DIGITAL COMPUTER LABORATORY
UNIVERSITY OF ILLINOIS
URBANA, ILLINOIS

INTRODUCTION TO THE THEORY OF DIGITAL MACHINES

Math., E.E. 294 Lecture Notes

by

W. J. Poppelbaum

CHAPTER I

DIGITAL COMPUTERS AND NUMBER SYSTEMS

1.1 Analog Computers and Digital Computers

A computer is a calculating machine capable of accepting numerical data and performing upon them mathematical operations such as addition, taking the square root, etc. The computer can also accept non-numerical data by establishing, via a code, a correspondence between the information at its input and the numbers used inside. The mechanism involved in computation can use any one of the common physical agents (mechanics, electricity, etc.).

The data inside the machine can be in the form of continuously variable measurements, such as voltages in a given range, angles; we then talk of an analog computer (example: slide-rule). If the data are in the form of discrete numbers (assembly of a finite number of multi-valued digits), we speak of a digital computer (example: desk calculator). With such a computer nearly unlimited precision can be obtained even when standard hardware is used, while the results of pure analog computation are usually only known within a fraction of a per cent. It should be remarked that combinations of the two principles are possible and used in some installations.

General Organization of a Digital Machine

A digital computer can take the simple form of a desk calculator using toothed wheels. In the decimal system these wheels would have ten discrete positions, 0 ... 9. Individual operations are then controlled by a human operator, customarily using a writing pad which contains the list of instructions to be performed, the numbers to be operated on and the intermediate results. The time for multiplying two numbers of 10 decimal digits each is of the order of 10 seconds; the time necessary to write down the result and for pushing the keys can be almost neglected.

In an electronic computer the digits are represented by the electrical states of electronic circuits, i.e., circuits using transistors. Usually these circuits (called flipflops and assembled in registers) have two states (e.g., a high voltage output or a low voltage output), which means that only two discrete

values, 0 and 1, are available per digit. We must then use the binary system in which the numbers 1, 2, 3, 4, 5 etc. are represented by 1, 10, 11, 100, 101 etc.

The time for multiplying two numbers of 30 binary digits ($\longrightarrow$in precision to 10 decimal digits) is of the order of 10 - 100 microseconds; manual control and the use of a pad to jot down intermediate results would be very inefficient. The writing pad is replaced by a memory (in principle a great number of flipflop registers) which stores from the outset the list of instructions and which, by way of a control-unit, established the electrical connections necessary to perform the operations. The memory is also used to store back the intermediate results. An electronic machine will be automatic at the same time, in the sense that it proceeds all on its own through the problem due to the stored program. The part of the machine corresponding to the desk calculator is called the arithmetic unit. The latter is usually connected to the input-output equipment (tapes with holes or magnetic coating with reading and writing devices). As the name implies, this input-output equipment allows the machine to communicate with the outside world, e.g., store numbers in the memory after having read holes punched in tapes (or cards), or punch holes corresponding to the memory contents at the end of a problem. This general layout of the computer is the same for installations as widely different as the "Illiac" and the IBM 650.

Figure 1-1 below summarizes the general organization.

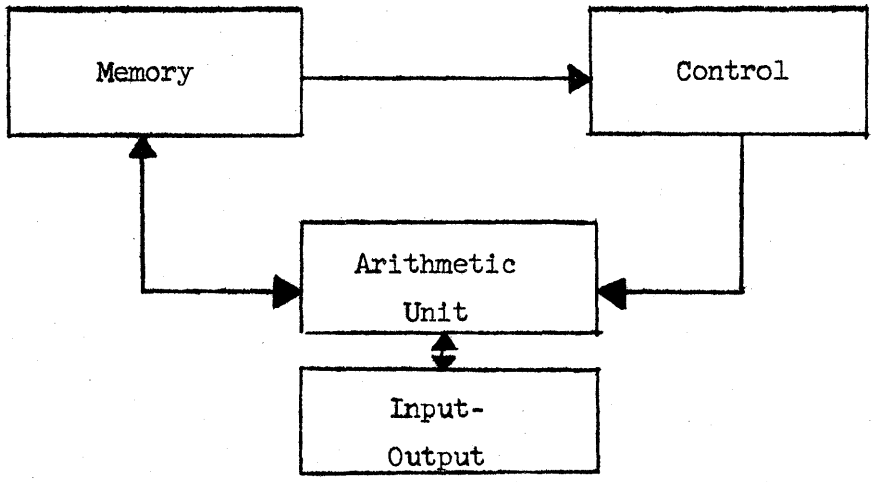


Figure 1-1
General Organization of a Digital Computer

The "Thinking" Ability of Computers

The astonishing usefulness of a modern computer is due to the possibility of having it make simple decisions. These are usually of the following form:

1. If number in register A $\geq$ number in register B, follow instructions stored in list 1 in memory (e.g., locations 56, 57, 58 ...);
2. If number in register A < number in register B, follow instructions stored in list 2 in memory (e.g., locations 82, 83, 84 ...).

The transfer from one list to another depending on the contents of registers is called a conditional transfer, jump or branch.

Example 1

Take the numerical calculation of the value of a function expressed as a series. The number of terms we have to take in order to obtain a fixed precision varies with the value of the argument. We can decide that we are going to calculate up to the n^{th} term where this term is smaller than a given quantity δ. After being through with the calculations of each term, we shall test and see if it is bigger than δ. If so, we shall go on to the next term; if not, we shall form the sum of all the terms calculated up to this stage, and then proceed with the rest of the problem.

Example 2

Suppose that 4 numbers e.g. 19, 7, 12 and 17 are stored in memory locations 1, 2, 3 and 4 respectively and that we want to put them in increasing order. This is done by a process known as "merging". First 19 and 7 are ordered by a command of the type "subtract memory location 2 from memory location 1 if the result is positive interchange their contents, otherwise leave the order". Similarly memory locations 3 and 4 are ordered. Now we have two ordered lists

7	12
19	17

We compare the top members: The smaller one is 7. This we put on top of the "merged list" and strike out the corresponding number in the array, giving

12

19 17

We know that 17 follows 12 (this list being partially ordered) so the only question is: does 19 follow 12 or 17? This can be decided by two more decisions of the type used before and the problem is solved in a language suitable for a computer.

The ability to exercise "judgment" and to choose between two alternatives saves a great amount of time, but, of course, the programmer must write down the details of what to do in each case before the computation starts. It is possible to extend this ability to judge in such a way that the computer virtually assembles its own program (list of instructions) once the list of subroutines is given and the general method of calculation is prescribed in symbolic form. This is called automatic programming.

1.2 Fundamental Computer Vocabulary

Serial and Parallel Machines

As mentioned, the numbers are stored in register, i.e., sets of flip-flops. In order to calculate, the digital computer shifts the digits from one register to another, adds, subtracts, multiplies or divides the contents of two registers and transfers the result to a third. We can see that all arithmetic operations can be performed if we provide an adder, equipment which takes the negative of a number held in a register (subtracting the digits means adding the negative) and shift facilities which transfer into another register and simultaneously give a displacement of digits by one digit position to the right or left. Since multiplication is a series of additions and shifts, and division a series of subtractions and shifts, such an arithmetic unit would be capable of performing the four operations of arithmetic.

There are two fundamentally different methods for transmitting the digits from one register to the other (or through the adder). If a separate wire is used for each digit and all digits are transmitted simultaneously, we speak of parallel operation. If the digits are "sensed" one after the other and transmitted through a single wire, we speak of serial operation. To illus-

trate the latter case, we can think of a selector mechanism which always connects the flipflops having the same position in the order 1-1, 2-2, 3-3 ... n-n. Telephone systems use serial transfer of information.

It turns out that parallel operation gives higher computation speeds, while serial operation cuts down the amount of equipment used. It is difficult to ascertain the proportion in which we gain speed or reduce equipment in going from one system to the other. For n digital positions the gain is certainly less than a factor n.

Synchronous and Asynchronous Operation

In a synchronous machine there exists a central unit called a clock which determines by its signals the moment at which the steps necessary to perform an operation (such as addition, shifts, etc.) are initiated and terminated. For each type of operation we need a fixed number of cycles of the clock whether, in practice, the intermediate steps were long or short (the length usually depends on the numbers involved).

In an asynchronous machine there is no clock to sequence the steps. This can be attained by having each step send out an "end signal" which initiates the next step (kinesthetic machine). There are systems of various degrees of asynchronism, ranging from those in which the times of action of a set of circuits are simulated in a delay device (i.e., in which the end signal or reply-back signal is simply the previous end signal delayed by a sufficient amount of time to allow the set of circuits to operate properly; this amounts to a local clock), to systems in which the operation of each set of circuits is examined by a checking circuit which gives end signals if, and only if, the operation has really been performed. A special type of asynchronous machine is the "speed-independent" machine in which an element may react as slowly as it likes without upsetting the end result. One way to obtain speed independence is to build a "totally sequential" machine in which only one element acts at a time; this would have to be a serial machine.

It should be mentioned that often only a part of the computer is asynchronous. In "Illiac", for example, the arithmetic unit is asynchronous while the (electrostatic) memory is synchronous. In the IBM 650, both the arithmetic unit and the (drum) memory are synchronous.

Two-Level DC and Pulse Representation

Information, i.e., digit values, can be represented inside the machine by two different methods. Suppose that we have agreed upon a binary machine using only the values "0" and "1" for each digit. We can then decide to represent these values by sending pulses (of approximately rectangular shape and a duration of the order of 0.1 - 10 μs) from one register to the other. In such a pulse machine the presence of a pulse would mean "1", the absence, "0", (the inverse convention could be made too). Usually these pulses are sent (or not sent) at fixed intervals, i.e., a pulse machine is, in most cases, a synchronous machine (example: IBM 650).

In a direct-coupled machine we would represent the values of a digit by a given dc level. For instance, "1" would mean -20v and "0" would mean 0v (Illiac system). Any other correspondence would, of course, be just as good. The name "direct-coupled" stems from the fact that, contrary to pulse machines, no coupling capacitors may be used in the circuits for these cannot transmit dc levels. Note that current levels can be substituted for voltage levels in a dc representation.

Which design philosophy is chosen in a given machine depends on whether we would like to have simple circuits which are harder to service (pulse machines) or more elaborate circuits which are very convenient when it somes to checking their operation (dc-coupled machines). In a pulse machine we must inject pulses and observe their combinations and modifications as they go through the circuits. In a dc-coupled machine we only have to check for the proper behavior of each element using a voltmeter.

It is sometimes alleged that the two level dc representation allows faster operation since the signal only has to change once in order to transmit one bit (= binary digit) of information, while in a pulse the signal has to go up and down. This view is erroneous because the duty cycle of the active elements (transistors, tubes) is as much as "1" in a dc system (i.e., these elements can be on all the time) and less than 0.5 in a pulse system (rise time $\sim$ fall time, no tops and valleys in a fast system!). At equal average power dissipation, the speeds of the two systems are comparable.

1.3 Memory Systems. Single and Multiple Address Machines

At a first glance it may seem to be useful to have separate memories for numbers and orders (instructions). But if we take account of the fact that the memory stores also intermediate results and that conditional transfers of control often make the sequential read-out of orders impossible anyway, it seems preferable to use the same memory for both orders or numbers (common name "words"). Each order then has to specify the locations of the numbers it has to operate upon; the numerical specification of a memory location is called an address. The storage of orders and numbers in the same memory also makes possible modifications of orders during the calculation.

These memories or stores as they are also called, are divided into two kinds: so-called "random access" memories in which any word can be directly attained and the "back-up" memories in which a given word is contained in a long list which must be scanned. Typically the random access memory consists of magnetized cores (number of bits per word x number of word cores!) the state of magnetization of which represents 0 or 1. Reading out such a memory consists in setting the cores to a standard state and observing the change of magnetization by induced voltages. Another way of storing information in a random access memory is to transform each word into a sequence of dim or bright spots on a TV tube: these cathode-ray-tube memories (also called Williams-tube memories) must be regenerated periodically because they are volatile.

Back-up memories consist almost invariably of magnetic drums or magnetic tapes. In both cases each word is transformed into a sequence of magnetized or unmagnetized spots on a magnetic coating i.e. we have really a glorified tape-recorder. It is evident that both these systems are sequential in nature because we must wait for the drum (tape) to be in the correct position in order to start reading by means of a series of fixed reading heads.

Many modern computers contain a buffer memory between the arithmetic unit and the random access memory in which a certain amount of advanced processing can be done. These "memory plus simplified arithmetic unit" systems are called "look aheads" or "advanced control". They use as their storage medium simplified flipflops ("flow-gating" in Illiac II) or specially fast core memories.

Since all arithmetic operations involve two numbers, a and b, and give a result, c, ($c = a + b$, $a - b$, ab, a/b), we would need in the general case five pieces of information for each order:

1) the address of a;
2) the address of b;
3) the kind of operation to be performed;
4) the address to which c shall be sent;
5) the address of the next order.

For obvious reasons the above system is called a "4-address system". One can simplify the procedure enormously by introducing certain conventions:

1) the address of a is a fixed register in the arithmetic unit (which one may depend on the type of order);
2) the address of b is to be given as above;
3) the kind of operation is specified as before;
4) c is left in a fixed register unless the order specifies that it is to be sent to the memory, in which case a is taken to be in a fixed register;
5) the address of the next order is the number immediately following, unless a specific order to "transfer control" is given. The only address specified is then that of the next order; a, b and c are not involved.

A system which uses the above conventions is called a single-address system. It is easy to see that making only part of these conventions, one can obtain two-address and three-address systems.

1.4 Past and Present Digital Computers

Calculators of the mechanical type date back to Pascal, who, in 1642 invented an adding machine using toothed wheels to represent numbers. Leibnitz, in 1671, extended the principles used to obtain multiplication. The first time desk calculator was produced by Thomas de Colmar in 1820.

At this time Charles Babbage in England conceived the idea of using punched cards to direct a giant desk calculator in its efforts. The idea of storing programs for looms on cards had been introduced by J. M. Jaccard in 1804: patterns were produced by operating the weft selectors according to rows of punched holes in an endless belt. This machine had such advanced features as

transfers of control. On demand the machine would ring a bell an attendant would present to it tables of logarithms, sines etc., again in the form of punched cards. Unhappily the project was abandoned after having spent about $200,000 on it.

The first working model of a stored program computer was built by Howard Aiken at Harvard: The Harvard Automatic Sequence Control Calculator Mark I. It was used during World War II. It contained a 60' shaft to drive the diverse mechanical units. Bell Laboratories then produced several computers using relays rather than toothed wheels. All these were superseded by ENIAC, built by the Moore School of Electronics at the University of Pennsylvania using tubes exclusively (1946). Remington Rand soon came out with a commercial machine, Univac I and IBM, with some some delay, with its model 650 which is still widely used. Meanwhile John von Neumann, Burks and Goldstine made plans for a very comprehensive machine for the IAS in Princeton: Illiac I is a copy of this machine.

Recently three still more ambitious projects have been completed. IBM has designed its STRETCH computer (150,000 transistors), Remington Rand the LARC (60,000 transistors) and the University of Illinois Illiac II or NIC (30,000 transistors). All these machines have gone to the extreme limit of speed where their dimensions (via the propagation time of electrical signals of 1 mμs/foot) set a bound to their times: All three machines can multiply in less then 10 μs.

Table 1-1 gives some characteristics of well-known machines.

1.5 Positional Notation

Integer Bases

Let $b \neq 0, \pm 1$ be the base or <u>radix</u> of the system. This means that each digit can have n values α ranging from 0 to n-1 where $n = |b|$. Denoting by α the value of α in the k^{th} position and by K the upper limit of k, we can then represent

$$x = \sum_{k=-\infty}^{k=K} \alpha_k \, b^k \qquad (1\text{-}1)$$

Table 1-1

Characteristics of Some Well-known Computers

Name	Country	Manufacturer	Timing	Multiplication Time	Memory	Address	A.U.
LGP 30	U.S.A.	Librascope	synchr.	24000 µs	drum	1	serial
IBM 650	U.S.A.	IBM	synchr.	19000 µs	drum	2	serial
IBM 704	U.S.A.	IBM	synchr.	228 µs	cores	1	parallel
Illiac	U.S.A.	Univ. of Illinois	asynchr.	700 µs	el. st.	1	parallel
Univac 1103	U.S.A.	Remington-Rand	synchr.	290 µs	cores	2	parallel
Edsac II	Gr. Br.	Univ. of Cambridge	synchr.	300 µs	cores	1	parallel
Besm	U.S.S.R.	Inst. Prec. Mech.	synchr.	270 µs	el. st.	3	parallel
Ermeth	Switzerland	Polytechn. Zurich	synchr.	16000 µs	drum	1	parallel

by

$$\alpha_K \alpha_{(K-1)} \cdots \alpha_0 \cdot \alpha_{(-1)} \cdots \qquad (1\text{-}2)$$

The "radix" point being immediately to the right of the $b^0 = 1$ position.

Example

3.14 in base 10 is $3 \times 10 + 1 \times 10^{-1} + 4 \times 10^{-2}$. In order to distinguish it from 3.14 in base 7 (i.e. $3 \times 7 + 1 \times T' + 4 \times 10^{-2}$) we can write 3.14_{10} and 3.14_7 respectively.

The question comes up if any positive number x can be represented by an expansion of the form (1-1) for any value of b (positive or negative) different from 0 and 1. The answer to this problem is given by

Theorem 1: If b is integral ($\gtrless 0$) and $|b| \neq 1,0$, any positive number x has an expansion of the form (1-1).

Proof: If expansions exist for x' and x", there exists an expansion for the sum x' + x" which is obtained by the well-known process of "adding each column and taking account of the carries". This latter point is obvious for $b > 0$. If b is < 0, we can observe that the signs of the terms in (1-1) alternate. Let us take three terms

$$-\alpha'_{(2n+1)} |b|^{2n+1} + \alpha'_{(2n)} |b|^{2n} - \alpha'_{(2n-1)} |b|^{2n-1}$$

in the expansion of x' and

$$-\alpha''_{(2n+1)} |b|^{2n+1} + \alpha''_{(2n)} |b|^{2n} - \alpha''_{(2n-1)} |b|^{2n-1}$$

in that of x" and suppose that to the right of these terms no carries were necessary, i.e. let $|b|^{2n-1}$ be the term in which for the first time $\alpha'_{(2n-1)} + \alpha''_{(2n-1)}$ exceeds $|b|$. In order to carry we have to form $|b| \times (-|b|^{2n-1}) = -|b|^{2n}$ out of terms to the left. This can be done by observing that $-|b|^{2n} = -|b|^{2n+1} + (|b| - 1) |b|^{2n}$. Therefore the carry only influences the two terms to the left. This still holds if the three terms chosen have the sequence of signs -, +, -. A step by step process allows us

therefore to absorb all carries when we form x' + x", i.e., we can write down explicitly the expansion of the sum. Now we only have to prove that there is always an $x^* > 0$ as small as we like in the set of all expansions of the form (1-1): this is quite obvious. By summing a sufficient number of these "small x^* - expansions" we can then come as close as we like to a given x.

Positive Fractional Bases. The Most Economical Base

It is not hard to prove that we can extend the above arguments to positive bases of any kind (rational or irrational) if we take

1) $b \geq 0.5$ (still excluding 1)
2) $n = 2$ minimum and generally $n = 2 + [b] - [2 + [b] - b]$, where [b] is the greatest integer contained in b. (The above function gives the next highest integer!)

We can then supplement Theorem 1 by

Theorem 2: If b is any non-integral positive number, any arbitrary number x has an expansion of the form (1-1).

Proof: We can always scale down x by division by b^m (m = integer) in such a way that $x < 1$. Furthermore by the transformation $B = 1/b$ we can reduce the case $b < 1$ to the case $b > 1$. Then the expansion will only start to the right of the point and we can find the α's by multiplying both sides by b and comparing integral parts.

Example

Express 2_{10} in base $\frac{2}{3}$. We start by finding the expression in base $\frac{3}{2}$, giving us n = 2 i.e. the possible values of α_k are 0 and 1. Let us first scale 2 by division by $(\frac{3}{2})^m$ to obtain a quantity less than one: visibly m = 2 is sufficient. Our problem now looks as follows

$$\frac{2}{(\frac{3}{2})^2} = \alpha_{-1} (\tfrac{3}{2})^{-1} + \alpha_{-2} (\tfrac{3}{2})^{-2} + \alpha_{-3} (\tfrac{3}{2})^{-3} + \ldots.$$

By successive multiplication by $(\frac{3}{2})$ and comparing integer parts we find

$$\alpha_{-1} = 1,\ \alpha_{-2} = 0,\ \alpha_{-3} = 0,\ \alpha_{-4} = 1 \text{ etc. i.e.}$$

$$\frac{2}{(\frac{3}{2})^2} = 1\,(\tfrac{3}{2})^{-1} + 0\,(\tfrac{3}{2})^{-2} + 0\,(\tfrac{3}{2})^{-3} + 1\,(\tfrac{3}{2})^{-4} + \ldots.$$

or

$$2 = 1\,(\tfrac{2}{3})^{-1} + 0\,(\tfrac{2}{3})^{0} + 0\,(\tfrac{2}{3})^{1} + 1\,(\tfrac{2}{3})^{2} + \ldots.$$

which means that

$$2_{10} = \ldots. 100 \cdot 1_{(\frac{2}{3})}$$

Note that for a base < 1 the smaller terms lie to the left of the radix point.

An important practical question is: which base b is such that the minimum amount of equipment is necessary to express a given number of numbers M. Let the number of digits be m, then $M = b^m$ (actually $M = n^m$, but we can take b^m as an approximation). Also bm (actually nm) is an estimate of the amount of equipment necessary. The problem is thus: find b such that bm is minimum subject to the condition $b^m = M$. Setting bm = u we have

$$u = \frac{b}{\ln b} \cdot \ln M$$

For the most economical b we have $\frac{du}{db} = 0$, i.e.

$$\frac{(b \cdot \frac{1}{b} - \ln b)\,\ln M}{(\ln b)^2} = 0$$

That is: $\ln b = 1$, giving $b = e = 2.71828\ldots$.

It is interesting to fix $M = 10^6$ and to calculate bm for b = 2, 3, 4, ... The results are given in

Table 1-2

b	bm
2	39.20
3	38.24
4	39.20
10	60.00

We see therefore that base 2 is a good choice: for once the system dictated by the electronic nature of the number representation is also nearly the most efficient.

Arithmetic in Other Bases

One can show quite easily that all arithmetic operations can be performed in other bases (see F. E. Hohn, "Applied Boolean Algebra") as long as we take account of the modification of the addition and multiplication table.

Example

In base 5 these two tables look as follows:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	10
2	2	3	4	10	11
3	3	4	10	11	12
4	4	10	11	12	13

x	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	11	13
3	0	3	11	14	22
4	0	4	13	22	31

The multiplication of 143202_5 by 2431_5 can be done by multiplying 143202 by 2 (giving 341404 taking account of the fact that whenever the sum is more than 5, carries are generated), then adding to it - shifted by one digit position - the product of 143202 by 4 etc.

Conversion of Positive Integers from One Integer Base to Another

It is possible to convert from a base b to a base d by successive divisions by d: the remainders are retained, the first remainder being the least significant digit.

To see this we consider the two equivalent representations of the chosen integer:

$$\sum_{k=0}^{k=K} \alpha_k \, b^k = \sum_{l=0}^{l=L} \beta_l \, d^l$$

Suppose that the α_k are known and that we want to calculate the β_1. Division by d yields

$$\frac{1}{d} \sum_{k=0}^{k=K} \alpha_k b^k = \beta_L d^{L-1} + ... + \beta_2 d + \beta_1 + \frac{\beta_0}{d},$$

showing that β_0 is the remainder after the first division. The same reasoning applies to further divisions. After L + 1 divisions we have then found $\beta_L \ldots \beta_1 \beta_0$. Note that all operations are performed in the base b.

There is a special case if $d = b^m$ (m = integer), e.g. if we convert from binary to octal (2^3) or sexadecimal (2^4) bases. The digits can then be arranged in groups of m and each group converted separately:

$$\ldots + \alpha_{(2m-1)} b^{2m-1} + \ldots + \alpha_m b^m + \alpha_{(m-1)} b^{m-1} + \ldots + \alpha_1 b + \alpha_0$$

$$= \ldots + [\alpha_{(2m-1)} b^{m-1} + \ldots + \alpha_m] b^m + [\alpha_{(m-1)} b^{m-1} + \ldots + \alpha_0] b^0$$

Conversion of Positive Fractions from One Integer Base to Another

The method for converting fractions is quite similar to that for integers, except that successive multiplications by d are performed. To see this we consider the two equivalent representations of the chosen fraction:

$$\sum_{k=1}^{k=K} \alpha_k b^{-k} = \sum_{l=1}^{l=L} \beta_l d^{-l}$$

Suppose that the α_k are known and that we want to calculate the β_1. Multiplication by d yields

$$d \sum_{k=1}^{k=K} \alpha_k b^{-k} = \beta_{(-1)} + \beta_{(-2)} d^{-1} + \ldots + \beta_{(-K)} d^{-K+1}$$

Showing that $\beta_{(-1)}$ is the integer part after the first multiplication. The same reasoning applies to further multiplications.

1.6 Representation of Numbers in Computers

Fixed Point and Floating Point Computers

If the base of the number system is b (integral), the registers in the computer contain, for each digit, devices having either b states or a number of combinations of states > b, b out of which are used. The important thing is to have a one to one correspondence between the numerical value of a digit and the states (or combination of states). If m is the number of digits used, all integers between 0 and b^m can then be represented by combinations of digit-values. Usually of course, the representation is such that the successive devices indicate the numerical value of the digits in positional notation.

Rational fractions could be represented by indicating two integers in a given order. Practically this would not be convenient. Since irrational quantities must be represented by approximations anyway, it is usual to use a limited number of digits in the expansion of the rational or irrational quantity to the base b.

Since the product of two numbers of m digits will have more than m digits, the result of multiplications could not always be held in the registers. To avoid the difficulty, all numbers in a problem can be scaled down so that their absolute value is less than one: this means that a "radix point" (decimal point, binary point) is placed in a fixed position in the register and that all admissible numbers must be such that their non-zero digits lie to the right of this point. It should be noted that "overflow" can still occur in division: it is the task of the programmer to avoid this overflow by proper scaling. A computer using the above system of representation is called a fixed point computer for obvious reasons. Often it is possible to consider a given device as an integral computer (representing only integers, point to the right of the least significant digit) or as a fractional computer (with all numbers scaled down, point to the left of the most significant digit) at will: only the interpretation of the digits has to be modified.

In a floating point computer each number x (fraction or integer) is divided into two parts and written in the form

$$x = zb^y \text{ with } |z| < 1.$$

The registers are then split up and hold z and y separately. Of course, there are limits to the magnitude of the numbers one can represent, since $y < m$ (number of digits in the register). Note that the sign of y must be recorded too.

Floating-point computers are most useful when the magnitude of the numbers involved in a calculation varies widely or when this magnitude is not too well known at the outset, meaning that accurate scaling becomes difficult. Their disadvantage is that fundamental operations like addition or subtraction become quite involved: augend and addend must first be shifted so that their exponents are the same.

Illiac is a fixed-point computer, but it is possible to make it behave like a (slower) floating-point computer by special programming.

Representation of Negative Numbers in Computers

There are two common ways of representing negative numbers in a positive base-system (for negative bases the problem is trivial): as signed absolute values or as complements.

The signed absolute value system is difficult to apply in computers (especially of the parallel type). There are two reasons: in a subtraction the computer has no means of recognizing which term has the higher absolute value, meaning that the sign of the difference may have to be changed after the operation. Furthermore the simple process of "counting down" becomes awkward: one has to sense the passage through zero and then change from subtractions to additions, modifying the sign indication. It is interesting to note that the absolute value system implies a "schizophrenic zero": $+0 = -0$.

In the complement-system the fact is used that the numbers in the registers are always finite, e.g. a 10-decimal-digit integral machine can hold $10^{10} - 1 = 9999\ 999\ 999$ but not 10^{10}: it performs operations modulo 10^{10}. We can therefore add 10^{10} to any number and the machine representation will not change; to represent a given number initially outside the range we can therefore add or subtract integral multiples of 10^{10}. For example we can represent -3 by $-3 + 10^{10} = 0\ 000\ 000\ 007$. As can be seen easily all operations of addition and subtraction can then be performed without contradiction.

Instead of taking the complement with respect to 10^{10} (called ten's complement), we can take the complement with respect to 10^{10} -1 (called nine's complement). This has some technical advantages: all the digits are treated alike. We see that the ten's complement can be obtained from the nine's complement by adding one unit in the least significant digit. Using the nine's complement introduces a "schizophrenic zero" since 0 000 000 000 and 9 999 999 999 represent the same number.

When the sum of two numbers exceeds 10^{10} -1 the machine no longer indicates the sum modulo 10^{10} -1 but modulo 10^{10}: we can correct this state of affairs by adding one unit to the extreme right-hand digit. This procedure is called end-around carry.

All reasonings in the preceding paragraphs can be applied in the binary system. The two interesting complements are then the two's complement and the one's complement. The latter again necessitates the end-around carry and a schizophrenic zero. It has however the advantage that complementation simply means changing zeros to ones and vice-versa: this can be done without going through the adder.

Specific Example of a 40-Digit Binary Fixed-Point Representation (Illiac System)

We shall assume that each register holds 40 binary digits and that the binary points is between the first and the second digit on the left. We shall call the digits $y_0\ y_1 \ldots y_{39}$: then the numbers represented will have the form

$$y_0 \cdot y_1 \cdots y_{39}$$

We shall only represent numbers the absolute value of which is less than one. All positive numbers will then have a machine representation equal to their binary expansion:

$$x = \sum_{i=1}^{39} x_i\, 2^{-i} = 0 \cdot x_1\, x_2 \cdots x_{39}$$

will be represented by setting $y_0 = 0$ and $y_i = x_i$ for $i = 1 \ldots 39$. The highest positive number we can represent in this way is equal to $1 - 2^{-39}$ i.e. slightly less than one.

To represent negative numbers, we add 2. The negative number

$$x = -\sum_{i=1}^{39} x_i \, 2^{-i} = -0 . \, x_1 \, x_2 \, \ldots \, x_{39}$$

will therefore be first transformed into the two's complement which we shall call $\bar{x}$. Then $\bar{x} = x + 2$ i.e.

$$\begin{aligned} \bar{x} &= x + 2^0 + 2^{-1} + \ldots + 2^{-39} + 2^{-39} \\ &= 2^0 + (1 - x_1) \, 2^{-1} + \ldots + (1 - x_{39}) \, 2^{-39} + 2^{-39} \\ &= 2^0 + \sum_{i=1}^{39} z_i \, 2^{-i} \end{aligned}$$

The representation of this is obtained by setting $y_0 = 1$ (2^0) and $y_i = z_i$ for $i = 1 \ldots 39$. The smallest number we can represent is -1. It is now clear why y_0 is called the <u>sign digit</u>: if $y_0 = 0$ the number is positive, if $y_0 = 1$ the number is negative.

Let us examine the general relationship between x and the y_i representation of the machine. For this let us go back to negative numbers:

$$x = \bar{x} - 2 = 2^0 + \sum_{i=1}^{39} y_i \, 2^{-i} - 2 = -1 + \sum_{i=1}^{39} y_i \, 2^{-i} ,$$

while for positive numbers we have simple

$$x = \sum_{i=1}^{39} y_i \, 2^{-i}$$

therefore in all cases

$$x = -y_0 + \sum_{i=1}^{39} y_i \, 2^{-i} \qquad (1\text{-}3)$$

Finally it should be noted how the z_i's have been obtained in the case of negative numbers:

$$\sum_{i=1}^{39} z_i \, 2^{-i} = \sum_{i=1}^{39} (1 - x_i) \, 2^{-i} + 2^{-39}$$

$\sum_{i=1}^{39} (1 - x_i) \, 2^{-i}$ is the one's complement of x: this can be seen by remarking that one's are changed to zeros and vice-versa or by taking the complement of x with respect to $1 - 2^{-39}$. We can summarize by saying: the machine representation of a negative number $- 0 \cdot x_1 \ldots x_{39}$ is the one's complement

1. $(1 - x_1) \ldots (1 - x_{39})$ plus one added in the least significant digit.

