15

PRINCIPLES OF LOGIC DESIGN

W. Richards Adrion

James H . Herzog

Robert A. Short

PRINCIPLES OF LOGIC DESIGN

W. Richards Adrion
James H. Herzog
Robert A. Short

Oregon State University
Corvallis, Oregon

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

1 Logic Design and Digital Machines

This study involves logic design and switching theory, in particular their practical application to the logic design and understanding of digital machines. Digital machines, of course, play an extremely important role in that large class of machines known as digital computers. But they also play an important role in many other kinds of practical devices important in the design of communications systems, digital control systems, counters, registers, digital meters, and so on.

The basic content of switching theory is very simple. It embodies that body of machines and machine behavior that can be realized with "switches", things that are either "on" or "off", and nothing, really, could be much simpler than that. Of course the world is really comprised of very many complex structures which are really composed of exceedingly simple lesser structures, so that we really shouldn't be too surprised that even though the elements of switching theory are quite simple, their consequences are not necessarily so.

The goals of our study are several, and include at least the following:

1) to develop some understanding and capability in using the techniques, design procedures, and models that have been developed for understanding and designing digital networks;

2) to explore in some modest detail the kinds of questions with which logic designers and practitioners concern themselves;

3) to develop an appreciation for the tremendous variation possible in digital design requirements and specifications, i.e., for the complexity of the 'finite' digital problem, and hence an understanding of the need for systematic design techniques by which to attack such problems;

4) to gain some practice with the fundamental tools and techniques of logic design so that the reader can adapt the techniques to the "new" problem presented by his own particular design constraints; and

5) to provide an introduction to the literature so that the discerning student can, in the future, dip into the ever growing literature in the field, and find it to some degree comprehensible, and advantageous to use.

1.1 The Basic Digital Design Problem

As an abstract statement, logic design, representing the practical applications of switching theory, is that formal discipline concerned with the analysis and synthesis techniques that are appropriate for the design of digital networks. Of course that is almost like saying that "Computer Science is that discipline involved with the study of computers". It's very true, but not at all illuminating. So let's be a little more explicit -- at least insofar as agreeing on what we mean by a digital network.

A digital network is an assemblage of digital elements. At least that divides our problem. And we can quickly agree that by assemblage we mean any arbitrary interconnection of digital elements. So that leaves everything really dependent on our agreement on what digital element is. Here we have one of three paths that we can follow. The first two are valueless, although frequently used. The first calls on circularity which is the basis for most dictionary definitions:

a) a digital element is the kind of thing of which digital networks are composed.

The second calls on authoritarianism:

b) everybody knows what a digital element is, it is a thing that assumes one of two possible output states as a function of its inputs, including possibly its own output

and that usually ends the whole discussion right there. Actually everything is contained in that one, although it's a little obtuse and hard to get any essential features out of it. Instead we'll take the third alternative and simply define what we mean by a digital element.

c) a digital element is one of two kinds of things: it is either
   i) a thing which is capable of exhibiting one of two outputs, and it does so as an instantaneous, deterministic function of its inputs, or it is
   ii) a thing which is capable of exhibiting one of two outputs, and it does so strictly as a function of what its input was an agreed (synchronous) moment ago.

This last definition we shall accept and use as our operative definition, for it divides switching theory into the two halves with which we shall be concerned:

combinational switching theory versus sequential switching theory. For the first kind of element indicated above is assumed to develop its output strictly as a function of its present inputs. Time is not normally considered to play a role in its operation. This is, of course, a simplifying assumption, but is appropriate for an adequately large range of devices. A basic assumption implicit in the above is that the number of possible inputs is finite, and that the number of values of each is likewise finite. The specification of a particular device, then, reduces to a determination of which combination of inputs produces a particular output, hence the term combinational networks.

The second kind of element inferred above introduces the final full range of complexity that enables us to describe and characterize everything of which digital machines are capable. For something that simply reproduces its own input an agreed time later introduces the notion of controlled time into the picture. It doesn't take a great leap of imagination to presume that it might be useful in some instances to take advantage of that time lapse to return the element's output to its own input in some way, so that its output can indeed be a function of its own previous outputs, i.e., of the past history of the element. This basic element is termed a simple delay element. It is the epitome of "memory" in machines and brings us to the larger class of networks known as sequential networks or sequential machines.

At first it might seem that we have made a rather severe set of restrictions in defining our baisc memory element as a simple delay. It can be shown, however, that the simple delay generalizes so that it can model any other kind of memory element, hence that the restriction in no way limits that kind of behavior we can observe in our resulting machines. And we shall quickly develop transformation procedures for moving from any of the machine types to any other.

Another assumption that we shall generally make is one that is satisfying from an engineering point of view, and furthermore is one that essentially delimits switching theory from the broader class of activity embraced by the term automata theory. This assumption refers to the finiteness that we shall presume regarding everything to be realized physically. Not only shall we usually assume a finite number of digital elements in any network, but also that the number of inputs is finite as well. Because of this finiteness of the number of digital elements we can conclude that the number of states of any machine is also finite, as well as the number of different outputs that are possible. Actually we have really said the same thing twice there because the states of a machine will simply refer to the different number of possible combinations on those outputs of the memory elements which are used internally in the machine itself. We shall make all these terms more precise as we need them. We note at present, however, that the usefulness of this concept of the state of a machine leads directly to the use of the term finite state machines to describe the kinds of digital machines with which we are concerned.

Examples of digital elements are many, and we shall point to only a few to make firm the kinds of things we are talking about. A familiar example is the relay coil and contact which schematically can be shown as in Figure 1.1.1.

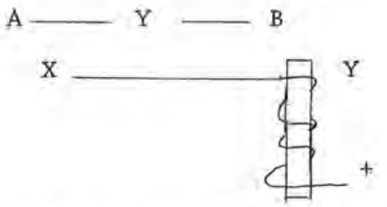


Figure 1.1.1. Simple relay machine

wherein some variable x (a two-valued variable: either ground or not-ground) determines whether the relay coil Y is energized or not. Of course relays have contacts, and this one is a "make" contact (i.e., it closes when the coil is energized) labeled y. Clearly we can refer also to a useful transmission function that characterizes the connectivity between terminals A and B of the contact network. It is either shorted or open as a function of the variable x. We note that this transmission function is also two-valued. (We also note in passing that this sort of schematic diagram is called a "detached schematic", i.e., the contact related to a certain coil need not be drawn in proximity to the coil itself. This obvious convenience makes the schematics of some quite complex contact networks easily drawn -- but an historically interesting note is that it wasn't until the mid-fifties that this obvious graphical simplification was taken advantage of. Prior to this time the insistence that each contact had to be shown next to its coil led to horrendously complex diagrams that were exceedingly difficult to analyze.)

It is then easy to indicate assemblages of such contacts such as in Figure 1.1.2 which defines a transmission function T which is two-valued (being shorted if

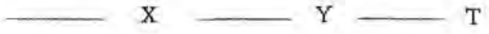


Figure 1. 1. 2. A contact network.

and only if both coils X and Y are energized), or like that in Figure 1. 1. 3 for

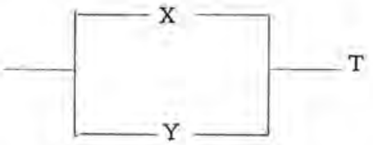


Figure 1. 1. 3. Altenative contact network.

which the function T is shorted if and only if either of the coils X or Y is energized. Clearly such transmission functions can become almost arbitrarily complex. For now we note only the possibility that the energization of a particular coil might be a function of itself. A simple example is shown in Figure 1. 1. 4 in which X is

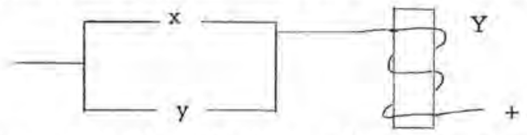


Figure 1. 1. 4. History-dependent relay network.

energized if either x or y is shorted, and of course once y is shorted then the coil is energized for all future time regardless of subsequent changes in x. Thus this network does react as a function of its own history, and verifies that a network need not be simply a (combinatorial) function of its own inputs. Thus this network remembers and this facility is only possible when feedback paths exist within the network.

Another example of a digital element is the cryotron of Figure 1. 1. 5 which

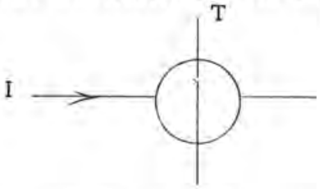


Figure 1. 1. 5. Cryotron element.

is a (super cooled) device wherein the transmission function T is a short circuit whenever the current I is nonzero. Thus the cryotron is essentially the same "kind" of element as a relay contact; both are examples of a broader class of things referred to as branch elements, and assemblages of such are referred to as branch networks. The essential feature is that the two-valued function involved is basically a transmission function that describes the condition of connectivity throughout the network.

This is in opposition to the familiar gate networks exemplified by electronic gate elements, realized by semiconductors or tubes usually, and symbolized by such diagrams as in Figure 1.1.6

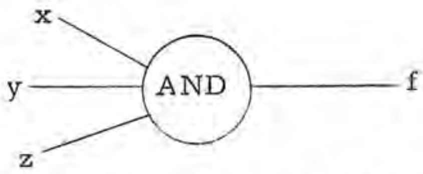


Figure 1.1.6. Gate element.

in which the "output" function f is at a high (or low) voltage if and only if inputs x and y and z are all at a high (or low) voltage. Of course these (as well as many other different logic gate types) can be interconnected in a great many ways so that functions of functions can be built up to any desired complexity.

We could mention any number of other kinds of digital elements such as magnetic cores, fluidic valves, mechanical linkages, mechanical switches, storage spots on electrostatic devices, etc., etc., but this would not further serve our present purposes.

The point is, to return to our original train of thought, that a digital network or machine is simply a collection of these things, suitably interconnected, usually with certain variables which we control ("inputs" or "independent variables") and certain variables which we wish to produce ("outputs" or "dependent variables"). The basic engineering problem, then, is the consideration of the transfer network N of Figure 1.1.7

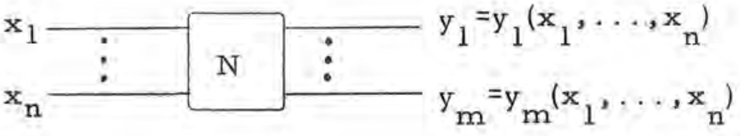


Figure 1.1.7. Basic digital machine model

so that we can produce a prescribed set of digital outputs as functions of the digital input. In these terms, the analysis problem is to determine the y_i's when given the x_i's and N. The synthesis problem is to determine an appropriate N when given the x_i's and the prescribed dependent y_i's.

This puts our problems into the desired engineering context, and it remains only for us to fill in the details. This is the function of the sequel.

1.2 Number Systems and Binary Encoding

A salient characteristic of digital networks that are realizable is that they are finite in all respects. It follows that the action of any digital machine is describable as being in one of a set of "conditions" or "states" corresponding to the particular values assumed by each of the elements of which they are composed. With the passage of time the machine perambulates through a set of such states as it responds to its input changes, as well as to autonomous changes in its own internal variables. Since the set of such states must be finite, as must the allowable set of inputs, it follows that a sufficient description of a machine is some sort of listing of one "number" followed by another, each number related one-for-one with the particular state it represents. Now in fact these numbers might be the natural numbers, in which case we are concerned with conventional arithmetic and arithmetic operations. Or they might be simply marks or symbols in another area of discourse, in which case we speak of non-numeric processing.

For openers, then, we shall briefly review some of the salient facts concerning numbers and their representations. We shall also consider assuring facts about the efficiency of bases, and we shall pay a passing glance at the propositional calculus, which is the historical antecedent to the switching algebra.

1.2.1 Positional or Polynomial Encoding

As a starter, then, we point to the familiar fact that a most useful way of representing a number is the one we use all the time--the so-called polynomial representation

$$N = \sum_{i=-m}^{n} d_i r^i$$

$$= d_n r^n + d_{n-1} r^{n-1} + \ldots + d_{-m} r^{-m}$$

or more commonly written as

$$N = (d_n d_{n-1} \ldots d_{-m})_r$$

where we also conventionally omit the r and the parentheses when the r is clear in context. We shall make little more of this subject as some familiarity will be assumed. The r is of course the base, usually a positive integer greater than 1 (but not necessarily) and the d_i refer to the digits, and these are also usually the integers such that $0 \leqslant d_i < r$.

Thus, for example,

$$(123)_6$$

represents a number to the base 6, while

$$(51)_{10}$$

is evidently a number to the familiar base 10. That they are in fact the same number points to the necessity for conversions from one base to another. Since every number is representable in the form

$$(N)_r$$

for appropriate r, it follows that there must always exist transformations from one base to another. In fact there are pretty algorithms for doing so in an impressive way, but for our purposes straightforward, "brute-force" methods will suffice. Thus

$$(123)_6 = 1(6^2) + 2(6^1) + 3(6^0) = 36 = 12 + 3 = (51)_{10}$$

while the fact that

$$(109)_{10} = 1(2^6) + 1(2^5) + 1(2^3) + 1(2^0) = (1101101)_2$$

can be affirmed simply by subtracting out the largest possible powers of 2 at each step in a successive, systematic sequence of subtractions. Thus

$$\begin{array}{r} 109 \\ \underline{\quad 64} \\ 45 \\ \underline{\quad 32} \\ 13 \\ \underline{\quad 8} \\ 5 \\ \underline{\quad 4} \\ 1 \end{array}$$

will develop the result above. The reader will be left to further develop these techniques in ways most compatible to his liking.

Similarly we shall not go into details of the arithmetics involved for the various bases, and a basic familiarity with the techniques involved is assumed.

As with base 10 arithmetic it suffices to agree on addition and multiplication tables. For base 2 these are so simple that we can quickly construct them as an example:

Addition x + y

x \ y	0	1
0	0	1
1	1	0

Sum

x \ y	0	1
0	0	0
1	0	1

Carry

Multiplication xy

x \ y	0	1
0	0	0
1	0	1

e.g.,

$$\begin{array}{r l}
1101 & = 13 \\
\underline{+1111} & = \underline{11} \\
11000 & \ 24
\end{array}
\qquad\qquad
\begin{array}{r l}
1011 & = 11 \\
\underline{\times 1001} & = \underline{\ 9} \\
1011 & \\
\underline{1011} & \\
1100011 & \ 99
\end{array}$$

with subtraction and division being analogously defined. These are straightforward augmentations and will not be detailed here.

Now this is all well and good, but several questions remain. In the first place we have affirmed a multitude of representations for numbers, and we have agreed that "number" in some sense must involve everything that finite state machines can possibly be about. But a moment's reflection will reveal that we really must say much more about such representations for we have described an infinity of representable numbers, whereas the salient fact about finite state machines is that we can only accommodate a finite set of things, hence in any given context can only represent a finite subset of the natural numbers.

Once we recognize the possibility that we can talk only about a finite set of the numbers in any given context, then we are faced with the annoying task of deciding just which finite set we shall talk about. This involves the notion of selecting the particular encoding of the numbers that we shall use, and we shall verify that indeed a very large number of different possible codes are possible. We shall look briefly at but a few of them, and the salient point we shall try to make is that different codes are used to enhance different features of the machine, but that furthermore each different code requires a re-examination of the computational algorithms involved, and that things are not exactly as simple and straightforward as the tables above would indicate.

First we shall affirm in a simple way that the base 2 is indeed one of the most natural bases for machines, arguing from the standpoint of efficiency.

1. 2. 2 Efficiency of Bases

Although everybody knows that base 2 is most commonly used in machines, it is worth noting that other representations have also been used. Ternary logic machines (base 3) have been widely discussed in the literature, and some machines have been built, principally to capitalize on the error-correcting and detecting features that are possible with such encodings. Base 8 frequently occurs in representations, principally because of the efficiency of packing numbers into base representations that are a power of two.

Decimal machines have also been built, more so in the early days, and the principal motivation here has been the necessity for human communication with the machine.

It must be admitted, however, that most of these nonbinary representations were really binary when looked at at a lower level than the numbers themselves, e.g., the decimal number 9, represented by a binary four-tuple 1001 looks strangely binary, but the point is that the algorithms for handling such "decimal" numbers were decimal algorithms, not binary algorithms. Thus though "9" might be represented by the four-tuple above, "10" might look like 0001 0000, which is strangely "unbinary", and the addition, for example, of two such decimal numbers involves strictly decimal rules for addition. Also there have been other direct representations of the nonbinary bases. Parametrons, for example, turn out to exhibit three states rather than two, and have been the stimulus for much of the ternary logic development. Even base ten elements have been used, principally in relay machines, however, where the element involved is such as a ten-position stepping switch, or the like. Electronic elements with ten stably discernible states are not easy to come by, however, hence direct representations of nonbinary bases have been few and far between.

Given the fact that radix r will suffice to represent any number, it is nonetheless interesting to ask whether there are any good engineering reasons for preferring one base over another. We have mentioned the physical fact that things like semiconductors, tubes, cores, cryotrons, etc., etc., operate most reliably when switched between one of two states (e.g., "saturated" versus "unsaturated", or "ON" versus "OFF"). These are compelling engineering reasons for naturally favoring the base two in machine realizations. But these are not entirely convincing since a number of discernible levels could be carefully engineered appropriate for some larger radix if we care enough, i.e., if it costs us enough not to do so.

To this end let us complete a modest, brief sort of engineering cost analysis that will suggest the appropriateness of bases for number representations from a little different sort of argument.

Since we are always talking about a finite representation, we shall suppose that the maximum number of numbers to be represented by our machine is some prescribed M. We shall represent these numbers with some n digits, with each digit capable of representing digit d to the base r. Clearly as r grows large, n will diminish, and conversely, but this is reasonable since M bounds the size of computational task we are able to do, and it is surely this computational task that provides the prime motivation for our design in the first place.

Of course as r grows larger it is reasonable to expect that the "cost" of realizing a single digit will grow as well. Certainly it is going to cost more to realize a ternary digit than a binary digit, or a decimal digit than either. As a reasonable and explicit engineering assumption, then, we shall suppose that the cost of realization of a digit is proportional to the base selected. Thus if number N is represented as

$$N = \Sigma d_i r^i$$

then the cost c of representing a single digit is

$$c = K_1 r.$$

For our total number of n digits, then, the total cost C is

$$C = K_1 nr.$$

This gives our total cost C as a function of two variables that are related, namely n and r. If we can specify another relation between them, then we ought to be able to eliminate one in our cost equation, and get our total cost in terms of one of them, in particular in terms of r alone. But this relation is given by our preselected constant M which represents the size of our computational task. Clearly for n digits in base r we have

$$M = r^n$$

and since M is a constant, then so also is ln M, so we can solve for n as

$$\ln M = n \ln r = K_2$$

or

$$n = \frac{K_2}{\ln r}$$

Substituting this expression for n into our original formula for total cost we get

$$C = K_1 nr = K_1 K_2 \frac{r}{\ln r} = K \frac{r}{\ln r}$$

expressing our total cost strictly as a function of r, the radix we choose.

Of course we're now in a position to plot our results as in Figure 1.2.1. Differentiating to detect the minimum we get

$$\frac{dC}{dr} = K \frac{\ln r - r \frac{1}{r}}{(\ln r)^2} = 0$$

$$\ln r = 1$$

$$r = e = 2.712...$$

as the optimal value of radix from these cost arguments.

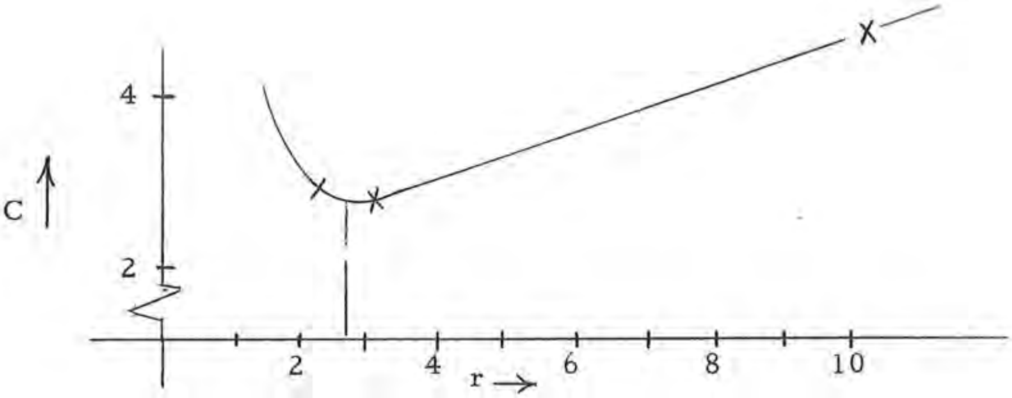


Figure 1.2.1. Cost as a function of radix.

Of course an irrational radix is a little hard to realize with physical elements, but the obvious conclusion is that base 3 is optimum from this point of view, base 2 is next best, base 4 is pretty good. Base 10, on the other hand, is about twice as expensive to realize as base 2.

Obviously this argument depends upon the original cost assumption, and the reader is invited to insert his own assumption based upon more particular relevance to the technology he might be considering. Probably the cost per digit is something more than just proportional, i.e., it probably costs considerably more to get a base 4 digit than simply twice as much as for a base two digit, especially when considerations of equal reliability, tolerance on element parameters, etc., are included. But at least to this point we have buttressed from a little different analytic point of view our almost complete preoccupation in the following with things that are two-valued.

1.2.3 Binary Codes

Probably the first point to be made about codes and coding is their all pervasiveness in all of our communication activities. There is almost literally nothing we can do about avoiding the use of codes in one form or another, whether we have reference to artificial or natural intelligence, hence it follows that it is to our best interest to study them intelligently, and with an awareness of their capabilities so that we can maximize their application to our own best interests.

Even in our cultural activities most of our communication is via codes of one sort or another, especially is there if much structure involved. For example, even in art a great deal of the form, proportion, ratios of vertical to horizontal lines, color schemes, etc., involve the acceptance of certain patterns which we accept as culturally good. Only in some of the recent abstract paintings could we agree that no coding is involved, but rather a direct appeal to primal feelings. Certainly much of our musical symbolism involves coding -- if this does not seem evident compare the music from two very different cultures, e.g., country Irish, say, versus Indian, and it is apparent codes are quite different from one another, and probably mutually unintelligible.

Of course we are interested in codes for the transfer of information of a more pristine sort. In these applications codes have been designed for two fundamental purposes: either to achieve secrecy, or to match the information to the particular vagaries of the communication channel. While a great deal has been written about the former, we shall be interested only in the latter. In the context of machines, the character of codes in adapting to the "channel" involved is molded by at least three considerations:

1) the physical characteristics of the channel (in the case of machines this results in codes being almost exclusively binary codes, and if not directly so, then at least binary at the next level, e.g., as in the binary-coded decimal codes);
2) the noise characteristics of the channel (in the case of machines this results in the design of codes that have various error detection and correction properties); and
3) the particular operation involved (in the case of machines this points attention to codes that have particular algorithmic simplifications perhaps, or that speed up addition, or in any other way accomplish the computational task better).

Even within the context of codes based upon binary representations the number of possibilities is literally limitless. We shall consider briefly only the well-known binary-coded decimal codes at this point, and shall see that even this restricted set presents a very large number of possibilities. Our present purpose is to define the codes, and to illustrate that the algorithmic properties involved are very code sensitive, i.e., that even such a simple algorithm as that for addition must be re-invented for each different code representation. We shall leave further examination of special codes for error detection and correction for a later chapter.

By far the most common types of binary encodings involve the binary-coded decimal codes and the name given to these codes reveals the purpose. We wish to encode the decimal digits in binary form. Clearly at least four bits are required in order to represent the decimal digits, but four bits yields 16 different characters which can be assigned to the ten decimal digits, and this can be done in a remarkably large number of different ways. The natural tendency (only because it's easy for us to read and to remember) is to assign the digits so that the four-tuples can be read in the natural binary ordering; thus

Digit	Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

so that, for example, decimal 927 would be represented as 1001 0010 0111, and all this is very simple. But notice we could have made the assignment of code character to decimal digits in any one of

$$(16)\,(15)\,(14)\cdots(7) = \frac{16!}{6!} \doteqdot 3\,(10^{10})$$

different ways, which truly leaves a lot of options. All of these would be binary-coded decimal (BCD) codes, and each would potentially yield different computational algorithms.

Of the BCD codes there are some that possess the property that there exists a set of numbers n_1, n_2, n_3, n_4 such that all ten decimal digits can be represented as

$$w_1 n_1 + w_2 n_2 + w_3 n_3 + w_4 n_4$$

for some set of $w_i = 0, 1$. The codes of this type are called the weighted codes and the n are called the weights. The code given above is such a code with weights of 8, 4, 2, 1 (since it corresponds with the polynomial binary representation), hence is sometimes called the 8421 BCD code. There are many such weighted codes, and all have the advantage of computational simplicity which accrues from being associated with the particular set of known weights. (Not all the weights need be positive either, nor unique from one another. For example there is a 6421 BCD code: 8 is coded as 1010, 9 as 1011. Many others exist as well, but we shall not concern ourselves further with them here.)

Of course if we don't limit ourselves to the efficiency of four-tuples for our representation, then the number of possibilities is literally boundless. For example a well known code based on five-tuples is the 2-out-of-5 code

Digit	2-out-of-5
0	00011
1	00101
2	00110
3	01001
4	01010
5	01100
6	10001
7	10010
8	10100
9	11000

which is a nice "packing" of the possibilities since this exhausts all the five-tuples with exactly two 1's since there are just 10 possible. What are the advantages of such a code? For one thing it is an example of a single-error-detecting code. That is any error in a single bit must transform the "word" into another which is not in the code. If we designate the weight of a word as the number of 1's contained therein, then the 2-out-of-5 code is a code of weight 2. Any single error must transform the word into one of weight either 3 or 1, hence any such error is detectable simply by counting the number of ones. (Obviously the well known "parity check" code possesses a similar error detecting capability. It is formed from the 8421 BCD code by adding a fifth bit determined so that the total number of 1's will be even; thus

Digit	Parity Code
0	00000
1	00011
2	00101
3	00110
4	01001
5	01010
6	01100
7	01111
8	10001
9	10010

and the occurrence of any single error will transform the parity code work into one of odd weight, hence into a detectable error.) The 2-out-of-5 code has one other important characteristic, however, and that is the constant weight-2 words. This means that whatever "drives" the ones does so with a constant "load" on the drivers. If this is important, then so is the code. An important application is in representing the decimal digits on punched paper tape. Not only is the load on the punch drivers constant, but so are the stress points on the tape so that no digit is apt to "tear" more than any other.

By now it should be clear that the number of possibilities is truly without limit, and one cuts and tailors to meet the particular application. Certainly for every code,a complementary code exists wherein 1's and 0's are replaced with one another. In most cases this would be an irrelevant change. But consider the complementary code to the 2-out-of-5 code. If the particular environment were such that 1's were less error prone than 0's (an unsymmetric channel) then clearly the complementary 2-out-of-5 code would be less error prone than the direct code, i.e., the "message" would get through correctly more often.

Of course if we are really worried about correcting errors, direct replication codes are sometimes used. For example, suppose that we simply replicate each bit position in the 8421 BCD code. Thus, for example, 6 would be encoded as 00111100, resulting in an 8 bit code word. Clearly if an error occurs in any single position we can detect it. It seems that this is not a useful thing to do in this case since we've already indicated the even parity code which will also detect an error in any single position. But we need to be careful. If detection of a single error is our only concern, then our observation is sound. But notice that the replicated code will do much more. It will detect an error in every original bit position, and furthermore will tell us exactly in which bit positions the errors occurred -- i. e., it is error locating as well as error detecting. Depending on the context, this might be important information as well. A conceivable response to the detection of an error is to request a retransmission or recomputation. Depending on the probability of error it may indeed be necessary to limit the amount of recomputation in order to get any useful work done at all.

Further replication continues to provide greater error capabilities. For example in a triply replicated 8421 BCD code, if 001101111010 is received we can quickly decode it as the decimal 6, since the code not only exhibits error detecting capabilities, but also can correct a single error in every original bit position. Thus even though three errors were present in the word, as above, we could correctly decode it. Caution is necessary though, and the use of codes is always fraught with dire consequences if the code is not able to adequately cope with the error producing characteristics of the environment. Thus if two errors had actually occurred in the last (original) bit position in the above example, then we would not only decode the word as a 7, but furthermore we would not even be aware that an error had occurred. (All is not lost even in this case for there are also useful codes called "burst" codes which tend to be good for errors that occur in clusters, e. g., as might be the usual event in the presence of lightning flashes, or other long duration but infrequent disturbances, but these are far beyond the scope of our present considerations.)

Rather, let us now turn to the question of the dependence of arithmetic algorithms (or any other algorithm for that matter) on the particular encoding that is used. This is not an evident thing, and we shall indicate that differences exist by pointing only to a very simple example. For it would seem clear that if it is the decimal numbers that are represented, then the addition algorithm, for example, should of course be the decimal addition table, with suitable allowances for occasional carrys of course. But this is only true at the level of the decimal character itself, and is not true at all at the level of the binary representation. If we are to build an adder, it is exactly at that binary level at which we must operate. The reader who is unconvinced at this point should address himself to the logic design of an adder where the two addends are decimal characters, but represented in terms of the 2-out-of-5 code. That code is well adapted to paper tape applications, but is definitely inappropriate if addition is what we have in mind. In fact the appropriate answer when asked to derive an algorithm for addition using the 2-out-of-5 code is:

1) convert all characters to the 8421 BCD code
2) use an appropriate 8421 BCD code algorithm for addition
3) convert the sum back to the 2-out-of-5 code

This is an eminently practical way of avoiding the real meaning of the question in the first place, and reaffirms the original point; the algorithm for one code is not the same as the algorithm for another.

To illustrate this in a simple example, let us compare the addition algorithms for the well known 8421 BCD with another variant of the BCD codes known as the excess-three, or XS3 code. Both are shown below for comparison purposes.

Binary	Decimal Character	
4-tuple	8421	XS3
0000	0	
0001	1	
0010	2	
0011	3	0
0100	4	1
0101	5	2
0110	6	3
0111	7	4
1000	8	5
1001	9	6
1010		7
1011		8
1100		9
1101		
1110		
1111		

and as we've noted before the original decimal characters can be assigned to only a subset of the sexteen possible four-tuples. The origin of the name excess-three should be clear--each BCD representation is simply the 8421 code "plus 3". Thus

4-tuples which have no assignment are known as forbidden characters since some special allowance must be made for them when they occur. To derive our 8421 algorithm, let us proceed by example. Suppose we add, say 5 + 2 in decimal. In the 8421 we confirm

$$\begin{array}{rr} 5 = & 0101 \\ \underline{2} = & \underline{0010} \\ 7 & 0111 \end{array}$$

and we perceive no difficulty since the code 0111 correctly denotes the proper decimal character. Now try 6 + 5:

$$\begin{array}{rr} 6 = & 0110 \\ \underline{5} = & \underline{0101} \\ 11 & 1011 \end{array}$$

which is correctly determined as "11" of course, but the machine doesn't know that, for "11" is not one of the decimal characters it has been taught to assign to each decimal position. The proper path to resolve the problem is to augment our algorithm so that whenever a "forbidden" character appears, and "11" is one of them, then we should "add 6" to the result in order to pass completely over the forbidden range. Of course a carry will be produced in this process, but that is the problem of the next stage. This gives 1 0001 as the correct result.

Finally, let's try something like 9 + 8:

$$\begin{array}{rr} 9 = & 1001 \\ \underline{8} = & \underline{\ \ 1000} \\ 17 & 1\ 0001 \end{array}$$

indicating the sum is one, and of course producing a carry. The carry is all right, but the sum is definitely wrong. The problem again is that we have passed over the forbidden range and landed, without signal, past it. Without signal, that is, except for the production of the carry, and that means that we should "add 6" again in order to ignore the six invalid or forbidden characters. Doing this in this example we get 1 0111, again the correct result.

This turns out to encompass all possible cases, and our 8421 addition algorithm for each digit becomes something like:

1) Perform regular binary addition

2) If the character perceived is a valid character without carry, then make no further correction; if not then

3) if an invalid character, add 6, producing a carry for the next stage; or

4) if a valid character but with a carry, then add 6 anyway, and pass the carry along.

Now all this can be compared with the algorithm for the excess-3 code. Again suppose we add 5 and 2:

$$\begin{array}{rr} 5 = & 1000 \\ 2 = & \underline{0101} \\ & 1101 \end{array}$$

Since each addend was in excess-3, then the sum must be in excess-6, hence must have a three removed in order to return it to the proper XS-3 character. Doing this

$$\begin{array}{rr} & 1101 \\ - & \underline{0011} \\ & 1010 \end{array}$$

which then compared with the proper code for decimal 7 is seen to be the correct XS-3 result.

Again let's try 6 and 5:

$$\begin{array}{rr} 6 = & 1001 \\ 5 = & \underline{1000} \\ & 1\ 0001 \end{array}$$

resulting in a carry, but also in an invalid character. To convert to XS-3 we should again subtract 3, but to "jump" the forbidden band we should this time add a 6. The net result is to add a 3 which we do

$$\begin{array}{r} 1\ 0001 \\ \underline{+\ \ 0011} \\ 1\ 0100 \end{array}$$

which is the correct XS-3 result for a "1" as the sum and a "1" as the carry.

No different kinds of things occur in this case, so the XS-3 algorithm becomes:

1) Perform regular binary addition.

2) If a carry is not produced, subtract 3.

3) Otherwise add 3.

Comparing the two algorithms, they are certainly different, although it remains moot whether one is "better" than the other or not. The 8421 requires a correction only in the event of an invalid character or a carry. The XS-3 requires the same correction at each addition, but only its sign depends upon the carry. The point to be illustrated here was that they were different, and if you were going to build an adder, based upon one or the other, it would be crucial to your career to know which.

1.3 Design Specifications

Paramount amongst the problems of logic design is the first step: that of transforming the verbal specification of exactly what is desired into a nonambiguous, realizable network which does that job which is intended. Since the first step in this process involves the human mind in determining exactly what is required, it should not be surprising that the initial specification of a problem statement and solution is often not well defined, and the steps to an ultimate solution involve a feedback sort of process whereby the "problem solver" must many times query the "problem poser" to reconsider just exactly what "answer" was wanted to a particular (unanticipated) set of values of the input variables.

In this section we shall briefly delineate two aspects relevant to the problem of design specifications. The first concerns the formal reduction of the arbitrary (binary) problem to the well-defined specification of the desired output for all possible sets of values of the inputs. The second is an historically accurate resume of the propositional calculus, at least those portions of interest to logic design. The calculus was the first formal attempt at quantifying human thought processes, and leads directly to the algebraic formalism of interest to logic designers: the switching algebra.

1.3.1 Binary Mappings and Combinational Functions

If we are to simplify our (computational) world to a set of binary values, then it follows that the only kinds of networks of such values that we can build are those which assign binary values to particular sets of the binary inputs involved.

This simplification of our world seems perhaps a little harsh, but it is complete, and anything that is computatable, in a very complete and abstract sense, is computable in terms of such a binary quantification.

As a very simple example, suppose we are concerned with the design of a heater control to regulate the temperature in our office. We can easily address the problem in terms of identifiable binary variables concerned with the environment of the office. First of all, we can agree that the heater should be either ON or OFF. Our remaining problem is to determine the operation of this control in terms of the independent variables of interest.

For example we might identify the variables of interest as whether our office window is open or closed (distinctly a binary variable), whether the outside temperature is less than 55° or not, and whether the door to the office is open (hence drafty) or not. All these are clearly binary variables and it is our problem to design the heater control as a desired function of these three variables:

w = true, implies that the window is open

t = true, implies that the outside temperature is less than 55°

d = true, implies that the door is open.

Our desired solution might turn out to be that:

> The heater is to be on if and only if the window is open, the temperature is greater than 55°, and the door is open; or the window is closed, but the temperature is less than 55°, and the door is closed.

Now this solution may be happy or not, depending on the particular individual. The point is that it is a solution in that it prescribes a response for the heater control as a function of the binary variables of interest. Suppose that we have a relay coil that operates in response to each of the variables in the sense that when a variable is "true" then the corresponding relay coil is operated. Then if, for example

— w —

indicated a contact on the w relay that is open when the w relay is not operated, while

— w' —

indicates a contact on the w relay that is closed when the w relay is not operated, then surely the network of Figure 1.3.1.1

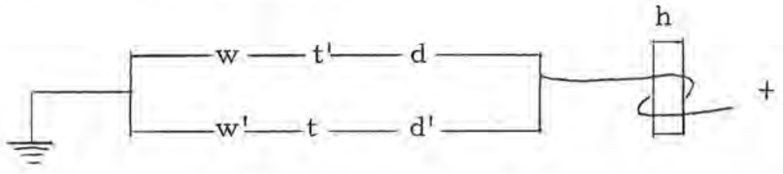


Figure 1.3.1.1. Switching network for heater control

properly operates the relay "h" which in turn controls the office heater.

This sort of design technique is surely appropriate to digital design, and is of sufficient generality to handle all possible such problems, whether they involve the control of heaters, or the next "bit" to be inserted in a register as a function of fifteen binary variables associated with a pattern to be recognized by a digital machine.

In some sense, then, our problem is completely solved by such ad hoc techniques. The reason that the problem is not completely solved, however, is that our simple example does not reveal the complexities of the problem that can exist when the number of variables involved in the problem statement grows larger than two or three. Furthermore, even our interpretation of the English statement that we posed above as a solution to the problem is not well defined, and can be interpreted in various ways.

To avoid these problems, it is necessary that an appreciable amount of formalism be adopted in order to handle such problems systematically, and without ambiguity. To this end we shall briefly review some of the first attempts at formalizing such verbal statements, and we shall see that these attempts lead directly to the switching algebra appropriate for making such verbal specifications explicit and realizable by switching networks of abritrary complexity.

1.3.2 The Calculus of Propositions

It might appear that the propositional calculus is properly the study of logicians and philosophers, and hardly should be of interest to practical engineers and computer designers. Our compelling interest is to work our way into a consideration of a switching algebra that will be of immediate utility in the design of networks of switches, and of things that act like switches.

Yet, as is often the case, the switching algebra that we shall develop is a simple example of the early propositional calculus studied by George Boole in the middle nineteenth century. Just as surely as Heaviside, Gardner and Barnes stood on the shoulders of Laplace, Fourier and Newton in the development of modern circuit theory, so did Shannon stand on the shoulders of Boole in his first presentation of modern switching theory not very many years ago.

There are several reasons for a brief consideration of the calculus. A not minor one is that it is historically relevant, and provides a convenient introduction to the stark formalism of the switching algebra. Another major reason is that the specifications of ultimately deterministic machines start off as English literal statements, and the formalism of something like the calculus is always necessary in order to make such statements exact, manipulatable, and, as we shall see, unambiguous.

Another reason is that it allows us to introduce many terms and procedures in the calculus which we shall use later in the algebra, and hopefully by that time they will seem like old familiar concepts.

Of course we have our option in the order in which we consider these matters. Many texts treat the propositional calculus, and then show the switching algebra as a particular case of it. Others develop the algebra in a purely formalistic way, and then give the propositional calculus as a "practical" application of it. We shall choose the former, but we should realize that the ordering is just a matter of taste.

1. 3. 2. 1 Elements of the Calculus

The objective of the propositional calculus, which is a branch of symbolic logic, is the pursuit of truth, and not many objectives could be much nobler than that. "Truth", however, much like the word "information" in information theory, must be taken in context, (i. e., Humpty Dumpty-like, it means exactly what we mean it to mean at the time that we say it, and nothing more nor less.)

Thus in this context, truth is a two-valued thing which can be assigned to declarative statements. That is, a declarative statement is looked upon as a variable which can assume one of two values: True (T), or not true (False, F). Of and other pair of marks would do all well (examples are $\top$, $\bot$; or 1, 0) but T, F will serve our purposes.

Such declarative statements are called propositions. Thus a proposition is a declarative sentence to which the value T or F can be assigned. For example, such a proposition is i, where

i : Iran is in South America.

This proposition reminds us that it is with logical truth that we are concerned, and the fact that the above statement is factually false is irrelevant. Thus a proposition is a logical variable with the evaluation T of F to be determined, or arbitrarily assigned. Of course, Boole wasn't concerned with anything so trivial as single propositions. His aim was to develop a model for the complex "laws of thought" by which compounds of elementary propositions could be composed and evaluated.

Thus in general we shall deal with a list of propositions designated by the symbols p, q, r, ... and in this form they are called unspecified or arbitrary propositions (i.e., propositions to which a truth value has not been assigned). That is, they are our propositional variables. It should be clear that propositions are to considered en toto; that is

p: Panama is in Asia
q: Quentin has red hair

are not to be split apart into classes, or sets, such as the "set of countries in Asia" or the "class of people having red hair" such as would be our want if we were considering the syllogisms of classical (or Lewis Carrol) type logic, and were attempting to come to conclusions about set membership. Rather our propositions are to be considered as an unbreakable set of atoms with which one of the two truth values are to be associated.

Of course we immediately disassociate ourselves from the logically undecidable sorts of statements like "the sentence I am uttering right now is false" as being in the class of things with which we are not concerned. These are games from a sort of "naive propositional calculus" which need not concern us.

Having recognized the variables of our calculus, we then recognize the values that they may assume as being the constants of our discourse; these are the marks T and F. Thus T and F are our propositional constants and we identify them immediately as those particular propositions that are either always true, or always false.

We shall of course want to relate propositions to one another. In particular given p and q, if we have the situation that p is T whenever q is, and conversely, then we write

$$p = q$$

which is called a tautology.

1.3.2.2 Compound Propositions

Given p, q, r, . . . (which are thus statements or elements describing some universe of discourse of interest to us--perhaps even the real world!) we wish to augment our propositions to include functions of these elemental propositional variables. This of course was Boole's goal as well; he conjectured that all of the reaches of rational human thought might be analyzable by examining the functions of simple propositional variables. Thus we want to consider things such as

$$f(p, q, r)$$

which, we shall insist, must be another proposition, i.e., another construct to which an assignment of T or F is again relevant. Thus f is a dependent variable which is to depend for its evaluation on the particular values of T or F assigned to the independent variables which are the elemental propositions p, q, r. . . .

Now can we go about systematically examining the nature of these functions of propositional variables? One way is suggested by the finiteness of the number of constants in our system. Since they are finite in number, we ought to be able to exhaustively examine the functions of n variables, starting with n = 0 and proceeding for n = 1, 2, 3, . . . We shall start this way, but shall not go very far before we conclude that any further is folly.

Are there any functions of no variables? Certainly. These are simply our two constants T and F. That is, the values of T and F are evidently independent of the assignment of values to any other propositions.

What about functions of a single variable? What do we mean by f(p)? Clearly f(p) = p is such a function, and it is T whenever p is, which is all rather trivial, and we don't seem to be making much headway. Of course there is only one other possible (nontrivial) case and this is the single variable f(p) that is T whenever p is F, and F whenever p is T. This f(p) is called the denial of p (alternatively the negative or complement of p). This f(p) is also written

$$\overline{p},\ p',\ \sim p,\ \text{not } p$$

and so on, depending upon the stylistic taste of the particular user. The point is that these symbols introduce the notion that functions can also be interpreted as operators on the variables of the argument of f. Thus we are considering in this instance unary operators, i.e., operators that require only a single argument. For example if p is the proposition "Panama is in Asia", then $\overline{p}$ is the proposition "Panama is not in Asia", or, probably better: "It is not the case that Panama is in Asia".

Have we exhausted the possibilities? Clearly so, in this case, although we can always confirm the situation by systematically examining all of the possibilities by enumerating them. This can be most easily done by means of the so-called truth table, or table of combinations, which is simply a way of displaying all the possible values that the independent variables can assume, and examining the number of possible f(p) that can be formed by assigning the same constants to f(p). Thus for the single-variable case we have

p	f(p)
F	a_0
T	a_1

where the a_i can be assigned arbitrarily as F or T. Thus the total number of different possible functions in such a table is exactly the number of different ways of filling in the column labeled f(p) by filling in the different constants. Since there are only two constants it follows that the number of different functions is exactly

$$2^{\text{No. of rows in the table}}$$

and in this case we have the possibilities

p	$f_0(p)$	$f_1(p)$	$f_2(p)$	$f_3(p)$
F	F	F	T	T
T	F	T	F	T

which we see are just

$$f_0(p) = F$$
$$f_1(p) = p$$
$$f_2(p) = \overline{p}$$
$$f_3(1) = T$$

and we are through with this case.

But what are the constants T and F doing in our tabulation, when we agreed that they were functions of no variables? Clearly, this sort of enumeration includes amongst the function of n variables, also the functions of n-1 variables, n-2 variables, etc. If a function f(p) is not actually a function of its argument p, we say that it is vacuous in p (or redundant in p, or degenerate). We shall concern ourselves with examining this situation more thoroughly later.

Now how about the functions of two arguments, i. e., the f(p, q)? Certainly there are more of them, but again we can enumerate all possibilities by extending our truth table

p	q	f(p, q)
F	F	a_0
F	T	a_1
T	F	a_2
T	T	a_3

and substituting all the possible values for the $a_i = 0, 1$. Since there are four rows, there will clearly be

$$4^2 = 16$$

functions of the two variables. As before, of course, some of these will be degenerate in the two variables. Also, some of them are of particular importance, so we shall draw them forth in a little more detail at this point.

AND.

Given p and q we form the f(p, q,) that asserts both of them are T. This f(p, q) is called their conjunction. Of course this is a two-variable operator, since two arguments are involved. It is symbolized by $f(p, q) = p\wedge q$ (or $p\cdot q$, or just pq, which we shall favor later on). Thus $p\wedge q$ is just the proposition that is true when and only when both p and q are true. We exhibit this function (or binary operator) by the appropriate truth table

p	q	$p\wedge q$
F	F	F
F	T	F
F	F	F
T	T	T

The rule for a combination, i. e., the name of the binary operation, is called a logical connective, and in the case of conjunction the connective is seen to correspond directly with the English usage of the word "and". Thus

p: Panama in is Asia
q: Quentin has red hair
$p\wedge q$: P is in A and Q has R.

Furthermore, although we shall not dwell on such properties, we note that the connective $\wedge$ displays all sorts of verifiable properties such as

$$p\wedge q = q\wedge p$$
$$p\wedge p = p$$
$$T\wedge T = T$$
$$T\wedge F = F\wedge T = F$$
$$p\wedge \overline{p} = F$$

which can be used to "evaluate" or "simplify" more complex forms, e. g.,

$$p\wedge p\wedge q\wedge p \,\Big|_{\substack{p\,=\,T\\ q\,=\,F}} = p\wedge q\wedge p\,\Big|_{T,\,F} = T\wedge F\wedge T = T\wedge F = F$$

and so on.

OR.

In similar fashion, English statements are often "or-ed" together. To simulate this two-variable operator we define the f(p, q) that is T whenever either p or q is T by the table

p	q	$p\vee q$
F	F	F
F	T	T
T	F	T
T	T	T

which defines the connective "$\vee$" (also written as $p + q$), and also called the disjunction or inclusive-OR of p and q.

Of course, admitting that there is something called the inclusive-OR suggests that there might be something called the exclusive-OR, and comparing the two possibilities reveals an ambiguity that occurs in common English usage. Thus, if we say either "p or q" we may mean to include "or both", or we may really mean "but not both", and it is not always clear in language usage just which is meant. If we say

"Either Paul is first in line or Quentin is first in line"

there is probably no ambiguity, since it's hard to admit that both might be first. But if we say

"Either Paul is first in line or Quentin is last in line"

then it is not at all clear whether we mean to include the possibility of both or not, since the events are not mutually exclusive.

In order to avoid this ambiguity we shall have to be precise where the meaning is not clear. We shall assume that $p + q$ represents the inclusive-OR read as "p or q or both", and for the other case we shall use $p \oplus q$ to be read as "p or q but not both". Thus the exclusive-OR, $p \oplus q$ is defined by the table

p	q	$p \oplus q$
F	F	F
F	T	T
T	F	T
T	T	F

The operator "$\oplus$" is also written sometimes as "$\not\equiv$" and called the "not equivalence" or "not-equiv" operation. This suggests the possibility of relations between the various operators that can be defined. For example, we can define intuitively another binary operator as follows. Surely p and q are equivalent if they are prescribed to have the same truth value, in order for the compound statement "p ' is equivalent to q" to be true. Thus the following truth table, using $p \equiv q$ for the relationship, surely defines $p \equiv q$, hence the operator "$\equiv$".

		(1)	(2)	(3)
p	q	$p \equiv q$	$\overline{p \equiv q}$	$p \not\equiv q$
F	F	T	F	F
F	T	F	T	T
T	F	F	T	T
T	T	T	F	F

Now column (2) shows the denial of $p \equiv q$, which is reasonably called "$\not\equiv$", and can be quickly identified as being the <u>same</u> operator as our exclusive-OR "$\oplus$". Hence the "not-equiv" operator is just another name for our exclusive-OR operator.

Incidently we have concluded that

$$\overline{p \equiv q} \;=\; p \not\equiv q \;=\; p \oplus q$$

and this "theorem" can be considered proved simply by the fact that we have shown the corresponding columns in the truth table to be identical.

For, after all, two function are the same if for every point in the domain (combinations of values of the arguments) they have the same value in the range (i. e., the same mapping to the function values). For a finite number of values of the discrete variables we can always (in principle at least) list all of the combinations of the arguments, and exhaustively verify whether the two functions are the same or not. Clearly the rows of the truth table constitute such a listing, and the identity of two columns verifies the assertion in a completely adequate way. This method of proof is perfectly general, provided we have time to list all the possibilities. It is called <u>proof by perfect induction</u>, and is so referenced in the literature.

As before, with respect to any of these operators, a number of useful computational tautologies can be derived. For example, with respect to $\vee$:

$p \vee p = p$; $T \vee p = T$

$p \vee \overline{p} = T$; $F \vee p = p$

$p \vee q = q \vee p$, etc., etc., and for the $\oplus$

$p \oplus p = F$

$p \oplus \overline{p} = T$, etc., etc.

In evaluation, we proceed as before; thus

$$pq \vee \overline{p}r \vee s \Bigg|_{\substack{p = T \\ q = T \\ r = F \\ s = T}} = TT \vee FF \vee T = T \vee F \vee T = T \vee T = T$$

and so on, for arbitrary expressions on these operators.

It is also important to note that just as in other case of binary operators these connectives might also be order sensitive. Thus it is generally important that we either agree beforehand on precedence of operators, or else we must carefully use parentheses in order to designate which compound operator we actually mean. Thus, for example, in the compound declarative sentence "John is sick or Mary is away and the house is cold", properly symbolized by

$$j \vee mh$$

it is not immediately clear whether we mean $j \vee (mh)$ or $(j \vee m)h$. Of course these might be the same, but it is not evident.

In order to verify whether they are the same or not, we can again call upon perfect induction to check:

j	m	h	mh	j ∨ m	j ∨ (mh)	(j ∨ m)h
F	F	F	F	F	F	F
F	F	T	F	F	F	F
F	T	F	F	T	F	F
F	T	T	T	T	T	T
T	F	F	F	T	T	F
T	F	T	F	T	T	T
T	T	F	F	T	T	F
T	T	T	T	T	T	T

and since the last two columsn are not identical, it follows that it is not the case that

$$j \vee (mh) = (j \vee m)h$$

hence that in general we must be careful to prescribe our ordering, or else to observe parentheses carefully in writing expressions in the binary operators we're considering.

Notice that the above was again an example of perfect induction. To show that two functions are not the same, it suffices to show that they differ in at least one row of the truth table that displays all of their possible values.

We could continue with an exhaustive consideration of the two-variable operators, but instead we'll pause for a bit and consider a few points that can be made at this time for arbitrary n-variable functions.

1.3.2.3 Some n-Variable Observations

What do we mean by an n-variable function in the first place? By extension we mean an $f(p_1, p_2, \ldots, p_n)$, where n is a finite (though arbitrary) number, and the p_i are again our elementary or independent propositional variables. The function is well-defined if for every set of "values" for the p_i's a value of f is determined as either a T or an F.

Again, because of the finiteness of things, we can describe such a function by simply listing all of its function values for each point in the domain. I.e., again our truth table formalism extends directly to any n, and we can write immediately that

P_n	...	P_2	P_1	$f(p_1, \ldots, p_n)$
F	...	F	F	a_0
F		F	T	a_1
F		T	T	a_2
	⋮			⋮
T		T	T	a_{2^n-1}

and be assured that each of the possible n-variable functions corresponds to a selection of the values of the a_i from either T or F.

Of course, we can also use this vehicle again to answer the question of how many n-variable functions there are. As before, the answer is the number of different ways of specifying the column for f in the table. This is still

$$2^{\text{No. of rows in the table}}$$

and the number of rows is in turn a function of the number of independent variables and is easily seen to be 2^n. It follows that the number of different n-variable functions is

$$2^{2^n}$$

which is eminently finite as predicted, and alluringly simple looking. But the allure is deceptive for the number N(n) of such functions grows alarmingly fast, and becomes very large even for reasonably small values of n. Thus we have

	n	N(n)
	0	2
	1	4
	2	16
	3	256
	4	$2^{16} = 65,536$
(Number of neurons in human nervous system)	5	$2^{32} = 4,294,967,296$
	6	$2^{64} \doteq 2(10^{19})$
(Total number of games of chess)	7	$2^{128} \doteq 4(10^{38})$
(Total number of atoms in universe)	8	$2^{256} \doteq 10^{75}$
	⋮	⋮

and it becomes clear that any sort of exhaustive examination is completely out of the question, and any proposed technique for doing so rapidly becomes hopelessly swamped.

Nevertheless, we can completely, and formally describe all possible situations by the augmented truth table:

	$p_n \ldots p_1$	$f_0(p_1, \ldots, p_n)$	f_1	$\ldots$	$f_{2^{2^n}-1}$
2^n rows	F ... F				
	⋮				
	T ... T				

At this point we might hope to proceed exactly as before for the case of two variables. That is, we might define n-variable connectives, although we have already seen that we can't do anything exhaustive. But some subset might be useful. For example, we can certainly define an n-variable AND by extension as

$$A_n(p_1, \ldots, p_n) = \text{ that f which is true whenever } \underline{\text{every}}\ p_i \text{ is true}$$

and the n-variable OR as something like

$$O_n(p_1, \ldots, p_n) = \text{ that f which is true whenever } \underline{\text{any}}\ p_i \text{ is true.}$$

Obviously, then, our binary connectives are special cases of such things, and we agree that when n = 2 we can write

$$A_2(p_1, p_2) = p_1 \wedge p_2$$

$$O_2(p_1, p_2) = p_1 \vee p_2$$

and so on.

But can we do better? We should certainly hope so, for keeping track of the possibilities, even the interesting ones, would simply become too horrendous a bookkeeping task.

It turns out that we can do better, in fact we can easily show that just the two-variable connectives we've defined so far will suffice to represent any n-variable connective that we wish.

To verify this assertion we first of all need to affirm associativity for our binary connectives; namely we need to affirm that

$$(p_1 \wedge p_2) \wedge p_3 = p_1 \wedge (p_2 \wedge p_3) = p_1 \wedge p_2 \wedge p_3$$

and that

$$(p_1 \vee p_2) \vee p_3 = p_1 \vee (p_2 \vee p_3) = p_1 \vee p_2 \vee p_3 .$$

Furthermore to simplify the reproduction, let us agree to write "$\wedge$" as " " from now on, i.e., $p_1 \wedge p_2 = p_1 p_2$, and write "$\vee$" with a "+", i.e., that $p_1 \vee p_2 = p_1 + p_2$.

Having shown that associativity holds we can immediately affirm that our two n variable connectives can be expressed in terms of our binary operators as

$$A_n(p_1, \ldots, p_n) = p_1 p_2 \ldots p_n$$

$$O_n(p_1, \ldots, p_n) = p_1 + p_2 + \ldots + p_n .$$

Now we argue a little further, toward the end of expressing all functions in terms of these operators. What does the function $A_n(p_1, \ldots, p_n)$ look like on the truth table? It is simply a function that has a T assigned to the last row, and an F everywhere else. Now any function of the form $p_1^* p_2^* \ldots p_n^*$, where p_i^* is a notational convenience to indicate that the variable p_i may appear either complemented or not, is called a <u>fundamental conjunction</u>, or a <u>fundamental product</u>, or a <u>minterm</u>, and is clearly just like A_n, i.e., it is a function that has a single one in its last column. Just as clearly, any minterm can be expressed in terms of A_n and the unary operator "‾" as

$$A_n (p_1^*, p_2^*, \ldots, p_n^*) = p_1^* p_2^* \cdots p_n^*$$

so that we have bootstrapped ourselves to the proposition that each of the 2^n n-variable functions that are minterms can also be expressed in terms of our binary operators, indeed just in terms of the And the Complement.

The final step we need is to observe that these minterms form a sort of orthogonal set of functions by which any arbitrary function can be expressed simply by ORing those pointed to by the rows wherein the arbitrary function has T's assigned. Since we have already shown the OR to be associative it follows that

$$f(p_1, \ldots, p_n) = \sum_{i=0}^{2^n-1} a_i p_1^* \cdots p_n^*$$

where our summation is over the rows wherein f has a T, and indicated by the value of the a_i. Thus if f has a T in row i, then $a_i = 1$, otherwise it is 0.

For example, for the 3-variable proposition represented by

p_3	p_2	p_1	$f(p_1, p_2, p_3)$
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	F
T	T	T	F
T	T	T	T

we have immediately that

$$f(p_1, p_2, p_3) = p_1\overline{p}_2\overline{p}_3 + p_1p_2\overline{p}_3 + p_1p_2p_3$$

and our general conclusion that any n-variable function can be represented only in terms of our binary operators if we choose. In other words, every function can be expanded as a summation of minterms, and such an expansion, which is clearly unique, within commutativity, for every f, is called the disjunctive normal form for f.

We could continue in this vein and illustrate lots of other n-variable facts of life. Suffice it at this point to content ourselves with the fact that even though the general situation becomes terribly complex and unwieldy, we can still express everything in terms of the simple binary connectives that we've already developed. Others are occasionally useful, however, and we shall examine a few more of them at this time.

1.3.2.4 Other Useful Connectives

Even though we have seen that we need nothing more than the connectives already introduced in order to talk about any function we wish, there are still many others of interest, and of ultimate practical importance. We shall consider only a few more briefly at this point.

Conditional

The function $p \supset q$, sometimes called "p implies q", and equivalent to the mathematical assertion "if p then q" is defined by the table

p	q	$p \supset q$
F	F	T
F	T	T
T	F	F
T	T	T

and although there is often some confusion on the logical validity of the second row, it is assignedon the basis of the classical logic conclusion that a false premise can imply anything.

We can relate this connective to a couple that we've already defined.

Theorem: $p \supset q = \bar{p} \vee q$

Proof: We use perfect induction:

p	q	$p \supset q$	$\bar{p}$	$\bar{p} \vee q$
F	F	T	T	T
F	T	T	T	T
T	F	F	F	F
T	T	T	F	T

and it suffices to confirm that the two columns for $p \supset q$ and for $\bar{p} \vee q$ are identical.

Biconditional

In this case we assume that the implication runs both ways, that is that $(p \supset q)(q \supset p)$. Since we have expressed this one in terms of the previous connective, we can easily derive what this function looks like:

p	q	$p \supset q$	$q \supset p$	$(p \supset q)(q \supset p)$
F	F	T	T	T
F	T	T	F	F
T	F	F	T	F
T	T	T	T	T

and examination of the last column shows that we've already met this connective before. It is just the function $p \equiv q$ that we've previously defined. It corresponds to the mathematical assertion "iff", i.e., "p if and only if q", and the only thing new is another theorem linking the various connectives:

Theorem: $p \equiv q = (p \supset q)(q \supset p)$

DeMorgan's Theorems

Obviously we could go on like this through a great many such relationships. We shall be content with only a few more that have special significance. One of these is the theorem:

Theorem: $\overline{pq} = \bar{p} \vee \bar{q}$; $\overline{p \vee q} = \overline{pq}$

Proof: Again trivial by perfect induction.

These two relations are the well known DeMorgan laws for two variables. We shall examine later their generalization to n variables. Suffice it for now to note that they enable us to conclude something further with regard to the general expansion result noted earlier. We have concluded that any function can be represented using only "$\wedge$", "$\vee$", and "$\overline{\ }$". Now, on the basis of the first assertion above we conclude that "$\wedge$" and "$\overline{\ }$" will suffice, for whenever we have a "$\vee$" occurring in an expression we can always substitute an expression containing only "$\wedge$" and "$\overline{\ }$". The assertion follows immediately. Of course, a little care must be taken in asserting these things too glibly; the first statement evidently requires that the variables be "primed" in order to make the substitution. But we need only observe that if p is a propositional variable, then so, of course is $\overline{p}$, so that an equivalent statement to the first DeMorgan assertion above is that

$$p \vee q \ = \ \overline{\overline{p}\,\overline{q}}$$

and we are home free with all sorts of such combinations.

Equivalently, we note that the second DeMorgan theorem shows that we can get along with just "$\vee$" and "$\overline{\ }$" if we wish.

This raises another interesting question: just how many connectives do we need in order to express any function that we wish? We earlier showed that three will suffice. We have just showed that two will suffice. Is it possible that amongst all of the possible variants, that there are some single connectives that will suffice? This turns out to be the case, and is the reason for the interest in the Sheffer Stroke function and the Pierce Dagger function.

Other Connectives

The Sheffer Stroke p|q is the two-variable connective defined by the table

p	q	p|q
F	F	T
F	T	T
T	F	T
T	T	F

Often referred to as "p stroke q", the English proposition is clearly "it is not the case that p and q". This phrasing suggests an important theorem relating the connectives:

Theorem: $p|q$ - $\overline{\bar{p}\bar{q}}$

The proof is obvious by examination of the truth table. The significance, however, is that the theorem is the first step in our argument to show that the stroke function is sufficient for everything. It enables us to conclude that whenever we have a conjunction of variables we can replace the conjunction by a stroke and a complement. Thus

$$pq = \overline{p|q}$$

and we are half way there. The remaining piece is suggested by simply replacing q with another p in the theorem, for then we have

$$p|p = \overline{pp} = \bar{p}$$

which says that whenever we have a "$\bar{\ }$", it too can be replaced with a function using only the stroke connective. Since we've previously concluded that "$\wedge$" and "$\bar{\ }$" are sufficient, it follows immediately that "$|$" is as well.

The term Stroke Function is not used as commonly at present as are the terms that key on the English statement, and the connective is usually called the "not-and" connective, or the "NAND".

The assertion that the NAND is sufficient does not mean that displaying this sufficiency in any given case is a trivial matter. We shall later be concerned with systematic techniques for such display, using the least number of NAND connectives. But for now we note that the reduction of even quite simple functions such as

$$\begin{aligned}(p+q)r &= \overline{(p+q)|r} \\ &= \overline{\overline{\bar{p}\bar{q}}\,|\,r} \\ &= \overline{(\bar{p}|\bar{q})|r} \\ &= \overline{[(p|p)|(q|q)]|r} \\ &= \left\{[(p|p)|(q|q)]|r\right\} \Big| \left\{[(p|p)\,|\,(q|q)]\,|r\right\}\end{aligned}$$

can get pretty messy, and involves careful use of the parentheses.

There is only one other connective that exhibits this same property, and this fact makes it important enough to display at this time. This is the Pierce Dagger function, symbolized by $p\downarrow q$ and defined by the table:

p	q	p↓q
F	F	T
F	T	F
T	F	F
T	T	F

Based upon its English statement equivalent that "it is not the case that either p or q" it is not surprising that this connective also is known as the "not-or" or the "NOR" connective.

To show that the NOR is also sufficient to represent any function we can take yet a little different tack, based upon our steadily increasing store of relations. Since we have seen that the NAND is sufficient, then it follows that if we can show a realization of the NAND using only NORs, then we can also make the same conclusion about the NOR. Let's try it that way:

$$\begin{aligned} p \mid q &= \overline{pq} = \bar{p} + \bar{q} = \overline{\overline{\bar{p} + \bar{q}}} \\ &= \overline{\bar{p} \downarrow \bar{q}} \\ &= \overline{(p\downarrow p) \downarrow (q\downarrow q)} \\ &= [(p\downarrow p)\downarrow (q\downarrow q)]\downarrow [(p\downarrow p) \downarrow (q\downarrow q)] \end{aligned}$$

and we are through. (Again these transformations are indicated at this point only to show existence, not efficiency. When "cost" becomes a factor, i. e., when two NOR gates costs more than one, then we shall worry about efficiency of the representation.)

Although the predominant practical interest has been in the two-variable connectives, there has also been considerable interest in other types. One of these is the so-called majority connective, M(p, q, r), which is usually defined on an argument of three variables (although it could clearly be defined in the same way for any argument of any odd number of variables) as the statement "at least a majority of p, q, r (i. e., at least two) are true". Thus it is defined by the table

p	q	r	M(p, q, r)
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	T

Of course we could define a multitude of similar connectives in terms of three or more variables, corresponding to various constraints familiar in English discourse. We shall avoid the temptation to do so, except to note that several of them, such as the majority connective, have played a significant practical role. Suffice it for now to illustrate the fact that the majority gate's flexibility can be suggested by showing how it can "emulate" the action of lesser connectives. E. g., analysis (or even simple examination of the corresponding English statements) will show that

$$M(p, q, F) = p \wedge q$$

while

$$M(p, q, T) = p \vee q$$

so that the gate is capable of being specialized to either of the useful connectives we've met in the two-variable case.

We shall not detail other connectives of interest. Clearly we shall ultimately make the transition which involves connecting each of these connectives to a logical "gate" realized, for example, electronically. The vagaries of electronic realizations have ordained that the fewer-input gate types are the easiest to build reliably, hence are of greatest interest to us. But the electronics do allow in many case a greater number of gate inputs than two, and to the extent that these are reliably realizable, then so too are the multiple input connectives associated with them. Thus, for example, a three or four input AND is probably not too difficult, but a seven input AND is probably pressing things.

1.3.2.5 "Logic Design" with "Connective Blocks"

We shall terminate these considerations with an extension of our propositional calculus to a graphical equivalent that is intended to be entirely suggestive of the sequel. We shall do this by making an identification of the elements of our calculus with the elements of a set of directed graphs.

For example the notion of a "connective" is suggestive of a "logic block". Thus $p \wedge q$ is a block marked "$\wedge$" that connects inputs p and q and produces $p \wedge q$ on its "output" as in Figure 1.3.2.5.1:

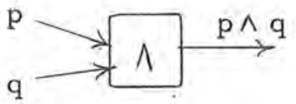
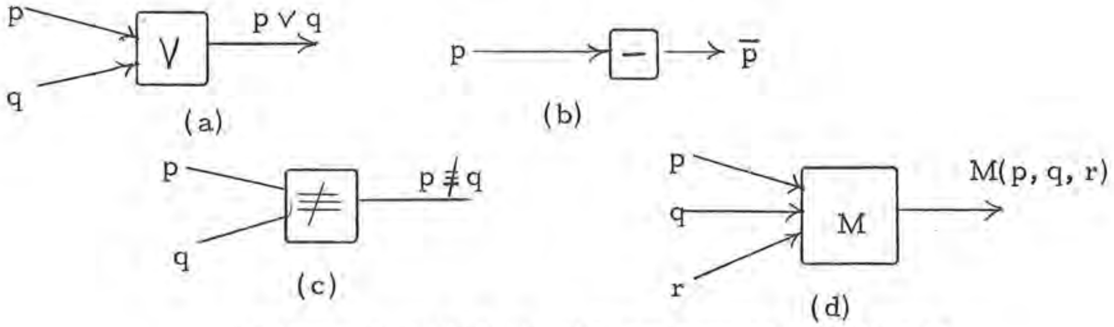


Figure 1.3.2.5.1. A connective block

Similarly, we could define the various other blocks appropriate to whatever connectives we might find of use such as in Figure 1.3.2.5.2:



(a)

(b)

(c)

(d)

Figure 1.3.2.5.2. Other connective blocks

Clearly, to complete the association of propositions with graphs, we designate the propositions as the "signals" that appear on the interconnecting lines of the graph. These signals can take on either of the values T or F. Finally, then, compound propositions must be representable by interconnecting the propositional blocks as required. The resulting networks of "connective blocks" must stand in one-to-one correspondence with the functions of the propositional calculus, and the evaluation of a function must be mapped into the value of the ultimate output to the network. For example the function $p \vee (\overline{q} \wedge r)$ must be equivalent to the network of Figure 1.3.2.5.3:

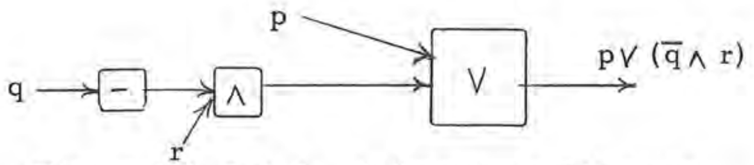


Figure 1. 3. 2. 5. 3. A function realization

while the function $(p \wedge q) \vee (p \wedge r) \vee (q \wedge r)$ must correspond to either of the networks shown in Figure 1. 3. 2. 5. 4.

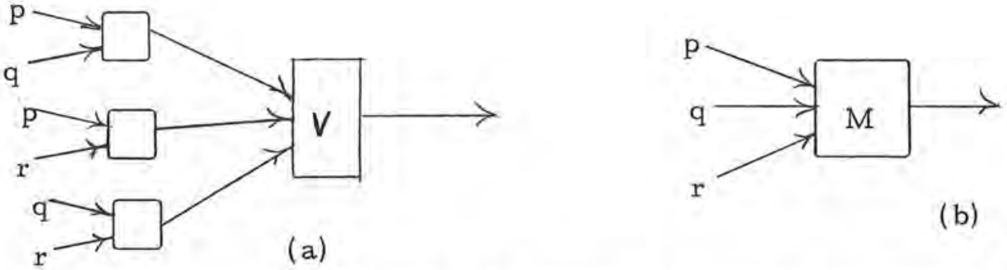


Figure 1. 3. 2. 5. 4. Alternative function realizations.

