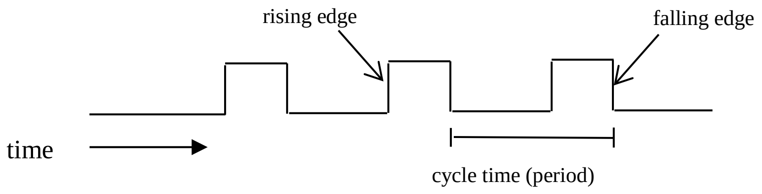
then we can combine eight such alu's to produce an 8-bit alu for operands $A_7 ... A_0$ and $B_7 ... B_0$ - a design technique known as *bit-slicing*. An example of bit-slicing is shown on page C-30 of your textbook.

Section 7. Basic Logic Circuits II - Sequential Circuits

Recall that a sequential circuit is a combinational circuit to which memory elements have been added. The output values of a sequential circuit are functions of both the input values and the current state of the memory elements.

Sequential circuits may be asynchronous or synchronous.

1. An *asynchronous sequential circuit* is one in which the circuit output depends on the order in which the input signals change; the changes can occur at any time.
2. A *synchronous sequential circuit* is one in which the memory elements can change values only at discrete instances of time.
    - Clock pulses are generated throughout the circuit so that changes can only take place upon the arrival of a pulse.
    - A clock is a circuit that emits a series of pulses (i.e. 0-1 and 1-0 state transitions) with a precise pulse length (width) and precise interval between consecutive pulses (*the clock cycle time*, or *period*).
    - The following diagram illustrates some of the common terminology used in association with clocks



    - The *frequency* of a clock is the reciprocal of the clock cycle time. The most common measure of frequency is the *hertz*, where 1 hertz = 1 cycle per second. Some example of frequencies are 500 MHz and 4 GHz, yielding cycle times of 2 nsec to .25 nsec respectively.

Memory Elements

By a "memory element" in a sequential circuit we mean a circuit component capable of maintaining a given value for an indefinite period of time. A typical memory element that stores a single bit will feature one or more inputs that allow one to change the value stored as well as at least one output to indicate the value stored. Often such memory elements will have a second output carrying the complement of the stored value.

A simple way to implement a memory element is through a "feedback" circuit such as the one on page 170 of your textbook.

To understand how this circuit works, let tp be the propagation delay for the NOR gate being used . Initially let us consider only those cases where the output lines labelled Q and $\bar{Q}$ do indeed have complementary values abut let's relabel the output lines to where z1↔Q and z2↔$\bar{Q}$. The following table summarizes the changes in value of z1 and z2 over time, starting at time a given time t0.

R	S	t0		t0+tp		t0+2tp		t0+3tp		comments
		z1↔Q	z2↔$\bar{Q}$	z1	z2	z1	z2	z1	z2	
0	**0**	**0**	**1**	0	1	0	1	0	1	stable
0	**0**	**1**	**0**	1	0	1	0	1	0	stable
0	**1**	**0**	**1**	0	0	1	0	1	0	stable Q=1
0	**1**	**1**	**0**	1	0	1	0	1	0	stable Q=1
1	**0**	**0**	**1**	0	1	0	1	0	1	stable Q = 0
1	**0**	**1**	**0**	0	0	0	1	0	1	stable Q = 0
1	**1**	**0**	**1**	0	0	0	0	0	0	stable* Q = 0
1	**1**	**1**	**0**	0	0	0	0	0	0	stable* Q = 0

*Notice in the last row, where R = S = 1 that while the values of z1 and z2 are stable, they are no longer complementary to each other. This will present a problem as we shall see shortly.

We summarize the above behavior more as follows:

R(t)	S(t)	Q(t)	Q^+=Q(t0+2tp)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	undesirable, it forces z1 =z2
1	1	1	

or more succinctly

R	S	Q^+
0	0	Q
0	1	1
1	0	0
1	1	undesirable

The memory element we described is for one known as an ***RS-latch.***

Types of memory elements - latched and flip-flops

One bit memory elements can be classified by two criteria:

1. Input conditions that change the value stored.
2. Control signals necessary for the input conditions to take effect.

Let us consider this second criterion first. Here there are three ways to control the time when input conditions are allowed to change the state of a memory element:

1. Any time the input signals change they “immediately” alter the stored value to its appropriate new value (here “immediately” means up to a delay, known as a *propagation delay*, which reflects the time it takes the input signals to work their way through the components of the memory element to produce and store the proper new value). Here there is no control per se over the input conditions.
2. Any time a special control signal is active (high or low, depending on the design of the memory element) the state of the memory element can be changed via its inputs. The control signal is variously known as the *strobe*, *enable*, or *clock pulse*.
3. Any time the control signal changes signal levels appropriately, that is, either from low to high (*positive edge-triggered*) or, alternatively, from high to low (*negative edge-triggered*).