CHAPTER II

LOGICAL ELEMENTS AND THEIR CONNECTION

2.1 The Fundamental Logical Elements

We shall call "logical element" or "decision element" a circuit having m inputs $x_1 \ldots x_m$ and n outputs $y_1 \ldots y_n$, each input and each output existing only at two possible voltage levels v_0 and v_1, which will be called "0" level and "1" level respectively. It will be supposed for the moment that all elements are dc-coupled and that the circuits are asynchronous. All lines and nodes can then only exist at the "0" or "1" level.

Each logical element can be defined in a static sense by giving its equilibrium table, i.e. the complete list of simultaneously possible input and output values. This does not necessarily imply that different input combinations give different output combinations or that the output is uniquely determined by the input combination: the element may be a storage element and retain information.

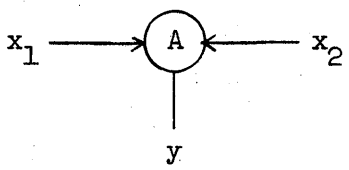
If the equilibrium table contains all possible input combinations and the outputs are uniquely determined by the inputs, we shall speak of a "truth table" and of the element as a "simple logical element" (or combinational element).

In practice the "0" and "1" levels for different lines may be different and instead of associating "0" with the level v_0 and "1" with level v_1 it may be necessary to associate "0" with a voltage range $(\overline{v_0}, v_0)$ and "1" with a voltage range $(\overline{v_1}, v_1)$, the ranges being non-overlapping. Also it may be necessary to speak of current ranges instead of voltage ranges.

AND-Circuit, OR-Circuit, NOT-Circuit and Flipflop

We shall examine in this section four fundamental logical elements, three of them (AND, OR, NOT) being "simple logical elements"; the flipflop being of the storage element type.

AND Circuit



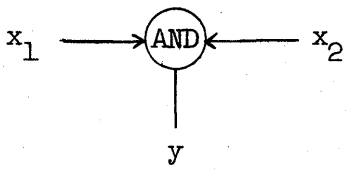
Truth Table

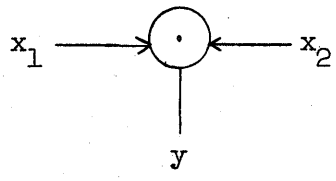
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

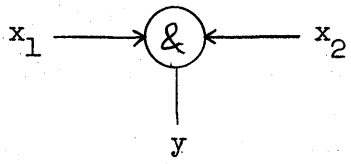
Figure 2-1
AND Circuit

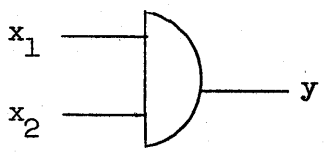
Description: In order for the output y to be a "1" both inputs x_1 and x_2 must be "1".

Remark: Other symbols used for this circuit are:



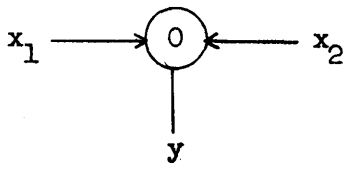






The generalization to multi-input AND's is evident.

OR Circuit



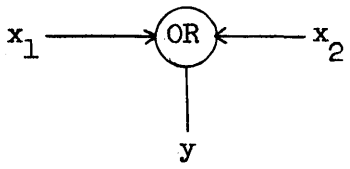
Truth Table

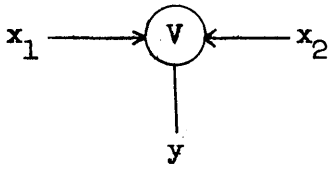
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

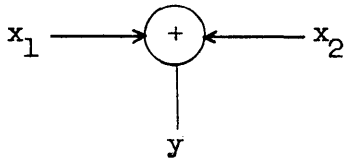
Figure 2-2
OR Circuit

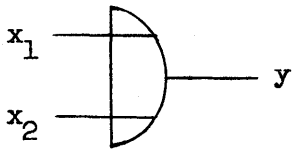
Description: In order for the output y to be a "1" it is sufficient that either x_1 or x_2 be a "1".

Remark: Other symbols used for this circuit are:



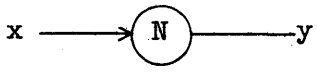






The generalization is multi-input OR's is evident.

NOT Circuit



Truth Table

x	y
0	1
1	0

Figure 2-3
NOT Circuit

Description: The input is the inverse of the output.

Remark: Other symbols used for this circuit are:

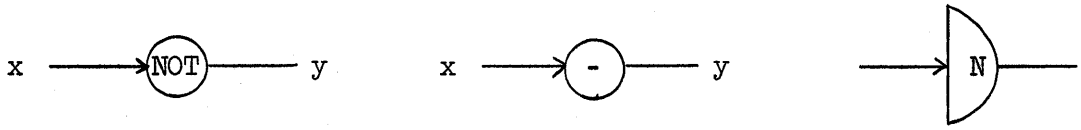


Figure 2-5 below shows how the physical equivalent of these three fundamental circuits can be obtained by the use of diodes, transistors, tubes and relays. It is assumed that two-level dc voltage representation is used with the more positive level corresponding to "1" (so-called "positive logic"). Relays are usually equipped with a contact that is made when the winding is energized ("make" contact) and with one that is broken under these conditions ("break" contact). Figure 2-4 shows these two possibilities symbolically.

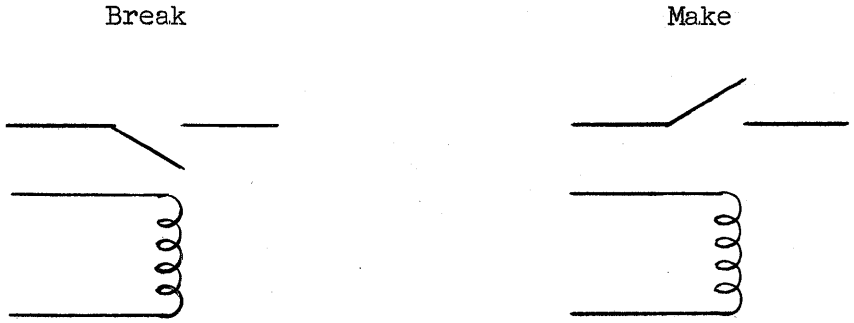


Figure 2-4
Symbols Used for Relays

Note that a diode NOT is not available: This is due to the fact that dc inversion is only possible in amplifiers. It should also be noted that by going from positive logic to negative logic the circuits producing AND now produce OR and vice versa except in the case of relays. The symbol ++ is meant to indicate a voltage in the 10v range, the symbol + a voltage in the 1v range. A similar convention applies to -- and -.

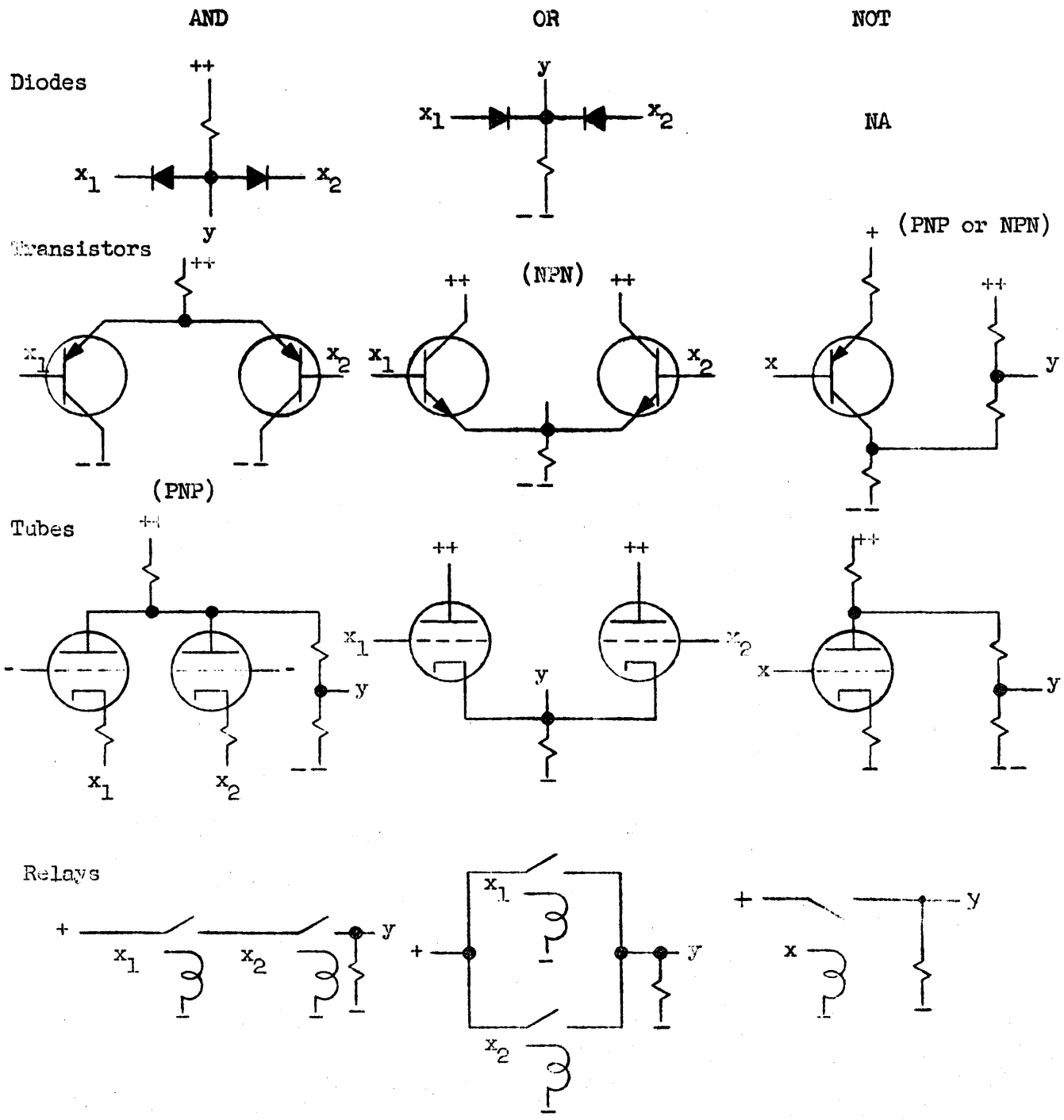
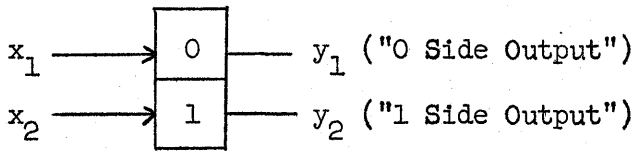


Figure 2-5

AND, OR and NOT and their Hardware Equivalents

for Positive Logic

Flipflop



Truth Table

x_1	x_2	y_1	y_2
0	1	0	1
1	0	1	0
0	0	Last state	
1	1	disallowed	

Figure 2-6
Flipflop

Description: If x_1 and x_2 are different, $y_1 = x_1$ and $y_2 = x_2$ (meaning that $x_1 = 1 \rightarrow y_1 = 1$ etc.). If $x_1 = x_2 = 0$, y_1 and y_2 stay in the preceding state if this has been caused by $x_1 \neq x_2$. If the input transition is from 1 1 to 0 0 the outputs will be different, but the two solutions 0 1 and 1 0 are possible; applying the 1 1 input is therefore not recommended.

The flipflop is an element of fundamental importance because it is able to "remember" the state, once it is set: usually both x_1 and x_2 are kept at "0". When x_1 goes to "1" and back again, the element will remain in the state

$$\left.\begin{array}{l} y_1 = 1 \\ y_2 = 0 \end{array}\right\} \quad \text{called "0" state of the flipflop.}$$

When x_2 goes to "1" and back again, the element will remain in the state

$$\left.\begin{array}{l} y_1 = 0 \\ y_2 = 1 \end{array}\right\} \quad \text{called "1" state of the flipflop.}$$

Practically the input combination 1 1 (and therefore the transition 1 1 $\rightarrow$ 0 0 which leaves the flipflop in an indeterminate state) never occurs. The state of the flipflop will therefore be "0" or "1" (representing the two possible values of a binary digit), according to the preceding combination.

It is interesting to note that a flipflop of the above type can be obtained by a combination of two OR's and two NOT's according to Figure 2-7.

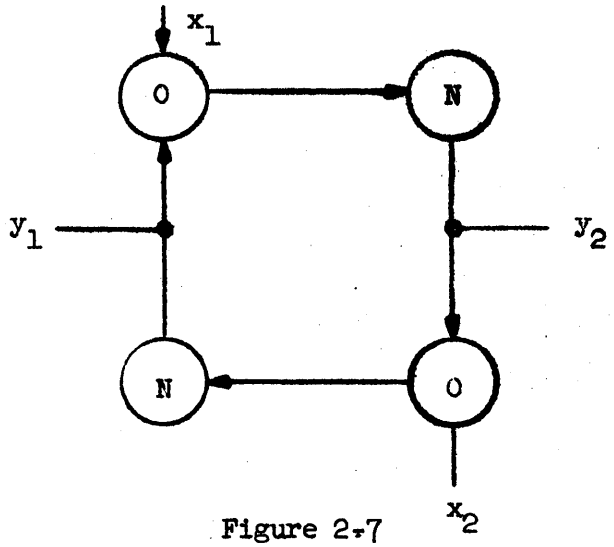


Figure 2-7
Possible Realization of a Flipflop

2.2 Gating - Shifting - Counting

Gating and Shifting

In order to transfer the digits held in one flipflop to another one, we can use the system indicated in Figure 2-8: two AND circuits are used to control the flow of information. For obvious reasons the procedure is called double gating.

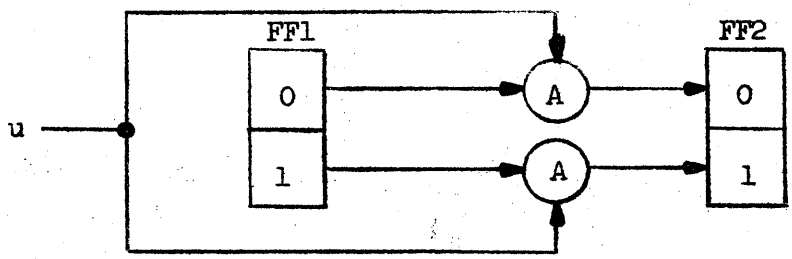


Figure 2-8
Double Gating

When u is made a "1" (gates enabled), the AND circuit connected to the output of the left flipflop which is "1" will apply this to the corresponding input of the right flipflop. When u goes back to "0", the right flipflop stays set.

The other system uses only one AND circuit as a gate but sets the right flipflop to a standard state (e.g. "0") before the gating begins: v is made "1" for a short time and "clears" the right flipflop. After v has gone back to "0", us is made "1". If the state at the left is "1"

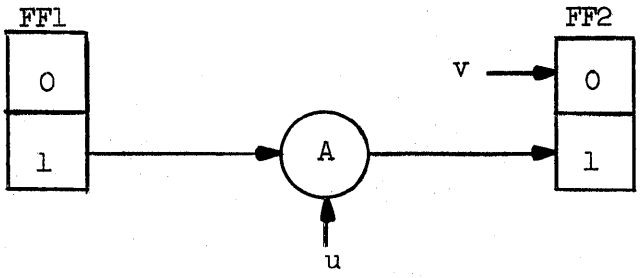


Figure 2-9

Clearing and Gating

the output of the AND circuit becomes a "1" and sets the right flipflop to "1". If the state of the left flipflop is "0", the right flipflop stays in its preceding (cleared) state, i.e. "0". One can, of course, clear to "1" and transfer "0".

The operation of shifting moves the information contained in a register one digital position to the left or to the right. A way to do this is indicated in Figure 2-10, which repeats the pattern of Figure 2-8:

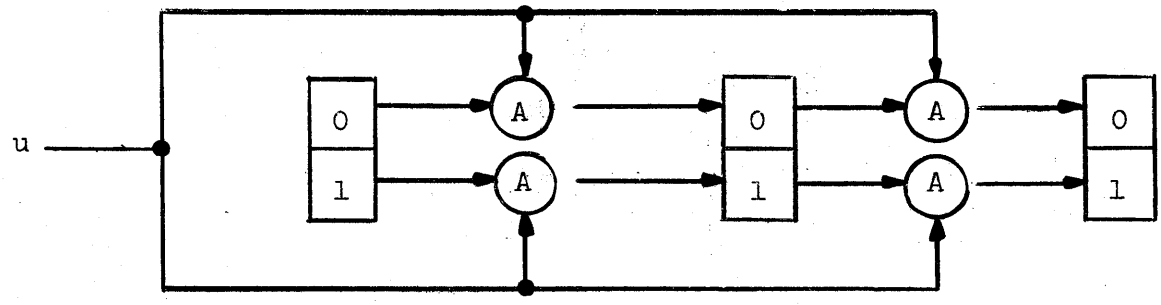


Figure 2-10

Shifting with a Single Register

The duration of the gating signal u must be carefully chosen: if it is too short, no transfer occurs, if it is too long, transfer over two digital positions may take place.

In order to suppress the maximum duration condition, it is feasible to shift in two operations, using an auxiliary register. Figure 2-11 shows the layout. First v is made "1": this produces a transfer of information "straight up". After v has gone back to "0", u is made "1": this produces a transfer of information "right down". The combined effect is that of a right shift. Illiac uses this double-shifting system, i.e. the registers in which shifting is necessary have an auxiliary or "temporary" register attached to them. Instead of using double gating, Illiac uses clearing and gating.

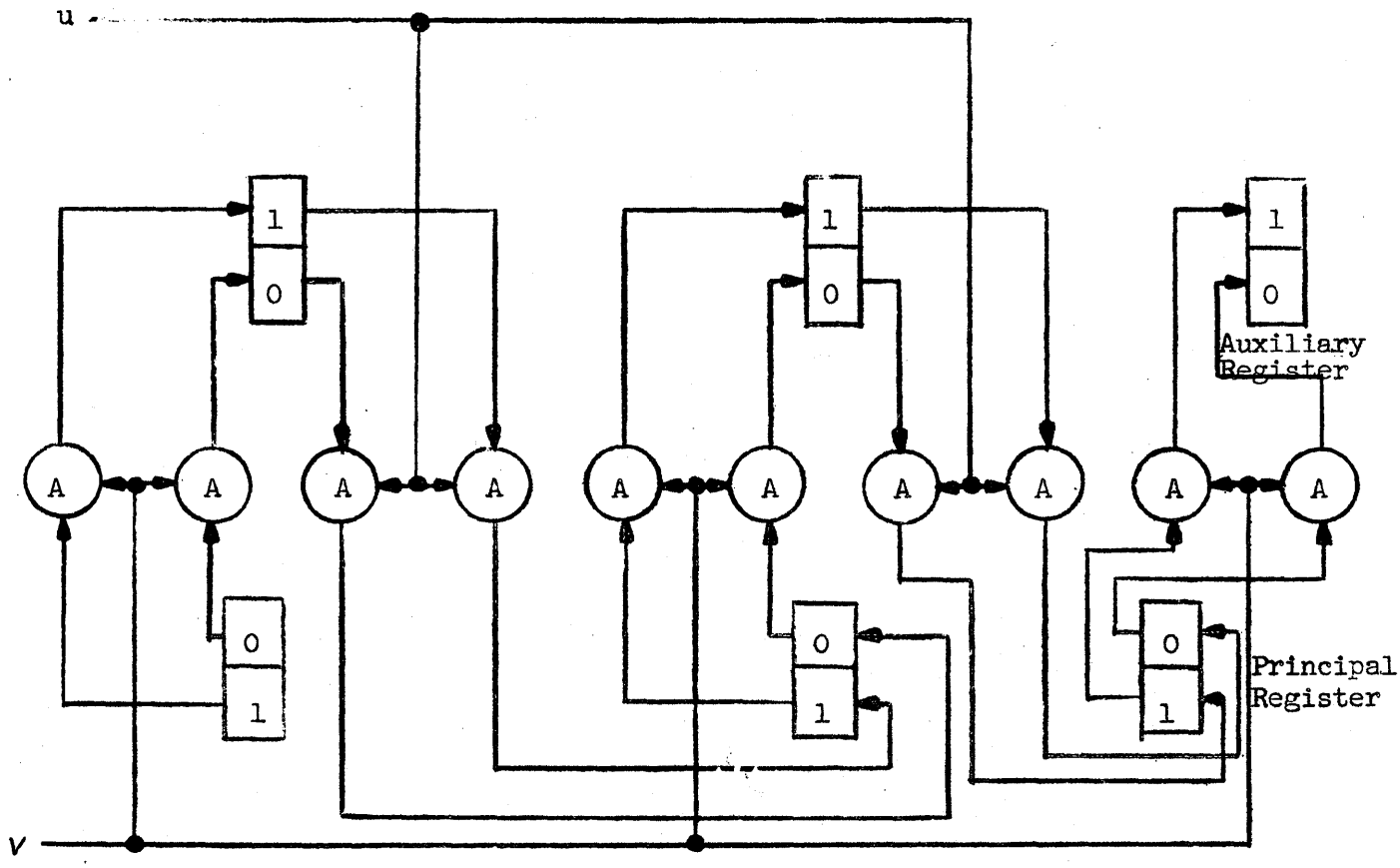


Figure 2-11
Double Shifting

Counting

A binary counter is a frequency divider in the sense that each stage has two inputs ("up" and "down") and two outputs ("up prime" and "down prime") and that for a sequence of "1" signals applied successively to down-up-down-up it gives just one "1" on "down prime" and one on "up prime" i.e. divides the number of applied "1's" by two. With n cascaded stages we can then divide the number of "1's" by 2^n.

One stage of such an asynchronous binary counter is given in Figure 2-12. At the beginning of the process both flipflops (called "false toggle" -F and "true toggle" -T respectively) are set to 0. The "up" pulse sets F to agree with T while the "down" pulse sets T to disagree with F. The sequence of states is thus 0 0, 0 1, 1 1, 1 0 and back to 0 0. If we connect the 0 1 and 1 0 output signals as "up prime" and "down prime" to a similar circuit, we will have achieved a frequency reduction by 2.

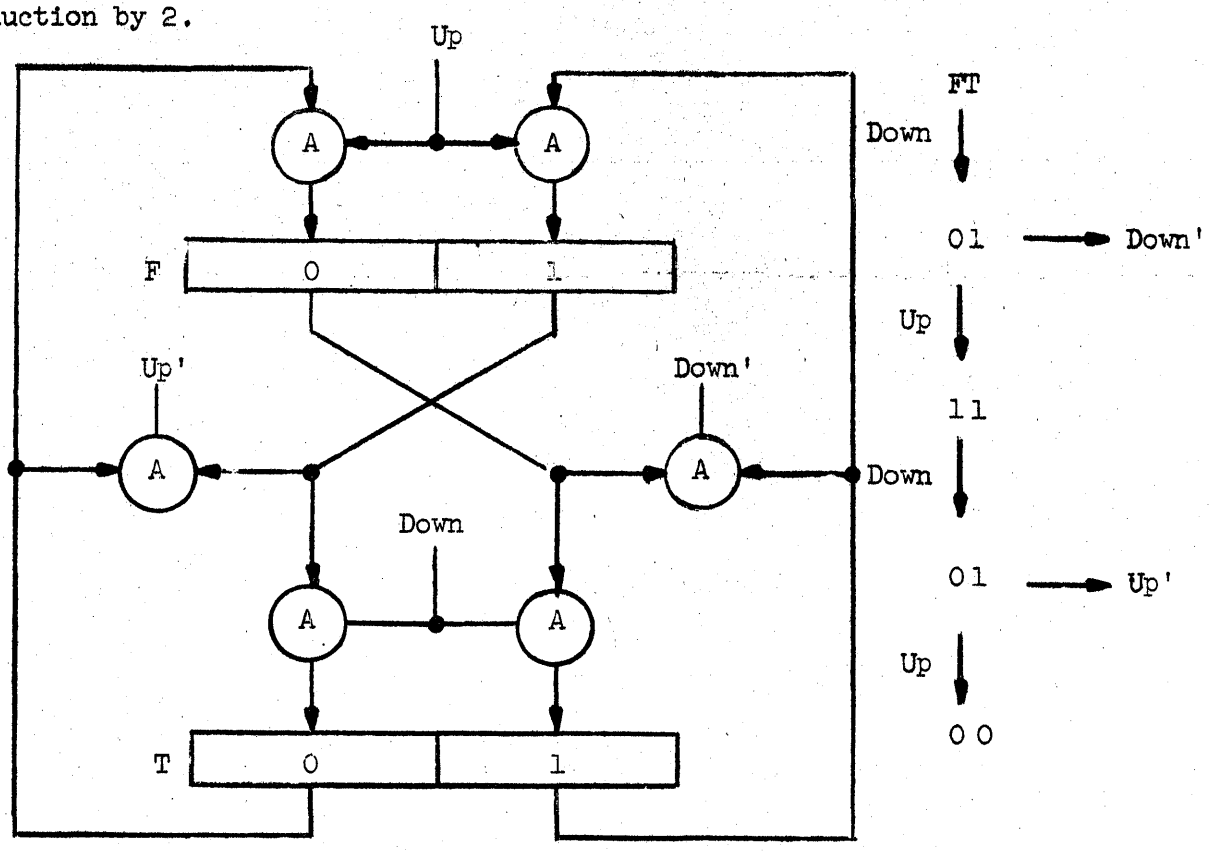


Figure 2-12

One Stage of a Binary Counter

2.3 Adding and Subtracting

When adding two binary digits x_i and y_i we obtain a sum digit s_i and a carry digit c_{i-1}. The relation between x_i, y_i, s_i and c_{i-1} is given by the table below.

Binary Addition Table

x_i	y_i	s_i	c_{i-1}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Later on we shall discuss methods which permit us to find the combinations of AND, OR and NOT circuits having given properties by deductive reasoning. Here we shall simply give the result: Fig. 2-13 shows what is called a "half adder." We see by direct inspection that (as required by the table) c_{i-1} is only "1" when both x_i and y_i are "1." s_i is "1" when x_i or y_i is "1," for then the inputs to the right AND circuit are both ones.

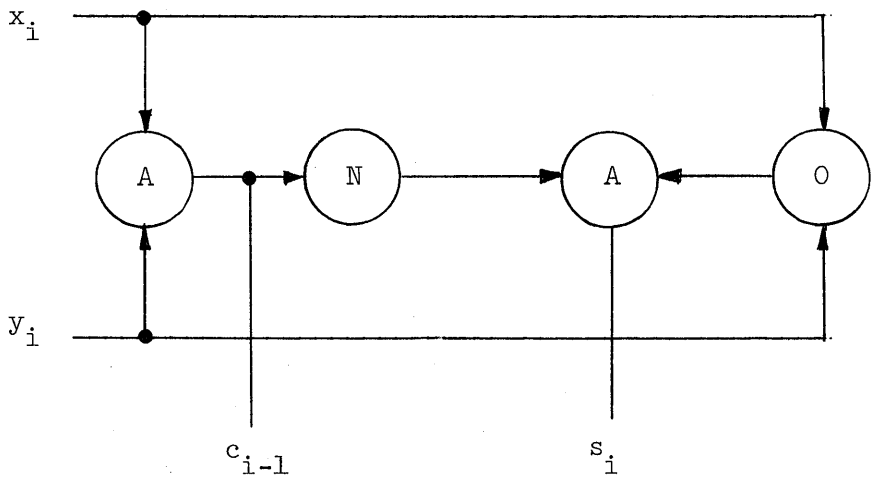


Figure 2-13. Half-Adder

In order to obtain the logical diagram for one digital position of a binary adder, we have to use 2 half-adders since we have to add in the carry c_i from the preceding stage according to the adjoining table. Figure 2-14 gives the layout: as can be seen, the sum output of the

Binary Addition Table with Carry-in

x_i	y_i	c_i	s_i	c_{i-1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

first half-adder and the carry-in are the inputs to the second half-adder. c_{i-1} is taken from either one of the half-adders through an OR circuit this corresponds to the two possibilities of formation of carries.

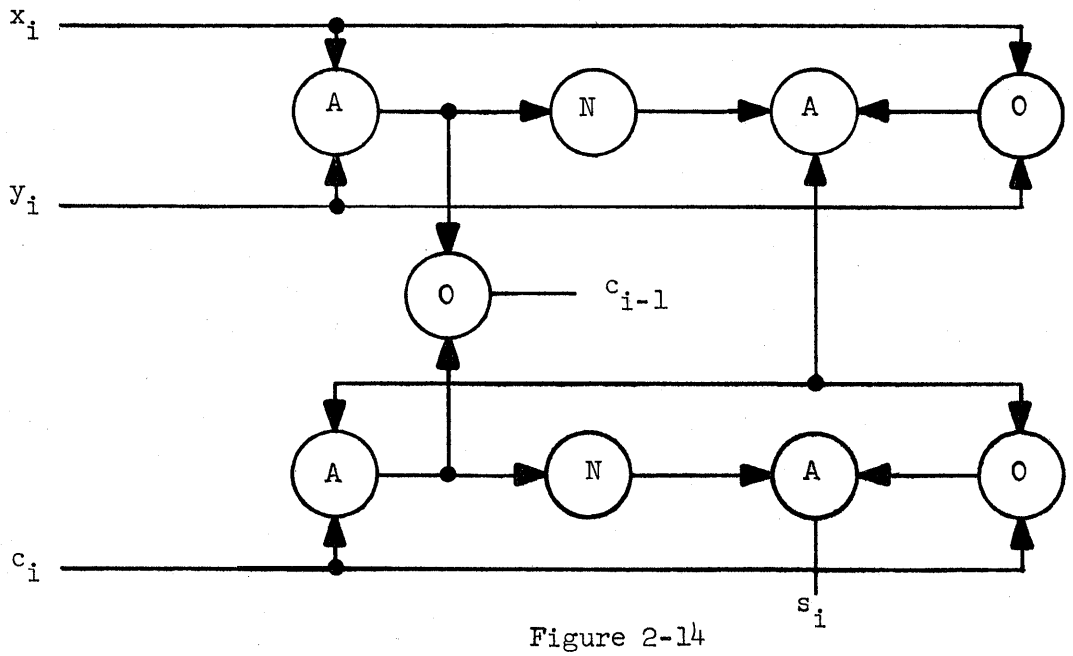


Figure 2-14

Logical Diagram for One Digital Position of a Binary Adder

Complementation and Subtraction

The inputs x_i and y_i in the preceding section come from two flipflops having the same digital position i but pertaining to two different registers. More precisely: x_i and y_i are taken from the "1 side output" of these flipflops. If we want the digitwise complements -- which we shall denote by $\overline{x_i}$ and $\overline{y_i}$ respectively ($x_i = 0$ $\overline{x_i} = 1$, $x_i = 1$ $\overline{x_i} = 0$!) -- we only have to take the "0 side output". We saw in Section 1.6 that negative numbers are represented in Illiac as complements of 2 and that all one has to do to obtain the representation of -0, $x_1 \ldots x_{39}$ is to take the digitwise complement and then add one in the least significant position. Since subtraction is the addition of a negative number, we can switch from the addition x + y to the subtraction x - y by taking as inputs to the adder-stages $\overline{y_i}$ instead of y_i. To add one in the least significant position we provide the stage i = 39 with a carry input (which, of course, is not used in addition). Figure 2-15 shows how, by the use of a complementing circuit using two AND's and one OR per digit position, we can perform additions and subtractions. One adder stage is represented by a box with 3 inputs and 2 outputs. If u = 1 the circuit adds, if u = 0 the circuit subtracts.