There are many other facets of classical logic that we could refer to for insight on useful tools to use in the sequel. For example the so-called "predicate calculus" enables us to sever our propositions into "predicate" and "subject" in useful ways. E. g., the proposition "All Greeks are philosophers" can be considered in the theory of sets or classes as referring to two sets of things: those that are Greeks, and those that are philosophers. Combinations of such considerations lead to interested class assertions familiar to all followers of Lewis Carroll and other devotees of classical logic. A useful side product are switching theoretic constructs such as Venn diagrams for demonstrating such class assertions, and proving elemental theorems in the switching algebra. We shall not look at these for now, but instead shall be content with the formalization of a system which is isomorphic to the propositional calculus that we shall call the "switching algebra".

1.4 The Problems of Existence and Minimization

Two basic concerns lurk in the background of every practical problem in logic design, even though the designer may not be aware of them in every case. In those cases where they do not play an important role, however, it is only because the particular restrictive assumptions of the problem have already taken them into account. It seems appropriate to close this introductory chapter with at least an acknowledgment of the nature of these two basic concerns, for they illuminate, and indeed result from the basically "engineering" characteristics of the digital machine design problem.

For engineering, after all, is primarily a problem solving discipline, and the first concern is not only an illumination of the various alternative solutions to a problem, but indeed whether a solution exists at all. So, too, in the case of the logic design problem: given a problem specification, either verbal or more precisely defined in terms of truth tables or algebraic representations, this specification must be played off against the practical constraints that must be observed by the designer. These practical constraints include some reasonable limitation on the kinds of logic gate complexity he can provide, including limitations on the fanin and fanout to the elements. More subtly though, they might also include limitations on the layout of the logic elements, the total area involved, the possibility of restrictions on the interconnection structure that links them, and ultimately forms the desired outputs, and so on.

Thus the first basic question of concern is that of existence of a solution for the desired problem. This question frames a lot of the questions considered in the following chapters, resulting in canonical forms, or techniques, by which standard solutions to all realizable specifications can be generated.

The second basic engineering consideration, however, given a set of solutions to a particular problem, involves determining an efficient solution amongst them. In logic design this process is described as minimization, and is the second concern to be highlighted here. In any given context, the properties of the solution being minimized will vary. In a relay contact network we might be concerned with the total number of simple contacts in a network, or alternatively in the total number of relay coils involved, under the constraint that a given coil can accommodate only a bounded number of contacts. In an electronic gate network the concern might be the total number of gates, or alternatively the total number of gate inputs, again usually with some reasonable bound on the total number of inputs a given gate can reliably sustain. In densely packed integrated chips, however, we might care not in the least about the total number of elements we use (within several thousands at least), but we might be very concerned about the total number of leads assigned to the periphery of the chip, or about the total number of levels of gates through which a signal must pass before appearing on an output of the chip, and so on.

This second question of minimization, regardless of the parameters of concern, is always the determinant of the particular logic design process employed, and always provides a measure of its quality or success as a design technique. The important point to note is that the problem of design then becomes one of infinite variety, depending only upon the local conditions that obtain, and which restrain the designer from complete freedom of choice in his options. In the following chapters we shall trace some of the most common methods of systematic design, but it should always be remembered that each technique is couched within a specific set of assumptions regarding what is important, or costly, hence exactly what should be minimized. It should always be remembered that each new environmental context will lead to a re-examination of the parameters of importance, hence of the particular logic design technique that is sutiable for the new application.

1.5 Notes - References - Problems

NOTES.

The basic material on number systems, coding, and verbal specifications are represented in some form in many textbooks and papers on logic design, switching theory, and computer organization. Extensive discussions on number systems and arithmetic are contained in Flores (4) and Richards (8). The basic notions of the propositional calculus trace back to Boole (1) over 100 years ago, and have since been adapted and augmented by many people. Recent treatments will be found in Kohavi (5) and Krieger (6). Many books are also available on the general digital design formulation problem. These include Chu (2), Dietmeyer (3), and Phister (7) amongst many others, and these will provide further references to the basic literature.

REFERENCES.

1. Boole, G., "An Investigation of the Laws of Thought," Dover Publications, Inc., New York, 1854.

2. Chu, Y., "Digital Computer Design Fundamentals," McGraw-Hill Book Co., New York, 1962.

3. Dietmeyer, D. L., "Logic Design of Digital Systems," Allyn and Bacon, Boston, 1971.

4. Flores, I., "The Logic of Computer Arithmetic," Prentice-Hall, Inc., Englewood Cliffs, N. J., 1963.

5. Kohavi, Z., "Switching and Finite Automata Theory," McGraw-Hill Book Co., New York, 1970.

6. Krieger, M., "Basic Switching Circuit Theory," The MacMillan Co., New York, 1967.

7. Phister, M., "Logical Design of Bigital Computers," John Wiley, New York, 1958.

8. Richards, R. K., "Arithmetic Operations in Digital Computers," D. Van Nostrand Company Inc., Princeton, N. J., 1955.

PROBLEMS.

1. Convert the following decimal numbers into binary numbers; for fractions express the binary numbers to at least six places and indicate if the conversion is exact or not:

   a) 57

   b) 0.40625

   c) 15/64

   d) 0.865

   e) 432.5625

   f) 2048

2. Convert the decimal numbers of Problem 1 into octal (base 8) numbers.

3. Given that $(79)_{10} = (142)_b$, determine the value of b.

4. Convert the following binary numbers into decimal form:

   (a) 0.101101

   (b) 100,000

   (c) 110.110

   (d) 111011.001101

5. Convert the following numbers to the base indicated:

   (a) $(111)_3 = N_2$

   (b) $(175)_8 = N_2$

   (c) $(413.32)_5 = N_{10}$

6. The number of atoms in the universe is alleged to be about 2^{80}. About how many decimal digits would be required to express this number decimally? How about in ternary (base 3)?

7. Given that $(16)_{10} = (100)_b$, determine the value of b.

   Given that $(292)_{10} = (1204)_b$, determine the value of b.

8. In the following series, the same integer is expressed in different number systems. Determine the missing member of the series.

   10,000, 121, 100, ?, 24, 22, 20, ...

9. Construct addition, subtraction, and multiplication tables for a radix 5 number system.

10. When the order of the digits of a certain three-digit decimal number is reversed, the same number to the base 16 results. Find the number.

11. Work out a simple test for determining if a given binary number is a multiple of three.

12. How many weighted 4-bit BCD codes are there using only positive integer weights?

13. Using the BCD XS3 code only, illustrate addition when applied to decimal 147 + 78.

14. Symbolize in the propositional calculus, and determine truth values of the following compound proposition:

"If it is hot in Arizona and it is raining outside or demonstrators are in the streets, then it is hot in Arizona, demonstrators are in the streets, and it is snowing in Argentina."

So that we all have the same start let us symbolize the independent propositions as follows:

It is hot in Arizona = H
It is raining outside = R
Demonstrators are in the streets = D
It is snowing in Argentina = S

If any ambiquities appear, please make sure that you make explicit how you have resolved them.

15. Prove the tautology

$$(E \supset F)\ (F \supset G)\ (G \supset E) = (E \vee F \vee G)' \vee EFG$$

Write in symbolic form the four possible interpretations of the following ambiguous statement:

"Peter is jaundiced or Quincy is neurotic and Roger is gone."

16. Draw a circuit using AND, OR and NOT elements to produce an output for each of the following conditions:

a) (A or B) and (not A or not B) and C

b) (A the same as B) and C, or (A different from B) and not C.

c) At least two of A, B, and C.

17. Using a truth table verify that

$$(A \supset B) \supset \overline{[A \vee (A \wedge B)]} = A \wedge \overline{B}$$

18. Express the connective $\oplus$ in terms of the connective $\downarrow$.

19. Binary digits a, b, c are to be added together to give $s_1 s_2$ as in

$$\begin{array}{r} a \\ b \\ +c \\ \hline s_1 s_2 \end{array}$$

Let X = the truth function variable that is true whenever x = 1, and false otherwise. Determine a truth functional expression for s_2 and s_1, and construct a diagram of logic blocks that corresponds to these functions.

2. Combinational Foundations

In this chapter we shall develop some basic tools by which we can address the logic design problem for combinational networks. Recall that the combinational network is one wherein the "sequence" of inputs plays no role, the network has no recall, and the outputs of a network are to be determined strictly as a "combinational" function of the particular values of the binary inputs to the network. It will be seen that all of our work is essentially a formalization directed toward realizing prescribed truth values for specifications defined in terms of a set of binary input variables.

2.1 Logical Connectives and Electronic Gates

There is no question that the application of the high speed, reliable electronic switch has made all the difference in the development of digital machines to their present significant level of application in sophisticated computing systems, as opposed to the level attainable by mechanical calculators, simple control devices, and the like. The application of the conceptually simple gates, operating very fast and in large assemblages, has made possible the impact of the modern, large-scale digital computer on society.

Yet the heart of the system, the electronic gate, is a very simple device that is nothing more than an embodiment of the logical connective gates we discussed in the section on the propositional calculus. At the level of the gate itself, of course, there must be careful tailoring of the physical parameters, and a proper identification of the values of the physical variables with the truth values of the binary system. But at the level of logic design we need only concern ourselves with the function of the gate as a logical connective, limited only by the number of variables over which it operates (i.e., the fanin), and the relative ease with which the different connective types can be realized within any given technology.

The identification of such electronic gates as models of our logical connectives suggests means by which we can systematically formalize the realization of arbitrary digital systems, and we proceed at this point directly to the relevant algebraic formulation that is appropriate for such systems.

2.2 The Switching Algebra

We are now going to concern ourselves with an algebra which directly corresponds with the action of "switches", where a switch is anything that exhibits, unequivocally, one of two possible positions, or states. We shall see that the algebra is a direct application, if you will, of the propositional calculus, and is one of the foundation stones of logic design to follow.

We shall take the approach of defining a formal system, referring briefly to a physical correspondent in order to validate the postulates of the system, and then concern ourselves with the several theorems and other conclusions of the algebraic postulates that will be relevant to our purposes.

2.2.1 Definitions

A switching algebra consists of a set of variables $x, y, z, \ldots$ that can take on either of the values, or marks, 0, 1 unambiguously, i.e.

$$x = 1 \text{ iff } x \neq 0$$

$$x = 0 \text{ iff } x \neq 1.$$

The algebra is made complete by defining several operations for it. Two of these are binary operations, + and -, while the third is a unary operation, denoted by a ' . We define these operations by means of a set of <u>postulates</u> which specify the effect of the various operations on the constants, 0 and 1.

The postulates which specify the system, then, are

$$\begin{array}{ll} 0 + 0 = 0 & 0 \cdot 0 = 0 \\ 0 + 1 = 1 + 0 = 1 & 0 \cdot 1 = 1 \cdot 0 = 0 \\ 1 + 1 = 1 & 1 \cdot 1 = 1 \end{array}$$

$$0' = 1$$

$$1' = 0$$

and from these all else can be shown to flow. Notice first of all that these postulates completely specify everything that can happen by way of the constants, and the effect of the operators upon them. Nothing less would be complete; anything more would be redundant, or worse yet, inconsistent.

Secondly, we take note of the complete dualism between the operators + and · , and the constants 0 and 1. If we take any of the postulates and map

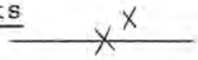
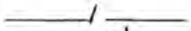
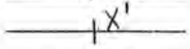
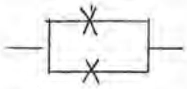
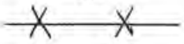
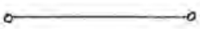
$$\begin{array}{c} 0 \longrightarrow 1 \\ 1 \longrightarrow 0 \\ + \longrightarrow \cdot \\ \cdot \longrightarrow + \end{array}$$

then we will find that we have generated another of the postulates correctly. Much that is useful follows from this simple observation -- in particular that we need expend only about half the usual amount of labor in theorem proving -- for each theorem proved we shall get a dual theorem for free, simply by making the substitutions observed above. Once in a while this process will fall flat, since some assertions will be the same as their duals, but in general the property is a productive one, and is often suggestive of alternative approaches to problems. We shall make much of this property of dualism in what follows.

2. 2. 2 Consistency

Of course in any system of postualtes there is the problem of consistency, and if the postulates are inconsistent then all sorts of dire consequences in terms of contradictory conclusions can follow. We shall follow the classic concern and belay possible concerns about consistency of our postulates by pointing to a particular physical system (out of many that we might have chosen--the propositional calculus itself being one) that satisfies the postulates. It will follow that theorems in our algebra will have immediate application in the physical system we point to, as well as the others that we can also show satisfy them.

Since any physical system will do, we'll again take the classic route and utilize the relay contact network. In particular we make the following associations between the two systems:

Algebra		Contact Networks	
Variable x	⟷	make contact on coil	
Variable x'	⟷	break contact	
+	⟷	parallel connection	
·	⟷	series connection	
0	⟷	open circuit	
1	⟷	closed circuit	

and these provide correspondences for each of the elements of our algebra. We can quickly check the validity of the postulates in the physical system. For example the postulate

$$0 + 1 = 1$$

translates to

(open circuit) parallel (closed circuit) = (closed circuit)

while

$$0 \cdot 1 = 0$$

translates to

(open circuit) series (closed circuit) = (open circuit)

etc., etc. Each of these is physically observable as are all the others, hence our system is consistent, and we can proceed to derive theorems in it without fear of contradiction, so long as we are careful.

2.2.3 Basic Theorems

The important point to note here is that all else in the system must follow from our postulates, i.e., be provable by reference to them. Of course we shall play the usual game that any proved theorem becomes part of the bag of tools available for the proof of subsequent results.

There are a collection of basic theorems which we shall collect here for reference. Some are surprising, and some are not. All are consequences of the postulates, although we shall only indicate proofs for a few. We shall attempt to cluster them under their common names, and shall try to include the dual theorems together.

Idempotent: $x + x = x$ $\qquad$ $x \cdot x = x$

(Proof by perfect induction. Simply consider all possible events.

$$0 + 0 = 0$$

$$1 + 1 = 1$$

Hence the result follows for all x.)

Variables and constants:

$$x + 1 = 1 \qquad x \cdot 0 = 0$$

$$x + 0 = x \qquad x \cdot 1 = x$$

Commutativity: $x + y = y + x$ $xy = yx$

(Notice we'll again use the convention of dropping the "•" whenever convenient.)

Associativity: $(x + y) + z = x + (y + z) = x + y + z$

$(xy)z = x(yz) = xyz$

Complementation: $x + x' = 1$ $x \cdot x' = 0$

Distributivity: $x(y+z) = xy + xz$ $x+yz = (x+y)(x+z)$

(Note our implicit acceptance here of the usual precedence relations. We'll not make more it it here, but recognize it as a problem to always be aware of. The first of these forms is not surprising. It affirms that multiplication distributes over addition, consistent with the real numbers, hence not surprising. The second, however, is always surprising at first sight since it asserts that addition distributes over multiplication. If the first is true, then the second obviously is by dualism. However, let's check the second anyway by perfect induction:

x	y	z	(x+y)	(x+z)	yz	x+yz	(x+y)(x+z)
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1
1	0	1	1	1	0	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

Since the last two columns are identical, the assertion is proved.)

Absorption: $x + xy = x$ $x(x+y) = x$

$x + x'y = x+y$ $x(x'+y) = xy$

(These look a little funny too, although they can be easily proved by perfect induction. Let's take advantage of the theorems we've already developed, however, to prove the first one by algebraic manipulation:

	Justification
$x + xy =$	
$x \cdot 1 + xy =$	$x \cdot 1 = x$
$x(y+y') + xy =$	$x + x' = 1$
$xy + xy' + xy =$	$x(y+z) = xy + xz$
$xy + xy + xy' =$	$x + y = y + x$
$xy + xy' =$	$x + x = x$
$x(y+y') =$	$x(y+z) = xy + xz$
$x \cdot 1 =$	$x + x' = 1$
x	QED .

We'll not often go through such detail, but this one provides a perfect example of the step-by-step application of the postulates and previously-proved theorems.

The second theorem obviously follows by dualism from the first. The third should be proved by the reader if there is any hesitancy about the argument made above, while the last one follows by dualism from the third.)

Consensus: $xy + x'z + yz = xy + x'z$ $\qquad (x+y)(x'+z)(y+z) = (x+y)(x'+z)$

(This one looks even a little stranger yet, and implies a sort of "law of the vanished middle". It is obviously relevant to the production of simplest forms which will ultimately be of interest. If there is any hesitancy remaining on the part of the interested reader, this should be removed by a careful algebraic construction of the proof. Again, they can both obviously be proved by perfect induction, and one follows from the other by dualism.)

DeMorgan (on complementation):

$$(x')' = x$$

$$(x+y)' = x'y' \qquad (xy)' = (x' + y')$$

(These are not at all easily proved algebraically--you might wish to try it!--although of course both are trivially proved by perfect induction. Algebraically one can proceed by filling in all the steps carefully in the following arguments:

$$
\begin{aligned}
1 = (x+x') &= (x+x')(y+y') = x'y'+x'y+xy'+xy \\
\therefore\ (xy)' &= x'y'+x'y+xy' \\
&= x'(y'+y) + xy' \\
&= x' + xy' = x'+y' \text{ as desired.}
\end{aligned}
$$

2.2.4 n-Variable Theorems

We have completed our basic repertoire of theorems in two or three variables. It remains only to identify the several n-variable theorems that will be of interest and utility to us. The first is a simple extension of the two-variable DeMorgan statements:

Theorem: $(x_1+x_2+\dots+x_n)' = x_1'x_2'\dots x_n'$

Proof: We can accommodate this one by a mixture of mathematical induction and perfect induction. First we assume that the assertion is true for n variables. I.e.,

$$(x_1+\dots+x_n)' = x_1'\dots x_n{}' = f_n$$

Then $f_n' = n_1+\dots+x_n$, and our desired new function on the left is just $(f_n' + x_{n+1})'$. To show that this is the same as $f_n x_{n+1}'$, the desired function on the right, we can recognize that f_n itself is just a binary variable, even though a dependent one, and that we can examine the entire situation with a table of combinations on x_{n+1} and f_n. Thus

x_n+1	f_n	f_n'	$f_n'+x_{n+1}$	x_{n+1}'	$(f_n'+x_{n+1})'$	$f_n x_{n+1}'$
0	0	1	1	1	0	0
0	1	0	0	1	1	1
1	0	1	1	0	0	0
1	1	0	1	0	0	0

And since the last two columns in the table are identical, then the assertion holds for n+1 variables as well. Alternatively we could have used our previous theorem on two variables directly:

$$(f_n' + x_{n+1})' = f_n' x_{n+1}$$

which is the desired assertion on n+1 variables. Of course having already proved the two-variable case, it follows that the theorem is true for all n.

There is a useful further generalization of DeMorgan which can be shown, although because it is a notational nightmare we shall refrain from proving it. Succinctly stated it is the

Theorem: $(f(x_1, x_2, \ldots, x_n +, \cdot))' = f(x'_1, x'_2, \ldots, x'_n \cdot, +)$

and in words this means that any algebraic form (consisting not only of variables, but of the operators as well -- as parts of the form) can be transformed as indicated by complementing all of the variables and by substitution + for ·, and conversely. Thus for example

$$((x_1 x_2 + x'_3)x_4)' = (x'_1 + x'_2)x_3 + x'_4$$

directly without any further ado.

An important type of n-variable theorem accomplishes a different kind of end, and is involved with expressing an aribtrary function in terms of simpler functions, e. g., in terms of functions of a lesser number of variables. These are therefore called expansion theorems and will play an important role in what follows. The best known of these are the so-called Shannon theorems:

Theorem: $f(x_1, \ldots, x_i, \ldots, x_n) = x_i f(n_1, \ldots, 1, \ldots, x_n) + x'_i f(x_1, \ldots, 0, \ldots, x_n)$

Proof: By perfect induction: let $x_i = 0$ and get $f(x_i, \ldots, 0, \ldots, x_n)$ for both sides; let $x_i = 1$ and get $f(x_1, \ldots, 1, \ldots, x_n)$ for both sides. That covers the universe, hence the theorem is true in general.

Of course the dual form must be true as well, so we get immediately

Theorem: $f(x_1 \ldots, x_i, \ldots, x_n) = (x_i + f(x_1, \ldots, 0, \ldots, xn))(x'_i + f(x_1, \ldots, 1, \ldots, x_n))$.

These theorems play a very important role in logic design. Not only do they suggest immediate structural realizations of functions (in terms of systematic tree expansions) but they suggest alternative algebraic forms which we shall develop now. For what they affirm is that, given any function, we can systematically expand that function in terms of a succession of variable expansions. For example, expanding on x_1 and x_2 must yield

$$f(x_1, x_2, \ldots, x_n) = x_1 x_2 f(1, 1, x_3, \ldots, x_n) + x_1 x_2' f(1, 0, x_3, \ldots, x_n) +$$
$$x_1' x_2 f(0, 1, x_3, \ldots, x_n) + x_1' x_2' f(0, 0, x_3, \ldots, x_n)$$

and so on. Continuing through all n variables we must get nothing but simple product terms on the form

$$x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} f(i_1, i_2, \ldots, i_n)$$

all summed together. The i_i are simply the binary constants, hence the expression on the right side is an evaluation of the function for the given n-tuple, i.e., it is either a 0 or 1, while we'll use the convention on the x_i that

$$x_i^0 = x_i'$$

$$x_i^1 = x_i$$

in order to keep the books straight. Thus in general

$$f(x_1, \ldots, x_n) = \sum_{\text{all } i_1 \ldots i_n} f(i_1, \ldots, i_n) x_n^{i_1} \ldots x_n^{i_n}$$

and this form is called the disjunctive normal form.

For the three-variable case the complete expansion looks like

$$f(x_1, x_2, x_3) = f(1, 1, 1)x_1 x_2 x_3 + f(1, 1, 0)x_1 x_2 x_3' + f(1, 0, 1)x_1 x_2' x_3 +$$
$$+ f(1, 0, 0)x_1, x_2', x_3') + f(0, 1, 1)x_1' x_2 x_3 + f(0, 1, 0)x_1' x_2 x_3'$$
$$+ f(0, 0, 1)x_1' x_2' x_3 + f(0, 0, 0)\, x_1' x_2' x_3'$$

Product terms like the above wherein each of the variables enters into each product are called fundamental products, fundamental conjunctions, or minterms. The latter term refers to the representation of a fundamental product on the n-cube (or map). Since the map has only a single vertex marked, it follows that such a term is a minimum cube on the n-cube, hence the name. The entire summation is also referred to by various other names, including fundamental sum-of-products, disjunctive canonical form, etc. An important feature of this form is that it is unique for each f, that is there is only one fundamental sum-of-products representation for each f. (This is easily proved by contradiction--assume there are two different ones--then they must differ in at least one fundamental product--but then the two forms cannot evaluate to the same value for the independent variable combination that makes that product true--ergo there cannot be two different forms.)

Clearly there are 2^n different fundamental products, and since the f(i, j, k) are the binary constants, it follows that there are exactly 2^{2^n} different functions that can be represented by these forms, hence the numbers game is right and all functions are covered. Since the f(i, j, k) are just the evaluations of the function for the various possible independent variable values, they of course correspond to the entries in the table of combinations. Thus in general

x_1	x_2	x_3	f
0	0	0	$f(0,0,0)=f(000)=f_0$
0	0	1	$f(0,0,1)=f(001)=f_1$
0	1	0	$f(0,1,0)=f(010)=f_2$
0	1	1	$f(0,1,1)=f(011)=f_3$
1	0	0	$f(1,0,0)=f(100)=f_4$
1	0	1	$f(1,0,1)=f(101)=f_5$
1	1	0	$f(1,1,0)=f(110)=f_6$
1	1	1	$f(1,1,1)=f(111)=f_7$

where we have displayed any of the several notational forms that we might use to indicate the same constant.

Of course, the dual expansion also must hold ture, and we can assert that

$$f(x_1, \ldots, x_n) = \prod_{\text{all}} \left(f(i_1', \ldots, i_n') + x_1^{i_1} + \ldots + x_n^{i_n} \right)$$

and this form is known as the conjunctive normal form. It is obviously a product of sums where a sum is a fundamental sum, or maxterm.

2.2.5 Alternative Operators and Functional Completeness

We can, in a sense, generalize our operators in our switching algebra, and these generalizations will result in alternative forms for representing, and realizing switching functions as well.

Of course these possibilities correspond to those we have already observed in the propositional calculus, hence we can rapidly survey those of interest at this time without much detailed development.

The two-variable operators will all be contained, of course, within the general two-variable truth table of possibilities.

x	y	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

As we've observed before we can consider any of these to be binary operators. Whether we ultimately do so, of course, will depend upon other factors than algebraic manipulation, such as ease of physical realizability, etc.

We can recognize some of these operators already. Thus

$f_8(x, y) = xy$ and is our AND operator

$f_{14}(x, y) = x+y$ and in our OR operator.

$f_3(x, y) = x'$ and is our NOT

as is $f_5(x, y)$ as well. Of course f_0 and f_{15} we recognize as our constants.

Another useful one which we shall explore further in the next section is $f_6 = x \oplus y$, which is the exclusive-or operator, and $f_9 = x \equiv y = x \oplus y'$ which is the complement of f_6.

Finally we note

$$f_1(x, y) = x'y' = (x+y)' = x{\downarrow}y$$

which is the Peirce dagger function, or NOR operator, and

$$f_7(x, y) = x'+y' = (xy)' = x|y$$

which is the Sheffer stroke function, or the NAND operator. These two are particularly useful in that they are the only operators we need. Let's examine this notion a little further.

A set of operators S is functionally complete or logically complete if we can write an expression for arbitrary f using only the operators of the set. Of course we can write expressions as defined so far using only the operators "+", ".", and "'". But it is a simple thing to extend our algebra to include operators of any kind so long as the operator is clearly specified.

Thus, for example, the extended expression

$$xy \oplus (y'+x\downarrow z).$$

surely has meaning, and can be evaluated just like every other expression so long as we have cleared up matters like precedence and the like. Also we can manipulate them algebraically so long as appropriate theorems regarding such manipulation can be demonstrated. The reader is forewarned at this point that the "obvious" rules for algebraic manipulation do not necessarily hold for these operators, and the road to logic design is strewn with rejected designs of those who assumed that the obvious was true. As a simple example we can point to a couple of cases in point. If the following expression occurs

$$a(b \oplus c)$$

in an expression, one might be easily tempted to use the "equivalent" expression

$$(a \oplus b)(a \oplus c)$$

to achieve some desired simplification. But this expression of equivalence has not been proved, and the reader would use it at his peril. It happens to be true, and the unwary reader might then be tempted to assume that

$$a \oplus bc = (a \oplus b)(a \oplus c)$$

would be equally valid, and here he would be wrong. There are two obvious routes to take in order to avoid the pitfalls that are possible, other than simply agreeing to never use expressions in our augmented algebra. One would be to compile a large catalog of relevant theorems involving all the operators that might be of use to us, in all their possible arrangements. This is untenable, and would require another notebook to accommodate. The other is to be careful, each time we use such a transformation involving an unfamiliar operator, to verify that the transformation is indeed proper, and in most cases this can be verified quickly by the methods of proof we've pointed to already.

To return to our train of thought, we have thus already verified that the set $S = (+, \cdot, ')$ is complete by reason of the Shannon expansion theorems. Any f can be expressed using only those three operators.

This creates a situation that has always been very challenging to switching theorists. Given any finite set of things that is complete in some fashion, the question of minimally complete sets always arises. To this end we shall say that S is minimally complete if the elimination of any element of S yields a set that is not complete.

In this context we ask whether $S = (+, \cdot, ')$ is minimally complete? Echoing an answer we observed before we note that anytime we have

$$h = f+g$$

we can choose to write instead that

$$h = (f'g')'$$

so that in a structural sense anytime that we have the operator of Figure 2. 2. 5. 1

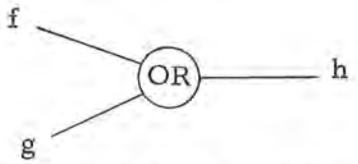


Figure 2. 2. 5. 1. OR operator.

we can always replace it by the operator of Figure 2. 2. 5. 2

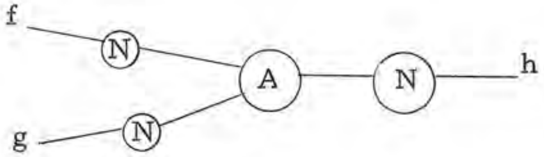


Figure 2. 2. 5. 2. Replacement of OR operator.

hence can always eliminate the + by the assembly of ANDs and NOTs. It follows immediately that $S = (., ')$ is complete. Since, try as we will, we cannot get $x \cdot y$ with the unary operator NOT nor can we ever generate x' with just an AND it follows that the set $(\cdot, ')$ is now minimally complete.

Similarly, based upon the other DeMorgan form, $S = (+, ')$ is also minimally complete as indicated by the substitution of Figure 2.2.5.3:

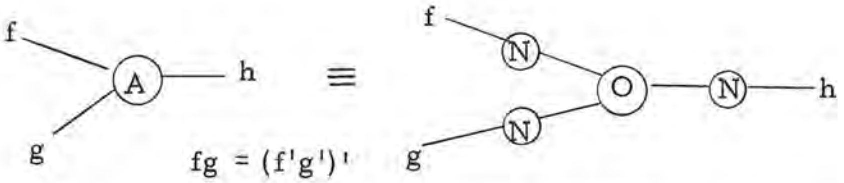


Figure 2.2.5.3. Replacement of AND operator.

Given, then, that there are minimally complete sets, and of course only a finite number of them we can then ask about the sets with the least number of elements. We have two candidates with only two. Are there any with only a single element? If so, they achieve the absolute minimum possible. To find these we use the general method of proof utilized above:

1) Given S is complete

2) Realize the elements of S using only elements of T

3) Then we can conclude that T is also complete.

It turns out that the NAND and the NOR are complete by themselves, and there are no others. The latter statement you'd have to work on a little to show; the former statement is easily verified by the methods we've already employed and summarized above.

<u>NAND</u>

The NAND is complete by the substitutions shown in Figure 2.2.5.4

$(xy)' = x'+y'$

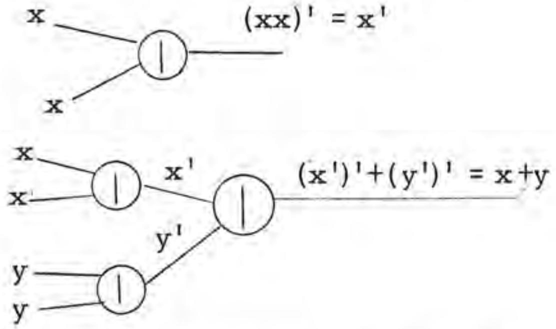


Figure 2.2.5.4. Completeness of the NAND

and it follows that since $S = (+, ')$ is complete, then so is $T = (|)$. I. e., $x' = x|x$; $x+y = (x|x) \mid (y|y)$.

NOR

In a similar fashion we note the substitutions of Figure 2. 2. 5. 5.

$(x+y)' = x'y'$

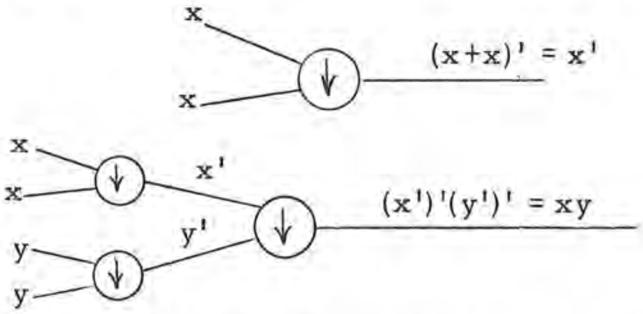


Figure 2. 2. 5. 5. Completeness of the NOR.

and since $S = (. , ')$ is complete, then so is $T = (\downarrow)$. I. e., $x' = x\downarrow x$; $xy = (x\downarrow x)\downarrow(y\downarrow y)$.

The switching theoretic implication is obvious. To realize any switching function one needs only NAND's, or NOR's, and nothing else. Although the implication is obvious, we should not draw any inference from the above other than that the possibility for doing so exists. I. e., we have asserted only the existence of logic design algorithms; the minimization of the number of operators (or gates) for a particular function is quite another thing, and this will occupy us subsequently.

2. 2. 6 Exclusive-Or Expansions

Another useful gate is the exclusive-OR, or mod-2 adder, and interesting possibilities also are possible using it. To work our way systematically toward the salient results we first recall the definitions

+	0	1
0	0	1
1	1	1

⊕	0	1
0	0	1
1	1	0

and we see that the + and the ⊕ are "almost" the same function. They are the same except for the specific case where the two arguments assume the truth value 1. To examine this particular exception, let us define two functions f and g as being <u>disjoint</u> iff $fg = 0$. That is two functions are disjoint if they never both assume the value 1 for the same set of the independent variable values. For example

x	y	z	f	g
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

certainly display two functions f and g that are disjoint.

Now if f and g are disjoint, it follows immediately that

$$f+g = f\oplus g$$

since it is simply impossible to arrange the independent variable values so that they are ever both 1 at the same time. It follows that in any expansion of the form

$$xf_1 + x'f_2$$

that

$$xf_1 + x'f_2 = xf_1 \oplus x'f_2$$

without further ado.

Next we note that n-variable conjunctions are eminently disjoint, e.g., that

$$(x_1x_2x_3)(x_1x_2x_3') = 0$$

hence that in the disjunctive normal form

$$f = x_1x_2x_3 + x_1'x_2x_3 + \dots \quad = \sum a_j\overset{*}{x}_1\overset{*}{x}_2\overset{*}{x}_3$$

we can immediately write that

$$f \quad x_1x_2x_3 \oplus x_1'x_2x_3 \oplus \dots \quad = \sum\!\!\!\!\!\!\!\bigcirc\; a_j\overset{*}{x}_1\overset{*}{x}_2\overset{*}{x}_3$$

since each term is disjoint from all the others. This result already gives a new canonical form, and affirms that $S = (\oplus, \cdot, ')$ is also a logically complete set.

So far this is rather trivial, but it does affirm that we can occasionally interchange two gates types without any caution but the observance of certain functional properties on the inputs to the gates.

Something quite different accrues, however, if we take but one more step, namely to affirm the following

Theorem: $x' = 1 \oplus x$

Proof:

x	x'	$1 \oplus x$
0	1	1
1	0	0

QED.

But this means that we can substitute as in Figure 2. 2. 6. 1

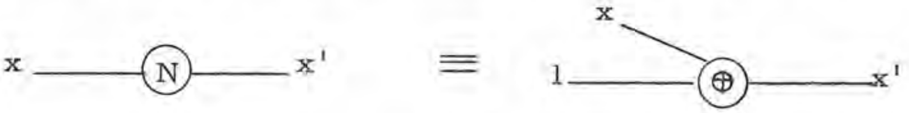


Figure 2. 2. 6. 1. Elimination of NOT operator.

at will, i. e., that we can eliminate all NOTs in a circuit (but we still need the constant) so that we conclude that yet another complete set is $S = (\oplus, \cdot, 1)$, and this leads to an interesting alternative canonical form for arbitrary functions.

We shall take the approach of concluding what the nature of the from is, and shall not now worry about transformations between the equivalent forms. If we proceed from the Shannon minterm form, we can clearly get nothing but product terms all ⊕ed together. Eliminating all the NOTs as in

$$x'_1x'_2x_3 = (1\oplus x_1)(1\oplus x_2)x_3 = x_3\oplus x_1x_3\oplus x_2x_3\oplus x_1x_2x_3$$

yields products of differing lengths, but only on the unprimed variables. Is the numbers game all right in this case? How many different kinds of product terms can we get? Exactly n for single variable products, $\binom{n}{2}$ for two-variable products, $\binom{n}{3}$ for three-variable products, etc., hence

$$\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$$

different kinds of products. Also one of the "products" might be the constant 1 itself. Each of these might be in any given expansion for n so that there must be a total of

$$2^{2^n}$$

different forms representable by such forms, which we knew we must have in order for all functions to be represented. Thus whereas the Shannon form is

$$f = \sum f(ijk)\, x_1^i x_2^j x_3^k \qquad \begin{aligned} x_i^j &= x_i \text{ if } i = 1 \\ &= x_i' \text{ if } i = 0 \end{aligned} \quad .$$

we now conclude the existence of an equally general form

$$f = \sum\!\!\!\!\!\!\oplus\; g(ijk)\, x_1^i x_2^j x_3^k \qquad \begin{aligned} x_i^j &= 1 \text{ if } j = 0 \\ &= x_i \text{ if } j = 1 \end{aligned}$$

Of course in any particular case we already have the machinery for generating the form. Thus, for example, if

$$\begin{aligned} f &= (1, 2, 4, 7) \\ &= x_1'x_2'x_3 + x_1'x_2x_3' + x_1x_2'x_3' + x_1x_2x_3 \\ &= x_1'x_2'x_3 \oplus x_1'x_2x_3' \oplus x_1x_2'x_3' \oplus x_1x_2x_3 \\ &= (1\oplus x_1)(1\oplus x_2)x_3 \oplus (1\oplus x_1)x_2(1\oplus x_3) \oplus x_1(1\oplus x_2)(1\oplus x_3) \\ &\qquad \oplus x_1x_2x_3 \\ &= x_3 \oplus x_1x_3 \oplus x_1x_2x_3 \\ &\qquad \oplus x_2 \oplus x_1x_2 \oplus x_2x_3 \oplus x_1x_2x_3 \\ &\qquad\quad \oplus x_1 \oplus x_1x_2 \oplus x_1x_3 \oplus x_1x_2x_3 \\ &\qquad\qquad \oplus x_1x_2x_3 \\ &= x_1 \oplus x_2 \oplus x_3 \end{aligned}$$

which is intuitively self-evident on examination of the original function, which is easily verified to be the general parity function of three variables.

On the other hand if the original form were not already disjoint, we can easily make it so by expanding within our conventional algebra until it is. For example

$$\begin{aligned} f = x_1x_2 + x_2x_3 &= x_1x_2x'_3 + x_1x_2x_3 + x_2x_3 \\ &= x_1x_2x'_3 + x_2x_3 \\ &= x_1x_2x'_3 \oplus x_2x_3 \\ &= x_1x_2\,(1 \oplus x_3) \oplus x_2x_3 \\ &= x_1x_2 \oplus x_1x_2x_3 \oplus x_2x_3 \end{aligned}$$

which is of the predicted form.

One nice thing about this form is the ready computation of complements. Thus if

$$f = x_1x_2 \oplus x_3 \oplus x_1x_3$$

then

$$f' = 1 \oplus f = 1 \oplus x_1x_2 \oplus x_3 \oplus x_1x_3$$

which is of the same form, and we're done.

Now there are a large number of similar canonic forms that we could derive based upon the other possible gate types that we've mentioned. For example, we could ask similar questions about the "equivalence" or "EQUIV" gate defined by

x	y	$x \equiv y$
0	0	1
0	1	0
1	0	0
1	1	1

In this case we could start from scratch, as we did from the $\oplus$ gate, or we could make use of what we already know and observe that

$$(x \oplus y)' = x \equiv y$$

or

$$x \oplus y = (x \equiv y)'$$

and that

$$x \equiv 0 = (x \oplus 0)' = x'$$

hence that

$$x \oplus y = (x \equiv y)' = x \equiv y \equiv 0$$

From this it follows that wherever we have a $\oplus$ gate we have the equivalence shown in Figure 2.2.6.2,

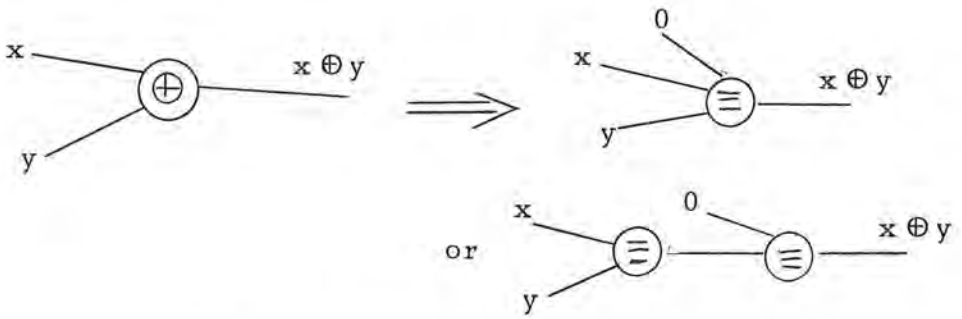


Figure 2.2.6.2. Exclusive-OR replacement.

and for every NOT we can equate the forms of Figure 2.2.6.3.

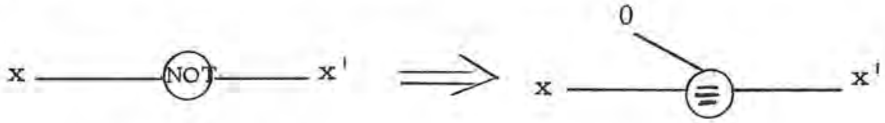


Figure 2.2.6.3. NOT replacement.

It follows then that there must be a canonic form consisting of AND's and EQUIV's only, as well as the constant 0. That is, just as earlier we concluded that $S = (\oplus, \cdot, 1)$ must be a complete set, so must $S = (\equiv, \cdot, 0)$ be a complete set. And for any such set we could detail the appropriate theorems and canonic forms to any degree we would wish. We could do similarly for other gate types and combinations as well.

But it is not our intention to do so. Instead it is only our intention to explore such possibilities to a depth sufficient to convince the reader that he can always build his own catalog of pertinent facts relative to the particular logic design job he might be faced with in the future. I.e., it is clear that the designer can always build his own handbook of design tools (theorems?) depending upon the particular freedoms and restrictions that he is faced with.

2.3 Switching Functions and Their Representations

We have tried to make some distinction between switching functions and the multitude of different representations that are possible for each one. The point has probably been overmade, yet in a context where there are so many different, ambiguous representations of the same thing, it seems important to distinguish between the function itself, which is well defined, immutable, unique, stable, reliable, etc., etc., and its various representations, within which most of the problems lie.

In this section we shall attempt to specifically define what a switching function is, we shall attempt to illustrate in an elementary way that most functions are just about as complex as they can be, and finally we shall catalog in a systematic way at least the most common forms with which switching functions are represented.

2.3.1 Basic Definition

Although there are many, many representations of switching functions, a catalog of these representation types is confusing without first agreeing on just what the fundamental definition of what we are talking about is. For it is in these distinctions that the ultimate engineering nature of the switching theory is revealed. Although a switching function is no different from any other mathematically defined function, it is always in its representations that the problems arise since it is the representations that correspond with physical realizations.

As a reference point, however, we shall define a <u>switching function</u> conventionally as a mapping from the set of n-tuples to the values of the switching constants, 0 and 1. Thus

$$f(x_1, \dots, x_n) = [0, 1]^n \longrightarrow [0, 1]$$

where the domain of the mapping is the binary n-tuples, and the range of the function is simply the constants.

If the domain of the mapping involves the entire set of 2^n n-tuples, then we have a <u>completely specified</u> function. Since there are 2^n points in the domain, and since each of these can be assigned independently to either of the constant values, there are clearly

$$2^{2^n}$$

switching functions of n variables (corresponding to the n^{th} Cartesian product of the single variable domain [0, 1]); hence the numbers game is again verified from this point of view.

The only reason for distinguishing completely specified functions is again of engineering relevance, rather than mathematical completeness. For if the entire set of n-tuples is not specified, then we have implications in the physical realization that may lead to simplifications in the physical realization.

In any event, if the entire set of n-tuples is not specified, then we have an incompletely specified function, and in the engineering context this simply means that it is not of consequence to us what the value of the function is for certain of the domain points. These correspond to input values that are known as "don't care" values, and they can sometimes lead to reduction in the physical complexity.

For example, consider the contact network of Figure 2.3.1.1

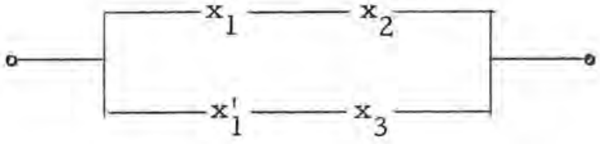


Figure 2.3.1.1. Example contact realization.

which we can algebraically (or otherwise) verify corresponds to the truth table column for f:

x_1	x_2	x_3	f	f_R
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	1 ⟵
1	1	0	1	1
1	1	1	1	1

In our understanding, then, the above contact realization is a particular representative of the function f. Now suppose that f were not specified for every set of input variables, e. g., suppose that f(101) were irrelevant to us. (The reason for this irrelevance might be any of a number: perhaps in the physical context it is known that the particular set of input values is precluded by some other event from ever happening; perhaps the consequence of input 101 is also to literally destroy the network itself, rendering it inconsequential what the output of the network itself is for the particular set of inputs.) Whatever the reason, if f(101) is not of importance, then the contact network of Figure 2. 3. 1. 2

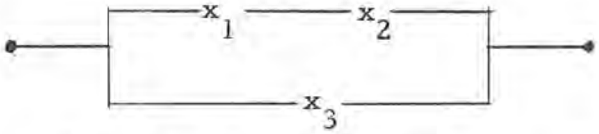


Figure 2. 3. 1. 2. Simplified contact realization

is also a realization of the incompletely specified f that lists 101 as a don't care condition. Of course the network itself does "care" for each point, and results in the column labeled f_R in the previous truth table.

Since it corresponds with f in every row except that don't care row, however, it is also a realization of the function f. Since it's cheaper, it is to be preferred, and our freedom to use it is warranted by the incomplete specification on our function.

It is evident, then, that the incompletely specified functions are a larger class that includes the completely specified functions. From the mapping point of view they correspond to including a third range point, say D for don't care, and can be counted

$$f(x_1, \ldots, x_n) : [0, 1]^n \longrightarrow [0, 1, D]$$

as including a total of 3^{2^n} different functions.

2. 3. 2 Function Complexity

There is some reason to suspect that the complexity of switching functions might not be as great as it might be, for we have seen that some portion of the functions of n variables include those of a lesser number of variables. These are called the degenerate, vacuous, or redundant functions. On the other hand if a function of n variables is truly dependent upon all n of its variables then it is called nondegenerate, or nonvacuous, or irredundant. Certainly if most of the functions of n variables are degenerate, then our problem as a function of n is not nearly as great as might be. We can examine this question directly by counting the number of functions involved.

As a first step, since everything is finite, we might simply start experimenting with increasing n. For $n = 0$ we have the two constants. For $n = 1$ we have seen that there are the two irredundant functions x and x', and then the two constants 0, 1 to make up the total of 4 functions. For the case $n = 2$ we can still successfully proceed in this fashion to classify the 2^{2^2} functions. We will find that six are redundant, i.e., 0, 1, x, x', y, y' and the rest are not:

$$xy,\ x'y,\ xy',\ x'y'$$
$$x+y,\ x'+y,\ x+y',\ x'+y'$$
$$xy'+x'y,\ xy+x'y'$$

At this point we should have exhausted our patience, and if one is not completely convinced of this, one should simply try to continue on and examine the case for $n = 3$ by inspection alone. There are 256 such functions, and the systematic examination of such would consume some little time. Fortunately we can call upon a little more sophistication at this point and observe first that the total number of functions for any case is made up of several contributors:

$$\text{Total Number of Functions} = 2^{2^n} = \text{Total number nondegenerate} + \text{Total number degenerate}$$
$$= N(n) + D(n)$$

where

$$D(n) = \text{Degenerate of } n-1 \text{ variables} + \text{degenerate of } n-2 \text{ variables} + \ldots + \text{degenerate of 1 variable} + \text{degenerate of 0 variables}$$

Now since the number of functions of n-1 variables that will contribute to the degenerate functions of n variables must be just

$$\binom{n}{n-1} N(n-1)$$

and of n-2 variables

$$\binom{n}{n-2}N(n-2)$$

and so on, it follows that we must be able to enumerate them all as

$$D(n) = \binom{n}{n-1}N(n-1) + \binom{n}{n-2}N(n-2) + \ldots + \binom{n}{0} N(0)$$

and since

$$\binom{n}{n}N(n) = N(n)$$

we can express the whole thing in tidy form as

$$2^{2^n} = \sum_{i=0}^{i} \binom{n}{i}N(i) = C(n) \quad .$$

This enables us to bootstrap our results up to any n of interest, for each N(n) depends upon knowledge of the previous N(i) as in

$$N(n) = 2^{2^n} - \sum_{i=0}^{n-1} \binom{n}{i} N(i)$$

Using this formula to check our previous results we can confirm that

$$N(2) = 2^{2^2} - \binom{2}{1}N(1) - \binom{2}{0} N(0)$$

$$= 16 - (2)(2) - (1)(2) = 10$$

for example, and that our next value

$$N(3) = 2^{2^2} - \binom{3}{2}N(2) - \binom{3}{1} N(1) - \binom{3}{0} N(0) = 256 - 30 - 6 - 2 = 218$$

and so on. Carrying on for a few more values as in the table below

n	C(n)	N(n)	$\frac{D(n)}{C(n)}$ 100%
1	4	2	50
2	16	10	37.5
3	256	218	14.8
4	65,536	64,594	1.43
5	4,294,967,296	4,294,642,034	0.75
	⋮	⋮	⋮

we see the salient fact of this discourse emerging rather rapidly. Namely that $N(n)$ becomes the dominant contributor to $C(n)$. (It can be shown that as n grows, $N(n)$ asymptotically approaches $C(n)$. It follows that, at least from the point of view of variable dependency, that almost all switching functions are just as complex as they can be.

2.3.3 Switching Expressions

It is clear that an expression needs to be carefully distinguished from a function. Any of the algebraic forms that we've been considering can be used to express the same switching function. In fact a moment's consideration will reveal that there are many more forms (and certainly we can include graphic, n-cube, and other circuit representations as alternative forms that can be used to express a switching function) than there are functions.

In general, then, a switching expression is any finite combination of the symbols x_i, 0, 1, +, $\cdot$, $'$. Defined recursively:

1. $x, y, \ldots, 0, 1$ are switching expressions.
2. If S_1, S_2 are switching expressions, then so are S_1+S_2, $S_1 \cdot S_2$, and S_1'.
3. Nothing else is a switching expression.

This really doesn't limit things much, but does give a nice, tight sort of feeling about the whole thing. Thus, for example

$$x = x + x = x + x + x = x + x + x + yy'$$

all represent the same switching function, while

$$x(x'+y) = xy = xy(z+z') = \ldots$$

is another infinite set of possibilities representing yet another function.

Clearly, then, each switching function has a large number of representations, including algebraic forms, and we can conclude that the functions themselves form a sort of equivalence class upon their own possible representations. The only reason for introducing this notion at all is that whenever one considers a class of equivalent things, one almost always looks for a class representative of some sort to represent the class. This is usually a member that is the simplest representative.

Except on purely abstract grounds, this simplest member is always chosen in response to the engineering question that involves the notion of "cheapest". A point to be made here, though, is that the notion of "cheapest" is a function of "context", and until the context is defined, then there is no rational reason for asserting that one class member is to be preferred to any other.

For example, probably no argument would be forthcoming about the assertion that of the equivalence class that includes

$$x = x + x = x + x + x = \dots$$

that the single member x is clearly the cheapest member. Often the criterion, that the above satisfies, that the cheapest member is the one that offers the least number of variable occurrences, is satisfactory. Thus we would get few quarrels in our application of the algebra in simplifying

$$\begin{aligned} xy + x'yz' + yz &= \\ y(x + x'z') + yz &= \\ y(x + z') + yz &= \\ yx + yz' + yz &= \\ yx + y(z'+z) &= yx + y = y \end{aligned}$$

and asserting that the manipulation has resulted in a minimal form.

But on the other hand, consider

$$\begin{aligned} &(x_1 + x_2(x_3 + x_1x_4'))' \\ &= (x_1 + x_2x_3 + x_1x_2x_4')' \\ &= (x_1(1+x_2x_4') + x_2x_3)' \\ &= (x_1+x_2x_3)' \qquad \text{(A)} \end{aligned}$$

and again we should probably agree that the above form, which has only three variable occurrences, is certainly simpler than the one we started with.

Yet

$$x_1'(x_2' + x_3') \qquad \text{(B)}$$

$$x_1'(x_2x_3)' \qquad \text{(C)}$$

$$(x_1 + (x_2' + x_3')')' \qquad \text{(D)}$$

are evidently equivalent expressions for the same function using only the simple algebraic theorems we've demonstrated. They all achieve the same number of literal occurrences. But suppose that it were very expensive to build the "'" operator. Then surely (A) is the best expression. On the other hand, if the physical realization of the expression puts a high valuation on ANDs, then certainly (D) is the cheapest, for it requires no ANDs at all. Similarly, (C) requires no OR's, etc. Clearly, then, the notion of "simplest" expression is one that depends on what you have in mind, and without further directions than simply "find the simplest expression for ...", one expression is as "good" as another.

This has been an attempt to label the "minimality" question as strictly one having engineering content. It is reassuring, perhaps, that obvious simplifications in the algebra do have physical counterparts. We indicate one such in the following.

2.3.4 Contact Network Applications

In simple contact networks (incidentally, with respect to contact networks, the word "simple" has a precise definition: a simple contact network is one composed of simple contacts as opposed to "transfer" contacts; the reader is warned that simple contact networks are not necessarily simple) the number of literal occurrences in an algebraic expression does directly correspond with the number of contacts required, hence with the most likely element of cost of such a network. Thus every algebraic expression (with primes on the variables only!) that results in an equivalent expression yields a different contact network that realizes the same switching function. And the expressions with the least number of variable occurrences are clearly the cheapest in terms of these simple contacts. Thus not only do the theorems have direct applications as in Figure 2.3.4.1

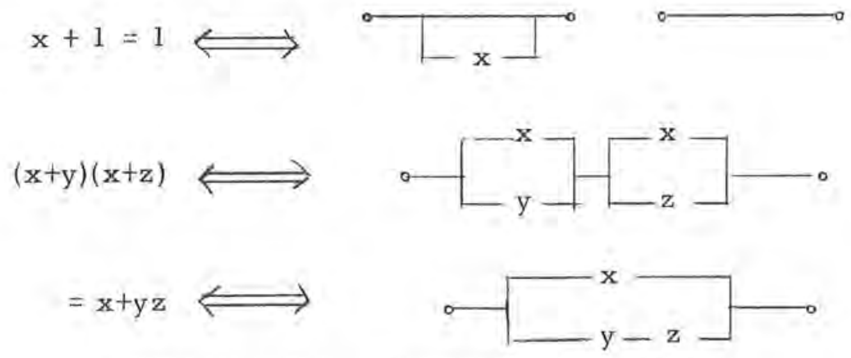


Figure 2.3.4.1. Theorem applications.

but so also do the result of long strings of algebraic manipulations have direct contact counterparts that exhibit different costs. Thus in Figure 2.3.4.2 the algebraic manipulations

$$x_1x_2 + x_1'x_3 + x_2x_3 \Longrightarrow$$

$$= x_2(x_1+x_3) + x_1'x_3$$

$$= x_1x_2 + x_1'x_3$$

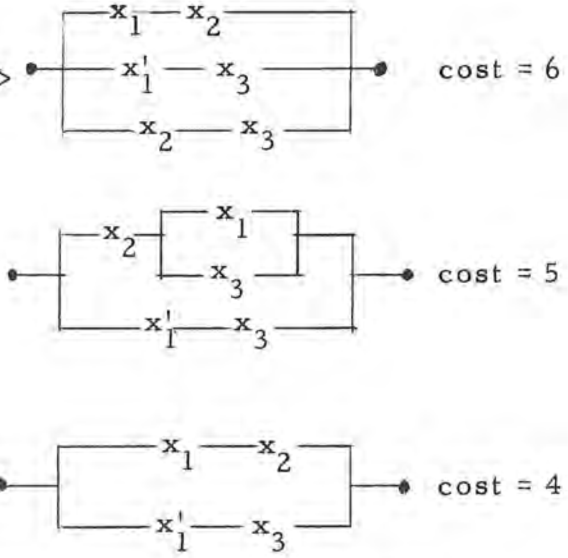


Figure 2.3.4.2. Equivalent networks with different costs.

yield several different networks, all realizing the same function, but at different costs.

2.4 Alternative Representations

Although our pristine definitions of a switching function as a mapping of the n-tuples onto the range 0, 1 is correct and complete, it is practically almost worthless. For we always deal with switching functions in terms of their more useful realizations as algebraic expressions, or maps, or the like. It would seem useful at this point to pause and catalog the various ways in which we shall represent switching functions, each of which has some utility in the practical embodiment of such functions.

2.4.1 Algebraic Representations

As we have seen, any expression in our algebra is a switching expression, hence is a representation of a switching function. As appropriate we shall augment our algebra to include other operators where convenient or useful. For now, however, we shall simply point to a typical expression

$$f(x_3, x_2, x_1) = x_2 x_1 + x_3 x_1'$$

as being such an expression, and we shall then use this function as a running example in confirming the nature of our other representations.

2.4.2 Tabular Representations

These are characterized by our now familiar <u>truth table</u>, or <u>table of combinations</u>:

x_3	x_2	x_1	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Although this representation should be thoroughly familiar by now, there are a few more things to be said about it in order to reduce possible confusion. In the first place, it is usually the case that the rows are arranged as above, that is with the normal binary ordering on the values of the independent variables. This need not be the case, however, and when it suits our purposes we shall reorder the rows in any way convenient to us. One such way that will arise involves an ordering of the rows so that those of the same weight are adjacent as in

x_3	x_2	x_1	f	
0	0	0	0	rows of weight 0
0	0	1	0	rows of weight 1
0	1	0	0	
1	0	0	1	
0	1	1	1	rows of weight 2
1	0	1	0	
1	1	0	1	
1	1	1	1	rows of weight 3

in which the natural binary ordering is not preserved in order to cluster the rows of like weight together. Of course the column for f is assigned in the same way in order to retain the identity of f.

Another point that needs to be made is that some agreement must be made about how the variables themselves are to be ordered in any particular instance. Thus, in the above table, we have assigned to variable x_3 the position of greatest weight when we interpret the positions in the normal binary ordering for the 8421 weighted code. Obviously if we had assigned x_1 this position instead we would get something representationally different. The reason for sometimes arranging the table as in the above is that in some arguments it is easier to augment from n to n+1 variables in the table if x_n is assigned the column on the left. On the other hand, given a fixed number of variables, it is natural to arrange the ordering starting with $x_1, x_2, \dots$. The same possible confusion arises when we write $f(x_1, \dots, x_n)$ in some instances, and $f(x_n, \dots, x_1)$ in others. The prime fact to remember is that these are representational inconveniences, and of course they do not affect the underlying switching function at all. The practical way out of the difficulty is to agree beforehand in any discourse just what the ordering is to be. Of course the particular ordering will greatly affect any representation that interprets the rows of the table in terms of their binary number equivalents, as will be seen to be the case in the following.

2.4.3 The Characteristic Number

If we agree to use the conventional binary ordering in the truth table, then of course the only portion of the table that transmits any information is the column for f itself, the so-called "output" column. Since this is the case, we need only reproduce that column in order to prescribe f. This results in the characteristic number, or the characteristic vector for f. This can be denoted C(f) and is usually laid over on its side. Thus for our running example

$$C(f) = (00011011)$$

2.4.4 The Decimal Sum Form

Obviously, since it suffices to simply list those rows where the output column for f has 1's in order to completely define f, and since we can assign a decimal number to each of those rows corresponding to the decimal equivalent of each row interpreted as a binary number, we can specify f simply by listing those rows. Thus for our running example we have

$$f = \Sigma\,(3, 4, 6, 7)$$

where we need to justify the meaning of the "Σ" sign, and we will do that immediately in the following section. The representation is also often written

$$f = \Sigma\, m(3, 4, 6, 7)$$

where m stands for minterm. This will also be apparent in the following section. Suffice it for now to observe that the "m" is completely redundant, so we shall choose to omit it. At any rate, such a representation that simply lists those rows of the truth table, in decimal form, is called a decimal sum form, and is seen to be not really very sophisticated, although widely used. It should be noted again, however, that the form is meaningless until we have agreed on the ordering of the variables in the representation.

2.4.5 The Disjunctive Normal Form

This is another algebraic form and follows immediately from the decimal sum form above, or from our earlier arguments on the meaning of the Shannon expansion theorems. Since each row of the table corresponds to a fundamental product, or minterm, it follows that there exists a *disjunctive normal form* which is a summation of fundamental products corresponding one-for-one with the rows of the truth table marked with a "1". This form can be read directly from the decimal sum form, for example, and we get

$$f = \Sigma\,(3, 4, 6, 7) = x_3'x_2x_1 + x_3x_2'x_1' + x_3x_2x_1' + x_3x_2x_1$$

as the disjunctive normal form for our running example. Of course we could have gotten the same result algebraically as in

$$\begin{aligned} f = x_2x_1 + x_3x_1' &= (x_3' + x_3)x_2x_1 + x_3(x_2' + x_2)\,x_1' \\ &= x_3'x_2x_1 + x_3x_2x_1 + x_3x_2'x_1' + x_3x_2x_1' \\ &= x_3'x_2x_1 + x_3x_2'x_1' + x_3x_2x_1' + x_3x_2x_1 \end{aligned}$$

which confirms our results.

As noted before, the principal consequence of the disjunctive normal form is that it is unique to each function. Hence it forms a convenient starting place for many synthesis techniques. Any sum-of-products which is not in the normal form is called a reduced sum-of-products, and the original specification we started with for f is an example. The reduced forms are *not* unique in general.

2.4.6 The Conjunctive Normal Form

By dualism we know that since there is a normal sum-of-products form for every f that is unique, it follows that there is a normal product-of-sums form for f that is likewise unique. We can quickly decide its nature by recalling that the form for three-variables looks like

$$\begin{aligned} f(x_3, x_2, x_1) = (x_3 + x_2 + x_1 + f(000))(x_3 + x_2 + x_1' + f(001)) \\ (x_3 + x_2' + x_1 + f(010))\ldots(x_3' + x_2' + x_1' + f(111)). \end{aligned}$$

Thus the *conjunctive normal form* consists of a product of fundamental sums, and it remains only to confirm exactly which of the fundamental sums appears in the expression for a given f. Since for any sum a we have

$$a + 0 = a$$
$$a + 1 = 1$$

we see that only those terms will appear in f that correspond with "0's" in the truth table representation. Thus, in a sense, while the disjunctive normal form yields a listing of the "1's" of f, the conjunctive normal form is a listing of the "0's" of f. Furthermore, notice that the binary constant appearing in each fundamental sum appears in complemented form from the way that the variables appear. Thus $f(110) = 0$ means that $x_3' + x_2' + x_1$ appears in the final product. With these provisions we confirm for our running example that

$$f = (x_3+x_2+x_1)(x_3+x_2+x_1')(x_3+x_2'+x_1)(x_3'+x_2+x_1')$$

corresponding to the 0's of f that exist in rows 0, 1, 2, 5.

As noted before, these fundamental sums are also called maxterms, in distinction from the previous minterms. Any fundamental product of sums that does not consist of maxterms is called a reduced product-of-sums. For the present f

$$f = (x_2+x_1')(x_3+x_1)$$

is an example of such, and can easily be generated from the conjunctive normal form by application of our algebraic theorems.

2.4.7 The Decimal Product Form

This is of course a direct extension from our decimal sum form, and is simply a listing of the rows of the truth table that correspond with the 0's of f. Thus it corresponds to the conjunctive normal form for f. Obviously it lists those rows that are not in the decimal sum form (since, above all, f must be well defined). Thus in our running example since

$$f = \Sigma\,(3, 4, 6, 7)$$

it follows immediately that the decimal product form must be

$$f = \pi\ (0, 1, 2, 5)$$

and of course these must correspond one-for-one with the terms observed previously in the conjunctive normal form.

2.4.8 Geometric Forms

We can also usefully conclude that all of these representations are simply other ways of looking at the vertices of the n-cube. Clearly, whenever it suits our purposes, we can designate f by marking the appropriate vertices on the cube directly as in Figure 2.4.8.1.

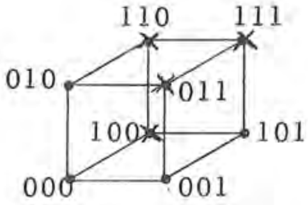


Figure 2.4.8.1. Representation on the n-cube

There is a variant of the n-cube that is called the lattice that is sometimes of use. In effect, to create the lattice we simply pick up the cube by its vertex of greatest weight, and let the rest of it "hang". The result is that vertices of equal weight are displayed on the same level. Thus Figure 2.4.8.2

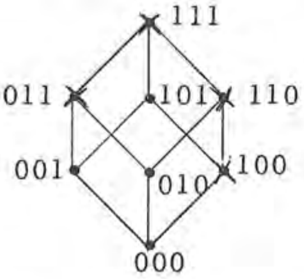


Figure 2.4.8.2. The lattice representation.

is the 3-cube lattice. The utility of the lattice will await further arguments, but it is clearly related to reordering of the truth table according to rows of equal weight.

Finally, of course, we mention in our catalog the transformation of the cube that results in the vertices being represented by cells in the map. These representations are the famous Karnaugh maps, and we simply recreate Figure 2.4.8.3 as the map

x_2

x_3

1	1		1
	1		

Figure 2.4.8.3. The Karnaugh map.

of our running example. The only point worthy of additional note here is that these maps need only satisfy the fundamental restriction that each variable must be associated with half the map, and that each such half must intersect the other halves in exactly half the nodes of the cube. I.e., the physically evident fact that various rotations of the cube do not change the fundamental properties of the cube is true. The consequence, however, is that Figure 2.4.8.4

x_4

x_1

x_2

0	4	12	8
1	5	13	9
3	7	15	11
2	6	14	10

x_3

Figure 2.4.8.4. A four-variable Karnaugh map.

with the minterms decimally indexed as shown, is the same cube as Figure 2.4.8.5.

x_3

x_1

x_4

0	2	6	4
1	3	7	5
9	11	15	13
8	10	14	12

x_2

Figure 2.4.8.5 Alternative four-variable map.

with the minterms indexed accordingly. This sort of arbitrariness causes no end of confusion in the minds of students who insist that a map must be oriented in exactly the way they first saw it in their first textbook. The salient philosophical conclusion is the realization that it doesn't matter at all, and the useful operational conclusion is that that each user of such maps should select a convention convenient to himself, and then use it consistently thereafter.

2.4.9 Function Measure or Weight

All the previous representations of a switching function have uniquely specified the particular function of interest. There are other characterizations that do not do so, yet are useful representatives of particular features of switching functions. One of these is the notion of weight or measure. The weight of a function or its measure w(f) is simply the number of 1's in the output column of the function. Thus

$$w(f) = \text{Number of 1's in } C(f)$$

and the weight of the fraction in our running example is

$$w(f) = 4.$$

It is clear that the w(f) does not uniquely specify a given function, but it does characterize a class of functions. We can list a few properties of the measure:

$$w(f') = 2^{n} = w(f)$$

$$w(f+g) = w(f) + w(g) - w(fg)$$

$$w(f \oplus g) = w(f) + w(g) - 2w(fg)$$

and so on.

Now the measure of a function turns out to be an equivalence relation on the entire class of functions, and it is in this role that it plays it most significant part. Thus we can quickly check that

1. $w(f) = w(f)$ (reflexivity)
2. $w(f) = w(g) \Longrightarrow w(g) = w(f)$ (symmetry)
3. $w(f) = w(g),\ w(g) = w(h) \Longrightarrow w(f) = w(h)$ (transitivity)

and it follows that the measure w(f) divides all switching functions into nonoverlapping classes, hence that

$$2^{2^n} = \sum_{i=0}^{n} \text{No. of functions of weight } i$$

and in several cases we shall find this a convenient way to partition the totality of switching functions.

2.5 Notes - References - Problems

NOTES.

Modern day switching theory and logic design, an admitted adaptation of the propositional calculus, dates back to 1938 when Shannon (8) modified the formalism in an appropriate way for relay contact circuits, and later extended the results to important synthesis techniques (9). The first massive, practical treatment of switching circuits is found in Keister, Ritchie and Washburn (4), while the first, carefully prepared analytical text on the subject is that by Caldwell (1). A thorough treatment of the algebra, with some applications, is found in Hohn (2), and since then many modern texts have appeared (3, 5, 6, 7, 10) and these will be found to give many further entries to the work done.

REFERENCES.

1. Caldwell, S. H., "Switching Circuits and Logical Design," John Wiley, New York, 1958.
2. Hohn, F. E., "Applied Boolean Algebra: An Elementary Introduction," (Second Edition), Macmillan, 1966.
3. Hill, F. J. and Peterson, G. R., "Introduction to Switching Theory and Logical Design," John Wiley, 1968.
4. Keister, W., Ritchie, S. A., and Washburn, S., "The Design of Switching Circuits," D. Van Nostrand, New York, 1951.
5. Kohavi, Z., "Switching and Finite Automata Thoery," McGraw-Hill, 1970.
6. Marcus, M. P., "Switching Circuits for Engineers," Prentice-Hall, Englewood Cliffs, N. J., 1962.
7. McCluskey, E. J., "Introduction to the Theory of Switching Circuits," McGraw-Hill, New York, 1965.
8. Shannon, C. E., "A Symbolic Analysis of Relay and Switching Circuits," Trans. AIEE, vol. 57, pp. 713-723, 1938.
9. Shannon, C. E., "The Synthesis of Two-terminal Switching Circuits," Bell Syst. Tech. J., vol. 28, pp. 59-98, 1949.
10. Torng, H. C., "Introduction to the Logical Design of Switching Systems," Addison-Wesley, Reading, Mass., 1964.