We shall use the term *latch* for those memory elements whose control signals operate as in 1. or 2. above, and the term *flip-flop* for memory elements which behave as in 3.

- Note, while widely used, this terminology is by no means universal. One can pick up textbooks and find references to memory elements described by such terms as “level-triggered flip-flops,” or “level-triggered latches.” Some authors refer to any memory element which uses an enabling signal as a “flip-flop,” leaving the term “latch” only for those which respond according to the first characteristic above. Rarely does this confusion in terminology cause problems.

- There is a special type of flip-flop known as a *master-slave* flip-flop which specifies not only when input signals are captured, but also when output signals take effect. We take up master-slave flip-flops in more detail when we discuss how to implement various types of latches and flip-flops.

Now let us examine some of the commonly used types of input signals used with latches/flip-flops and how these inputs determine the new state of the memory element. For simplicity we initially restrict our discussion to latches.

1. The *RS-latch* (for reset-set, but also known as the SR latch), whose implementation we just saw, has two inputs, R and S, and changes its current value Q, to its new value Q+ as shown in the following table (known as its *characteristic table*).

R	S	Q+
0	0	Q
0	1	1
1	0	0
1	1	not allowed

To see why R=1, S=1 is not allowed for an RS-latch, first note that it puts the latch in a stable state, but with z1 = z2 = 0. By itself, this causes no problems, but if we later set R = S = 0, which is the "resting" state for the latch (since when z1 and z2 are complementary these values are retained indefinitely) it causes these values to oscillate between 0 and 1 as the following table shows

R	S	t0		t0+tp		t0+2tp		t0+3tp		comments
		z1	z2	z1	z2	z1	z2	z1	z2	
0	0	0	0	1	1	0	0	1	1	unstable

As we shall see in our discussion of implementations, the RS latch can be used to implement the other types of latches.

When one assigns the value “1” to a latch (or flip-flop) the latch is said to be *set*, while giving the latch the value “0” is known as *reset*ting or *clear*ing the latch, hence the use of the identifiers S and R for this latch.

2. The *JK-latch* is similar to the RS-latch except the unstable condition is avoided.

J	K	Q+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

3. The *D-latch* (for “data”) has only one data input, which is called D.

D	Q+
0	0
1	1

Implementation of a Memory Elements

Clocked Latches

Most applications require that the time at which a latch is set or reset be controlled more precisely via an enabling input. Such an input can be incorporated into that of the basic latch as shown on page 171.

Note that any changes that occur in the R and S inputs to a clocked RS latch while the clock pulse is high will be followed "immediately" (up to propagation delay) by the appropriate change in the output signal.

Implementation of D Latches and JK Latches

As we noted when we introduced the *D latch* it accepts a single input signal, D, which causes the output signal, Q, to have the same value as D. An implementation for the D latch, also using an RS-latch implementation, is given on page 172. The *JK latch* is similar to the RS latch-flop (with inputs J and K acting as the set and reset lines respectively), except the inputs J=1 and K=1 cause the latch to change state (making it more useful). The following diagram shows how to implement a JK latch using an RS-latch (done in class).

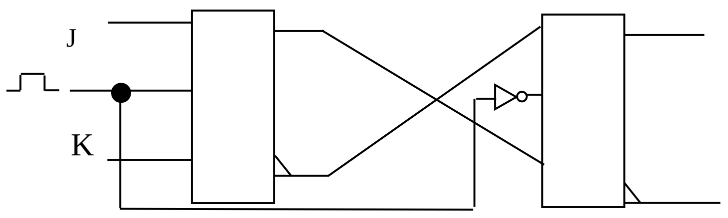
Edge-Triggered Flip-Flops and Master-Slave Flip-Flops

In some circuits the immediate changes in output that follow changes in input signals in latches cannot be allowed, especially in those where the value of Q at a time t0 + t depends on the value of Q at t0 (we will see examples where this arises later). Two types of memory circuitry which provide more control over times when they change states are the *edge-triggered flip-flop* and the *master-slave flip-flop*.