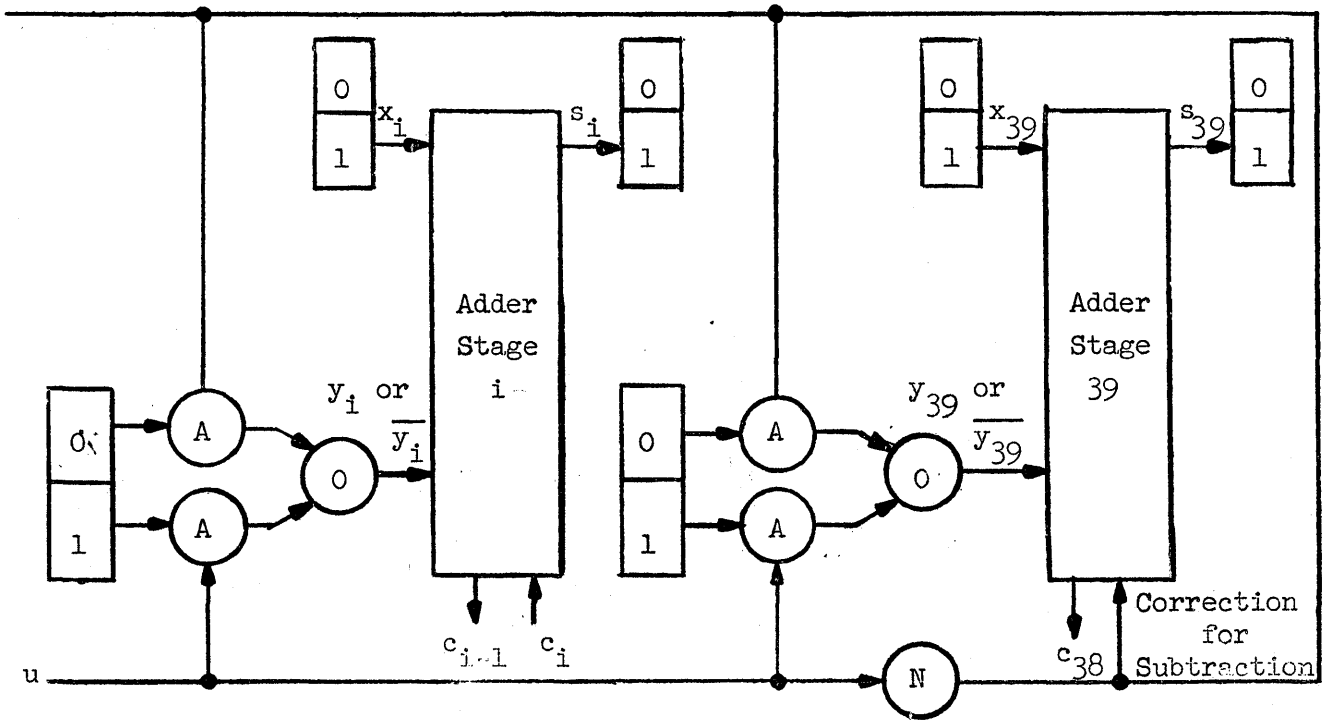


Figure 2-15

Addition and Subtraction Using Complementing Circuits

2.4 Decoding and Sequencing

In 2.2 we examined the clearing and gating procedure. It happens very often in an asynchronous computer of the Illiac type that a sequence of 4 signals "clear-gate-clear-gate" is required, these signals being non-overlapping and the next step being initiated only after we know that the preceding one has been completed. The four-step sequencing circuit of Figure 2-16 shows how the desired result is obtained.

First consider the combination of flipflops and four AND circuits $A_1 \ldots A_4$ i.e. leave the NOT circuits aside. The flipflops give 4 different combinations and for each combination one and only one AND circuit has a "1" output:

FF I	FF II	A_1	A_2	A_3	A_4
0	0	1	0	0	0
1	0	0	1	0	0
1	1	0	0	1	0
0	1	0	0	0	1

This output goes out into other parts of the machine and comes back with a "return-signal" or "reply-back-signal". We can imagine that a certain group of gates is enabled and that one of the gate outputs is used as a return signal. This return signal modifies one and only one flipflop and therefore produces the next combination, i.e. energizes the next AND circuit. If we now put in the NOT circuits (making 3 input AND circuits out of 2 input AND circuits) the next AND circuits can only give a "1" output, if the return signal of the preceding operation has gone back to "0": this guarantees non-overlapping "1" signals at the output of the AND circuits. Notice that connecting the returns to the outputs gives a "free-running" pulser with a 4 phase output.

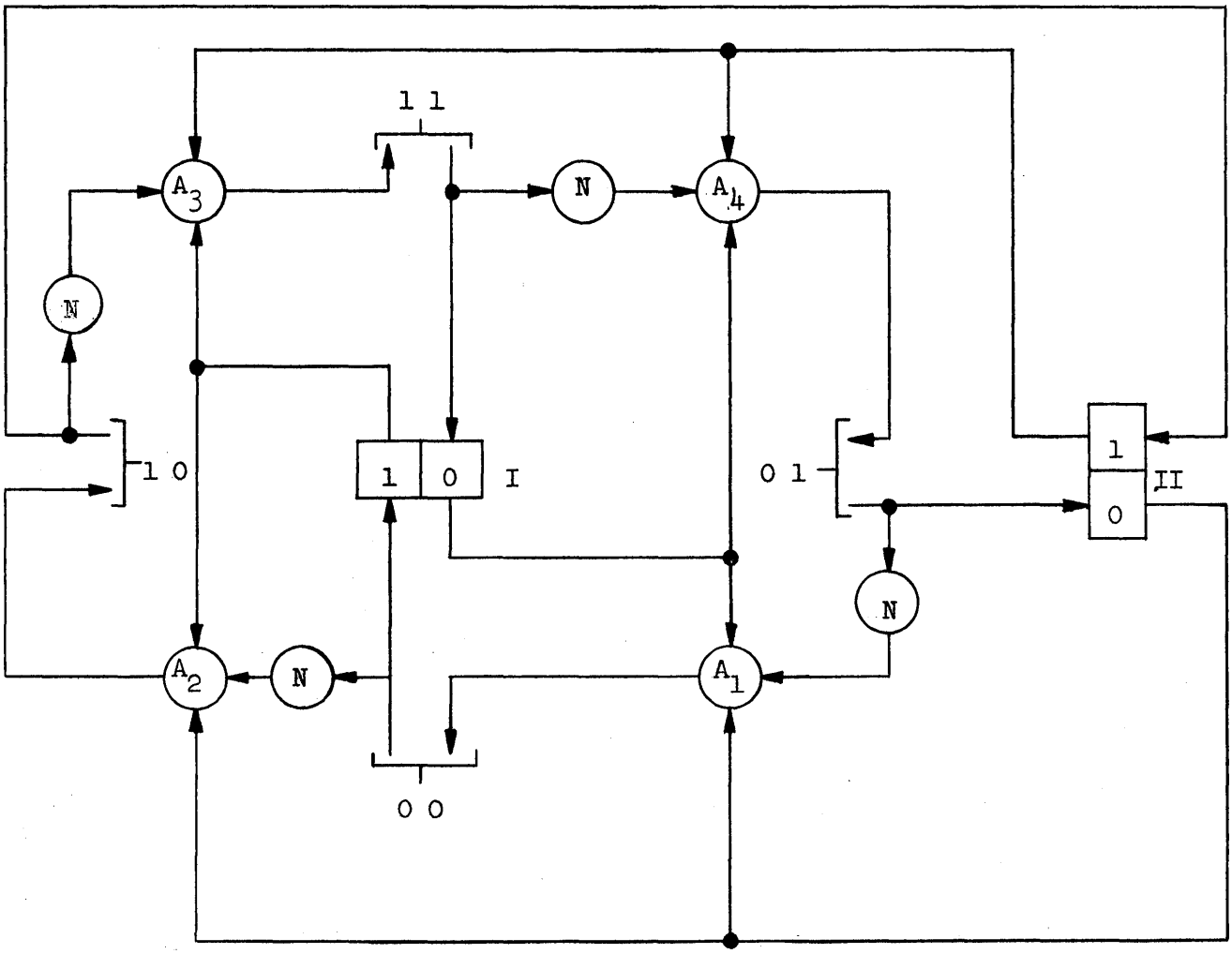


Figure 2-16
Four Step Sequencing Circuit

What is essentially done in the circuit of Figure 2-16 is that the two flipflops are cycled through all combinations of states, that each combination energizes one and only one AND circuit and that this signal (after some delay) steps the flipflops to their next state. This detection of certain combinations of output signals was already encountered in the last section in the asynchronous counter: such a detection of given

combinations of signals is called decoding. The general problem of detecting whether n wires $x_1 \ldots x_n$ have a given combination of zeros and ones can be solved by the use of an n input AND circuit into which are led directly all those wires where a "1" is required, while those requiring a "0" a connected via a NOT circuit. Figure 2-17 shows a decoder for the input combination 10110 on five wires.

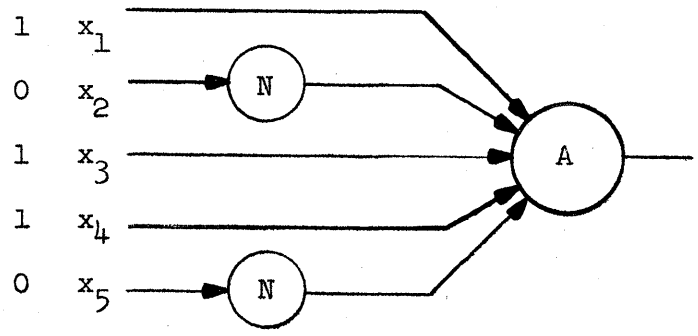


Figure 2-17
Decoder for a 10110 Combination

If it is desired to obtain a "1" output for several different combinations, one can clearly design a circuit as the one shown in Figure 2-17 for each combination and then combine the output of all AND's by a multiple input OR circuit. Figure 2-18 shows a circuit giving a "1" for the 3 input combinations 1111, 1101 and 0000.

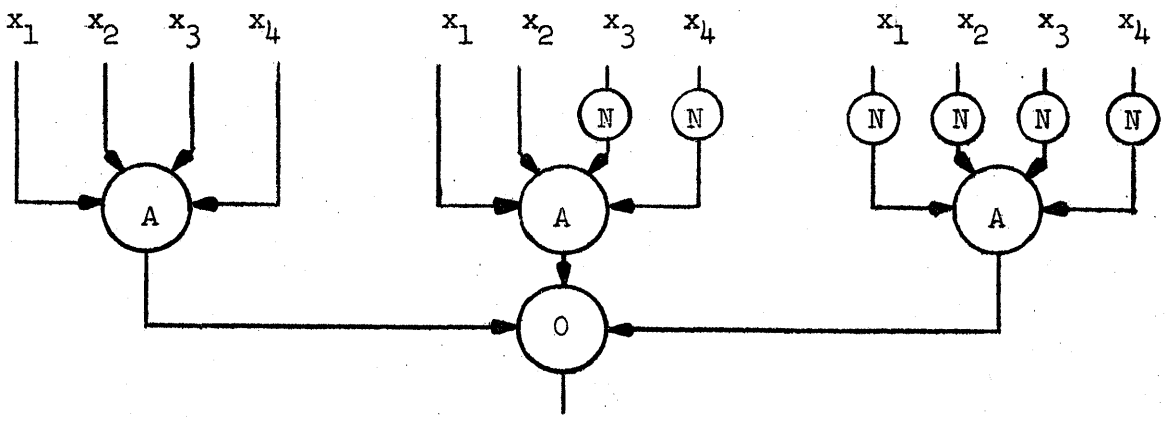


Figure 2-18
Circuit Detecting Several Input Combinations

Since visibly for n inputs $x_1 \ldots x_n$ the possible input lines to the AND's are either direct or complemented (i.e. inverted), all possible combinations n at a time can be formed by providing 2n lines x_1 $\overline{x_1}$, x_2 $\overline{x_2}$ ... etc. and having one wire of each pair go to an n-input AND circuit. It is customary to symbolize such a decoding circuit by a matrix of 2n lines (the horizontal lines in Figure 2-19) connected to x, $\overline{x_2}$ ... $\overline{x_n}$ intersected by a second set of lines (the vertical lines in Figure 2-19) which symbolize the AND function, which input being used being determined by a dot at the appropriate intersection. Such a circuit is called a <u>matrix</u> circuit for obvious reasons.

<u>Remark</u>: Often the combination of the diverse AND outputs by an OR is symbolized by a line parallel to the 2n lines with short segments determining the choice of OR inputs. Figure 2-19 repeats 2-18 in this notation.

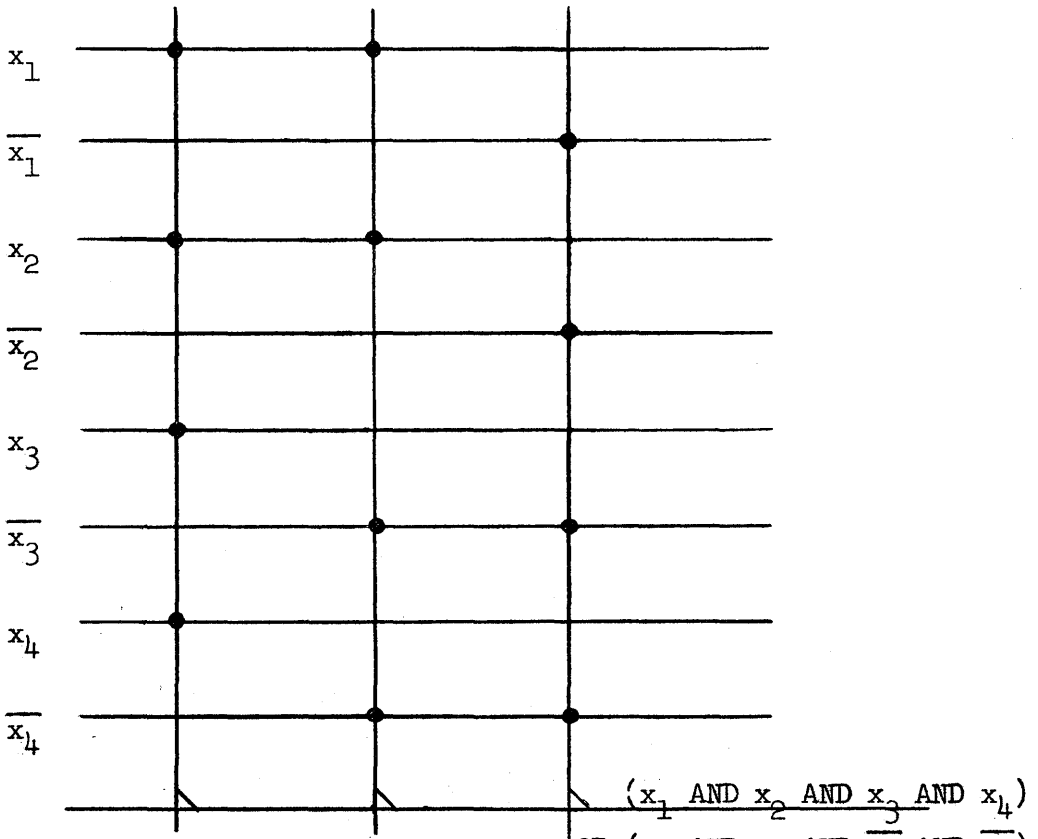


Figure 2-19
Matrix Circuit

Often it is useful to introduce the notion of complexity of a circuit by the rule

$$\text{complexity} = \text{total number of inputs.} \qquad (2\text{-}1)$$

Supposing that we wanted to form all 2^4 input combinations in Figure 2-19, we would visibly need $2^4 \times 4 = 64$ inputs i.e. the complexity for an n-input circuit would be $2^n \times n$. It turns out that for $n > 3$ it becomes advantageous to decode by a tree or pyramid as shown in Figure 2-20. It is not too difficult to show that here all input combinations can be formed with complexity $2^{n+2} - 8$ which is less than $n2^n$ if $n > 3$.

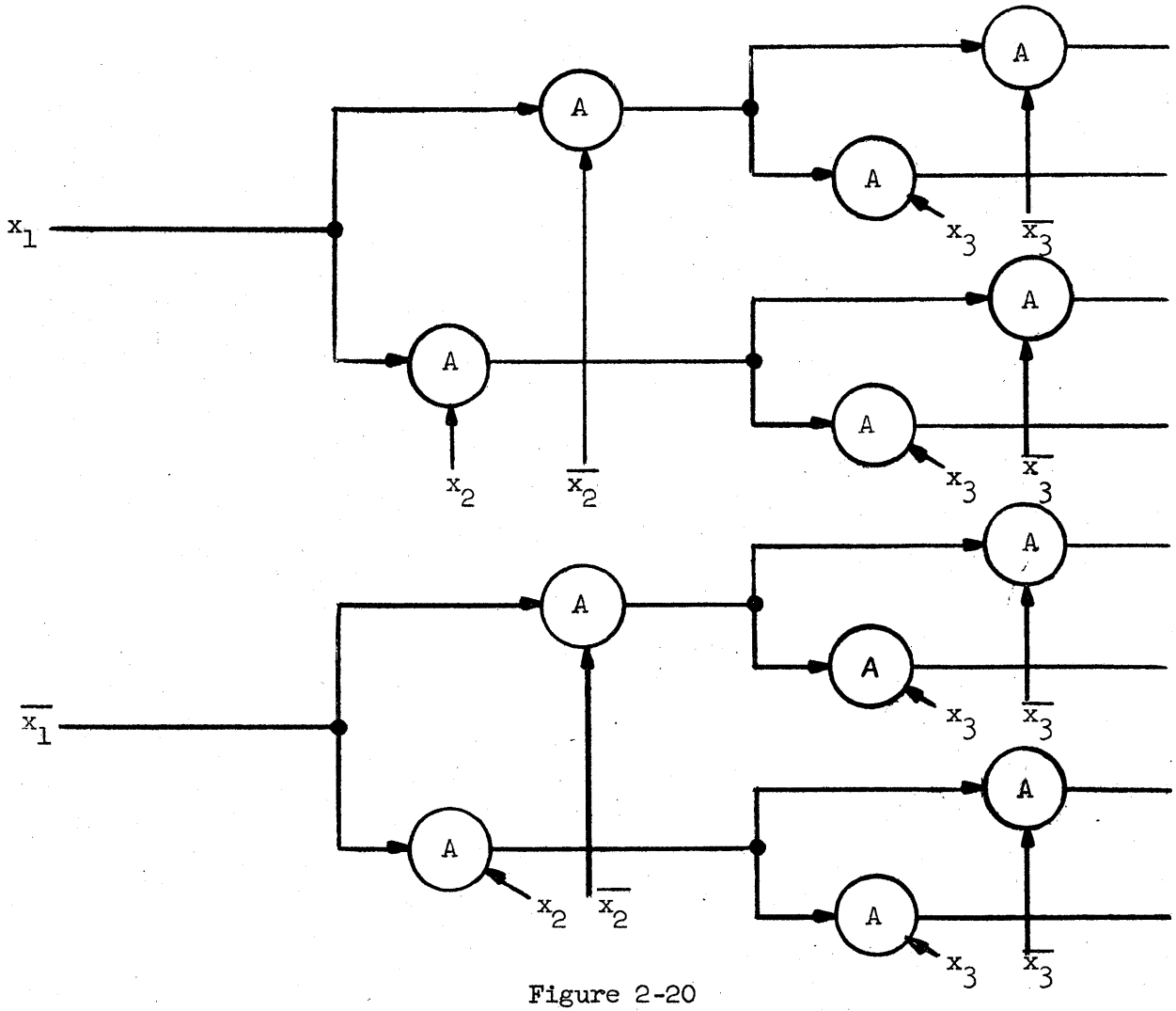


Figure 2-20

Tree or Pyramid Forming all Combinations of 3 Variables

2.5 Complex Logical Elements

The preceding sections have shown that all the fundamental operations in a computer can be done using AND, OR, NOT and FF elements, the latter being actually a feedback combination of OR and NOT. It is easy to show that a single element, namely an AND-NOT or an OR-NOT (NOR) is sufficient to perform all functions: in order to do this we only have to show that AND, OR and NOT can be constructed. Figure 2-21 shows how AND-NOT's can be used.

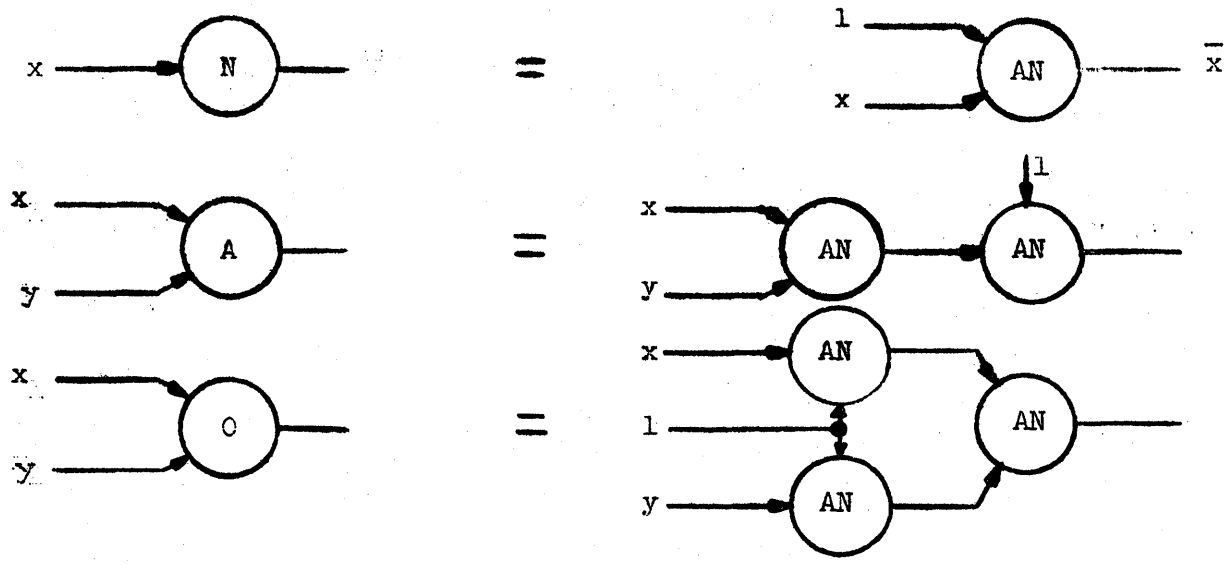
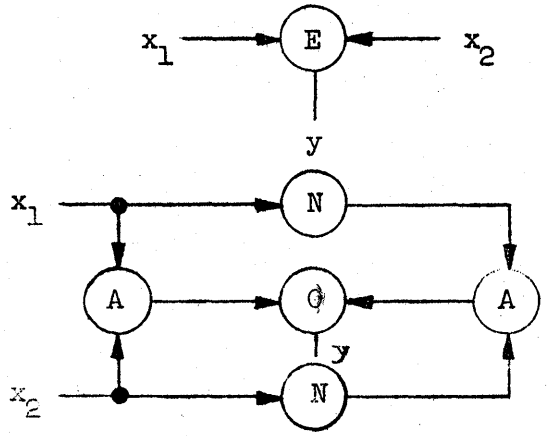


Figure 2-21
AND-NOT Equivalents of NOT, AND and OR

Usually it is not very wise to reduce all functions to combinations of AND-NOT or OR-NOT. To the contrary: designers often introduce new elements which can be made up out of simpler ones but which occur so often that a special name is given to them. We shall introduce them by their truth table together with an equivalent combination of AND's, OR's and NOT's.

Equivalence Circuit



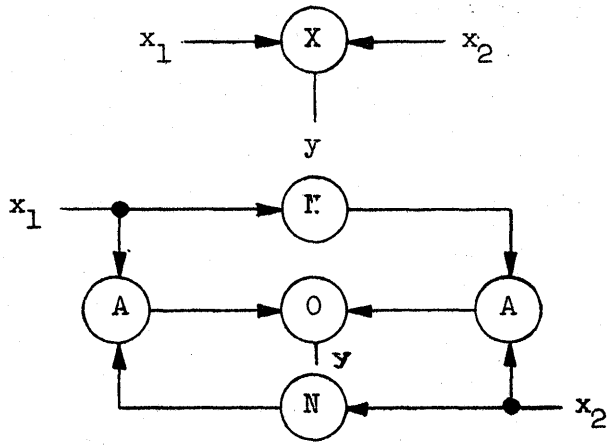
Truth Table

x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	1

Figure 2-22
Equivalence Circuit and Equivalent

Description: The output is "1" if and only if the two inputs agree.

Exclusive OR



Truth Table

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2-23
Exclusive OR and Equivalent

Description: The output is "1" if one or the other but not both of the inputs are "1".

Remark: It is easily seen that an equivalence circuit becomes an exclusive OR or vice-versa if one of the inputs is inverted.

Complementary Flipflop

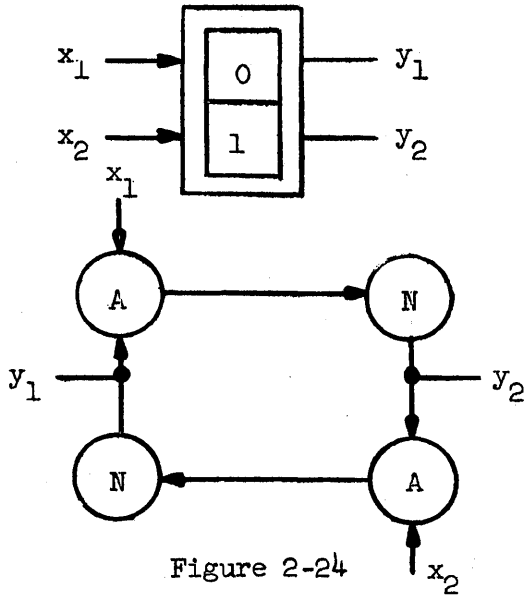


Figure 2-24

Complementary Flipflop and Equivalent

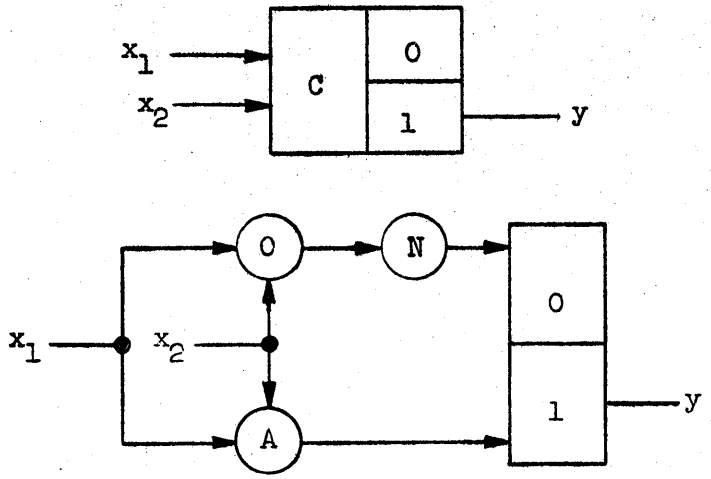
Truth Table

x_1	x_2	y_1	y_2
0	1	0	1
1	0	1	0
1	1	last state	
0	0	disallowed	

Description: If x_1 and x_2 are different, $y_1 = x_1$ and $y_2 = x_2$. If $x_1 = x_2 = 1$, y_1 and y_2 stay in the preceding state if this has been caused by $x_1 \neq x_2$. If the input transition is from 0 0 to 1 1 the outputs will be different, but the two solutions 0 1 and 1 0 are possible; applying the 0 0 input is therefore not recommended.

Remark: The complementary flipflop differs from the flipflop discussed in 2.1 by the interchange of 1 1 for 0 0 for the "hold" condition and that of 0 0 for 1 1 for the disallowed condition.

C Element



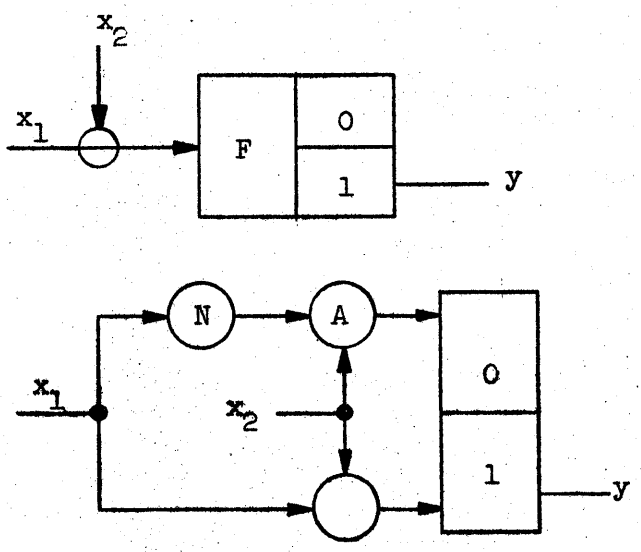
Truth Table

x_1	x_2	y	
0	0	0	
1	1	1	
0	1		} last state
1	0		

Figure 2-25
C Element and Equivalent

Description: When x_1 and x_2 coincide, the output follows the input. When $x_1 \neq x_2$, the last state is remembered.

F Element



Truth Table

x_1	x_2	y	
0	1	1	
1	1	1	
0	0		} last state
1	0		

Figure 2-26
F Element and Equivalent

Description: If $x_2 = 1$, the output follows the input x_1. If $x_2 = 0$, the last state is remembered.

It will become apparent in the discussion of whole systems of logical elements that it is not possible to use great numbers of cascaded AND's or OR's (i.e. such circuits connected in series) because in many such circuits (diode circuits as shown in Figure 2-5) the signals are slowly thrown out of the permissible bands due to voltage drops etc. In order to "renormalize" such a signal it becomes then necessary to insert an amplifier or "level restorer". This can only be circumvented if a NOT circuit is present in the chain: we know from Figure 2-5 that this implies amplification. The two symbols of Figure 2-27 represent renormalizing amplifiers. It should be noted that logically these circuits have the same properties as a piece of wire, i.e. $y = x$.

Figure 2-27
Symbols for an Amplifier or Level Restorer
(Non-inverting)

2.6 Sophisticated Adding, Counting and Sequencing

Separate Carry Storage

It is easily verified that the 1/2 adder AND-NOT-AND-OR combination of Figure 2-13 can be replaced by an exclusive OR in parallel with an AND: the latter will give the carry while the exclusive OR gives the sum. Figure 2-14 can therefore be redrawn as in Figure 2-28.