PROBLEMS.

1. Design a three switch, light circuit (out of switches A, B, C) such that the light L changes state whenever any switch is thrown.

2. Show that

   (a) $(x+y'+xy)(x+y')x'y = 0$

   (b) $(x+y'+xy')(xy+x'z+yz) = xy+x'y'z'$

   (c) $(AB+C+D)(C'+D)(C'+D+E) = ABC'+D$

   In each step of these demonstrations, which you should do algebraically, please refer to the algebraic theorem you are using.

3. Simplify the following algebraic expressions. For the first six, indicate which theorems you use in effecting the simplification:

   (a) $xy + xyz + yz$

   (b) $xy + xy'z + yz$

   (c) $xy + x'yz' + yz$

   (d) $(xy' + z)(x + y')z$

   (e) $xy' + z + (x + y')z$

   (f) $(xy' + z)(x' + y)z'$

   (g) $xy' + z + (x'y)z'$

   (h) $(x + y')(y + z')(z + x')(xyz + x'y'z')$

4. Prove the following theorems of switching algebra (i) by perfect induction and (ii) by direct applications of previously proved theorems in only one or two variables:

   (a) $xy + yz + zx = (x + y)(y + z)(z + x)$

   (b) $x'y'z' + x'y'z + x'yz + xyz = (x' + y)(y' + z)$

   (c) $(a + b'c'd')(a' + bcd) + (d + abc)(d' + a'b'c') = abc + a'b'c'$

5. Determine whether $ab + a'b' + bcd = ab + a'b' + a'cd$ by any means you wish.

6. Given the switching function $f(w, x, y, z) = xz+w'y'z'+wx'y'$ determine: (assume the variables are ordered w, x, y, z with w the highest weight)

   (a) the truth table representation

   (b) the characteristic number

   (c) the decimal sum form

   (d) the decimal product form

   (e) the disjunctive normal form

   (f) the conjunctive normal form

(g) the Karnaugh map

(h) w(f)

7. Is S = ($\supset$, G) a logically complete set? If so, prove it. If not, develop a function that cannot be realized by this set. The symbol $\supset$ refers to a two variable operation defined by

x	y	$x \supset y$
0	0	1
0	1	1
1	0	0
1	1	1

8. Express the following switching function and its complement (a) in conjunctive normal form, (b) in disjunctive normal ford, (c) in a Reed expansion, (d) in decimal representation, (e) as a characteristic number, and (f) as a set of marked vertices on the four-cube;

$$f = \overline{w}x\overline{y}z + \overline{x}\overline{y}\overline{z} + wy(w + \overline{z})$$

9. Find the complements of the following functions:

(a) $f = a + bc$
(b) $f = (a + b)(a'c + d)$
(c) $f = ab + b'c + ca'd$

Prove that your answers are correct by showing that $ff' = 0$ and $f + f' = 1$.

10. Prove that the exclusive OR operation is commutative, and associative, and also that the AND operation is distributive with respect to it.

11. Construct Karnaugh maps for

(a) $f = X_1X_2 + X_1X_3 + X_2X_3$

(b) $f = X_2'X_3'X_4' + X_1'X_3'X_4' + X_1'X_2'X_4' + X_1'X_2'X_3'$

(c) $f = X_1'(X_2X_3 + X_4)$

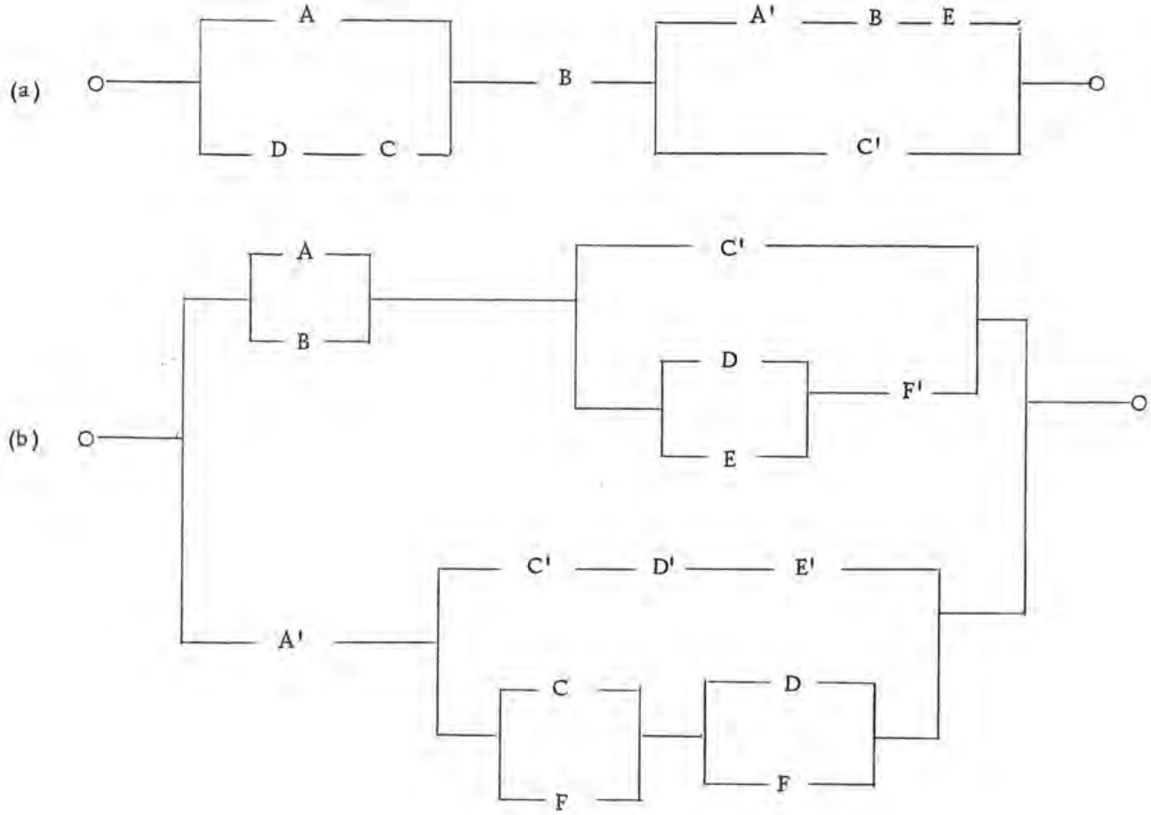
(d) $f = X_1 \oplus X_2 \oplus X_3 \oplus X_4$

12. Prove by mathematical induction on n that the function

$$f(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

is equal to 1 if and only if an odd number of the variables $x_1, x_2, \ldots, x_n$ are equal to 1. (The symbol $\oplus$ stands for exclusive - or.)

13. Write the transmission function (i. e., the algebraic statement representing a closed circuit) for networks (a) and (b) below. Form the complements of each, and draw the networks represented by the resultant transmission functions.



14. A switching function f of n variables is called a neutral function if and only if w (f) = 2^{n-1}, where w(f) is called the weight of f. How many neutral functions of n variables are there?

15. A function f is called symmetric if and only if all minterms of a given weight are either contained in f, or not, for each possible minterm weight. (the weight of a minterm is simply the number of 1's in the binary representation of the minterm; e. g., w'xyz is minterm 0111 and is of weight 3.) How many symmetric functions of n variables are there?

3 Elements of Logic Design

We now turn our attention to the organization of physical networks that will accomplish the realization (i. e., another alternative representation) of switching functions. We shall be concerned with the organization or layout of such networks, and not at all with their electronic realization (or mechanical, or fluidic, etc., etc.)

Such organizational aspects of networks are what is usually referred to as logic design. Thus what is involved is the application of the representations that we have been considering to the realization of networks under the various constraints under which the designer of physical systems must work. Logic design, then, is essentially an engineering discipline because it always must be subservient to questions of cost, and the "goodness" of any particular design must always be measured in terms of cost.

In any discussion of logic design, then, one must always proceed from a well defined base of just what the nature of the devices are, and just what it is in the physical construct that is important in terms of cost. For such cost criteria are the only bases for deciding whether any given design is superior to others, or not.

Frequently the principal cost criteria have concerned total number of elements involved, or total number of different element types, or minimal speed of network response. More recently the appropriate criteria have not involved the total number of elements, but rather total area required by a layout, the regularity of the layout, its reliability by some appropriate measure, etc., etc. The point is that logic design is simply meaningless without first agreeing on just what these criteria and limitations are.

Thus the topic of logic design is one with endless variety, just as with other aspects of switching theory that we've noted previously. We can only hope to give a few examples in order to illustrate how our basic theory can be applied in the practical situation. The particular technique that will be of ultimate interest to the reader will depend upon the particular constraints he is faced with as a result of the technology of interest at the time of application. Only with small probability will that technique be a minor variation on any of the approaches we shall summarize. But it is hoped that the basic techniques in boxing in a technical problem will serve as launching points for the development of other logic design techniques when their need arises.

3.1 Branch Networks

Combinational networks derive their name from the fact that their outputs are strictly functions of the particular combinations appearing on their inputs, i.e., time plays no significant role. Combinational networks in turn are of two principal types; branch networks and gate networks. (The question is not idle: an early paper in switching theory addressed itself to exactly this question: whether there could be other kinds of networks than the two we understand presently as represented by the branch and gate networks; needless to say there exists a proof that no other kinds of basic network types can exist.

The branch network is basically a network wherein the transmission in the network is altered by the action of the branch elements. The relay contact is the example par excellance, but there are other examples as well, such as cryotrons, and certain transistor configurations. They are typically bilateral devices, and can be connected in any fashion, and the only question of interest is the conflux of variables that results in the completion of a circuit between a pair of terminals.

The gate network is quite a different beast. It is essentially a graph made up of interconnected nodes wherein each node monitors several "inputs") and provides at its "output" (and there may be several replicated versions of this output) a signal that represents the truth value of the particular combination of values appearing on the inputs. With one special restriction, gate networks may also be interconnected in any way, and the function is realized on a specified one of the possibly available "output" lines. The special restriction is that we agree to never connect together the outputs of two different gates, for this results in lack of determinism in the mathematics, and possible melted terminals in the physical embodiment.

Thus Figure 3.1.1

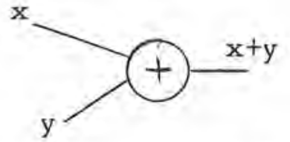


Figure 3.1.1. An OR gate.

is an OR gate that forms the output x+y, and Figure 3.1.2

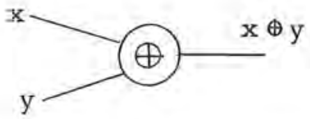


Figure 3.1.2. An exclusive-OR gate.

is an exclusive-or gate that forms the output x⊕y on its output, but Figure 3.1.3

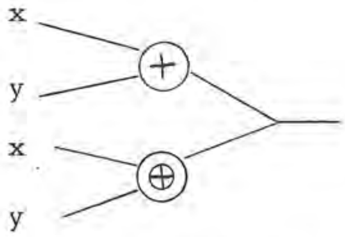


Figure 3.1.3. A prohibited connection.

is a network that we agree to never build since not only is the output not deterministic for the input combination "1, 1", but the physical realization is unstable as well.

The branch network has unquestionably seen a long decline in interest, simply because most networks nowadays are of the other variety. Probably, however, with respect to cleverness and intuition, and sheer perseverance, the branch network has provided the most challenge to clever network theorists and tinkerers. Very sophisticated analysts, from graph theory specialists to matrix theory sophisticates, have dabbled with the branch networks, with very few general results regarding minimality resulting from all these efforts.

The usual criterion in the branch networks translates into the number of variable occurrences in the switching expression representing the function. Thus, in terms of relay contacts, the expression

$$f = x'yz + x'y'z'$$

translates directly into the network of Figure 3.1.4

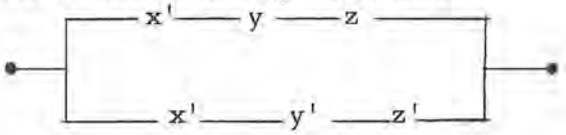


Figure 3.1.4. Typical branch network

with a cost in simple contacts of 6. Clearly the algebraic factorization of f into

$$f = x'(xz + y'z')$$

corresponds to the network of Figure 3.1.5

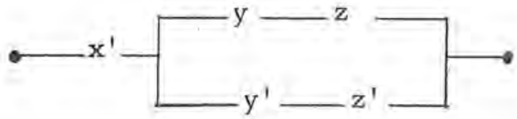


Figure 3.1.5. Simplification of branch network.

which has a cost of only 5, hence is cheaper by this criterion. Thus any algebraic form that has a lesser number of variable occurrences in the expression is immediately a cheaper network. The entire game then becomes one of clever algebraic manipulation in order to produce forms of least complexity, although we may be willing to use many levels of parentheses in order to do so. Thus the network of Figure 3.1.6 corresponding to $x(yz + y'(w+z')) + x'w(z+y')$

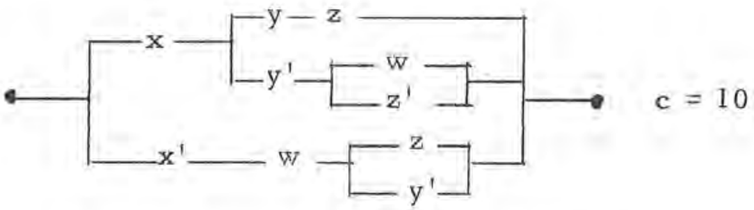


Figure 3.1.6. Branch network example

is certainly cheaper than the network corresponding to an expansion on the minterms in a direct parallel realization.

The whole house of cards comes tumbling down, however, when it is realized that branch networks that correspond to expressions in our algebra are necessarily limited to be series-parallel networks. For that, after all, was the fundamental physical correspondence made with our binary operators: "+" was to correspond with things in "parallel", while "·" was to correspond with things in "series". Every expression in our algebra must then correspond with what we might call an essentially series network, or with an essentially parallel network. For example,

$$f = xyz' + x'(w'y + z(y'z'))$$

is essentially parallel as indicated in Figure 3.1.7

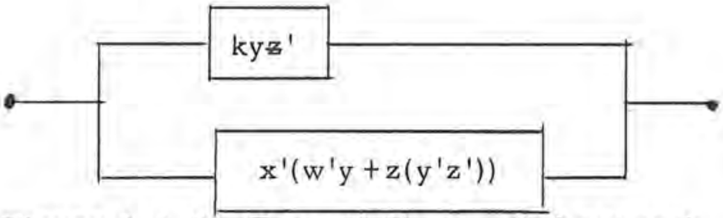


Figure 3.1.7. Essentially parallel network

while

$$f = (x' + zw)\,(xy' + z)$$

is essentially series

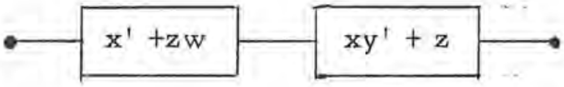


Figure 3.1.8. Essentially series network.

as shown by the decomposition in Figure 3.1.8. Of course each subnetwork in each case is in turn either essentially series or essentially parallel, and we can continue to break each of these down in turn. The result is a series-parallel network of contacts that corresponds one-for-one in contacts with the number of variable occurrences in the algebraic expression. The hooker in the whole argument is that many networks can be constructed that are simply not series-parallel networks, hence are simply not describable (in terms of associating contacts directly with variable occurrences) in our algebraic expressions. Probably the simplest example of this kind of network is that of Figure 3.1.9

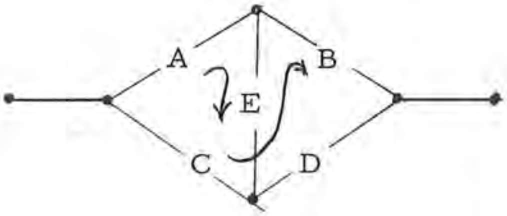


Figure 3.1.9. Example bridge network.

which uses the familiar bridge form, but is simply not directly the realization of any series-parallel algebraic form. This property depends crucially, of course, upon the bilateral nature of the elements. It matters not to the contact that ground passes in one direction through contact E for one path, and the other way for another path.

This observation would not be so unsettling if it were possible to show that there always exists a series-parallel form that is as "cheap" as any non-series-parallel form, but alas, such is not the case. The network given above is also an example of such an assertion, and the interested reader can try to beat the above cost of 5 contacts in any series-parallel network, and in doing so he will quickly become a believer.

Even more unsettling, it can be proved that most all networks can be minimally realized only in networks that are not series-parallel.

Worse yet, all the networks we have given so far have at least been planar, that is they can be drawn without crossovers on a sheet of paper. Certainly the series-parallel networks are most eminently planar, as is the bridge network given above. But some functions cannot be minimally realized unless we consider networks that are nonplanar. An example is the parity (even) function on four variables shown in Figure 3.1.10.

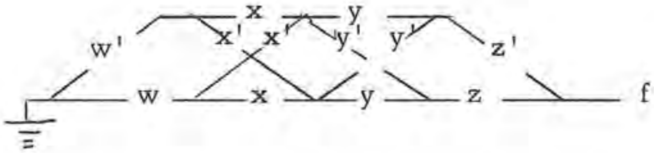


Figure 3.1.10. A nonplanar network.

This network cannot be drawn without the crossovers shown, and it can be shown that no planar network can achieve the same number of only 12 contacts. Moreover it can further be shown that most networks are not only non-series-parallel in their minimal realization, but they are also nonplanar.

This would appear to be a most unhappy state of affairs, and many attempts at different realizations have been investigated in an attempt to systematize these networks and their minimization. The interested reader might look into the literature under <u>tie sets</u> and <u>cut sets</u> for attempts at providing analysis (and by inversion, synthesis) techniques that will handle this more general kind of network.

The honest reporter must state that all of these attempts have been in vain, except for the most simple of networks involving a very few variables. There are obvious connections with graph theory and matrix theory in describing the connectivity of contact networks. These too, except intellectually, have provided no new practical insights except for the case of a very few variables. Even the Sunday afternoon game players have had a great deal to do with the minimization of these networks, and the results have been displayed in tables of demonstrably minimal contact networks for all functions through four variables that are available in several reference works. But the point is that there are no such tables for five variables, much less six, or seven. It seems strange that such complexity accrues so rapidly for functions of such a limited number of variables.

As a practical consequence of these observations, the cost criterion that simply counts the number of simple contacts, while still a reasonable one, is inadequate in most instances taken by itself. Rather for networks of reasonable size at all the usual ploy is to look for some regularity of structure within which the network is embedded, and then to eliminate as many contacts as possible by ad hoc methods.

A simple example of such a regular network is the minterm network of Figure 3.1.11 (in three variables, which is obviously generalizable to an arbitrary number of variables).

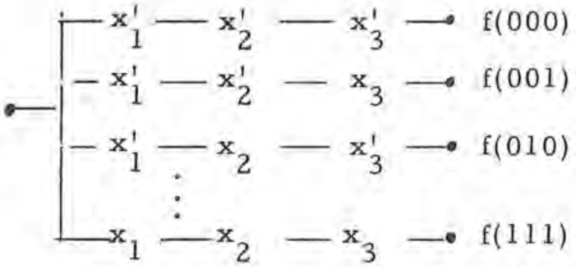

x_1' — x_2' — x_3' — f(000)
x_1' — x_2' — x_3 — f(001)
x_1' — x_2 — x_3' — f(010)
x_1 — x_2 — x_3 — f(111)

Figure 3.1.11. Minterm branch network

We simply design a set of parallel lines, each one corresponding to a minterm, and to realize a particular function we connect together the appropriate minterm set that comprises f, and our realization of the arbitrary f is complete. Of course we can then eliminate the unused lines, reducing the total number of contacts required for the particular f. If we let C(n) be the maximum number of simple contacts required for the realization of an arbitrary f it follows that

$$C(n) \leqslant n2^{n}$$

and at least we have bounded the potential complexity of our problem.

Simple as it is, this sort of approach is completely characteristic of many of the systematic techniques by which the design of efficient contact networks can be approached. One always has a design algorithm at hand by which to proceed, usually the regularity of the structure yields an upper bound on the possible complexity of the realizations, and always some contacts can be eliminated in the final realization. A direct extension from the basic network above are the tree networks that depend upon successive factorization of the variables one at a time. For example the three-variable tree looks like that in Figure 3.1.12

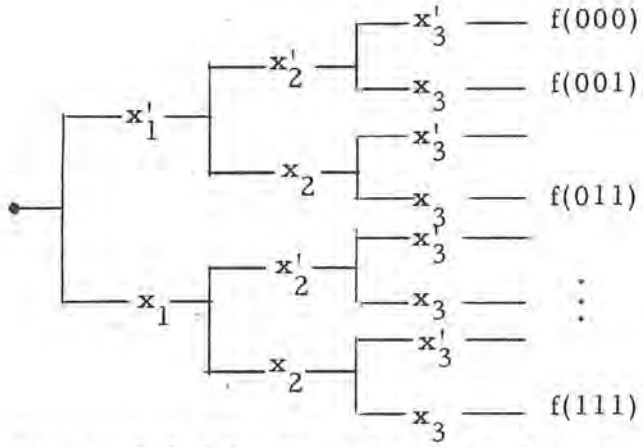


Figure 3.1.12. Tree branch network

and such tree networks have provided starting points for a great many systematic techniques. Of course the number of contacts is readily bounded for the n-variable tree by counting the number of contacts in each level starting from the single node at the left (root) and proceeding to the minterm nodes at the right (leaves):

$$C(n) \leq 2 + 4 + 8 + \ldots + 2^{n} = 2(1 + 2 + \ldots + 2^{n-1}) = 2(2^{n}-1)$$

which is a considerable better complexity result than for the minterm network itself. Any particular function is easily realized, e.g., $f = \Sigma(2, 3, 5, 6, 7)$ looks like the network of Figure 3.1.13.

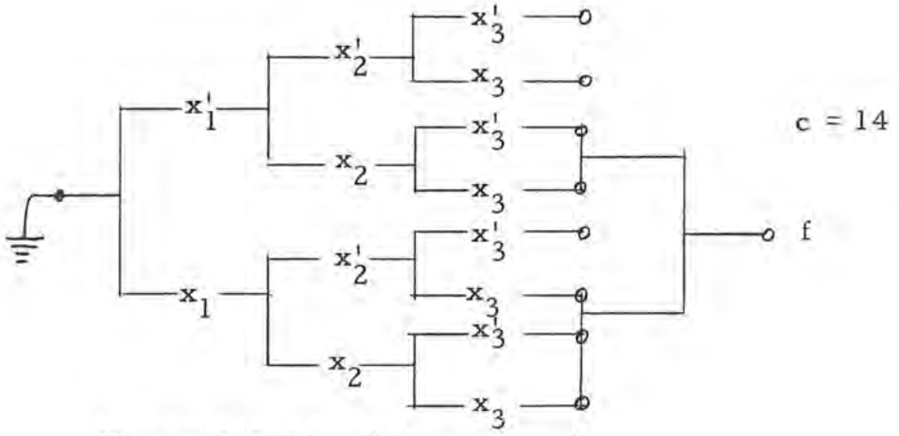


Figure 3.1.13. A tree example

Of course we can trim away any unused contacts, as well as shorting any contacts whose output terminals are already shorted. For this network this yields the reduction shown in Figure 3.1.14.

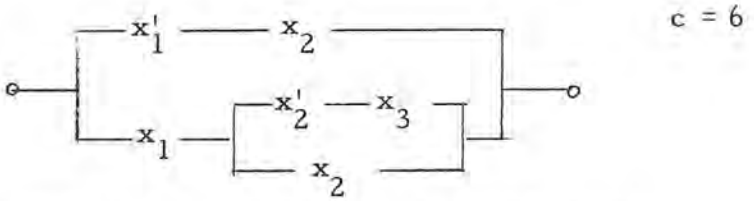


Figure 3.1.14. Partially reduced tree

Also for this network we can readily see an application of the theorem $x+x'y = x + y$ to eliminate one more of the x'_2 contacts,yielding the network of Figure 3.1.15.

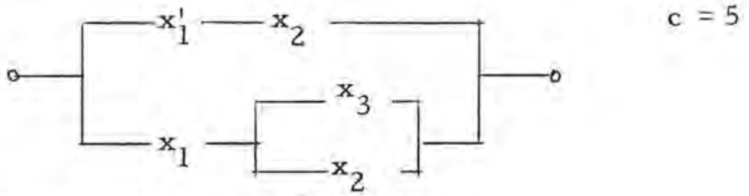


Figure 3.1.15. Reduced tree.

as our final result.

In the example above we could of course get the same results ultimately by algebraic manipulation since our ultimate realization turned out to be series-parallel. The tree realizations have greater power than this example would indicate, though, and can also lead to nonplanar realizations. We can illustrate this in the case of the parity network previously presented. The four-variable function for even parity can be verified to be $f = \Sigma(0, 3, 5, 6, 9, 10, 12, 15)$, hence is realized by the four-variable tree of Figure 3.1.16

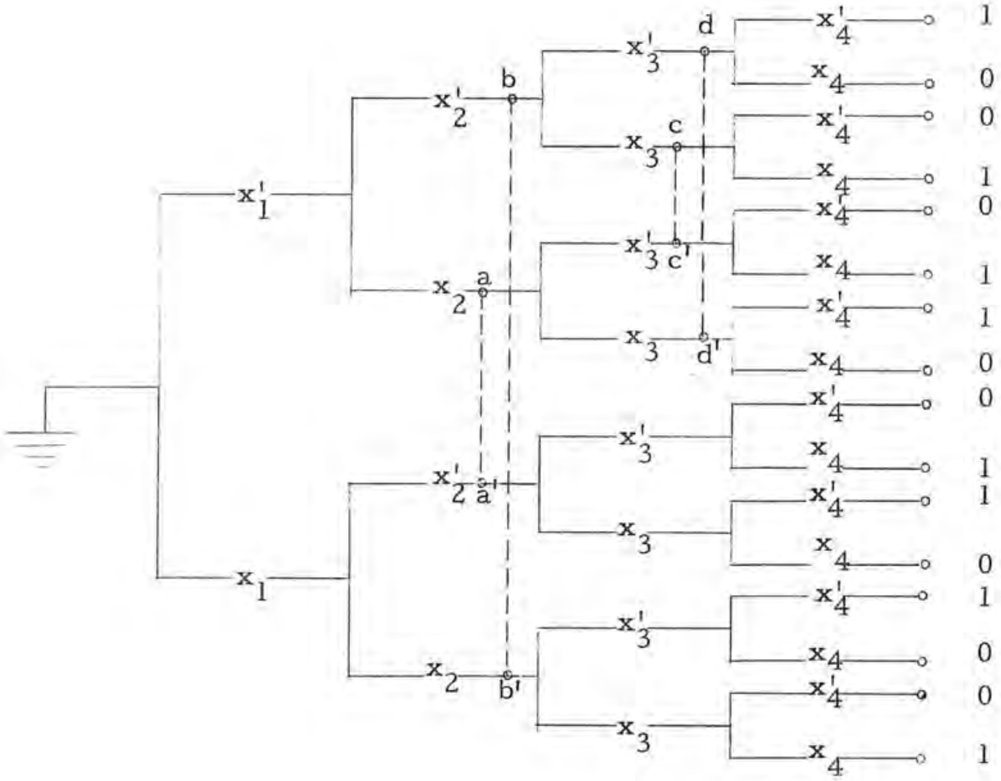


Figure 3.1.16. Tree realization of parity network.

This time we have used 1's on the outputs to show those terminals that would be joined together to produce the desired f. The 0's on the outputs indicate the terminals that would be left unconnected. Now we view the network from the two marked points a-a' say. Since the subtrees from these two points lead to the same selection of outputs, it follows that either one of the subtrees can be eliminated if we but shortcircuit along the dotted line indicated between the points a and a'. Similarly we observethe same possibilities for points b-b', c-c' and d-d'. Taking advantage of this observation, and it's at exactly this point that the nonplanarity arises, we get as a simplified version of the tree the following network of Figure 3.1.17.

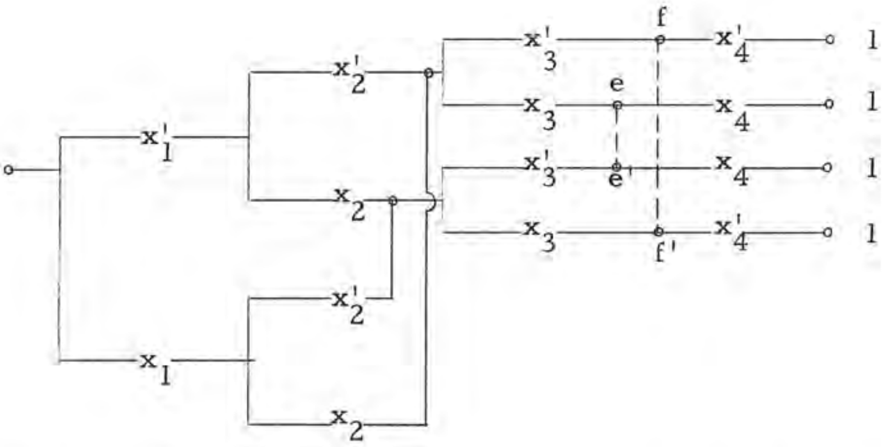


Figure 3.1.17. Partially reduced parity network.

In this form we again note the same possibility of shorting the points e-e' and f-f'. Doing this we achieve, after rearranging the layout, the network we observed before and reproduced in Figure 3.1.18.

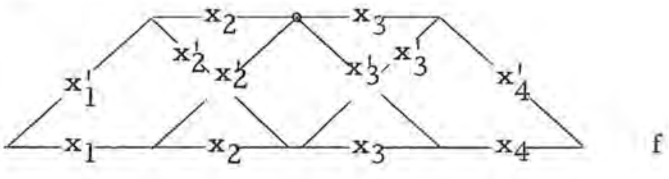


Figure 3.1.18. Reduced parity network.

This process is generalizable, and the arbitrary parity function network on n variables can be realized using only 4n-4 contacts.

We could explore all sorts of other directions at this point for the branch networks. For example, the complete trees, since they can be used to realize any f,are a member of a class of networks called all-function networks. There are many other forms of all-function networks, and some of these achieve better complexity bounds than the complete trees. We shall save these for a later discourse.

Another cost factor that is discussed widely in the literature takes the approach that the simple contact is not the important thing to count, but rather the transfer contact, which is a combination of a make and break contact realized in one package. The transfer contact requires only three "springs" in its construction, versus four springs in the separate construction of the two kinds of simple contacts. Therefore this approach attempts to minimize springs rather than contacts, and yields different minimal networks in many cases. Another approach seeks to "balance out" the appearances of the different variables in the realization. The trees above obviously use many occurrences on some variables (near the leaves) and only a few on others (near the root). In order to make the loads on different relays more nearly equal, there is another large body of techniques that seeks to distribute the variables differently. There is no reason that each subtree, as we proceed from the root, need be expanded on the variables in the same order.

These and other similar questions are noted, and we shall do nothing more with them at this point.

3.2 Gate Networks

In the gate networks we have considerably more design flexibility than in the branch networks and it is even more needful to properly define our area of discourse in order to be able to say anything meaningful at all. For, after all, a contact can only do one thing: it can either provide a short circuit between two terminals or an open circuit. A general n variable gate, however, can be whatever we define it to be; in particular it can be whichever of the 2^{2^n} functions of n variables that we choose it to be. Of course it is more difficult to build an n variable gate, than an n-1 variable gate. But it's important to recognize at the outset that even more closely than for the branch networks our design problems are restricted only by what we can afford to pay.

Thus, the answer to the question: what is the maximum number of gates needed to realize an arbitrary function of n variables? is exactly one. That is $C(n) = 1$ for all n if we use as our cost criterion the number of gates required. Surely there must be more to logic design than this! And of course there is, for our answer above, while correct, is very misleading, for it would require us to build arbitrarily complex gates of very high cost, hence reveals nothing more than that in this context the number of gates is not a realistic cost measure. Actually we would be better advised to peer within our complex n-variable gate and determine how many littler "gates" we would actually require in order to realize it.

For after all, what is a gate? It is simply an assembly as in Figure 3.2.1 with n inputs

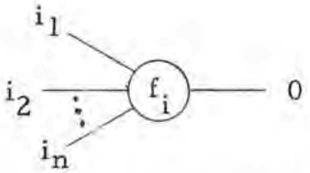


Figure 3.2.1. General n-variable gate

and a single output. The n inputs are binary-valued (corresponding to some assignment of the actual physical values such as "high voltage" = 1, and "low voltage" = 0) such that the output which is also binary-valued is determined as some specified combinational function of the inputs. We would mark the gate somehow, as with f_i above to indicate just what the function is, and then we could proceed to construct deterministic networks from such gates. It is evident that such gates correspond directly to our propositional connectives on the one hand, and to the operators in the extended algebra on the other. Gates correspond directly with what we earlier called "connective blocks" and networks of gates correspond with the propositional networks designed at that time.

It is evident, then, that without further cost restrictions, <u>every</u> expression in our extended algebra correspond structurally to a gate network. For example the expression

$$f(w, x, y, z) = (xy+z)\supset[(w\oplus z)\quad x']$$

is realized directly by the gate network of Figure 3. 2. 2

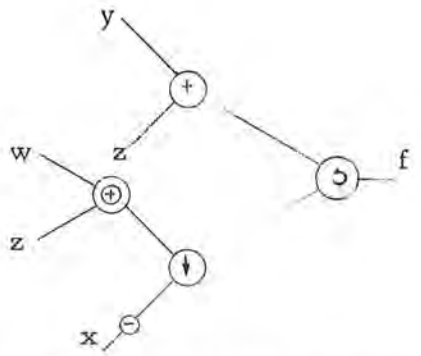


Figure 3. 2. 2. Example gate network.

and it would again appear that we've said everything that needs to be said. The cost criterion would reasonably be the total number of gates involved, and simplification of a network would correspond to algebraic manipulations that reduce the number of operator appearances, i. e., connectives, in the algebraic form. Of course if we want to be more reasonable we would begin to limit our options by at least requiring a "fanin" limitation on the gates we can use, i. e., on the number of input leads, and probably on the "fanout" as well, i. e., on the number of gates we would allow to be driven by a given gate. These assumptions would be reflected from our physical construction difficulties, however, and have nothing to do with weaknesses in our "logic".

Although as logicians we are willing to assume that things happen without the passage of time, as circuit designers we are not. The more gates that a given signal need pass through in contributing to the output, the greater is the passage of time necessary for that propagation to take place. Hence we might also be concerned about the number of levels of which a network is composed, and might require, for example, that no network be "deeper" than three levels in order to speed up the computational operation.

Although as logicians we might also be willing to manipulate almost any kind of operator, as algebraists such will probably be trying on our patience since we would need a great many manipulative theorems, and as circuit designers again we would probably rather not. Certain of the connectives can be realized more easily in certain technologies, and it is always an engineering truism that you can make a great many things out of a small set of possibilities than out of a large set such that each item is more accurate and reliable. For whatever the reason, the logic design context is always phrased in terms of a relatively small allowable set of operators or gate types. For all these reasons, the logic design in a particular case is always severely restricted, and each restriction yields different algorithms for design. We shall consider only a few as examples.

3.2.1 Two-Level AND/OR Networks

The two-level AND/OR network (or its NAND, NOR variants) is usually what people mean when they say "logic design". The development of such networks is a classic example of how algorithms can be developed, and how different functional representations can be brought to bear on a problem. Beyond that, the ultimate problem of producing demonstrably minimal circuits is still an unsolved problem (and is likely to remain so) hence reiterates a theme that we've heard before in these discourses.

Suggested by the basic Shannon decomposition theorem on minterms, the two-level AND/OR is assumed constructed of AND and OR gates only. We are willing to consider AND gates of an arbitrary number of inputs, as well as for the OR gates, although we must admit when pressed that this is not really practically obtainable. When pressed on this point we shall dodge the issue by pointing out that we can always realize a gate on an arbitrary number of variables by cascading those of a lesser fixed number of inputs.

Now the first stumbling block is that we remind ourselves that the set

S = (+, ·) is simply not logically complete. Of course we want to realize arbitrary functions nonetheless, and the additional provision that we shall use to attain these ends is to assume that we have both the unprimed and primed variables available as inputs to the network. Aesthetically this looks a little like a self-serving assumption, but practically it turns out to be not unreasonable at all. Quite frequently the generation of a given function f is accompanied by generation of f' as well and both of these will usually be circulated wherever necessary as input variables for other networks. At any rate, that is the nature of our assumption, and with the primed variables also available it is clear by reason of the Shannon expansion theorems that AND's and OR's are complete for any functional realization.

Now we make another assumption that can primarily be justified only in terms of overall network speed: we shall assume that only two levels of logic are to be allowed. Two levels are clearly sufficient, again based on reference to the Shannon minterm form, but the principle reason for desiring only two levels is because of the speed of network response. Clearly we cannot realize all functions with only a single level of AND's or OR's, so the two-level networks achieve the maximum speed possible within our logic assumptions. After having made the assumption, however, we can observe that an additional advantage to the two-level form is that it is regular in form, and each network looks like every other in its basic characteristics.

The basic structure, then, of the two-level AND/OR is to assume a level of AND's feeding a collecting OR gate that forms the output. Thus the basic network looks like that in Figure 3.2.1.1.

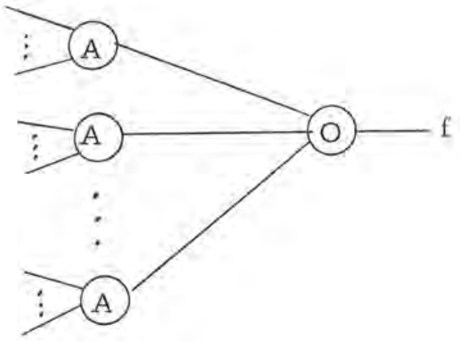


Figure 3.2.1.1 Basic AND/OR form.

As a first pass at a reasonable cost criterion we can take a direct count of the number of gates involved. Since the Shannon form indicates that we always have a minterm expansion of any function, and since there are never more than 2^n of these, we can surely bound the cost of such network by

$$C(n) \leqslant 2^n + 1$$

For example, the direct realization of $f = \Sigma(1, 3, 12, 13)$ by means of its minterms is given in Figure 3.2.1.2

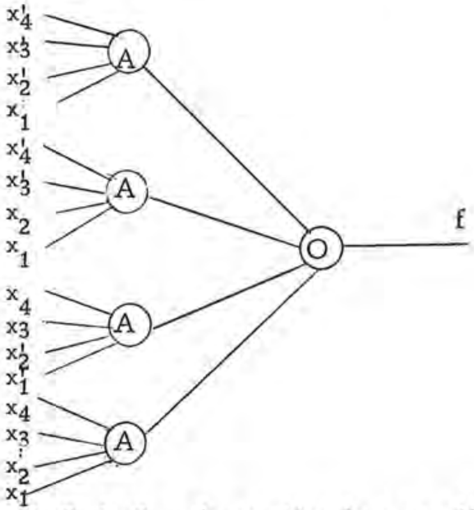


Figure 3.2.1.2. A particular realization.

at a cost of 5 gates, and it should be clear that any arbitrary function can be so realized by presenting each of its minterms on one of the input AND gates.

We can even observe the structure of another all-function network by this approach. Suppose we associate a minterm with each input AND and provide it with an extra input for the binary constant that goes with that minterm. Thus the general structure of such an all-function network is as in Figure 3.2.1.3

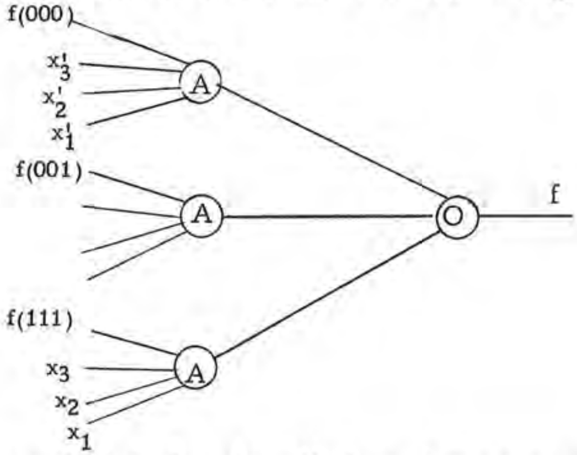


Figure 3.2.1.3. An all-function network.

and depending on how the f(ijk) inputs are specialized, it is obvious that any of the n-variable functions can be realized.

Our present purpose, however, is to develop an algorithm for the minimal realization of an arbitrary, but specified function within such a form. From the structure of the form it is clear that the only algebraic form that can correspond is a simple sum of a set of simple products (where by simple we mean with no parentheses--thus $(x+y)z$ is not a simple product, although $xy+xz$ is the simple sum of a couple of simple products).

Of course the Shannon minterm decomposition is an example of such a representation, and we conclude that the AND/OR network corresponding to the minterm expansion always exists for every function. If m is the number of minterms in f, then surely

$$C(n) \leqslant m+1$$

for any function. Of course we might hope to do better. In general the minterms of a function can be algebraically combined in various ways to yield sums of a lesser number of products, hence a realization that requires fewer gates. For example, consider the function $f = \Sigma\,(0, 2, 3, 4, 5, 7)$ which requires 7 gates in the direct AND/OR realization. Algebraic manipulation yields as alternative representations for f the expressions

$$f = x'z' + y'z' + yz + xz$$

$$f = x'z' + xy' + yz$$

$$f = x'y + y'z' + xz$$

which require respectively 5, 4, and 4 gates in the two-level AND/OR realization. They are sum-of-products forms, but are clearly not composed of minterms. Any sum-of-products which is not a fundamental (or minterm) sum-of-products will be called a reduced sum. A reduced sum which exhibits

1) the least possible number of product terms, and
2) within this restriction, the least number of variable occurrences in the expression

will be called a minimal sum for f, and we shall abbreviate it as f_{ms}. The first restriction above clearly corresponds to the least total number of gates required in any realization, while the second reduces the number of gate inputs to the least possible number within the first restriction. We shall accept for this discourse the cost function implied by the two properties listed above, hence the cheapest two-level AND/OR network is one corresponding to f_{ms}, and our complete attention in deriving a relevant algorithm will be in developing f_{ms} for any arbitrary function.

Now a first approach at deriving f_{ms} might be algebraic. The three forms given for our example f above are all algebraically derived (the interested reader might be interested in verifying the assertion) by starting with the minterm form for f and simplifying. A disturbing fact is that the results of such algebraic simplification display "local minima" because of the following fact. Given any reduced sum-of-products we shall say that that form is irreducible of the deletion of any literal results in a function that is no longer f. All of the three forms given above are irreducible. From the forms of the example, while we can readily conclude that any f_{ms} must certainly be irreducible, it does not follow that any irreducible f is an f_{ms}. On the other hand, it can be shown that the last two forms do meet both of our cost criteria, hence we can conclude that f_{ms} is not a unique thing in general, that is that there may be several reduced sums that are minimal. All of this promotes a sort of feeling of elusiveness about the whole discourse that is unsettling without further narrowing down of our options, and further pointing to the particular representations that might be of help in our search for appropriate f_{ms}.

The key characteristic that turns out to be of help is the following: the only algebraic manipulation that is of use in moving from the minterm form to any of the reduced sums (without introducing any parentheses) is of the form

$$f = Ax + Ax' = A .$$

But this is just another way of saying that two minterms can be combined as above, if and only if they differ in a single variable, or in terms of the n-cube, if and only if they lie on adjacent nodes. This suggests that our n-cube representations might be of utility in deriving such simplifications, and such turns out to be the case, for the n-cube representation, particularly the map forms known as the Karnaugh maps, provide efficient means for getting at the problem directly.

Thus the usefulness of the Karnaugh map, at least in this context, derives from the fact that adjacencies on the map show where simplifications of the desired type can be made. For example the map of Figure 3. 2. 1. 4

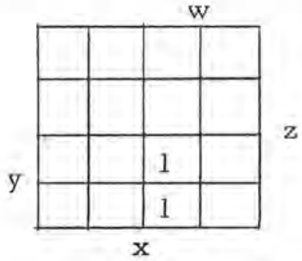


Figure 3, 2, 1. 4. Map of a particular f

indicates two adjacent nodes are 1 in f and that immediately we can combine them as in

$$wxyz + wxyz' = wxy,$$

and in fact the final product itself can be read directly by identifying that part of the map described by the two adjacent 1's: wxy.

At this point, the interested reader should verify for himself that any simple product of the variables must correspond to a subcube on the general n-cube, hence that the various simple products of which our minimal sums must be composed comprise recognizable patterns of 1's on the marked Karnaugh map, patterns that correspond to the various differing dimension subcubes of the map. For example:

wyxz	0-cube	an isolated cell of the map	1 minterm
wxy	1-cube	two adjacent 0-cubes	2 minterms
wx	2-cube	two adjacent 1-cubes	4 minterms
w	3-cube	two adjacent 2-cubes	8 minterms
1	4-cube	two adjacent 3-cubes	16 minterms

and so on. Although these comments are made in terms of the 4-cube, it should be obvious that they generalize for general n-cubes.

Furthermore, these various kinds of cubes can all be readily identified (visually) on the map (again the 1-cube provides a convenient example -- and remember the end-around adjacencies) as being of the kinds shown in Figure 3. 2. 1. 5.

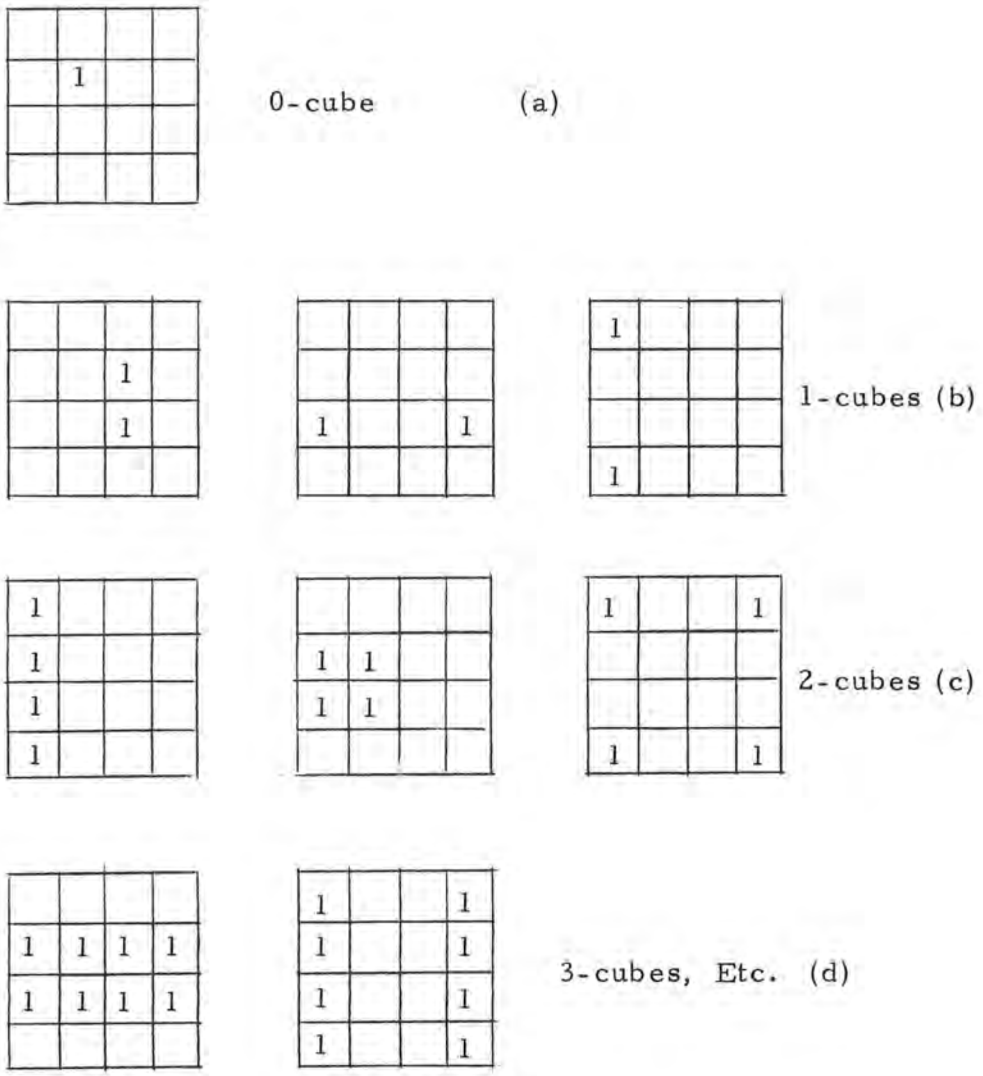


Figure 3.2.1.5. Various cubes on four-variable map.

i.e., the various simple products of which our f_{ms} might be composed correspond to the various subcubes indicated above, that is to the various sort of generalized squares that might appear as "patterns" in the n-cube.

Clearly, since these subcubes represent products in potential minimal sum-of-products forms, they play an important role in the representation of arbitrary f in terms of such products.

To this end we introduce a few definitions that you will find are widely spread in the literature describing these affairs. We shall say that g is an <u>implicant</u> of f if whenever g is true, it follows that f is also. In the present context, if g is a product term that implies that f = 1 whenever g = 1, then g is an implicant of f. Thus if

$$f = yz + xz$$

then surely both yz and xz are implicants of f.

Furthermore, we shall say that g is a prime implicant of f if

1) g is an implicant of f, and
2) the deletion of any variable from g results in a product term that is not an implicant of f.

Thus in our previous example, z is certainly not an implicant of f, nor is y, hence it follows that yz is a prime implicant of f. On the other hand,

$$f = xyz + x'yz + xy'z$$

is the minterm expansion for f, and of course all of these minterms are also implicants of f. On the other hand, x'yz is not a prime implicant since yz is also an implicant of f, and it can be gotten simply by deleting the variable x' from x'yz.

To this point we can make some conclusions regarding our search for f_{ms}. In the first place it must obviously be irreducible algebraically, although we've already verified by an earlier example that this is not sufficient. Furthermore we can now conclude that

$$f_{ms} = \Sigma \text{ (a set of prime implicants}$$
$$= \Sigma \text{ (a set of maximal cubes that cover f)}$$

where we say that f_{ms} covers f if it is 1 whenever f is. Thus our problem is reduced to the graphical one of determining a covering set of cubes for f, where each cube is as large as possible, and the total number of cubes is minimized (since each corresponds to a product term in the sum-of-products expansion).

A first step before continuing the argument is to convince ourselves that each function is associated with a unique set of prime implicants. Every minterm marked 1 on the map determines at least one maximal cube that contains it. For example in the map of Figure 3. 2. 1. 6

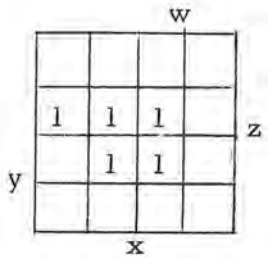


Figure 3. 2. 1. 6. Map of example f.

minterm 1 is contained in prime implicant w'y'~~z~~', while minterm 13 is contained in prime implicant xz. On the other hand minterm 5 is contained in both w'y'z and in xz. Clearly we can generate the total list of prime implicants by successively going through each minterm and listing the prime implicants which contain it. For the example above our list would consist of only the two entries: xz, and w'y'z.

A function that uses all the prime implicants of f is called the complete sum for f, which we'll designate f_{cs}. A reasonable conjecture at this time, then, would be to ask whether

$$f_{ms} = f_{cs}$$

as is certainly the case for our example above. To dispose of this conjecture let us consider the map for the function considered earlier, namely f = Σ (0, 2, 3, 4, 5, 7) as shown in Figure 3. 2. 1. 7.

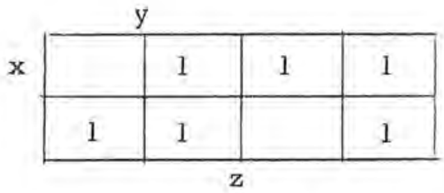


Figure 3. 2. 1. 7. Map of f.

If we determine all the prime implicants for this f from the map we will generate the list:

$$f_{cs} = x'y + yz + xz + xy' + y'z' + x'z'$$

which is irreducible, and even worsethan we had before for this function. So clearly it is not the case that $f_{ms} = f_{cs}$ and we must look further.

Rather the minimal sum consists of the minimal covering with such cubes, and in the case of our example function either of the two coverings shown in Figure 3. 2. 1. 8

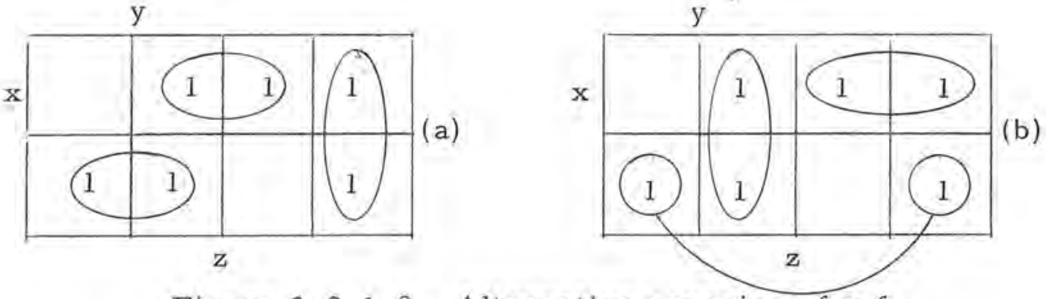


Figure 3. 2. 1. 8. Alternative coverings for f.

will do, corresponding to

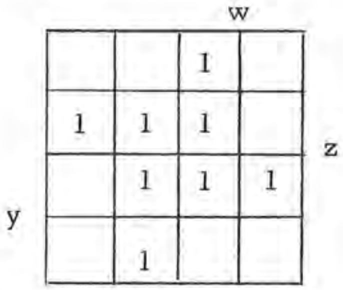
$$f_{ms} = x'y + xz + y'z' \text{ and } f_{ms} = yz + xy' + x'z'$$

as we pointed out before. This points out another disturbing fact: f_{ms} is not even unique, and for functions of considerable complexity it is possible that a very large number of minimal forms can exist.

The above example also points to a counterexample for another conjecture we might be tempted to make. That is, there are certain prime implicants about which we obviously have no choice about including or not. These are the so-called essential prime implicants. If in a prime implicant there exists at least one vertex such that no other prime implicant contains that vertex, then that prime implicant is called essential. Clearly if f_{ms} is to cover that particular vertex, then it is necessary that that particular prime implicant be included in the sum. In the earlier $f_{ms} = xz + w'y'z$ that we used as an example, it is evident that both prime implicants are essential: the first is substantiated by vertex 1, while the second is required by the presence of vertex 13. A reasonable conjecture, then, would be to ask whether it is the case that

$$f_{ms} = \Sigma \text{ (all essential prime implicants)}$$

and a relevant data point to support this conjecture would appear, for example, to be functions of the form shown in Figure 3. 2. 1. 9



		1	
1	1	1	
	1	1	1
	1		

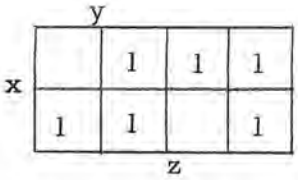
Figure 3.2.1.9. An essential prime implicant example.

where we can quickly see that only the essential prime implicants need be included in the expansion for f_{ms}. (Readers who are seeing these arguments for the first time will be strongly tempted to include the prime implicant xz in this expansion; in such cases the implications should be considered carefully, and then avoided from now on in the future.) Thus

$$f = wxy' + wyz + w'xy + w'y'z$$

is the minimal sum in this case; it includes only the essential prime implicants, and the reason that xz is not included is that all of its nodes are already covered by those which are essential.

But on the other hand, the three variable example we considered earlier and reproduced in Figure 3.2.1.10



	1	1	1
1	1		1

Figure 3.2.1.10. An essential prime implicant counter example.

simply has no essential prime implicants whatsoever. (The reader is invited to point to one.) Hence the conclusion, if our conjecture is correct, is that f = 0 in this case, which is of course patent nonsense. Hence we must conclude that the conjecture is wrong, and that in general

$$f = \Sigma\,(\text{all essential prime implicants})$$
$$+ \Sigma\,(\text{some nonessential prime implicants}).$$

Hence the covering problem is reduced to selecting some minimal subset of the nonessential prime implicants that is necessary to complete the covering job. It is relevant to point out at this point that even for this simple AND/OR structure, we have exhausted the determinism that exists anywhere in the literature on this subject. The selection of the minimal set of nonessential prime implicants is an unsolved problem, in general, and in fact there are conjectures that it lies within the class of "unsolvable"[1] problems. The best we can do,then, is to exhibit means by which some solutions can be found. Of course in the case of just a few variables it will be relatively easy to exhibit all possible solutions. But the ultimate selection of just which solution to pick will be left up to the discerning logic designer.

As an example, then, we might consider the function

$$f = \Sigma(1, 2, 4, 5, 6, 11, 12, 13, 14, 15)$$

to be realized in a minimal two-level AND/OR network. Our program calls for developing the prime implications of f, and from these to select some minimal subset on which to base the realization. To this end we map f as in Figure 3. 2. 1. 11

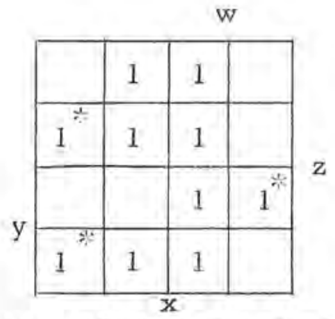


Figure 3. 2. 1. 11. Minimization of example f.

and conclude that the prime implicants are

essential	A	w'y'z	covering minterms	(1, 5)
	B	w'yz'		(2, 6)
	C	wyz		(11, 15)
nonessential	D	xy'		(4, 5, 12, 13)
	E	wx		(12, 13, 14, 15)
	F	xz'		(4, 6, 12, 14)

Of these, we have noted that A, B, and C are all essential. (These can be justified by the starred nodes on the map; each is uniquely associated with the given prime implicant, hence must be included in any expansion.)

Thus at this point we conclude that

$$f_{cs} = A + B + C + D + E + Y$$

and the remaining portion of our problem is concerned with which subset of the nonessential prime implicants we need as well.

To this end we shall propose a systematic method for displaying our choices that is known as the prime implicant table. In this table we shall initially carry along the essential prime implicants as well, not that they are needed for this particular example, since they (and their covered minterm sets) are already known. But most machine related techniques (such as the Quine-McCluskey tabular equivalent of our map methods to be considered later) generate all of the prime implicants, i. e., f_{cs}, and it still remains to distinguish those that are essential.

The prime implicant table is a way of displaying the coverages of all the prime implicants, and of selecting which are required to do the job. It shows the minterms to be covered as row markers, and the prime implicants as column markers:

	A	B	C	D	E	F
1	x					
2		x				
4				x		x
5	x			x		
6		x				x
11			x			
12				x	x	x
13				x	x	
14					x	x
15			x		x	

Now in this display we can readily determine which are the essential prime implicants. They are the ones with only a single entry in a row. I. e., A is essential since it is the only one covering minterm 1. Using this test we conclude that A, B, C are all essential. Hence columns A, B, C and rows 1, 2, 11 can be crossed from the table. We also note, however, that each of A, B, C cover other minterms as well. For example A covers minterm 5, and we need not consider that row any longer. Similarly we need not further consider rows 6 or 15 for the same reason, and these have been marked out as well in the reduced prime implicant table leaving

	D	E	F
4	x		x
12	x	x	x
13	x	x	
14		x	x

which describes succinctly the balance of the covering problem for this function. We can finish our problem by inspection in this case. Clearly none of the single prime implicants will cover all four minterms, certainly any two will, hence we certainly don't need all three. Again let's systematize our process, however, so that we can handle problems of greater complexity. Conveniently enough, the balance of the problem can be easily phrased in terms of a propositional statement regarding the necessary columns that must be included. For example if minterm 4 is to be covered, it is plain that we must have either column D or column F in the expansion. But for minterm 12, we must have D or E or F as well, while for minterm 13 we must have D or E, and so on. Clearly this propositional statement can be represented by the expression

$$P = (D+F)\ (D+E+F)\ (D+E)\ (E+F)$$

and we will accomplish our covering if this expression has truth value 1.

Now the expression as stated as a product of sums is not in a convenient form, for what we want is a listing of which prime implications will suffice. I.e., we want a list tantamount to the proposition "P is true if D and F is", such as would be given by a sum of products representation of P. By now this should be old hat; it simply requires us to expand P to a sum-of-products form by "multiplying out", and every product term in the result will be a sufficient covering set of prime implicants. Thus

$$\begin{aligned} P &= (D+F)\,(D+E+F)\,(D+E)\,(D+F) \\ &= (D+F)\,(D+E)\,(E+F) \\ &= (D+EF)\,(E+F) = DE+EF+DF+EF \\ &= DE + DF + EF \end{aligned}$$

which affirms algebraically what we observed previously, that any pair of the prime implicants will do. This does not complete the whole story, although it would appear to do so. In this form we do not know the relative sizes of D, E, F (i. e., how many variables are in each), so we actually have a little more checking to do. But in this case they are all of the same size, hence the choice is immaterial.

To complete our example, then, our minimal form is not unique, there are three of them, and they are

$$f = A + B + C + \begin{cases} D + F \\ D + E \\ E + F \end{cases}$$

and any one of the three will yield a minimal circuit. For the first one we get the network of Figure 3. 2. 1. 12

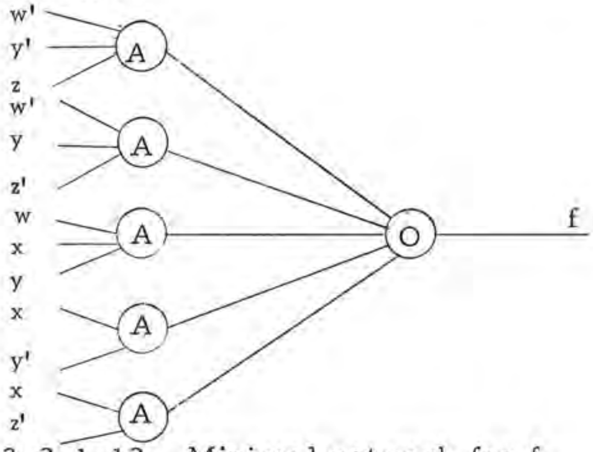


Figure 3, 2, 1. 12. Minimal network for f.

and we can guarantee its minimality.

3. 2. 2 Two-Level OR/AND Networks

Whenever considerable effort has been expended in solving a certain problem, it is always good sense to examine the relevance of the solution to other related problems, hence expanding the application range of the newly developed bag of tricks. Such should also be the case in logic design algorithms, and having developed a well defined procedure for two-level AND/OR networks we should hope to apply it almost directly to closely related networks. This we shall do in this and the succeeding sections.

Because of the duality we have constantly noted in the algebra, we must suspect that the algorithm could be almost directly applied to the minimization of two-level OR/AND networks. That such networks exist is evident from the Shannon maxterm expansion, and the steps taken to their minimization in terms of reducing the total number of ORs and of OR inputs should be identical. They are. Recall that the maxterms present in the expansion are related to the 0's of the function, rather than the 1's. Combinations of the sums that will result in efficient reduced products-of-sums will therefore, by duality, be detectable from a map minimization of the 0's of f.

Thus for our previous example we have

$$f = \Sigma(1, 2, 4, 5, 6, 11, 12, 13, 14, 15) = \pi(0, 3, 7, 8, 9, 10)$$

and we can represent these 0's on the map of Figure 3.2.2.1

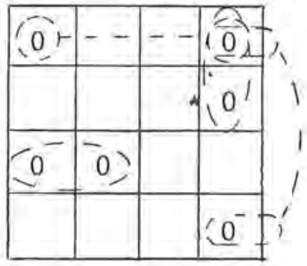


Figure 3.2.2.1. Complementary map for f.

and coalesce these into the duals of prime implicants as shown in the map above. In this instance they all turn out to be essential and are stated as duals as $wx'y'$, $wx'z'$, $x'y'z'$, and $w'yz$. The sums corresponding to these pseudo implicants are of course

$$f = (w'+x+y)\,(w'+x+z)\,(x+y+z)\,(w+y'+z') \quad .$$

(The reader is again warned that the bitwise inversion previously noted with respect to the decimal product form has also taken place here.) Now since all the implicants were essential the expansion is unique, thus the minimal product form, f_{mp} is unique even though f_{ms} was not.

The network that corresponds to this minimal form for f is in Figure 3. 2. 2. 2

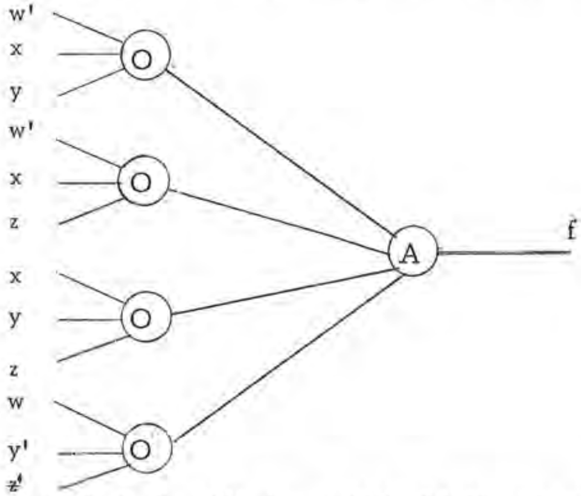


Figure 3. 2. 2. 2. Minimal OR/AND network.

and we note that it is not only different, but it is also cheaper than the two-level AND/OR form. Thus the judicious designer, if he has the design freedom to do so, will always examine both forms before accepting either.

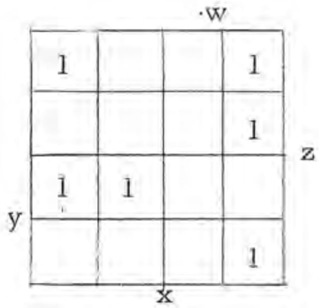
An alternative way (which had better yield the same result) is to look at the two-level OR/AND as the complement of a two-level AND/OR. That is by our generalized DeMorgan theorem, anything that complements all the variables, replaces ANDs with ORs, and ORs with ANDs of a network, must leave the network still minimal, <u>but for some other function</u>, namely the complement of the original function. By this reason our tactic should be to realize f' in a minimal AND/OR network (which is exactly our previous technique), then replace ANDs with ORs and conversely, and complement all the variables. The result must be a two-level minimal OR/AND network. That it must be minimal is evident; for if it were not there would be some simpler form. But then we could invert the transformation and imply a simpler realization for f'. But we already had a minimal realization for f', hence a contradiction.

Taking this tack on our example function, we have

$$f = \Sigma\,(1, 2, 4, 5, 6, 11, 12, 13, 14, 15)$$

$$f' = \Sigma\,(0, 3, 7, 8, 9, 10)$$

corresponding to the map of Figure 3. 2. 2. 3



1			1
			1
1	1		
			1

Figure 3.2.2.3. Map for f'.

and hence the $f'_{ms} = wx'y' + wx'z' + x'y'z$. Taking the complement of both sides we of course get the original network shown above, as we must. But the point of view is a little different, and is often easier to remember.

Thus the two-level OR/AND is completely solved using techniques for the AND/OR case.

3.2.3 Two-Level NAND Networks

A little surprising at first glance is that the two level NAND and NOR networks are also completely accommodated by the determination of f_{ms}. This takes a little argument, however, which we proceed to develop, and we shall choose to do so in terms of the kinds of structural equivalents we've observed before.

One of our elementary theorems was that $(x')' = x$, and interpreted structurally this means that into any lead we can insert a couple of NOT gates without changing the "terminal behavior" at all. Thus we have Figure 3.2.3.1

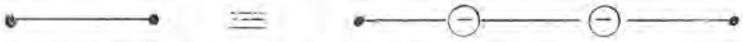


Figure 3.2.3.1. $(x')' = x$.

and the only significance is that this trivial observation can sometimes be usefully applied. For example, consider the basic AND/OR structure of Figure 3.2.3.2.

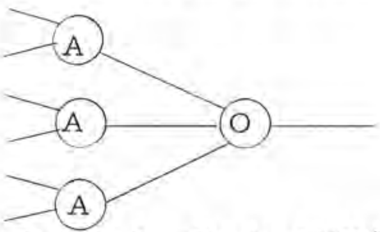


Figure 3.2.3.2. Basic AND/OR structure.

By our theorem we can insert a pair of NOTs into each lead between the ANDs and the OR without changing anything. Thus we have Figure 3. 2. 3. 3 and can assert that it

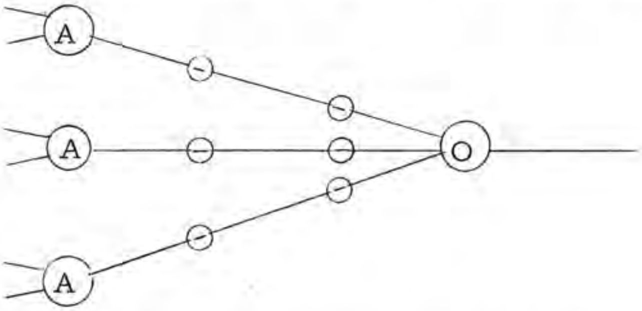


Figure 3. 2. 3. 3. Transformation of basic network.

is the same network as far as the inputs and output are concerned. But then we can observe a couple of other structural equivalences on the NAND gate as shown in Figure 3. 2. 3. 4

$$A | B = \overline{AB} = \overline{A} + \overline{B}$$

or

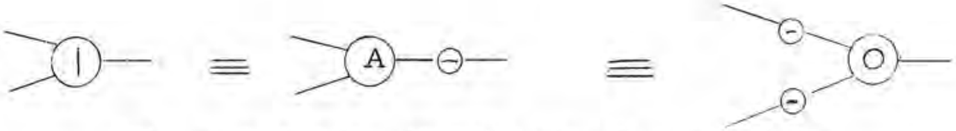


Figure 3. 2. 3. 4. NAND equivalent networks.

and now we are ready to thrust home. In the AND/OR network with inserted NOTs, there is certainly no harm in associating one NOT with the AND, and one with the OR as indicated by the clustering in Figure 3. 2. 3. 5.