For edge-triggering, additional circuitry is placed at the clock input of a clocked latch so that a prescribed edge of the input pulse, either the rising edge or the falling edge, causes a brief pulse to be generated (see page 173)

This brief time when the latch is enabled allows its inputs to be accepted, with the appropriate output signal then following.

In the master-slave flip-flop, two clocked-latches (a master and a slave) are organized as follows:



Under this arrangement, prior to the rising edge of a clock pulse the J and K inputs should have reached the desired signal levels. While the *clock* is 1 the master latch will assume the appropriate next state while the slave latch maintains the old output value. When the falling edge of the pulse arrives the new state will be transferred to the slave latch.

Registers

While occasionally latches or flip-flops are used by themselves, it is more typical that a group of flip-flops acts together in a coordinated manner. Such a grouping is known as a *register*. More precisely, a *register* is a collection of flip-flops together with combinational circuitry to control when and how information is transferred to the individual flip-flops. One of the simplest schemes is the parallel load register.

Example: an eight-bit register with parallel load (see textbook, page 175) The register is called a *parallel load* register because all the register's bits can be loaded simultaneously.

As we noted above, the example of the register just given is one of the simplest types of registers, and is to the flip-flop what a word gate is to a logic gate.

In order to design more complex registers that we have seen here we will need a systematic method for designing sequential circuits.

Design Procedure for Sequential Circuits (this is not in your textbook)

1. **Draw a state diagram from the problem statement**. A *state diagram* a graphic method that uses circles and arcs between circles to present the output values of the circuit and the next states of the memory elements as functions of the inputs and present states. Here
   - circle = a state, with state of flip-flops A and B represented inside by the binary number AB
   - arc = a state transition; the notation, I/O, by the arc gives the input value(s), I, that caused the state change and the resulting output value(s), O.

   We will see examples of state diagrams shortly when we actually design some sequential circuits.
2. **Derive a state table from the state diagram**. A *state table* for a sequential circuit is a table which presents the outputs of the circuit and the next states of the flip-flops as functions of the inputs and present states. A state diagram is a graphic method for representing the information in a state table.
3. **Determine the type of latch/flip-flops to be used in the circuit and for each latch/flip-flop determine the value each input of the latch/flip-flop must have to cause the necessary change in state**.
4. **Derive an input equation for each flip-flop input and the circuit output equations**.
5. **Draw the circuit diagram.**

The derivation of the input tables for each flip-flop in the circuit can be simplified by using an *excitation table* for the class(es) of flip-flop(s) being used. This is a table which gives the input signals needed to change the latch from one state to another. It is derived from the characteristic table for the latch.

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

JK flip-flop

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

D flip-flop

Excitation Tables for JK and D flip-flops
(X = "don't care;" can use 0 or 1)

Examples:

1. A ***serial (binary) adder*** is a circuit which accepts two numbers as input in serial (i.e. bit-by-bit) and generates their binary sum, which is sent out serially. We now apply the design principles for sequential circuits to design a serial adder using JK flip-flops.

2. A ***modulo-n synchronous counter*** is a register that is often used to keep track of the number of times an event occurs, with the occurrence of the event being signaled by an input value of 1 and the state of the counter updated accordingly to reflect the occurrence of the event. Upon the occurrence of the nth event, an output value of 1 is generated and the counter reset to indicate 0 events; in all other cases the output value is 0. Design a modulo-4 counter using JK flipflops.

3. A ***shift-right register*** is a register capable of shifting its binary information the right by one bit upon the arrival of a signal 1, while storing the value 0 in the leftmost bit; otherwise the values of the bits remain unchanged. There is no output value. Design a 3-bit shift-right register using D flip-flops.

4. A ***sequence recognizer*** is a circuit designed to generate an output value of 1 when the last bit in an n-bit sequence of input values has arrived, and to generate an output value of 0 otherwise. The sequences may be overlapping. Use D flip flops to implement a sequence recognizer for the sequence 0,1,0,0. Here assume the order of arrival of the bits is from left-to-right.

   Example: the sequence of input values 1,0,0,1,0,0,1,0,0,1 would produce the output sequence: 0,0,0,0,0,1,0,0,1,0