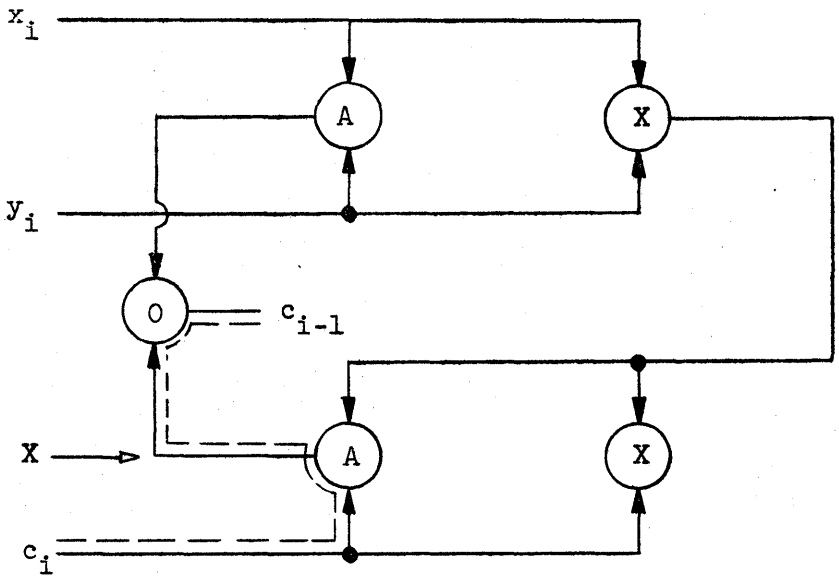


Figure 2-28

One Digit Position of a Binary Adder Using Exclusive OR's

Quite visibly c_i influences c_{i-1} via the dotted path: this corresponds to the well known fact that if we add 0 and 1 in a given stage and there is a carry from the last digit position, there will be a "propagated carry". Under some circumstances a carry can possibly be propagated through the whole register i.e. from the least significant digit to the most significant digit. Such a propagation can take a great amount

of time and operations in which repeated additions occur (like multiplication) are excessively slowed down. A way around this difficulty is to sever the carry propagation path in X and dump the output of the AND into a separate flipflop. If we make the input to the OR "0" we shall then simply have a "pseudo-sum" coming out of s_i while the carry is stored separately; considering the whole adder and its registers we would then have a register holding x_i's, one holding y_i's, a "pseudo-sum" register holding the s_i's and finally a carry storage register holding the output - say b_{i-1} - of the lower AND. At each moment the read sum could be obtained by adding the "pseudo-sum" to the separate carries.

In order to be useful in repetitive addition it is desirable to have an adder which allows a number to be added to another one stored in the separate-carry-pseudo-sum manner. It is clear that this can be achieved by using the arrangement of Figure 2-29 in which the OR circuit is used to absorb the carries from a previous addition. The signal z_{i-1} coming out of this OR visibly only affects the next stage since the carries out of stage i (i.e. the signal b_{i-1}) is again stored separately. Figure 2-30 gives the connections to be used if the number in X is to be added again and again to itself. Initially registers C and Y are cleared and then they hold successively (in pseudo-sum-separate-carry form) 2, 3 etc. times the contents of X: They correspond to what is ordinarily called the accumulator. Registers B and S are - together - the temporary accumulator. By alternating between the up and the down gates, we can cycle through as many additions as desired. At the end the sum is obviously obtained in two parts and more equipment is needed to "absorb the carries". One way of doing this is to use the contents of C and Y as the inputs to a classical adder.

Borrow Storage Counter

It is easy to see that problems of carry propagation also affect the counter of Figure 2-12, i.e. its speed of counting is limited by a possible carry propagation over all stages. D. E. Muller of the University of Illinois has extended the idea of separate carry storage

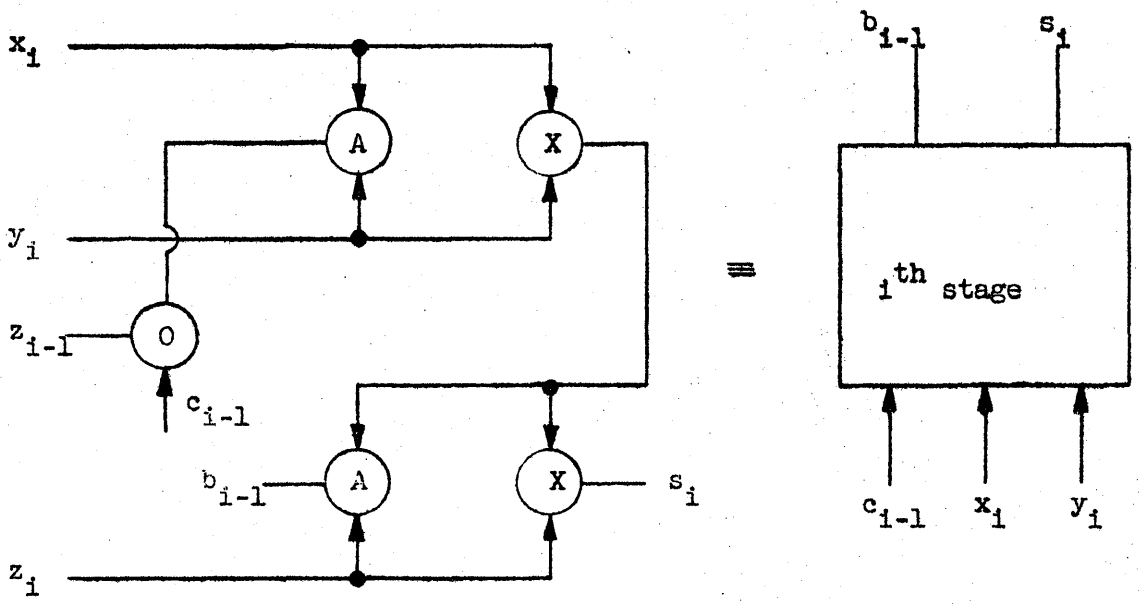


Figure 2-29

One Stage of a Separate Carry Storage Adder

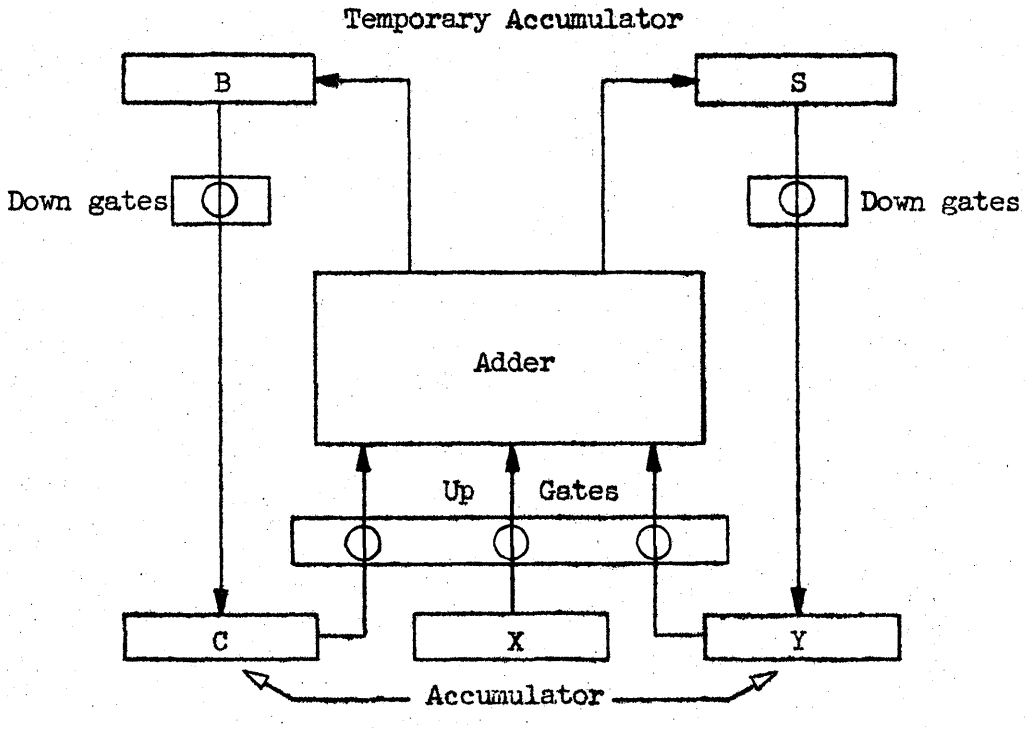


Figure 2-30

Accumulator and Adder in a Separate Carry Storage System

to counters. Figure 2-31 shows the last two stages of such a counter. There are again two principal flipflops per stage: The true toggle t_i and the false toggle f_i and they are connected in the usual fashion, i.e. a down shift (DN-pulse) sets f_i to agree with t_i while an up shift (UP-pulse) sets t_i to disagree with f_i. There is, however, a major difference: no decoding is used to obtain frequency division and furthermore the counter counts down from a number initially set in $\ldots\ b_2\ b_1\ b_0$. This counting down would visibly necessitate borrows at certain stages of the game: These borrows are stored separately in $\ldots\ a_2\ a_1\ a_0$ (or, after a down shift, in $\ldots\ c_2\ c_1\ c_0$). The effect of such a borrow is to permit a shift from f_i into t_i and simultaneously a_i, while $c_i = 0$ inhibits this transfer and sets a_i to 0. One can see (see table below) that if to these rules we add an "unconditional" last stage in which t_0 and a_0 always receive the complement of f_0 on an up shift, the result will be a counting operation in which the number held at any given moment is

$$[\ldots\ t_2\ t_1\ t_0] - 2\ [\ldots\ a_2\ a_1\ a_0]$$

At the beginning all registers are cleared to 0 and the number n to be counted down from is set into $\ldots\ t_2\ t_1\ t_0$. At the end (i.e. after n UP and n DN pulses) the upper register indicate zero. One more down pulse is sufficient to also clear the lower register to zero, thus readying the counter for a new counting operation. In the table below the state of all flipflops is shown in counting down from 3. The column t indicates $[\ldots\ t_1\ t_0]$ while a indicates $[\ldots\ a_1\ a_0]$.

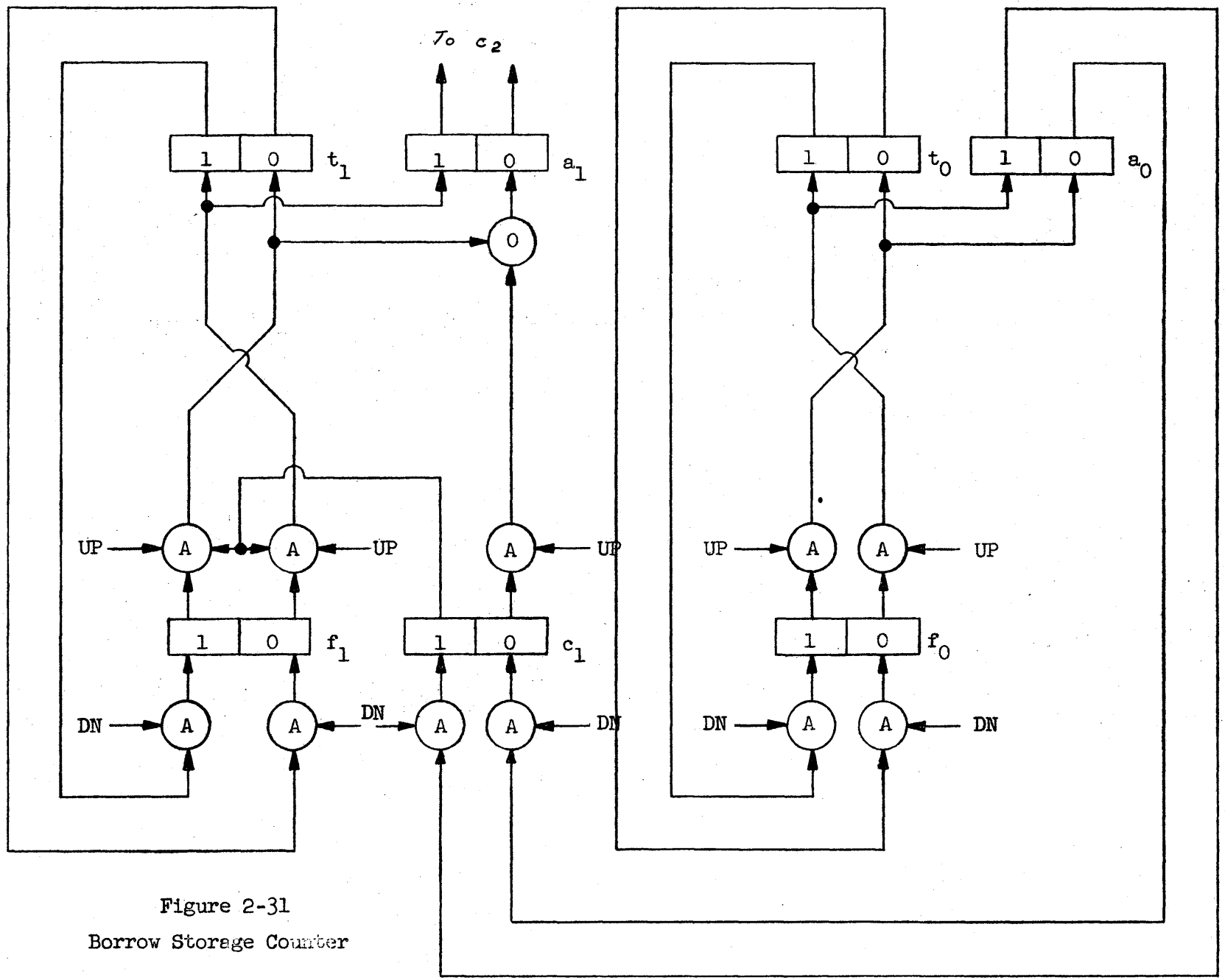


Figure 2-31

Borrow Storage Counter

Table 2-1

Operations in a Borrow Storage Counter

t_1	a_1	t_0	a_0	t	2a
f_1	c_1	f_0			
1	0	1	0	3	0
0	0	0			
1	0	1	0	3	0
1	0	1			
1	0	0	0	2	0
1	0	1			
1	0	0	0	2	0
1	0	0			
1	0	1	1	3	2
1	0	0			
1	0	1	1	3	2
1	1	1			
0	0	0	0	0	0
1	1	1			
0	0	0	0	0	0
0	0	0			

Interlaced Sequencing

It is often necessary to alternate a given operation (Op 3) with two other (Op 1 and Op 2) in such a fashion that if and only if both Op 1 and Op 2 have occurred it becomes possible to do Op 3. Vice versa: Op 3 must be terminated before Op 1 or Op 2 can even start. In such circumstances we speak of interlacing and write

$$\left.\begin{matrix}\text{Op 1}\\ \text{Op 2}\end{matrix}\right\} \text{Op 3} \left\{\begin{matrix}\text{Op 1}\\ \text{Op 2}\end{matrix}\right\} \text{Op 3} \quad \ldots.$$

Figure 2-32 shows a possible sequencing circuit having all the required properties. It is "speed independent" in the sense that no requirements whatsoever have to be placed on the relative speeds of operation of the logical elements. We can think of the boxes marked Op 1 etc. as being simply in-phase amplifiers introducing a certain time lag (equal to the time required to do the corresponding operation).

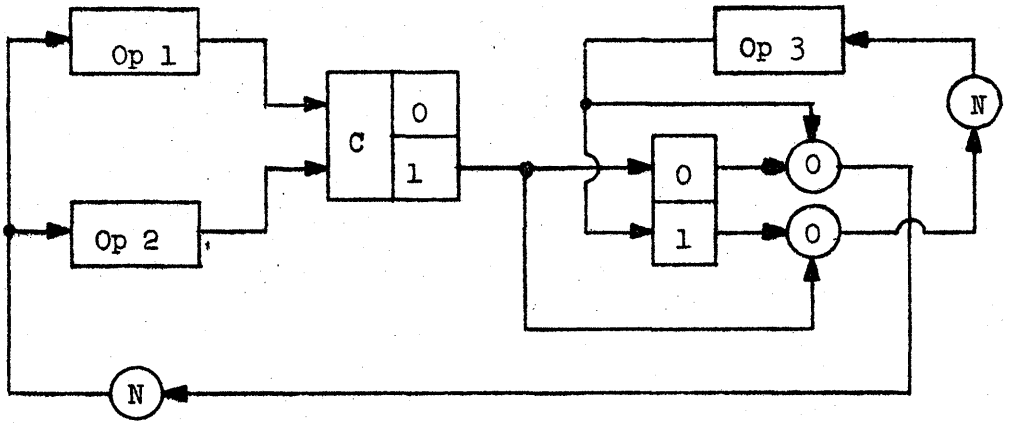


Figure 2-32
Circuit for Interlaced Sequencing

The operation of this circuit is as follows. Suppose that Op 1 and Op 2 have occurred, injecting two "1" signals into the C-element: The output of this element now sets the flipflop into the "0" state thus making the input to the lower NOT "1" and the input to Op 1 and Op 2 "0" (after some time this makes the upper input to the flipflop "0" again). As the flipflop changes state, its lower output becomes zero and this zero, together with reply back zero mentioned above, finally allows the upper NOT to energize the input to Op 3. This sets the flipflop back into the one state, thus cutting off the input to Op 3 and after the output of Op 3 has also gone back to zero the lower NOT receives a zero input and starts up Op 1 and Op 2 again.

2.7 Dynamic (Synchronous) Logic

Up to now no major difficulties resulted from the fact that no information concerning the operation time of individual logical elements was available: we talked essentially about asynchronous circuitry. Very often savings in both time and equipment can be obtained by specifying the delays signals suffer in the logical circuitry, at least to the extent of making sure that an ordering relationship is known i.e. if two parallel signal paths are present it is known which one is faster. Often such an ordering is obtained by inserting into one of them suitably chosen delay elements. We shall discuss below some of the more common dynamic circuits.

Delay Element

Figure 2-33
Delay Element

The delay element shown in Figure 2-33 is essentially an amplifier which is slowed down by capacitive loading of the output or intermediary points or it is a transmission line formed of lumped L and C elements adjusted to give a given delay between the input and the output. In the following discussions we shall assume that delay elements have amplification. Often Δ indicates the time delay in seconds.

Free Running Multivibrator (Clock)

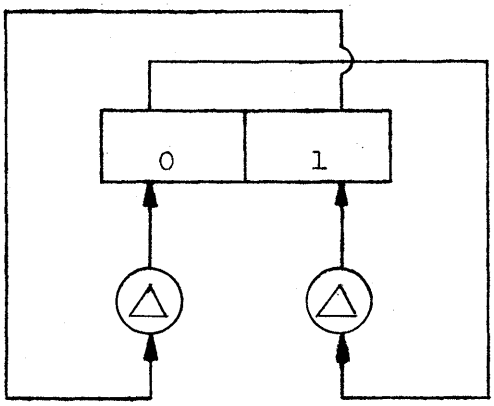


Figure 2-34
Free Running Multivibrator

Figure 2-34 shows a flipflop whose outputs are coupled back to the opposite inputs via delay elements (here we shall assume them equal). Visibly the operation cycle consists of the following steps: suppose that the flipflop has just been set into the "1" state. After a given delay Δ the new outputs i.e. 0 1 will arrive at the input in the form 1 0 and switch the flipflop back to the "0" state. This gives a 1 0 output which comes back - after the delay - in the form 0 1 which again sets the flipflop to "1".

If the setting time of the flipflop can be neglected, the oscillations at either the "0" or the "1" side of the flipflop are as shown in Figure 2-35; they have the period 2Δ . The fact that the

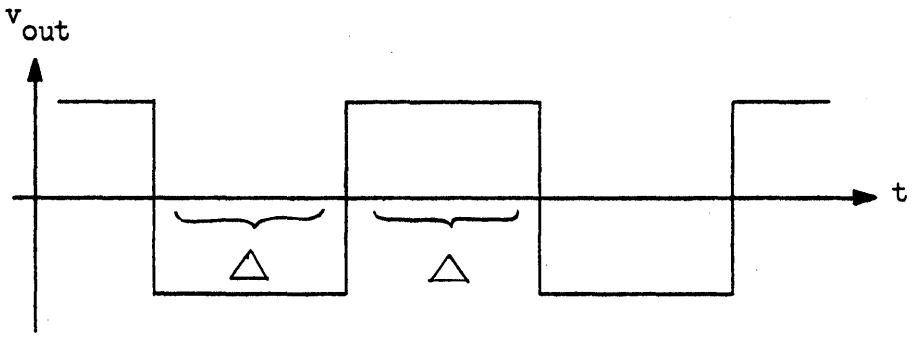


Figure 2-35
Symmetric Oscillations of a Free Running Multivibrator

pulses are of regular duration makes such a free running multivibrator useful as a clock, i.e. a timing control for the operations in a computer. In the circuits below we shall often assume the existence of such a clock.

Actually there is trouble in the circuit of Figure 2-34 if the two delays are different and since it is impossible to design these delays to be exactly equal, it is better to make provision for the more general case. Figure 2-36 shows a possible solution and Figure 2-37 the waveforms. Note that this time the two outputs, X and Y, have no longer the same shape. The operation of this circuit can be understood from the "interlaced sequencing" circuit of Figure 2-32, except that Op 2 does not exist, making the C-element useless.

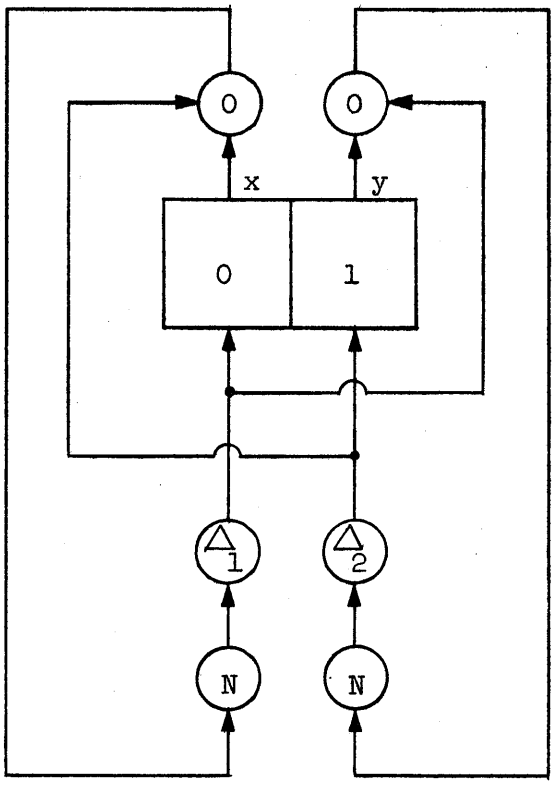


Figure 2-36

Asymmetric Free Running Multivibrator

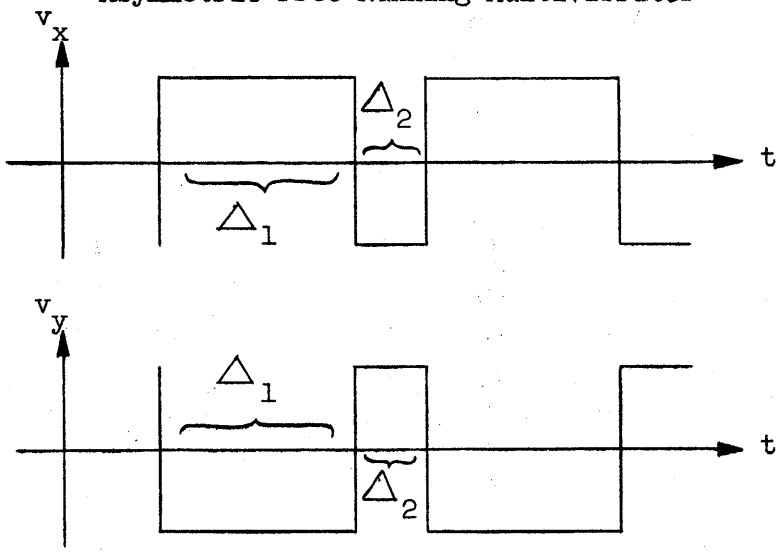


Figure 2-37

Asymmetric Oscillations of a Free Running Multivibrator (Positive Logic)

Regenerative Broadening

A common problem is to lengthen a pulse to make it as long as a clock pulse, i.e. to design a circuit which, if at the beginning of the clock pulse a "1" is present, stretches this "1" to the full extent of the clock pulse even if the sampled pulse disappears during this clock pulse. Figure 2-38 shows such an arrangement using an AND and an OR. Note that practically an amplifier is needed in the feedback loop.

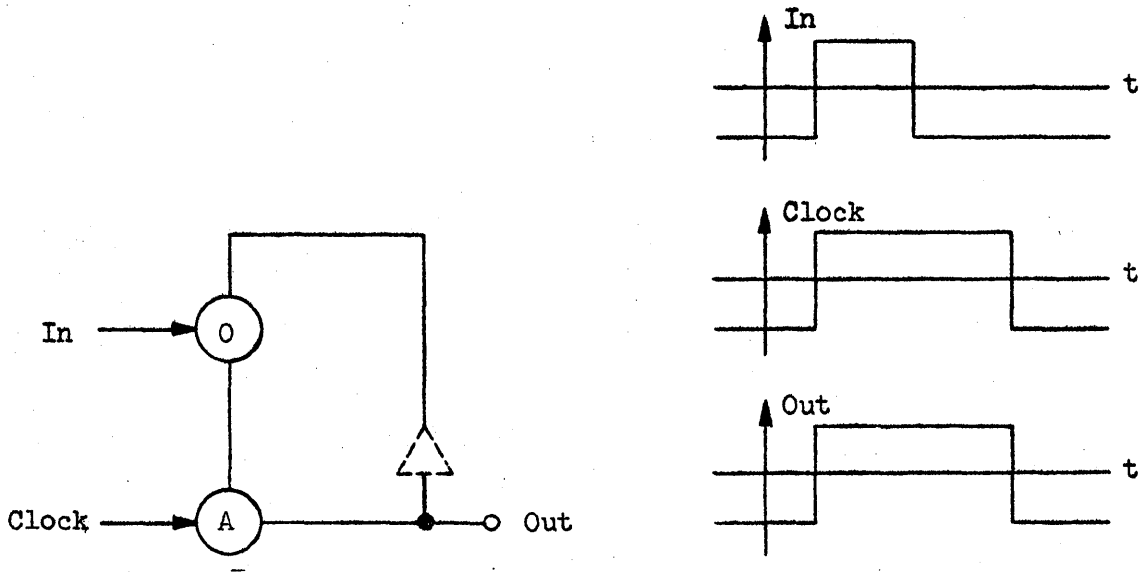


Figure 2-38
Regenerative Broadening

Latch Circuit

A more sophisticated version of the circuit described above is the latch, which differs in that the clock pulse cannot only "capture" a "1" and hold it even if the input goes back to "0", but also "capture" a "0" and hold it in the event that the zero actually changes to a one during the clock pulse. Figure 2-39 shows the layout: note that here a delay has to be used in order to make sure that the AND in the feedback

loop can receive a "1" from the clock and also a "1" from the input sampled while the clock was still "0" (Clock = 1 → $\overline{\text{Clock}}$ = 0 and vice-versa!). It should also be remarked that this delay should be just long enough to allow sufficient overlap to start regeneration. Often the delay is obtained by a suitably designed amplifier (which must be used in the feedback loop as we saw above). This circuit,

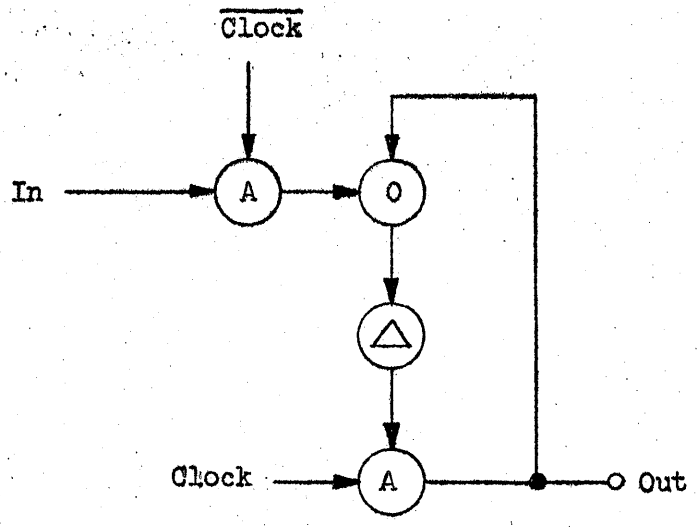


Figure 2-39
Latch Circuit

which can therefore latch onto a "1" or a "0", conserves during the whole clock pulse the information present at its beginning. A useful application is shown in Figure 2-40 in which an accumulator register feeds information into an adder (the second input coming from a fixed number register). The output of the adder goes into a latch circuit. It is easily seen that no gates are needed between the adder and the latches preceding the accumulator, since the clock time can be chosen such that the output of the latches sets the accumulator, but that this new setting does not "race" through the adder to produce the next set of inputs, the latches having sampled the information at precisely the instant at which the adder settles down for the first time. The delay Δ is chosen equal to the longest expected delay in the adder.

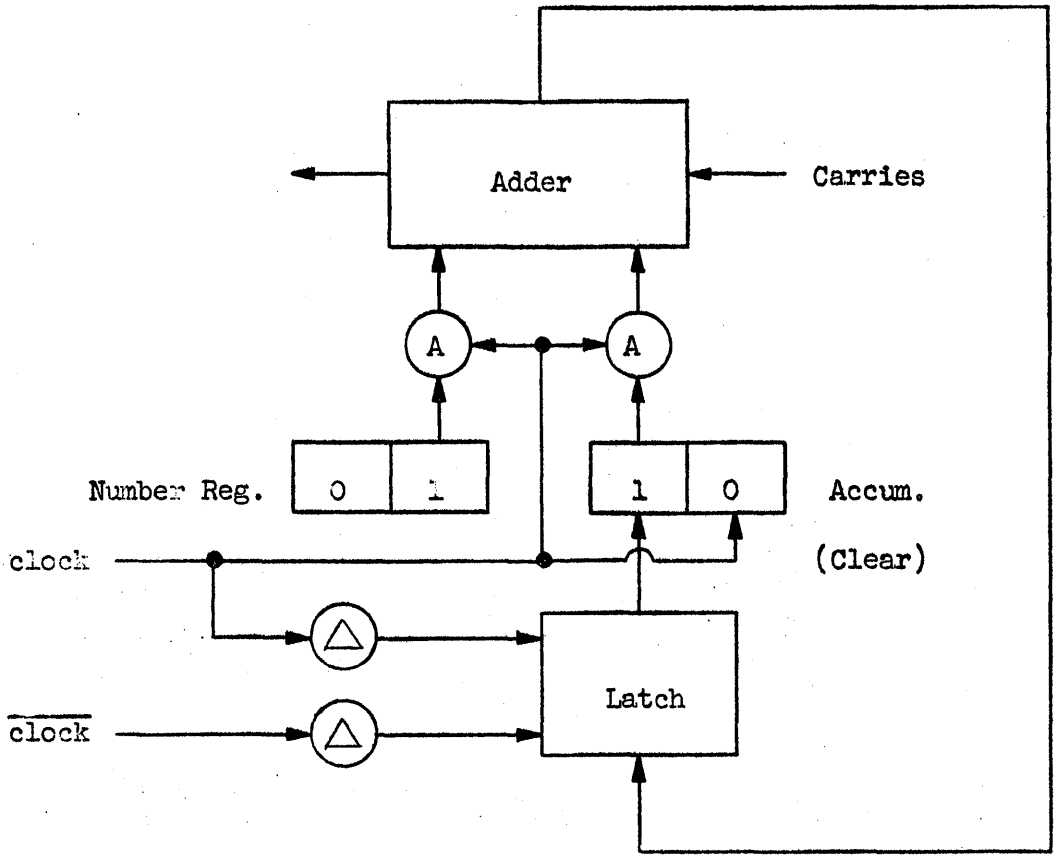


Figure 2-40
One Clock Period Adder Using a Latch

2.8 Synchronous Serial Operation

The full advantages of synchronous logic can be reaped by using a serial i.e. sequential processing of information pulses. This implies in particular that numbers are sent "least significant digit first". We shall discuss below some of the more frequently encountered circuits for serial operation.