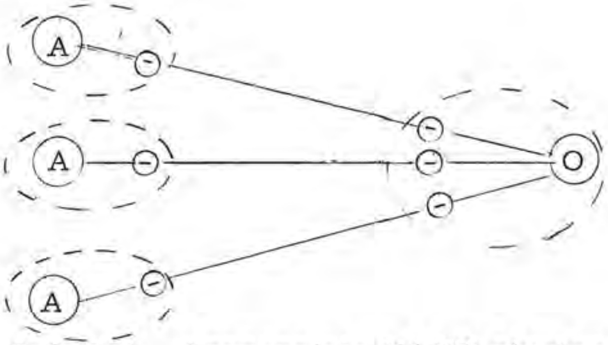


Figure 3. 2. 3. 5. Appropriate NAND clusters.

Clustered this way, however, it is evident that each cluster is a NAND and can be so replaced without any change in the terminal behavior. The equivalent network to the original is as in Figure 3. 2. 3. 6

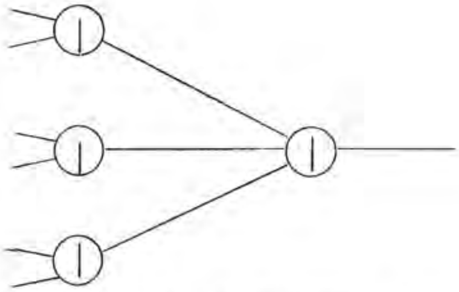


Figure 3. 2. 3. 6. Basic NAND structure.

utilizing nothing but NAND gates, and being directly a map of the minimal AND/OR network we started with. Thus the design algorithm for minimal two-level NAND networks is somewhat as follows:

1. Design the minimal AND/OR network
2. Replace every gate in sight with a NAND

and it's as simple as that.

As a design example, consider f = Σ (0, 4, 5, 6, 7, 13, 15) which maps as Figure 3. 2. 3. 7

				w	
	1	1			
		1	1		z
y		1	1		
		1			
			x		

Figure 3. 2. 3. 7. Example function map.

with the obvious $f_{ms} = w'y'z' + xz + w'x$, and the corresponding minimal NAND network shown in Figure 3. 2. 3. 8.

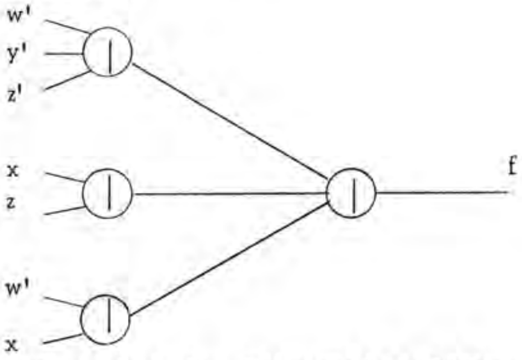


Figure 3. 2. 3. 8. Minimal NAND network.

3.2.4 Two-Level NOR Networks

Having accomplished the NAND situation with so little trouble, we should not be surprised that the NOR case is equally as easily dispatched. By duality we might also expect the transformation to be based upon the OR/AND network rather than the AND/OR. Thus consider the transformations suggested by the NOR algebraic identities of Figure 3.2.4.1.

$$A \downarrow B = \overline{A + B} = \overline{A} \cdot \overline{B}$$

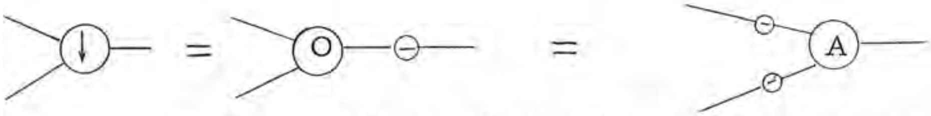


Figure 3.2.4.1. Basic NOR transformations.

It follows that in the two-level OR/AND we can make the direct transformations shown in Figure 3.2.4.2

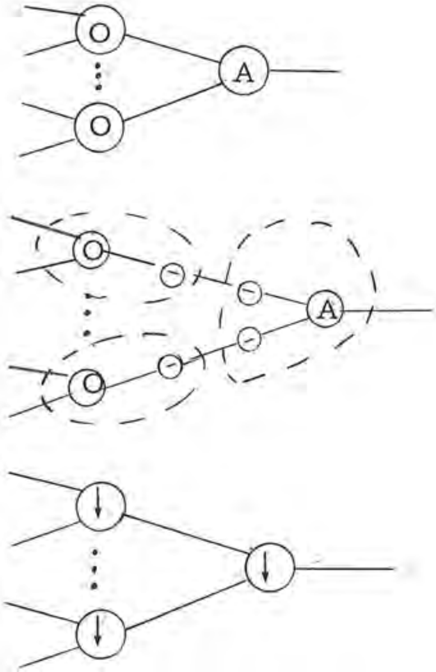


Figure 3.2.4.2. Transformation from OR/AND to NOR/NOR.

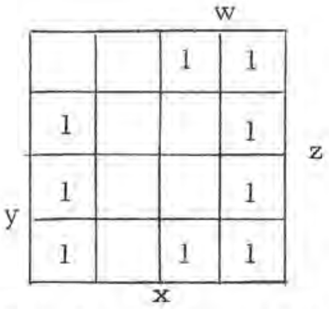
hence conclude that the algorithm for the two-level NOR synthesis can be phrased as

1. Design the minimal two-level OR/AND

2. Replace every gate with a NOR.

Alternatively, we can design the minimal AND/OR network for f', replace every gate with a NOR, and then complement the input variables. Either route will suffice. A point to be reiterated is that minimality is assured in these constructs always by reference to the original structure. If the result is not minimal, then neither was the starting network. But it was, so then also is the derived consequence from it.

We conclude again with the same example f = Σ (0, 4, 5, 6, 7, 13, 15). This time we construct the map for f' as in Figure 3. 2. 4. 3,



		1	1
1			1
1			1
1		1	1

Figure 3. 2. 4. 3. Example NOR map.

determining the f'_{ms} as x'z + x'y + wz', whence the minimal two-lever NOR network is as in Figure 3. 2. 4. 4.

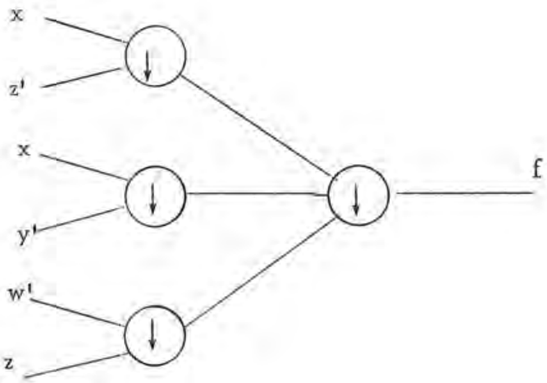


Figure 3. 2. 4. 4. Minimal NOR network.

3. 2. 5 Incompletely Specified Functions

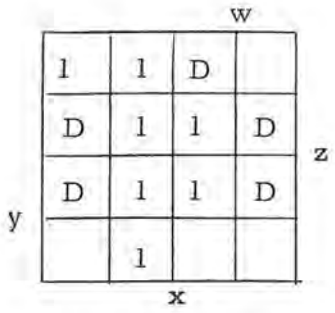
Next we point to a different sort of specification on switching functions, with the end in view of affirming how our same methods of composition can handle the new situation with equal ease.

In certain cases there are certain of the input combinations that are of no consequence in the specification--either because the external environment is such that they will never happen, or because if they do happen other consequences (e. g., self-destruction) will render the output produced by our networks of no consequence. Whatever the reason whenever certain input combinations are present they are called Don't Cares, and they provide a certain extra degree of freedom that the designer can use in simplifying his networks. In the mathematical domain these would correspond to specifying only a subset of the domain, i. e., to incompletely specified functions, as previously noted.

The point is, from the designer's point of view, since these inputs are don't cares so far as the specification is concerned, the designer is then free to assign them as he will in order to further simplify his realizations.

There is an important point to be made here, which often results in confusion when this subject is first considered. Once the physical network is built, then it definitely does care what the output is for all inputs. I. e., the network is deterministic, and will respond in some predetermined way to all inputs--the only point is that the designer can select those particular outputs to suit his own purposes.

For example, suppose we are asked to realize the function on the preceding page, under the conditions that minterms (1, 3, 9, 11, 12) are don't care inputs. We would denote this as $f = \Sigma\ (0, 4, 5, 6, 7, 13, 15) + \Sigma_D\ (1, 3, 9, 11, 12)$ and map it as in Figure 3. 2. 5. 1



1	1	D	
D	1	1	D
D	1	1	D
	1		

Figure 3. 2. 5. 1. Don't care function.

with the D's indicating the don't care nodes. The minimal solution in this case would appear to be to assign the value of 1 to the D's at 1, 3, 9, 11 and the value of 0 to the D at 12. This results in

$$f = z + w'x + w'y'$$

as the simplest realization of this incompletely specified f, in the two-level AND/OR form of Figure 3.2.5.2.

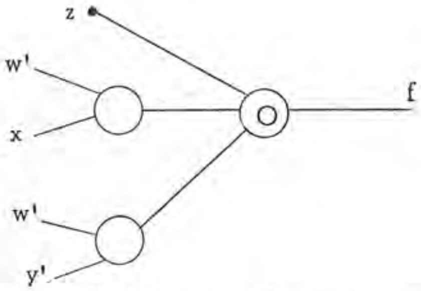
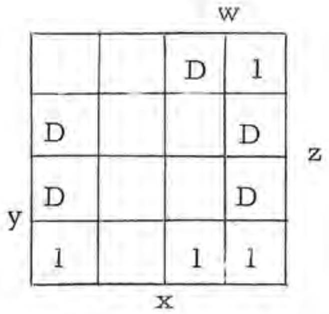


Figure 3.2.5.2. Minimal don't care network.

On the other hand, were we designing a minimal two-level OR/AND for the same function we might look at f' (but with the same don't care nodes) as indicated in Figure 3.2.5.3



		D	1
D			D
D			D
1		1	1

Figure 3.2.5.3. Alternative don't care map.

and come up with the minimal f' as

$$f' = wz' + x'y$$

which in the two-level OR/AND form yields for f the network of Figure 3.2.5.4.

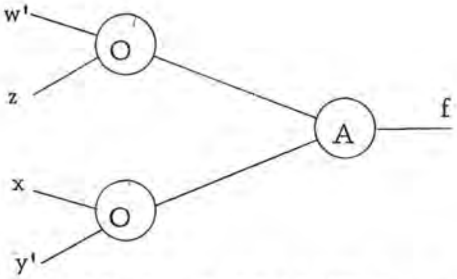


Figure 3.2.5.4. Minimal OR/AND form.

Notice that in this case the D's were assigned quite differently, and in general are freely assigned in any fashion that will make the network simpler. But notice that the two networks are the same only for the specified nodes; for don't care inputs they may not produce the same output value at all. Yet they are both realizations of the particular f as partially specified.

3.2.6 Quine-McCluskey Reductions

The Quine-McCluskey procedure is a tabular representation process for generating the prime implicants of any function, i.e., the result of the process is f_{cs}. While the map methods get cumbersome at five or six variables, the Quine-McCluskey table, adapted to machine realizations, can carry the process on to a significantly greater number of variables, say 12 to 14. The process is best adapted to machines, and is tedious to do by hand, hence our present goal is only to discuss the notions involved, and the interested reader is invited to explore further into specific embodiments of the process on his own.

The basic notion is to simply revert back to numerical equivalents of what we went to the maps to perceive visually. Each minterm is represented by a binary number corresponding to its variables, e.g., wx'yz would be represented by 1011. Adjacencies between two minterms are then detected by the distance between the corresponding codes, i.e., minterms 1011 and 1010 would be detected and replaced by 101-, with the "-" indicating an eliminated variable. All possible such minterm adjacencies are systematically so tested, resulting in two sets (either of which might be empty):

1) a set of minterms which didn't combine with anything, hence are themselves prime implicants, and

2) a set of 1-cubes (e.g., such as 101- above) which are listed in the second column of the developing table.

Now the set of 1-cubes generated in the second column also present possibilities for even further reduction. These can be tested pairwise at a time, exhausting all possibilities, and generating thereby yet a third column of 2-cubes that exist in the function. For example if the coded 1-cube 101- existed in the second column along with the code 111-, then this combination would imply that combination 1-1- should be entered in the third column, indicating that wy was also an implicant of f. In brief this process is continued until no new implicants are indicated.

As a simple example, consider f = Σ (0, 5, 10, 15). We would list the minterms in the first column as in

1st
0000
0101
1010
1111

and would test each possible pair to see if any were distance 1 apart. None are, so we conclude immediately that the minterms themselves are the prime implicants, and the minimal sum is

$$f_{ms} = w'x'y'z' + w'xy'z + wx'yz' + wxyz.$$

Notice that we did at least arrange the minterms in groups corresponding to increasing weight in the minterm representation. That way we only have to examine adjacent groups in order to seek adjacencies; for surely a word of weight 1 can be distance 1 only from words of weight i-1 or of weight i+1.

As a bit more complex example, let us consider f = Σ (0, 4, 5, 6, 7, 13, 15) which we've used as an example before. Again in column 1 we list the minterms in groups corresponding to their weight as in

1st	2nd
√0000 (0)	0-00 (0, 4)
√0100 (4)	
0101 (5)	
0110 (6)	
0111 (7)	
1101 (13)	
1111 (15)	

and have indicated our first pairwise comparison, i.e., that between 0000 and 0100 in the first column. Since these are distance 1 apart we make an entry in the second column of 0-00, indicating that there is an implicant corresponding to w'y'z' in f. Notice also that we are carrying along the minterms contained in each code, which the machine wouldn't do, but it will make it easier for us to keep track of the books. Notice we've also checked minterms 0000 and 0100. The implication of these check marks should not be missed. They do not mean that we won't consider these minterms in other possible combinations with the remaining minterms; they do mean that we can conclude that these particular minterms are not prime implicants since they obviously are included in yet a larger one. We continue in this way, comparing next, for example, 0100 with 0101 yielding 010- as another entry in the second column. Doing this systematically we get

1st	2nd
✓0000 (0)	0-00 (0, 4)
✓0100 (4)	010- (4, 5)
	01-0 (4, 6)
✓0101 (5)	01-1 (5, 7)
✓0110 (6)	-101 (5, 13)
✓0111 (7)	011- (6, 7)
✓1101 (13)	-111 (7, 15)
✓1111 (15)	11-1 (13, 15)

and notice that we have checked all entries in column 1, hence there are no minterms that are also prime implicants. Now all the entries in column 2 are implicants, and we proceed to form a column 3 to determine whether any of them combine into 2-cubes. To combine any pair in column 2 it is again necessary that they differ by distance 1 in their non "-" columns, and that, of course their "-" columns coincide. Thus 0-00 and 010- cannot combine, nor can 01-1 and 10-1, but 010- and 011- can into 01--. Proceeding in this fashion we get for the 3rd column

1st	2nd	3rd
✓0000 (0)	*0-00 (0, 4)	01-- (4, 5, 6, 7)
✓0100 (4)	✓010- (4, 5)	01-- (4, 6, 5, 7)
✓0101 (5)	✓01-1 (4, 6)	-1-1 (5, 7, 13, 15)
✓0110 (6)	✓01-1 (5, 7)	-1-1 (5, 13, 7, 15)
✓0111 (7)	✓-1-1 (5, 13)	
✓1101 (13)	✓011- (6, 7)	
✓111 (15)	✓-111 (7, 15)	
	✓11-1 (13, 15)	

and our process terminates for there are no combinations possible in the fourth column. Two points are worth noting. In the second column we were unable to combine the first entry with anything. We have denoted this by a "*" and this means that we have found a prime implicant. Also notice in the fourth column that everything occurs twice. This is simply a verification that the 2-cubes can be built up two different ways by combining 1-cubes as shown in the two maps of Figure 3. 2. 6. 1.

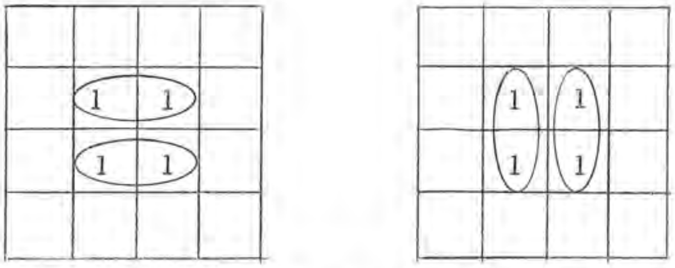


Figure 3. 2. 6. 1. Different two-cube formations.

This effect will always occur so we shall simply remove one of each duplicate pair by a check mark as well. In the formation of a 4th column indicating 3-cubes as implicants, we would of course consider only one of such a duplicate pair in making possible further combinations. Since there are no further combinations possible in the 3rd column, we star the remaining entries that we've been unable to combine further, for this means we have identified them as prime implicants as well.

Thus our final table looks like

1st	2nd	3rd
✓0000 (0)	* 0-00 (0, 4)	* 01-- (4, 5, 6, 7)
✓0100 (4)	✓010- (4, 5)	✓ 01-- (4, 6, 5, 7)
✓0101 (5)	✓01-0 (4, 6)	* -1-1 (5, 7, 13, 15)
✓0110 (6)	✓01-1 (5, 7)	✓ -1-1 (5, 13, 7, 15)
✓0111 (7)	✓-101 (5, 13)	
✓1101 (13)	✓011- (6, 7)	
✓1111 (15)	✓-111 (7, 15)	
	✓11-1 (13, 15)	

from which we conclude that the prime implicants are coded as

0-00

01--

-1-1

corresponding to $f_{cs} = w'y'z' + w'x + xz$ as we saw before. Of course in this case we know that the result is also f_{ms}, but in general we should still have before us the task of selecting the minimal covering set.

3.2.7 Multiple Level Networks

Although most of the standard techniques for the derivation of minimal networks are directed toward the two-level forms, it is inevitable that multiple level networks must always be considered in practical cases of any generality. Of course the salient feature of the two-level networks is that they cause the least time delay possible in the transmission of a signal from input to output.

Of course the number of levels may not be an important criterion in some cases. We may seek to minimize something else, like gate inputs, and in these cases the use of multiple levels can often effect further economies.

General procedures for developing such networks that are demonstrably minimal according to any particular criterion, however, are usually difficult to apply, and are often computationally intractable. As a result, recourse is usually made to ad hoc techniques that come up with networks that are "pretty good" as a rule, but about which no assertions regarding minimality can easily be made. We shall content ourselves in this discourse with simply a couple of examples in order to illustrate the kinds of problems involved.

Consider the simple example

$$f = xyz + x'y' + y'z'$$

which we can quickly check is already in minimal sum form. It follows that the cheapest two-level AND/OR network is in Figure 3.2.7.1.

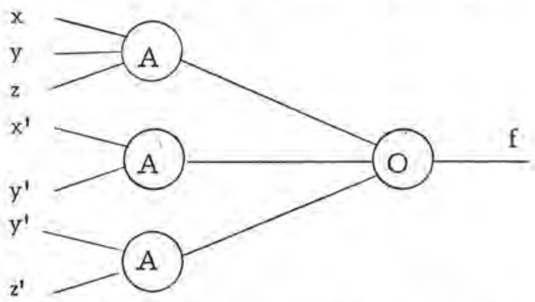


Figure 3.2.7.1. Minimal two-level network for example.

We note that this network "costs" us four gates, and ten gate inputs. Suppose that it is the number of gate inputs that is of interest to us. In this event we note that we can rewrite f by factoring out the common y' in the rightmost two terms of the minimal sum as

$$f = xyz + y'(x' + z')$$

which corresponds to the network of Figure 3. 2. 7. 2

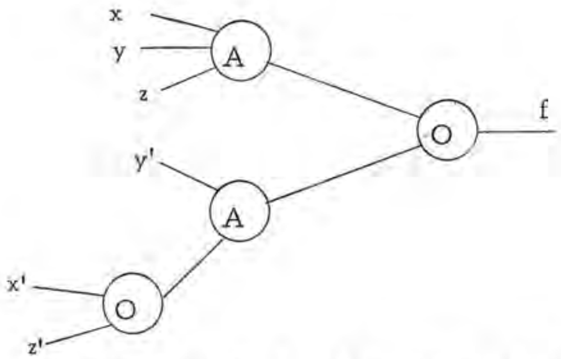


Figure 3. 2. 7. 2. A multiple level form.

requiring only nine gate inputs. Hence a saving can be effected. Notice, however, that the network requires three levels of gates.

Of course, at least in terms of networks comprised of AND gates and OR gates, multiple level networks must always correspond to some factorization of the sum-of-products or of the product-of-sums forms, and conversely. Thus, for example, the expression

$$f = xy(w + z') + x'z(w + yz')$$

certainly is realized with AND's and OR's and since some factorization is indicated we can conclude that it corresponds to a network of a greater number of levels than two. In particular it corresponds to the network of Figure 3. 2. 7. 3

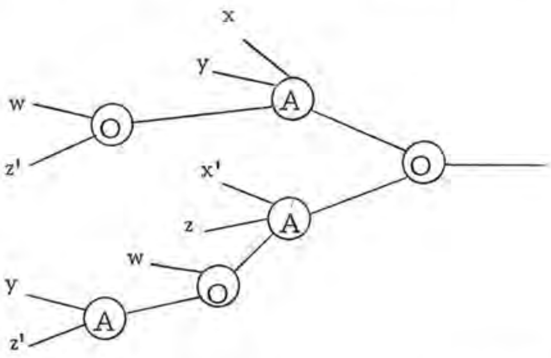


Figure 3. 2. 7. 3. A four-level network.

which exhibits four-levels. It will be left to the reader's curiosity to decide whether simpler networks exist that realize this same function more simply.

Another compelling reason for gaining some familiarity with the multiple level networks is the practical limitations of fanin that must exist in any particular case. The minimal sum form for a function of n variables implies AND gates of up to n inputs, and an OR gate that may require 2^{n-1} inputs: Certainly for most practical networks these input requirements are untenable and some other approach must be taken, and such always results in an increase in the number of levels.

Suppose we wish to limit ourselves to gates of no more than two inputs. This is extreme, but certainly such limitations imply gates of greater reliability than any others. One solution to the dilemma would be to realize the arbitrary AND gate by either cascading the simpler gates, or by treeing them. For example a five-input AND gate can certainly be realized as in Figure 3.2.7.4

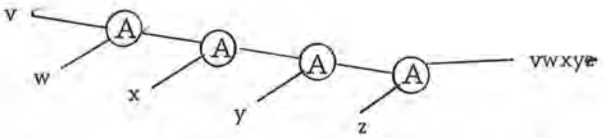


Figure 3.2.7.4. A 5-input AND.

or as in Figure 3.2.7.5

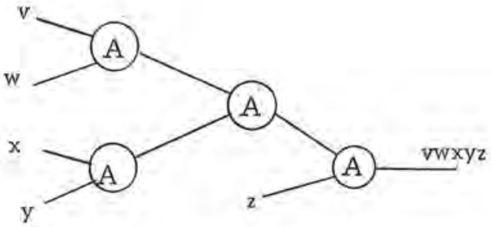


Figure 3.2.7.5. Alternative 5-input AND.

which utilize simpler gates with only two inputs. Obviously the multiple input OR gate can be similarly realized by cascading or treeing simpler OR gates. As an example, an earlier realization of

$$f = xyz + y'(x' + z')$$

lessened the number of gate inputs, but still required one AND gate of three inputs. If our limitations held us to two-input gates, then an alternative realization for the same network would be as in Figure 3.2.7.6

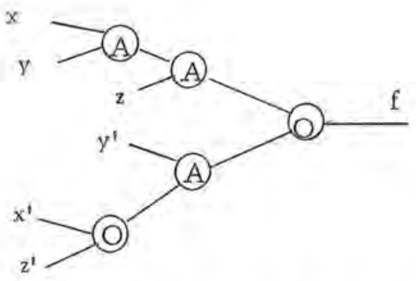


Figure 3. 2. 7. 6. Fanin limited network.

using nothing by two-input gates.

Also of interest would be the extension of these possibilities to gates of other types, for example NANDs and NORs. To illustrate these possibilities algebraically is a notationally messy (hence error prone) task. However, in most cases, transformations such as we've utilized before will suffice. Suppose, for example, that we have the function

$$f = vwxyz + v'w'x'y'z'$$

to realize. In minimal two-level form the network is clearly as in Figure 3. 2. 7. 7

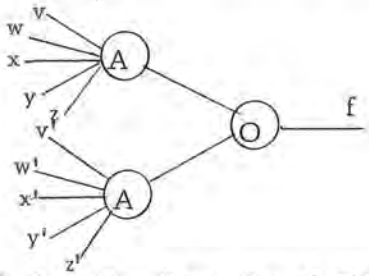


Figure 3. 2. 7. 7. Two-level NAND example.

requiring five-input AND gates. Suppose we are limited to only three inputs. Certainly one solution is as in Figure 3. 2. 7. 8

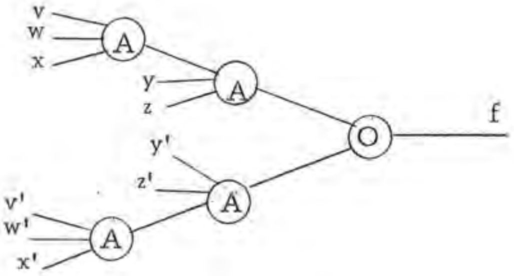


Figure 3. 2. 7. 8. Network with fanin of three.

by our methods above, and requiring three levels. Now suppose we further wish to realize the network utilizing only NAND gates, again with the three-input limitation on fanin. Utilizing the identity transformation that assures us that two NOTs in cascade does not affect the logic function on a lead, we can redraw the previous network as in Figure 3.2.7.9

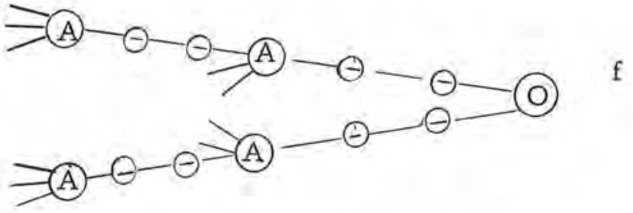


Figure 3.2.7.9. Insertion of NOT gates.

where we have inserted NOTs judiciously in order for us to proceed with the transformation to NANDs. The transformation is indicated by the dotted lines in Figure 3.2.7.10

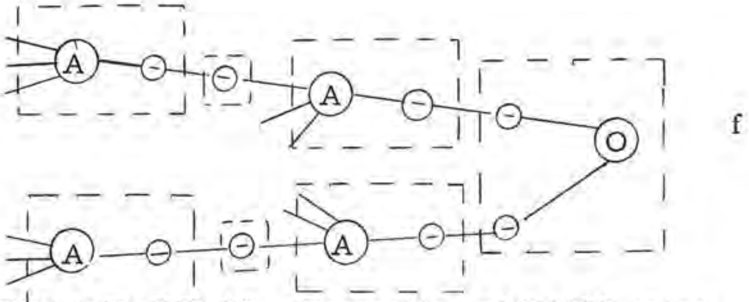


Figure 3.2.7.10. Formation of NAND gates.

where we note that we have one set of NOTs left over that we can't combine with either an AND or an OR. But that's all right, for of course a NOT is just a particular case of the NAND (one with its inputs tied together) and these observations lead us directly to Figure 3.2.7.11

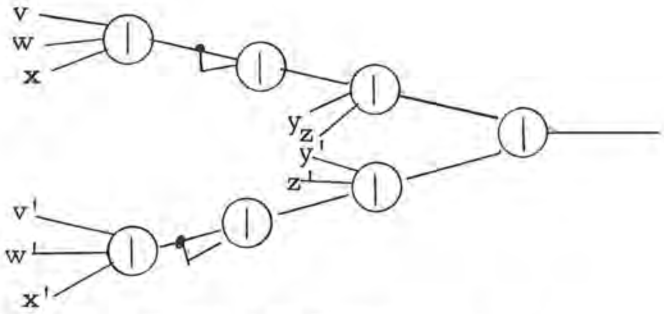


Figure 3.2.7.11. Final NAND gate realization

as a final realization of the function utilizing NANDs of no more than three inputs.

We could do a similar set of transformations for a NOR realization, or indeed for any mix of such gates that we might wish. We could extend these kinds of techniques to similar considerations for other sets of logic gates, but for our purposes the present discourse will suffice for our brief look at the nature of the multilevel design problem.

3.3 Multiple Output Networks

We shall mainly ignore this large class of networks, which is the obvious generalization of single-output network synthesis, except to make a couple of observations on the nature of the problem.

Clearly, as we move from the single output realization of Figure 3.3.1

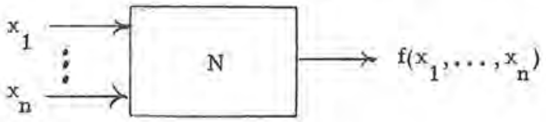


Figure 3.3.1. Single output network.

to the multiple output realization of a set of swtiching functions of Figure 3.3.2,

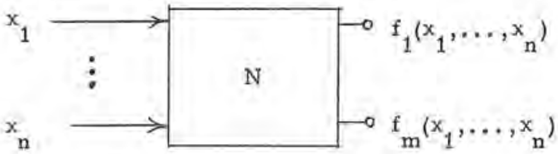


Figure 3.3.2. The general multiple output network.

the basic first question of interest to switching theorists, namely that of <u>existence</u> of a solution, is already answered. For we always have an immediate realization in terms of m separate, single-output networks, as indicated in Figure 3.3.3.

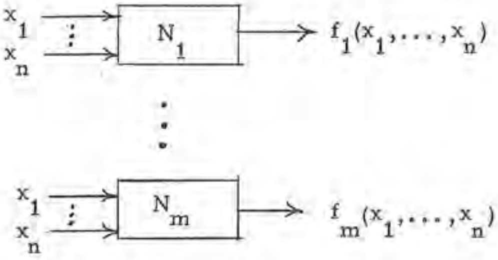


Figure 3.3.3. Single output realization of multiple output network.

Thus the only practical problem of interest is that of minimization--in many cases large savings can be made by realizing substantial portions of the N_i in common. This, then, is the only object of multiple output synthesis.

3.3.1 Bilateral Networks

We have previously noted that the bilateral (contact) networks can be tricky, largely resulting from the fact that they are bilateral in the first place. Any path we set up from one point to another, say from point A to point B, also sets up a path from point B to point A. This fact makes multiple output savings more complex in general for relay networks (indeed impossible for purely contact networks not involving coils internal to the network).

A large and interesting class of problems involve multiple output sets which are disjunctive, that is each pair f_i, f_j in the desired set of functions $f_1, \ldots, f_m$ are disjoint, i.e., $f_i f_j = 0$. Even in this case, when trying to eliminate as many contacts as possible, the designer must be very careful not to introduce so-called "sneak paths", which are quite literally design mistakes that result in disjoint outputs being connected together inadvertently for some values of the input variables.

It is not our purpose to examine bilateral synthesis to any degree of thoroughness at this time. Instead we shall only point to a couple of examples of what can be done.

The complete tree on n variables (sometimes called the complete decoding tree) often provides a convenient starting point for the realization of sets of disjoint functions; indeed it directly provides a realization of the set of functions wherein each function is a minterm (which is why it's called a complete decoding network). Thus Figure 3.3.1.1

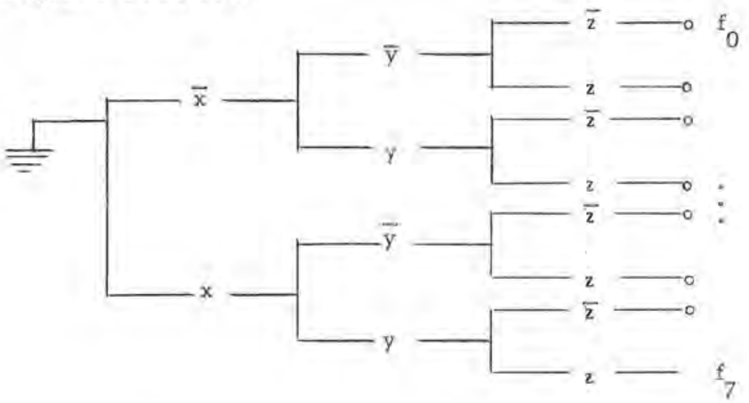


Figure 3.3.1.1. The complete decoding tree.

is an evident realization of the set of functions

$$f_0(x, y, z) = x'y'z'$$
$$f_1(x, y, z) = x'y'z'$$
$$\vdots$$
$$f_7(x, y, z) = xyz$$

Of course any other disjoint set can be realized just by collecting together the minterms contained in each. For example the set

$$f_1 = \Sigma\,(0, 1, 2, 3, 7)$$
$$f_2 = \Sigma\,(4, 6)$$

is evidently realized by the tree of Figure 3. 3. 1. 2,

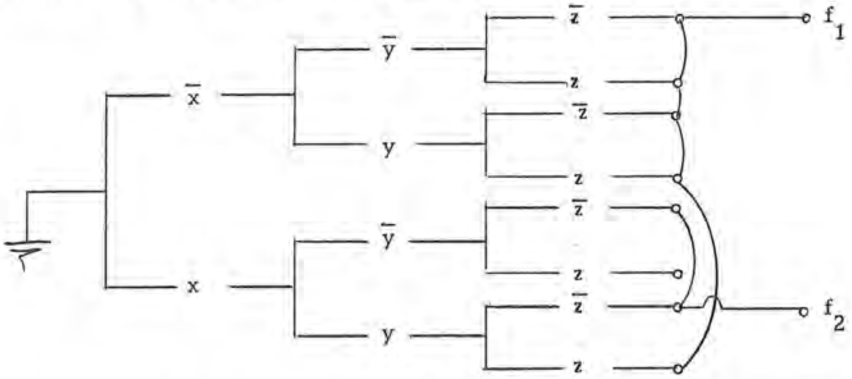


Figure 3. 3. 1. 2. Realization of two output functions.

and carrying out the obvious simplifications observed, before, we quickly get the network of Figure 3. 3. 1. 3

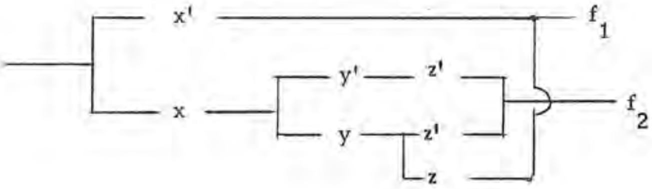


Figure 3. 3. 1. 3. Simplified two-output network.

as a pretty good realization of the function pair. (One must forgo the temptation to "factor out the pair of z' contacts since this would introduce an incorrect path to f_1.)

Many function classes possess particularly simple and regular tree expansions in this sense. An important class of such functions are those called the symmetric functions, in particular the elementary symmetric functions. A function that consists of all the minterms of a given weight i and nothing else is an elementary symmetric function S_i. Thus in terms of three variables

$$S_0(x, y, z) = x'y'z'$$
$$S_1(x, y, z) = x'y'z + x'yz' + xy'z'$$

$S_2(x, y, z) = x'yz + xy'z + xyz'$

$S_3(x, y, z) = xyz$

These functions turn out to play a special role in many areas. Our purpose here is to illustrate their realization in terms of the complete tree. Again in terms of three variables we get an immediate realization as in Figure 3.3.1.4.

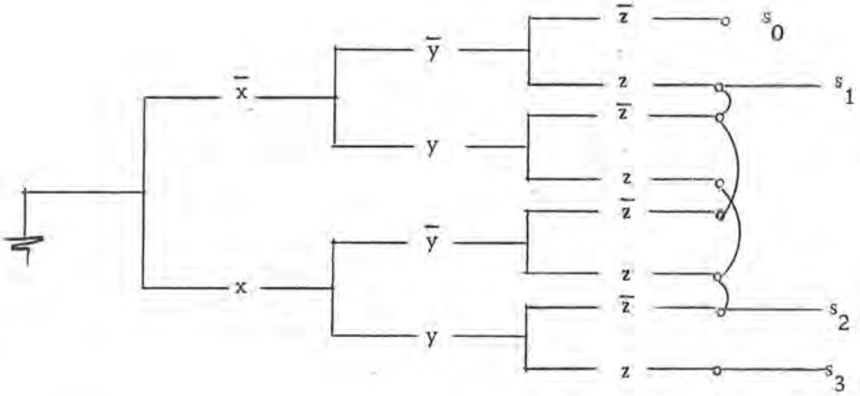


Figure 3.3.1.4. Realization of symmetric functions.

This time we are tempted to factor a "z" from the S_1 output, and a "z" from the S_2 output, and if we do it carefully it will work this time, and we get something like Figure 3.3.1.5,

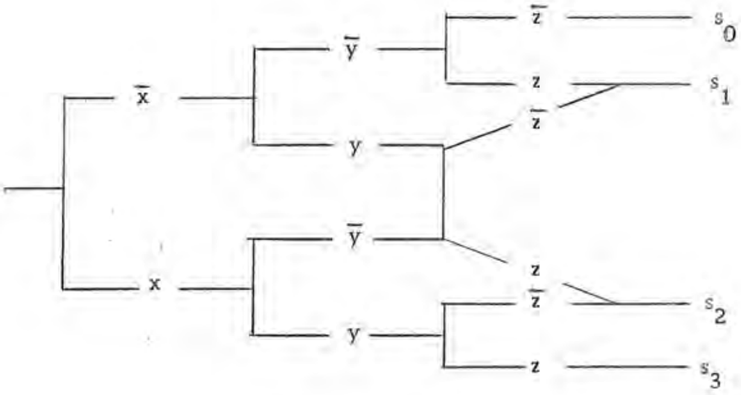


Figure 3.3.1.5. Simplified symmetric function network.

and if we redraw this carefully we get a very well known form, as shown in Figure 3.3.1.6.

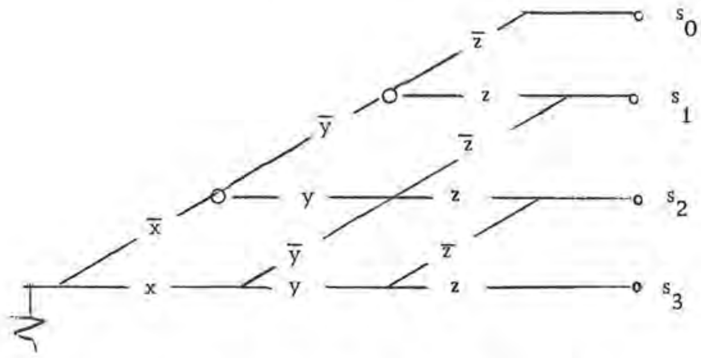


Figure 3.3.1.6. Final form of network.

This form generalizes for any number of variables. For example for four variables we have the network of Figure 3.3.1.7

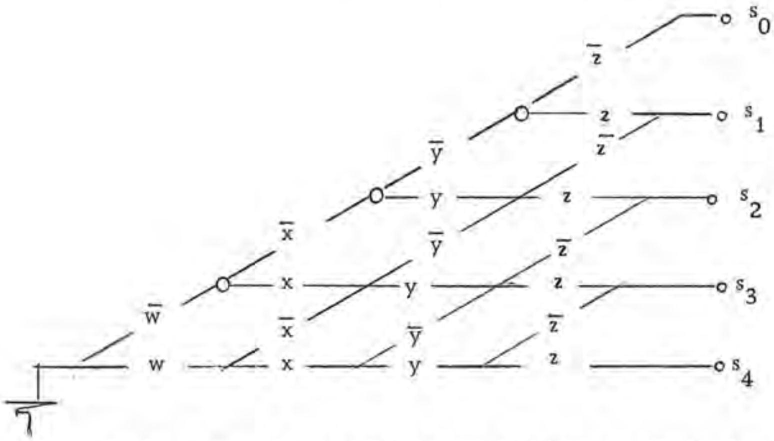


Figure 3.3.1.7. General form of symmetric function network.

and so on. The number of contacts C(n) grows like $2+4+6+\ldots \doteq 2(1+2+3+\ldots+n)$ or as n(n+1), again a considerably better growth rate than for the general tree.

Again there are many more things we could say about branch element synthesis, but these examples will suffice for our present purposes.

3.3.2 Gate Networks

The goal in multiple output gate networks is the same as before; having generated a particular gate output, we'd like to use it in as many places as possible in common.

We shall proceed mainly in terms of a couple of examples in order to indicate possible approaches to the problem of synthesis. Suppose we have the three functions set to realize:

$$f_1 = (0, 4, 7)$$
$$f_2 = (0, 3, 4)$$
$$f_3 = (3, 4, 5, 6, 7)$$

If we examine the corresponding maps of Figure 3. 3. 2. 1

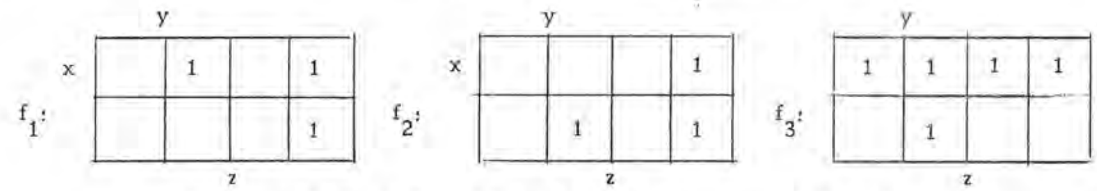


Figure 3. 3. 2. 1. Maps of three-output function set.

we find that the best prime implicant expansions are

$$f_1 = xyz + y'z'$$
$$f_2 = x'yz + y'z'$$
$$f_3 = x + yz$$

and it seems evident that the best solution would be to use the implicant y'z' in common as in Figure 3. 3. 2. 2.

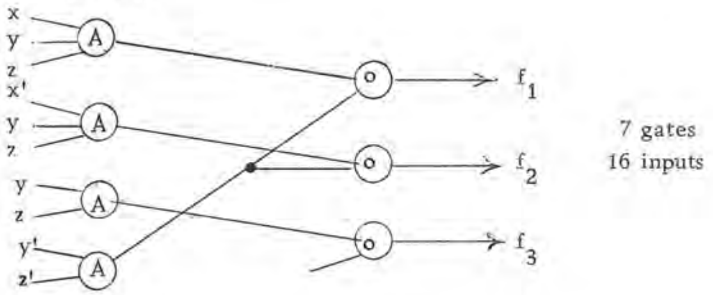


Figure 3. 3. 2. 2.

But a little closer examination shows that the yz term can be formed from two other expressions that are also needed, i. e., xyz for f_1, and x'yz for f_2. Since these two have to be formed separately anyway, it makes better sense to use them to form yz, rather than form it separately. Doing this we get the network of Figure 3. 3. 2. 3,

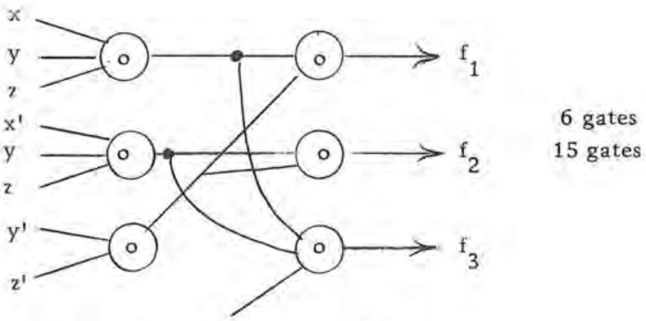


Figure 3.3.2.3. Simplest multiple output network.

which is cheaper than our original try.

This possibility would have been systematically detected had we gone to the complete class of so-called multiple input prime implicants in making our choices. This complete class consists not only of the prime implicants functions, but of all their intersections as well. Thus

$f_1 = \Sigma\,(0, 4, 7)$	with p.i.'s	xyz	$y'z'$
$f_2 = \Sigma\,(0, 3, 4)$	with p.i.'s	$x'yz$	$y'z'$
$f_3 = \Sigma\,(3, 4, 5, 6, 7)$	with p.i.'s	x	yz
$f_{12} = \Sigma\,(0, 4)$	with p.i.'s	$y'z'$	
$f_{13} = \Sigma\,(4, 7)$	with p.i.'s	xyz	$xy'z'$
$f_{23} = \Sigma\,(3, 4)$	with p.i.'s	$x'yz$	$xy'z'$
$f_{123} = \Sigma\,(4)$	with p.i.'s	$xy'z'$	

from which we can list the prime implicants to serve as candidates for our covers as

$$A = xyz$$
$$B = y'z'$$
$$C = x'yz$$
$$D = x$$
$$E = yz$$
$$F = xy'z'$$

Then we can proceed to a generalization of the prime implicant table which shows each output function in an independent section, such as in

			↓ A	↓ B	↓ C	↓ D	E	F
f_1	*	0		x				
		4		x				x
	*	7	x					
f_2		0		x				
	*	3			x			
		4		x				x
f_3		3			x		x	
		4				x		x
	*	5				x		
		6				x		
		7	x			x	x	

On this table we have indicated certain rows with stars: these indicate rows that have a single x in them, hence require that the corresponding prime implicants be realized. These implicants are indicated by arrows on the columns.

Of course these selected implicants will also cover other rows as well, and we could systematically proceed to a reduced prime implicant table, as before, in order to select which of the remaining implicants are needed in order to completely cover all the functions. Doing this in this case we shall find that neither implicants E nor F are needed, hence we are through, and need realize only A, B, C, D which is what we included in our realization of the network above.

Thus the multiple output case from this point of view can be approached using the same machinery developed for the single output case. Obviously the tables get bigger, and the detection mechanism must be more carefully executed, but the principle is the same.

In passing we shall mention another conceptual approach which is characteristic of many arguments made in combinational network theory, although we shall not pursue examples of it at this time. The notion is to somehow reduce the multiple problem at hand to a simpler problem represented by the "previous case", which is presumably completely solved. In this case we can force the general multiple output problem into a strictly single-output problem provided that we appropriately "dummy" things up. Using our three variable three-function problem as an example, let us create a single output function f of five variables

$$f(v, w, x, y, z) = vwf_1(x, y, z) + vw'f_2(w, y, z) + v'wf_3(x, y, z)$$

using the dummy variables v and w to point to which of the original functions is involved. (Clearly we could so handle up to four output functions with the two dummy variables.) Now we have explicitly a single-output function, and can minimize it to exhaustion using our techniques for such minimization. Presumably our result is some other realization for f than the form above. It remains to form the three separate outputs, but this can easily be done in a final gate for each output where we specify whixh output we wish by use of the dummy variables again. Thus $f_1 = vwf$, $f_2 = vw'f$, $f_3 = v'wf$, and so on, as in Figure 3.3.2.4.

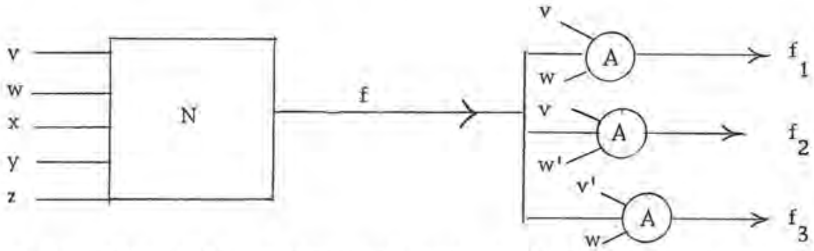


Figure 3.3.2.4. Multiple output set reduced to single output problem.

Depending upon the facility with which we can carry out the single-output minimization on the increased number of variables, this approach can be a quite powerful technique.

3.4 Iterative Combinational Networks

While the two-level logic networks optimize the speed with which a network can respond with the truth value of an arbitrary switching function, there are many other features that we might prefer to optimize. One of these is an enforced degree of commonality with which each variable is treated, as exemplified by the iterative combinational networks.

The simplest iterative network is the linear cascade, or one dimensional network characterized by the general structure of Figure 3.4.1

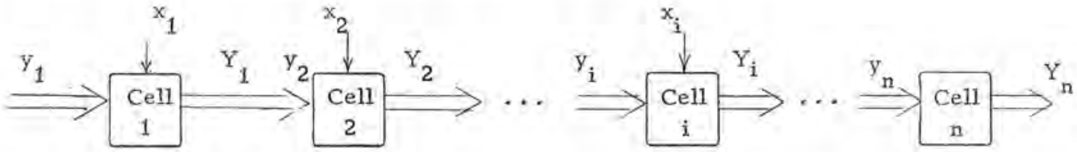


Figure 3.4.1. General iterative combinational cascade.

wherein each variable enters a cell, and it is understood that each cell internally is to be the same as every other. The variables y are assigned to the leftmost inputs of cell i, and the variables Y are assigned to the outputs on the right which proceed on down to the next cell of the cascade. Obviously $Y_i = y_{i+1}$. Of course the y_1 must be considered boundary conditions, to be supplied externally, and the variables Y_n are generally used to form the "outputs" of the iterative network. (Other conceptualizations are possible, and useful: e.g., there is no reason to limit the cell inputs to one independent variable, in general there might be k such x inputs; there is also no reason to restrict the outputs to the last cell, each cell might also be required to produce a set of, say, m outputs $z_1, \ldots, z_m$ along the entire cascade. We shall make these generalizations only if required in our discourse, however. There is also no good reason to limit ourselves to the linear, or one dimensional cascade either. Two dimensional, or indeed n-dimensional arrays are also but direct generalizations of the linear cascade. Various degrees of connectivity between cells of the cascades and arrays are also possible, but these need not concern us in this first-pass discourse either.)

Such iterative networks are also described as cellular arrays or networks, and much that is relevant to them in the literature will be found under this name.

For present purposes the linear iterative cascade will serve eminently, and we shall restrict our present discussion to it.

The essential feature about the linear cascade is contained in a consideration of the nature of the y_i, the signals generated to the left of cell i. For they must obviously contain complete information about the first i-1 variables, sufficient so that on observing their value, cell i can examine variable x_i, and generate a sufficient set of similar signals to pass along to cell i+1.

Now it can be shown in general that any function can be realized in such a cascade, although some functions can be realized more simply than others. Since our goal is to illustrate the nature of the cascades themselves, we shall restrict ourselves only to the kinds of functions that have simple realizations.

Functions that are most easily realized are those in which the particular variables involved in the specification are immaterial, but rather some grosser characterization of the true minterms is possible. Such a characterization is exemplified, for example, by any specification of a function f that requires it to be true depending upon the number of variables that are true, e.g., the function that is true iff exactly 3 of the independent variables are true, i.e., $S_3(x_1, \ldots, x_n)$. Another would be that function which is true iff an even number of variables has been true. It is irrelevant in these descriptions just which variables are involved, but only how many. Another sort of problem is one involving a pattern amongst the variables: for example we might specify f to be true iff the pattern 1011 has appeared somewhere amongst the variables in their ordering from 1, ..., n. We might not care what the rest of the variables have been, or we might require that they all be 0. These are two different problems, but both are within the class that we are pointing to. The point is, again, that it does not matter which variables satisfy the criteria, but only that the total cascade of variables do.

For problems of such nature, the role to be played by the y_i is clear; the y_i must be sufficiently complex only to transmit the information to cell i as to the "state" of affairs up to that point in the cascade, so that cell i can generate similar "state" information along to the next cell in the cascade, of course as appropriately modified by the action of cell variable x_i itself.

A simple example should help to clarify some of the notions involved, and will suggest a general procedure to be followed. As a starter let's fall back on an old acquaintance, the parity function. Let us assume that we wish to realize the even parity function in an iterative cascade. Notice that the parity function is one of the class of things we've been pointing to: it matters not which variables enter into the relation, only that an even number of them have been true. Now we focus attention strictly on cell i of the cascade, as in Figure 3. 4. 2.

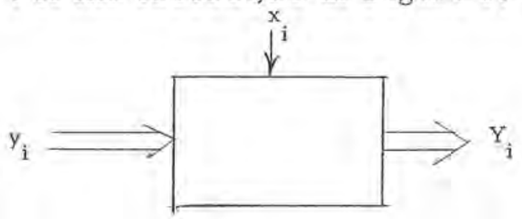


Figure 3. 4. 2. Typical cascade cell.

Obviously we can do nothing about variable x_i except respond to it. The significant thing is to examine the nature of the y_i. They represent signals from the left indicating what has happened so far, and for this problem we need only the answer to one bit of information: have the true variables to cell i been even or odd? We can "encode" this information about the state of the cascade through cell i-1 by using only one binary lead, although the encoding is still up to us. For the single binary variable from the left let us assign $y_i = 0$ to odd parity and $y_i = 1$ to even parity. Clearly if we have the case $y_i = 1$ and $x_i = 0$, then we wish to prescribe that $Y_i = 1$ as well for the action of cell i has not affected the parity to be passed along to cell i + 1. On the other hand if $x_i = 1$, then we wish to prescribe that $Y_i = 0$, indicating that the parity has changed. But this is clearly as prescription of cell i as a two-input network on y_i, x_i, as input variables which realizes the function Y_i. Using our familiar truth table techniques we can prescribe cell i as defined by

x_i	y_i	Y_i
0	0	0
0	1	1
1	0	1
1	1	0

whence $Y_i = x_i y_i' + x_i' y_i = x_i \oplus y_i$ realized by either the cell of Figure 3. 4. 3

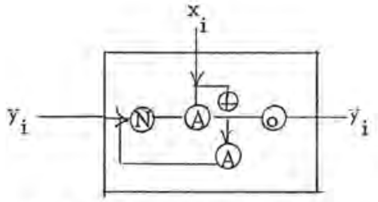


Figure 3.4.3. A cell realization.

or by simply the cell of Figure 3.4.4.

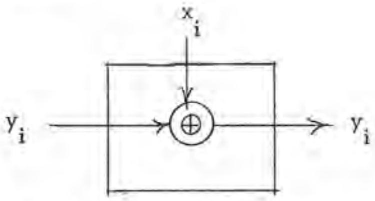


Figure 3.4.4. Alternative cell realization.

Evidently it is important that the first cell of the cascade be properly set to the value $y_1 = 1$, since that would be consistent with the notion of even parity. Finally f is "realized" on the last cell of the cascade, Y_n, by detecting whether $Y_n = 1$ or not. Thus a three-variable cascade would look like that of Figure 3.4.5.

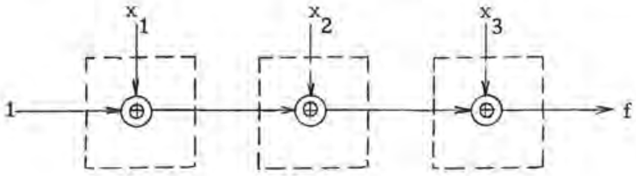


Figure 3.4.5. The three-variable cascade.

To harken back to familiar things, let us verify what the same question would yield for the branch network realization. In this case we start from the abstract picture of the nature of the information to be transmitted by the "state" variables: we must transmit two states, one for even parity and one for odd parity. Again because the branch networks are basically a different beast, however, we must retain both states as separate leads. The reason for this is that the particular state transmitted forward will be indicated by a ground on one of the leads, hence the two states must be kept separate, for a ground proceeding on both would indicate that the input to the cell i was "both" states, while a ground on neither would indicate "neither" state was occupied. Either is equally untenable, so we keep the states distinct on two separate leads. Thus Figure 3.4.6

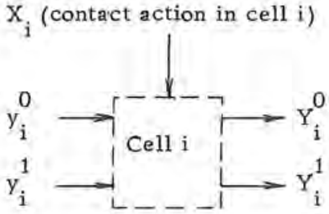


Figure 3.4.6. Typical branch element cell.

is our general picture in this case, and it remains only to fill in the box with contacts in order to generate the Y_i. We could go to a truth table representation in this case as well, although it would have many don't cares. Suffice it to intuite by inspection in this case, and this is usually possible for contact networks, that the formulation in this case is

$$Y_i^1 = y_i^1 x_i' + y_i^0 x_i$$

$$Y_i^0 = y_i^1 x_i + y_i^0 x_i'$$

which is directly realized by the contact configuration of Figure 3.4.7

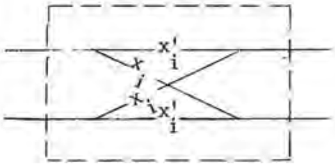


Figure 3.4.7. Branch element realization of typical cell.

which ought to look strangely like the configuration we've seen before when deducing the form of the parity function network from the complete tree.

As another example, suppose that we wish to realize in a branch-type iterative cascade three different outputs on an arbitrary number of variables, in particular the outputs are to correspond to the number of true variables modulo 3. The reader is invited to confirm that the cell in Figure 3.4.8 on the contacts of x_i do indeed realize this function.

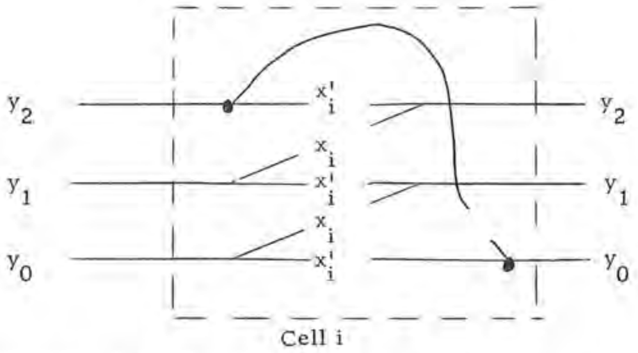


Figure 3.4.8. Mod tree cell realization.

On the other hand, a gate type realization could take advantage of a binary encoding on the input leads to realize this same function set. We only need observe which of the four states from the left obtains, and to do this we need only two binary variables. But as we know, two binary variables can encode four different states, hence we shall have one input combination as a don't care. We shall arbitrarily assign this don't care to the 11 states on y_1, y_2 and agree to assign the encoding

y_1	y_2	State
0	0	0 mod 3
0	1	1 mod 3
1	0	2 mod 3
1	1	don't care

leading to the truth table for output functions Y_1 and Y_2:

x_i	y_1	y_2	Y_1	Y_2
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	D	D
1	0	0	0	1
1	0	1	1	0
1	1	0	0	0
1	1	1	D	D

Using our map techniques to find reasonable realizations for these functions we get the functions of Figure 3.4.9,

Y_1: y_1 above, x_i beside the first row, y_2 below

	D	1	
1	D		

Y_2: y_1 above, x_i beside the first row, y_2 below

	D		1
	D	1	

Figure 3.4.9. Mod three cell intercell functions.

or

$$Y_1 = x_iy_2+x_i'y_1 \qquad Y_2 = x_iy_1'y_2'+x_i'y_2$$

as realized by the cell of Figure 3.4.10.

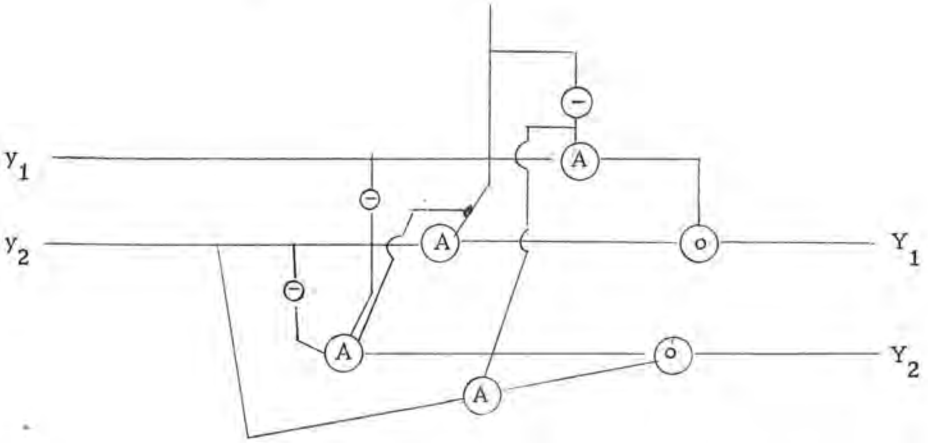
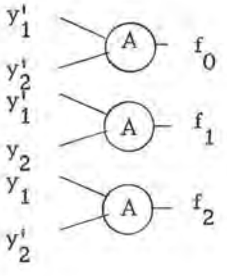


Figure 3.4.10. Gate realization of typical cell.

Of course the final output from the cascade is denoted by the encoding on the final Y_1, Y_2 pair. If these outputs are desired as a set of final output lines, then we shall have to have a collecting set of AND gates on the output something like



with intervening NOT gates as required.

As a final example of this kind of design, we shall consider a problem more in the "pattern recognition" class, rather than in the "weight" classes treated so far. Suppose we wish to realize in an iterative cascade the function which is true iff the pattern 1101 has been observed, and otherwise all the independent variables are 0.

First we must carefully decide the disjoint set of states that will properly describe the cascade to cell i, so that it can correctly modify the state of cell i+1. A sufficient way of describing the necessary set of states is the following:

State	Description
A	Have seen nothing but 0's so far
B	Have just seen a 1
C	Have just seen a 11
D	Have just seen a 110
E	Have just seen a 1101
F	Flush-pattern cannot be realized

The only new concept here is the "flush" state and it corresponds to the fact that the information to cell i in the cascade is such that the function can never be realized, regardless of the remaining length of the cascade; hence the value of x_i (and x_{i+1}, etc., become immaterial).

This is a six-state cascade, hence at least three variables are needed to encode the information. Of course three variables can encode eight states, so we shall have at our disposal two states to assign as don't cares. Otherwise the procedure is exactly analogous to the preceding, and the remainder of the solution for this cascade is left to the intuition, and interest of the reader.

3.5 Notes - References - Problems

NOTES.

Minimization problems have been extensively covered in the literature in almost every conceivable variety. Early map methods were introduced by Veitch (8), although the usual present form of the map is due to Karnaugh (2). Tabular reductions were devised by McCluskey (5) as an adaptation of earlier work by the logician W. V. O. Quine. The case of multiple-level networks is treated by Lawler (4) and Karp (3), while the multiple output minimization problem is found in Bartee (1) and McCluskey and Schorr (7). The basic work on iterative combinational networks is also by McCluskey (6). Again many other modern texts will be found to include a review of most of this material.

REFERENCES.

1. Bartee, T. C., "Computer Design of Multiple Output Logical Networks," IRE Trans. Electr. Compt., vol. EC-10, pp. 21-30, 1961.

2. Karnaugh, M., "The Map Method for Synthesis of Combinational Logic Circuits," Trans. AIEE, vol. 72, pt. 1, pp. 593-598, 1953.

3. Karp, R. M., "Functional Decomposition and Switching Circuit Design," SIAM J., pp. 291-335, June 1963.

4. Lawler, E. L. "An Approach to Multilevel Boolean Minimization," J. ACM, vol. 11, pp. 283-295, 1964.

5. McCluskey, E. J., "Minimization of Boolean Functions," Bell Syst. Tech. J., vol. 35, pp. 1417-1444, 1956.

6. McCluskey, E. J., "Iterative Combinational Switching Networks: General Design Considerations," IRE Trans. Electr. Compt., vol. EC-7, pp. 285-291, 1958.

7. McCluskey, E. J., and Schorr, H., "Essential Multiple-Output Prime Implicants," in Mathematical Theory of Automata, Proc. Polytech. Inst. Brooklyn Symp., vol. 12, pp. 437-457, 1962.

8. Veitch, E. W., "A Chart Method for Simplifying Truth Functions," Proc. ACM, Pittsburgh, Pa., pp. 127-133, 1952.

PROBLEMS.

1. Consider the function $f(w, x, y, z) = \Sigma(0, 4, 5, 7, 8, 9, 13, 15)$ (with w, x, y, z arranged in order of decreasing weight):

   a) Realize f in a contact network using the least number of simple contacts that you can.

   b) Determine f_{cs}. Is it unique? If not, list them all.

   c) Determine f_{ms}. Is it unique? If not list them all.

   d) Realize f in a minimal two-level AND/OR network.

   e) Realize f in a minimal two-level OR/AND network.

   f) Realize f in a minimal two-level NAND network.

   g) Realize f in a minimal two-level NOR network.

2. Draw a relay contact network for each of the following functions:

   a) $f = A(BC' + B'C)$

   b) $f = ABC + AB'C + A'BC$

   c) $f = AD + BC + (B + C)(A + D)$

   d) $f = (AB + C)(BC + D)(CD + A)$

   e) $f(A, B, C, D) = \Sigma(0, 2, 6, 15)$

   f) $f(A, B, C, D) = \pi(1, 5, 10, 13)$

3. Find all minimal sums for the functions in Problem 2.

4. Draw minimal two-level gate networks for the functions in Problem 2 (using AND-OR logic).

5. See how much further you can simplify the realizations of Problem 4 if you are not restricted to two-level networks.

6. a) Realize $g = \Sigma(0, 4, 5, 7, 8, 9, 13, 15) + \Sigma_D(2, 3, 11)$ in a minimal two-level AND/OR network.

   b) Realize g in a minimal two-level OR/AND network.

7. a) Design a minimal AND-OR network (two-level) for the function

   $$f = \quad (0, 1, 2, 5, 6, 13, 14).$$

   b) Do the same for $f = \Sigma(1, 4, 6, 8, 11, 12) + \Sigma_D(2, 5, 13, 15)$

      (a) first by ignoring the don't cares

      (b) then by utilizing them

8. For the specification $f = \Sigma(0, 1, 3, 4, 9, 10, 11) + \Sigma_D(5, 15)$ with the variables ordered according to decreasing weight, find a minimal sum expression for f. Is your expression unique? If not, how many equivalent forms are there? ($f = f(x_4, x_3, x_2, x_1)$).

9. Find an economical contact realization for the function specified in problem 8. (The number of single contacts is the cost criterion.)

10. Find a minimal NOR gate realization (two-level) for the function specified in problem 8. (You may assume that complemented variables are available as inputs, as well as the uncomplemented variables.) Is your network unique? If not, how many minimal forms are there?

11. You are given a box of exclusive-OR gates and AND gates. You are asked to realize $f(x_4, x_3, x_2, x_1) = \Sigma(3, 8, 9, 10, 12, 13, 15)$.

But on investigation you find that your exclusive-OR gates have only 3 inputs, and your AND's only 2 inputs, and that only uncomplemented variables are available as inputs.

You do find, however, that the input combinations corresponding to rows 2, 11, and 14 of the table of combinations are guaranteed never to occur.

What's the best network you can design under these conditions if each of your gates is equally costly?

12. Find one of the simplest functions of four variables, using uncomplemented variables only, for which (a) $w(f) = 7$, (b) $w(f) = 10$, and $w(f) = 13$. "Simplest" means the least number of variable occurrences, and we allow only AND and OR operations. Write out the minimal expression in each case.

13. Use the tabular Quine-McCluskey method to develop all the prime implications of the seven-variable function $f = \Sigma(47, 67, 75, 99, 107)$.

14. A binary full adder is a 3-input, 2-output net that provides the sum (S) and carry (C) digits in response to two inputs (a and b) and a previous carry (c). Synthesize a gate network (assuming all x_i and x_i' available) using AND and OR gates to simultaneously realize S and C with the minimum number of gate inputs.

15. Suppose that you also had exclusive -OR gates available in Problem 14. How would your realization change?

16. Realize a typical cell in an n-variable iterative cascade which will produce a true output if and only if exactly two of the input variables in succession have been true. How many minterms does the n-variable function contain?

a) Assume relay contacts only for your realization.

b) Assume gates (ANDs, OR's, and NOTs only) are available.

4 Introduction to Sequential Design

4.1 Sequential Logic Circuits

The logical circuits we have studied up to this time are known as combinational logic circuits. Schematically they can be represented as shown in Figure 4.1.1. The output Y is a function of the input X.

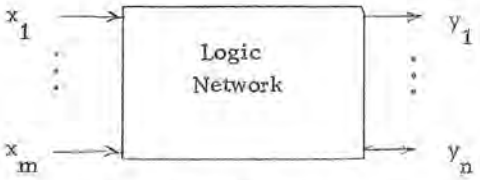


Figure 4.1.1. The Combinational Logic Circuit.

Each element of the output may be written as a logical function of the input vector $X = (x_1, x_2, \ldots x_m)$

$$\begin{aligned} y_1 &= f\,(x_1, \ldots x_m) \\ y_2 &= f_2(x_1, \ldots x_m) \\ &\vdots \\ y_n &= f_n(x_1, \ldots x_m) \end{aligned} \qquad \text{Eq. 4.1.1}$$

The elements in the box are any combination of logic gates connected in the customary way as discussed previously. In particular no outputs are allowed to be "fed back" to form inputs of other logic gates.

In this chapter we shall extend the design procedures to include networks which respond not only to present inputs but also the cumulative effect of all past inputs. Figure 4.1.2 shows a generalized sequential logic network.

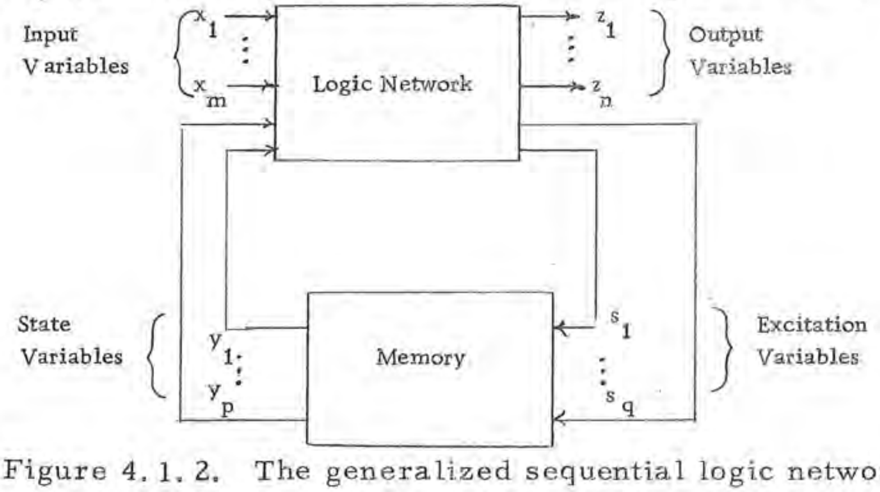


Figure 4.1.2. The generalized sequential logic network.

The output is now a function not only of the input variables but also of the state variables of the systems (or the state of the system). The complete behavior of the device can be described by the following equations:

$$\begin{aligned} z_1 &= F_1 (x_1, \ldots x_m, y_1, \ldots y_p) \\ &\vdots \\ z_n &= F_n (x_1, \ldots x_m, y_1, \ldots y_p) \end{aligned} \qquad \text{Eq. 4.1.2}$$

$$\begin{aligned} s_1 &= G_1(x_1, \ldots x_m, y_1, \ldots y_p) \\ &\vdots \\ S_q &= G_q (x_1, \ldots x_m, y_1, \ldots y_p) \end{aligned} \qquad \text{Eq. 4.1.3}$$

At this point it is convenient to define a new symbol related to the state variable y. It can be seen from Figure 4.1.2 that there is a cyclical flow of information through the logic network and memory. As a result of this flow the state variables change as a function of the excitation variables. The change in state variables is not immediate, however, since it is moderated by the memory. We circumvent this problem by defining a variable y* which indicates the next state of state variable y. Equation 4.1.4 indicates the relationship between the excitation variables and the next state.

$$\begin{aligned} y_1^* &= H_1 (s_1, s_2, \ldots s_q) \\ &\vdots \\ y_p^* &= H_p (s_1, s_2, \ldots s_q) \end{aligned} \qquad \text{Eq. 4.1.4}$$

Substituting the values of the excitation variables in Eq. 4.1.3 into 4.1.4 gives

$$\begin{aligned} y_1^* &= I_1 (x_1, x_2, \ldots x_m, y_1, \ldots y_p) \\ &\vdots \\ y_p^* &= I_p (x_1, x_2, \ldots x_m, y_1, \ldots y_p) \end{aligned} \qquad \text{Eq. 4.1.5}$$

The exact representation of the relationships shown in Eq. 4.1.4 and 4.1.5 is dependent on the block in Figure 4.1.2 labeled Memory. A digital memory device is defined to be any device capable of perpetuating an output after the stimulus eliciting the output has been removed. For design purposes we shall deal almost exclusively with various types of flip flops to perform the memory function. Other forms of memory such as magnetic cores present no challenge to the theory but needlessly complicate the design implementation and will be avoided.

The flip flop memory elements we shall study are of two types: clocked and asynchronous. In a clocked flip flop all changes of state are initiated by an external synchronizing signal called the system clock. The use of a synchronizing clock greatly simplifies the design of sequential circuits. Clocked circuits are used almost exclusively in digital computers and most special purpose sequential digital systems. It is the clock which initiates all state changes in an orderly and paced fashion.

Asynchronous logic design does not use a clock signal. State transitions are initiated instead only by the excitation variables. Changes in state variables may cause changes in excitation variables which may cause additional changes in state variables etc.

Asynchronous networks generally perform faster than their clocked counterparts but introduce design problems associated with oscillation, differences in propagation times of gates, and trouble shooting.

4.2 The State Diagram

For both analysis and design purposes the state diagram provides a convenient means of specifying the sequential behavior of a network. The states on a state diagram is indicated by a series of circles; one for each possible state of the system. The state may be specified either by listing the value of the state variables or by assigning an alphabetic symbol.

The effect of system inputs ($x_1, x_2 \ldots$) is indicated by arrows originating from the present state and terminating on the next state. In some cases the present state and next state may be identical. Each arrow is labeled with a binary representation of the values of the input variables, a slash (/), followed by the binary representation of the output variables.

Two general examples will be shown. The first is a forward/backward binary counter; the second is a network to determine the cessation of rotation of a mechanical device.

Example 4.2.1

Develop the state diagram for a clocked sequential network which will count forward in the binary sequence if $x = 1$. The circuit is to count backward if $x = 0$. The count is to contain only 2 bits and follow the sequence $z_1 z_2$ = 00, 01, 10, 11, 00, 01 etc.

Solution:

Figure 4.2.1 shows a possible state diagram for the device described.

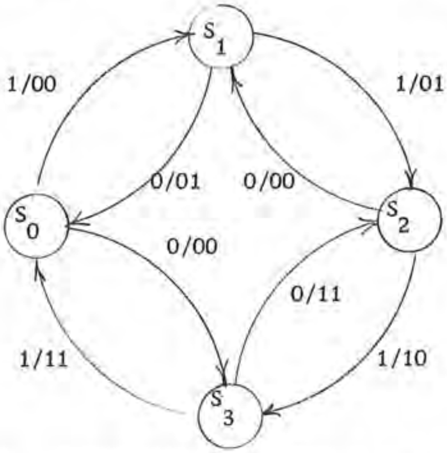


Figure 4.2.1. State diagram for a forward/backward binary counter.

In Figure 4.2.1 the state variables have not been assigned. To distinguish between the four states shown, a minimum of two state variables are required.

N = number of states

n = number of state variables

$$2^n \geq N \qquad \text{Eq. 4.2.1}$$

For a complete specification of the device we may assign the four combinations of our two state variables in any manner as long as each state has a unique state variable assignment.

Figure 4.2.2 shows a possible state variable assignment indicating the arbitrary nature of the assignment.

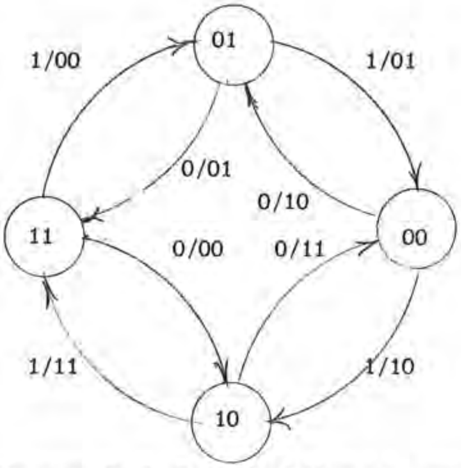


Figure 4.2.2. Arbitrary assignment of state variables.

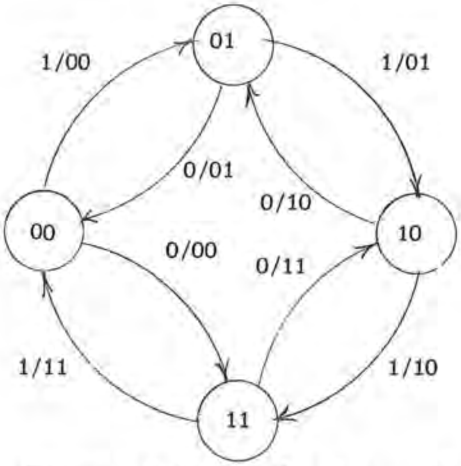


Figure 4.2.3. Assignment of state variables such that state variables are identical to output variables.

In Figure 4.2.3 the state variables have been assigned in such a way that they have the same value as the output variable. This is often possible in counters. When this is done the overall design may be simplified considerably. This will be considered further in Chapter 5.

Example 4.2.2

A rotating device is equipped with an electro-optical sensor system as shown in Figure 4.2.4 which provides a logic 0 when the sensor is "looking" at the black portion of the rotating disc and a logic 1 when "looking" at the white portion. A clocked sequential logic device is needed to determine if the disc has stopped rotation. The disc is defined to be stopped if three consecutive 1's or 0's are received from the sensor.

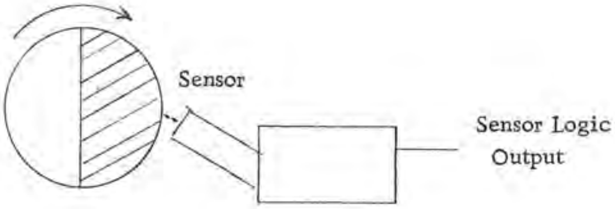


Figure 4. 2. 4. Rotation sensor.

Solution:

An initial state S_0 is selected. Three consecutive 1's or 0's direct us to a state which provides a failure indication. Recovery is automatic when motion again starts.

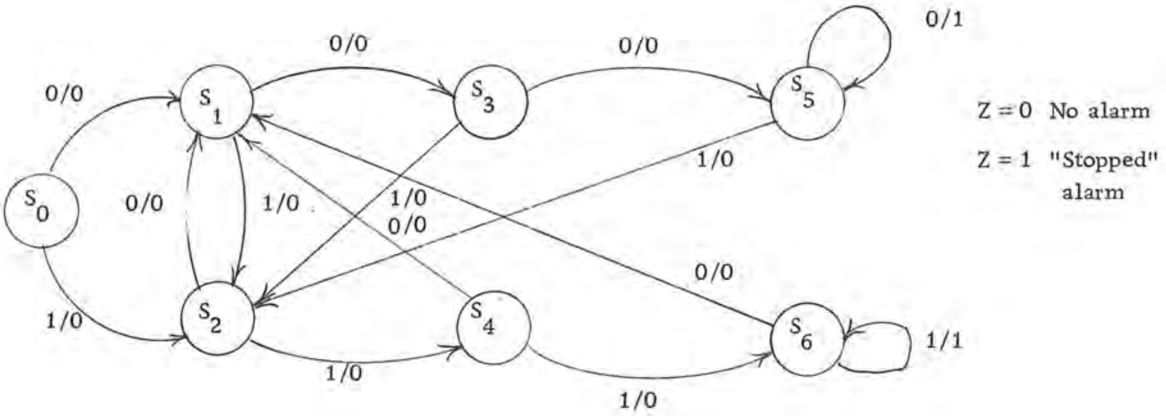


Figure 4. 2. 5. State diagram of rotation sensor.

For the device in the example three state variables would be required for implementation.

4.3 The Latch

The flip flop will comprise the basic memory building block for the sequential circuits which will be designed. Flip flops come in several different configurations. The skilled designer should be capable of selecting the best type for a particular application.

The latch or Set-Reset flip flop is the most basic unit of logic memory. It can be easily constructed from NAND gates or NOR gates. It is commonly used in asynchronous applications. It also is a building block of the clocked flip flops to be discussed later.

The latch is easily implemented as one of the circuits shown in Figure 4.3.1. The interter input on the NAND representation is included to allow both flip flops to have changes initiated by a logic 1.

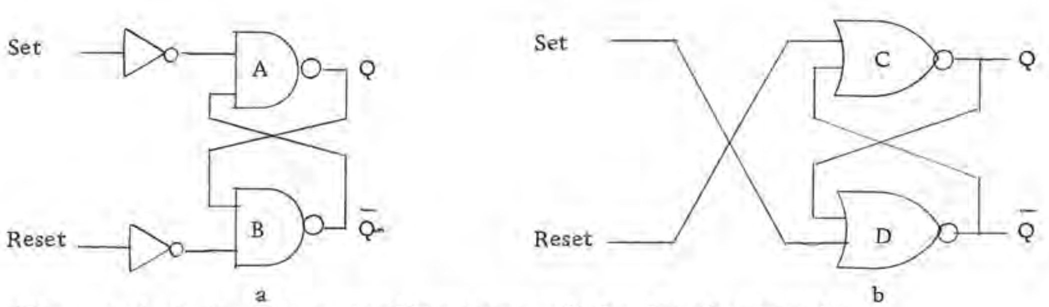


Figure 4.3.1. Two realizations of the latch circuit.

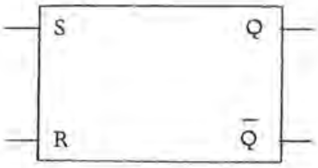


Figure 4.3.2. Symbol for the latch.

Consider the NAND gate R-S flip flop of Figure 4.3.1(a). Assume the R-S inputs take on the condition R = 0, S = 1. Gate A has a 1 output because of the logic 0 input from the inverted set input. Gate B has an output of logic 0 since both of its inputs are logic 1. One input is the inverted reset input and the other is the logic 1 input from the output of gate A.

The flip flop is said to be in the 1-state since Q = 1. If S is now changed to 0 giving R = 0, S = 0, gate A remains at logic 1 because of the logic 0 input it receives from Gate B. Gate B remains at logic 0 because both of its input are logic 1. With both R = 0 and S = 0 the flip flop "remembers" that sometime in the past the condition S = 1, R = 0 existed and retains Q = 1 at its output.

If the S = 1, R = 0 condition is repeated, there will be no change in the outputs of the flip flop. In this case Gate A now has two inputs which are logic 0; only one is necessary to provide the logic 1 output. Gate B retains its two inputs of logic 1 and remains at logic 0.

The flip flop of Figure 4.3.1(a) is completely symmetric in its logic construction with respect to the reset and set inputs. The inputs S = 0, R = 1 force the flip flop to the condition where the output of Gate A is 0 and the output of Gate B is 1. This is called the 0 state of the flip flop.

The circuit of Figure 4.3.1(b) is also an R-S flip flop. The condition R = 1, S = 0 causes gate D to become logic 0. The R = 0 and 0 output from Gate D cause Gate C to become logic 1. This output condition will remain the same if R = 0, S = 0. Again the circuit has memory properties and behaves identically to its NAND gate companion for conditions R = 0, S = 0; R = 1, S = 0; R = 0, S = 1.

The condition R = 1, S = 1 causes both outputs of the NAND gate flip flop to become logic 1: it causes both outputs of the NOR gate flip flop to become logic 0. The condition R = 1, S = 1 implies a certain ambiguity of purpose and is normally considered an invalid input condition.

The response of a flip flop to its input condition (R, S) can only be described if the initial state of the flip flop is known. Figure 4.3.3 is a state diagram describing the behavior of the latch with inputs x_1x_2 = SR.

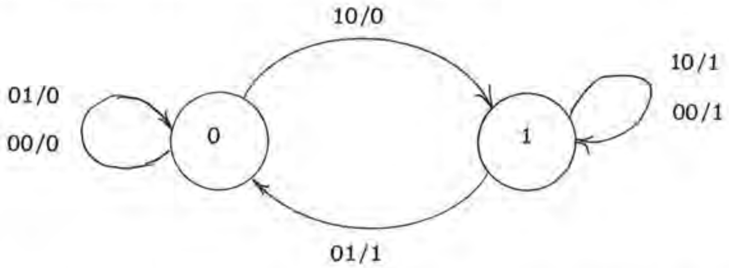


Figure 4.3.3. State diagram description of the latch.

Table 4.3.1 presents a tabular description of the latch. It appears in much the same form as a truth table. The next state (Q^*)is presented as a function of the inputs R, S and present state Q.

Present State Q	Input R	Input S	Next State Q^*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	Input Invalid
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	Input Invalid

Table 4.3.1. Characteristics of the latch.

For design purposes it is useful to present the inputs required to produce a desired state transition. In Table 4.3.2 the X is used as a "don't care" condition.

Present State Q	Next State Q^*	Inputs R	Inputs S
0	0	x	0
0	1	0	1
1	0	1	0
1	1	0	x

Table 4.3.2. Design characteristics of the latch.

The following examples suggest possible applications of the latch. The design procedure is largely heuristic. Chapter 5 will present a more detailed design procedure and outline possible problem areas.

Example 4.3.1

As part of the safety system of a manned spacecraft, several system components are continuously monitored to detect failure conditions. A failure condition is indicated by a logic 1 signal appearing at the D terminal which is generated whenever the component deviates from its prescribed behavior. Sometimes the failure condition will exist for only a brief instant of time and then return to its normal range. Design a system to continuously monitor the failure detect circuit and "remember" if a failure condition appears.

Solution:

The latch is an ideal component for such a situation. A possible design is shown in Figure 4.3.4. The latch is initially put in the 0 state by momentarily entering a logic 1 on the reset terminal. A failure causing a logic 1 on the set line will put the flip flop in the 1 state and actuate the indicator. The indicator will remain on until the flip flop is again placed in the 0 condition.

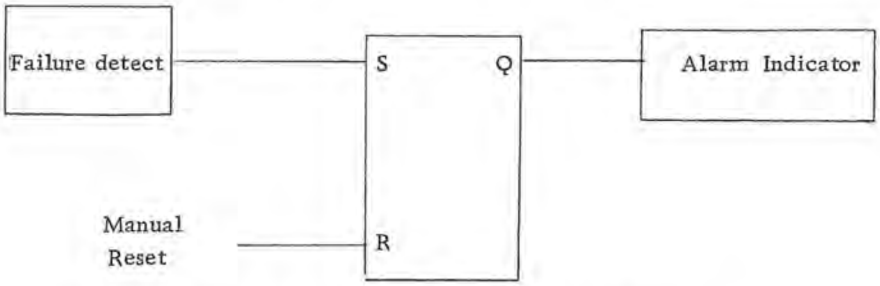


Figure 4.3.4. Failure detect system.

4.4 Clocked Flip Flops

Two types of flip flops are commonly used in the design of clocked sequential circuits; these are the JK flip flop and the D type. Two variations of clocking also exist for each type of flip flop.

Type D

The type D flip flop has a single input terminal labeled D. Its terminal behavior is described in Table 4.4.1.

Present State Q	Input D	Next State Q^*
0	0	0
0	1	1
1	0	0
1	1	1

Table 4.4.1. Terminal characteristics of the type D flip flop.

The D flip flop also has one or more terminals which can cause transitions without the aid of a system clock. These are commonly called preset and clear and are normally used for entering initial conditions into the flip flops.

JK Flip Flop

The JK flip flop uses two input variables to specify the next state condition. The terminal beh.ıvior is described in Table 4.4.2.

Present State Q	Input J	K	Next State Q^*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Table 4.4.2. Characteristics of JK flip flop.

For design purposes the JK inputs necessary to produce specified state changes are shown in Table 4.4.3.

Present State Q	Next State Q^*	Inputs J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Table 4.4.3. Design characteristics of the JK flip flop.

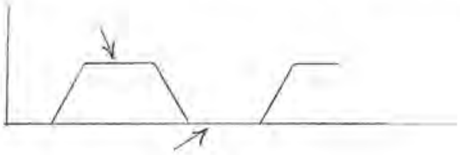
The JK flip flop also may have additional preset and clear terminals for entering initial conditions.

The Clock

The clock signal is used in sequential design to initiate all changes of state in flip flops. Since the state variables effect the excitation variable of the flip flops it is important to isolate the present state from the next state. If this is not done the possibility for uncontrolled oscillation exists.

Two techniques are commonly used, the "master-slave" and the "edge trigger". In the master-slave formulation two internal latches are used in "bucket brigade" style to pass information from the input "master" section to the output "slave" section. The sequential operation is shown in Figure 4.4.1.

1. Isolate slave from master
2. Enter information from inputs to the master.



3. Disable inputs
4. Transfer information from slave to master (new output)

Figure 4.4.1. Transfer of information in the Master-Slave flip flop.

In the master-slave flip flops all state transitions occur on the 1 to 0 transition of the clock pulse. This is helpful in the heuristic design of counting circuits.

In the edge triggered flip flop, input information is transferred to the output on the leading edge of the clock pulse. The transfer occurs only during the small instant of time that the clock amplitude is between two thresholds.

The behavior is shown in Figure 4.4.2.

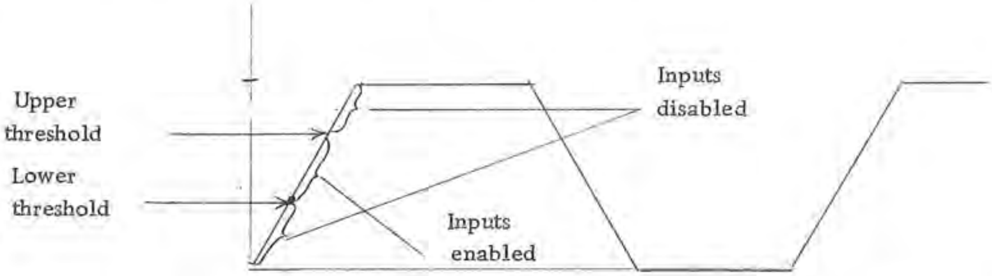


Figure 4.4.2. Transfer of information in the edge triggered flip flop.

In the edge triggered flip flop all state transition occur on the 0 to 1 transition of the clock pulse.

4.5 Binary Counters

Either the Type D or J-K flip flop may be connected into binary counters using heuristic design techniques. These counters may be either of the ripple type in which changes in the state of flip flops start at the least significant bit and "ripple" sequentially toward the most significant bit or a synchronous type in which all bit locations change simultaneously.

Figure 4.5.1 and 4.5.2 show ripple counters using the two types of flip flops. In both cases the input signal (the pulses to be counted) is introduced at the clock terminal of the least significant bit location. The least significant bit changes state with every pulse. A change of state of each bit location is initiated by a 1 to 0 change of its next least significant neighbor. The 1 to 0 changes are easily utilized by connecting the output of one bit location to the clock input of its neighbor.

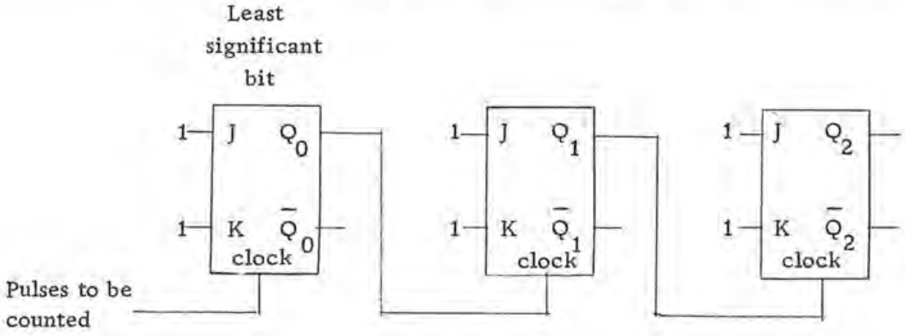


Figure 4.5.1. Typical ripple counter using JK master-slave flip flops.

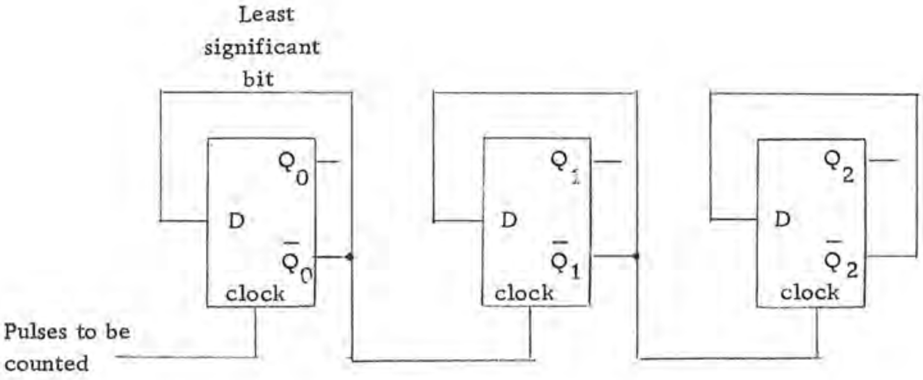


Figure 4.5.2. Ripple counter using type D edge triggered flip flops.

Binary counters of any arbitrary length may be constructed by repetitions of the basic circuitry. The ripple effect may cause some problems. Consider the counter with $Q_2Q_1Q_0 = 011$. The next pulse should lead to $Q_2Q_1Q_0 = 100$. Since changes must start with the least significant bit a series of transitional conditions lasting a short time will exist between the 011 and 100 states. These transitions are indicated in Table 4.5.1.

	Q_0	Q_1	Q_2
Initial state	1	1	0
Least significant bit changes	0	1	0
Q_1 bit changes	0	0	0
Q_2 bit changes final state	0	0	1

Table 4.5.1. Intermediate states in a ripple counter

Care must be used to be certain that circuitry connected to the counter does not respond falsely to the intermediate conditions shown in Table 4.5.1. The time required for ripple propagation in long counters may make them unsuitable for some operations.

The speed of the binary counter may be considerably increased by causing all transitions to occur simultaneously rather than in a serial mode. Figure 4.5.3 shows a three stage synchronous counter. The input logic to each of the JK inputs is easily verified by noting that an input affects a counter section only when all less significant counter sections contain logic 1's. All state changes occur simultaneously. The disadvantage of the parallel transition counter is the extra logic gates required.

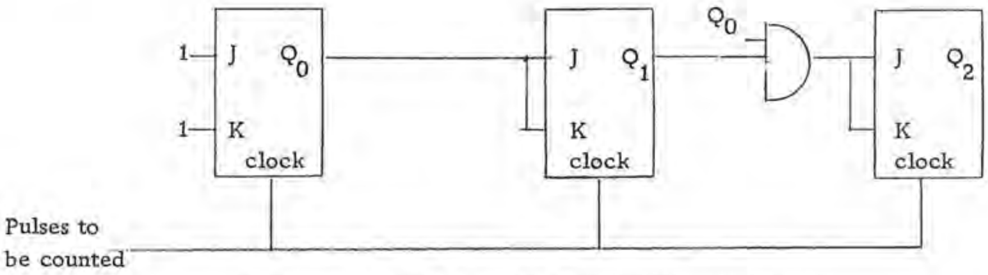


Figure 4.5.3. Synchronous counter.

4.6 Shift Registers

The shift register finds many applications in the construction of arithmetic circuits and encoding networks for communication. It is conveniently constructed from J-K flip flops or Type D Flip Flops as shown in Figure 4.6.1 and 4.6.2. With each clock pulse information initially placed in the register is shifted one location to the right. Information shifted into the first stage is determined by its input conditions.

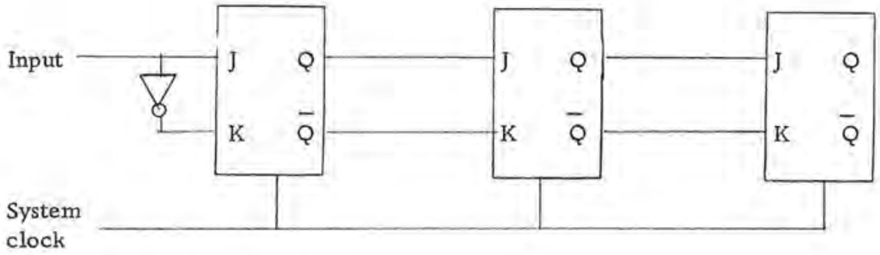


Figure 4.6.1. Shift register constructed with JK flip flops.

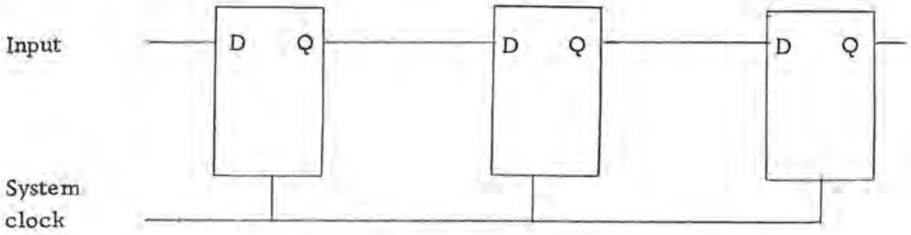


Figure 4.6.2. Shift register constructed with type D flip flops.

Recirculating Shift Register

Figure 4.6.3 shows a variation of a shift register in which the output of the last stage is used as an input to the first stage. Information is continually recirculated. A single logic 1 circulated through several register sections might conveniently be used as a means for providing sequential logic signals for timing purposes.

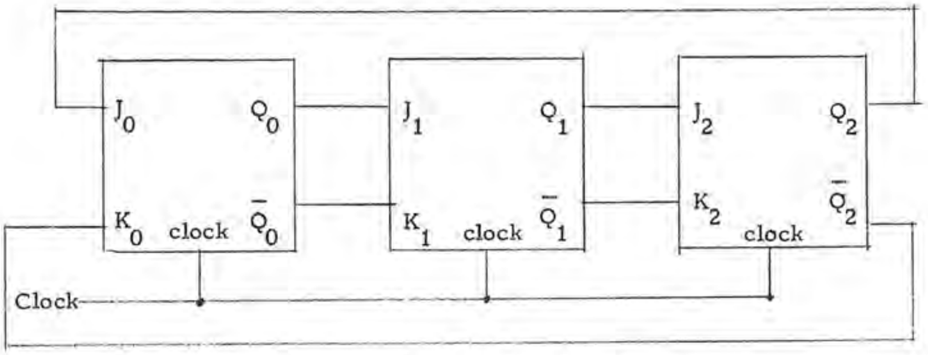


Figure 4.6.3. Recirculating shift register.

An array of recirculating shift registers in which one register is used to hold one bit location of a computer word is sometimes used as a sequential access memory system.

Shift Register Generator

Shift register generators are formed by generating a logic function of the various flip flop outputs and using it as an input to the shift register. The general case is shown in Figure 4.6.4.

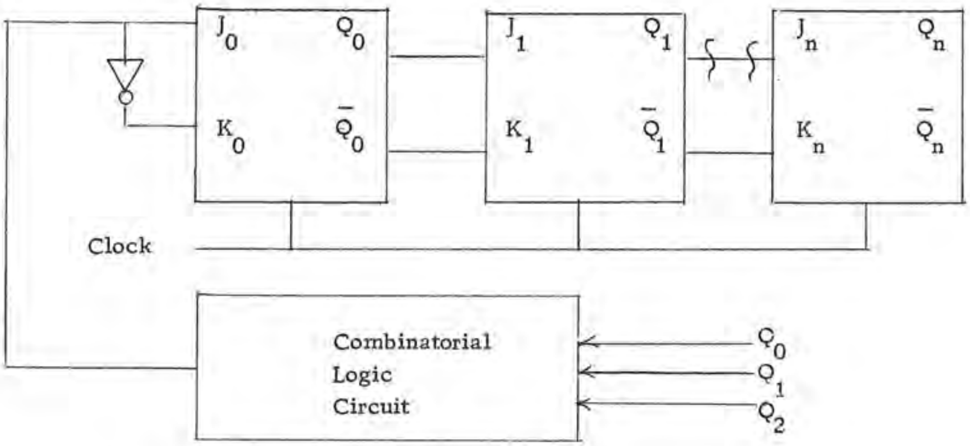


Figure 4.6.4. The shift register generator.

If the combinational logic network is correctly chosen, the set of outputs $Q_0Q_1 \ldots Q_n$ will assume 2^n-1 unique states before a repetition occurs (n = number of shift register elements). Table 4.6.1 indicates combinational logic functions which will produce maximum length sequences.

n	Function	n	Function
1	Q_0	11	$Q_8 \oplus Q_{10}$
2	$Q_0 \oplus Q_1$	12	$Q_1 \oplus Q_9 \oplus Q_{10} \oplus Q_{11}$
3	$Q_1 \oplus Q_2$	13	$Q_0 \oplus Q_{10} \oplus Q_{11} \oplus Q_{12}$
4	$Q_2 \oplus Q_3$	14	$Q_1 \oplus Q_{11} \oplus Q_{12} \oplus Q_{13}$
5	$Q_2 \oplus Q_4$	15	$Q_{13} \oplus Q_{14}$
6	$Q_4 \oplus Q_5$	16	$Q_{10} \oplus Q_{12} \oplus Q_{13} \oplus Q_{15}$
7	$Q_5 \oplus Q_6$	17	$Q_{13} \oplus Q_{17}$
8	$Q_1 \oplus Q_2 \oplus Q_3 \oplus Q_7$	18	$Q_{10} \oplus Q_{17}$
9	$Q_4 \oplus Q_8$	19	$Q_{13} \oplus Q_{16} \oplus Q_{17} \oplus Q_{18}$
10	$Q_6 \oplus Q_9$	20	$Q_{16} \oplus Q_{19}$

Table 4.6.1. Logic functions for generating maximum length sequences.

For all cases in Table 4.6.1 any initial condition except 00...0 may be used. For example n = 3, with initial condition 100, will produce the following sequence:

Initial condition		100	
Clock pulse	1	010	
	2	101	
	3	110	
	4	111	
	5	011	
	6	001	
	7	100	(repeat of initial condition)

The shift register has many of the same features as a binary counter. Both assume approximately 2^n unique states and then repeat the same sequence. The binary counter has the advantage of incrementing one count in a binary counting sequence. The shift register generator has the advantage of being completely synchronous in its state changes (no rippling) while requiring relatively little extra logic.

Some uses for shift register generators include the following:

1. Sequencing of operations involving sequential timing signals. The output states of the SRG are decoded to provide a series of sequencing logic signals.
2. The random appearing property of the bits is often used as the basis for a binary random noise generator. By the use of digital to analog conversion techniques an analog noise generator having easily specified characteristics may be constructed.

4.7 The Arithmetic Register

Registers for temporary storage of operands and instructions play an important role in the operation of digital computers. Figure 4.7.1 shows a typical register configuration for the Arithmetic operations of a digital computer.

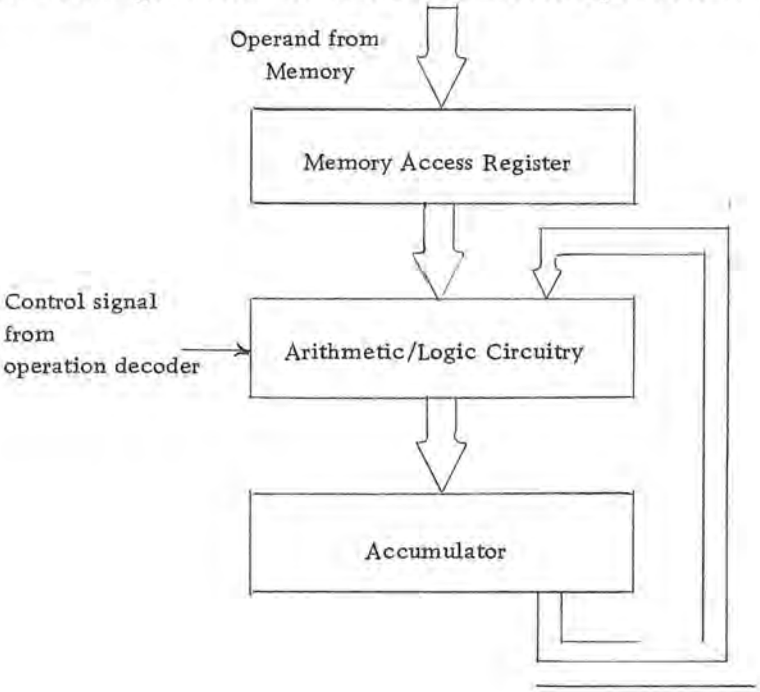

Operand from Memory
Memory Access Register
Control signal from operation decoder
Arithmetic/Logic Circuitry
Accumulator

Figure 4.7.1. Typical register configuration for arithmetic in a digital computer.

One of the operands is assumed to be the accumulator. The second is retrieved from bulk memory and stored in the Memory Access Register. The contents of the two registers are combined in the Arithmetic/Logic circuitry as specified by the control signal from the operation decoder. The flow of information between all registers is controlled by clock signals generated in the control unit of the computer.

Figure 4.7.2 is an example of computer circuitry to perform the following operations.

Add	- Add contents of accumulator and memory access register and store sum in accumulator.
Shift	- Shift the contents of the accumulator left one position.

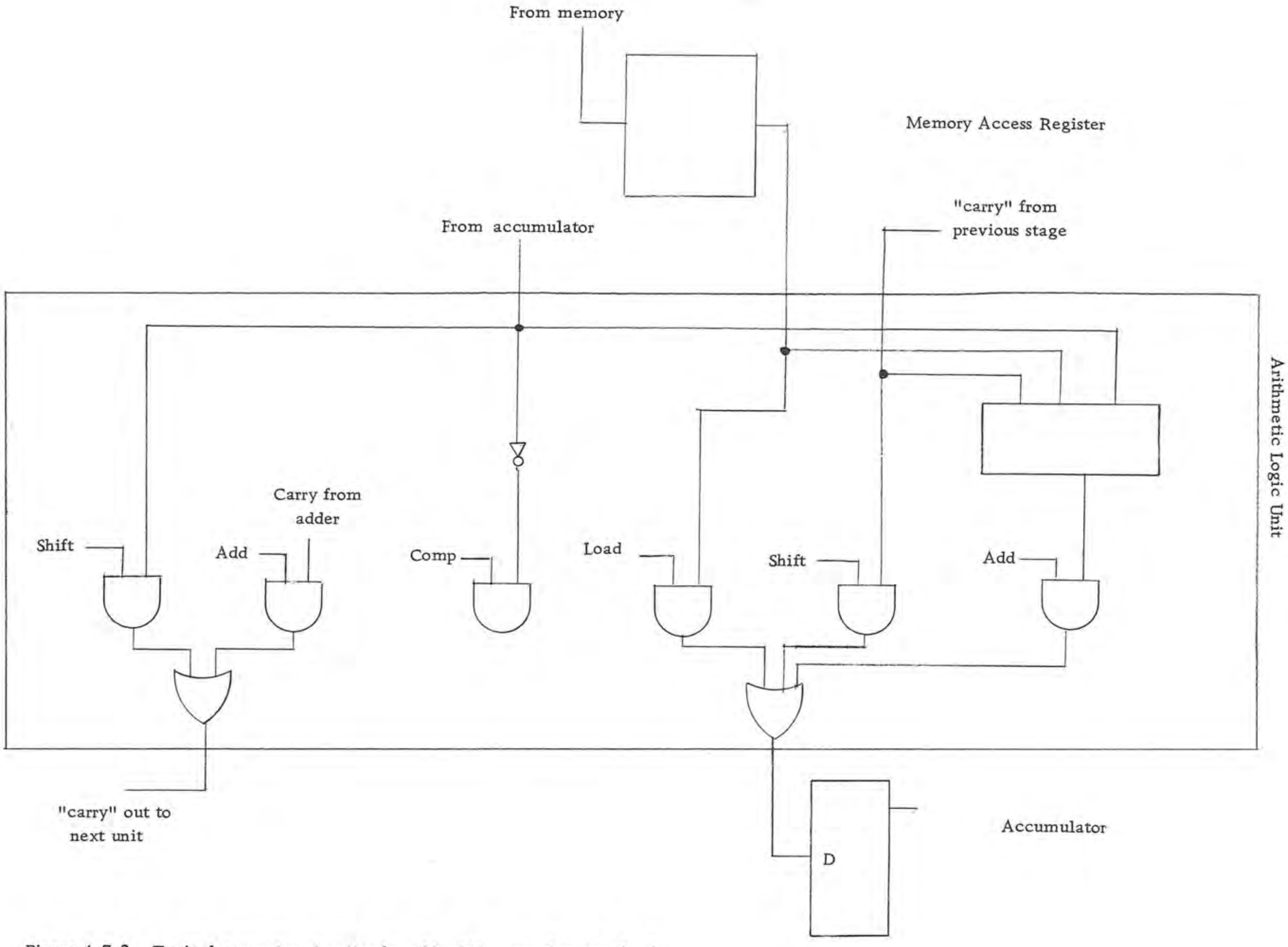


Figure 4.7.2. Typical computer circuitry for add, shift, complement, load.

Complement - Complement all bits of the accumulator.

Load - Transfer the contents of memory access register to the accumulator.

Although a complete operational description is not appropriate at this point, it is informative to note the dependency of even large computer systems on relatively simple registers, counters, and combinational logic.

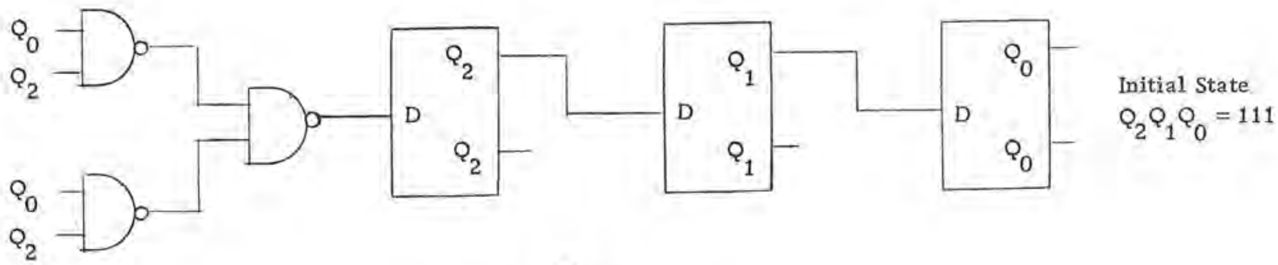
4.8 Notes - References - Problems

The material in this chapter is covered in a wide variety of standard texts in the field. Of the texts listed below, the treatment in Chapter VII of Booth (1) is particularly relevant. Material relating to detailed flip flop characteristics is best obtained from manufacturer's specification sheets.

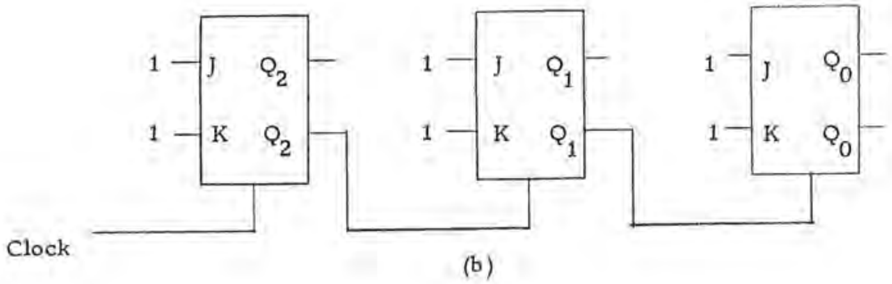
1. Booth, T. L., (1971), "Digital Networks and Computer Systems", John Wiley, New York.
2. Dietmeyer, D. L., (1971), "Logic Design of Digital Systems", Allyn and Baron, Boston.
3. Krieger, M., (1967), "Basic Switching Circuit Theory", MacMillan, New York.

EXERCISES

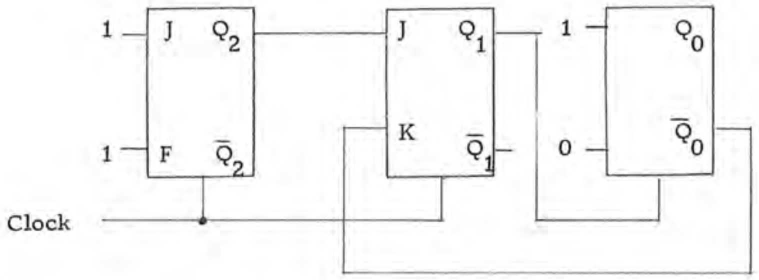
1. Derive a state diagram for a network which will have a 0 output until four consecutive logic 1's have been sequentially received. The output is to change to 1 and remain 1 thereafter.
2. Repeat problem 1 with the condition that the logic 1's need not be consecutive.
3. Determine the sequence of states for the following networks of flip flops.



(a)



(b)



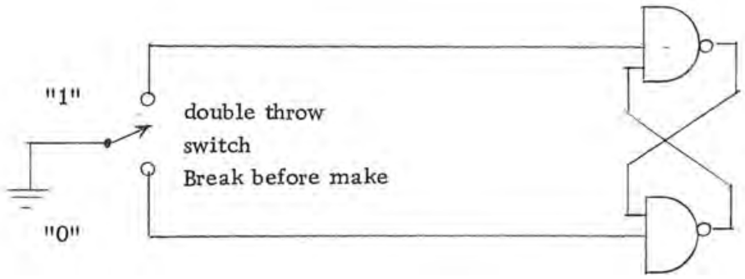
c

Answer

Q_2	Q_1	Q_0
0	0	0
1	0	0
0	1	0
1	0	1
0	1	1
1	1	1
0	1	1
1	1	1

etc.

4. Discuss the behavior of the following circuit as a contact bounce eliminator.



5 Design of Clocked Sequential Circuits

5.1 Clocked Sequential Logic

This chapter is concerned primarily with the design of sequential networks using clocked flip flops for memory elements. Both the type D flip flop and the JK flip flop will be considered. A synchronizing clock signal will be implicit in all designs in this chapter and will generally not be included on circuit designs. At this point it is not necessary to distinguish between "edge triggering" and "master-slave" operations providing both types are not intermixed in the same design.

5.2 The Design Procedure

Design of clocked sequential networks can proceed in a very straight forward manner using a minimum of intuitive and "heuristic" methods. For any design the following suggested steps provide a partitioning of the design problem into an orderly sequence of steps.

5.2.1 Prepare a State Diagram

In this operation the complete description of the desired network should be put in a well organized form. The state diagram is an ideally suited form for this prupose. Special care may be warranted to assure that all states used are independent. The use of two or more states which are equivalent will not affect the performance of the circuit but may needlessly complicate the design of the network and result in a higher "cost" realization in terms of component requirements.

The "equivalence" problem and an algorithmic approach to its solution will be discussed in section 5.6.

5.2.2 Assign State Variables

The number of states in the state diagram determines the minimum number of state variables sufficient to design the network. Each state variable requires one storage element or flip flop. Use of the minimum number of state variables will result in the use of the minimum number of flip flops. The assignment of state variables to the states may be done in an arbitrary manner providing all states have a unique assignment.

Although the state assignment is arbitrary, it has an effect on the overall "cost" of the network. A few suggestions will be given which will be sufficient for a good start. For some designs it may be essential to examine alternative state variable assignments and their effect on system cost. The following rules suggest a reasonable way of assigning state variables.

1. If your network requires an initial state, assign this state the assignment 00...0. Besides agreeing with intuition it allows the use of the asynchronous "clear" terminal on the flip flops to initialize operation. (Some flip flops do not have a pre-set capability. If the pre-set terminal

is available, any state can be easily initialized.)

2. Pick the state which has the most other states adjacent to it (transitions to on from other states). Try to assign the adjacent state state variables which differ in only one position.
3. In the case of counters or timing sequence generators, it is often desirable to have one or more of the state variables be identical to the system output. If this is the case one or more of the state variables can also function as output variables.

The above are only suggestions and may be violated at will without destroying the basic integrity of the network.

5.2.3 State Transition Specification

Using the results of 5.2.1 and 5.2.2 specify the next state as a function of the present state and present input. This may be conveniently done in tabular form (the transition table) or in the form of a K map. In most cases the assignment of state variables will not exhaust the total number of unique assignments possible. The next state designation for unused state assignments may be treated in two ways.

1. They may be considered don't care conditions.
2. They may be assigned to a "recovery" state.

Method 1 may result in a condition in which the next state is the same as the current state. A malfunction or transient which might cause the circuit to enter such a state would result in the inability of the network to escape. A cycle of unused states is another possible situation which should be avoided.

By directing all unused states back into a properly chosen "used" state of the system, recovery from some types of malfunctions may be possible.

5.2.4 Output Specification

Specify the output as a function of the present state and input. This information is also contained in the state diagram and may be conveniently expressed in a tabular form or a K map.

5.2.5 Excitation Design

Using the design characteristics of the appropriate memory device specify the excitation variables (J, K on the JK flip flop or D on the type D flip flop). In the case of the D flip flop the excitation variable, D, is identical to the next state and the table prepared in 5.2.4 can be used.

5.2.6 Design Completion

Complete the design by using combinational logic design procedures to design the networks specified in 5.2.4 and 5.2.5.

The design procedure described will be further elaborated upon by example in the following sections.

5.3 Design of Sequence Generators/Counters

Counters and sequence generators are usually distinguished by a lack of input variable interaction. Transitions between states are initiated by the clock signal. In the case of an event counter the logic signal related to the occurrence of the event may form the clock for the network.

Example 1

Design a network to produce sequentially and cyclically the following output in 5 timing intervals. Type D flip flops are to be used.

		$2_1 2_2$
	1	00
	2	01
	3	11
	4	01
	5	00
repeat		00
		01
		11
		etc.

Solution:

The procedure described in the previous section will be used.

5.3.1 State Diagram

Figure 5.3.1 shows an acceptable state diagram for this network. Five states are needed. Since no inputs are needed, nothing appears before the "/" in Figure 5.3.1.

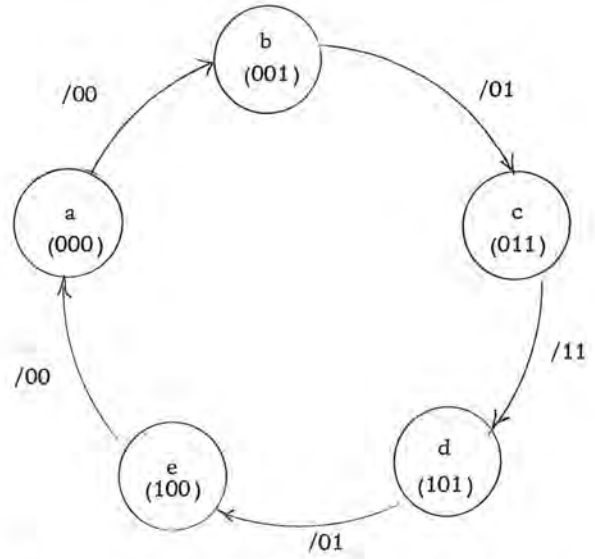


Figure 5.3.1. State diagram for sequence generator.

5.3.2 Assignment of State Variables

A minimum of three state variables are required. A possible assignment is shown within the parenthesis in the state diagram. State variables 2 and 3 are identical with the specified device output. This eliminates the need to design a separate output network. The first state variable is selected to assure uniqueness in the state designation.

5.3.3. State Transition Specification

The following K map specifies the next state as a function of the present state and input (in this example there is not input). Unused states are assigned don't care entries.

$Q_3 \backslash Q_1Q_2$	00	01	11	10
0	001	xxx	xxx	000
1	011	101	xxx	100

Figure 5.3.2. K map of next state as a function of present state.

5.3.4 Output Specification

In this example the state variables have been selected to be identical with the required output. No further design is necessary

$$z_1 = Q_2$$

$$z_2 = Q_3$$

5.3.5 Excitation Design

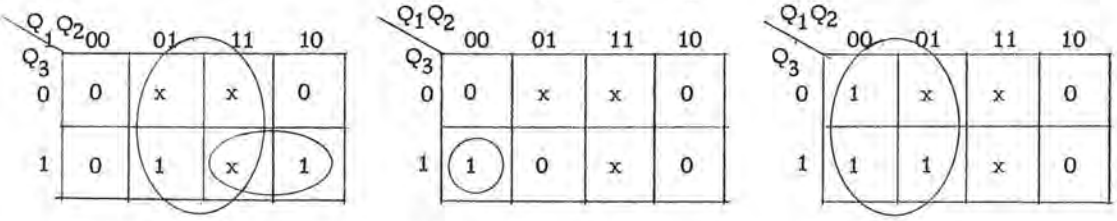
Because of the simplicity of the D flip flop the excitation variable design is relatively straight-forward for D_1, D_2, and D_3.

Present State Q	Next State Q^*	D
0	0	0
0	1	1
1	0	0
1	1	1

Table 5.3.1. Design characteristics of type D flip flop.

5.3.6 Design Completion

From Table 5.3.2 the logic for D_1, D_2, D_3 may be derived



Q_3 \ Q_1Q_2	00	01	11	10
0	0	x	x	0
1	0	1	x	1

$D_1 = Q_2 + Q_1Q_3$

Q_3 \ Q_1Q_2	00	01	11	10
0	0	x	x	0
1	1	0	x	0

$D_2 = \bar{Q}_1\bar{Q}_2Q_3$

Q_3 \ Q_1Q_2	00	01	11	10
0	1	x	x	0
1	1	1	x	0

$D_3 = \bar{Q}_1$

Table 5.3.2.

The complete design is shown in Figure 5.3.3.

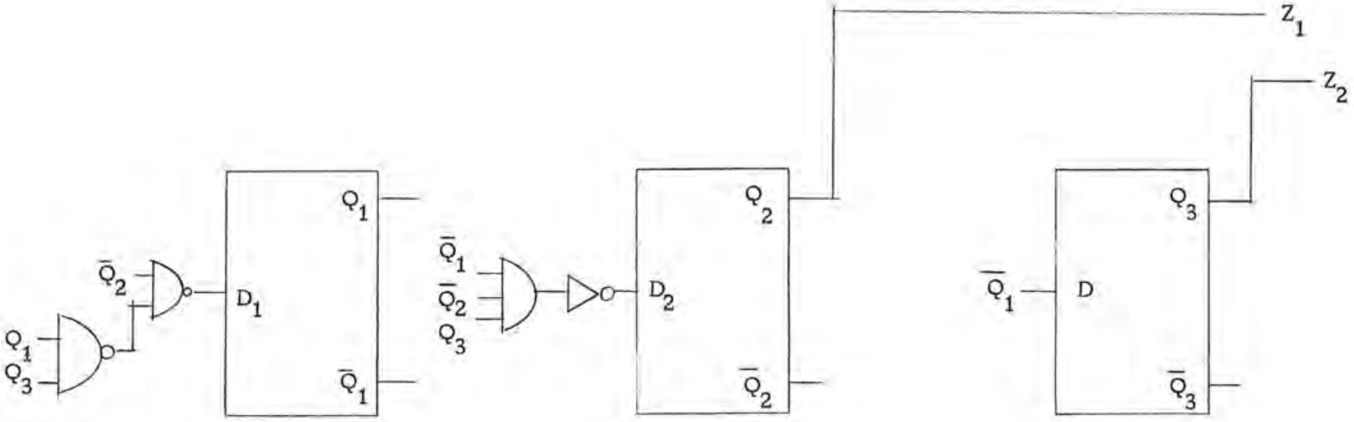


Figure 5.3.3. Completed design for counter.

Example 2. Repeat Example 1 using JK flip flops.

Solution:

Up through step 5.3.4, all results are identical to Example 1. The excitation design is the first departure from Example 1. The design characteristics of the JK flip flop are repeated in Table 5.3.3.

Present State Q	Next State Q^*	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

Table 5.3.3. Design characteristics of JK flip flop.

Using the state transition information of Figure 5.3.2 and the design charactersitics of Table 5.3.3 K maps for each of the excitation variables may be prepared. Figure 5.3.4 shows the completed design.

$Q_3 \backslash Q_1Q_2$	00	01	11	10
0	0	x	x	x
1	0	1	x	x

$J_1 = Q_2$

$Q_3 \backslash Q_1Q_2$	00	01	11	10
0	0	x	x	0
1	1	x	x	0

$J_1 = \bar{Q}_1 Q_3$

$Q_3 \backslash Q_1Q_2$	00	01	11	10
0	1	x	x	0
1	x	x	x	x

$J_3 = \bar{Q}_1$

$Q_3 \backslash Q_1Q_2$	00	01	11	10
0	x	x	x	1
1	x	x	x	0

$K_1 = \bar{Q}_3$

$Q_3 \backslash Q_1Q_2$	00	01	11	10
0	x	x	x	x
1	x	1	x	x

$K_2 = 1$

$Q_3 \backslash Q_1Q_2$	00	01	11	10
0	x	x	x	x
1	0	0	x	1

$K_3 = Q_1$

Table 5.3.4. Design minimization for excitation network.

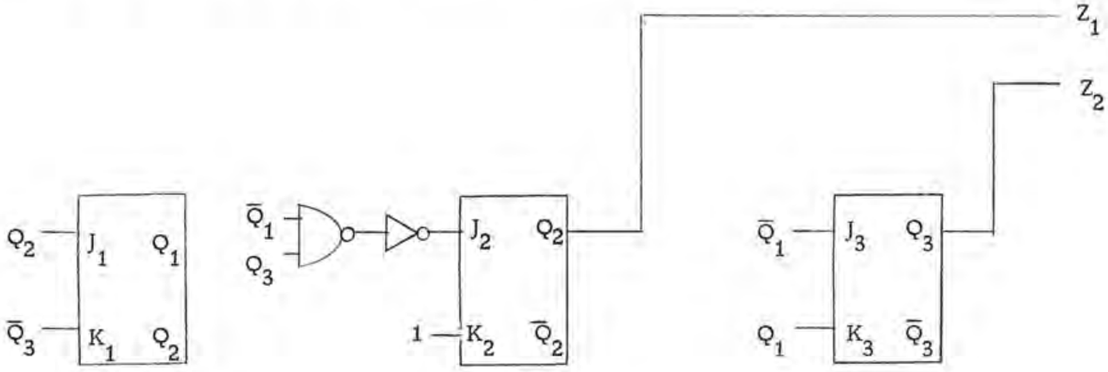


Figure 5.3.4. Completed counter design using JK flip flops.

5.4 Design of Input Sequence Detectors

Sequence detectors are usually specified to respond according to the time history of an input signal. They are useful for a variety of purposes involving arithmetic and monitoring functions. The design procedure is identical to the counting circuits of the previous section.

Example 3

Complete the design of the rotation sensor specified in Figure 4.7 of chapter 4. The device is to determine if rotation has stopped by sensing a sequence of three consecutive 1's or 0's.

Solution:

Three state variables are required. A possible state assignment is shown in Figure 5.4.1. Type D flip flops will be used.

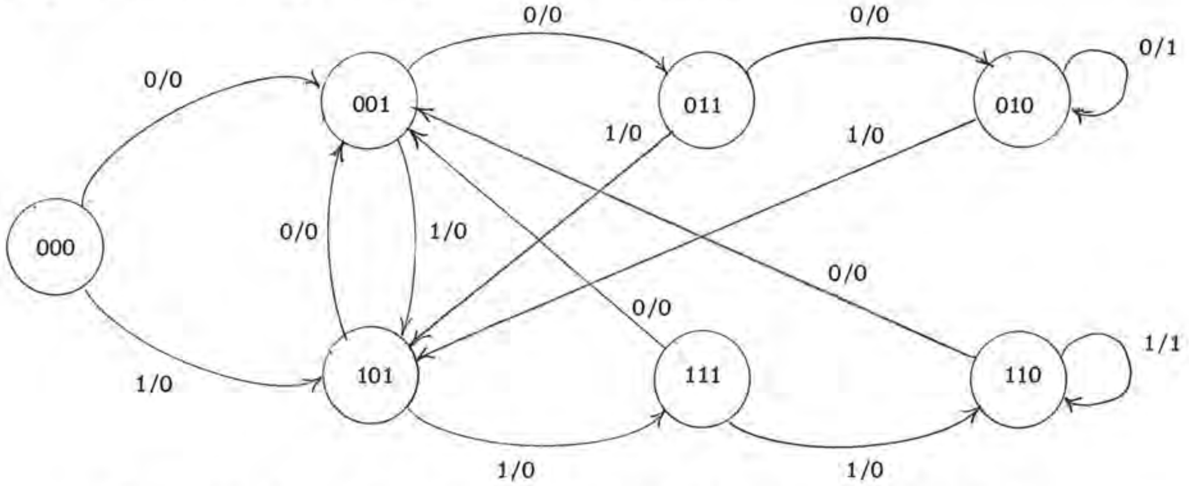
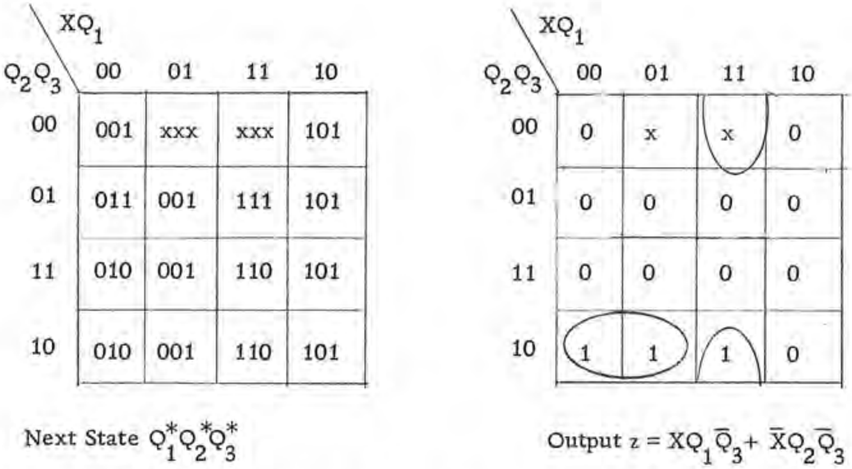


Figure 5.4.1. State assignment for the rotation sensor.

The required state transitions are shown in the Figure 5.4.2. Note that in the K map representation the input variable is now included.



Q_2Q_3 \ XQ_1	00	01	11	10
00	001	xxx	xxx	101
01	011	001	111	101
11	010	001	110	101
10	010	001	110	101

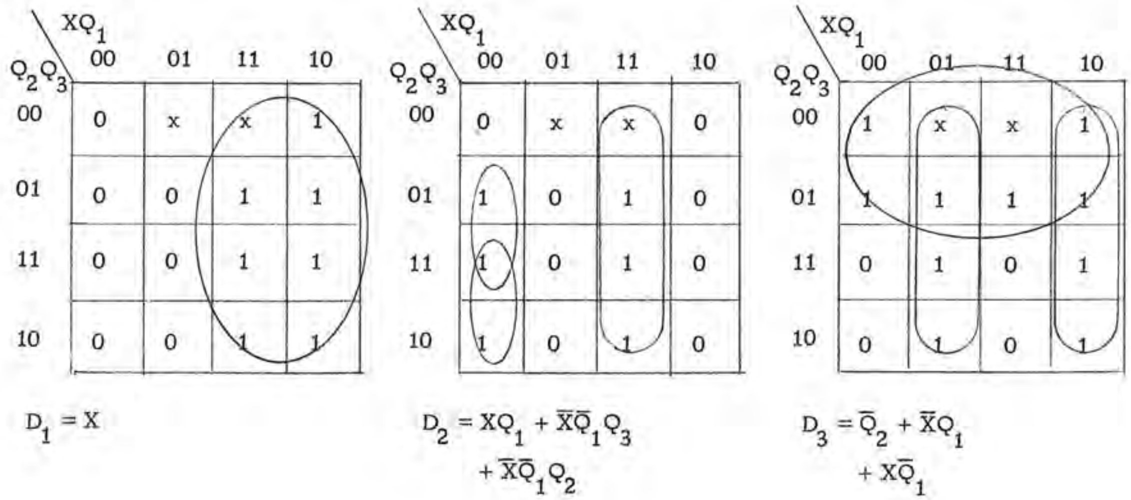
Next State $Q_1^*Q_2^*Q_3^*$

Q_2Q_3 \ XQ_1	00	01	11	10
00	0	x	x	0
01	0	0	0	0
11	0	0	0	0
10	1	1	1	0

Output $z = XQ_1\overline{Q}_3 + \overline{X}Q_2\overline{Q}_3$

Figure 5.4.2. Table of next state and outputs as a function of present state and input.

The individual K maps for D_1, D_2, D_3 may be prepared directly from Figure 5.4.2. The output logic may be minimized directly from the output K map of Figure 5.4.2.



Q_2Q_3 \ XQ_1	00	01	11	10
00	0	x	x	1
01	0	0	1	1
11	0	0	1	1
10	0	0	1	1

$D_1 = X$

Q_2Q_3 \ XQ_1	00	01	11	10
00	0	x	x	0
01	1	0	1	0
11	1	0	1	0
10	1	0	1	0

$D_2 = XQ_1 + \overline{X}\overline{Q}_1Q_3$
$+ \overline{X}\overline{Q}_1Q_2$

Q_2Q_3 \ XQ_1	00	01	11	10
00	1	x	x	1
01	1	1	1	1
11	0	1	0	1
10	0	1	0	1

$D_3 = \overline{Q}_2 + \overline{X}Q_1$
$+ X\overline{Q}_1$

Figure 5.4.3. K maps for excitation variable design.

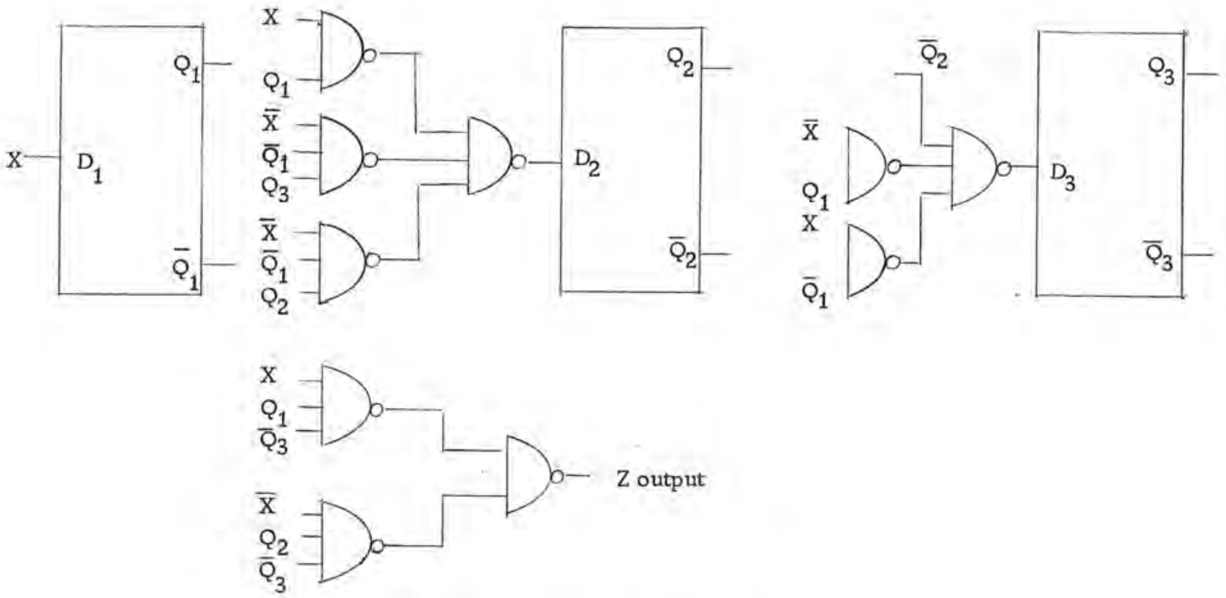


Figure 5.4.4. Complete design.

Example 4

Design a clocked sequential circuit for use as a combination lock. The initial input condition as obtained from push button switches is $x_1x_2 = 00$. The sequence $x_1x_2 = 01, 11, 10,$ is to open the lock. The lock is to remain open until the input $x_1x_2 = 00$ at which time it is to again lock.

Solution:

The clock frequency will be chosen high enough so that it will be impossible to change two input variables in the time interval between clock pulses.

The state diagram for the circuit is shown in Figure 5.4.5. Four states are needed; they are arbitrarily assigned two state variables.

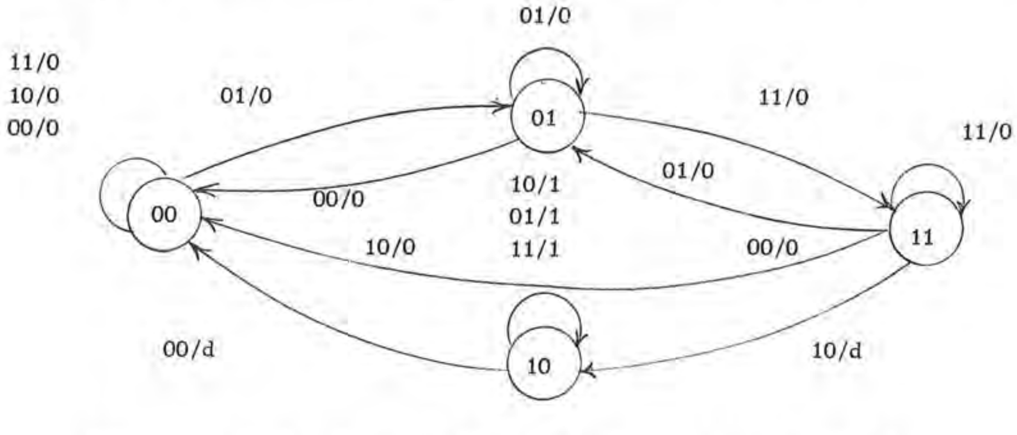


Figure 5.4.5. State diagram for combination lock.

The output variable has been assigned a value of Z = d for the transition into state 10 and the transition from state 10 to 00. In both cases the output is changing. It makes little difference if the value assumes its new assignment as rapidly as possible or remains at its prior value for an additional clock pulse. The don't care condition may allow some simplification of the logic. The following transition table represents an alternate method of specifying the required state changes, excitation logic, and output logic.

Present State		Input		Next State		Flip Flop		Logic		Output
Q_1	Q_2	x_1	x_2	Q_1^*	Q_2^*	J_1	K_1	J_2	K_2	Z
0	0	0	0	0	0	0	d	0	d	0
0	0	0	1	0	1	0	d	1	d	0
0	0	1	0	0	0	0	d	0	d	0
0	0	1	1	0	0	0	d	0	d	0
0	1	0	0	0	0	0	d	d	1	0
0	1	0	1	0	1	0	d	d	0	0
0	1	1	0	0	1	0	d	d	1	0
0	1	1	1	1	1	1	d	d	0	0
1	0	0	0	0	0	d	1	0	d	d
1	0	0	1	1	0	d	0	0	d	1
1	0	1	0	1	0	d	0	0	d	1
1	0	1	1	1	0	d	0	0	d	1
1	1	0	0	0	0	d	1	d	1	0
1	1	0	1	0	1	d	1	d	0	0
1	1	1	0	1	0	d	0	d	1	d
1	1	1	1	1	1	d	0	d	0	0

Table 5.4.1. State transition table for combination lock.