Delay Lines (Recirculating Registers)

It is possible to use a transmission line of sufficient length to store sequences of pulses. Such a line can be thought of as a chain of delay elements: in order to store n pulses we need n times a delay equal to the period of the clock. The chain usually contains at its end a circuit for regenerative broadening. This has for effect not only to give to pulses a standard shape and length, but also to resynchronize them with the clock, i.e. to make sure that all pulses are still equally spaced after an indefinitely great number of passages through the line. It should be remarked that delay lines are often of the accustic type in order to circumvent size problems one would encounter with electric lines storing 1000 or more bits. The accustic delay line is simply a sound propagating rod connected between a loudspeaker and a microphone (called "transducers") at megacycle frequencies; bursts of sine waves are used rather than the modulating pulses themselves: This simplifies the design of the transducers.

The two main problems with recirculating registers are 1) to "load" the line by establishing in it a train of pulses conveying the information initially present in a set of flipflops 2) to "unload" the line by dumping into a set of flipflops the dynamic information "running off the end" of the line.

Figure 2-41 shows a possible loading mechanism. When both the load signal and a clock pulse occur, the information in the flipflops is made available to the line via the input OR in front of (or as one can see from Figure 2-38 actually part of) the regenerative broadening circuit which feeds the line (represented by a series of delay elements). Delays equal to one, two etc. times the clock period are inserted between the one-side output of the flipflops and a common collecting OR circuit. The latter goes into the input OR mentioned above via an AND which disconnects the flipflops in case no loading signal is present: in the absence of the load signal the upper AND closes the loop and makes sure that no information is lost. Note that more than one word can be stored and that a counter is required to time the load signal correctly so that a new word does not start in the middle of one already being recirculated.

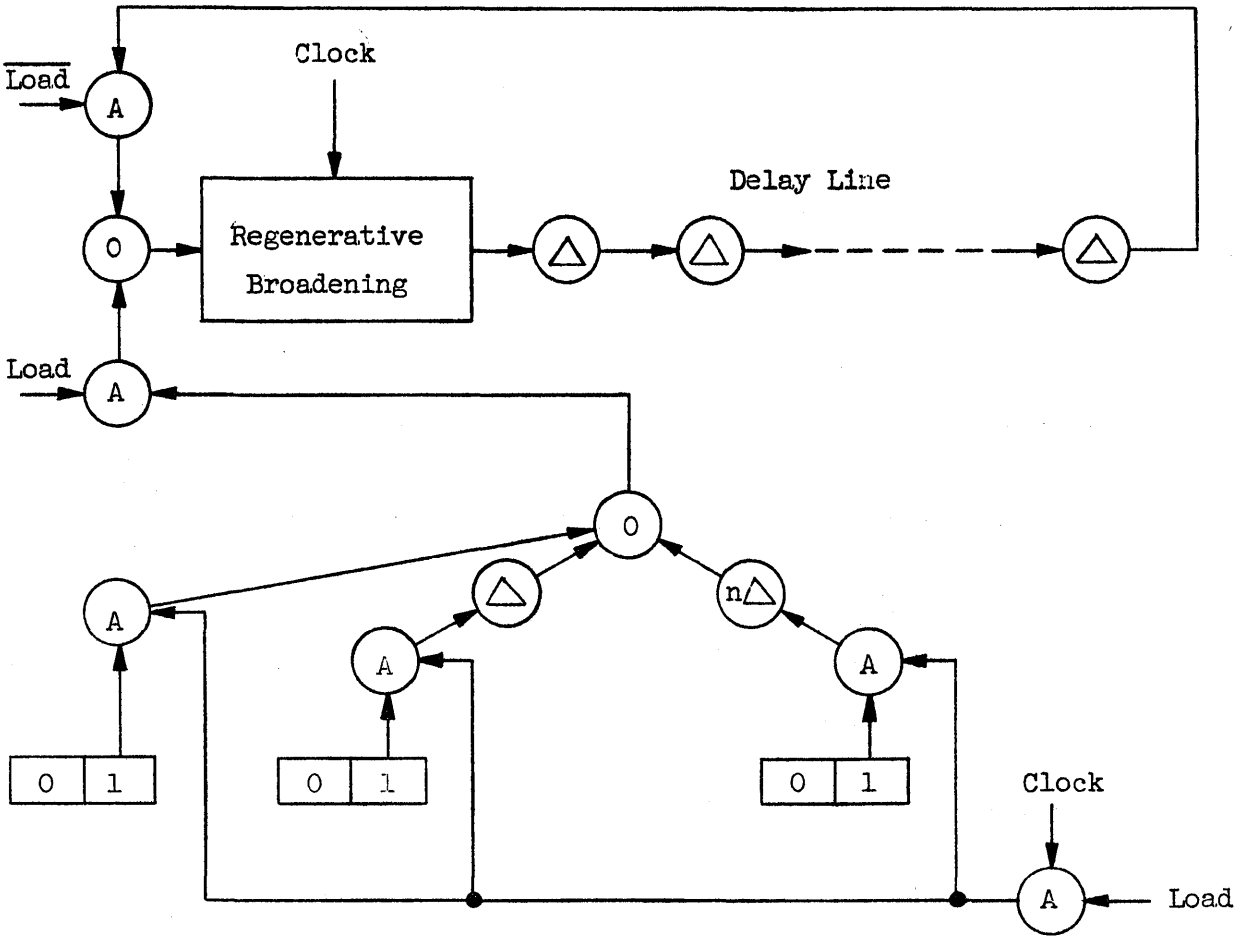


Figure 2-41
Loading of a Line from a Flipflop Register

The unloading of a recirculating register can be most easily accomplished by tapping the line at one-clock-period intervals and sending the signals present at these taps simultaneously (via AND gates) into a set of previously cleared flipflops; Figure 2-42 shows the principle. In case of acoustical lines (or if one does not want to tap the main storage line) the dynamic information is actually switched to a separate (lumped-constant LC) delay line with taps, called a "staticizer".

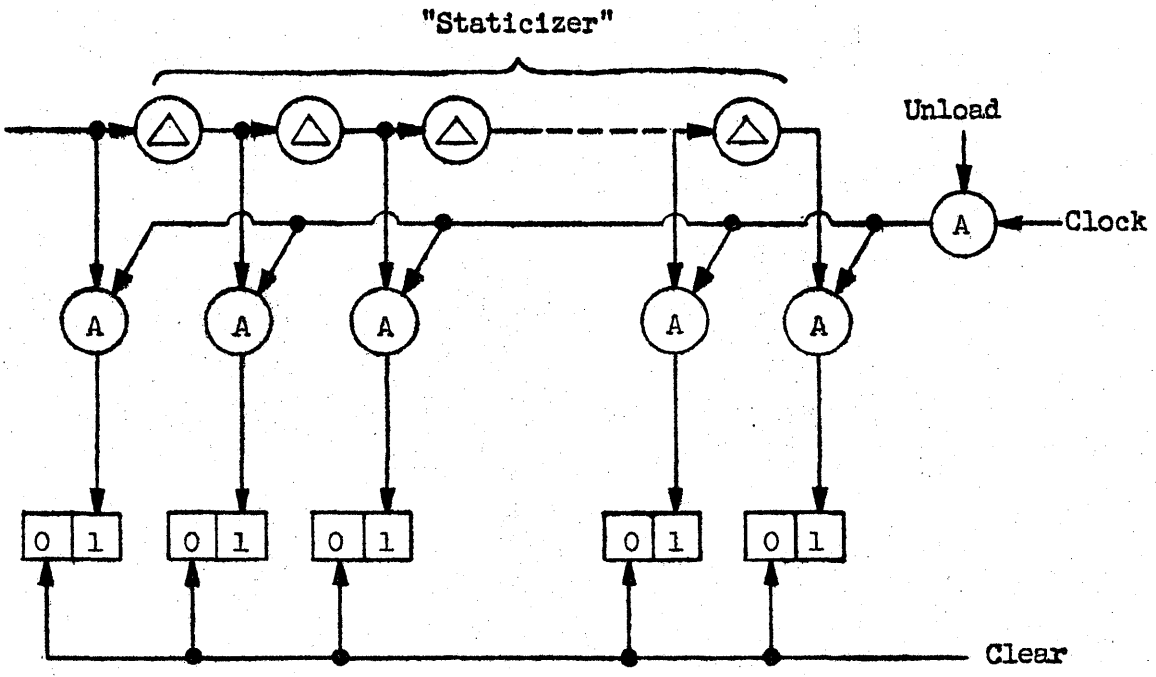


Figure 2-42
Unloading a Staticizer into a Flipflop Register

Dynamic Flipflop

It is easily seen that a delay line giving a delay equal to one clock period coupled to a regenerative broadening circuit is simply a dynamic flipflop: once a pulse is trapped in this loop it will reappear periodically. It is usual to add an AND in the loop fed by the inverse of a clear signal in order to be able to set such a flipflop back to the 0 state in which no pulse ever appears. Figure 2-43 shows the arrangement.

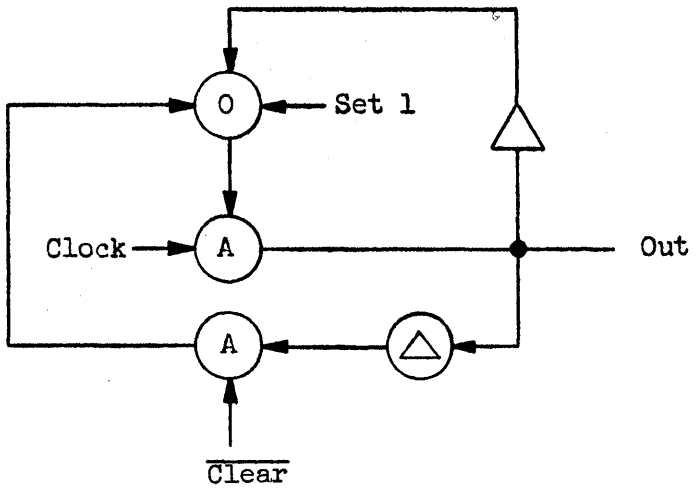


Figure 2-43
Dynamic Flipflop

Serial Adder

One of the main advantages of serial operation is that only one adder stage is necessary in order to produce a pulse train giving the sum of two pulse trains. Note that instead of speaking of the sum s_i in stage i we now talk about the i^{th} pulse s(i) counted from the beginning of the train, or more exactly the i^{th} clock-period, since no pulse occurs when the corresponding digit is zero. The same remark holds, of course, for the inputs x(i), y(i) and the carries. Adding in the carry from the preceding stage now is simply replaced by delaying the carry of the previous clock period. Figure 2-44 shows the extreme simplicity of a serial adder.

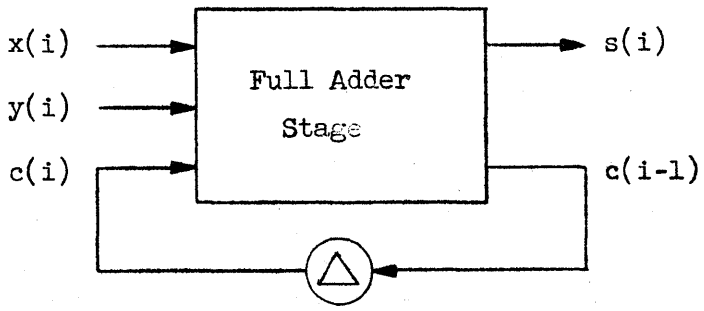


Figure 2-44
Serial Adder. All pulse trains are injected with the least significant digit first.

Counter

Figure 2-45 shows how counting can be performed by using two AND's in front of a flipflop and controlling the second input from the opposite side of the flipflop output. Visibly such an arrangement will steer each incoming pulse to that side of the flipflop input which will produce a change - this means that for each incoming pulse the flipflop changes state. If we sample one of its sides (after the transient dies down, a delayed clock signal is used to control the output AND) we shall evidently obtain a pulse only for each second incoming pulse i.e. we actually have one stage of a binary counter.

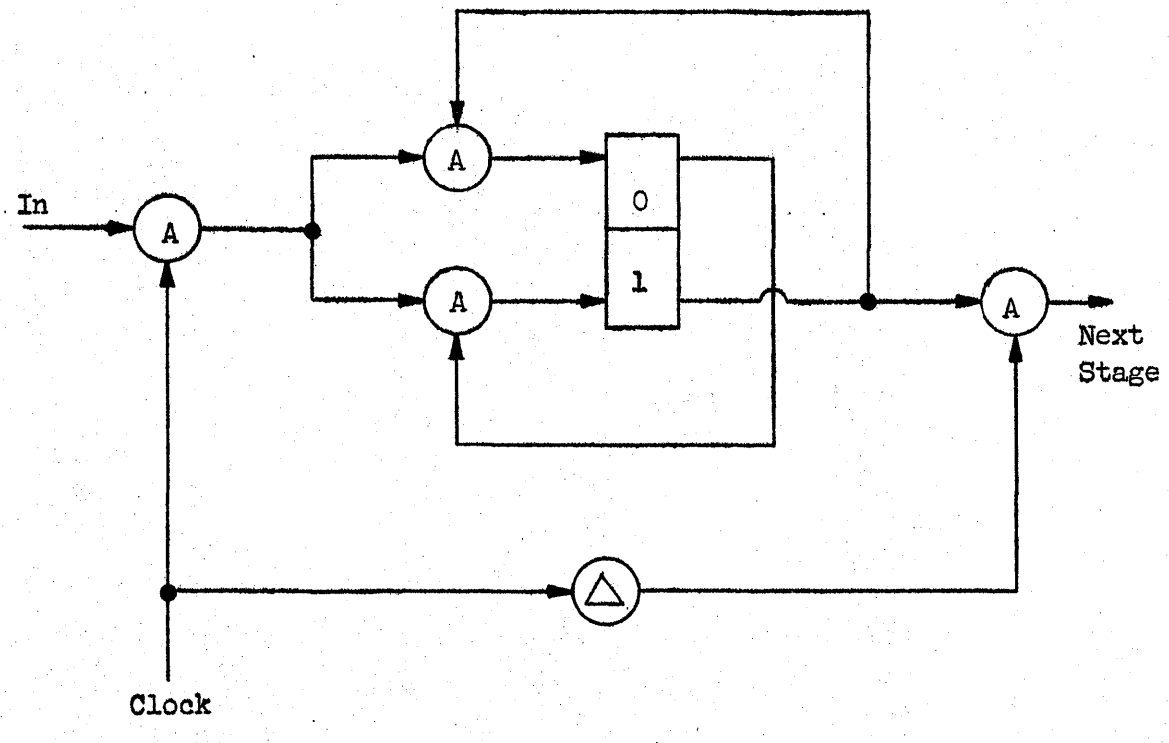


Figure 2-45
One Stage of a Synchronous Counter

Starting and Stopping a Sequence

One of the problems that occurs in serial machines is to switch the output of a clock onto a line in such a fashion that no "half-pulses" occur, i.e. making sure that the switching occurs between two clock pulses. Figure 2-46 shows how this can be done. The idea is to set a first flipflop FF1 by the start/stop signals and to transfer this information on the next interval between clock-pulses to FF2; the latter cannot be changed while the clock pulse comes along since the input AND's cannot transmit information while the clock pulse is on. Note that the setting time of the flipflop may have to be taken care of by introducing a delay between the clock and the output AND.

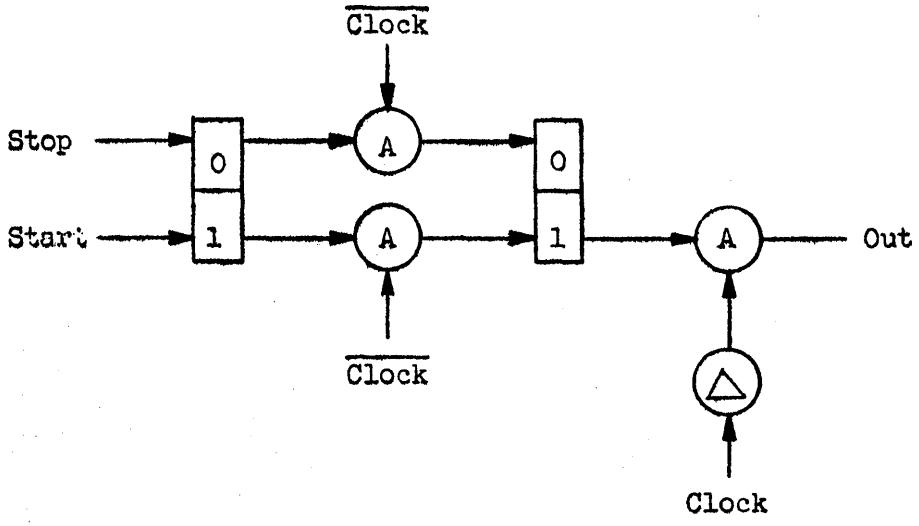


Figure 2-46
Stopping and Starting a Pulse Sequence

CHAPTER III

BOOLEAN ALGEBRA

3.1 The Postulates of Boolean Algebra

Although Boolean Algebra is applicable to more than two values, it is useful to think of the postulates below as summarizing the behavior of the logical circuits discussed in Chapter II. Symbolizing the output of an AND-circuit with inputs x_1 and x_2 by $x_1 \cdot x_2$ (i.e. writing $y = x_1 \cdot x_2$), the output of an OR-circuit with inputs x_1 and x_2 by $x_1 \vee x_2$ (i.e. writing $y = x_1 \vee x_2$) and finally by denoting the NOT-operation by a bar (i.e. writing for a NOT-circuit $y = \bar{x}$), we can define the three fundamental operations $\cdot$ $\vee$ - in two-valued Boolean Algebra by truth tables.

Truth Tables for Two-Valued Boolean Algebra

$y = x_1 \cdot x_2$

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

$y = x_1 \vee x_2$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

$y = \bar{x}$

x	y
0	1
1	0

We can now easily verify that $\cdot$ and $\vee$ satisfy the postulates of idempotence, commutativity, associativity and distributivity:

$$x \cdot x = x \qquad x \vee x = x \qquad \text{(idempotence)} \quad (3\text{-}1)$$

$$x_1 \cdot x_2 = x_2 \cdot x_1 \qquad x_1 \vee x_2 = x_2 \vee x_1 \qquad \text{(commutativity)} \quad (3\text{-}2)$$

$$x_1(x_2 \cdot x_3) = (x_1 \cdot x_2) \cdot x_3 \qquad x_1 \vee (x_2 \vee x_3) = (x_1 \vee x_2) \vee x_3 \qquad \text{(associativity)} \quad (3\text{-}3)$$

$$x_1 \cdot (x_2 \vee x_3) = (x_1 \cdot x_2) \vee (x_1 \cdot x_3) \qquad \text{(distributivity 1)} \quad (3\text{-}4)$$

$$x_1 \vee (x_2 \cdot x_3) = (x_1 \vee x_2) \cdot (x_1 \vee x_3) \qquad \text{(distributivity 2)} \quad (3\text{-}5)$$

We shall now introduce the "null-element" 0 and the "identity element" 1 respectively.* Then the following properties of intersection ($\cdot$) and union (v) hold:

$$0 \cdot x = 0 \qquad \text{(0-intersection)}$$
$$0 \text{ v } x = x \qquad \text{(0-union)} \qquad (3\text{-}6)$$

$$1 \cdot x = x \qquad \text{(1-intersection)}$$
$$1 \text{ v } x = 1 \qquad \text{(1-union)} \qquad (3\text{-}7)$$

Finally the NOT-operation, which we shall simply call complementation, satisfies the laws of complementarity, dualization (also called De Morgan's Theorem), and involution:

$$x \cdot \overline{x} = 0 \qquad x \text{ v } \overline{x} = 1 \qquad \text{(complementarity)} \qquad (3\text{-}8)$$

$$\overline{(x_1 \cdot x_2)} = \overline{x_1} \text{ v } \overline{x_2} \qquad \text{(dualization 1)} \qquad (3\text{-}9)$$

$$\overline{(x_1 \text{ v } x_2)} = \overline{x_1} \cdot \overline{x_2} \qquad \text{(dualization 2)} \qquad (3\text{-}10)$$

$$\overline{(\overline{x})} = x \qquad \text{(involution)} \qquad (3\text{-}11)$$

The notation in Boolean Algebra can be simplified to some extent by leaving out the parenthesis in expressions involving operations $\cdot$ or v only: the law of associativity permits this. Furthermore one can omit the symbol $\cdot$ altogther and write $x_1 x_2$ instead of $x_1 \cdot x_2$. Finally we can agree to interpret expressions involving $\cdot$, v and - in which parenthesis do not appear in such a way that the $\cdot$ operation is performed first and the - operation last. E.g. we would interpret $\overline{x_1 \text{ v } x_2 x_3}$ to mean $[\overline{x_1 \text{ v } (x_2 \cdot x_3)}] = \overline{x_1} \cdot (\overline{x_2 x_3}) = \overline{x_1} \cdot (\overline{x_2} \text{ v } \overline{x_3})$.

Inspection of the above postulates shows that there exists a certain symmetry in the postulates. More precisely: when we take a postulate and interchange $\cdot$ and v and 0 and 1 we obtain another postulate, called dual of the

---

* In the two valued algebra there are no other elements besides the null and the identity element.

first. For example $1 \vee x = 1$ has as its dual $0 \cdot x = 0$. Similarly any theorem we prove from the postulates will have a dual: this dual is proved starting with the dual postulates.

There is a startling analogy of the postulates of Boolean Algebra with those of arithmetic when we replace $\cdot$ by multiplication x and v by addition + : postulates (3-2)-(3-4) are valid in arithmetic. (3-1) however and (3-5) are obviously untrue, e.g. "multiplication is distributive over addition" while "addition is not distributive over multiplication". It is important to notice that there is no simple cancellation law in Boolean Algebra. The existence of cancellation laws is always a consequence of the existence of inverses with respect to multiplication in arithmetic: $x^{-1} = 1/x$. Therefore $xy = xz$ entails $y = z$ when we multiply through by x^{-1}.

If there were inverses x^{-1} with respect to the operations v and $\cdot$ of Boolean Algebra, we should have

$$x \vee x^{-1} = 0 \quad *$$

$$xx^{-1} = 1$$

This is clearly impossible, for the first equation would imply that $\bar{x} \vee x \vee x^{-1} = \bar{x} \vee 0 = \bar{x}$ i.e. $\bar{x} = 1 \vee x^{-1} = 1$ which is certainly not generally true. A similar argument holds for the second equation.

The nearest approach to a law of cancellation in Boolean Algebra is given by the

Theorem on Cancellation: If $x_1 z = x_1 y$ and $x_1 \vee z = x_1 \vee y$ then $z = y$ for an aribitrary x_1

Proof: Take $x_1 \vee z = x_1 \vee y$ and form $\bar{x}_1(x_1 \vee z) = \bar{x}_1(x_1 \vee y)$. By the laws of distributivity and complementarity we then have

$$\bar{x}_1 z = \bar{x}_1 y$$

---

* Note that x^{-1} for v would be different from x^{-1} for $\cdot$ just as $-x$ and $1/x$ are generally different.

therefore $(x_1 \vee \bar{x}_1)\, z = (x_1 \vee \bar{x}_1)\, y$ since $x_1 z = x_1 y$

i.e. $z = y$ by the law of complementarity.

Furthermore there is no such thing as a polynomial in Boolean Algebra because by the idempotence law all powers of x are equal to x.

It is useful to mention at this stage a particular case of the laws of distributivity and idempotence, called law of absorption:

$$x_1 \cdot (x_1 \vee x_2) = x_1$$
$$x_1 \vee x_1 x_2 = x_1 \qquad (3\text{-}12)$$

To verify the second equation we can write it in the form $x_1(1 \vee x_2)$, while the first one is reduced to the second by writing it in the form

$$x_1 x_1 \vee x_1 x_2 = x_1 \vee x_1 x_2.$$

To stress the analogy of Boolean Algebra with arithmetic, it is customary to call the formation of xy "multiplying y by x" and that of $x \vee y$ "adding y to x".

Inclusion and Consistency. Exclusive OR and Sheffer Stroke

There is a certain number of other symbols used in Boolean Algebra. The first one is the inclusion symbol $\leq$ which is defined as follows:

$x_1 \leq x_2$ means that

$$x_1 x_2 = x_1 \qquad (3\text{-}13)$$

$$x_1 \vee x_2 = x_2 \qquad (3\text{-}14)$$

That the two definitions are equivalent is assured by the law of consistency: we can prove that (3-14) follows from (3-13) and vice-versa:

Assume $x_1 x_2 = x_1$

Then $x_1 \vee x_2 = x_1 x_2 \vee x_2$ by substitution

$= x_2$ by absorption

Assume $x_1 \vee x_2 = x_2$

Then $x_1 x_2 = x_1(x_1 \vee x_2)$ by substitution

$= x_1$ by absorption.

From the definition of $\leq$ we can easily see that this operation satisfies the laws of reflexivity, anti-symmetry and transitivity and that of universal bounds:

$$x \leq x \qquad \text{(reflexitivity)} \qquad (3\text{-}15)$$

$$\text{If } x \leq y \text{ and } y \leq x, \text{ then } x = y \qquad \text{(anti-symmetry)} \qquad (3\text{-}16)$$

$$\text{If } x \leq y \text{ and } y \leq z, \text{ then } x \leq z \qquad \text{(transitivity)} \qquad (3\text{-}17)$$

$$0 \leq x \leq 1 \qquad \text{(universal bounds)} \qquad (3\text{-}18)$$

The first two equations are verified by applying the definition of $\leq$. The third follows from the fact that we have simultaneously

$$xy = x$$

$$y \vee z = z$$

$$\therefore xz = x(y \vee z) = xy \vee xz = x \vee xz = x \ .$$

The fourth equation finally simply restates (3-6) and (3-7).

Another operational symbol useful in Boolean Algebra is $\oplus$ called "exclusive or" * and defined by

---

* Often the symbol $\veebar$ is used.

$$x \oplus y = \bar{x}y \vee \bar{x}y \tag{3-19}$$

The great usefulness of this operation arises from the fact that it allows us to form binary sums. Another important point is that in equations involving only $\oplus$ the ordinary laws of cancellation hold. This is due to the fact that every element x has an inverse x^{-1} with respect to the $\oplus$ operation such that

$$x \oplus x^{-1} = 0 \qquad \text{(existence of an inverse)} \tag{3-20}$$

Before we prove this latter point, let us note that $\oplus$ satisfies a number of the properties discussed in the last section:

$$x_1 \oplus x_2 = x_2 \oplus x_1 \qquad \text{(commutativity)} \tag{3-21}$$

$$x_1 \oplus (x_2 \oplus x_3) = (x_1 \oplus x_2) \oplus x_3 \qquad \text{(associativity)} \tag{3-22}$$

$$x_1 (x_2 \oplus x_3) = x_1x_2 \oplus x_1x_3 \qquad \text{(distributivity)} \tag{3-23}$$

These properties can be established from the definition of $\oplus$: this definition also gives us immediately the equation

$$0 \oplus x = x \qquad \text{(existence of a zero)} * \tag{3-24}$$

To prove (3-20) it is sufficient to verify that we can set $\underline{x^{-1} = x}$:

$$x \oplus x = x\bar{x} \vee \bar{x}x = 0 \tag{3-25}$$

$$\therefore x \oplus y = x \oplus z \longrightarrow y = z \tag{3-26}$$

for we can "add" (with the operation $\oplus$) x to both sides, which leaves y = z.

It is useful to join to equation (3-24) and (3-25) the pair

$$x \oplus 1 = \bar{x} \qquad x \oplus \bar{x} = 1 \tag{3-27}$$

---

* Equations (3-20) - (3-24) together with the commutative law for $\cdot$ are the postulates of a "ring"; $\oplus$ is therefore often called "ring-sum".

and $$x \oplus y = \bar{x} \oplus \bar{y} \tag{3-28}$$

We shall finally introduce a last operation: The Sheffer Stroke /; by definition:

$$x_1/x_2 = (\overline{x_1 x_2}) \tag{3-29}$$

It can then be easily seen that

$$\bar{x} = x/x \tag{3-30}$$

$$x_1 x_2 = (x_1/x_2)/(x_1/x_2) \tag{3-31}$$

$$x_1 \vee x_2 = (x_1/x_1)/(x_2/x_2) \tag{3-32}$$

$$x_1 \oplus x_2 = [x_1/(x_2/x_2)]/[x_1/x_1)/x_2], \tag{3-33}$$

which means that all the operations defined so far can be deduced from the Sheffer Stroke.*

3.2 Canonical Expansions

The purpose of this section is to show that all Boolean functions of a given number of variables $x_1\ x_2\ \ldots\ x_n$ can be written in a certain standard form called "canonical expansion". Before we prove this let us extend the dualization laws to n variables.

DeMorgan's Theorem 1:

$$\overline{x_1 \vee x_2 \vee \ldots \vee x_n} = \bar{x}_1\ \bar{x}_2 \ldots \bar{x}_n \tag{3-34}$$

Proof: Call $x_2 \vee \ldots x_n$ for short x, then $\overline{x_1 \vee x} = \bar{x}_1\ \bar{x}$ by (3-10). This process of reduction finally leads to (3-34).

---

* This corresponds, of course, to the well known fact that all logical networks can be synthesized from AND-NOT circuits.

DeMorgan's Theorem 2:

$$\overline{x_1\, x_2 \cdots x_n} = \overline{x_1} \vee \overline{x_2} \cdots \vee \overline{x}_n \tag{3-35}$$

Proof: This theorem is the dual of the preceding one. The proof is the dual of the preceding proof.

We shall now introduce the notion of minterm and maxterm. Given the n variables $x_1\, x_2 \ldots x_n$ a minterm is the product of all n variables, uncomplemented, partially complemented or all complemented. Such a complemented or uncomplemented variable is called a "literal". There are clearly $2^n = N$ minterms which we shall call $m_0\, m_1 \ldots m_{N-1}$. We can order these terms by the following:

Convention: Let $k_1\, k_2 \ldots k_n$ be the binary expression for k. Then

$$m_k = (k_1\, x_1 \vee \overline{k_1}\, \overline{x_1})\, (k_2\, x_2 \vee \overline{k_2}\, \overline{x_2}) \ldots (k_n\, x_n \vee \overline{k_n}\, \overline{x_n}) \tag{3-36}$$

For example the minterms of two variables x_1 and x_2 are, in the order $m_0\, m_1\, m_2\, m_3$: $\overline{x}_1\, \overline{x}_2$, $\overline{x}_1\, x_2$, $x_1\, \overline{x}_2$, $x_1\, x_2$. Similarly we can define the maxterms of n variables: they are the sums of all n variables, uncomplemented, partially complemented or all complemented. There are clearly $2^n = N$ maxterms which we shall call $M_0\, M_1 \ldots M_{N-1}$. We can order these terms by the following

Convention: Let $k_1\, k_2 \ldots k_n$ be the binary expression for k. Then

$$M_k = (k_1\, x_1 \vee \overline{k_1}\, \overline{x_1}) \vee (k_2\, x_2 \vee \overline{k_2}\, \overline{x_2}) \vee \ldots \vee (k_n\, x_n \vee \overline{k_n}\, \overline{x_n}) \tag{3-37}$$

For example the maxterms of two variables x_1 and x_2 are, in the order $M_0\, M_1\, M_2\, M_3$: $\overline{x_1} \vee \overline{x_2}$, $\overline{x_1} \vee x_2$, $x_1 \vee \overline{x_2}$, $x_1 \vee x_2$.

These definitions being accepted, we can restate DeMorgan's Theorems by the

Theorem on the Relationship of Maxterms and Minterms:

$$\bar{m}_k = M_{\bar{k}} \tag{3-38}$$

$$\bar{M}_k = m_{\bar{k}} \tag{3-39}$$

where $$\bar{k} = 2^n - 1 - k \tag{3-40}$$

Proof: The binary expression of $\bar{k}$ is clearly the one's complement of that of $\bar{k}$. So $M_{\bar{k}}$ will have a complemented variable whenever m_k had an uncomplemented one and vice-versa. Furthermore the passage from a minterm to a maxterm and vice-versa replaced AND's by OR's and vice-versa. This is therefore precisely the process described in DeMorgan's Theorems.

Now let us look at all the possible sums of minterms. There is $1 = {}^{N}C_0$ sum not involving any minterms, (i.e. the "sum" 0 itself) $N = {}^{N}C_1$ involving one minterm, ${}^{N}C_2$ involving two minterms etc. The number of different sums (i.e. combinations) is therefore ${}^{N}C_0 + {}^{N}C_1 + \ldots + {}^{N}C_N = (1 + 1)^N = 2^N = 2^{2^n}$. We shall call these 2^N sums the elemental OR forms.

Theorem on Elemental OR Forms:

No two of the 2^{2^n} elemental OR forms are equal.

Proof: Let F and G be two different elemental OR forms. Then G (say) contains at least one term m_j not contained in F. Choose values of $x_1 \ldots x_n$ such that $m_j = 1$, then all $m_i \neq m_j$ will be zero. For these values therefore $F \neq G$.

Another way of stating this theorem is to write

$$m_i \, m_j = 0 \tag{3-41}$$

In the same way we can discuss the possible products of maxterms: the number of different products is again $2^N = 2^{2^n}$. We shall call these 2^N products the elemental AND forms.

Theorem on Elemental AND Forms:

No two of the 2^{2^n} elemental AND forms are equal.

Proof: This theorem is the dual of the preceding one; the proof is dual.

Another way of stating this theorem is to write

$$M_i \vee M_j = 1 \tag{3-42}$$

We can now prove important theorems on the sum of all minterms and the product of all maxterms:

Theorem on the Sum of All Minterms:

$$m_0 \vee m_1 \vee \ldots \vee m_{N-1} = 1 \tag{3-43}$$

Theorem on the Product of All Maxterms:

$$M_0 M_1 \ldots M_{N-1} = 0 \tag{3-44}$$

Proof: It will be sufficient to prove (3-43), since (3-44) is the dual. Consider therefore $m_0 \vee m_1 \vee \ldots \vee m_{N-1} = m$ (say). There will be 2^{n-1} terms containing x_1 and 2^{n-1} terms containing $\overline{x}_1$ i.e.

$$m = (x_1 \cdot \ldots) \vee (x_1 \cdot \ldots) \ldots \vee (\overline{x_1} \cdot \ldots) \vee (\overline{x_1} \cdot \ldots)$$

$$= x_1 X_1 \vee \overline{x_1} X_2 \qquad \text{collecting terms.}$$

But by symmetry $X_1 = X_2$, therefore

$$m = x_1 X_1 \vee \overline{x_1} X_1 = X_1 (x_1 \vee \overline{x_1}) = X_1$$

where X_1 does not contain x_1 or $\overline{x}_1$. Pursuing this reduction process we shall finally come to $m = X_{n-1}$ where X_{n-1} does not contain $x_1 \ldots x_{n-1}$ or their complements. This means that $X_{n-1} = x_n \vee \overline{x_n} = 1$. This proves the theorem.

Before we can discuss the central theorem of this section we need two more Lemmas.

Exponential Composition Theorem for Minterms:

Let f be an arbitrary function and ν an arbitrary number of variables. Then the sum of the product of f with all the minterms formed from the ν variables is f:

$$fm_0 \vee fm_1 \vee \ldots \vee fm_{2^\nu - 1} = f \tag{3-45}$$

Exponential Composition Theorem for Maxterms:

Let f be an arbitrary function and ν an arbitrary number of variables. Then the product of the sums of f with all the maxterms formed from the ν variables is f:

$$(f \vee M_0)(f \vee M_1) \ldots (f \vee M_{2^\nu - 1}) = f \tag{3-46}$$

Proof: It will be sufficient to prove (3-45), since (3-46) is the dual. (3-45) is evident when we collect terms:

$$fm_0 \vee fm_1 \vee \ldots \vee fm_{2^\nu - 1} = f(m_0 \vee m_1 \vee \ldots \vee m_{2^\nu - 1}) = f ,$$

since (3-43) can be applied.

We now state the two theorems about canonical expansions.

Theorem on Canonical Expansions Using Minterms:

Every Boolean function f involving the symbols $\cdot$, $\vee$ and $-$ can be represented as one and only one product of maxterms, i.e. as one and only one elemental AND form.

Proof: Again it will be sufficient to prove the first theorem of this dual pair. To make things easier we shall consider a particular example: it is easy to see how the process is applied in general. Let

$$f = [(\overline{x_1 \bar{x}_2}) \vee \bar{x}_3] \, (\overline{\bar{x}_1 \vee x_3}).$$

The complementing bar can always be moved inside the parenthesis by applying DeMorgan's Theorem. This gives

$$f = (\bar{x}_1 \vee x_2 \vee \bar{x}_3)\,(x_1 \cdot \bar{x}_3) \qquad (n = 3)$$

We can now "multiply out", i.e. apply the distributive laws. There will result an expression which is a sum of products. In our example

$$f = \bar{x}_1 x_1 \bar{x}_3 \vee x_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_3 \bar{x}_3 \,.$$

We can now apply the idempotence law and the law of complementarity:

$$x_i x_i = x_i \,, \quad x_i \bar{x}_i = 0$$

This gives in our example

$$f = x_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_3$$

Now we can use the exponential composition theorem to "inflate" every term which does not contain all n variables into 2^{ν} minterms, where ν is the number of variables not contained in the term: we multiply this term by 1 in the form of the sum of all possible minterms of the ν variables. In our example x_2 is missing in $x_1 \bar{x}_3$. We multiply by $x_2 \vee \bar{x}_2$ giving

$$x_1 \bar{x}_3 = x_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 \bar{x}_3$$

i.e.

$$f = x_1 x_2 \bar{x}_3 \vee x_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 \bar{x}_3$$

In this way f is expressed as a sum of minterms. We finally replace the sum of all identical minterms by one minterm. In our example

$$f = x_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 \bar{x}_3 = m_6 + m_4$$

It is now evident that this reduction of f to a sum of minterms in unique: if there were two different canonical expansions, two different elemental OR forms would be equal. This is impossible by a theorem proved before.

It is very useful to know that the reduction process of f to an elemental OR form is sufficient to find the elemental AND form and vice-versa. The transformation from one to the other is given by the following theorems.

Theorem on the OR-to-AND Transformation:

Suppose that f has been expressed as a sum of minterms $f_m = \Sigma m_i$ (Σ means of course applying the operation v!) and that we wish to express f as a product of maxterms $f_M = \Pi M_j$ (Π means of course applying the operation $\cdot$!). Let $\Sigma^* m_j$ be the sum of minterms not in f_m. Then

$$f_m = \Pi \bar{m}_j = \Pi M_{\bar{j}} \tag{3-47}$$

In words: in the sum of minterms not in f_m, interchange $\cdot$ and v and reverse the complementation.

Theorem on the AND-to-OR Transformation:

Suppose that f has been expressed as a product of maxterms $f_M = \Pi M_i$ and that we wish to express f as a sum of minterms $f_m = \Sigma m_j$. Let $\Pi^* M_j$ be the product of maxterms not in f_M. Then

$$f_m = \Sigma \bar{M}_j = \Sigma m_{\bar{j}}$$

In words: in the product of maxterms not in f_M, interchange v and $\cdot$ and reverse the complementation.

Proof: As usual it is sufficient to prove the first theorem. Now clearly $\Sigma m_i \vee \Sigma^* m_j = 1$ by (3-43) i.e.

$$f_m \vee \Sigma^* m_j = 1 = f_m \vee \bar{f}_m$$

Furthermore

$$f_m \Sigma^* m_j = 0 = f_m \overline{f_m}$$

because of (3-41). By the theorem on cancellation of Section 3.1 this means that

$$\overline{f_m} = \Sigma^* m_j$$

$$f_m = \overline{\overline{f_m}} = (\overline{\Sigma^* m_j}) = \Pi \overline{m}_j = \Pi M_{\overline{j}}$$

by DeMorgan's theorem and (3-38). Since $f = f_m$ (f_m being the canonical expansion!) this completes the proof: $f = \Pi M_{\overline{j}}$.

It is appropriate to make an important remark. Up to now we have transformed given Boolean functions (of the "$\cdot$ v - type") into canonical expansions. The question comes up: can one (in a way analogous to finding a polynomial function passing through given points) determine a Boolean function assuming given values 0 or 1 in given "points". To answer the question, note that a "point" corresponds to a given combination of 0 and 1 in a binary number of n digits, where n is the number of variables we allow ourselves. With n variables we have 2^n minterms: If the function is to be = 1 in "point" $k = k_1 k_2 \ldots k_n$, this can only be achieved by including m_k in the expansion f_m of f. If the function is to be = 0 in "point" $k = k_1 k_2 \ldots k_n$, this can only be achieved by omitting m_k in the expansion f_m of f. This leads to a new theorem:

Synthesizing Theorem: Suppose that the 2^n combinations of n variables each correspond to a definite value 0 or 1 of a Boolean function $f(x_1 x_2 \ldots x_n)$; then

$$f = \Sigma m_k, \quad k = k_1 k_2 \ldots k_n$$

where m_k are the minterms corresponding to the combinations $k_1 k_2 \ldots k_n$ which give f = 1.

We shall see later what happens when some of the 2^n combinations do not correspond to any defined value of f.

It turns out that the synthesizing theorem is so easy to apply, that it is often advantageous to calculate the values of $f(x_1 \ldots x_n)$ for all combinations of the variables and then form the sum of the minterms corresponding to the "ones".

Finally it is clear that the synthesizing theorem has a dual. It is left to the reader to discuss the latter point.

3.3 Simplification of Boolean Expressions. Harvard Chart.

First of all we must define what we mean by "simplification": It means reducing a given expression (for instance a canonical expansion) to a form in which there is a minimum of variable occurrence. In this section we shall show that any given f can be reduced to a "minimum v polynomial" i.e. a sum of terms, each being the product of complemented or uncomplemented variables, without being necessarily a minterm. The expression "minimum" here means: each term having as few variables as possible and the polynomial having as few v signs as possible.

It is evident that by dual considerations we could discuss "minimum · polynomials". Again it will be left to the reader to generalize the processes.

The reduction of a minimum v polynomial to simpler expression can often be achieved by "collecting terms" i.e. "undistribution": if f has been reduced to $x\bar{y} \vee x\bar{z}$ we can write $f = x(\bar{y} \vee \bar{z})$; visibly this latter expression is not a v polynomial. This reduction of a minimum v polynomial to an expression having fewer variable occurrences is by no means straightforward: skill and flair (meaning: expanding terms or adding terms which do not change the function, so called "non-essential terms") are often necessary. This can be seen in the following:

Example: $f = tuy \vee tuwz \vee twxy \vee wxz$

This can be written

$$f = tu(y \vee wz) \vee wx\,(ty \vee z)$$

But using the idempotence law we can rewrite the first expression

$$f = ttuy \vee tuwz \vee twxy \vee wwxz$$

$$= tu(ty \vee wz) \vee wx(ty \vee wz)$$

$$= (tu \vee wx)\,(ty \vee wz)$$

This second expression is certainly simpler. Therefore an apparent initial complication leads to a simpler end result.

The reduction of f to a minimum - v - polynomial is, however, a straightforward process and we shall discuss one method of reduction: the Harvard Chart. It is often faster to use direct simplification, as described later in this section, but the Harvard Chart is an easy way to accomplish the first step in an automatic fashion.

To simplify, let us take the reduction of functions of three variables $x_1\ x_2\ x_3$. The chart then contains:

1. $2^3 = 8$ rows, each one corresponding to a possible minterm of f.
2. $2^3 = 8$ columns, the columns corresponding to a combination of variables one, two and three at a time and to the values of f at the 8 "points" 000, 001 ...., 111.

Figure 3-1 gives the aspect of the three variables chart for the example

$$f = \bar{x}_1\bar{x}_2x_3 \vee \bar{x}_1x_2x_3 \vee x_1x_2\bar{x}_3 \vee x_1x_2x_3$$

Remark: If f is not given as a canonical expansion, we can calculate f(000), f(001) etc. and use the synthesizing theorem of the last section.

The seventh column of the chart contains the eight possible minterms in order, the eighth column indicates by 0 and 1 which minterms occur in the canonical expansion and which do not. Columns 1-6 are filled

out as follows. In 1 we put x_1 or $\bar{x}_1$ depending on whether the minterm in the same row occurs in the expansion of x_1 or that of $\bar{x}_1$. Similarly for columns 2 and 3. In column 4 we put $\bar{x}_1\bar{x}_2$, $\bar{x}_1\bar{\bar{x}}_2$, $x_1\bar{x}_2$, or x_1x_2 depending again on whether the minterms in a given row is included in the expansion or not. The same argument holds for columns 5 and 6. This process clearly puts to the left of every minterm all the products of two variables or single variables which can give rise to this minterm: if f does not contain this minterm, the reduced v polynomial will certainly not contain any of the terms in the same row.

	1	2	3	4	5	6	7	8
	x_1	x_2	x_3	x_1x_2	x_1x_3	x_2x_3	$x_1x_2x_3$	f
1	~~$\bar{x}_1$~~	~~$\bar{x}_2$~~	~~$\bar{x}_3$~~	~~$\bar{x}_1\bar{x}_2$~~	~~$\bar{x}_1\bar{x}_3$~~	~~$\bar{x}_2\bar{x}_3$~~	~~$\bar{x}_1\bar{x}_2\bar{x}_3$~~	~~0~~
2	$\bar{x}_1$	$\bar{x}_2$	x_3	$\bar{x}_1\bar{\bar{x}}_2$	$\bar{x}_1x_3$	$\bar{x}_2x_3$	$\bar{x}_1\bar{x}_2x_3$	1
3	~~$\bar{x}_1$~~	~~x_2~~	~~$\bar{x}_3$~~	~~$\bar{x}_1x_2$~~	~~$\bar{x}_1\bar{x}_3$~~	~~$x_2\bar{x}_3$~~	~~$\bar{x}_1x_2\bar{x}_3$~~	~~0~~
4	$\bar{x}_1$	x_2	x_3	$\bar{x}_1x_2$	$\bar{x}_1x_3$	x_2x_3	$\bar{x}_1x_2x_3$	1
5	~~x_1~~	~~$\bar{x}_2$~~	~~$\bar{x}_3$~~	~~$x_1\bar{x}_2$~~	~~$x_1\bar{x}_3$~~	~~$\bar{x}_2\bar{x}_3$~~	~~$x_1\bar{x}_2\bar{x}_3$~~	~~0~~
6	~~x_1~~	~~$\bar{x}_2$~~	~~x_3~~	~~$x_1\bar{x}_2$~~	~~x_1x_3~~	~~$\bar{x}_2x_3$~~	~~$x_1\bar{x}_2x_3$~~	~~0~~
7	x_1	x_2	$\bar{x}_3$	x_1x_2	$x_1\bar{x}_3$	$x_2\bar{x}_3$	$x_1x_2\bar{x}_3$	1
8	x_1	x_2	x_3	x_1x_2	x_1x_3	x_2x_3	$x_1x_2x_3$	1

Figure 3-1

Harvard Chart for $f = \bar{x}_1\bar{x}_2x_3 \vee \bar{x}_1x_2x_3 \vee x_1x_2\bar{x}_3 \vee x_1x_2x_3$

The Harvard Chart is now used by following the rules listed below:

1. Strike out the rows corresponding to the zeros in the f column. This eliminates all minterms (and the "constituants") which could give rise to them. In our example rows 1, 3, 5 and 6 are eliminated.

2. Strike out in each column all entries crossed out in Step 1 above. This means: if a given constituant is not contained in f (being in a cancelled row) it is no use trying to introduce it elsewhere. In our example we thus cross out all entries in columns 1, 2 and 3. In column 4 however two entries are left and the same holds for columns 5 and 6. In column 7 there are 4 uneliminated entries: all these entries are marked with circles.

3. We must now find a minimum set of entries such that there is one in each row for which a 1 is marked in column f; this means: we search for a minimum set which (in a canonical expansion) will give all the minterms in f. In our example inspection shows that terms $\bar{x}_1 x_3$ and $x_1 x_2$ (in shaded circles) form this minimum set. We therefore have as the minimum v polynomial

$$f = \bar{x}_1 x_3 \vee x_1 x_2$$

Remark: It is sometimes considered advantageous to replace the entries in Figure 3-1 by numbers obtained as follows: a complemented variable corresponds to 0, an uncomplemented variable to 1. To each constituant then corresponds a different binary number. This number is written in the decimal system. The chart then takes the aspect indicated in Figure 3-2. When the minimum set of entries has been found, one can easily go back to the constituants.

We now have to discuss the second (and much vaguer) part of the reduction process i.e. the reduction of a minimum v polynomial to an expression having the least variable occurrences. This is done by "flair and skill", "collecting terms" and by using the following (easily verified) equations:

x_1	x_2	x_3	x_1x_2	x_1x_3	x_2x_3	$x_1x_2x_3$	f
0	0	0	0	0	0	0	0
∅	∅	X	∅	2	X	1	1
0	1	0	1	0	2	2	0
∅	X	X	X	1	3	3	1
1	0	0	2	2	0	4	0
1	0	1	2	3	1	5	0
X	X	∅	3	2	2	6	1
X	X	X	3	3	3	7	1

Figure 3-2
Numerical Harvard Chart Corresponding to Figure 3-1

$$x_1\bar{x}_2 \vee x_1x_2 = x_1 \tag{3-48}$$

$$x_1 \vee x_1x_2 = x_1 \tag{3-49}$$

$$x_1 \vee \bar{x}_1x_2 = x_1 \vee x_2 , \tag{3-50}$$

as well as the equation

$$x_1x_2 \vee \bar{x}_2x_3 \vee x_1x_3 = x_1x_2 \vee \bar{x}_2x_3 \tag{3-51}$$

or one of its equivalent forms. This latter type of reductible first member is characterized by two variables multiplying an uncomplemented and a complemented variable and occurring again in product form: the last product then is superfluous.

It is sometimes useful to apply a test for a superfluous term by applying the following Rule for Superfluous Terms: if a term is suspected to be superfluous, take values of the variables which make this term equal to one. Insert these values in all other terms: if the remainder of the terms also gives a one, the term tested is superfluous.

Take for example equation (3-51) and set $x_1 = 1$, $x_3 = 1$ then $x_1x_2 \vee \bar{x}_2x_3 = 1$, $\therefore$ x_1x_3 is superfluous.

Finally it should be remarked that eliminating terms may very well lead to a "trap" situtation similar to that described at the beginning of the section: an expression may contain no superfluous terms and appear no longer reductible. Adding a superfluous term may permit further simplification.

3.4 Quine's Method

It is easily seen that a Harvard Chart for 4 variables has 16 rows (because of 2^4 minterms), excluding the one containing the headings. It also has 16 columns for the constituants, excluding the two last ones for the minterms themselves and the value of f: such a chart is obviously cumbersome. For 5 variables the size gets entirely out of hand. McCluskey and Quine have developed a method in which an arbitrary Boolean function given in canonical form is first reduced to prime implicants i.e. essentially a possible set of constituants. The actual choice of the set to be used is then made on a very much simplified Harvard Chart called "Prime Implicant Chart". To simplify matters still further, a numerical shorthand is used in which all terms are denoted by their corresponding binary number and the notion of index is introduced: the index is the number of 1's in the binary number. The search for prime implicants then follows this pattern:

1. Write down all minterms in f (in binary shorthand) dividing them into groups. The first group has the lowest index (not necessarily 0!), the second group the next highest index etc. This is done by dividing the vertical list by horizontal lines where the index changes.

2. Compare groups of indices differing by 1 (i.e. neighboring groups) to find terms differing in one digit only. Write down in a second column the "reduced terms" obtained by replacing the digit which differs by a dash. Check off the terms used in this process. (E.g. in a 3-variable problem involving $\bar{x}_1\bar{x}_2x_3$ and $\bar{x}_1\bar{x}_2\bar{x}_3$ denoted by 001 and 000 we write as the reduced term 00-)

3. Divide the second column again into groups by horizontal lines, the first group containing those reduced terms stemming from the first two groups of the original list, the second group those stemming from the comparison of group 2 and group 3 of the original list etc. These new groups visibly have increasing numbers of 1's.

4. The second column again has its adjacent groups compared: now doubly reduced terms appear (written with two dashes) when two reduced terms only differ in one digit. The doubly reduced terms are put down in a third column and divided into groups according to which combination of groups in column 2 they stem from. Again the terms used in column 2 are checked off.

5. The process stops when no new columns can be formed. All terms which have not been checked off are the <u>prime implicants</u>.

Visibly we have done in a somewhat automatic manner what amounts to the combination of terms of the form $Xx \vee X\bar{x}$ into X until no further reduction is possible. This is similar to the search for constituants in the Harvard Chart. It can actually be shown that the two methods give identical results.

The next step is to draw up a prime implicant chart having as many columns as there are minterms in f (i.e. by no means corresponding to <u>all</u> the minterms) and as many rows as there are prime implicants. The problem is now to choose that set of prime implicants which is minimum (in number of prime implicants) and at the same time gives rise to all minterms used. Often this chart is drawn in the form of numbers giving

the minterms connected to vertical lines and the prime implicants connected to horizontal lines; a cross marks those minterms which occur in the expansion of a given prime implicant.

It can happen that there are columns with one cross only: the corresponding prime implicants are called basic prime implicants and their row is called a primary basis row. If some minterms are left over, there are often secondary prime implicants (and secondary basis rows) which account for all minterms included in two other rows. The simplified expression then contains the sum of the basic prime implicants plus the sum of the secondary prime implicants plus a (sometimes arbitrary) choice of remaining prime implicants to account for the remaining minterms.

The example below, taken from S. H. Caldwell, "Switching Circuits and Logical Design" shows how a 5-variable situation is handled. It may be interesting to note that the search for prime implicants can be programmed for a digital computer with relative facility.

Example: Consider a canonical expression for a function f of 5-variables given by

$$f = m_0 \vee m_1 \vee m_3 \vee m_8 \vee m_9 \vee m_{13} \vee m_{14} \vee m_{15} \vee m_{16} \vee m_{17} \vee m_{19} \vee m_{24} \vee m_{25} \vee m_{27} \vee m_{31}$$

or in a more convenient notation

$$f = \Sigma\, 0, 1, 3, 8, 9, 13, 14, 15, 16, 17, 19, 24, 25, 27, 31$$

The search for the prime implicants then takes 4 columns. To simplify we have indicated at the left the decimal subscript of m and in the reduced terms the decimal subscripts of the terms used.

0	00000 v
1	00001 v
8	01000 v
16	10000 v
3	00011 v
9	01001 v
17	10001 v
24	11000 v
13	01101 v
14	01110 v
19	10011 v
25	11001 v
15	01111 v
27	11011 v
31	11111 v

0,1	0000- v
0,8	0-000 v
0,16	-0000 v
1,3	000-1 v
1,9	0-001 v
1,17	-0001 v
8,9	0100- v
8,24	-1000 v
16,17	1000- v
16,24	1-000 v
3,19	-0011 v
9,13	01-01 ⟵ A
9,25	-1001 v
17,19	100-1 v
17,25	1-001 v
24,25	1100- v
13,15	011-1 ⟵ B
19,15	0111- ⟵ C
19,27	1-011 v
25,27	110-1 v
15,31	-1111 ⟵ D
27,31	11-11 ⟵ E

0,1,8,9	0-00- v
0,1,16,17	-000- v
0,8,16,24	--000 v
1,3,17,19	-00-1 ⟵ F
1,9,17,25	--001 v
8,9,24,25	-100- v
16,17,24,25	1-00- v
17,19,25,27	1-0-1 ⟵ G
0,1,8,9,16,17,24,25	--00- ⟵ H

(The last expression comes about as the result of several combinations!)

Calling the prime implicants in order A,B,C,D,E,F,G,H we have the chart shown in Figure 3-3.

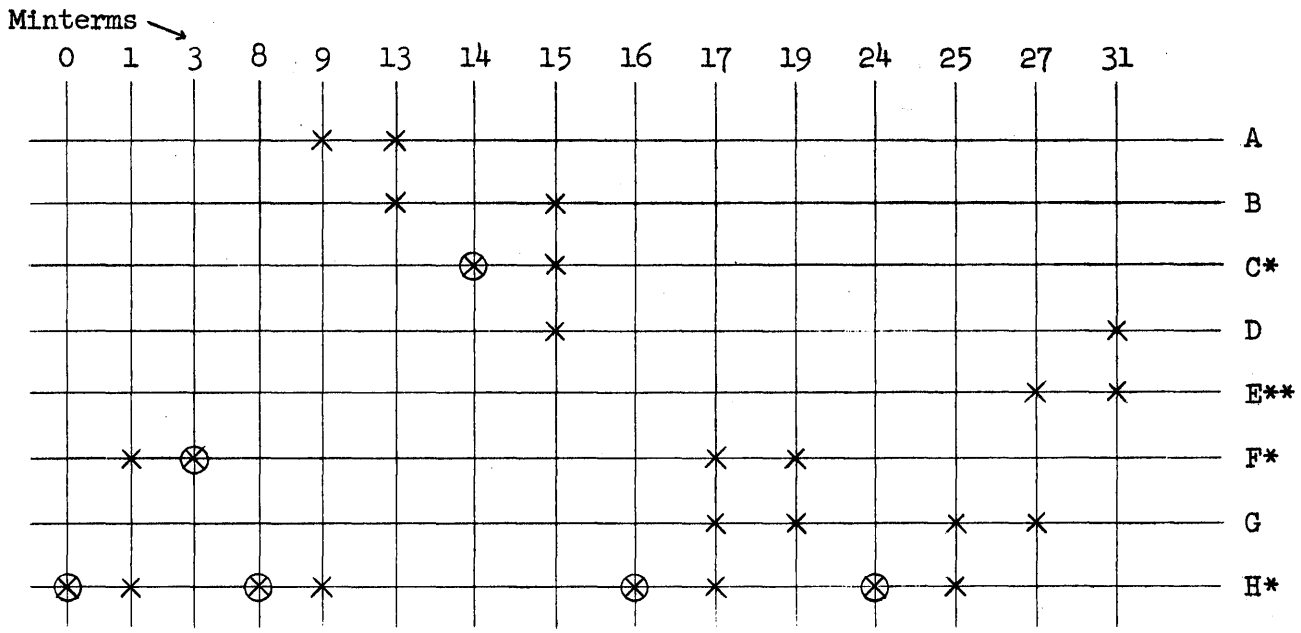


Figure 3-3

Prime Implicant Chart for a Special 5-Variable Function

* ⟶ Primary Basis Rows, ** ⟶ Secondary Basis Rows.

Since columns 0,3 and 14 have one cross only, C,F and H are primary basis rows. When we strike out the columns that are covered by C,F and H we find that only columns 13,27 and 31 remain. Visibly row E has crosses in both 27 and 31 which means that the remaining minterm 27 of G and the remining minterm 31 of D are taken care of by choosing E as a secondary basis row. Minterm 13 can be taken care of by adding either A or B: which one is chosen is arbitrary (if the row ⟶A contained more crosses, we would choose it, because by the construction it would contain fewer literals!). We thus arrive at the following form for f:

$$f = C \vee F \vee H \vee E \vee A$$

Going back to the meaning of these terms we have

$$C \longrightarrow 0111- \longrightarrow \bar{x}_1 x_2 x_3 x_4 = \bar{x}_1 x_2 x_3 x_4$$

$$F \longrightarrow -00-1 \longrightarrow \bar{x}_2 \bar{x}_3 \; x_5 = \bar{x}_2 \bar{x}_3 \; x_5$$

$$H \longrightarrow --00- \longrightarrow \bar{x}_3 \bar{x}_4 = \bar{x}_3 \bar{x}_4$$

$$E \longrightarrow 11-11 \longrightarrow x_1 x_2 \; x_4 x_5 = x_1 x_2 \; x_4 x_5$$

$$A \longrightarrow 01-01 \longrightarrow \bar{x}_1 x_2 \; \bar{x}_4 x_5 = \bar{x}_1 x_2 \; \bar{x}_4 x_5$$

i.e.

$$f = \bar{x}_1 x_2 x_3 x_4 \vee x_2 \bar{x}_3 x_5 \vee \bar{x}_3 \bar{x}_4 \vee x_1 x_2 x_4 x_5 \vee \bar{x}_1 x_2 \bar{x}_4 \bar{x}_5$$

Remark: Note that since prime implicants are formed by combining 2,4,8 or 16 minterms etc., the number of crosses per prime implicant is a power of two!

3.5 Other Interpretations of Boolean Algebra

The Algebra of Logic

When we make statements we use propositions: i.e. "Illinois has no mountains" (P), or "There are 48 hours in a day" (Q), or "The barber shaves all men who do not shave themselves" (R). These propositions are either true (P), or false (Q) or undecidable (R); the last proposition is of this type, for it is not evident whether the barber shaves himself or not. Excluding undecidable propositions, there is attached to each proposition a truth value $p \longrightarrow P$, $q \longrightarrow Q$ etc. such that a true proposition corresponds to 1 and a false proposition to 0. Above, visibly, $p = 1$ and $q = 0$.

Often we form logical connectives by using modifications or combinations of propositions. In particular we can deny a statement i.e. form its complement: if P is the statement "Illinois has no mountains", the statement $\bar{P}$ is "Illinois has mountains". If S is the proposition "Illinois has no natural lakes" we can form the "product" statement $P \cdot S$ "Illinois has no mountains and Illinois has no natural lakes". If T is the proposition "I am all wrong", then $P \vee T$ is the "union" statement "Illinois has no mountains or I am all wrong".

The use of $\cdot$ and v shows that there is a relationship between Boolean Algebra and logic. One sees easily that this is the following: if we call truth value x of a complex proposition **X** involving v and $\cdot$ a variable which is 1 if **X** is true and 0 of **X** is false, then

$$\mathbf{X} = f(P\ Q\ R \text{ involving v and } \cdot)$$

gives rise to

$$x = f(p\ q\ r \text{ involving v and } \cdot)$$

Example:

P = "Modern cars are slow"

Q = "Modern cars are underpowered"

R = "Modern cars eat a lot of gas"

The statement $\mathbf{X} = (\overline{P \text{ v } Q})\ R$ then reads:

> "It is not true that modern cars are slow or that modern cars are underpowered. Modern cars eat a lot of gas."

Let us examine $x = (\overline{p \text{ v } q})\ r$. Looking at P Q and R we see that $p = 0$, $q = 0$, $r = 1$. Therefore $x = (\overline{0 \text{ v } 0})\ 1 = 1$ i.e. **X** is a true true statement. Note that here $(\overline{P \text{ v } Q}) = \overline{P} \cdot \overline{Q}$!

There are other symbols for operations ("connectives") that we have used that reappear in the algebra of logic. Especially $P \leq Q$ means "either P is false or, if P is true, Q is true". Now we remember that $p \leq q$ was defined as $p \cdot q = p$ (or by the consistency principle $p \text{ v } q = q$!) Quite visibly the proposition $P \leq Q$ corresponds to the relationship $p \leq q$ between the truth values. We could similarly talk about $\oplus$, / etc. but a discussion would only lead to a reiteration of the postulates of Boolean Algebra. We leave it to the reader to verify that these postulates are satisfied by propositions.