The logic circuits needed for J_1, K_1, J_2, K_2, Z may now be designed using a four variable K-map.

$x_1x_2 \backslash Q_1Q_2$	00	01	11	10
00	0	0	d	d
01	0	0	d	d
11	0	1	d	d
10	0	0	d	d

$J_1 = Q_2x_1x_2$

$x_1x_2 \backslash Q_1Q_2$	00	01	11	10
00	d	d	1	1
01	d	d	1	0
11	d	d	0	0
10	d	d	0	0

$K_1 = \overline{x}_1\overline{x}_2 + Q_2\overline{x}_1$

$x_1x_2 \backslash Q_1Q_2$	00	01	11	10
00	0	d	d	0
01	1	d	d	0
11	0	d	d	0
10	0	d	d	0

$J_2 = \overline{Q}_1\overline{x}_1x_2$

$x_1x_2 \backslash Q_1Q_2$	00	01	11	10
00	d	1	1	d
01	d	0	0	d
11	d	0	0	d
10	d	1	1	d

$K_2 = \overline{x}_2$

$x_1x_2 \backslash Q_1Q_2$	00	01	11	10
00	0	0	0	d
01	0	0	0	1
11	0	0	0	1
10	0	0	d	1

$Z = Q_1\overline{Q}_2$

Figure 5.4.6. K maps for design of combination lock.

The complete design for the combination lock is shown in Figure 5.4.7. Further minimization might be possible by trying a different assignment of state variable, using the product of sums representation, or trying to find common terms for a multiple output representation.

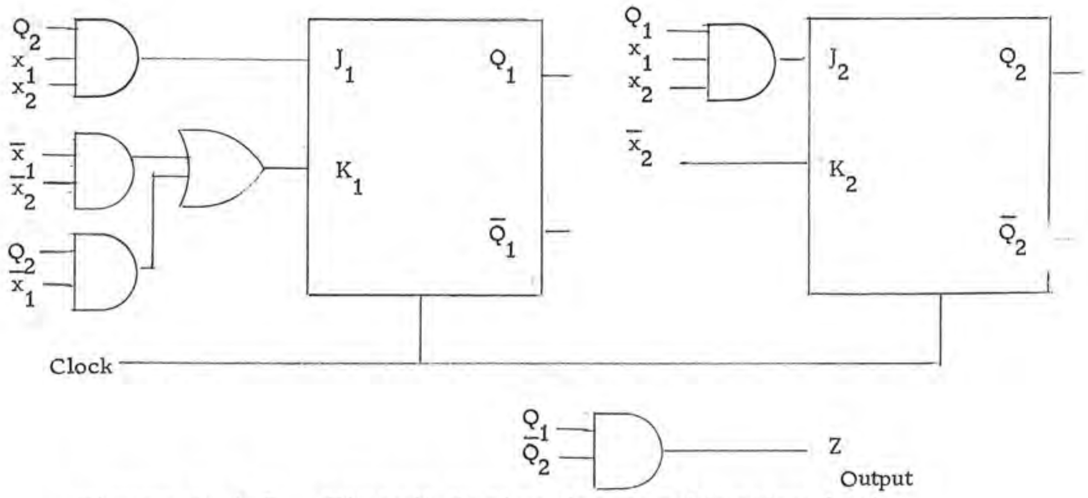


Figure 5.4.7. Circuit design for combination lock.

5.5 Minimization Revisited

In the design of combinational logic the problem of minimization is well defined. Techniques such as K maps, Quine-McCluskey tabular procedures and computer aides help the dèsigner to produce good realizations with a minimum of effort.

With sequential logic the design of minimum cost circuits is more demanding. Each of the following parts of the design process can contribute extensively to the economy of the final design.

1. Specification of circuit behavior - the state diagram
   a) minimal representation
   b) state redundancy
2. Assignment of state variables
   a) relation of state variables to output variables
   b) The number of state variables used. (The minimum number of state variables does not necessarily produce the minimum cost final result.)
3. Combinational logic minimization
   a) Two level realization
   b) multi level realization
   c) multiple output design
   d) read only memory
4. Hardware choices
   a) NAND or NOR gates
   b) Type D or JK flip flop

Most of the above considerations are the responsibility of the logic designer. Even bad choices will result in a working circuit, but perhaps at a higher cost.

In the state diagram description of a sequential design it is possible that superfluous states may exist. Testing for such conditions and eliminating them may result in significant cost reduction.

Two states may be combined into a single state if they are <u>equivalent</u>. A simple test for equivalence can be performed on either the state diagram or the state transition table.

1. For the state diagram (the arrow out test) two states are equivalent if all arrows out of both states for corresponding inputs terminate on the same or equivalent states and have identical outputs. It is possible for one state to have input conditions not allowed by the other.
2. For the state transition table; two states are equivalent if all corresponding inputs to both states result in a transition to the same or equivalent states with identical outputs.

The removal of redundant states is important for economy of design. With fewer states it may be possible to use fewer memory elements. It may also allow greater minimization in the design of the excitation for the flip flop memory drive.

The following example illustrates the process of redundant state removal (combination) and resulting design simplification.

Example 5

A rotating device is equipped with a disk and optical sensors as shown in Figure 5. 4. 8. Design a clocked sequential circuit which will distinguish between clockwise and counter-clockwise rotation of the disk.

Solution:

The logic outputs from the sensors form a repetitive sequence which is different from the clockwise and counter-clockwise cases as shown in Table 5. 4. 2.

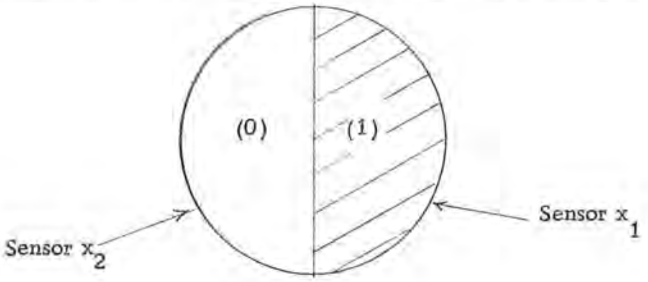


Figure 5. 4. 8. Rotating disk with sensors.

x_1	x_2	x_1	x_2
1	0	1	0
1	1	0	0
0	1	0	1
0	0	1	1
Clockwise Sequence		Counter Clockwise Sequence	

Table 5.4.2. Input variables for clockwise and counter clockwise disk rotation.

Figure 5.4.9 is a state diagram for the circuit. Z = 1 is used to represent clockwise rotation. The circuit automatically responds to changes in direction.

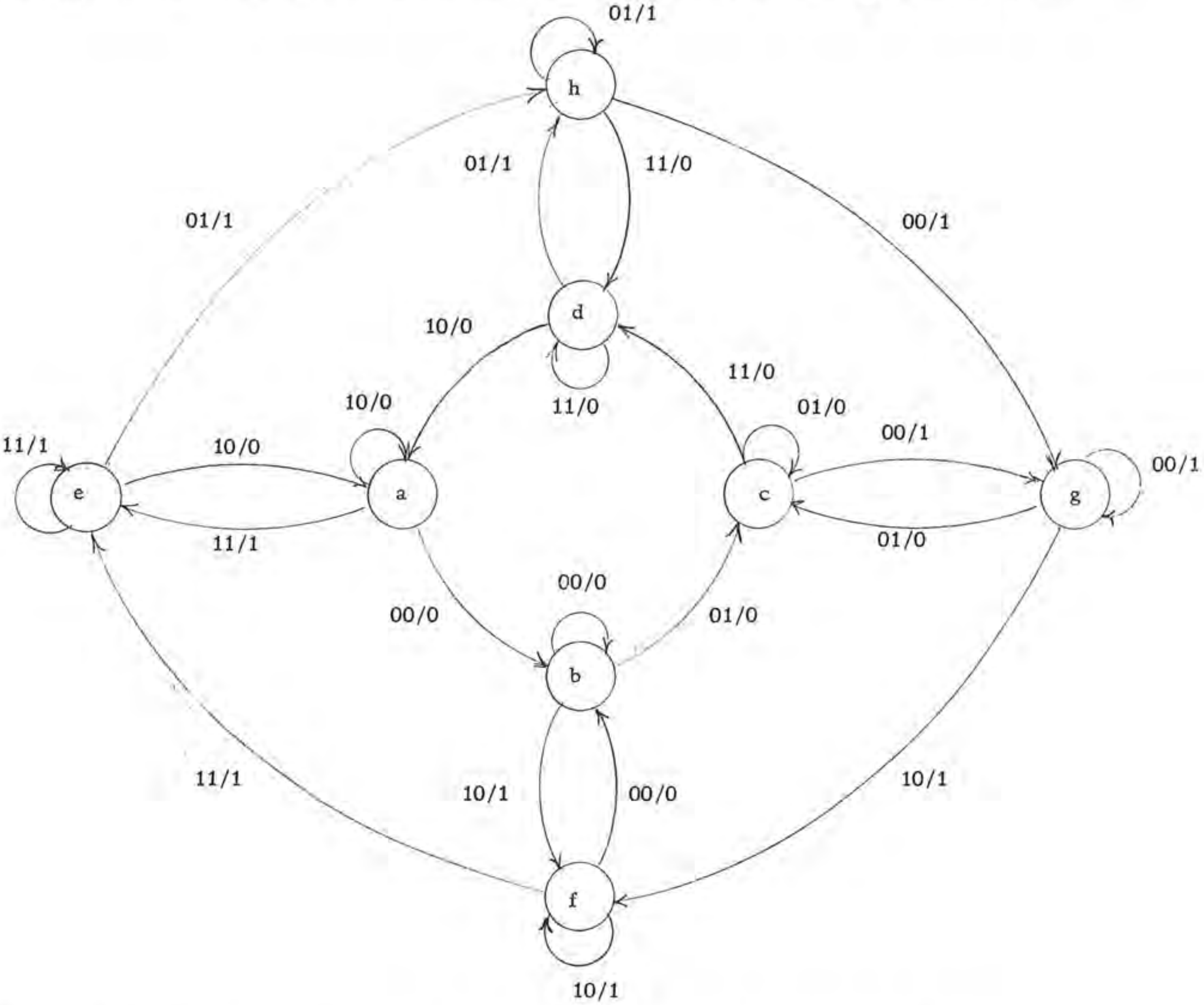


Figure 5.4.9. State diagram for circuit to detect direction of rotation for a disk.

State equivalence may be investigated by tabulating "arrows out" conditions. The results are shown in Table 5.4.3. The state destination is shown following the second slash.

State	Arrows Out		
a	10/0/a,	00/0/b,	11/1/e
b	00/0/b,	10/1/f,	01/0/c
c	01/0/c,	00/1/g,	11/0/d
d	11/0/d,	01/1/h,	10/0/a
e	11/1/e,	10/0/a,	01/1/h
f	10/1/f,	00/0/b,	11/1/e
g	00/1/g,	01/0/c,	10/1/f
h	01/1/h,	11/0/d,	00/1/g

Table 5.4.3. Arrows out conditions for Figure 5.4.9.

Two states are equivalent if their arrows out conditions are identical.

States a, b, may not be combined because the common input condition 10 results in different outputs.

States (a, e) (b, f) (c, g) and (d, h) may be combined. The modified state diagram is shown in Figure 5.4.10.

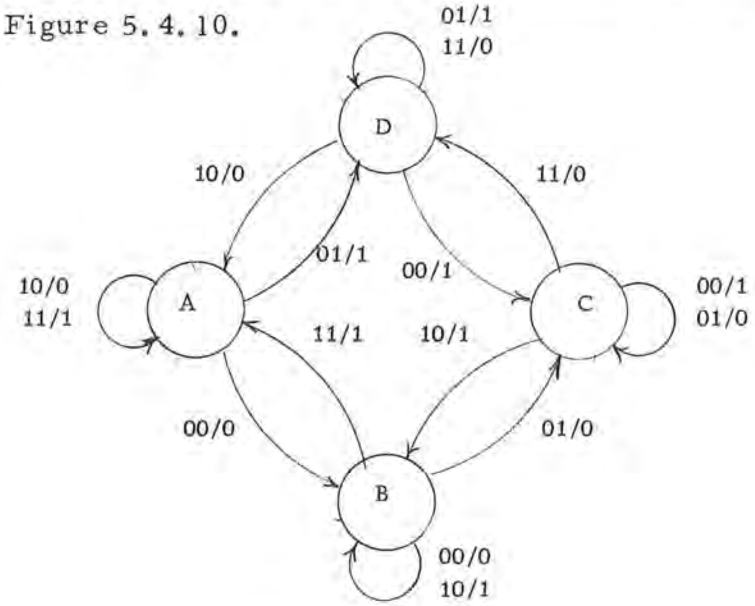


Figure 5.4.10. Reduced state diagram formed by combining states of Figure 5.4.9.

The combination of equivalent states has resulted in a reduction to four states; thus only two state variables are needed. These may be arbitrarily assigned to complete the design.

5.6 Sequential Design Using Read Only Memory

The availability of reliable, economical, and compatible read only memory (ROM) offers an attractive design alternative for sequential logic network. Use of ROM requires only passing attention to the state variable assignment problem since all assignments will work equally well. Consider Table 5.6.1 which is a "next state" and "output" description of the combination lock of Example 4.

Address				Contents		
Present State		Input		Next State		Output
Q_1	Q_2	X_1	X_2	Q_1^* (D_1)	Q_2^* (D_2)	Output Z
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	1	0
0	1	1	0	0	1	0
0	1	1	1	1	1	0
1	0	0	0	0	0	1
1	0	0	1	1	0	1
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	0	1	0	1	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Table 5.6.1.

Table 5.6.1 can be easily implemented using ROM. The combined four bits of "Present State" and "input" information can be used as an "address" input to a

ROM. In the address are stored the excitation variables necessary for state transition and the output variable.

If type D flip flops are used the D excitation is identical to the next state. The complete design is shown in Figure 5.6.1.

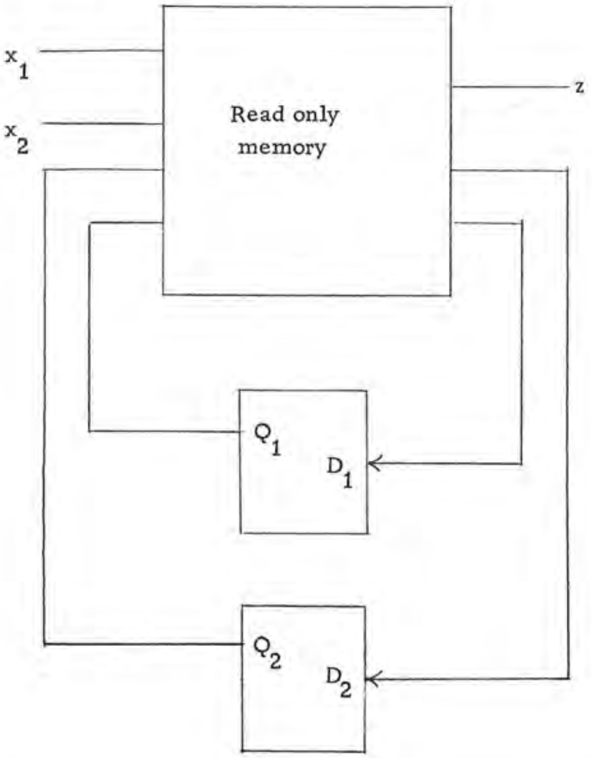


Figure 5.6.1. Use of ROM for a sequential logic network.

5.7 Notes - References - Problems

Most standard textbooks present a treatment on clocked sequential design. A good general treatment is presented in Booth (1). For treatments with more mathematical rigor the works by Dietmeyer (2), Hill and Peterson (3) and Krieger (4) are suggested.

1. Booth, T. L., (1971), "Digital Networks and Computer Systems", John Wiley, New York.
2. Dietmeyer, D. L., (1971), "Logic Design of Digital Systems", Allyn and Baron, Boston.
3. Hill, F. J., Peterson, G. R., (1968), "Introduction to Switching Theory and Logical Design", Wiley, New York.
4. Krieger, M., (1967), "Basic Switching Circuit Theory", MacMillan, New York.

EXERCISES

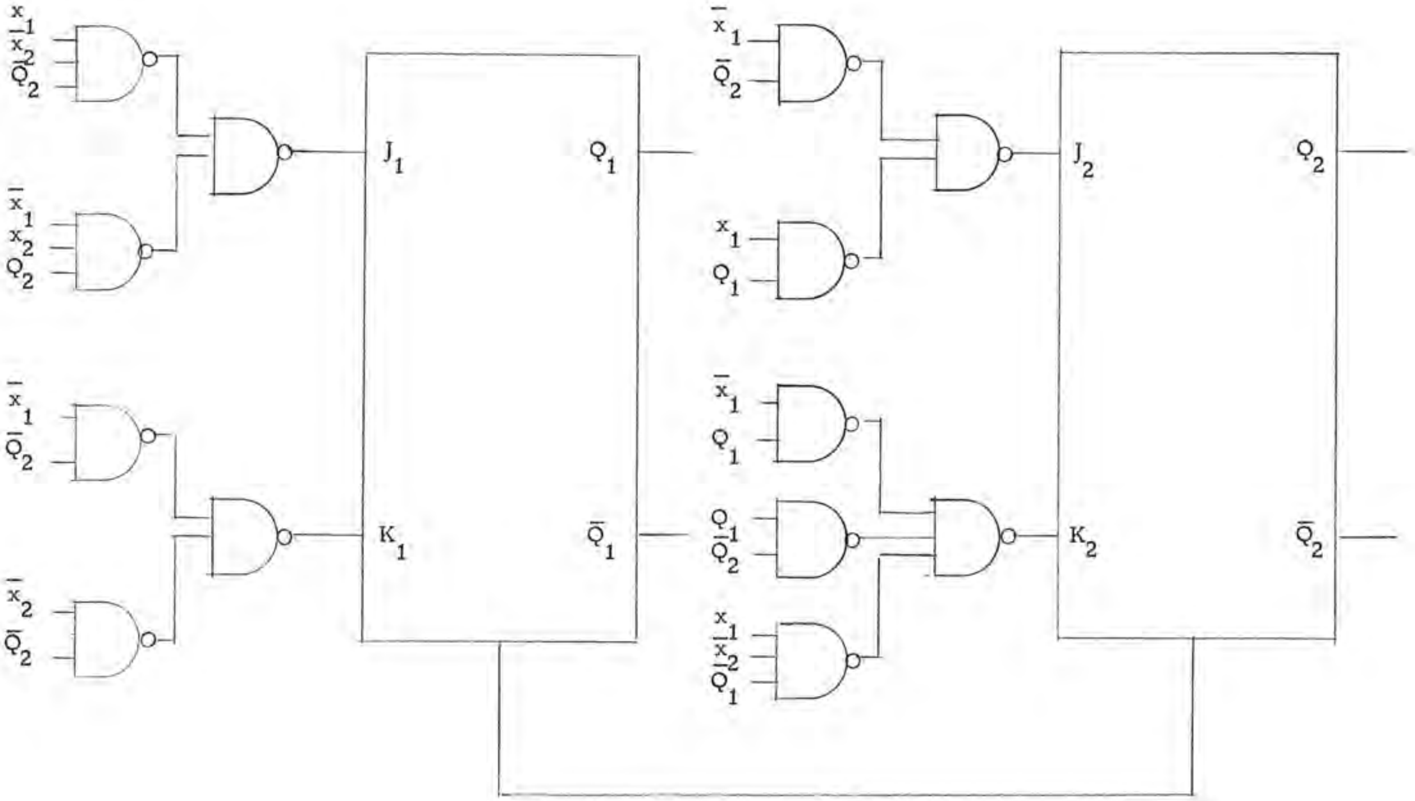
1. Design a counter using JK flip flops which will produce the sequence:

00
01
11
10

Repeat using type D flip flops.

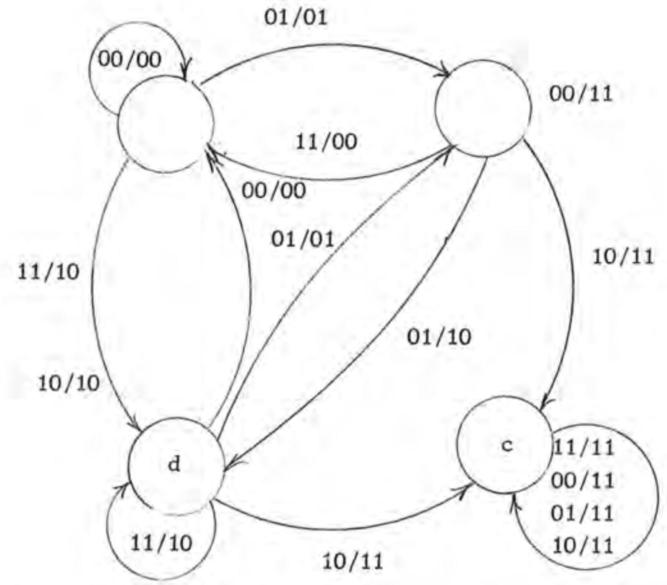
2. Using the same instrumented disk shown in Figure 4.7, design a network which will have a logic 1 output if the disk remains in the same position for three consecutive clock pulses. (Assume that the disk does not normally make a full revolution in one clock period.) Include both the state transition table and the state diagram.

3. Analyze the following clocked sequential circuit which uses J-K flip flops and NAND gates. Obtain the state diagram.



4. Develop the state diagram for a synchronous sequential circuit which initially starts with an output of 01. At each clock pulse thereafter, the output is to be replaced by the binary representation of the modulo 4 product of the present output and a 2 digit binary number presented as an input. The input may be different at each clock pulse.

5. Design a synchronous sequential circuit using only type D flip flops and NOR gates which will implement the following state diagram. Show all steps in your design process including the state transition table and the Karnaugh maps for the excitation of the flip flops. Use only 2 level logic. Minimize the logic required.



6. Design a clocked sequential circuit to generate the sequence:

Q_1	Q_2
0	0
0	1
1	0

For a memory element use a device with the following characteristics:

Present State Q	Input T	Next State Q
0	0	0
0	1	1
1	0	1
1	1	0

7. Complete the design of example 5 using type D flip flops.

6 Design of Asynchronous Sequential Circuits

6.1 Introduction

In a clocked sequential circuit the present state and the next state are delineated by a clock pulse. The clock establishes a common time for the state transition of all state variables.

In asychronous circuits, state transitions are not controlled by a clocking element. They are instead determined by the response time of the device to its external stimuli. Typical asynchronous elements are the Set-Reset Flip Flop and the electromechanical relay.

6.2 Analysis of Relay Sequential Circuits

A relay circuit becomes sequential rather than combinational when the contact network which energizes the relays contains contacts associated with the relays themselves.

Consider the relay circuit of Figure 6.2.1.

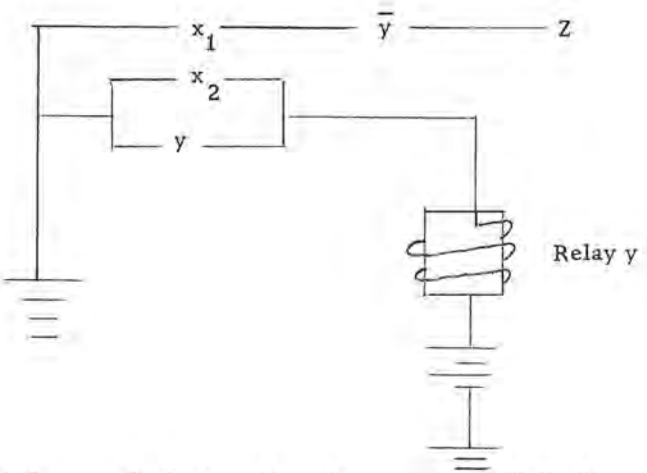


Figure 6.2.1. A relay sequential circuit.

The state variable associated with this circuit is y, the state of the relay. The analysis may be completed by means of a state transition table. The next state of y is determined by the present state of y and the effect of the input variables.

Present state	Input		Next state	Output
y	x_1	x_2	y^*.	z
0	0	0	0	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Table 6.2.1. State transition table for relay circuit of Figure 6.2.1.

The same information present in Table 6.2.1 may be shown in a state diagram.

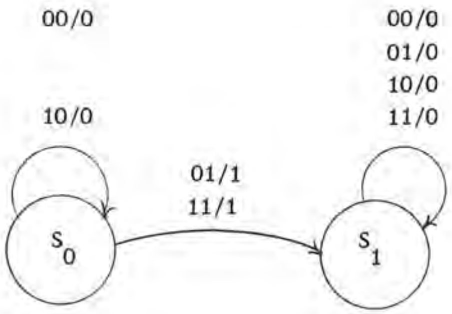


Figure 6.2.2. State diagram for relay circuit of Figure 6.2.1.

Some observations may be made here.

1. State transitions are initiated by changes in the input variables.
2. Transitions take place immediately. No external clock governs the change of state.
3. Non transitory conditions are indicated by "self loops" on the state diagram.

When conditions are such that the next state and the present state are identical the state is said to be stable. Otherwise the state is unstable.

Example

Analyze the following relay circuit.

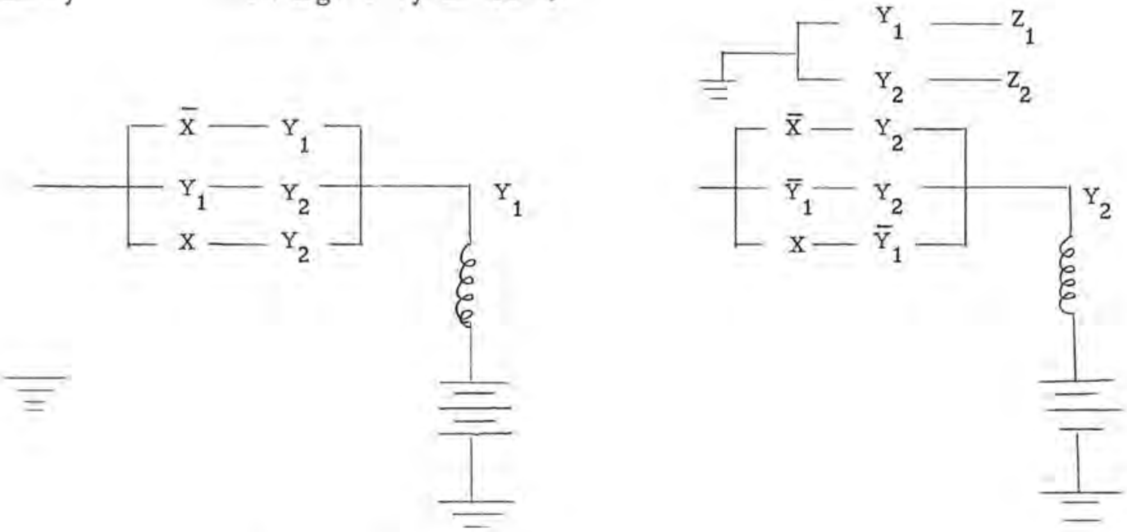


Figure 6.2.3.

Solution:

First obtain the state transition table and state diagram.

Present state		Input	Next state		Output	
Y_1	Y_2	X	Y_1^*	Y_2^*	Z_1	Z_2
0	0	0	0	0	0	0
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	1	1	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	1	0	1	1	1	1
1	1	1	1	0	1	1

Table 6.2.2. State transition table for Figure 6.2.3.

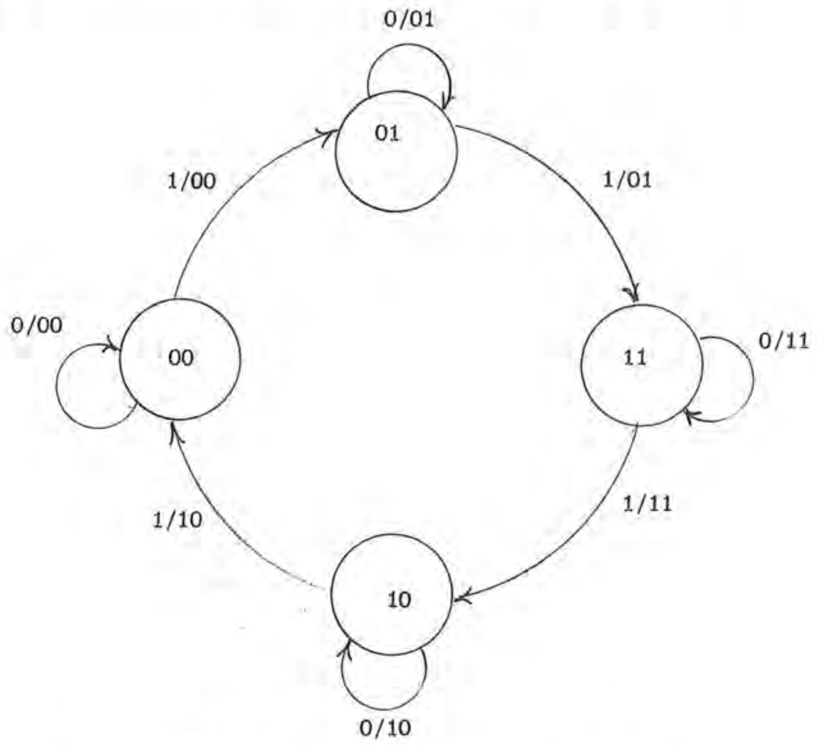


Figure 6.2.4. State diagram for Figure 6.2.3.

From Table 6.2.2 it can be seen that a state is never stable if X = 1. For the condition X = 1 the device cycles through all four possible states at a speed dependent only on the inherent speed of the device. This situation is called a buzzer cycle.

The buzzer cycle may be desirable as a basis for a digital oscillator. The buzzer cycle is unique to asynchronous sequential circuits. The circuit might be the basis of a random number generator. As the cycle X = 1, X = 0 is completed the output will correspond to one of the binary numbers 00, 01, 11, 10.

The circuit of Figure 6.2.4 demonstrates another phenomenon unique to asynchronous circuits, the race condition.

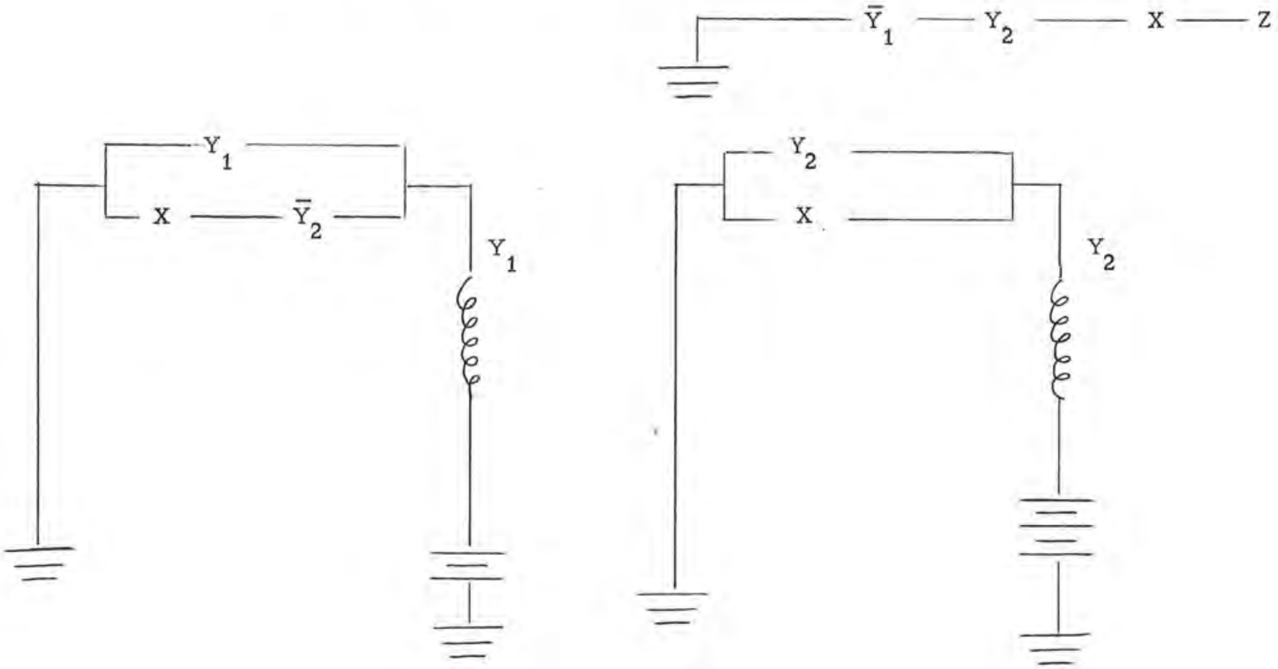


Figure 6.2.4.

Present state		Input	Next state		Output
Y_1	Y_2	X	Y_1^*	Y_2^*	Z
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Table 6.2.3. State transition table for Figure 6.2.4.

The problem depicted is subtle. Assume that the system starts in state 00 with $x = 0$; the state is stable. If x is changed to $x = 1$ the transition table indicates that the next state is to be $y_1y_2 = 11$. If, however, the devices associated with y_1 and y_2 are not identical, one device may assume the value 1 before the other. This means that momentarily the device will be in either $y_1y_2 = 01$ or $y_1y_2 = 10$. If the former results, no further changes will occur since $y_1y_2 = 01$ is a stable state. Thus the specified next state transition never happens. If the latter occurs the unstable condition $y_1y_2 = 10$ and $x = 1$ is entered. The next state for this condition is $y_1y_2 = 11$, the desired condition.

The conditions described above are called races. They are identified by state transition tables in which the next state differs from the present state by two or more state variables. Races may be critical, in which case the desired next state may never be reached, or the race may be safe, where the correct state is always eventually reached.

Another subtle problem which may occur in asynchronous logic is the Switching hazard.

Figure 6.25 and Table 6.24 demonstrate this phenomenon.

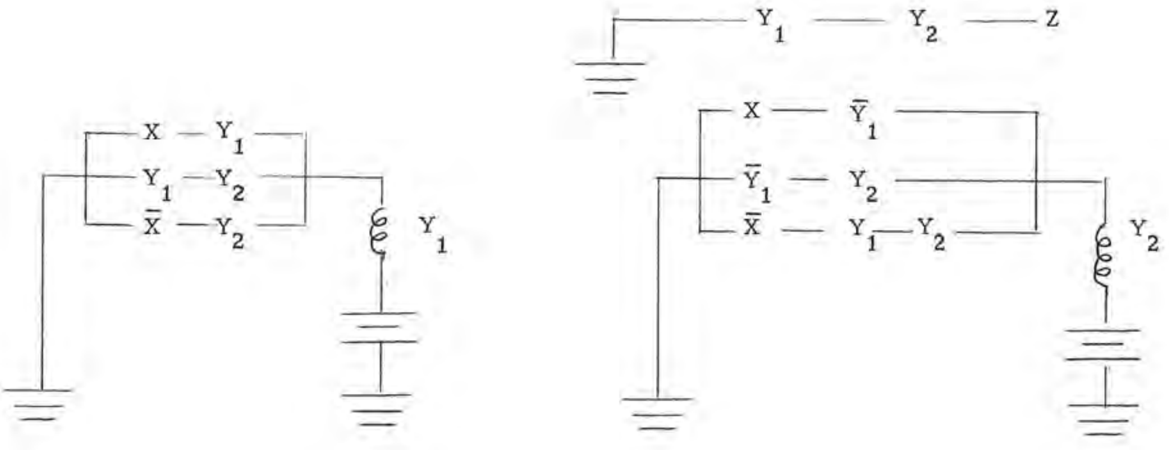


Figure 6.2.5.

Present state		Input	Next state		Output
y_1	y_2	x	y_1^*	y_2^*	z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	1

Table 6.2.4.

Assume that the device originates in the condition $y_1y_2 = 00$, $x = 0$. Following this $x = 1$ is entered and the device assumes the state $y_1y_2 = 01$, $x = 1$ (a stable condition). Now $X = 0$ is entered. The resulting condition desired is $y_1y_2 = 11$. The transition between $y_1y_2 = 01$ and $y_1y_2 = 11$ should not affect the state variable Y_2. An examination of the circuitry, however, shows the following. For $y_1y_2 = 01$, the y_2 relay is energized through the contact path $\bar{y}_1y_2$. In the condition $y_1y_2 = 11$, $x = 0$, relay y_2 is energized through the contact path y_1y_2. If the y_1 relay changes in such a way that there is an instantaneous condition when <u>neither</u> contacts y_1 or y_1 are closed, then relay y_2 will lose its excitation and the entire device will eventually enter state $y_1y_2 = 00$.

The switching hazard is due to imperfect switching characteristics of asychronous devices. Hazards may be avoided by careful design in which the excitation for a state variable is always continuous. This topic will be considered again in the next section.

6.3 Design of Asynchronous Sequential Circuits

Problems in the described previous section present some restrictions in the design of Asynchronous Circuits. In particular the problem of critical races and switching hazards must be avoided. The following restrictions will be applied to all asynchronous design.

1. Only one input variable may change at a time.
2. In state transitions, only one state variable may change. This condition eliminates the possibility of race conditions.
3. No swtiching hazards are allowed. This condition may be met by assuring the continuity of all excitation functions during state transitions.

Example 6.3.1

Design a counter which will count modulo 4 the number of times x is a logic 1.

Solution:

The state diagram for this device is shown in Figure 6.3.1. Eight states are required, one for each input x = 1 and for the subsequent input x = 0. The device is assumed to start in state a.

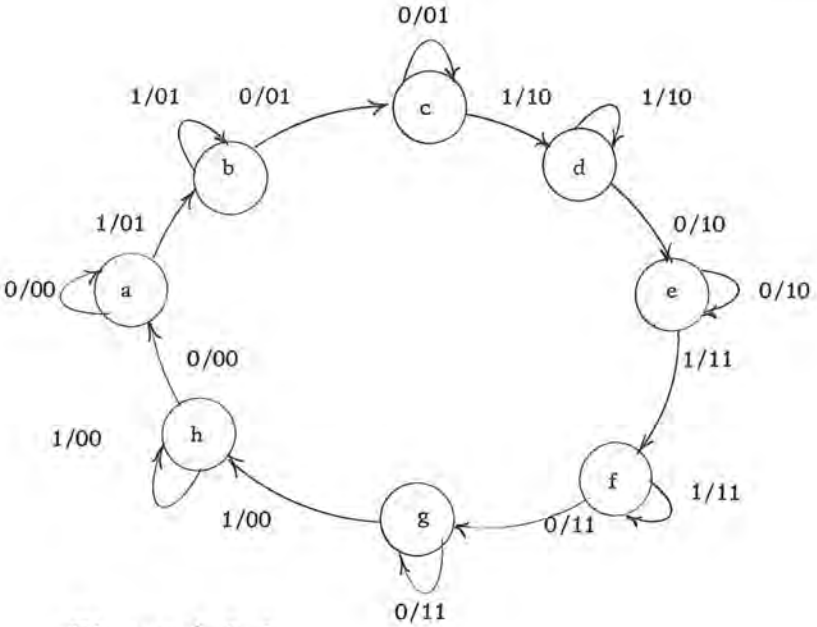


Figure 6.3.1.

In assigning state variables condition 2 regarding races must be satisfied. Three state variables will be sufficient only if they can be selected in such a way as to avoid races.

A possible state variable assignment is:

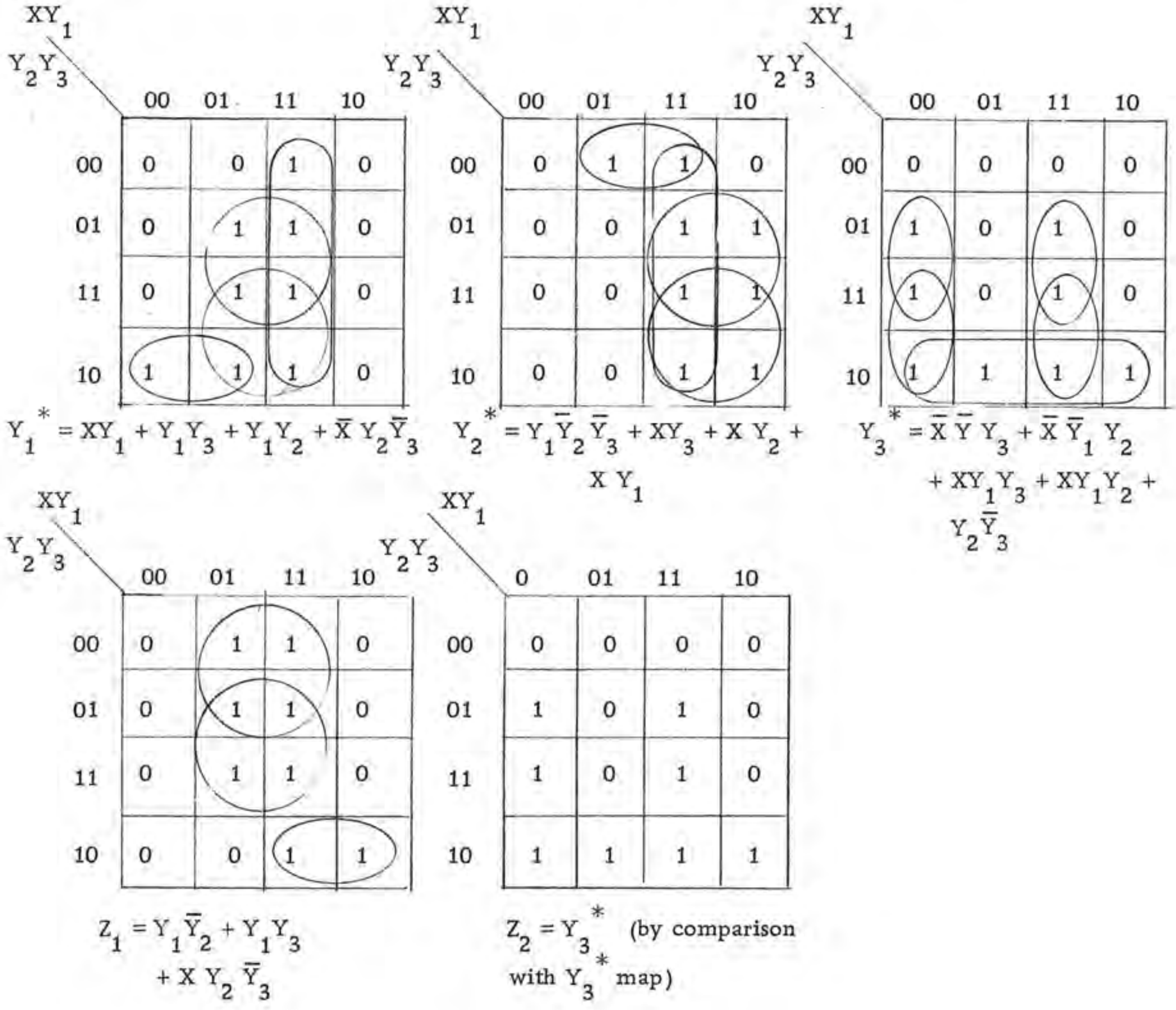
a	000	e	110
b	001	f	111
c	011	g	101
d	010	h	100

The state transition table for this state assignment is shown in Table 6.3.1.

Present state			Input	Next state			Output	
y_1	y_2	y_3	x	Y_1^*	Y_2^*	Y_3^*	Z_1	Z_2
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	1
0	0	1	0	0	1	1	0	1
0	0	1	1	0	0	1	0	1
0	1	0	0	1	1	0	1	0
0	1	0	1	0	1	0	1	0
0	1	1	0	0	1	1	0	1
0	1	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0
1	0	1	0	1	0	1	1	1
1	0	1	1	1	0	0	0	0
1	1	0	0	1	1	0	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	1	0	1	1	1
1	1	1	1	1	1	1	1	1

Table 6.3.1. State transition table.

For design of the individual networks the K map may be used.



Y_2Y_3 \ XY_1	00	01	11	10
00	0	0	1	0
01	0	1	1	0
11	0	1	1	0
10	1	1	1	0

$Y_1^* = XY_1 + Y_1Y_3 + Y_1Y_2 + \bar{X}Y_2\bar{Y}_3$

Y_2Y_3 \ XY_1	00	01	11	10
00	0	1	1	0
01	0	0	1	1
11	0	0	1	1
10	0	0	1	1

$Y_2^* = Y_1\bar{Y}_2\bar{Y}_3 + XY_3 + XY_2 + XY_1$

Y_2Y_3 \ XY_1	00	01	11	10
00	0	0	0	0
01	1	0	1	0
11	1	0	1	0
10	1	1	1	1

$Y_3^* = \bar{X}\bar{Y}Y_3 + \bar{X}\bar{Y}_1Y_2 + XY_1Y_3 + XY_1Y_2 + Y_2\bar{Y}_3$

Y_2Y_3 \ XY_1	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	0	1	1

$Z_1 = Y_1\bar{Y}_2 + Y_1Y_3 + XY_2\bar{Y}_3$

Y_2Y_3 \ XY_1	0	01	11	10
00	0	0	0	0
01	1	0	1	0
11	1	0	1	0
10	1	1	1	1

$Z_2 = Y_3^*$ (by comparison with Y_3^* map)

Figure 6.3.2.

Before selecting a minimum contact network, an assurance must be made that no switching hazards exist. This can be done by assuring that allowable state transition do not require a change in excitation paths for a state variable which retains a value of 1.

In the transition from state 001 to 011, the contact network exciting Y_3 must not be interrupted. Similarly the transitions between states 011, 010, 110, 111 must not interrupt Y_2.

One way to insure against hazards is to include <u>all</u> of the prime implicants of the function in the final logic realization. This assures that all adjacent entries in the K maps for each state variable are covered by a common contact path.

The logic required for Example 6.3.1 is shown in Figure 6.3.3.

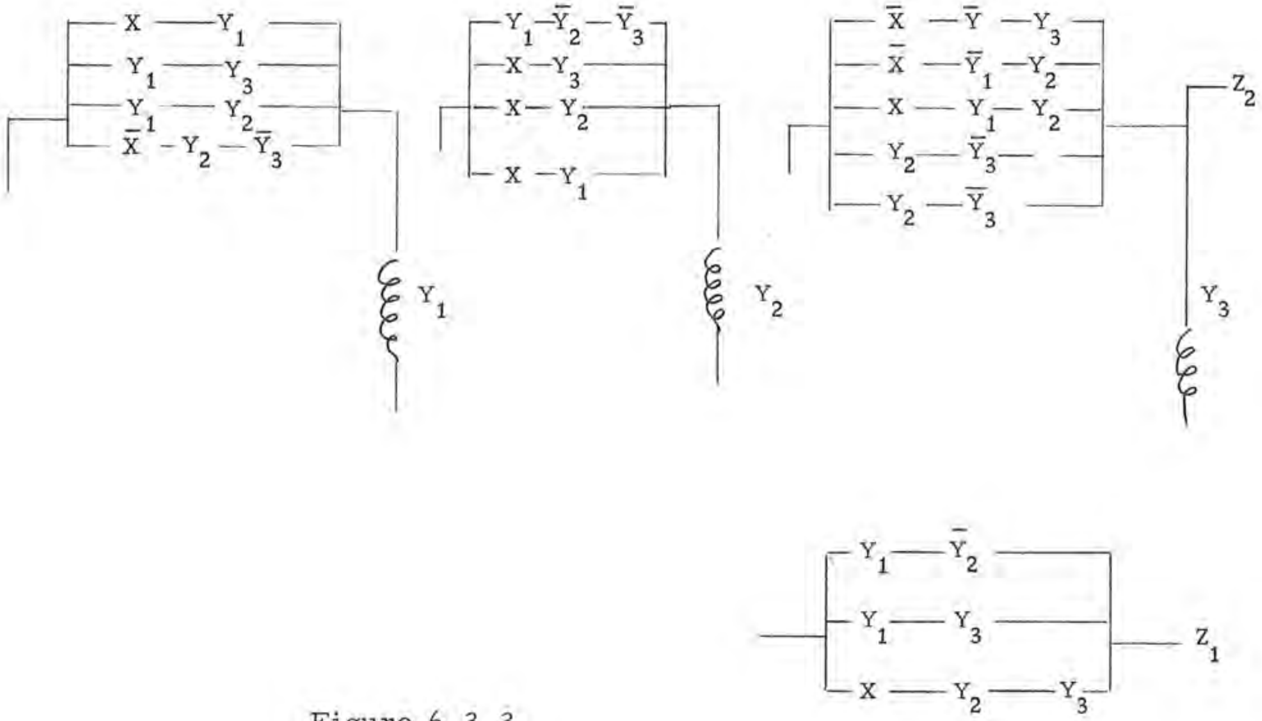


Figure 6.3.3.

Example 6.3.2

Design a Sequential Circuit using (a) Relays and (b) NAND gates to agree with the following state diagram.

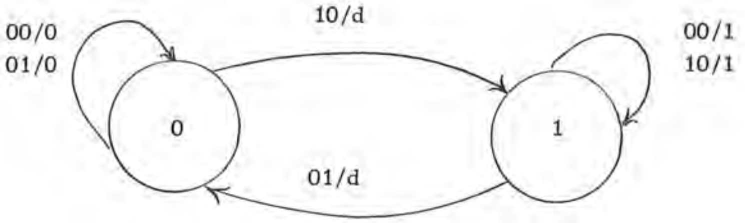


Figure 6.3.4.

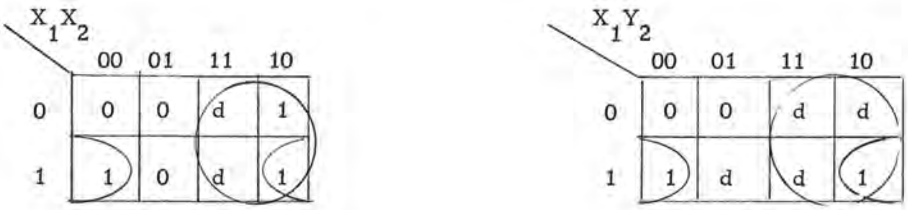
Solution:

The state transition table for the circuit is shown in Table 6.3.2.

Y	X_1	X_2	Y^*	Z
0	0	0	0	0
0	0	1	0	0
0	1	0	1	d
0	1	1	d	d
1	0	0	1	1
1	0	1	0	d
1	1	0	1	1
1	1	1	d	d

Table 6.3.2.

The K maps for Y and Z are shown in Figure 6.3.5.



X_1X_2	00	01	11	10
0	0	0	d	1
1	1	0	d	1

X_1Y_2	00	01	11	10
0	0	0	d	d
1	1	d	d	1

Figure 6.3.5.

$$Y^* = X_1 + \bar{X}_2\, Y \qquad\qquad Z = Y^*$$

Case a: Relay Circuit

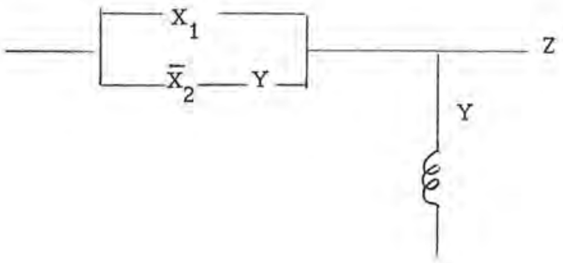


Figure 6.3.6.

Case b: NAND Gate Solution

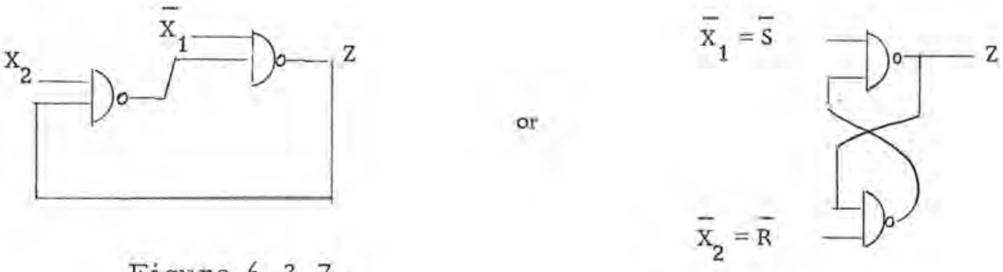


Figure 6.3.7.

The solution to Example 6.3.2 is the common Set-Reset Flip Flop.

Example 6.3.3

Design an Asynchronous Sequential Circuit using (a) Relays, (b) NAND gates, (c) Set Reset Flip Flops which will detect the sequence X_1X_2 = 10, 11, 01 each time it occurs and produce an output Z = 1 at the completion of the sequence.

Solution:

The state diagram for this network is shown in Figure 6.3.8.

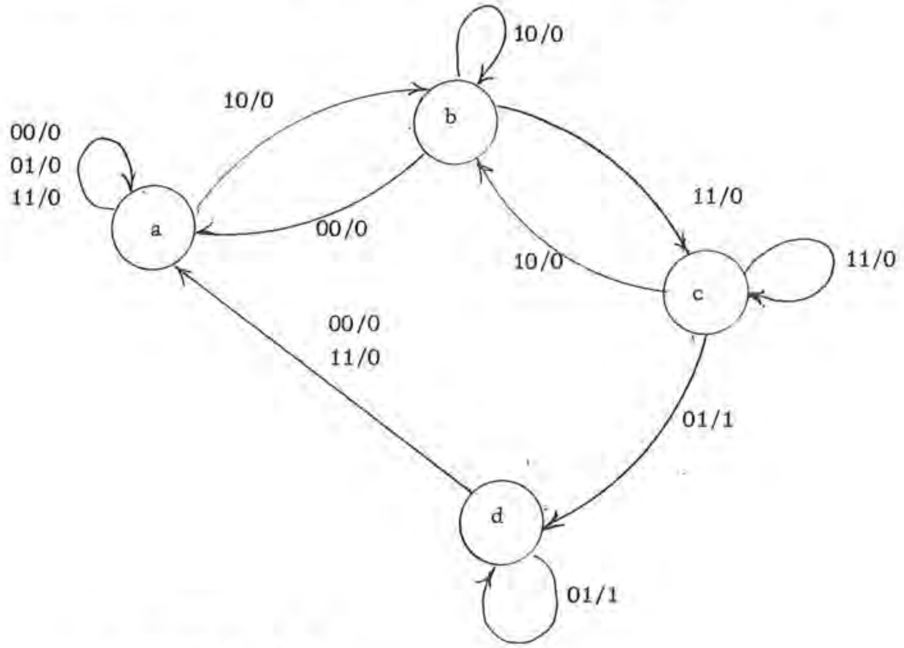


Figure 6.3.8.

A possible assignment of state variables is as follows:

a = 00 b = 01 c = 11 d = 10

In designing the network for the Set Reset Flip Flop, the following design characteristics are used. This assumes a NAND gate realization of the Flip Flop. To avoid the unnecessary use of inverters on the input terminals the design characteristics are expressed in terms of $\bar{S}$ and $\bar{R}$.

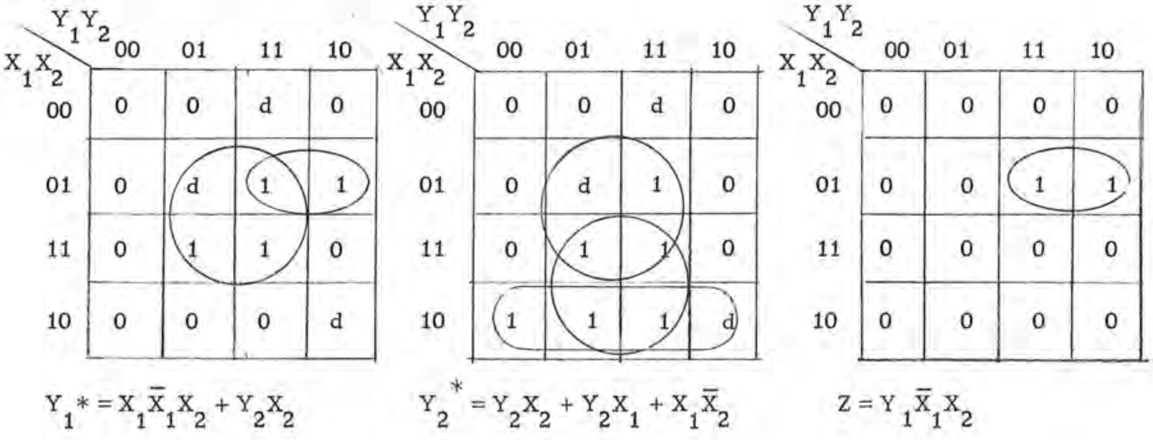
Present State Q	Next State Q^*	$\bar{S}$	$\bar{R}$
0	0	1	x
0	1	0	1
1	0	1	0
1	1	x	1

Table 6.3.3. Design characteristics of Set Reset Flip Flop

Present State		Input		Next State			Excitation Variables			
Y_1	Y_2	X_1	X_2	y_1^*	y_2^*	z	$\bar{S}_1$	$\bar{R}_1$	$\bar{S}_2$	$\bar{R}_2$
0	0	0	0	0	0	0	1	x	1	x
0	0	0	1	0	0	0	1	x	1	x
0	0	1	0	0	1	0	1	x	0	1
0	0	1	1	0	0	0	1	x	1	x
0	1	0	0	0	0	0	1	x	1	0
0	1	0	1	d	d	0	x	x	x	x
0	1	1	0	0	1	0	1	x	x	1
0	1	1	1	1	1	0	0	1	x	1
1	0	0	0	0	0	0	1	0	1	x
1	0	0	1	1	0	1	x	1	1	x
1	0	1	0	d	d	0	x	x	x	x
1	0	1	1	0	0	0	1	0	1	x
1	1	0	0	d	d	0	x	x	x	x
1	1	0	1	1	1	1	x	1	x	1
1	1	1	0	0	1	0	1	0	x	1
1	1	1	1	1	1	0	x	1	x	1

Table 6.3.4. Transition table for sequence detector.

K maps may now be prepared for Y_1 Y_2 (for use in parts a, b) and $\bar{S}_1$, $\bar{R}_1$, $\bar{S}_2$, $\bar{R}_2$ (for use in part C).



X_1X_2 \ Y_1Y_2	00	01	11	10
00	0	0	d	0
01	0	d	1	1
11	0	1	1	0
10	0	0	0	d

$Y_1{*} = X_1\bar{X}_1X_2 + Y_2X_2$

X_1X_2 \ Y_1Y_2	00	01	11	10
00	0	0	d	0
01	0	d	1	0
11	0	1	1	0
10	1	1	1	d

$Y_2^{*} = Y_2X_2 + Y_2X_1 + X_1\bar{X}_2$

X_1X_2 \ Y_1Y_2	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	0	0	0
10	0	0	0	0

$Z = Y_1\bar{X}_1X_2$

Figure 6.3.9. K maps for sequence detector.

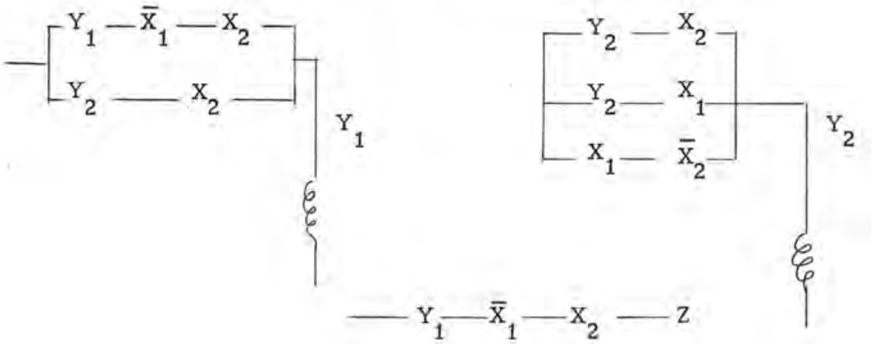


Figure 6.3.10. Contact network for sequence detector.

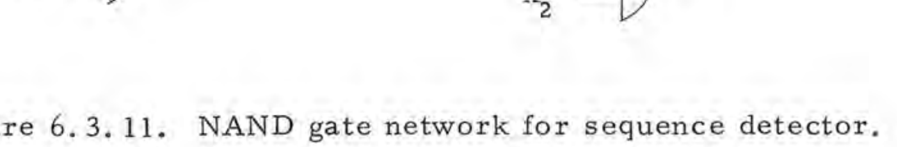
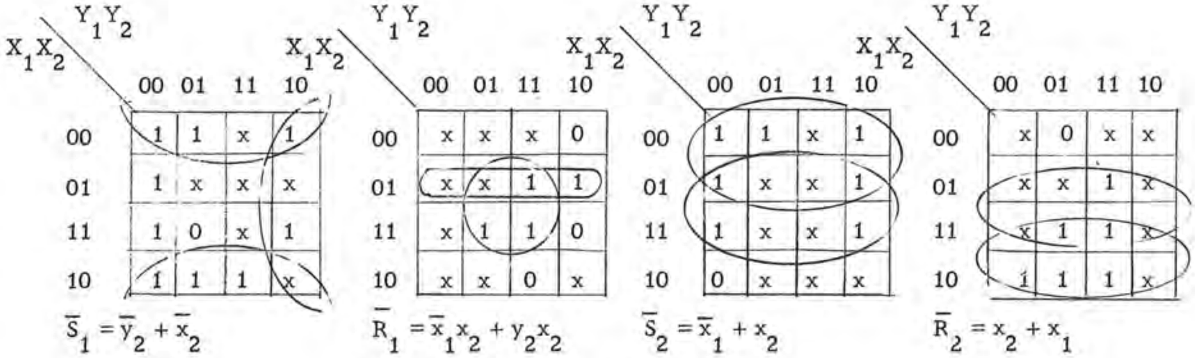


Figure 6.3.11. NAND gate network for sequence detector.



X_1X_2 \ Y_1Y_2	00	01	11	10
00	1	1	x	1
01	1	x	x	x
11	1	0	x	1
10	1	1	1	x

$\overline{S}_1 = \overline{y}_2 + \overline{x}_2$

X_1X_2 \ Y_1Y_2	00	01	11	10
00	x	x	x	0
01	x	x	1	1
11	x	1	1	0
10	x	x	0	x

$\overline{R}_1 = \overline{x}_1x_2 + y_2x_2$

X_1X_2 \ Y_1Y_2	00	01	11	10
00	1	1	x	1
01	1	x	x	1
11	1	x	x	1
10	0	x	x	x

$\overline{S}_2 = \overline{x}_1 + x_2$

X_1X_2 \ Y_1Y_2	00	01	11	10
00	x	0	x	x
01	x	x	1	x
11	x	1	1	x
10	1	1	1	x

$\overline{R}_2 = x_2 + x_1$

Figure 6.3.12. K maps for excitation variables.

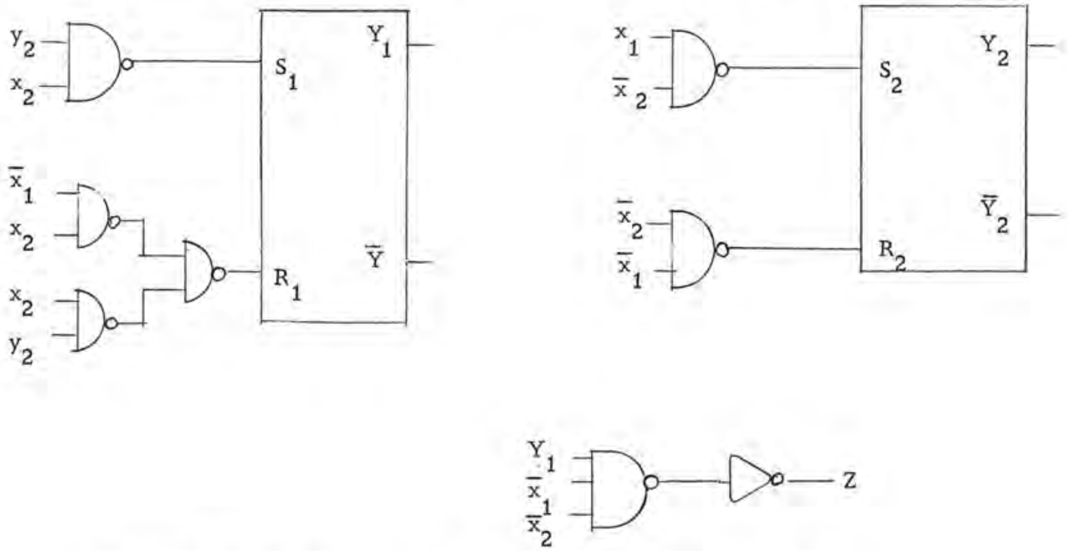


Figure 6.3.13. Set reset flip flop network for sequence detector.

6.4 Assignment of State Variables

The problem of assigning state variables is central to the design of Asynchronous Sequential Circuits. The following example illustrates the difficulty which may arise because of the possibility of race conditions.

Example 6.4.1

Assign state variables and prepare a transition table for the following asynchronous network.

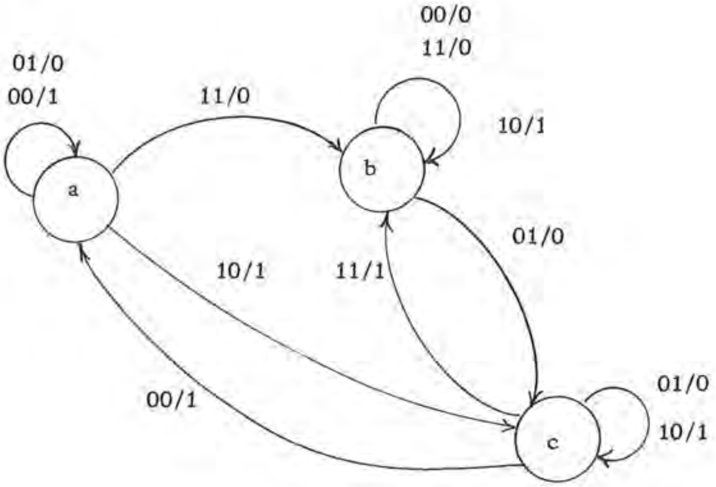


Figure 6.4.1.

Any assignment of state variables for example:

a = 00

b = 01

c = 11

will require a change in more than one state variable and result in a race condition. A possible solution is to introduce a new transitory state, d, as shown in Figure 6.4.1. The transition between states a and c may be "routed" through state d to avoid races.

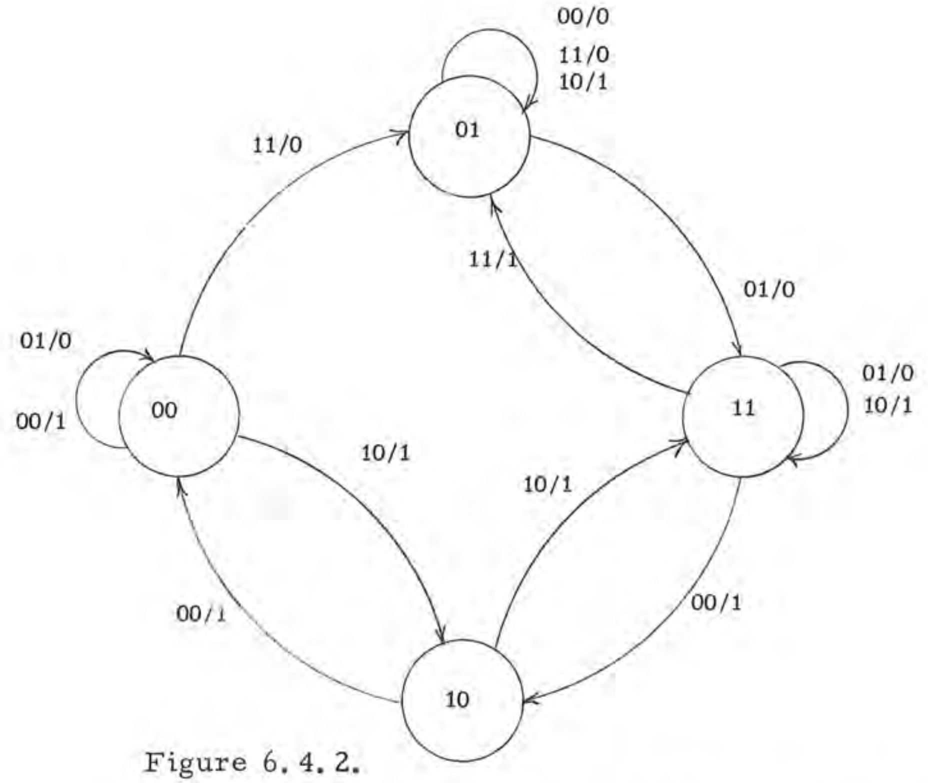


Figure 6.4.2.

Figure 6.4.3 shows a state diagram with four states. Because of the transitions allowed, there is no assignment of two state variables which will allow this circuit to be constructed. There are no "spare" states such as used in Example 6.4.1.