Subsets of a Set

A set is a collection of objects or elements having some common identifying property e.g there is the set of all humans. A subset is a set included in a larger set: the set of all males is a subset of the set of all humans. Note that there are "null sets" i.e. sets without elements: the set of all humans having wings is a null set. The interesting thing is now that Boolean Algebra can be applied directly to all subsets x, y, z (e.g. x = males, y = children, z = females) of a given set S (e.g. S = humans). The given set S is called the "universal set" and in this application of Boolean Algebra is denoted by 1. Similarly null (sub-) sets are denoted by 0. Some subsets are complements of each other, i.e. in our example $\bar{x} = z$ (and evidently $x = \bar{z}$) meaning that all elements in one are definitely not in the other and vice-versa and that together they form the universal set.

It is often convenient to represent the universal set by all the points in a given closed curve and the subsets by smaller enclosed areas inside. Such a figure is called a Venn diagram. Often the universal set is taken to be enclosed in a rectangle. We could represent the set of humans, males, children and females as in Figure 3-4. Note that the region representing children must overlap both the male and the female region.

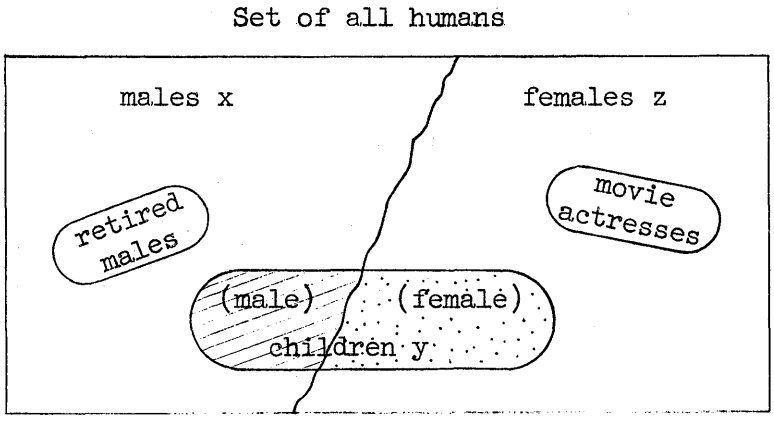


Figure 3-4

Venn Diagram for the Set of All Humans

The next step identifies the remaining fundamental operations v and · with operations on sets. v is identified with the union of two sets. In our example we can introduce a = set of male children b = set of female children. Then a v b = y is the set of all children. Such united sets must not necessarily correspond to adjacent regions in the Venn Diagram: let x_0 = set of retired males and z_0 = set of movie actresses; then x_0 v z_0 do not have to touch. Of course this means that there is no easily found common property of sets x_0 and z_0 (except perhaps that of being "unhappy people"). The symbol · is used to denote intersection: the intersection of two sets is a new subset containing all elements simultaneously in both the sets intersected. Above x y = a (i.e. the intersection of males and children are the male children), y z = b etc. When the intersection is a null set, we say that the sets used are disjoint; visibly x_0 and z_0 are disjoint because no retired male is a movie actress and vice-versa. We can write formally $x_0 \cdot z_0 = 0$.

We can now verify that all the postulates of Boolean Algebra are verified by subsets of a set. In particular such statements as (3-18) become quite intuitive: every subset is included in the universal set and can be no smaller than the null set. The attractive feature of illustrating Boolean Algebra by set theory is that Venn Diagrams give to the notion of minterm an easily graphed significance. Figure 3-5 shows the minterms of three variables x_1 x_2 and x_3.

Simplification of Boolean function can be obtained by drawing a Venn Diagram. Take for instance

$$f = \bar{x}_1 x_2 \bar{x}_3 \text{ v } \bar{x}_1 x_2 x_3 \text{ v } x_1 \bar{x}_2 \bar{x}_3 \text{ v } x_1 x_2 \bar{x}_3$$

illustrated on Figure 3-5 by the shaded and dotted areas. Visibly all these areas can be obtained by taking the union of $\bar{x}_1 x_2$ and $x_1 \bar{x}_3$. Therefore

$$f = \bar{x}_1 x_2 \text{ v } x_1 \bar{x}_3$$

Note that factoring on a Venn Diagram is made possible by the representation of minterms differing in one literal only by adjacent areas.

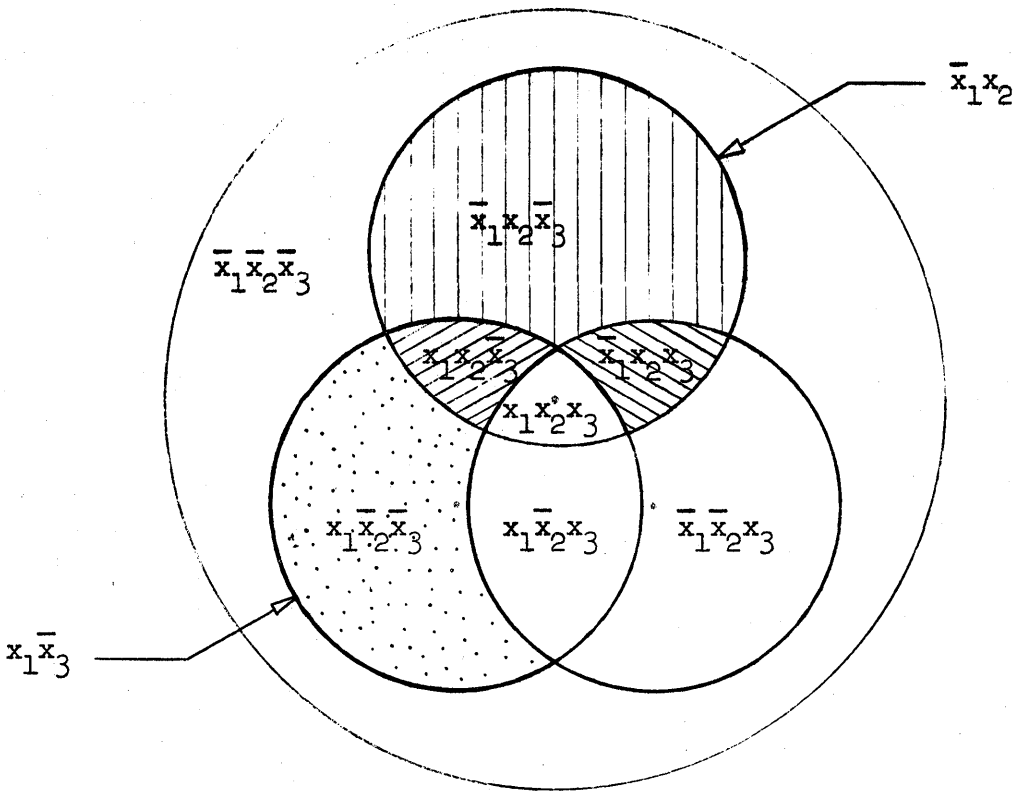


Figure 3-5
Venn Diagram for the Minterms of Three Variables

3.6 Karnaugh Maps

Veitch and Karnaugh have proposed a method of simplification of Boolean functions which uses essentially a highly simplified Venn Diagram. Let us consider a 2-variable case: Figure 3-6 shows how the notion of "adjacent regions" (i.e. regions having a common boundary line) can be taken from a Venn Diagram and transferred to regions arranged in a ring. It also shows how one can "cut open" this ring in order to form a 2-variable Karnaugh map: in the latter the left and right edges are considered adjacent by definition. Note that $\bar{x}_1 \bar{x}_2$ is represented by 00 etc. It is also possible to represent 2-variables in a square according to the upper part of Figure 3-6.

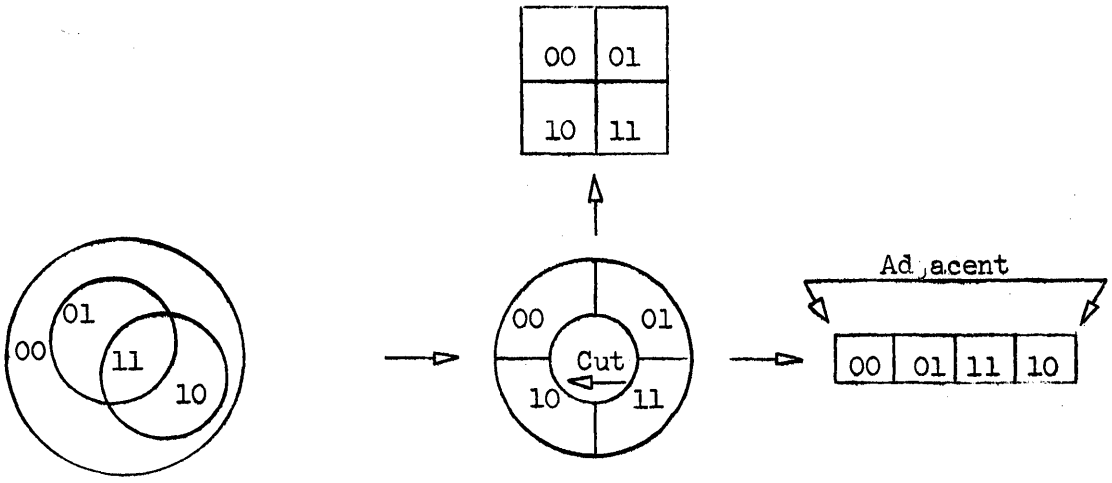


Figure 3-6
Transformation of a Venn Diagram into a Karnaugh Map

Figure 3-7 shows a 3-variable Karnaugh Map. Again the convention "left edge adjacent to right edge" gives us a layout such that adjacent squares correspond to binary expressions differing in one digit only, i.e. minterms differing only in the complementation of one literal. Figure 3-8 extends the map to 4-variables: visibly we again differ in adjacent squares by 1 digit of the binary number if we agree to not only consider the left and right edge as adjacent but also the upper and lower edge.

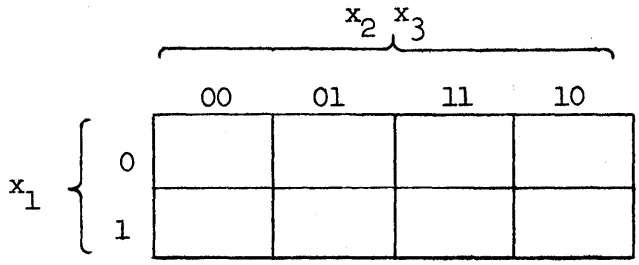


Figure 3-7
3-Variable Karnaugh Map

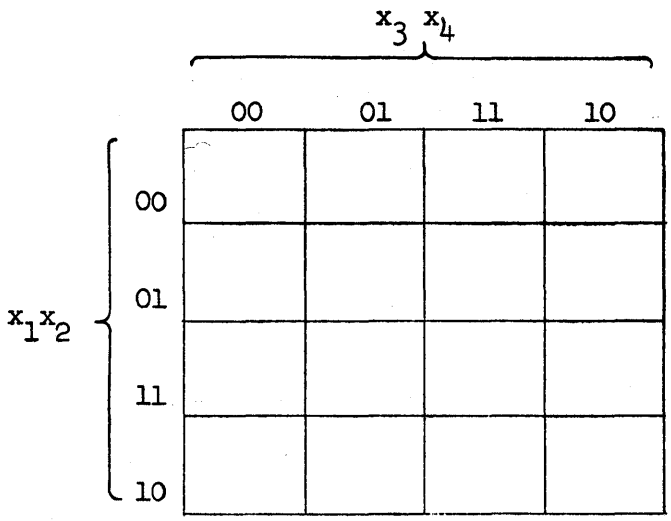


Figure 3-8

4-Variable Karnaugh Map

Note that it is actually possible to transform these fictitious adjacencies into real ones by drawing the 4-variable map on a toroid (doughnut) as in Figure 3-9. Obviously the practical usefulness of a Karnaugh Map in space in slightly doubtful.

To represent a function f on a map, we place 1's in the squares for which the corresponding minterm is included in f and 0 elsewhere. The function

$$f = \bar{x}_1 x_2 \bar{x}_3 \vee \bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3 \vee x_1 x_2 \bar{x}_3$$

is thus represented by the map of Figure 3-10. The operations to be performed to find that $f = \bar{x}_1 x_2 \vee x_1 \bar{x}_3$ are to examine the map for the presence of 1's in adjacent squares. Two such squares can be combined to give a term with one variable less. Four such adjacent squares would be combined to give a term with two variables less.

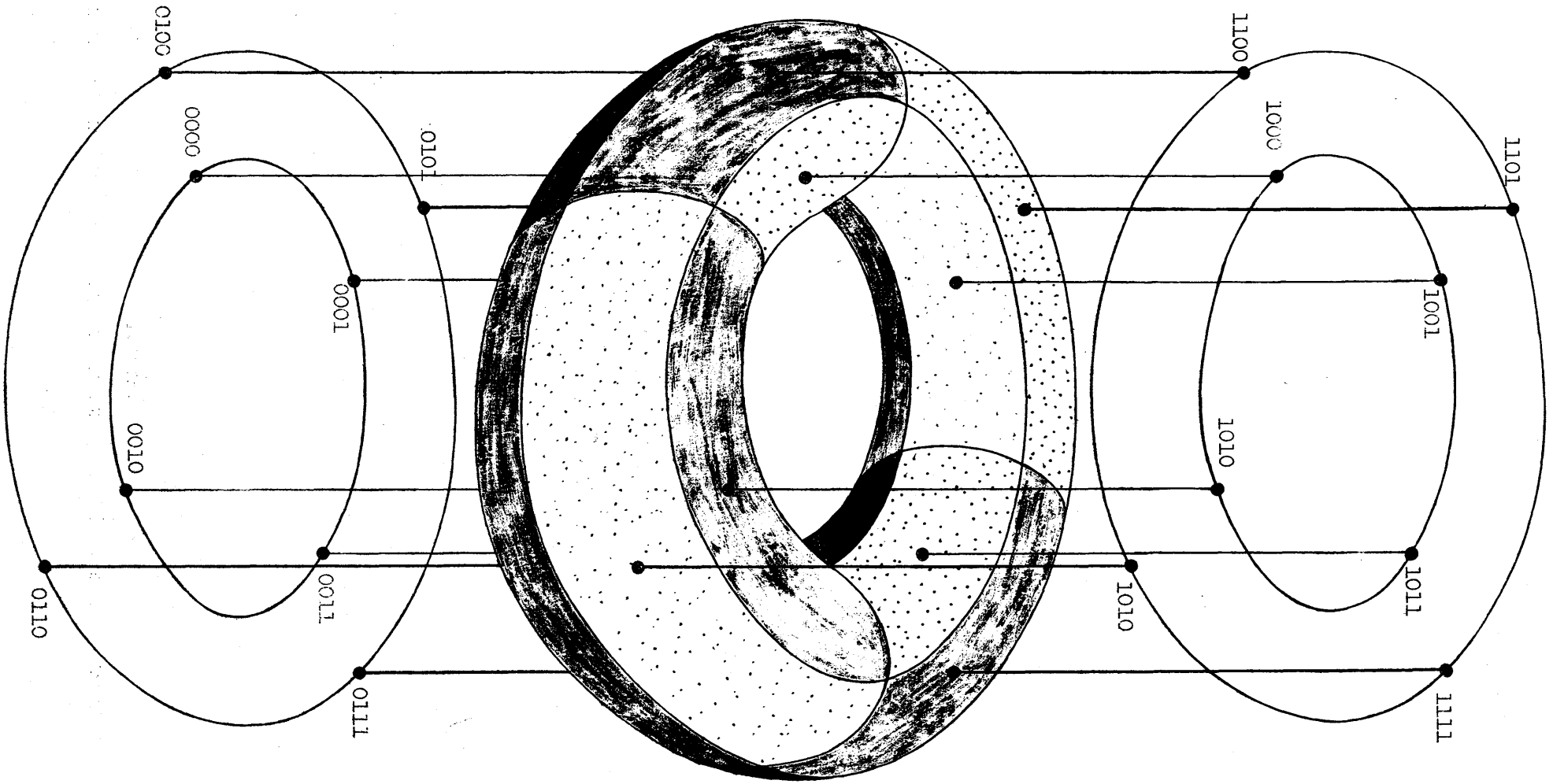


Figure 3-9

4-Variable Karnaugh Map Without Fictitious Adjacencies Drawn on a Toroid

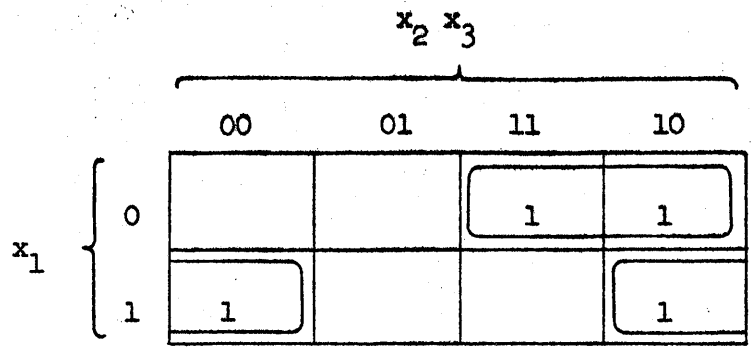


Figure 3-10

Karnaugh Map for $\bar{x}_1x_2\bar{x}_3 \vee \bar{x}_1x_2x_3 \vee x_1\bar{x}_2\bar{x}_3 \vee x_1x_2\bar{x}_3$

(i.e. 010 v 011 v 100 v 110)

In our example we can combine 011 and 010 into 01- $\rightarrow \bar{x}_1x_2$ and 100 and 110 into 1-0 $\rightarrow x_1\bar{x}_3$.

Figure 3-11 gives an example for a 4-variable map. Here

$$f = \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4 \vee \bar{x}_1\bar{x}_2\bar{x}_3x_4 \vee \bar{x}_1\bar{x}_2x_3\bar{x}_4 \vee \bar{x}_1x_2\bar{x}_3x_4$$
$$\vee x_1\bar{x}_2\bar{x}_3\bar{x}_4 \vee x_1\bar{x}_2\bar{x}_3x_4 \vee x_1\bar{x}_2\bar{x}_3\bar{x}_4$$

is drawn on the map. Remembering again that the 4 edges must be considered asjacent by pairs, we see we can form two 4-square combinations and a 2-square combination to cover all 1's. This leads to a simplified expression

$$f = \bar{x}_2\bar{x}_4 \vee \bar{x}_2\bar{x}_3 \vee \bar{x}_1\bar{x}_3x_4 .$$

This last case is a good illustration of how we can also use negatives on maps. It is easily seen that $\bar{f}$ would be represented by

interchanging 0 and 1 on all squares: This is a consequence of the fact (see the "Theorem on AND to OR Transformation" of Section 3.2) that $\bar{f}$ contains all the minterms which are not in f. It is possible that the pattern formed on the "negative map" (usually drawn by grouping the zeros on the original map!) is much cimpler. In our example this is certainly the case, since all 0's can be covered by three 4-square combinations, i.e.

$$\bar{f} = x_2\bar{x}_4 \vee x_3x_4 \vee x_1x_2 .$$

This is shown in Figure 3-12.

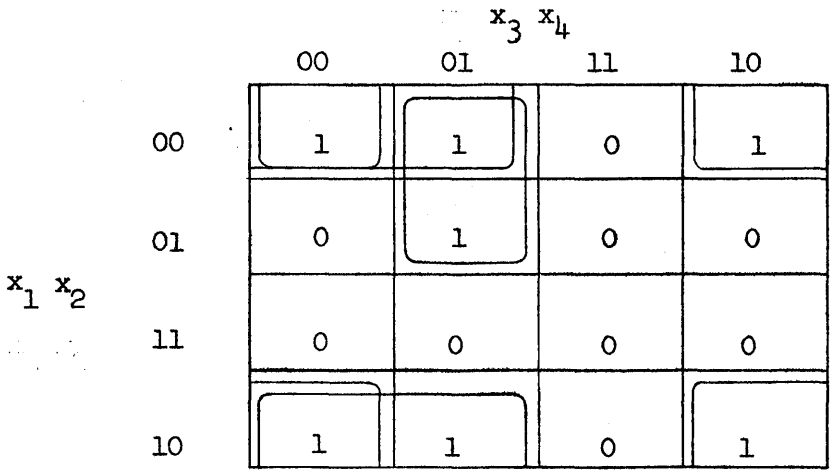


Figure 3-11

Karnaugh Map for $f = m_0 \vee m_1 \vee m_2 \vee m_5 \vee m_8 \vee m_9 \vee m_{10}$

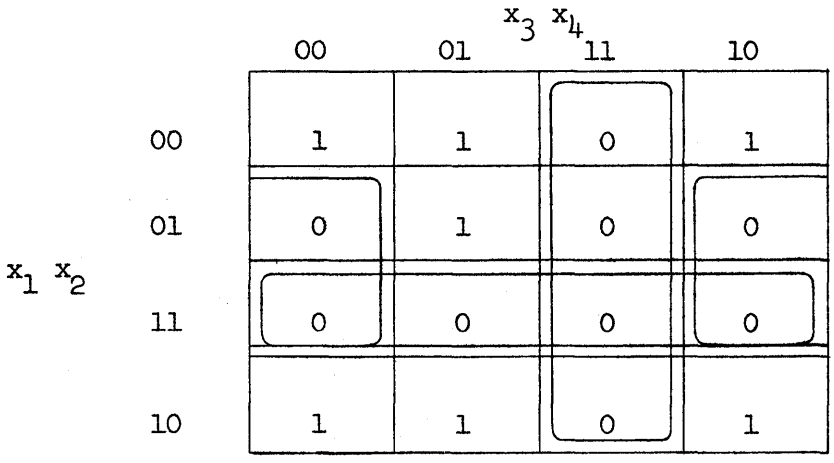


Figure 3-12

Karnaugh Map for the Negative of Figure 3-11

When we desire to draw maps for more than 4-variables, we still would like to draw adjacent to each square the five or more minterms that only differ in one digit. Short of drawing such a map in space (over 6-variables this fails too!) we can only introduce new conventions, calling "adjacent" squares which are either physically adjacent (including the use of the edge-convention) or which fulfil some other easily verified criterion. It turns out that for both 5 and 6-variables such a criterion is symmetry with respect to certain lines as shown in Figure 3-13 and 3-14. That in the 5-variable case, squares symmetrically placed with respect to the vertical center-line differ by one digit only, can be verified by noting that the column-headings 000, 001, 011, 010, 110, 111, 101, 100 have this symmetry: 000 is paired off with 100 etc. In other words: it is the fact that the column headings can be arranged to show this symmetry and to differ by one digit from left to right that makes a 5-variable map feasible. For 6 variables we shall obviously use row headings similar to the column headings; this time two lines of symmetry have to be taken account of in all decisions about adjacency.

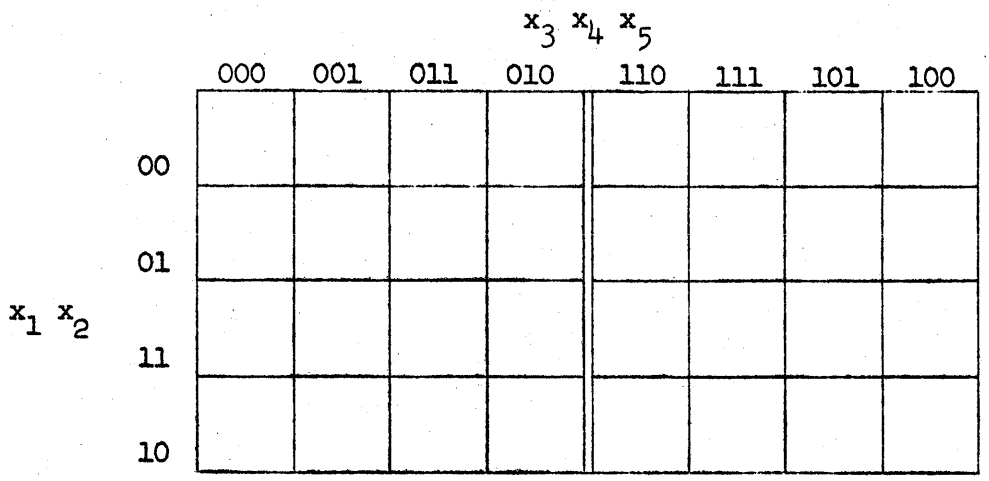


Figure 3-13
5-Variable Karnaugh Map

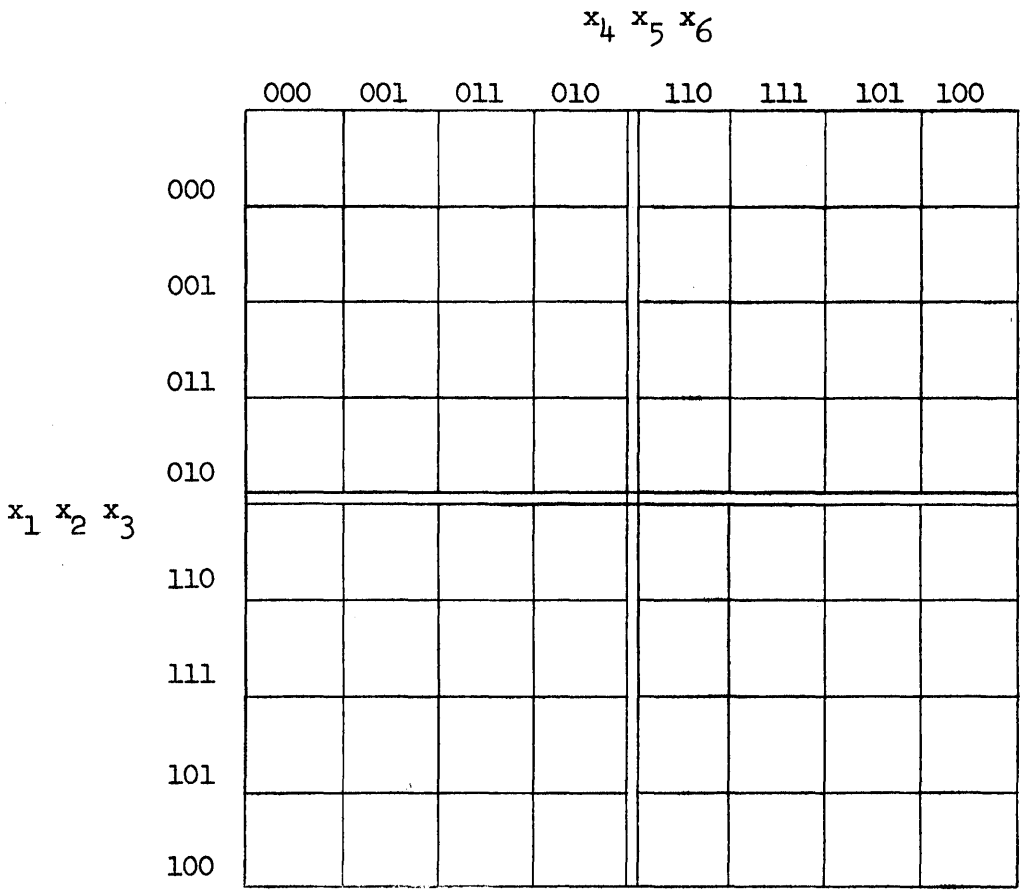


Figure 3-14

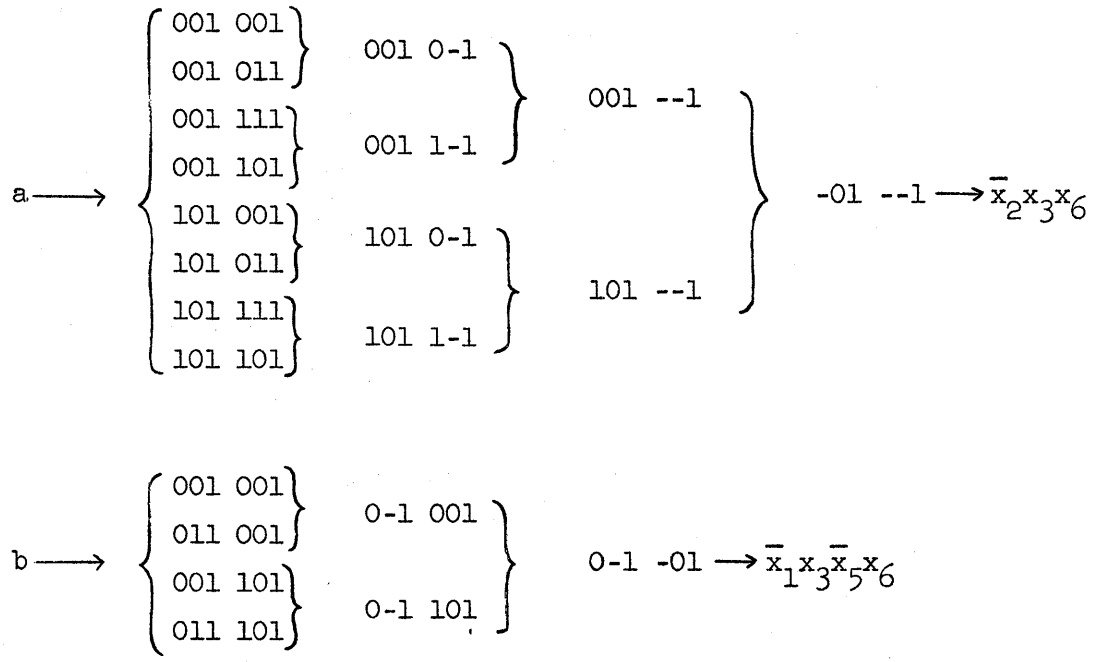
6-Variable Karnaugh Map

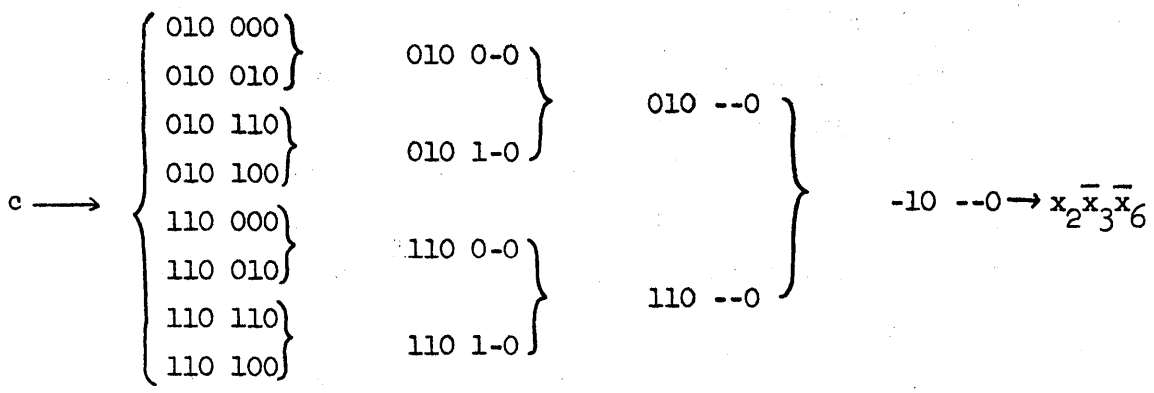
As an example let us consider a 6-variable case in which f contains the following minterms:

9 ⟶ 001 001
11 ⟶ 001 011
13 ⟶ 001 101
15 ⟶ 001 111
16 ⟶ 010 000
18 ⟶ 010 010

20 ⟶ 010 100
22 ⟶ 010 110
25 ⟶ 011 001
29 ⟶ 011 101
41 ⟶ 101 001
43 ⟶ 101 011
45 ⟶ 101 101
47 ⟶ 101 111
48 ⟶ 110 000
50 ⟶ 110 010
52 ⟶ 110 100
54 ⟶ 110 110

Figure 3-15 shows the map. It is easy to see that because of the high degree of symmetry complete coverage can be obtained by grouping together the 1's marked





Therefore $f = \bar{x}_2x_3x_6 \vee \bar{x}_1x_3\bar{x}_5x_6 \vee x_2\bar{x}_3\bar{x}_6$

	000	001	011	010	110	111	101	100
000	0	0	0	0	0	0	0	0
001	0	1 a/b	1a	0	0	1a	1 a/b	0
011	0	1b	0	0	0	0	1b	0
010	1c	0	0	1c	1c	0	0	1c
110	1c	0	0	1c	1c	0	0	1c
111	0	0	0	0	0	0	0	0
101	0	1a	1a	0	0	1a	1a	0
100	0	0	0	0	0	0	0	0

Figure 3-15

Example of a 6-Variable Karnaugh Map

Multi-Level Factoring

The simplification (or "factoring") on a Karnaugh map yields a minimum - v - polynomial. It is often possible to obtain further simplification by abandoning the minimum - v - form: this is the process referred to in 3.3 as involving "skill and flair". The Karnaugh map itself can be a useful tool for this further simplification as an example will show. Take

$$f = m_1 \vee m_5 \vee m_9 \vee m_{10} \vee m_{11} \vee m_{13} \vee m_{14}$$

This function is represented on the map of Figure 3-16a and leads to the simplified form

$$f = \bar{x}_3 x_4 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_3 \bar{x}_4$$

which can be further simplified to give the (non v-polynomial) form

$$f = \bar{x}_3 x_4 \vee x_1 x_3 [\bar{x}_2 \vee \bar{x}_4]$$

	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	0	1	0	1
10	0	1	1	1

Figure 3-16a

Karnaugh Map for $f = m_1 \vee m_5 \vee m_9 \vee m_{10} \vee m_{11} \vee m_{13} \vee m_{14}$

The reduction of $x_1\bar{x}_2x_3 \vee x_1x_3\bar{x}_4$ to $x_1x_3\,[\bar{x}_2 \vee \bar{x}_4]$ can be done directly by observing that $x_1x_3\,[= x_1x_3\,(x_2 \vee \bar{x}_2)(x_4 \vee \bar{x}_4) = x_1x_2x_3x_4 \vee x_1\bar{x}_2x_3x_4 \vee x_1x_2x_3\bar{x}_4 \vee x_1\bar{x}_2x_3\bar{x}_4]$ covers the four squares in the bottom left-hand corner of 3-16a. Not all of these squares are included (1111 has a zero in it) but it is obviously possible to include all four - neglecting this zero - if we make sure to multiply x_1x_3 by an expression which is zero for $x_2x_4 = 1$; this expression should be as simple as possible. The important thing is that we can draw an x_2x_4 map directly on the map of Figure 3-16a: it will not be in the form of the standard Karnaugh map but rather in the form shown in the upper part of Figure 3-6 as an alternate. Figure 3-16b shows this "sub-map" separately: from it we see that a way of covering the 2-variable function $f(x_2x_4)$ which is 1 everywhere except in 11 is $\bar{x}_2 \vee \bar{x}_4$. This result could, of course, be obtained directly if - after covering 1111, 1110, 1011 and 1010 - we interpreted the lower right-hand corner of Figure 3-16a in terms of Figure 3-16b. One way to do this is shown in Figure 3-17: we include 0's in our covering, but mark them with an asterisk and eliminate them by multiplying the term covered by the expression read from this covering interpreted as a sub-map.