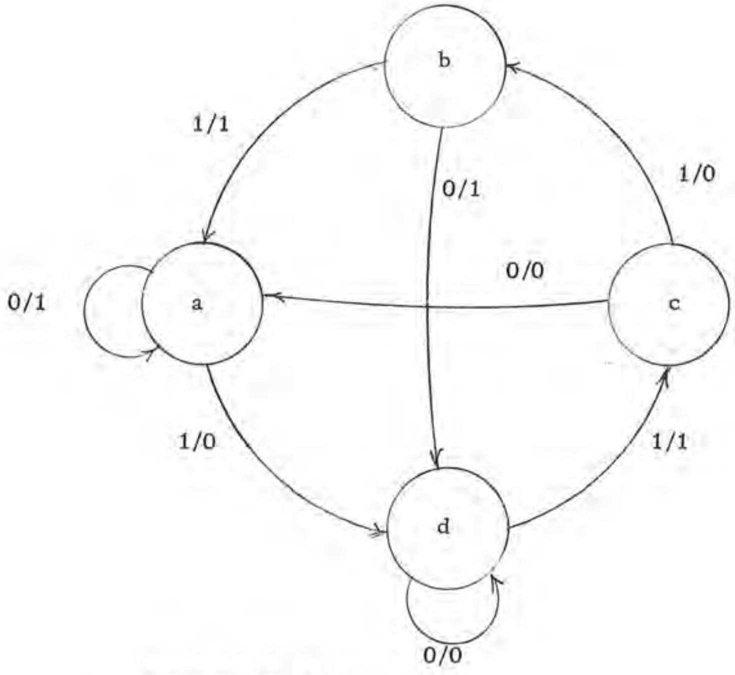


Figure 6.4.3.

For the situation in Figure 6.4.3, additional states must be created by the use of additional state variables. One such possible assignment is shown in Figure 6.4.4 in which additional states have been introduced.

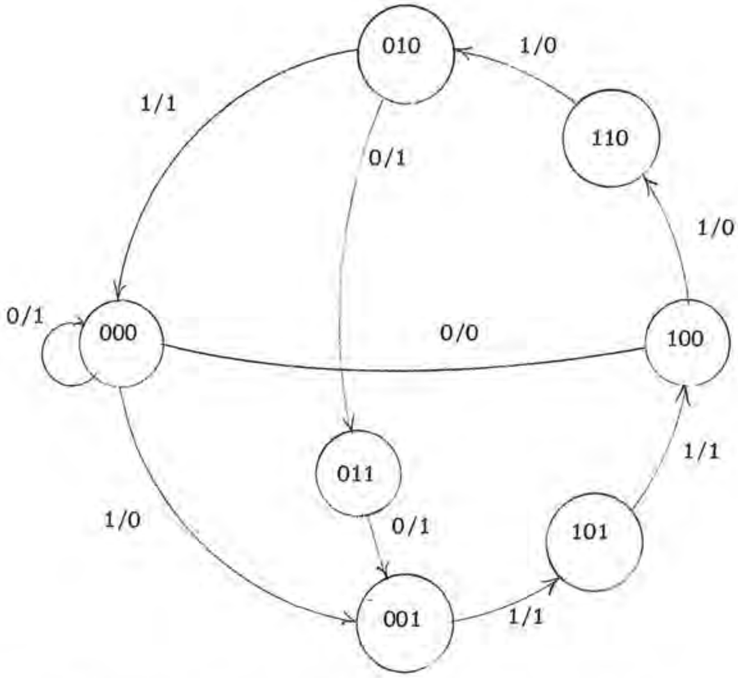


Figure 6.4.4. Possible assignment of state variables and introduction of transitory states.

The complexity of the final logic design is intimately connected with the state variable assignment selected.

6.5 Notes - References - Problems

Thorough treatments of the asynchronous design problem are presented in a variety of texts. Caldwell (1) presents most of his material in the form of relay design. Krieger (4) presents a design oriented approach. Dietmeyer (2) and Hill and Peterson (3) present extensive treatment of the problem of state assignment and minimization.

1. Caldwell, S. H., (1958), "Switching Circuits and Logical Design", Wiley, New York.
2. Dietmeyer, D. L., (1971), "Logic Design of Digital Systems", Allyn and Boran, Boston.
3. Hill, F. J., Peterson, G. R., (1968), "Introduction to Switching Theory and Logical Design", Wiley, New York.
4. Krieger, M., (1967), "Basic Switching Circuit Theory", MacMillan, New York.

EXERCISES

1. Design an asynchronous sequential circuit using relays which will sequentially turn on outputs Z_1, Z_2, Z_3, Z_4 each time a push button is depressed once. The sequence of outputs is to be:

Z_1	Z_2	Z_3	Z_4
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1

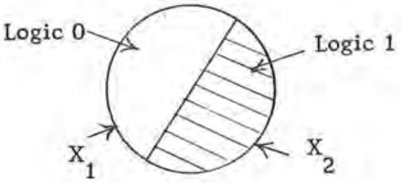
Assume the switch is depressed long enough to complete the sequence. Include a state transition diagram with your design.

2. Develop a state diagram for a clocked J-K Flip Flop. Consider the clock pulse to be one of the possible inputs. How many state variables are needed?

3. Assume that you have been commissioned to design a gambling device for use in Las Vegas. A "Dealer" and five others play. Each player and the dealer have a light bulb in front of them. One player depresses a push button and the lights in front of each participant Sequential turn on and off rapidly. When

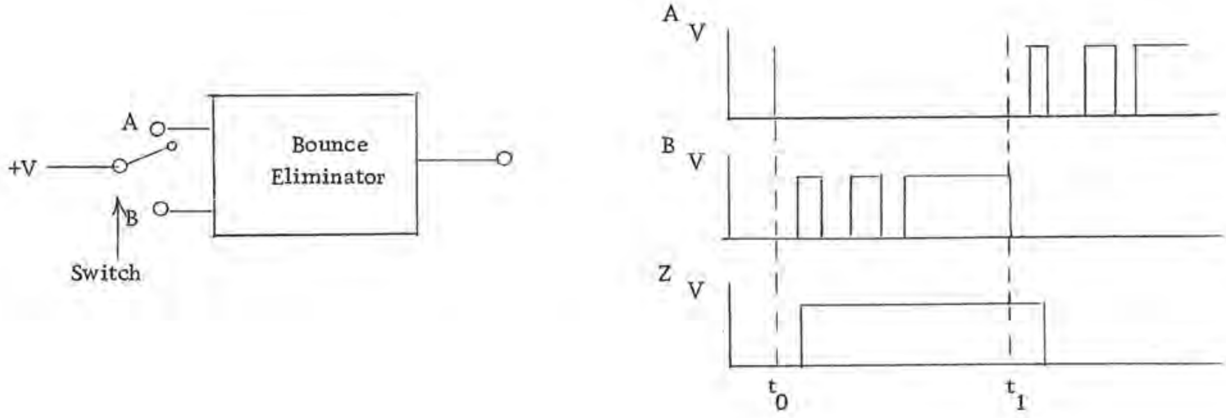
the button is released a single light remains on, the winner. Specify the state diagram and an acceptable set of state variables.

4. A rotating disc has two stationary wiper-type pickups which detect a logic 0 on one side of the disc and a logic one on the other.



Design an asynchronous circuit using (a) relays and (b) NAND gates which will have an output Z = 0, if the disc rotates counter clockwise and Z = 1 if the disc rotates clockwise. Include a state diagram.

5. At time t_0 a toggle switch moves from A to B. It makes initial contact at B, and then bounces several times (exact number not known) before settling permanently at B. It does not bounce back to A. At t_1 the switch is moved back to A with similar results. Design a bounce eliminator using only NAND gates to produce a Z output as shown, following B but eliminating the multiple transitions.



7 Fault Tolerant Design

In the early days of digital computers a certain, often high, level of failure and down time was tolerated. However, with digital technology being applied to ultra reliable systems such as health care, space craft and guided missiles, the study of fault tolerant digital design has received much attention.

A fault tolerant system can be defined as a system capable of some predefined level of operation in the presence of faults. These faults can include momentary or permanent hardware failures or software errors. The system can be designed to handle its entire work load in the presence of faults (fail-safe) or the system can be reconfigured or the performance allowed to degrade within bounds (fail-soft) when the fault condition occurs.

All techniques for fault tolerant design at all levels make use of some form of redundancy. At the system level, hardware and software means can be implemented for detection of faults, location of the faulty unit and reconfiguration of the system. Sparing is the most common system level form of hardware redundancy.

Diagnostic routines are implemented at the programming level consisting of a set of instructions which exercise all the hardware circuits in order to detect fault conditions. Instructions in a higher level language can be complied into redundant machine instructions as a validity check. For example, a multiply (*) in FORTRAM can be implemented with both a MULTIPLY machine instruction and a series of ADD and SHIFT machine instructions.

At the hardware level, fault tolerance can be achieved through either hardware or coding redundancy. Triple Modular Redundancy, Interwoven Logic and Quadded Logic are examples of hardware redundancy. Parity check codes, Hamming codes, AN codes and Residue codes are common coding techniques for error detection and correction.

Redundancy can be of the fault masking variety or consist of diagnosis-recovery procedures. In the former, a level of fault tolerance is usually obtained through a voting scheme where the behavior of a faulty unit is masked from the system operation as long as a majority of the redundant units are operating properly.

In the latter case, detection and location of a fault condition iniates corrective action which may consist of repeating the operation or switching a spare unit. Regardless of the amount of redundancy used, the system cannot last forever unless faulty units are located and replaced. A set of tests for hardware must be derived through analysis or simulation. These tests are then translated into diagnostic routines at the machine language or micro programming level. Test points in the system are identified and monitored while the diagnostic is run. Results can lead to detection, location and replacement of the faulty unit.

7.1 Hardware Redundancy

Redundant hardware means extra cost. Consideration of the cost versus reliability tradeoff is essential. Suppose a non-redundant (simplex) system has reliability, R_o and failure probability, $F_o = 1 - R_o$. Using an exponential model with a failure rate λ_o, the system realibility is

$$R_{sys} = R_o = e^{-\lambda_o t} \qquad \text{Eq. 7.1}$$

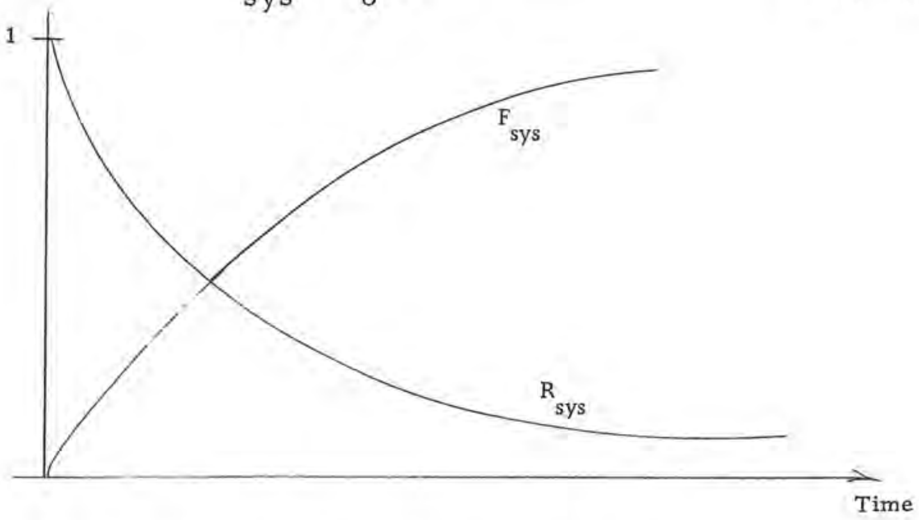


Figure 7.1. Reliability of a simplex system.

An important parameter of reliability, the mean time to faiture (MTF) can be defined

$$MTF = \frac{1}{\lambda} \qquad \text{Eq. 7.2}$$

Consider a redundant system (Figure 7.2) with n identical active parallel units each with reliability

$$R_o = e^{-\lambda_o t} \quad .$$

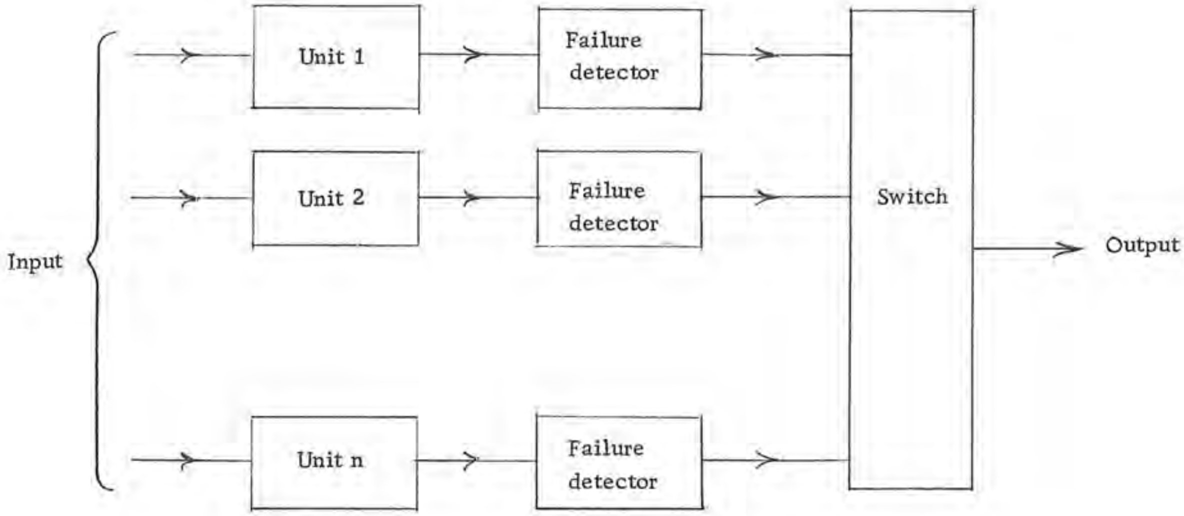


Figure 7.2. Redundant system.

The failure probability for the redundant system of Figure 7.2 is

$$F_{sys} = \Pi F_i = F_o^n$$

where F_o is the failure probability of each individual unit. The system reliability is

$$\begin{aligned} R_{sys} &= 1 - F_{sys} \\ &= 1 - F_o^n \\ &= 1 - (1 - R_o)^n \\ &= 1 - (1 - e^{-\lambda_o t})^n \end{aligned} \qquad \text{Eq. 7.3}$$

The MTF for a single unit is λ_o. For the system,

$$MTF_{sys} = \frac{1}{\lambda_o} \sum_{k=1}^{n} \frac{1}{k} \qquad \text{Eq. 7.4}$$

For a system with one active spare unit,

$$\begin{aligned} R_{sys} &= 1 - [1 - e^{-\lambda_o t}]^2 \\ &= 1 - 1 + 2e^{-\lambda_o t} - e^{-2\lambda_o t} \\ &= 2e^{-\lambda_o t} - e^{-2\lambda_o t} \end{aligned} \qquad \text{Eq. 7.5}$$

and

$$MTF_{sys} = \frac{1}{\lambda_o}\left(1 + \frac{1}{2}\right) = 1.5\ MTF_o$$

Hence, a 50 percent increase in Mean Time to Failure is achieved at a 100+ percent increase in hardware (2 units + detectors + switch) and a 100 percent increase in power. The resulting reliability is given in Figure 7.3.

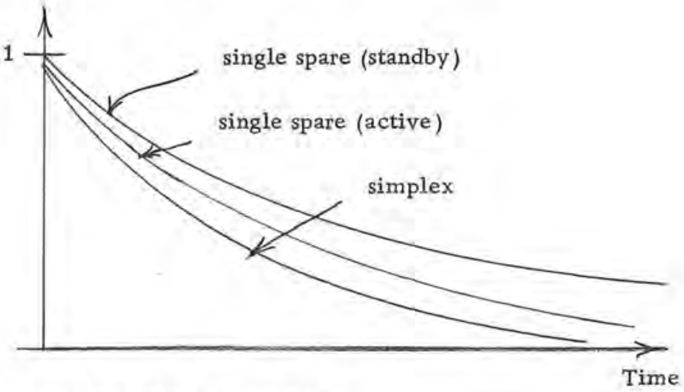


Figure 7.3. Reliability curves.

An alternative approach is to use inactive standby units which are powered up only when a failure is detected in the operating unit. The reliability of a system can be calculated as (n identical units)

$$R_{sys} = e^{-\lambda_o t} \sum_{r=0}^{n-1} \frac{(\lambda_o t)^r}{r!} \qquad \text{Eq. 7.6}$$

$$MTF_{sys} = \frac{n}{\lambda_o}$$

Solving 7.6 for n = 2 (1 active, 1 standby)

$$R_{sys} = e^{-\lambda_o t}\ (1 + \lambda_o t) \qquad \text{Eq. 7.7}$$

$$MTF_{sys} = \frac{2}{\lambda_o} = 2\ M_o \qquad \text{Eq. 7.8}$$

Hence, using standby redundancy a 100 percent in MTF can be obtained with still 100 percent increase in hardware, but no increase in power. A plot of Eq. 7.7 is given in Figure 7.3.

An example of a successful implementation of standby sparing is in the

JPL STAR (Self Test and Repair) computer (2). The STAR uses a highly modular system with each arithmetic unit having 2 spares. Diagnosis and switching is done by a Test and Repair Processor (TARP). The TARP itself is highly redundant using a Hybrid TMR design discussed later in this section. The actual switching is performed by powering up one unit and turning off another. All units are connected to a common bus, also spared, with dot-OR ed logic so that a powered down unit does not effect the input nor the output of the active units.

The most well known technique for hardware redundancy is Triple Modular Redundancy (TMR). TMR is a fault masking technique. Consider the TMR system shown in Figure 7.4.

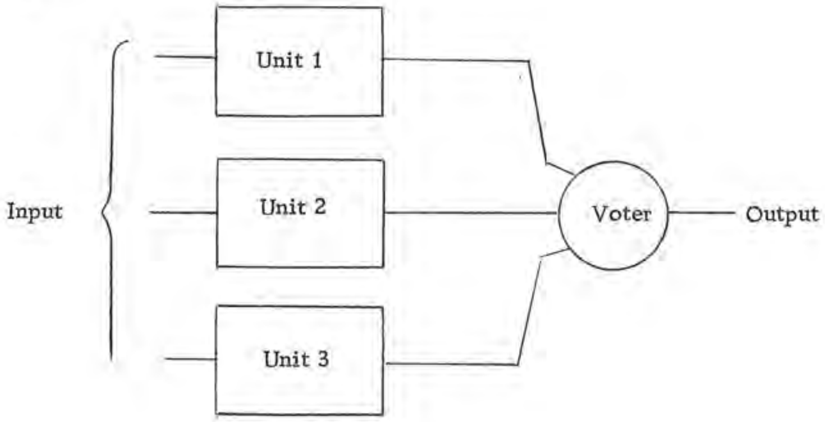


Figure 7.4. Triple Modular Redundant System.

Assuming identical units with independent failure rates, and a perfect voter, the realiability of the TMR system can be calculated by

$$R_{TMR} = R_o^3 + 3\,R_o^2\,(1 - R_o) \qquad \text{Eq. 7.9}$$

Assuming an exponential model

$$R_{TMR} = \left(e^{-\lambda_o t}\right)^3 + 3\left(e^{-\lambda_o t}\right)^2 \left(1 - e^{-\lambda_o t}\right)$$

$$= 3\,e^{-2\lambda_o t} - 2\,e^{-3\lambda_o t} \qquad \text{Eq. 7.10}$$

$$MTF_{TMR} = - \int R_{TMR}\,dt$$

$$= 5/6\,\lambda_o \qquad \text{Eq. 7.11}$$

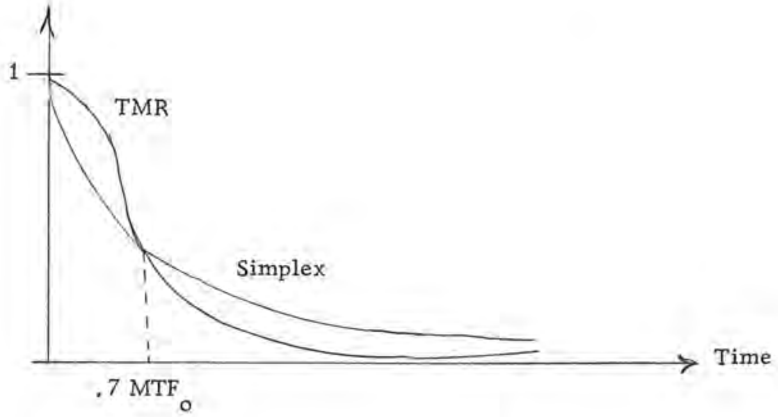


Figure 7.5. A comparison of TMR and Simplex Reliability.

Hence for a 200+ percent increase in components and power, TMR achieves a 17 percent decrease in MTF! The short term reliability of a TMR system, however, is better than the simplex system.

What are the advantages of TMR? Make the following assumptions (to be examined and weakened later):

1. System is divided into m identical modeuls.
2. Voter logic is failure free.
3. Modules are equally reliable and statistically independent.
4. System fails if any module fails.

If the module reliability could be determined as

$$R_o = R_{sys}^{1/m} \qquad \text{Eq. 7.12}$$

then

$$\begin{aligned} MTF_{sys} &= \int R_{sys}\, dt \\ &= \int R_o^{\,m} dt \\ &= \int e^{-m\lambda_o t} \\ &= \frac{1}{m\lambda_o} = \frac{1}{m}\, MTF_o \qquad \text{Eq. 7.13} \end{aligned}$$

The more the system is modularized, the lower failure rate for each module. No gain is obtained unless the modules can be chosen such that

$$MTF_o > m\,MTF_{sys}$$

In other words, the amount of modularity must be chosen such that the Mean Time to Failure for each module is greater than m times the MTF_{sys}. If the modules are so chosen and triplicated, the reliability of the system is:

$$R_{sys_{TMR}} = R^M_{TMR} = (3\,R_o^2 - 2\,R_o^3)^M$$

$$(3\,R_{sys}^{2/M} - 2\,R_{sys}^{3/M})^M \qquad \text{Eq. 7.14}$$

Using an exponential model $R_o = e^{-\lambda_o t}$ and $\lambda_o < \frac{1}{M}\,\lambda_{sys}$, the reliability curves obtained are given in Figure 7.6.

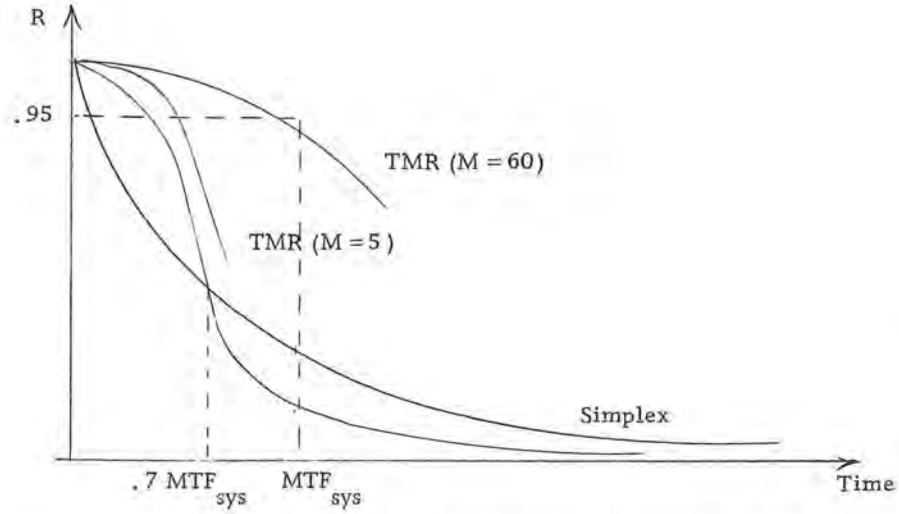


Figure 7.6. Reliability curves for modular TMR system.

Reconsider the assumptions. The first assumption relates to the choice of M identical modules. These modules need not be identical, but the system will depend on the least reliable module. One fault tolerant technique called bit slicing attempts to package all circuits which operate on one bit position of a data word together as a single unit. This has two advantages. Errors in a single bit of a data can be readily detected, but location of the faulty unit is much more difficult. With bit slicing all circuits for that one bit can be replaced. Secondly modularization of this form does provide uniformity.

It is the second assumption dealing with "perfect" or failure free voters which is subject to criticism. Voter logic is usually a simple majority gate and can be made highly reliable, but not perfect. Consider the scheme shown in Figure 7.7.

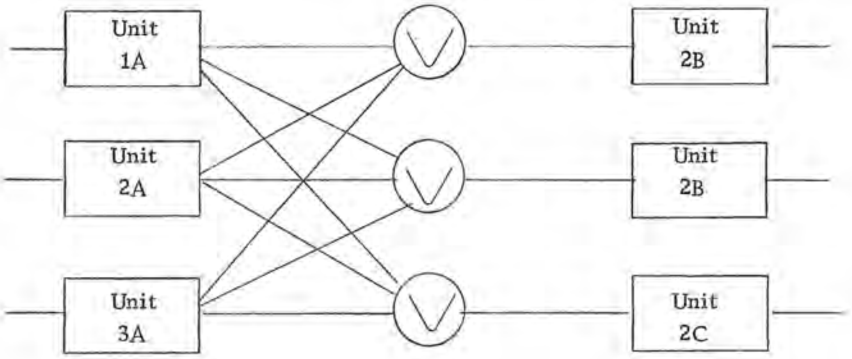


Figure 7.7. TMR with triplicated "imperfect" votes.

For this case, a voter failure is equivalent to a single unit failure in the next stage. If the voter reliability is R_v and assumed to be constant relative to the unit reliability R_o, then the TMR unit reliability with non perfect voters is given by

$$R_{TMR} = 3(R_vR_o)^2 - 2(R_vR_o)^3 \qquad \text{Eq. 7.15}$$

The system reliability is

$$R_{sysTMR} = [3(R_vR_o)^2 - 2(R_vR_o)^3]^M$$

A sketch of system reliability with $R_o = e^{-\lambda_o t}$ and $R_v = 0.999$ as a function of the modularity is given in Figure 7.8. The point of maximum reliability is

$$M_o = 1n\, R_{sys}/1n\, R_v$$

$$= {}^{\lambda}s/_{\lambda_v} \qquad \text{Eq. 7.16}$$

the ratio of the simplex (non redundant) system reliability to the voter reliability.

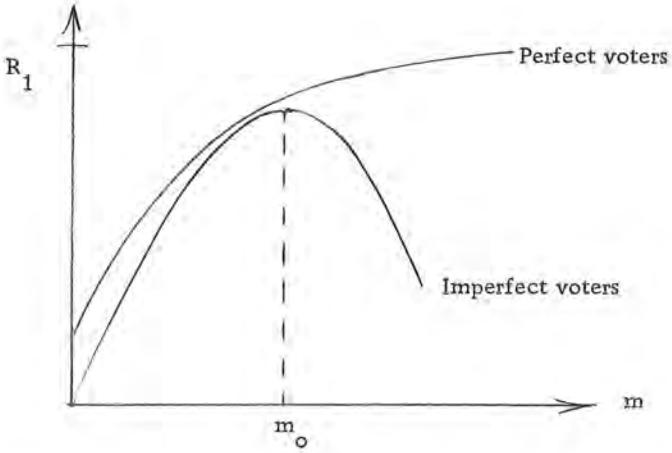


Figure 7.8. TMR system reliability as a function of modularity.

Note that if Eq. 7.16 is solved for $\lambda_v = \frac{\lambda_s}{m_o}$, m_o can be thought of as the number of modules which must be chosen to cause the module failure rate to equal that of the voter.

Of course, TMR techniques can be extended to higher redundancy. N modular redundancy for N = 2 n + 1, n = 1, 2, can be employed. TMR is just NMR with N = 1. A sketch of NMR reliability is given in Figure 7.9.

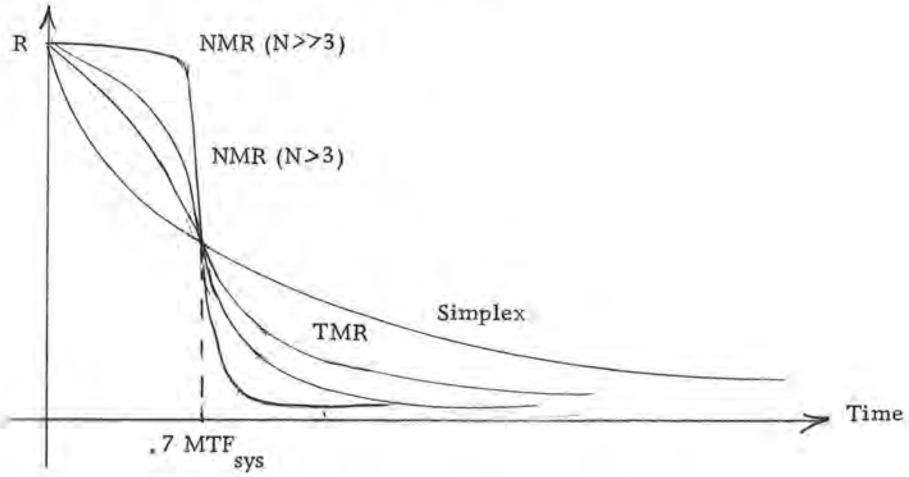


Figure 7.9. Reliability curves for NMR systems.

Using a redundancy of N > 3 increases the short term reliability ($t < 0.7\ MTF_{sys}$), but decreases the long term reliability. In summary, NMR is advantageous if the redundancy is employed in a modular system where $MTF_{module} >> MTF_{system}$ and $MTF_{module} \cong MTF_{voter}$. NMR is best implemented in systems such as guided missile control where ultra high short term reliability is essential. The cost of NMR is high and since it is a fault masking technique, diagnosis is difficult and often impossible. Furthermore, a redundant unit will fail when nearly half the units are still operating properly.

Various attempts have been made to design adaptive NMR systems where each unit has a variable percentage of the vote. An improvement in reliability is gained, but at the cost of increased hardware.

Recently (10), a technique combining NMR and standby sparing has been suggested. It is called Hybrid NMR and represented by H(N, S) where N is the degree of active redundancy and S is the number of spares. An example of a Hybrid NMR system is given in Figure 7.10.

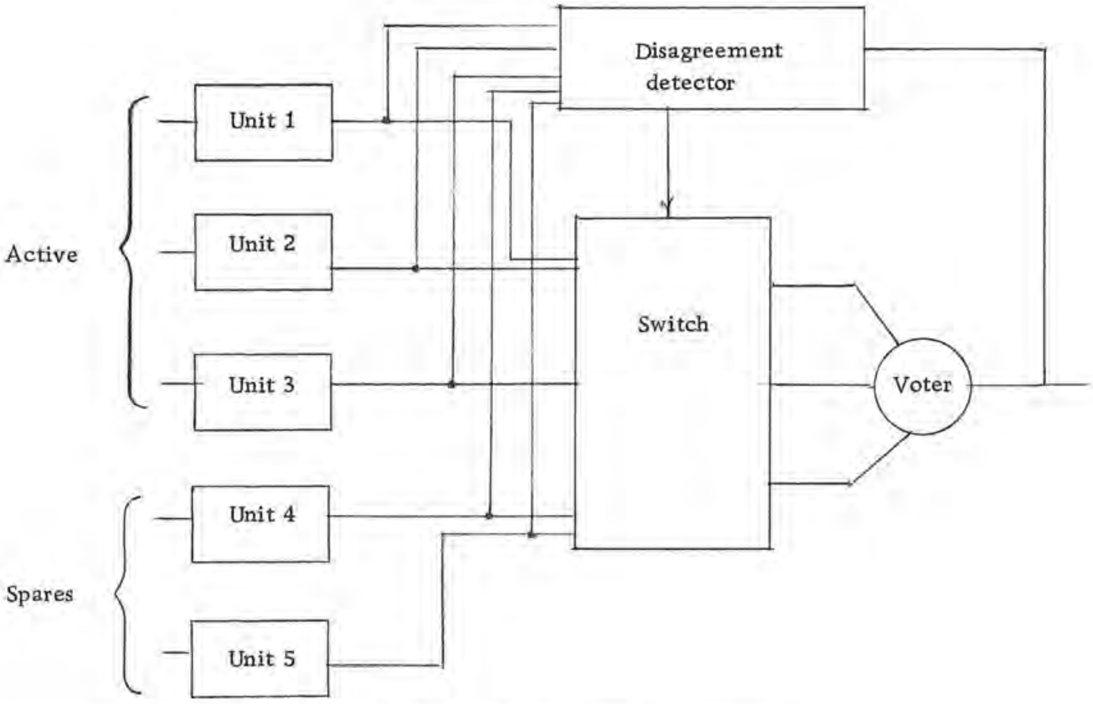


Figure 7.10. A hybrid NMR, H(3, 2), system.

In the system of Figure 7.10, the operation is essentially TMR until a unit fails. If the Disagreement Detector detects a difference in output of the voter and one of the TMR units, it switches out the faulty unit and switches in a spare unit.

Hybrid TMR is costly and requires the extra overhead of a switching unit and a disagreement detector. The switch can be distributed and along the voter logic made part of the next stage. The disagreement detector must be designed with high reliability. The long term reliability of hybrid NMR is appreciably better than NMR. Hybrid TMR has the advantage that the system will fail only if all but one unit has failed. Reliability curves are sketched in Figure 7.11.

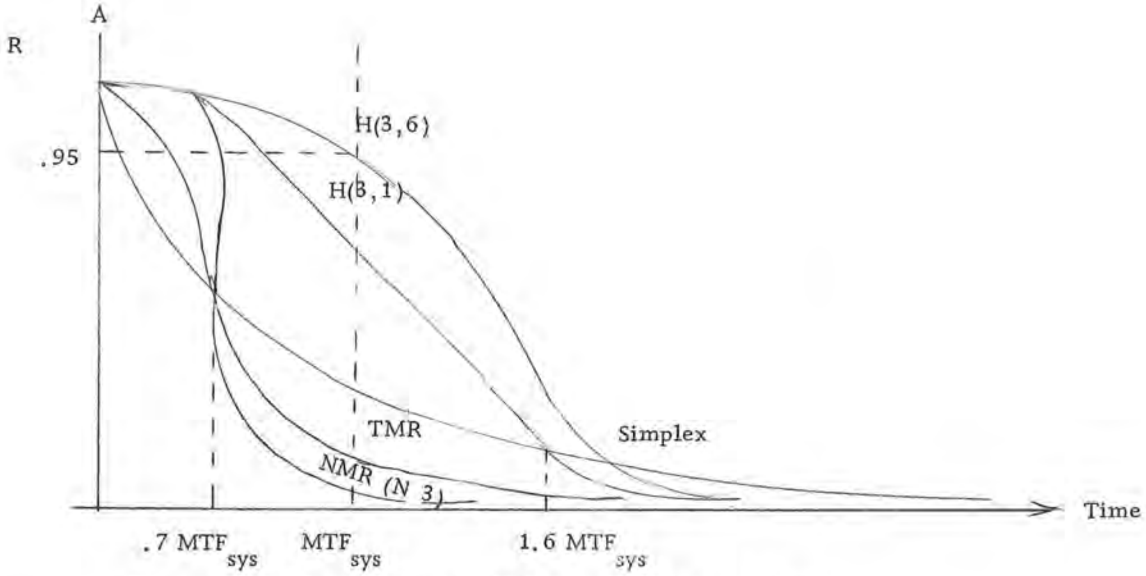


Figure 7.11. Hybrid NMR reliability curves.

TMR or Hybrid NMR techniques can be used for logic design, however, the cost of triplicating each gate and adding a majority gate is prohibitive. Most often TMR is used at a functional unit level.

Redundancy techniques exist for the logic level which make use of the fault masking properties of certain gates. Consider the AND gate in Figure 7.12(a) and the AND gate with redundant inputs in Figure 7.12(b)

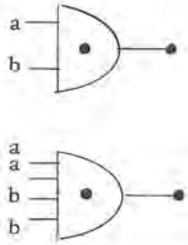


Figure 7.12. AND gates.

A "stuck at 1" at any single input to the AND gate in Figure 7.12(b) will not change the operation of the gate.

$$C_2 = aabb = ab$$

If one of the a inputs is "stuck at 1",

$$C_2 = 1abb = ab$$

Hence if independent redundant inputs are provided to an AND gate, the gate will mask single "stuck at 1" faults. The same is not true for "stuck at 0" faults. However, an OR gate masks "stuck at 0" but not "stuck at 1". Quadded logic (19) makes use of the fault masking properties of AND and OR gates. Consider the non redundant network in Figure 7.13.

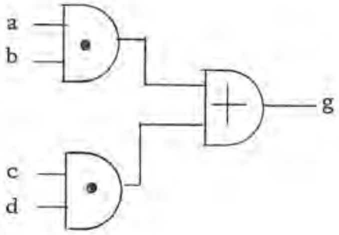


Figure 7.13. Logic network.

A quadded logic design for the network of Figure 7.13 is given in Figure 7.14. Note that each gate has been quadded and each input to a gate has been doubled.

If the redundant inputs to the AND gates in Figure 7.14 can be made independent, then the first level gates will mask all "stuck at 1" faults. For example, if a_1 is faulty and "stuck at 1" but a_2, a_3, a_4 are not faulty then the outputs of all 4 AND gates realizing ab will be correct. The assumption that multiple faults will not occur is necessary to insure proper operation. Some double faults are masked. A "stuck at 1" fault on both a_1 and b_1 or a_1 and a_2 will be masked, however a fault on both a_1 and a_3 will cause the outputs at gates A11 and A13 to be faulty.

A "stuck at 0" fault on a_2 will cause both A11 and A13 gate outputs to be "stuck at 0". If these gate outputs were inputs to the same OR gate input, that gate would operate improperly. Hence, the interconnection pattern is very

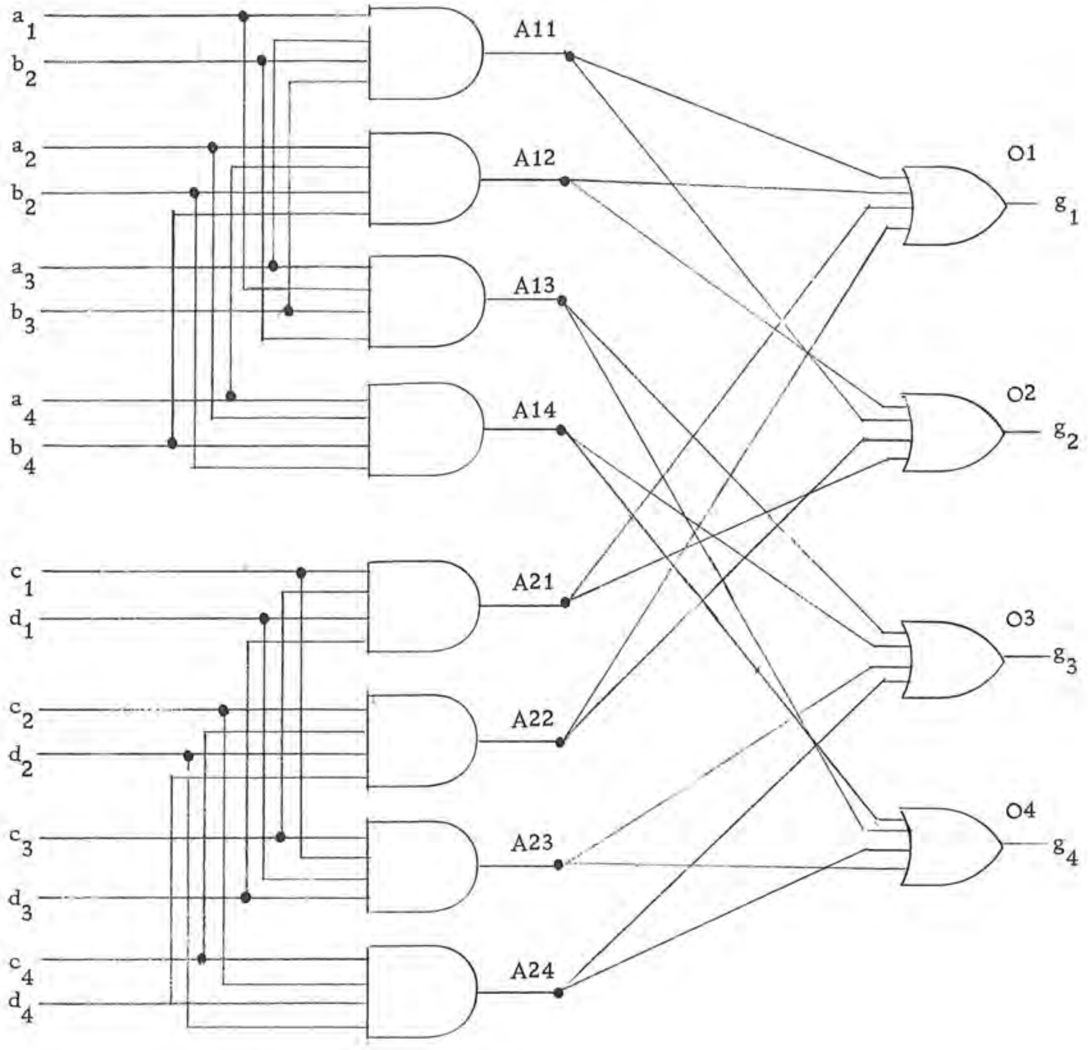


Figure 7.14. A quadded logic design for the network of Figure 7.13.

important. Single faults on the network inputs must not be allowed to cause double faults on inputs to gates in later stages. The interconnection pattern for the first level of Figure 7.14 can be represented as (13, 24). Inputs with subscripts 1 and 3 are fed to gates with subscripts An1 and An3, e.g., a_1 and a_3 are inputs to A11 and A13, c_2 and c_4 are inputs to A22 and A24. All single "stuck at 1" faults are masked. Double "stuck at 1" faults on input lines not fed to the same gates are masked. Double "stuck at 1" faults on input lines fed to the same gates and all single "stuck at 0" faults cause two AND gate outputs to be faulty.

Care must be taken to insure that pairs of AND gate outputs which may both be faulty due to single "stuck at 0" or double "stuck at 1" input faults are not inputs to the same OR gate. The interconnection pattern for the OR gate inputs in Figure 7.14 can be represented as (12, 34). For example A13, A14, A23 and A24 outputs are fed to gates 03 and 04. A double "stuck at 1" or a single "stuck at 0" input fault can cause faults only on gates An1 and An3 or An2 and An4. Since these gate inputs are never input to the same OR gate, all single "stuck at 0" and double "stuck at 1" input faults are masked from the gate outputs. Examination of Figure 7.14 will also show that all single "stuck at 0" AND gate output faults are also masked.

Quadded logic can be used for OR-AND networks in a similar manner. Application of Quadded logic design to AND-AND or OR-OR networks will mask only "stuck at 1" or "stuck at 0" faults respectively.

A NAND gate with redundant inputs will mask "stuck at 1" faults in the same manner as an AND gate. Furthermore, a "stuck at 0" input fault on a NAND gate causes a "stuck at 1" output fault. If Quadded logic techniques are used in NAND logic design, all faults will be masked except "stuck at 0" inputs to the last stage NAND gate. "Stuck at 0" input faults to a NAND, since they become "stuck at 1" output faults, will be masked by the next stage NAND gate. A similiar result can be obtained by using Quadded logic NOR gate design.

The problem of supplying independent redundant network inputs is partially solved by the fact that each Quadded logic network provides four independent outputs for each output in the non-redundant network. Hence, if there are several stages of logic networks to be designed, each stage will supply the next state with the appropriate independent inputs.

Quadded logic design can be used for sequential networks as well as combinational logic. For example, a R-S flip flop can be constructed with Quadded logic as shown in Figure 7.15.

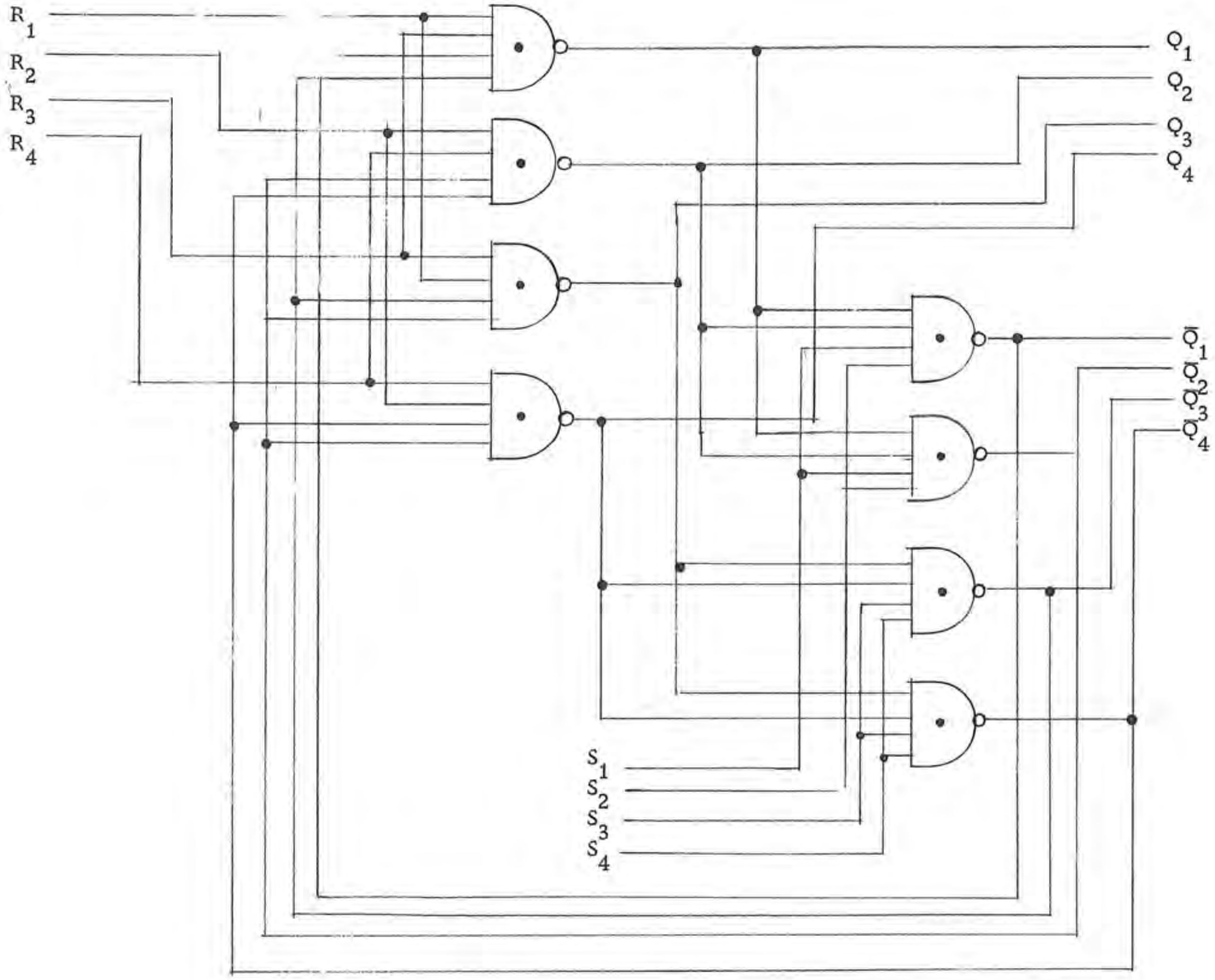


Figure 7.15. Quadded R-S flip flop.

Quadded logic is a special case of a general class of redundant logic called Interwoven Logic (12). In the general case, each k input gate is replaced with n^2 gates, each with nk inputs. It can be shown that all n - 1 order faults can be corrected (masked) using interwoven logic. For quadded logic n = 2 and each k input gate was replaced with $2^2 = 4$ gates with 2k inputs and all 2 - 1 = 1 (single) faults were masked.

Several other techniques for hardware redundancy have been suggested (5, 9, 20). However, standby sparing, TMR and Quadded Logic are typical examples. Some applications where hardware redundancy has been extensively used are discussed in section 7.4.

7.2 Coding Redundancy

Error correcting codes have been used extensively in communication, particularly digital communication. Shannon's (15) very well known work in noisy channels is the outstanding example. In a digital system the channel becomes the collection of data paths and storage devices such as registers, the memories and the various input/output links.

The approach to the design of error correcting codes for digital systems is much different than that for communication channels. The measures of efficiency are not the same, the nature of errors is different and the cost considerations more complex.

Although coding techniques have been the subject of much research, only very simple codes have actually been implemented. Coding techniques can be used to introduce redundancy into the logic or to information being processed. Signal redundancy will be the subject of this section, with emphasis on those codes which use combinational logic encoders and decoders.

Non-error correcting codes such as BCD, ASCII and EBCDIC are quite heavily used in digital systems. Simple error detecting codes such as parity codes and 2 out of 5 codes are fairly common. Other linear codes such as the Hamming codes and complex parity codes are not unknown. Cyclic codes, polynomial codes, etc., account for a great amount of research, but very little implementation.

The most important concept in designing error correcting codes is that of distance. In digital systems where binary codes are most common, distance depends on the length of the coded data in bits and the number of data encoded.

Definition: The Hamming distance between two encoded binary data vectors (words) is defined as the number of bits in which they differ.

For example,

$$\underline{a} = 10010001$$

$$\underline{b} = 01011101$$

the Hamming distance d $(\underline{a}, \underline{b})$ between $\underline{a}$ and $\underline{b}$ is

$$d\ (\underline{a}, \underline{b}) = 4$$

If two data words are separated by only a single bit (d(a, b) = 1) then an error in that bit cannot be detected.

Definition: The minimum distance of an encoding of a set of N data words is the minimum Hamming distance between any two words in the set.

Theorem: A minimum distance k coding for a set of N data words will detect $D \leq k - 1$ bit errors and concept $C \leq \frac{k-1}{2}$ bit errors.

$$k - 1 \geq C + D$$

A code, where the Hamming distance between any two words is two, is capable of causing single bit errors to be detected. A parity code is such a code where an extra bit is added to keep the number of 1's in each data word odd (or even). For example, let four items A, B, C, and D be coded with 2 bits as 00, 01, 10, and 11 respectively. An odd parity code would be

A = 001 B = 010 C = 100 D = 111

The right most bit is the parity bit. Note that the minimum distance for this code is 2. If a data word 000 appeared it would obviously be in error, since it corresponds to none of the allowed data. However, location of the bit in error is impossible, since 000 could be A, B or C with a single bit error. Another minimum distance two code is the 2 out of 5 code. Decimal numbers are encoded as follows:

	BCD Code	Parity (add)	2 of 5
0	0000	00001	00011
1	0001	00010	00101
2	0010	00100	00110
3	0011	00111	01001
4	0100	01000	01010
5	0101	01011	01100
6	0110	01101	10001
7	0111	01110	10010
8	1000	10000	10100
9	1001	10010	11000

Figure 7.16. Minimum distance two codes for decimal numbers.

Clearly to be an error detecting code, the minimum distance $\geqslant 2$. To obtain error correction, the distance between code words must be such that not only is it disernable that an error has occured but what the nature of the error is.

Consider the code where only 000 and 111 are allowed code words. A single error in the first word will cause it to be 001, 010 or 100. If double errors are unlikely, then reception of 010 would indicate that a single error had occurred in the second bit of 000. Hence, single error correction requires a minimum distance $\geqslant 3$.

The most well known single error correcting code is the Hamming code. The basic idea is to take a n bit uncoded data work and add p check bits so that the minimum distance is 3. Furthermore each bit position is given a decimal number. The p check bits are each parity bits over different subsets of the m data bits. When a coded data word is received, the check bits are recalculated and compared with the check bits in the code word. A binary number is developed from these comparisons which specifies the decimal number of the bit in error.

There are n data bits and p check bits, hence n + p possible single errors. The binary number indicating the error bit position is calculated from the check bits and is p bits long. The following inequality must be satisfied.

$$2^p \geqslant n + p + 1 \qquad \text{Eq. 7.17}$$

The p bit binary number can indicate 2^p error conditions. There are n + p single bit errors plus the no error case.

For example, let n = 4

$$2^p \geqslant 4 + p + 1$$

hence $p > 3$.

The information bits are x_3, x_5, x_6, x_7 and the parity check bits are calculated as follows.

$$p_1 = x_3 \oplus x_5 \oplus x_7$$

$$p_2 = x_3 \oplus x_6 \oplus x_7$$

$$p_4 = x_5 \oplus x_6 \oplus x_7$$

Let $x_3x_5x_6x_7 = 1001$

$$p_1 = 1 \oplus 0 \oplus 1 = 0$$

$$p_2 = 1 \oplus 0 \oplus 1 = 0$$

$$p_4 = 0 \oplus 0 \oplus 1 = 1$$

the coded data word is

$$p_1p_2x_3p_4x_5x_6x_7 = 0011001$$

At the receiving end a binary word $c_1c_2c_3$ is calculated as follows

$$c_1 = p_1 \oplus x_3 \oplus x_5 \oplus x_7$$

$$c_2 = p_2 \oplus x_3 \oplus x_6 \oplus x_7$$

$$c_3 = p_4 \oplus x_5 \oplus x_6 \oplus x_7$$

The errors indicated are tabulated in Figure 7.17.

$c_1c_2c_3$	error
000	no error
001	error in p_1
010	error in p_2
011	error in x_3
100	error in p_4
101	error in x_5
110	error in x_6
111	error in x_7

Figure 7.17. Hamming code error correction.

Let the received word be the correct word

$$p_1p_2x_3p_4x_3x_6x_7 = 0011001$$

and calculate

$$c_1 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

$$c_2 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

$$c_3 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$$

From the table $c_1c_2c_3 = 000$ implies no error. Let the received word be (error in x_5)

0011101

$$c_1 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$c_2 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

$$c_3 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$$

From the table $c_1c_2c_3 = 101$ implies an error in x_5.

In applications where data is stored in blocks, parity code and Hamming code check bits can be computed for bit slices as well as word slices.

	11010	1	
	10001	0	
data	10110	1	Parity on rows (word slice)
	10000	1	
	11111	1	
	01001	0	
Parity over cols (Bit slice)	11011	0	

Figure 7.18.

In Figure 7.18 a simple parity code can be used for error correction by generating parity bits over both columns and rows of stored data. A Hamming code used in same manner could detect triple errors and correct double errors.

There are many other codes, however these examples are the most common. References to other codes can be found at the end of the chapter.

Another class of codes which are useful in digital systems design are arithmetic codes. Arithmetic codes fall into two categories, separate and non-separate. Parity and Hamming codes are separate codes in that the data is present in the coded word in its original form.

The AN or product code is an example of a non-separate code. The coded word is computed by taking the product of a constant A and the data N. For example, $A = 3$, $x = 0101$, the AN coded word is

$$Ax = 001111$$

Obviously the choice of A as an odd number makes all binary AN codes have minimum distance ≥ 2. The problem is to determine an A which yields a code with minimum distance > 2. The distance obtained with AN codes depends on the range of data N.

Theorem: For any A and radix r such that A and r are relatively prime, if N is restricted to range

$$O \leq N < M_r (A, d)$$

Where $M_r(A, d)$ is the smallest number whose product with A has weight less than d in radix r, then minimum distance $\geq$ d.

The unit $M_r (A, d)$ must be found by an exhaustive procedure. For example, let A = 11 (1011_2)

$$N = 0 \quad AN = 0 = 000000_2$$

$$N = 1 \quad AN = 11 = 001011_2$$

$$N = 2 \quad AN = 22 = 010110_2$$

$$N = 3 \quad AN = 33 = 100001_2$$

Note that the weight of 22 in base 2 is 3, but the weight of 33 in base 2 is 2. Hence $M_2(11, 3) = 3$, since 3 is smallest number having weight less than 3 in radix 2. This means that A = 11 can be used for an error correcting AN code only for numbers in the range $0 < N < 3$. The table in Figure 7.19 lists $M_2 (A, 3)$ for several A.

A	$M_2 (A, 3)$	Approximate redundancy
11	3	4
13	5	4
23	89	5
29	565	5

Figure 7.19. M_2 (A 3) for several AN codes.

Note that all eight bit binary numbers are in the range $0 \leq N < 256$ and from Figure 7.19, A = 29 will yield an AN code with minimum distance ≥ 3 for numbers in range $0 \leqslant N < 565$. The redundancy for AN codes is approximately $\log_2 A$. For $A = 29$, the resulting AN code will have 5 redundant bits. A Hamming code for 8 bit data would require at least 4 redundant bits.

A common separate arithmetic code is the residue code. A code word consists of an ordered pair $(N, R_N 0$ where

$$R_N = N \bmod A \qquad \text{Eq. 7.18}$$

For example, $N = 5$, $A = 3$

$$R_N = N \bmod A = 2$$

The residue code is $(N, R_N) = 10110$. Again it is obvious that if A is odd, residue codes have minimum distance $\geqslant 2$. A residue code behaves much like an AN code. It follows that if $R_N = N \bmod A$, then $R_N = (2^m \cdot N) \bmod A$.

$$\begin{aligned} 2^m \cdot N &= qA + R_N \\ &= (q + 1)A - (A - R_N) \\ &= (q + \underline{\lambda})\,A - \bar{R}_N \end{aligned}$$

Hence

$$2^m N \quad \bar{R}_N = (q + 1)A \qquad \text{Eq. 7.19}$$

If the value of m in Eq. 7.19 is chosen to be the smallest integer greater than $\log_2 A$, the term $2^m N + \bar{R}_N$ is actually a residue code using the complement of the residue, $(N_1 \bar{R}_N)$. Calculations for the range of N for which a given A yields a minimum distance 3, error connecting residue code is similar to the calculation for AN codes.

The next question to be answered deals with the impact of the various codes on the hardware. Separate codes can employ separate checking circuitry. For example, a parity check on iinterregister data transfers might look like Figure 7.20.

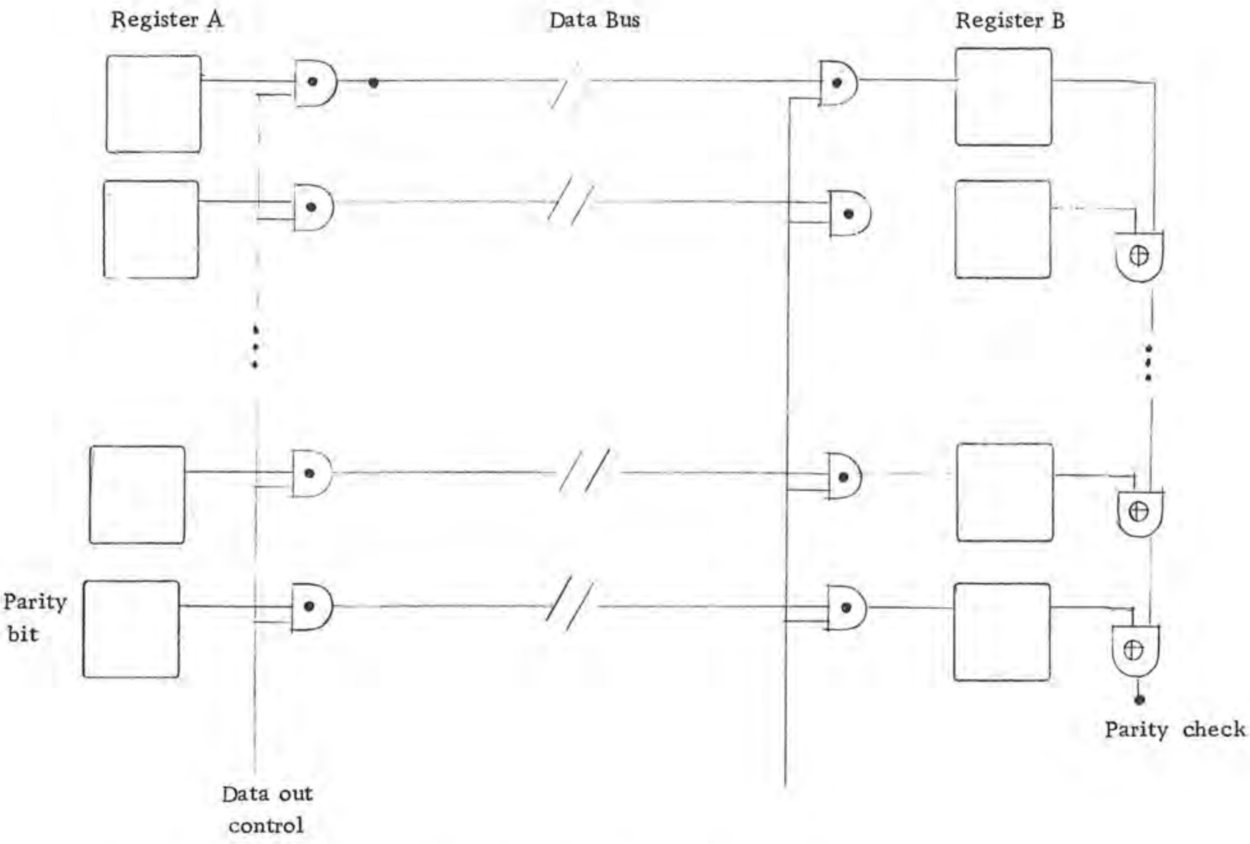


Figure 7.20. Parity checked data transfer.

Parity codes can also be implemented in arithmetic logic. Recall that a full adder can be represented by the expressions

$$S_i = a_i \oplus b_i \oplus c_{i-1} \qquad \text{Eq. 7.20}$$

$$c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1} \qquad \text{Eq. 7.21}$$

Where a_i, b_i are the bits of the n bit words $\underline{a}$ and $\underline{b}$ being added, c_{i-1} is the carry in, s_i is sum bit and c_i is the carry out. If $\underline{a}$ and $\underline{b}$ have a parity check bit,

$$p_a = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

$$p_b = a_1 \oplus b_2 \oplus \cdots \oplus b_n$$

the parity bit on the sum is

$$\begin{aligned}p_s &= s_1 \oplus s_2 \oplus \cdots \oplus s_n \\ &= (a_1 \oplus b_1 \oplus c_0) \oplus (a_2 \oplus b_2 \oplus c_1) \oplus \cdots \oplus (a_n \oplus b_n \oplus c_{n-1}) \\ &= (a_1 \oplus a_2 \oplus \cdots \oplus a_n) \oplus (b_1 \oplus b_2 \oplus \cdots \oplus b_n) \oplus (c_0 \oplus c_1 \\ &\qquad \oplus \cdots \oplus c_{n-1}) \\ &= p_a \oplus p_b \oplus (c_0 \oplus c_1 \oplus \cdots \oplus c_{n-1})\end{aligned}$$

A parity checked adder design is shown in Figure 7.21.

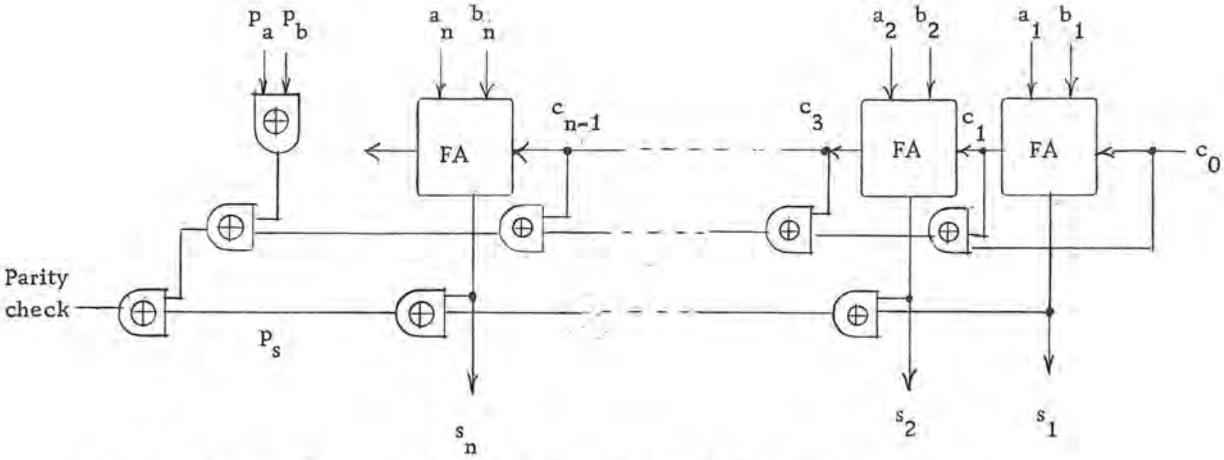


Figure 7.21. A parity checked parallel adder.

In the case of AN codes, note that

$$AN_1 + AN_2 = A(N_1 + N_2) \qquad \text{Eq. 7.22}$$

Hence the sum (or difference) of two AN coded data should be AN coded. An AN coded adder with check circuitry is shown in Figure 7.22.

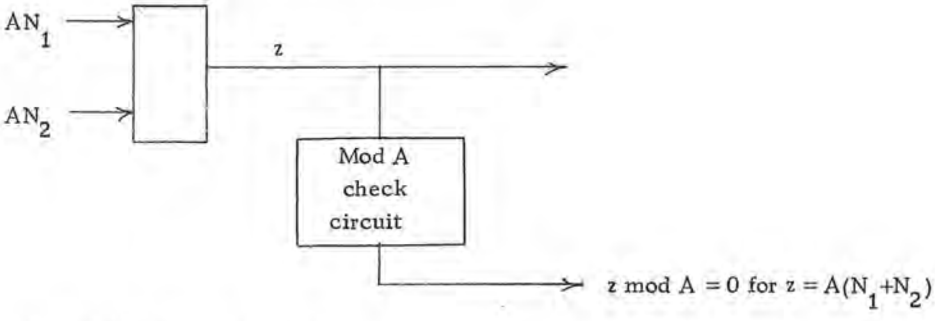


Figure 7.22. AN coded adder.

For AN coded multiplication, two divisions by A are necessary. A product should be $A^2N_1N_2$ and must be divided by A to obtain the proper result AN_1N_2. Furthermore, the output must be divided again by A to check for zero residue.

Since residue codes are separate codes, the check circuit must perform the same operation modulo A as the arithmetic unit for the data.

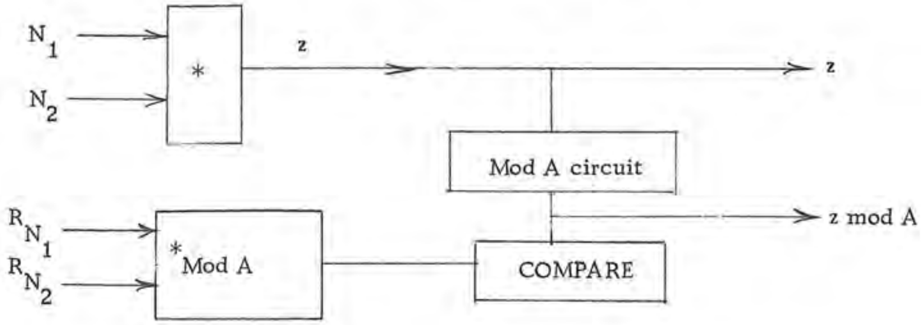


Figure 7. 23. Residue coded arithmetic unit.

Figure 7. 23 shows a separate modulo A arithmetic unit for the residues R_{N_1} and R_{N_2}, since

$$(N_1 * N_2) \bmod A = (R_{N_1} * R_{N_2}) \bmod A,$$

the check circuit merely compares the residue of the arithmetic unit output with that calculated by the modulo A arithmetic unit on the residues.

7.3 Fault Diagnosis

Fault diagnosis techniques can be divided into two categories, algebraic methods and path sensitization. Examples of the former are the Boolean Difference (16), Poages method (13), Armstrong's ENF (1) and the SPOOF (7). The most well known path sensitization procedure is the Roth D Algorithm (14).

The Boolean difference of a function, representing some logic network, with respect to a variable yields a Boolean expression which defines the input required to make the output sensitive to a fault on the variable.

Definition: The Boolean Difference of a function $f(x_1, \cdots, x_n)$ with respect to variable x_i is

$$D_i f = \frac{df(x_1) \cdots, x_n)}{dx_i} = f(x_i, \cdots x_{i-1}, 0, x_{i+1}, \cdots x_n) \oplus$$

$$f(x_1, \cdots, x_{i-1}, 1, x_{i-1}, \cdots, x_n) \qquad \text{Eq. 7.23}$$

Theorem: Let $f(x_1, \cdots, x_n)$ represent the logic network N. Let $D_i f$ be the Boolean Difference of $f(x_1, \cdots, x_n)$ with respect to variable x_i

1. If $D_i f = 0$, a change in x_i will never cause a change in f.
2. If $D_i f = 1$, a change in x_i will always cause a change in f.
3. If $D_i f = g(x_1, \cdots x_{i-1}, x_{i+1}, \cdots, x_n)$ then a change in x_i will cause a change in f iff $g(x_1, \cdots x_{i-1}, x_{i+1}, \cdots, x_n) = 1$.

This theorem can be used to determine the set of inputs for which a fault on x_i is testable. If there exists an input for which a change in x_i will cause a change in f, then for example a stuck at 1 fault on x_i can be tested by applying a 0 to x_i and determining if the output changes from the expected value.

Consider the gate network in Figure 7.24 below.

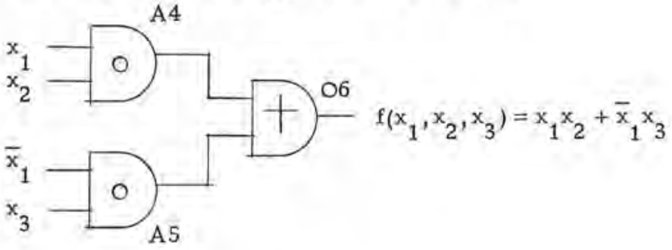


Figure 7.24. A gate network.

The Boolean Differences are

$$D_1 f = (0X_2 + 1X_3) \oplus (1X_2 + 0X_3) = X_2 \oplus X_3 \qquad \text{Eq. 7.24}$$

$$D_2 f = (X_1 \cdot 0 + \overline{X}_1 X_3) \oplus (X_1 \cdot i + \overline{X}_1 X_3) = X_1 \qquad \text{Eq. 7.25}$$

$$D_3 f = (X_1 X_2 + \overline{X}_1 \cdot 0) \oplus (X_1 X_2 + \overline{X}_1 \cdot 1) = \overline{X}_1 \qquad \text{Eq. 7.26}$$

The network output f is sensitive to changes in X_1 if $X_2 \oplus X_3 = 1$. For example, the input 001 tests for a "stuck at 1" on X_1. The network output should be f(001) = 1, however, if X_1 is "stuck at 1", the output is f(101) = 0. The wrong output indicates X_1 may be "stuck at 1". The inputs 001 and 010 test "stuck at 1" on X_i, the inputs 101 and 110 test for "stuck at 0" on X_1.

The network of Figure 7.24 is sensitive to changes in X_2 if $X_1 = 1$. Hence 100 and 101 test "stuck at 1" on X_2. Note that 101 also tested "stuck at 0" on X_1. The input 101 detects a fault but cannot distinguish whether it is a "stuck at 0" on X_1 or a "stuck at 1" on X_2.

In a non redundant network the presence of any single fault can be detected by applying all possible inputs and checking the output. The Boolean differences can be used to find a minimal set of inputs which detect all faults.

A fault table for the network of Figure 7.24 is given in Figure 7.25. A "1" in column X_1 indicates the input 001 detects a "stuck at 1" on the variable X_1. Note that entries for each gate output are also given. The Boolean differences are

$$f = A4 + \overline{X}_1 X_3 = X_1 X_2 + A5 = 06$$

$$D_4 f = X_1 + \overline{X}_3$$

$$D_5 f = \overline{X}_1 + \overline{X}_2$$

$$D_6 f = 1$$

Hence, a "stuck at 0" on the output of A1 will be tested if $A1 = X_1 X_2 = 1$ and $X_1 + \overline{X}_3 = 1$. The inputs 110 and 111 satisfy these constraints.

Input	Variable gate	X_1	X_2	X_3	A4	A5	06
	000		1		1	1	1
	001	1	1			0	0
	010	1	0		1	1	1
	011		0			0	0
	100			1	1	1	1
	101	0		0	1	1	1
	110	0		1	0		0
	111			0	0		0

Figure 7. 25. Fault table for the network of Figure 7. 24.

Using a fault table, as shown above, determination of a minimum set of inputs which detect all single faults is an example of the covering problem. All the techniques for finding a minimum sum of prime implicants can be used. For the example in Figure 7. 25, three minimal sequences, (001, 010, 101, 110), (001, 011, 101, 110) and (001, 010, 110, 111) will detect all single faults in the network in Figure 7.

The problem of determining an input sequence which will locate or distinguish a particular fault is much more difficult. For example, a "stuck at 1" fault on the output of either A4 or A5 is indistinguishable from a "stuck at 1" fault on the OR gate output. The sequence 000, 001, 011, 100, 101, 111 will distinguish all input faults, but not internal faults. An incorrect output for inputs 000, 001 and correct outputs for all other inputs indicates X_1 is "stuck at 1".

As the complexity of the network grows, algebraic techniques like the Boolean Difference become more and more unwieldy. Algebraic techniques are not easily programmed and for complex networks, the computer is a very necessary tool.

The Roth D Algorithm is a commonly used procedure for deriving fault diagnosis tests for logic network. As an introduction to the algorithm, consider the network in Figure 7. 26.

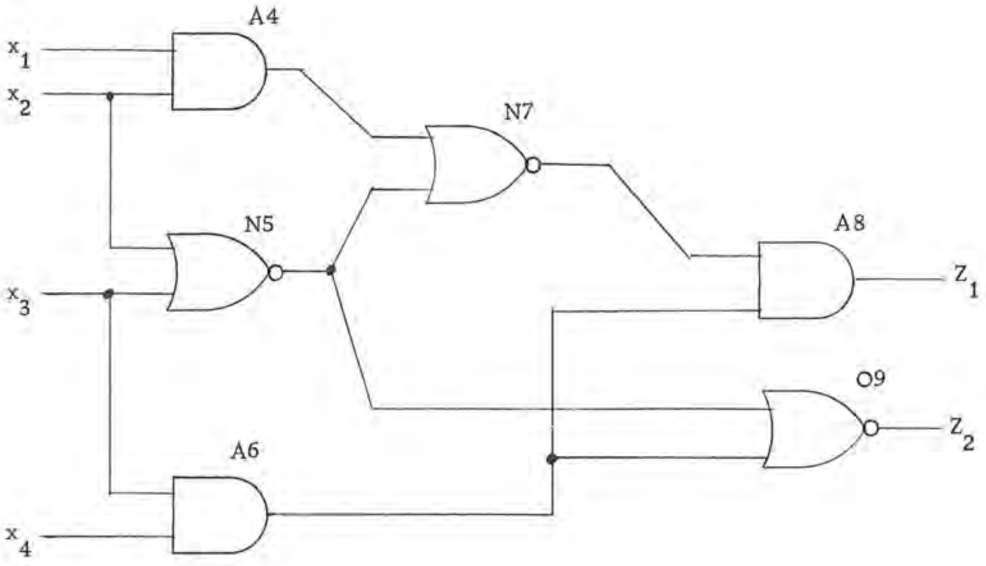


Figure 7.26. Combinational logic network.

An input which will test a particular fault on a network input or gate output can be determined by path sensitization. For example, suppose X_1 is "stuck at 1". First a path from the fault to an output must be determined which is the path via gates A4, N7 and A8. The next step is to sensitize the path to the particular fault. The output of A4 is sensitive to a fault on X_1 if $X_2 = 1$. The output of N7 is sensitive to the output of A4 and thus the fault on X_1 if the output of N5 is 0. Finally, A8 is sensitive to the output of N5 if the output of A6 = 1. Once having identified the gate inputs necessary to sensitize the path, the technique proceeds backwards making the appropriate assignments.

Beginning at gate A8, gate A6 was required to have an 1 output, hence $X_3X_4 = 1$. At gate N7, N5 = 0 requiring that $\overline{X_2 + X_3} = \overline{X}_2\overline{X}_3 = 0$. The input combination $X_1X_2X_3X_4 = 0111$ satisfies all requirements and sensitizes the path X_1, A4, N7, A8 to a "stuck at 1" fault on Y_1.

Of course, due to the network configuration, it is not always possible to sensitize a given path. For example, assume a "stuck at 0" fault on the output of gate N5. There are two paths to an output, N5, N7, A8 and N6, 09. To make the latter path sensitive, requires that the A6 gate output be 0. This implies that $X_3X_4 = 0$. The inputs to gate N5 must be assigned $\overline{X}_2\overline{X}_3 = 1$ to test a "stuck at 0" on N5. Any input with $X_2 = X_3 = 0$ will sensitize this path. Attempts to sensitize path N5, N7, A8 result in a contradiction. Proceeding forward along the path requires that A4 = 0 and A6 = 1. Retracing we find that A6 = 1 requires $X_3 X_4 = 1$ and A4 = 0 implies $X_1 X_2 = 0$. Finally to test "stuck at 0" on N5, $\overline{X}_2\overline{X}_3 = 1$. No input combination will satisfy these constraints, because X_3 must be 1 as input to gate A6 and 0 as input to gate N5. Hence a "stuck at 0" on N5 cannot be tested via N5, N7, A8.

Manually implementing path sensitizing techniques is laborious. The Roth D Algorithm is a formalization of these techniques which lends itself well to programming.

The D algorithm uses a Boolean n-cube representation for the gates and the network. Three important types of n-cubes are called singular covers, primitive D cubes and propagation D cubes.

The singular cover is derived by expanding the network function $f(x_1, \cdots, x_n)$ to include the outputs as well as the inputs. A new function $g(x_1, \cdots, x_n, Z)$ is created, where Z is the network output.

$$g(n_1, \cdots, x_n, Z) = Zf(x_1, \cdots, x_n) + \overline{Z}\,\overline{f}(x_1, \cdots, x_n)$$

For a two input NAND gate

$$f(x_1, x_2) = \overline{x_1 x_2} = \overline{x}_1 + \overline{x}_2$$

$$g(x_1, x_2, Z) = x_1 Z + \overline{x}_2 Z + x_1 x_2 \overline{Z}$$

The singular cover represents the possible "states" of the network in terms of inputs and outputs and is represented using the Quine-McCluskey notation for the prime implicants of $g(x_1, \cdots, x_n, Z)$. For the example above,

0-1
-01
110

is the singular cover. All gate input/output assignments must satisfy the singular cover. For instance 111 is an impossible assignment of the inputs and output of a NAND gate.

A primitive D cube represents the necessary inputs to cause the output of a gate to be sensitive to a particular fault. The term 01D represents a primitive D cube for a "stuck at 1" fault on the x_1 input to the NAND gate above. The D represents that the output will change due to a fault on x_1, $x_1x_2 = 01$ is the input required to make Z sensitive to the fault.

A propagation D cube represents the sensitivity of outputs to changes in inputs. The propagation D cubes for a NAND gate are $D1\bar{D}$ and $1D\bar{D}$, indicating that if either input is set to 1, the output will propagate (the inverse) of any change on the other input.

The D algorithm starts by calculating the singular cover for each gate. The primitive D cube for a particular fault is derived from the singular cover for the faulty gate. The propagation D cubes are obtained from the singular covers of all remaining gates. The next step is called "D drive" and consists of intersecting the primitive D cube and successive propagation D cubes along the path. This is just the forward path sensitizing procedure. If a D drive to an output is successful, a consistency check is made using the singular covers in the same fashion as manual path sensitization.

As an example consider the simple network of Figure 7.24. The singular covers are

	$X_1X_2A_4$		X_1X_3A5		A4A506
A4:	111	A5:	011	06:	1-1
	0-0		1-0		-11
	-00		-00		000

Assume a "stuck at 1" on x_2 input to A4, the primitive D cube is

$$X_1X_2A4 = 10D$$

indicating a "stuck at 1" on X_2 will cause incorrect output for A4. The propagation D cubes for A5 and 06 are

	X_1X_3A5		A4A506
A5:	0DD	06:	0DD
	$D1\bar{D}$		D0D

The primitive D cube for the fault can be intersected with a propagation D cube for 06

$$(X_1X_2A_4 = 10D) \quad (A4A506 = D0D)$$

$$= X_1X_2A4A506 = 10D0D$$

Recall that the singular covers represent the only possible input/output assignments. The consistency procedure, in effect, examines the singular covers to determine if a consistent assignment can be made on the inputs which will propagate the fault to output. For this example, the singular cover for A5 ($X_1X_3A5 = 1 - 0$) is consistent with the D drive above and results in the following assignment to test for "stuckat 1" on X_2 input to A4:

$$X_1X_2X_3A4\ A5\ 06 = 10 - D0D$$

This can be interpreted to mean that an input assignment 101 or 100 will cause the fault on X_2 to be propaged via path A4, 06.

During the logic design phase of any digital system, a simulation of the design can be performed to derive a set of diagnostics to test the system when operable. Simulation can be classified as manual, physical or digital.

In manual simulation, the digital system is partioned into functional blocks. Test sets for each block are derived using algebraic or path sensitizing techniques. Using the test sets, a diagnostic program can be written which exercises as fully as possible all the logic circuits in the functional blocks. In complex systems, a major problem is the location of suitable test points so meaningful data on the response of the system to the diagnostic routines can be obtained. Manual simulation is slow, subject to human error and not responsive to design changes.

An alternative approach is to partition the system into units as before, but working with the actual system or a prototype, faults can be physically inserted. A unit is replaced with a faulty unit or more often another system which can simulate the unit under all fault conditions. The system under test is then allowed to run its normal work load. Data is taken at test points and is used to construct diagnostic routines.

The entire logic for the system can be simulated on a digital computer. This has the drawback that only logic faults can be simulated, but has the advantage of speed and high response to design changes. Digital simulation is very useful in predicting timing problems in the logic which may cause transient faults.

7.4 Comments - Problems - References

A good test on fault detection is Friedman and Menon (8). Another book by Chang, Manning and Metze (6) has excellent discussion on simulation. The major reference on the Boolean Difference is the paper by Sellers, Hsaio and Bearnson (16). The original IBM report by Roth (14) on the D algorithm contains a complete APL description of the algorithm. Peterson (11) has written a well known text on coding. Several papers by Avizienis (2, 4) detail arithmetic code implementations and include a study of the cost effectiveness of codes.

The initial paper which led to TMR was written by Von Neuman (20). An excellent book edited by Wilcox and Mann (21) includes most of the early work on hardware redundancy.

Interested readers might examine the two special issues of the IEEE Transaction on Computers concerning fault tolerant computing (22, 23). Short has published two very complete literature surveys (18, 19) on progress in fault tolerance.

REFERENCES

1. Armstrong, D., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Sets," IEEETC, Vol. EC-15, No. 1, pp. 66-73, February, 1966.

2. Avizienis, A., G. Gilley, F. Mathur, D. Rennels, J. Rohr, D. Rubin, "The STAR (Self Testing and Repair) Computer. An Investigation of the Theory and Practice of Fault Tolerant Computer Design," IEEETC, Vol. C-20, No. 11, pp. 1312-1321, Nov. 1971.

3. Avizienis, A., "Design of Fault Tolerant Computers," FJCC 1967, AFIPS Conference Record Vol. 31, pp. 733-743.

4. ________________ "Arithmetic Error Codes: Cost and Effectiveness Studies for Applications in Digital System Design," IEEETC, Vol. C-20, No. 11, p. 1322-1331, November 1971.

5. Ball, M. and F. Hardie, "Majority Voter Design Considerations for a TMR Computer", Computer Design, pp. 100-104, April 1969.

6. Chang, H., E. Manning, G. Metze, Fault Diagnosis of Digital Systems, Wiley-Interscience, New York, 1970.

7. Clegg, F., "The SPOOF-A New Technique for Analyzing the Effects of Faults on Logic Networks," Digital Sys. Lab Tech. Report 11, SU-SEL-70-073, 1970.

8. Friedman, A., P. Menon, Fault Detection in Digital Circuits, Prentice-Hall. 1971.

9. Klashka, "Two Contributions to Redundancy Theory", 5th Symposium on Switching and Automata Theory, October 1967.

10. Mathur, F. P., "On Reliability Modeling and Analysis of Ultra-Reliable Fault-Tolerant Digital Systems", IEEETC Vol. C-20, P. 1376-1382, November 1971.

11. Peterson, W., Error Correcting Codes, Wiley, New York, 1961.

12. Pierce, W., Failure Tolerant Computer Design, Academic Press, New York, 1965.

13. Poage, J. and E. McCluskey, "Derivation of Optimum Test Sequences for Sequential Machines," 5th Symposium on Switching and Automata Theory, p. 121-152, 1964.

14. Roth, J. P., "Diagnosis of Automata Failures: a Calculus and a Method," IBMJRD, pp. 278-291, July 1966.

15. Shannon, C. and W. Weaver, The Mathematical Theory of Communication, Univ. of Illinois Press, Urbana, Ill. 1949.

16. Sellers, F., M. Hsaio, L. Bearnson, "Analyzing Errors with the Boolean Difference," IEEETC Vol. C-17, No. 7, pp. 676-683, July 1968.

17. Short, R., "The Attainment of Reliable Digital Systems Through the Use of Redundancy - A Survey," IEEE Computer Group News, Vol. 2, p. 2-17, March 1968.

18. Short, R., and J. Goldberg, "Soviet Progress in the Design of Fault-Tolerant Digital Machines," IEEETC, Vol. C-20, p. 1337-1353, November 1971.

19. Tyron, J., "Quadded Logic", in Redundancy Techniques for Computing Systems, Spartan Books, Washington, D. C., 1962.

20. Von Neuman, J., "Probalistic Logics and the Synthesis of Reliable Organisms for Unreliable Components," Automata Studies, Princeton University Press, Princeton, N. J., 1956.

21. Wilcox and Mann, Redundancy Techniques for Computing Systems, Spartan Books, Washington, D. C., 1962.

22. "Special Issue on Fault Tolerant Computing", IEEETC, Vol. C-20, No. 11, November 1971.

23. "Special Issue on Fault-Tolerant Computing," IEEETC, Vol. C-22, No. 3, March 1973.

PROBLEMS

1. Define or describe:

   Fault tolerant
   Fail Safe
   Fail Soft
   Roll back
   MTF
   "Stuck-at" Faults
   Fault Detection
   Fault Diagnosis
   Redundancy
   Active Spares
   Standby Spares
   TMR
   Radial Logic
   Quadded Logic

2. Assuming exponential reliability $R_o = e^{9\lambda t}$ for a single unit with failure rate λ., MTF = $1/\lambda$ sketch the reliability curves for:

   a. One operating unit + 1 active standby

   b. One operating unit + 1 passive standby

   c. TMR w/perfect voter

   d. Hybrid NMR-TMR + 2 spares

   e. TMR w/triplicated voter (non-perfect)

3. Quadded, Radical and Interwoven logic make use of a fault masking capability inherent in certain gates. Explain.

4. A full adder can be described by

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

$$c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1}$$

   Design a full adder with Quadded logic.

5. Repeat 4 using TMR.

6. What does "minimum distance" imply about the error correction/detection capability of a linear code.

7. Define AN, and Residue codes. Show how each can be implemented for an adder unit.

8. What is the maximum N for which a AN code using A =11 is a minimum distance 3 code? What is the redundancy?

9. Contrast the algebraic type techniques (Boolean Diff.) and the path sensitizing methods (Roth D) for fault diagnosis in terms of their ease of application, generality and range of application.

10. Derive the Boolean Differences $D_{a_i} c_i$, $D_{b_i} c_i$, $D_{c_{i-1}} c_i$ for the carry logic of a full adder.

11. Derive a fault table and a minimum sequence for detecting all faults in 2 two stage AND-OR realizations of the carry logic of a full adder.

12.

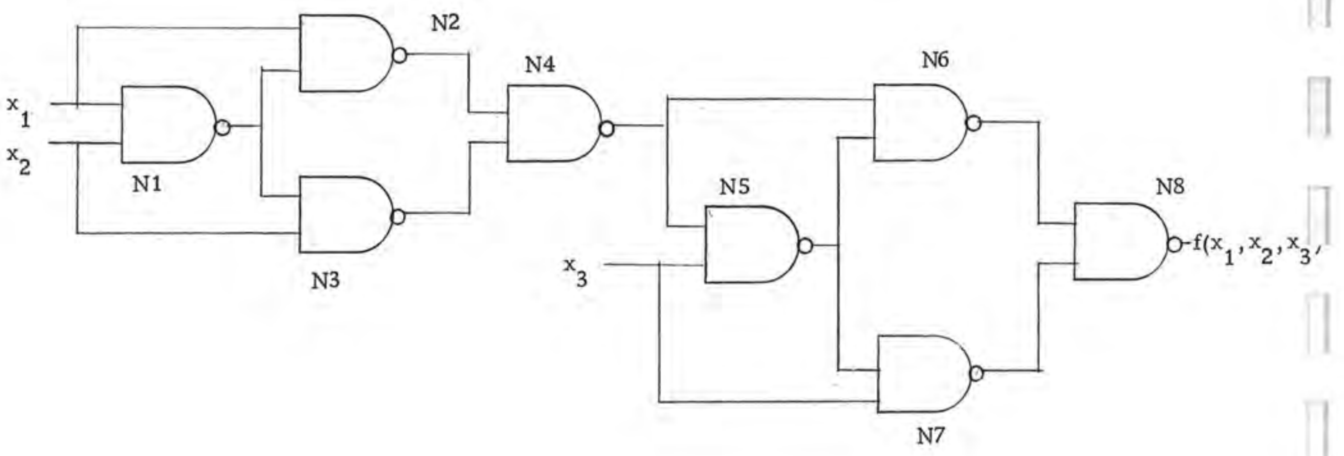


Figure P7. 1.

Use path sensitizing to determine the input required to test a sa1 on the input x_1 to N_2 in Figure P7. 1.

8 The Impact of New Technology on Logic Design

The subject of this chapter will be an abreviated look at how advances in semiconductor technology have effected the job of the logic designer. Particular emphasis will be placed on Read Only Memory design and the promise of large scale integration.

Many of the classical techniques for logic minimization are based on relay or transistor logic. The emphasis today and tomorrow will not be on minimizing the logic element as much as reducing fan in/out and simplifying the interconnection patterns. More attention will be given to increasing the system modularity through the use of standardized functional units.

8.1 Read Only Memory Design

Obviously any n variable Boolean function can be synthesized by a table lookup procedure. A Boolean function can be represented by a truth table. If the minterms of a function are used to represent memory addresses and the memory contents are the value of the minterm in a given function, the function can be realized by having the input cause a readout of the corresponding memory location.

For example, a BCD to 7 segment decoder driver can be synthesized by the network block diagrammed in Figure 8.1.

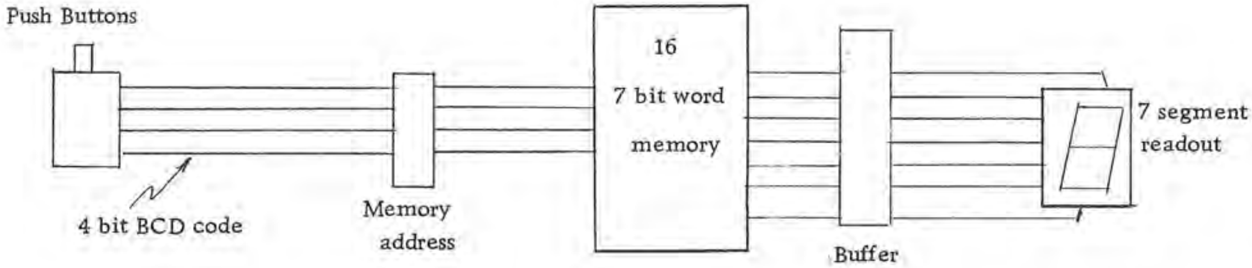


Figure 8.1. A BCD - 7 segment decoder.

This concept is not new, but became practical only with the advent of inexpensive Read-Only Memories (ROM's). Beyond the obvious application above, ROM's can also be used to realize trigonometric logrithmic and other functions which are complex to synthesize with digital logic. This can be done with table look up and interpolation or through ROM storage of routines to be implemented on a companion arithmetic unit. The well known Hewlett-Packard HP35 hand calculator uses 3 ROM's of 256, 10 bit instructions each and performs an iterative psuedo division/multiplication algorithm with an arithmetic logic chip for the calculation of transcendental functions.

ROM's can implement the control function of a digital system as in HP-35. The concept used is that of microprogramming. A microprogram consists of a series of micro instructions. The bits of micro instructions correspond to control points within the system and can be used to transfer data, initiate arithmetic operations and test conditions. Rather than design a sequential network to sequence a series of arithmetic operations, the necessary micro instructions can be stored in a ROM and the read out cycled to provide the necessary timing.

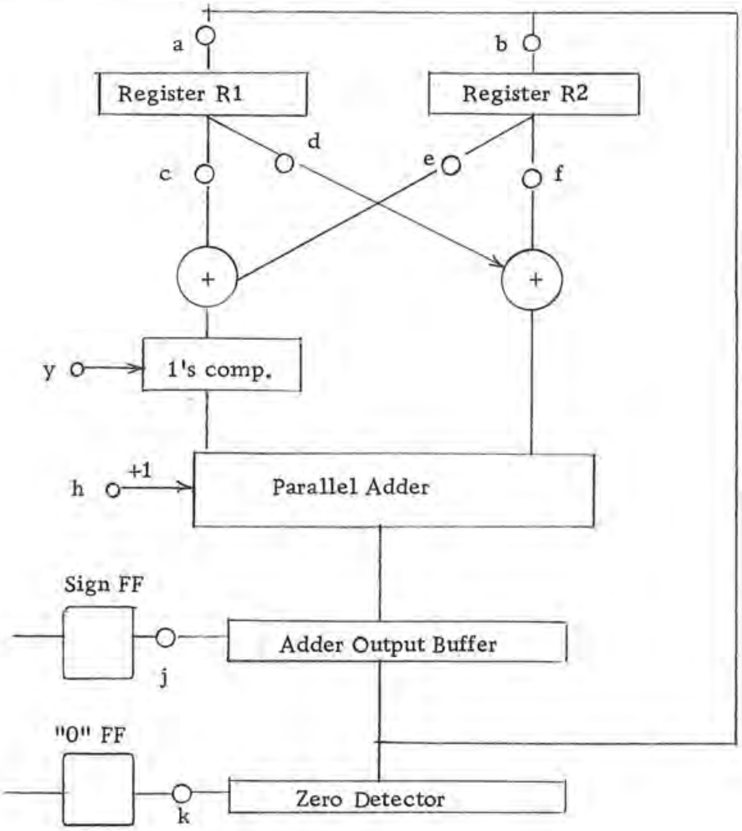


Figure 8.3. Arithmetic unit.

There are 10 control points in the simple arithmetic unit of Figure 8.3. Some control data transfers, e.g., control point f controls the transfer of R2 to the adder, some control operations, e.g., control point g can complement the left side input to the adder and some test conditions, e.g., point j activates the sign test flip flop. A 10 bit micro instruction word can be used to set the control points.

a	b	c	d	e	f	g	h	i	k	next m-instruction

Figure 8.4. Micro instruction format.

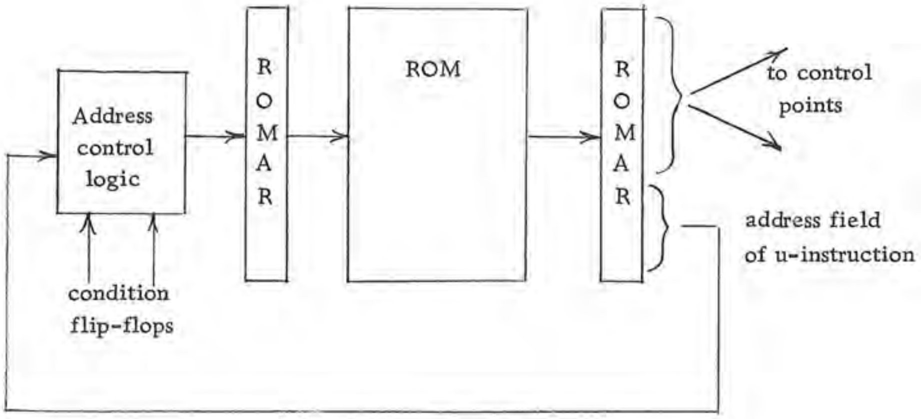


Figure 8.5. ROM organization.

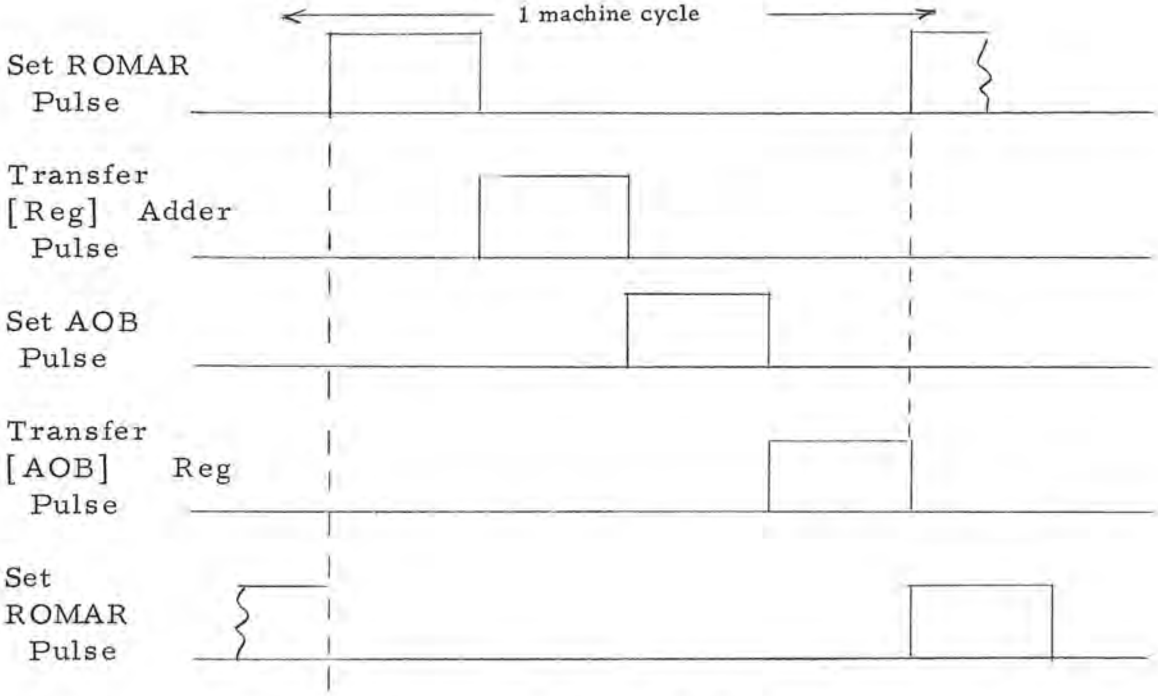


Figure 8.6. Timing for microprogrammed control for Figure 8.5.

The five major cycles of the ROM control for the arithmetic unit of Figure 8.3 are shown in Figure 8.6. The first pulse causes a micro instruction (format given in Figure 8.4) to be read into the ROM Data Register. The 10 control bits activate the control points, the remaining bits (next micro instruction address) are fed to the address control logic. On the second pulse data from the registers are transferred to the adder. The next pulse causes the adder output to be stroked into the output buffer (AOB). The forth pulse transfers data back to the registers and the final pulse initiates a read of the next micro instruction.

For example, the micro instruction 1001101100XXX causes the contents of R1 minus R2 to be stored back in R1. Subtraction is by two's complement arithmetic.

Multiplication can be performed using this simple ROM controlled arithmetic unit if a few assumptions are made and one register R3 is added. Assume multiplier is in R2 and the multiplicand in R3. Assume both are positive and that the product will not overflow R1. Let 1 be the control point for a data transfer from the R3 to the right hand OR gate adder input.

The algorithm is the simple multiplication by successive addition. The microprogram is given in Figure 8.7. A flow chart is given in Figure 8.8.

← ROM Storage →

	a	b	c	d	e	f	g	h	j	k	l	branch condition		address	
000	1	0	1	1	0	0	1	1	0	0	0	0	0	--	0 → R1
001	0	1	0	0	1	0	1	0	0	0	0	0	0	--	R2 → R2
010	0	1	0	0	0	1	0	1	0	1	0	0	1	101	R2 + 1 → R2 R2 = 0?
011	1	0	1	0	0	0	0	0	0	0	1	0	0	--	R1 + R3 → R1
100	0	0	0	0	0	0	0	0	0	0	0	1	1	010	JUMP
101	X	X	X	X	X	X	X	X	X	X	X	X	X	--	HALT

00	no branch
01	branch on AOB = 0
10	branch on AOB ⩾ 0
11	unconditional branch.

Figure 8.7. Microprogrammed multiplication.

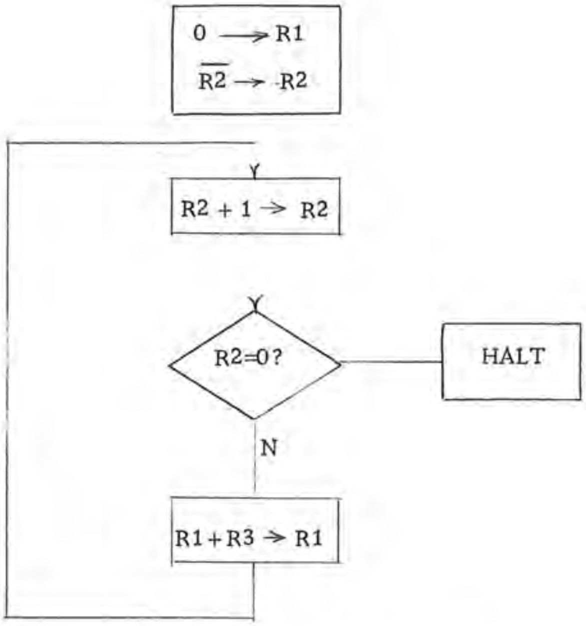


Figure 8.8. Flow chart for multiplication.

The next micro instruction is in the next memory location unless the branch condition is met. Of course, alternate schemes exist for micro instruction formats. More storage efficiency is gained through coded micro instruction formats, but the hardware complexity increases by addition of decoders. Micro insturctions may not be located in consecutive ROM locations, as in this example and other branching techniques must be used.

Any sequential network can be realized using a ROM as shown in Figure 8.9. This technique is a generalization of the one discussed above.

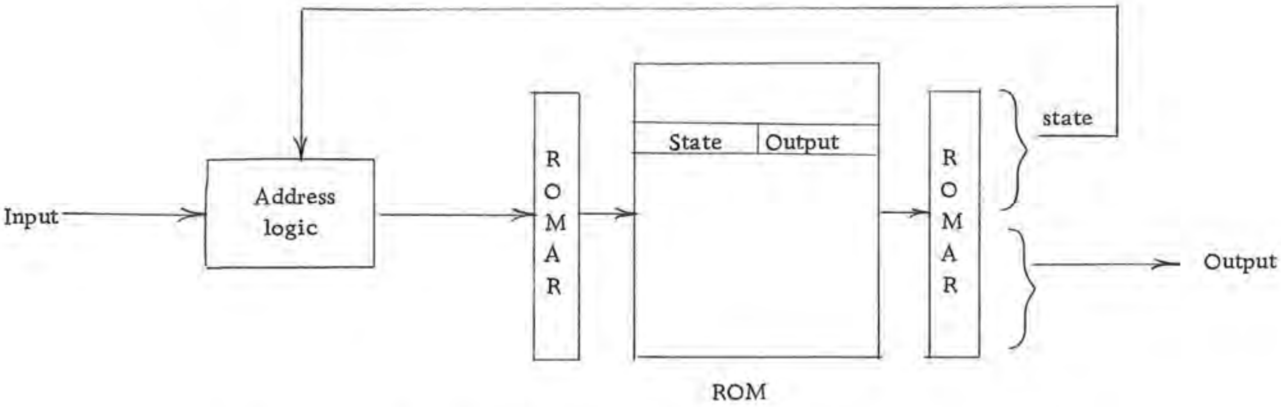


Figure 8.9. ROM realization of sequential network.

One can assume that the cost of hardware is linearly proportional to the complexity of machine cycle and the number of machines cycles required. The cost of a ROM realization has a fixed cost, the hardware associated with the control points and above that, the cost of additional memory, address registers, etc., which grows linearly with the complexity but at a slower rate. The cost versus complexity for each is roughly graphed in Figure 8.10.

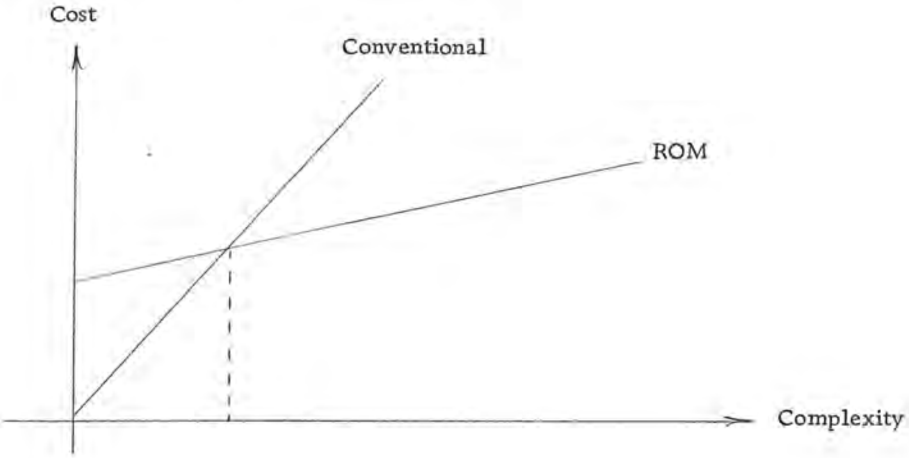


Figure 8.10. Cost considerations.

Obviously below the crossover point, conventional logic is better, above it ROM "stored logic" control is better. Some economic models [2] for microprogrammed control have shown the crossover point to be a surprisingly low level of complexity.

Some advantages of "ROM driven" control are:

1. flexibility and tailorability
2. changeability
3. ease of design
4. ease of maintainence, checking
5. uniformity
6. economy

Some of the disadvantages of microprogramming are:

1. loss of operational efficiency
2. performance loss
3. economy

8.2 Cellular Arrays

With the advent of Large Scale Integration (LSI), logic designers and theorists have been investigating new forms of logic. Most logic design techniques, prior to LSI, have tried to minimize the number of gates (AND-OR, NAND) in a network realization or the number of contacts in relay circuits. Modern technology finds cost savings manifested in uniformity and minimization of functional blocks and interconnections. With LSI capability, there is no need to restrict functional blocks to single gates or gate packages. One of the first attempts to use LSI technology to an advantage was the study of cellular cascades and cellular arrays.

A cellular cascade is a collection of logic cells with uniform connections as shown in Figure 8.11.

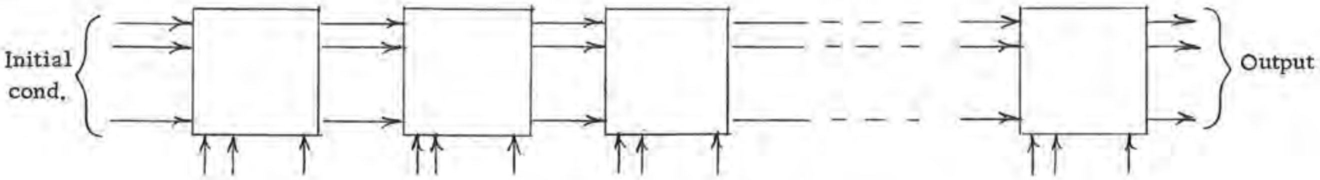


Figure 8.11. Generalized cellular cascade.

The simplest example of a cellular cascade is single rail cascade, with each cell realizing some 2 variable function $f(x_i, y_i)$.

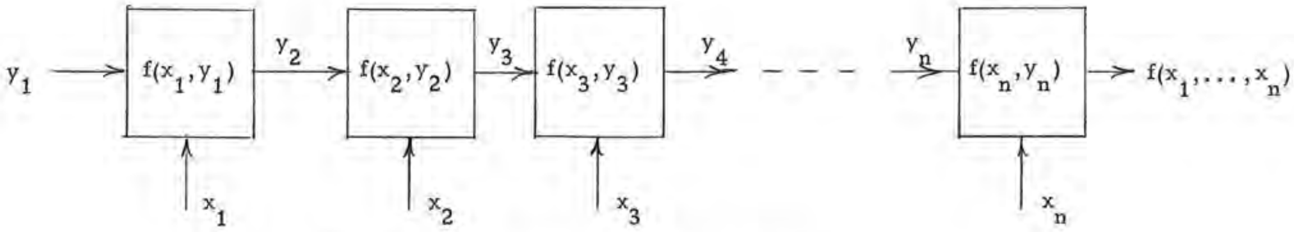


Figure 8.12. Single rail cellular cascade.

It can be easily demonstrated that a two cell single rail cellular cascade can realize all 2 variable functions, with the cells realizing only $x_i y_i$, $x_i + y_i$, y_i, $x_i' + y_i$, $x_i' y_i$, $x_i \oplus y_i$.

Some n-variable functions which can be realized with single rail cascade are: Any parity function, Any product term, Any sum term, Let us consider cascades whose inputs are not necessarily in fixed order. Any switching function can be decomposed

$$f(x_1, \ldots, x_n) = x_i g_{i1}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) +$$

$$x_i g_{i0}(x_i, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$

and the following statements can be made:

1. f is cascade realizable if $g_{i1} = 0$ or 1 and g_{i0} is cascade realizable.
2. f is cascade realizable if $g_{i0} = 0$ or 1 and g_{i1} is cascade realizable.
3. f is cascade realizable if $g_{i0} = g_{i1}$ or $g_{i0} = g_{i1}$.

Example

$$f(a, b, c, d) = a'bcd' + a'bcd + a'b'cd$$

$$= a'(bcd' + bcd + b'cd)$$

$$= a'g_{a0}\,(b, c, d)$$

hence f is realizable if g_{a0} is realizable.

$$g_{a0}\,(b, c, d) = c(bd' + bd'b'd) = c(b + d)$$

$$= cg_{c1}\,(b, d)$$

Hence g_{a0}9b, c, d) is realizable if g_{c1} is realizable, but since g_{c1} is a 2 variable function and all two variable functions are realizable then f(a, b, c, d) is realizable. The cascade of Figure 8.13 realizes f(a, b, c, d)

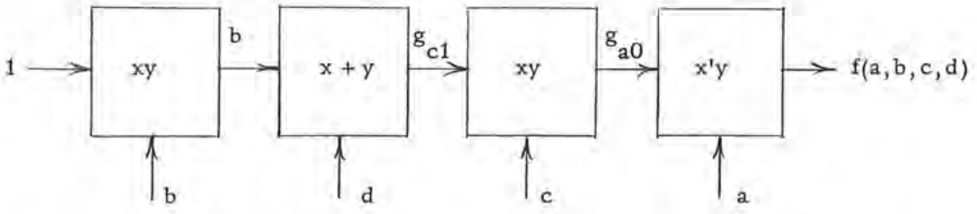


Figure 8.13. Cellular cascade realizing f(a, b, c, d) = a'bcd' + a'bcd + a'b'cd.

Note that the inputs are not in alphabetical order. Since not all functions are cascade realizable, the cellular array must be introduced.

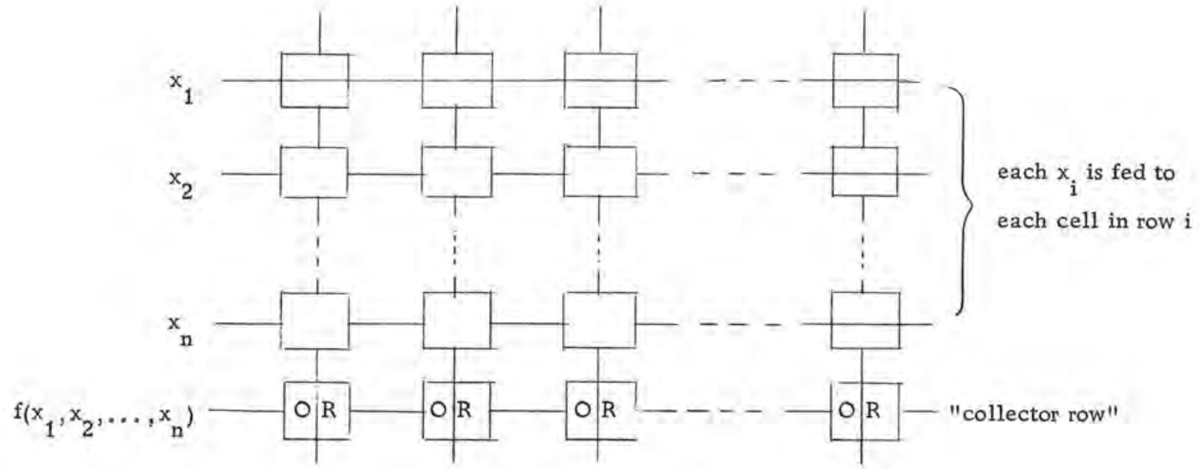


Figure 8. 14. Cellular array.

Since any product term can be realized by a cascade and the product terms "collected" by the horizontal "OR" cascade, then any function can be realized by a cellular array. The size of the array need be no bigger than $(n=1)2^n$ cells. The "collector" row of cells is set to $x_i + y_i$ or y_i depending on whether a given minterm is included. For example, a universal 3 variable array is given below:

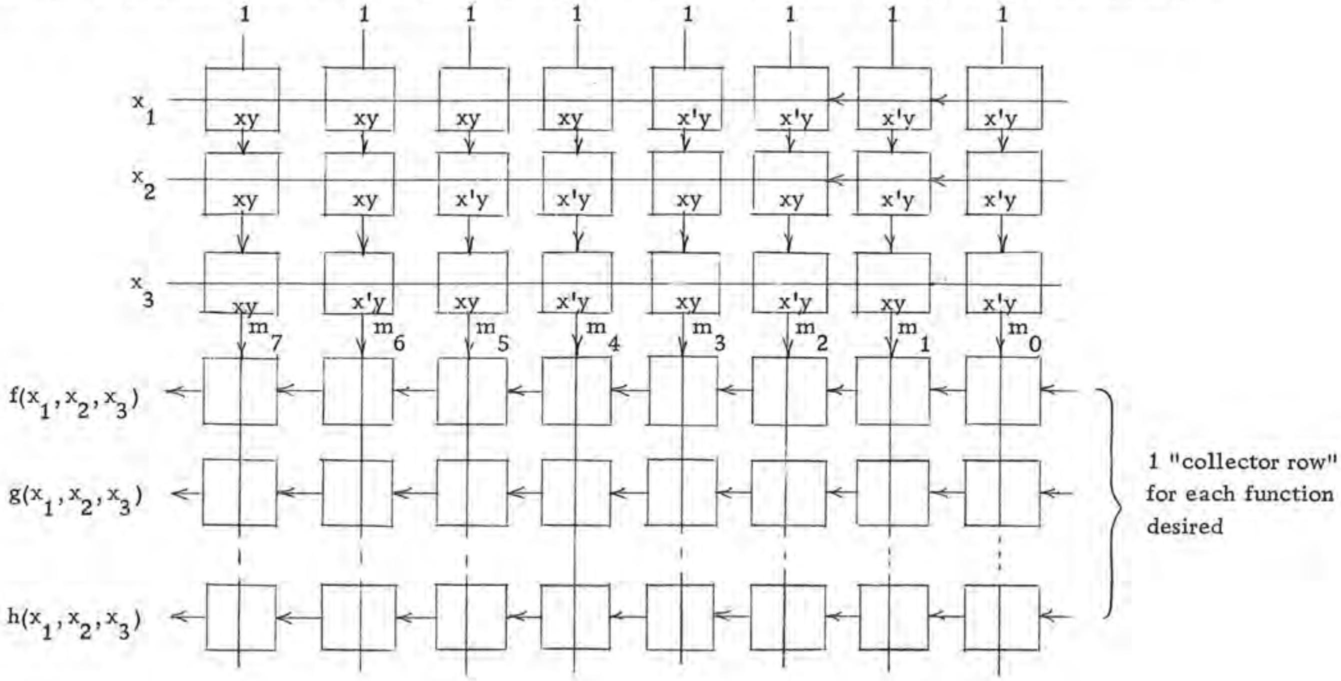


Figure 8. 15. Universal cellular array.

Of course this array is costly. The array is not restricted to having a single collector cascade and the savings accrued from realizing multiple functions using such a cascade may make its use feasible.

There is no good algorithm for finding a minimum cellular array realization for any arbitrary switching function. A heuristic ("seat of the pants") procedure makes use of the minimum sum of prime implicants representation of a function. Each column (vertical cascade) of the array realizes a prime implicant and the horizontal "collector" cascade can be used to form the sum of prime implicants. The procedure extends to multiple output functions.

For example, let

$$f_1(a, b, c, d) = a'b'c' + cd$$

$$f_2(a, b, c, d) = bd + a'bd$$

A cellular realization is shown in Figure 8.16.

Again, the advantage of cellular arrays is in the uniformity of interconnection. On an LSI chip, there is no problem laying out the circuit due to overlapping signal paths. A disadvantage is in the necessity to custom make a new chip for each logic function. This problem can be avoided by using programmable cellular arrays. One of the earliest papers (4) on cellular arrays gives a design for a general cell, whose output function can be controlled by burning interconnections within a cell using a laser. Many people have tried to determine a minimum set of control inputs which can be used to set the cells to the desired function. Most of this work is outside of the scope of these notes. The idea of programmable logic will be discussed in the section on Universal logic.

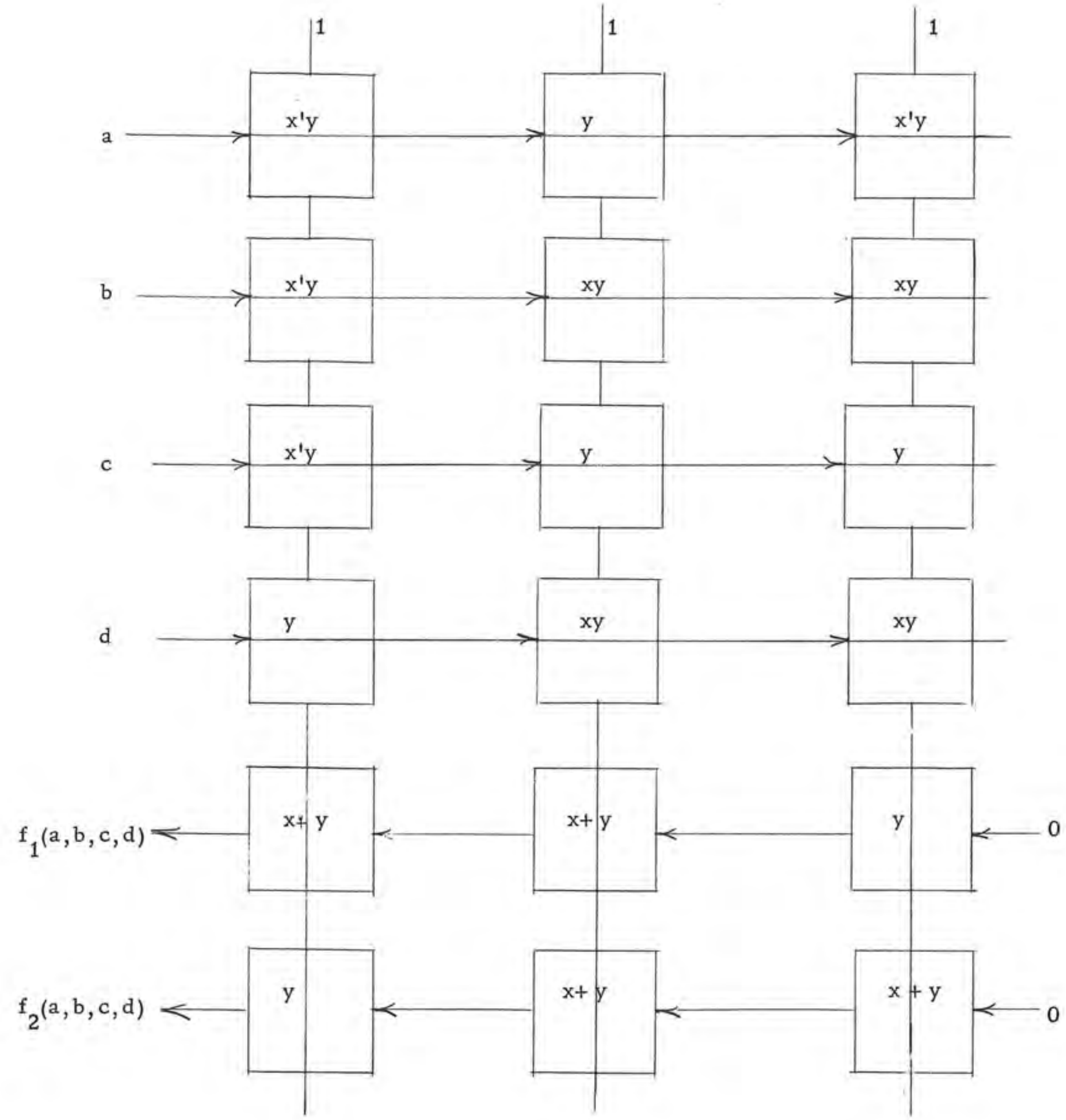


Figure 8.16. A cellular array example.

8.3 Programmable Logic Arrays

Recently National Semiconductor introduced the Programmable Logic Array (6). A logic diagram for a PLA is given in Figure 8.17.

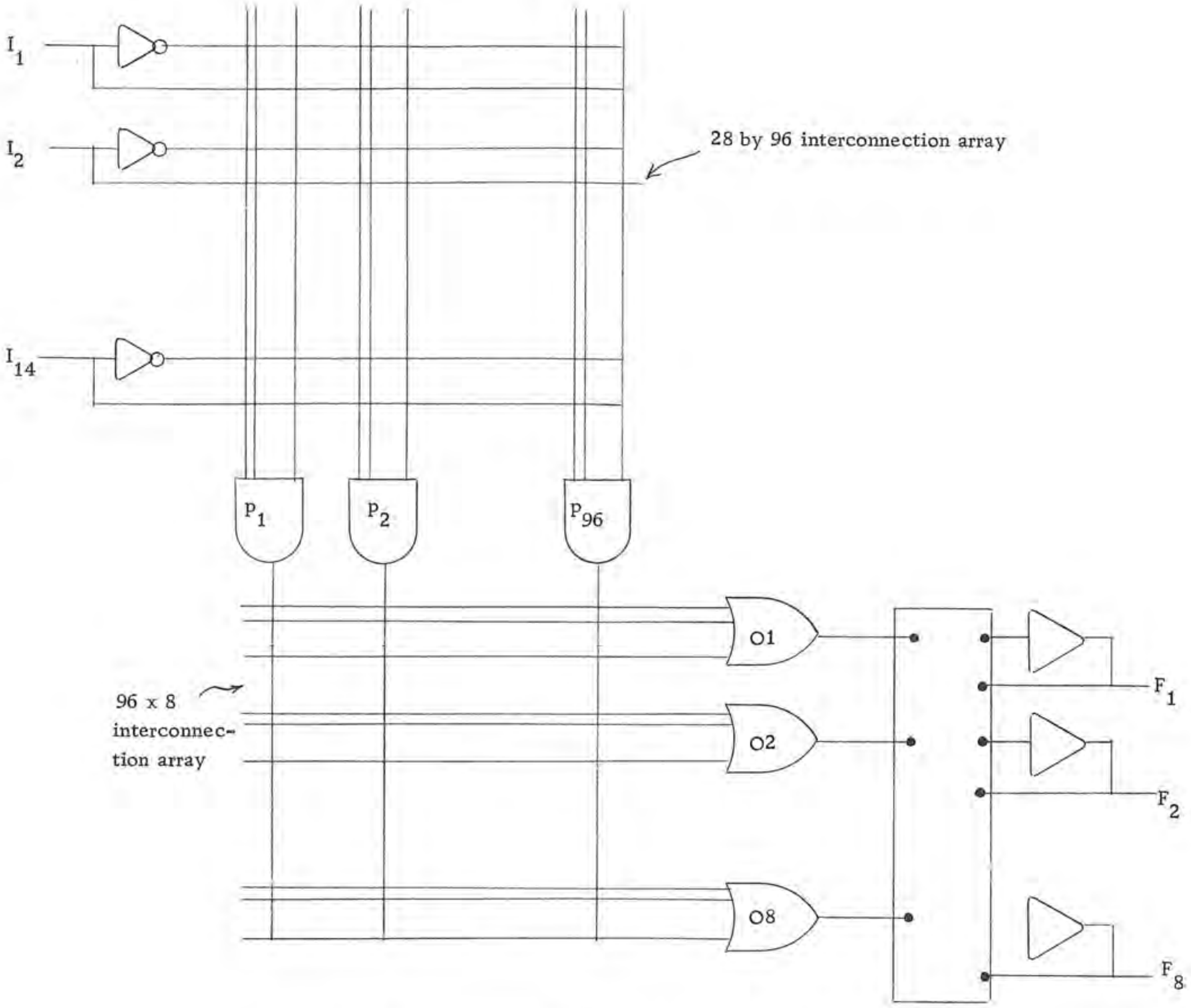


Figure 8.17. National Programmable Logic Array.

The PLA can create 96 products of up to fourteen variables or their complements and then collect eight sums of up to the 96 products. The outputs of the OR gates can be complemented. The interconnection arrays are specified by the purchaser.

PLA's have immense potential for logic design. Although not universal, they are obvioulsy powerful enough for many logic design applications. PLA's are often a better design tool than ROM's.

National supplies an "off-the-shelf" PLA programmed to convert from 12 bit Hollerith graphic code to 8 bit ASCII code. Calculations will show that the same decoder requires twelve 256, 4 bit word ROM's (a common available size) or three 1028, 8 bit word ROM's (a more expensive version). ROM propagation time is less than that for PLA, approximately 60ns versus 90ns for PLA..

PLA's can be used in the micro instruction address logic for a ROM control unit and for creating specialized arithmetic logic.

8.4 Universal Logic

As we have seen with the National PLA's alternate forms of logic (versus the traditional AND-OR, NAND networks) can use LSI technology to an advantage. We saw how a "universal" function chip could be constructed with cellular arrays in a previous section. In this section a general approach to the design of a universal logic block (ULB), with the hope of introducing techniques which one day may lead to the "computer on a chip" LSI seems to promise.

Definition. A switching function $U(z_1, z_2, \ldots, z_m)$ is said to be universal in n variables $x_1, \ldots, x_n$ for a set of available input functions S, if for any n variable switching function $f(x_1, \ldots, x_n)$ there is a way of assigning all the z_i inputs to functions in S such that $U(z_1, \ldots, z_m) = f(x_1, \ldots, x_n)$.

This definition can be clarified through use of the following example:

Let $S = (x_1, x_2, \ldots, x_n, \ 0, 1)$

the following function is universal

$$U(z_1, \ldots, z_n, z_n+1, \ldots, z_n+2^n) = \sum_{i=1}^{2^n} z_{n+i} m_{i-1}(z_1, \ldots, z_n)$$

Where $m_i(z_1, \ldots, z_n)$ is the ith n-variable minterm function.

How is this function universal? Any n variable function $f(x_1, \ldots, x_n)$ can be written as a sum of minterms (standard sum of products). By assigning $z_i = x_i$ for all $1 \leq i \leq n$ and z_j, $j > n$ such that $z_j = 1$ iff the corresponding minterm (m_{j-n-i}) is included in the canonical expansion of f, the result is $U(z_1, \ldots, z_n, z_{n+1}, \ldots, z_{n+2^n}) = f(x_1, \ldots, x_n)$. The variables $z_{n+1}, \ldots, z_{n+2^n}$ are control or selector variables selecting the appropriate minterms.

For n=2

$$U(z_1 z_2, z_3, z_4, z_5, z_6) = z_3(z_1' z_2') + z_4(z_1' z_2) + z_5 (z_1 z_2') + z_6(z_1 z_2)$$

the function $f(x_1, x_2) = x_1 + x_2'$ can be realized by setting $z_1 = x_1$, $z_2 = x_2$, $z_3 = 1$, $z_4 = 0$, $z_5 = 1$, $z_6 = 1$. Prove this for yourself by substituting in U.

The idea of building a LSI universal logic block (ULB) is a good one, provided the number of control inputs does not become too large. The minimum number (not necessarily obtainable) of control inputs can be calculated as follows:

1. There are m inputs to the ULB (assumed).
2. Suppose x_1, x_1', x_2, x_2', . . . , x_n, x_n', 0, 1 are allowed inputs to the ULB (total 2n + 2)
3. The maximum number of functions realizable by the ULB is equal to the number of ways the allowed 2n + 2 input functions can be assigned to the m input terminals = $(2n + 2)^m$.
4. This must exceed or equal the number of n variable switching functions (2^{2^n}).

Hence, $(2n + 2)^m \geq 2^{2^n}$

$$m\log_2(2n + 2) \geq 2^n$$

$$m \geq \frac{2^n}{\log_2(2n + 2)}$$

The table of Figure 8.18 compares for several values of n, the "lower bound" m, the ULB shown above which uses n + 2^n inputs, and the "best known" solution to date.

n	2^{2^n}	m	$2^n + n$	"best"
3	256	5	11	5
4	65536	6	20	7
5	$4 \cdot 10^9$	9	37	5
6	$2 \cdot 10^{19}$	17	70	24
⋮	⋮	⋮	⋮	⋮
10	-10^{314}	231	1034	228*

*used equivalence classes.

Figure 8.18. Comparison of ULB input requirements.

Through 6 variables, the ULB approach seems reasonable since a ULB for 6 variable functions can now be constructed on a 32 pin chip.

The next question to be answered deals with finding a design procedure for minimum ULB's. The following ULB has been proposed as a general model:

$$U(z_1, \ldots, z_n, z_{n+1}, \ldots, z_m) = z_{n+1}R_1 + z_{n+2}R_2 + \ldots + z_m R_{m-n}$$

Where each R_i is a disjoint ($R_i \cdot R_j = 0, i \neq j$) sum of n variable minterms. The first n z_i are assigned so that $z_i = x_i$. The allowed inputs are $x_i, x_i', x_2, x_2', \ldots, x_n, x_n'$, 0, 1. The remaining $z_i (n \leq i \leq m)$ are assigned from a subset of the allowed inputs called an I block. First, we shall give an example.

$$\text{I block} = (x_3, x_3', 0, 1)$$

$$R_1 = x_1x_2x_3 + x_1x_2x_3$$

$$R_2 = x_1x_2'x_3 + x_1x_2'x_3'$$

$$R_3 = x_1'x_2x_3 + x_1'x_2x_3'$$

$$R_4 = x_1'x_2'x_3 + x_1'x_2'x_3'$$

Note, since

$$(x_3 \cdot R_x) = (x_1x_2'x_3),\ (x_3' \cdot R_2) = (x_1x_2'x_3'),\ (0 \cdot R_2) = 0,\ (1 \cdot R_2) = (x_1x_2'x_3 + x_1x_2'x_3')$$

elements of the I block can be used to select the complete sum R_i, any sub-sum of minterms contained in R_i or 0 by ANDing R_i with the appropriate I block element. Hence,

$U(x_1, x_2, x_3, z_4, z_5, z_6, z_7) = z_4R_1 + z_5R_2 + z_6R_4$ is a ULB for 3 variable functions.

For example, $f(x_1, x_2, x_3) = x_1x_2 + x_1'x_3$

$$= x_1x_2x_3 + x_1x_2x_3' + x_1'x_2'x_3 + x_1'x_2x_3$$

If $z_4 = 1,\ z_5 = 0,\ z_6 = x_3,\ z_7 = x_3$,
then

$$U(z_1, z_2, z_3, z_4, z_5, z_6, z_7) = f(x_1, x_2, x_3).$$

This example can be extended to n-variable ULB's which require n + 2^{n-1} input lines. Prove this to yourself by finding a n-variable ULB with I block $(x_n, x_n', 0, 1)$. Note that a 6-variable ULB would require 38 inputs, greater than the known minimum. A sum of minterms R is I block selectable if for any desired sub-sum of minterms included in R that sub-sum can be obtained by ANDing R and some element of the I block.

Theorem. Let R be a s x n binary matrix describing an I block selectable sim of minterms, let the I block be complete (if a is an element of I than a' is an element of I) and let H be a n x k binary matrix.

If the rows of RH are distinct, then for all binary vectors $\underline{v}_j$ in the null space V of H, the matrices

$$R(\underline{v}_j) = R + \begin{bmatrix} \underline{v}_j \\ \underline{v}_j \\ \vdots \\ v_j \end{bmatrix}$$

describe disjoint sums of minterms which are selectable by the I block described by R.

Example: I block = $(x_3, x_3', 0, 1)$

$$R = \begin{matrix} 000 \\ 001 \end{matrix}$$

$$H = \begin{matrix} 0 \\ 0 \\ 1 \end{matrix}$$

$$RH = \begin{matrix} 000 \\ 001 \end{matrix} \cdot \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} = \begin{matrix} 0 \\ 1 \end{matrix} \text{ the rows are distinct}$$

$$v = (\,(000),\ (010),\ (100),\ (110)\,)$$

$$\text{since } (000) \cdot \begin{matrix} 0 \\ 0 \\ 1 \end{matrix} = (0);\ (000) \text{ is in the null space of H}$$

$$R(000) = \begin{matrix} 000 \\ 001 \end{matrix} = x_1'x_2'x_3' + x_1'x_2'x_3$$

$$R(010) = \begin{matrix} 000 \\ 001 \end{matrix} + \begin{matrix} 010 \\ 010 \end{matrix} = \begin{matrix} 010 \\ 011 \end{matrix} = x_1'x_2x_3' + x_1'x_2x_3$$

$$R(100) = \begin{matrix} 100 \\ 101 \end{matrix} = x_1x_2'x_3 + x_1x_2'x_3$$

$$R(110) = \begin{matrix} 110 \\ 111 \end{matrix} = x_1x_2x_3' + x_1x_2x_3$$

What the theorem above allows us to do is to find, given an I-block, a set of disjoing sums of minterms which are selectable using the elements of the I block.

For up to 7 variables, the most general design procedure uses an I block of the form $(x_n, x_n', 0, 1)$, partitioning the minterms into 2^{n-1} disjoint I block selectable sums.

The following guide lines will lead to another general design for an ULB. Suppose the I block is $(x_1, x_1', x_2, x_2', \ldots, x_n, x_n', 0, 1)$. In other words, all available inputs are used as control variables. Recall that R, the matrix describing the I block selectable sum of minterms has dimension s x n. If s is a power of two (for some r, $s = 2^r$) construct R as follows:

1. Every s bit binary vector or its complement is a column of R (except 000....0 and 111...1).
2. The first r columns of R form a submatrix whose rows are all r bit binary vectors. Note that the number of columns of R is identical to the number of variables for which the ULB is universal and since there are $2^s - 2$ s bit binary vectors other than 000...0 and 111...1 and each vector or its complement must be present as a column, then there are at least $2^{s-1}-1$ columns in R. The dimension s is the number of minterms in block.

Example $x = 4$ $n = 8$

$$R = \left(\begin{array}{c:c} 00 & 000010 \\ 01 & 001100 \\ 10 & 010100 \\ 11 & 100000 \end{array}\right) \qquad H = \begin{pmatrix} 10 \\ 01 \\ 00 \\ 00 \\ 00 \\ 00 \\ 00 \end{pmatrix}$$

For this example there are $2^6 = 64$ vectors in the null space of H, hence 64 blocks of 4 minterms. The block described above is

$$R = x_1'x_2'x_3'x_4'x_5'x_6'x_7'x_8' + x_1'x_2x_3'x_4'x_5x_6x_7'x_8' + x_1x_2'x_3'x_4x_5'x_6x_7'x_8'$$

$$R + x_1x_2x_3x_4'x_5'x_6'x_7'x_8'$$

Note the first term can be selected by $x_7 \cdot R$, the second by $x_5 \cdot R$ and so forth.

Universal Logic Blocks have been found which use fewer inputs than the ones which can be designed by these general procedures. For instance,

$$U(z_1, \ldots, z_5) = z_5 + z_1 z_2 z_4' + z_1' z_3' z_4' + z_1 z_2 z_3' z_4 + z_1' z_2' z_3 z_4$$

is universal in 3 variables where all z_i are assigned from the set $(x_1, x_1', x_2, x_3, x_3', 0, 1)$. This requires a table look up for the assignment.

8.5 Notes - Problems - References

An interesting approach to ROM design is given by Clare (1) and the best text on microprogramming is by Husson (2).

Cellular Array research is summarized in a paper by Minnick (4) and programmable arrays are discussed by Kautz (3).

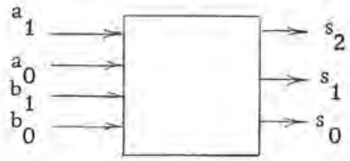
A brief introduction to Universal logic and a good list of references can be found in Preparata (5). A note on design using PLA appeared recently in (1).

REFERENCES

1. Clare, C. R., Designing Logic Systems Using State Machines, McGraw-Hill, New York, 1973.
2. Husson, S., Microprogramming: Principles and Practices, Prentice-Hall, Englewood Cliffs, N. J., 1970.
3. Kautz, W., "Programmable Cellular Logic", in Recent Developments in Swtiching Theory, A. Mukhopadhyay, Ed., Academic Press, New York, 1971.
4. Minnick, R. C., "A Survey of Microcellular Research", JACM, Vol. 14, pp. 203-241, April 1967.
5. Preparata, F., "Universal Logic", Proc. of 1st Texas Symposium on Computer Systems, August 1972.
6. Priel, V. and P. Holland, "Application of a High Speed Programmable Logic Array," Computer Design, Vol. 12, No. 12, December 1973.

PROBLEMS

1.



The network above is a two bit full adder with inputs $\underline{a} = (a_1 a_0)$ and $\underline{b} = (b_1 b_0)$, the output is $\underline{s} = \underline{a} + \underline{b} = (s_2 s_1 s_0)$. Example: $\underline{a} = (10)$, $\underline{b} = (11)$, $\underline{s} = (101)$.

a. Design the adder using a 12 input Universal Logic Block with allowable control inputs $(a_0, a_0', 0, 1)$. (Define the ULB, but do not design the network).

b. Design the adder using a Cellular Array. (Solution requires a 11 columns and 7 rows).

c. Design the adder using Read Only Memory with 16 memory locations, each containing a 3 bit word.

2. Using the Arithmetic Unit of Figure 8.3 and assuming that the OR gates perform a bit wise OR function on the parallel data transfer, find micro instructions for

a. $R_1 + R_2 \longrightarrow R_1$

b. $R_1 \cdot R_2 \longrightarrow R_1$ (3 instructions)

c. Can the exclusive-OR function be realized?

3. Write a flow chart and a microroutine for a division algorithm (by successive subtraction). Assume the divisor is in R2, the dividend in R3, leave the quotient in R1 and the remainder in R3. Another control point m for a data transfer AOB ⟶ R3 is needed.

4. Find a two variable universal logic Block (ULB) that has 4 inputs.

5. Prove that a logic block realizing $U(z_1, z_2, z_3) = z_1 z_2 \oplus z_3$ is universal in 2 variables x, y on the set $(x, x', y, y', 0, 1)$. Realize all 16 functions of x, y as a proof.

Example:

$$f(x, y) = x + y$$

$$z_1 = x', z_2 = y, z_3 = x$$

$$\begin{aligned} U(z_1, z_2, z_3) &= x'y \oplus x = x'yx' + (x'y)'x \\ &= x'y + (x + Y')x \\ &= x'y + x + xy = x + y \end{aligned}$$