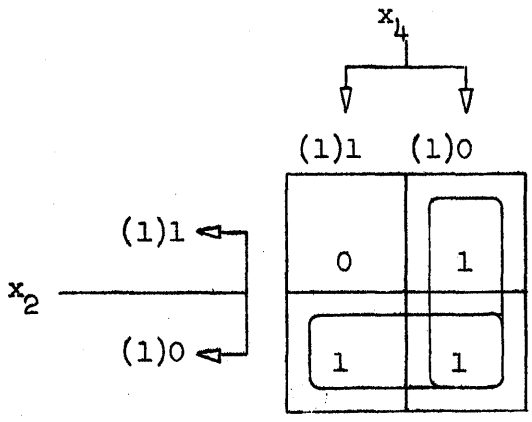


Figure 3-16b
Sub-map of the Map in Figure 3-15

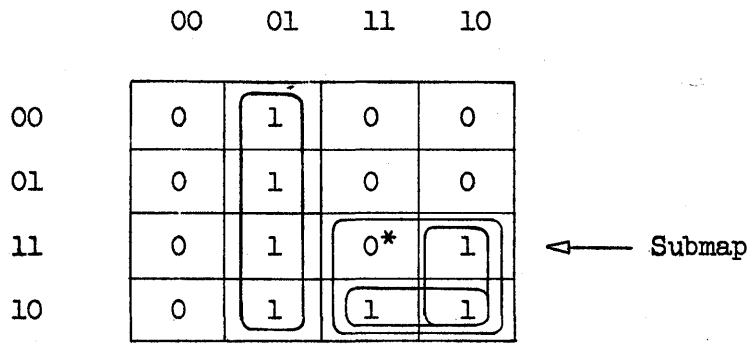


Figure 3-17
Multi-Level Factoring for the Problem of Figure 3-16a

3.7 Don't Care Conditions (Optional Terms) and Multi-function Problems

Optional Terms in Karnaugh Maps

It often happens that the output of a combinational circuit is only defined for a limited number of input combinations. A typical example of such a situation would be a base which has inputs from two flipflops and which gives a "1" output if the flipflop states agree. In Figure 3-18 we can, therefore, never have x_1 and x_2 equal or x_3 and x_4 equal. This means

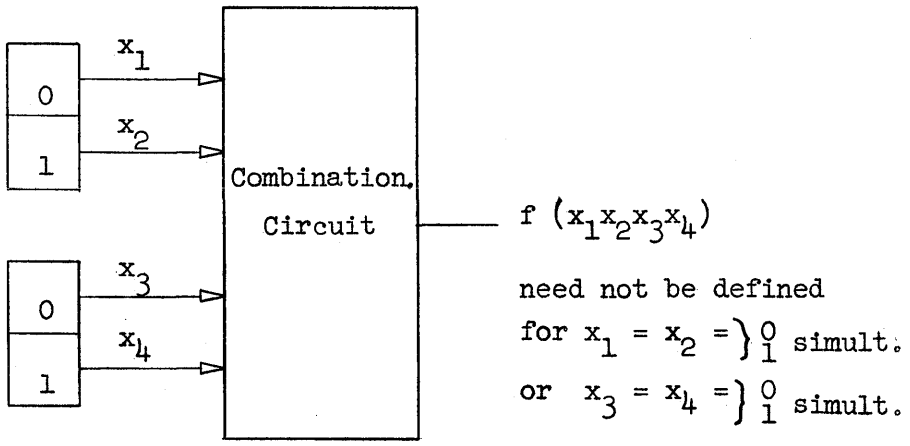


Figure 3-18
Don't Care Condition Circuit

that we can list the properties of f in a table as follows:

	x_1	x_2	x_3	x_4	f
0	0	0	0	0	D.C.
1	0	0	0	1	D.C.
2	0	0	1	0	D.C.
3	0	0	1	1	D.C.
4	0	1	0	0	D.C.
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	D.C.
8	1	0	0	0	D.C.
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	D.C.
12	1	1	0	0	D.C.
13	1	1	0	1	D.C.
14	1	1	1	0	D.C.
15	1	1	1	1	D.C.

The idea is now to simplify f to the utmost (using for instance Quine's method or a Karnaugh map) using the fact that we can assign to f an arbitrary value for certain input combinations. Our problem could be stated by writing

$$f = \Sigma\ 5,\ 10$$

$$d = \Sigma\ 0,\ 1,\ 2,\ 3,\ 4,\ 7,\ 8,\ 11,\ 12,\ 13,\ 14,\ 15$$

This gives us a Karnaugh map with a 1 in squares 5 and 10, a 0 in squares 6 and 9 and an x elsewhere. As shown in Figure 3-19 it is obvious that if we assume that the circled x's are 0 and the others 1, we obtain a high degree of symmetry and consequently a simple form for f. In our example $f = x_2x_4 \vee \bar{x}_2\bar{x}_4$. This was, of course, evident from the outset, since the state of the flipflop could have been sensed by 2 wires rather than four.

	00	01	11	10
00	x	⊗	⊗	x
01	⊗	1	x	0
11	⊗	x	x	⊗
10	x	0	⊗	1

Figure 3-19
Don't Care Condition Karnaugh Map

Optional Terms in Quine's Method

The treatment of optional terms in Quine's method is exceedingly straightforward. Let f and d be respectively the sum of the minterms producing a certain pattern of 1's and the sum of the minterms producing a certain pattern of x's (→ don't care conditions). Then we search for the prime implicants of all terms in f and d (this will give usually more prime implicants than if we had taken f only). The prime implicant chart, however, is made up using only the minterms in f: one of the consequences of this is that the number of intersections (or crosses) per prime implicant is no longer a power of two. Since we have more prime implicants and fewer minterms to make up, it is evident that usually simplification beyond that for f alone can be obtained!

Example: Let us take a 4-variable problem with

$$f = \Sigma\, 2,\ 3,\ 7,\ 9,\ 11,\ 13$$

$$d = \Sigma\, 1,\ 10,\ 15$$

Successive reduction gives:

1	0001 v	1,3	00-1 v	1,3,9,11	-0-1	← A
2	0010 v	1,9	-001 v	2,3,10,11	-01-	← B
3	0011 v	2,3	001- v	3,7,11,15	--11	← C
9	1011 v	2,10	-010 v	9,11,13,15	1--1	← D
10	1010 v	3,7	0-11 v			
7	0111 v	3,11	-011 v			
11	1011 v	9,11	10-1 v			
13	1101 v	9,13	1-01 v			
15	1111 v	10,11	101- v			
		7,15	-111 v			
		11,15	1-11 v			
		13,15	11-1 v			

The prime implicant chart is shown in Figure 3-20. Visibly

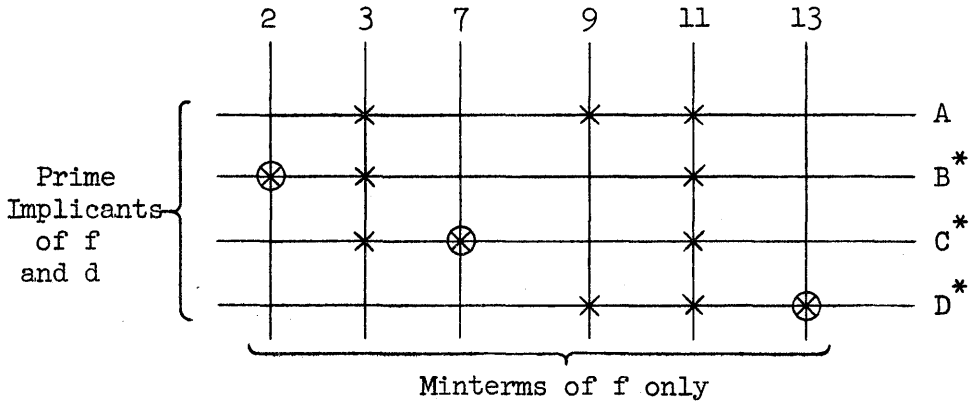


Figure 3-20

Prime Implicant Chart for a Don't Care Problem

the function can be formed by taking B, C and D only i.e.

$$f = \bar{x}_2 x_3 \vee x_3 x_4 \vee x_1 x_4$$

Simultaneous Simplification of Several Functions

A very common problem is to design a box having inputs $x_1 \ldots x_n$ and several outputs $f_1 (x_1 \ldots x_n)$, $f_2 (x_1 \ldots x_n) \ldots f_m (x_1 \ldots x_n)$. Again we would like the contents of this box to be as simple as possible. Unluckily no general methods are known. A method which is sometimes quite useful is the method of assumed form in which one assumes that $f_1 \ldots f_m$ contain a common factor ϕ such that

$$f_1 = \phi (x_1 \ldots x_n) \cdot F_1 (x_1 \ldots x_n)$$

$$f_2 = \phi (x_1 \ldots x_n) \cdot F_2 (x_1 \ldots x_n) \quad \text{etc.}$$

The problem is then to find a set ϕ $F_1 \ldots F_m$ which makes all functions very simple. The obvious difficulty is that in most cases ϕ can be simplified at the expense of $F_1 \ldots F_m$ and vice-versa: which choice is best can only be determined by trial and error.

The idea is now to represent $f_1 \ldots f_m$ ϕ and $F_1 \ldots F_m$ by Karnaugh maps. Those for $f_1 \ldots f_m$ are determined by the problem. In order to find the other ones we use the following

Theorem: If f_i has a zero in a given square, then either ϕ of F_i must have a zero in the corresponding squares. If f_i has a one in a given square, then both ϕ and F_i must have a one in the corresponding squares.

Proof: The theorem is evident, since all it says is that $f_i = \phi \cdot F_i$ implies that for a given combination of inputs ($\rightarrow$ to a square on the map) $f_i = 1$ means both $\phi = 1$ and $f_i = 1$, while $f_i = 0$ is satisfied if $\phi = 0$ or $F_i = 0$.

This theorem then says that the ϕ-map must have the 1's of all the f_i-maps combined. Once these 1's are drawn in, we can add optional 1's if we are careful to "block them out" on the F_i maps by 0's. Furthermore we can add optional 1's on the F_i maps once we have made sure that the 0's of f_i are secured by appropriate blocking of 1's in ϕ not in f_i. This juggling process finally leads to relatively simple maps for all functions and therefore solves the problem. An example will illustrate the method.

Example: Simplify simultaneously

$$f_1 = x_1 x_2 \bar{x}_3 \vee x_1 x_4$$

$$f_2 = x_1 \bar{x}_2 x_4 \vee \bar{x}_1 x_2 x_3 \bar{x}_4$$

We draw first Karnaugh Maps for f_1 and f_2 and three further ones for ϕ, F_1 and F_2, the latter three for the moment without entries. Figure 3-21 gives a convenient layout. The rules are as follows:

1. Fill in the ϕ-map with the 1's of both f_1 and f_2.
2. Add 1's on the ϕ-map in such a way that it becomes as symmetric as possible.*
3. Draw a map for F_1 having all the 1's of f_1. Put 0's in all positions in which ϕ shows a 1 but f_1 a 0. (This "blocking of ones" is obviously necessary because of the theorem cited above.)
4. Proceed similarly for F_2.
5. Symmetrize the F_1 and F_2 maps by adding 1's in appropriate squares. (Note that this does no harm since ϕ has 0's in those positions.)

---

* This last point is doubtful: less symmetry in ϕ may mean more symmetry in F_1 and F_2!

In Figure 3-21 we symmetrize ϕ by making the squares marked x $\rightarrow$ 1 while F_1 and F_2 are symmetrized by making a $\rightarrow$ 0 and b $\rightarrow$ 0 while y $\rightarrow$ 1 and z $\rightarrow$ 1. The end result is that

$$\phi = x_1 \vee x_3\bar{x}_4$$

$$F_1 = x_2\bar{x}_3 \vee x_4$$

$$F_2 = \bar{x}_1x_2 \vee \bar{x}_2x_4$$

f_1

	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	0
10	0	1	1	0

f_2

	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	0
10	0	1	1	0

ϕ

	00	01	11	10
00	0	0	0	x
01	0	0	0	1
11	1	1	1	x
10	x	1	1	x

F_1

	00	01	11	10
00	a	y	y	0
01	y	y	y	0
11	1	1	1	0
10	0	1	1	0

F_2

	00	01	11	10
00	b	z	z	0
01	z	z	z	1
11	0	0	0	0
10	0	1	1	0

Figure 3-21

Multifunction Simplification in the Method of Assumed Form

CHAPTER IV

OPERATION OF A DIGITAL COMPUTER SYSTEM

4.1 The Illiac I Arithmetic Unit

Figure 4-1 gives the general layout of the arithmetic unit in a rather typical computer i.e. Illiac I. A, $\overline{A}$, Q, $\overline{Q}$, R^3 and R_3 are registers holding 40 binary digits each. The names commonly given to these registers are:

A: accumulator register
$\overline{A}$: temporary accumulator register
Q: multiplier-quotient register
$\overline{Q}$: temporary multiplier-quotient register
R^3: number register
R_3: order register

In the diagram we symbolize any kind of gate (or group of gates) by a circle containing a combination of a letter and G or a G with a subscript. R, O, B, Y and G stand for red, orange, black, yellow and green. When any one of the group of gates RG, OG, BG or YG is open, the effects are a shift between A and $\overline{A}$ and (except for OG) simultaneously between Q and $\overline{Q}$. More specifically

RG shifts left down
OG shifts straight down
BG shifts right down
YG shifts straight up.

The group of gates GG connect the output of the adder to $\overline{A}$. The inputs to the adder come from A (connected permanently) and from R^3: when u = 0, the contents are transferred directly to the adder, while u = 1 entails complementation (two's complement, formed as in Figure 2-15).

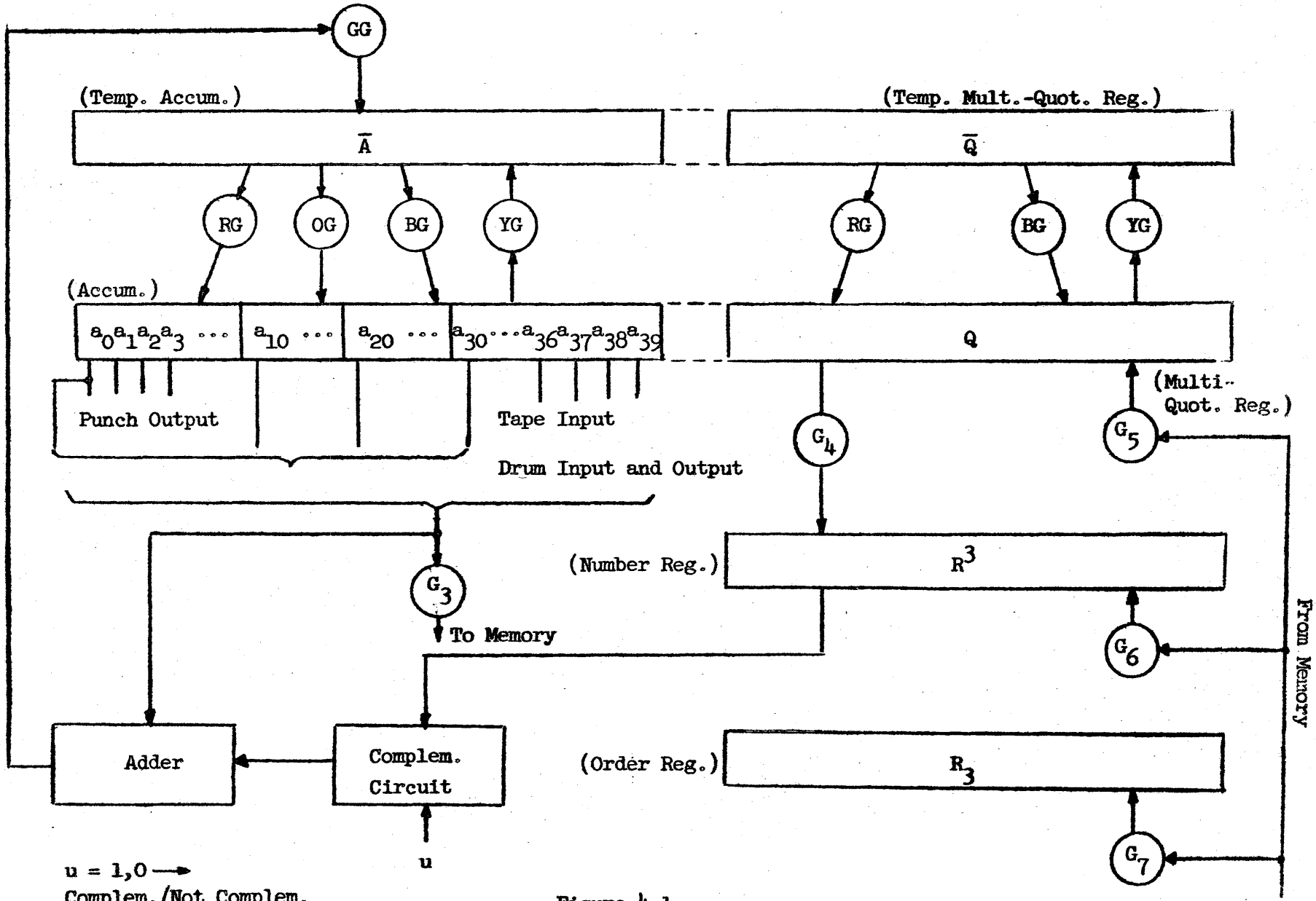
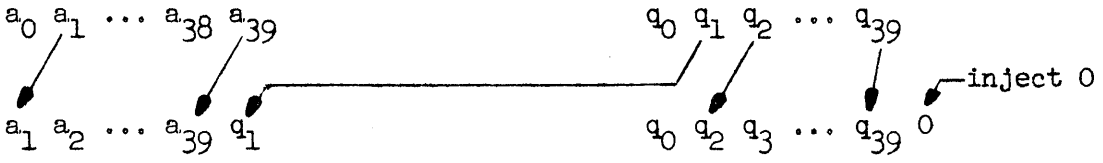


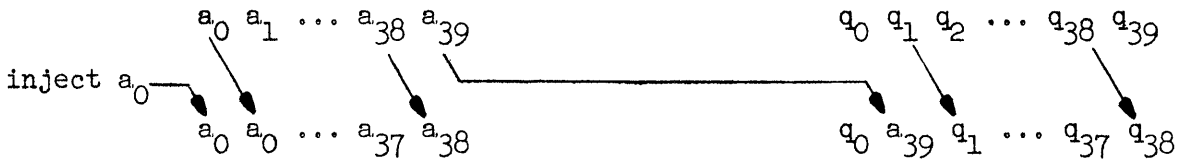
Figure 4-1

Illiac Arithmetic Unit

The lines between A and Q as well as $\overline{A}$ and $\overline{Q}$ indicate that, when any one of the groups RG or BG is used, the registers act as one. This means that for a right-down shift the least significant digit of $\overline{A}$ goes into the most significant non-sign position of Q. Incidentally a_0, where a_0 is the sign digit in $\overline{A}$, is shifted into the most significant position of A during this operation. A therefore starts out with $a_0 a_0$. For a left-down shift the most significant non-sign digit of $\overline{Q}$ goes into the least significant position of A. During this operation a "0" is shifted into the least significant position of Q. Figure 4-2 illustrates the effect of shifting right of left. Because of these connections between the accumulator and the multiplier-quotient register, one often talks of a double-length register AQ or $\overline{AQ}$.



Illiac Left Shift



Illiac Right Shift

Figure 4-2
Illiac Shift Patterns

The accumulator is used for intermediary storage in all communications with the outside world or the memory. For the tape input, digits are read serially from the tape into the four least significant positions and then shifted left (passing through $\overline{A}$). For the punch or teletype output an analogous situation holds, except that the four most significant digits are used. In case the drum

is used (for input or output), digits a_0 a_{10} a_{20} and a_{30} are connected simultaneously to the drum. After 11 (actually only 10 useful) shifts the "tetrads" have exhausted the whole word. This scheme permits speeding up the transfer. Finally, when G_3 is open, 40 digits are read into the memory in parallel fashion.

The memory can send information to Q (through G_5), to R^3 (through G_6) and to R_3 (through G_7). R_3 contains the order (actually a pair of orders -- see next section) currently being followed. Each order consists of an instruction of 8 digits (add, multiply etc.) and an address of 10 digits. Decoding circuits decode the instruction and set the internal connections in the machine in an appropriate way.

It should be noted that Q cannot communicate with the memory. In order to be able to read out information in Q, this information is first transferred to R^3 (via G_4) and then added to zero in A (via G_2). The way followed is thus: $Q \rightarrow R^3 \rightarrow$ Adder $\rightarrow \overline{A} \rightarrow A$. This seemingly complicated procedure allows the contents of Q to be modified on their way to A.

4.2 Illiac I Control

Decoding and Dispatch Counting Circuits

Figure 4-4 shows the block-diagram of this part of the machine. Lying in R_3 is shown an order pair consisting of two eight digit instructions, two ten digit addresses and two waste spaces of two digits each -- Figure 4-3 shows this arrangement in detail.

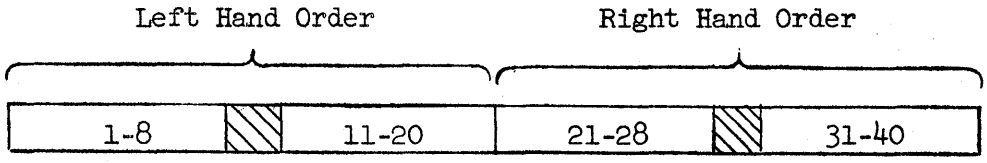


Figure 4-3
Illiac I Order Pair

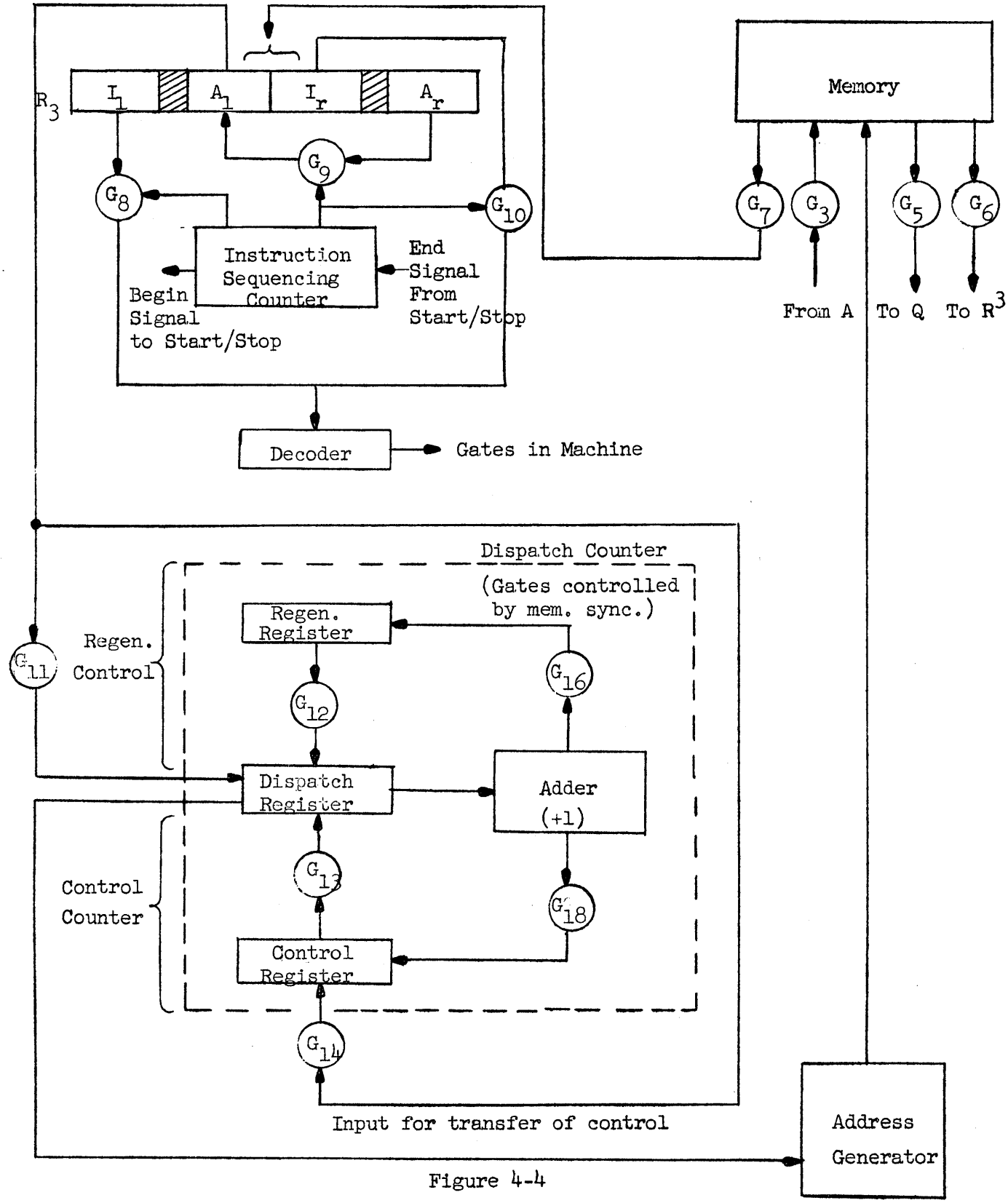


Figure 4-4

Illiac I Decoding and Dispatch Counting Circuits

This order pair has been brought into R_3 in the following way. The control register contains the address of the location of the order pair. This address is gated into the dispatch register and sets the address generator. The address generator chooses the memory location, the contents of which are transferred to R_3 via G_7. Now one unit is added to the address in the dispatch register and the result is gated into the control register.

Next the instruction sequencing counter puts the left hand address into the decoder, a device which sets certain flipflops in the machine and opens certain paths according to the type of instruction. Simultaneously the left hand address is gated into the dispatch register.

If the instruction happens to involve a transfer of control, the new address is transferred to the control register via G_{14} and then gated into the dispatch register upon the arrival of an end-signal. The process described above then occurs a second time. If there is no transfer of control, the left hand address is put into the dispatch register and sets up the necessary memory connections via the address generator. The machine then executes the given instruction, the instruction sequencing counter being tied to the start/stop control. Upon the arrival of the end-signal of the operation, the instruction sequencing counter gates the right hand instruction into the decoder and simultaneously the right hand address into the part of the register previously occupied by the left hand address: the right hand address thus goes into the dispatch register. A transfer of control operates as before. If there is no transfer, the right hand instruction is executed. The end-signal again causes the contents of the control register to be gated into the dispatch register. If there was no transfer of control in the two instructions, the contents of the control register at this time are the address of the previous order pair increased by one: the machine goes through the memory location in sequence. It is clear that programming errors can cause number locations to be interpreted as order locations.

As the memory system in Illiac I is regenerative, i.e. since all memory locations must be scanned periodically and renewed, there is a regeneration register attached to the dispatch register. During the

regeneration cycle the contents of this register, increased by one, are gated into the dispatch register and determine, via the address generator, the location of the next word to be renewed.

Control for the Arithmetic Unit

As indicated in Figure 4-5, there are 4 principal parts in this section of Illiac I: the shift control, composed of the shift sequencing unit (See Section 2.4) and the clear and gate selector, a shift counter (as described in Section 2.2), a recognition circuit and a start/stop control.

The recognition circuit has a dual purpose. If the instruction in R_3 is a shift instruction, the address part gives the number of shifts to be performed. The recognition circuit then compares the number of shifts as counted by the shift counter to this predetermined number and, upon coincidence, acts on the start/stop control. If the instruction in R_3 is a multiply or division instruction, the machine must go through 39 add-and-shift or subtract-and-shift cycles. This time the recognition circuit acts on the start/stop control upon the advent of 39 shifts.

Notice that the shift counter has a "reset" input: this clears all counter flipflops to zero before operations commence. The "up" and "down" pulses are taken directly from the gates between A, $\overline{A}$, and Q and $\overline{Q}$: red, orange or black gates give a "down" signal while the yellow gate gives an "up" signal. Which group of gates is actually chosen depends upon the signals from the decoder, i.e. upon the instruction followed.

4.3 Illiac Memory Circuits and Interplay Control

Figure 4-6 shows the layout of the memory circuits and what is called the "interplay control", i.e. the part of the machine which directs the transfer of information from the (synchronous) memory to the (asynchronous) registers and vice-versa.

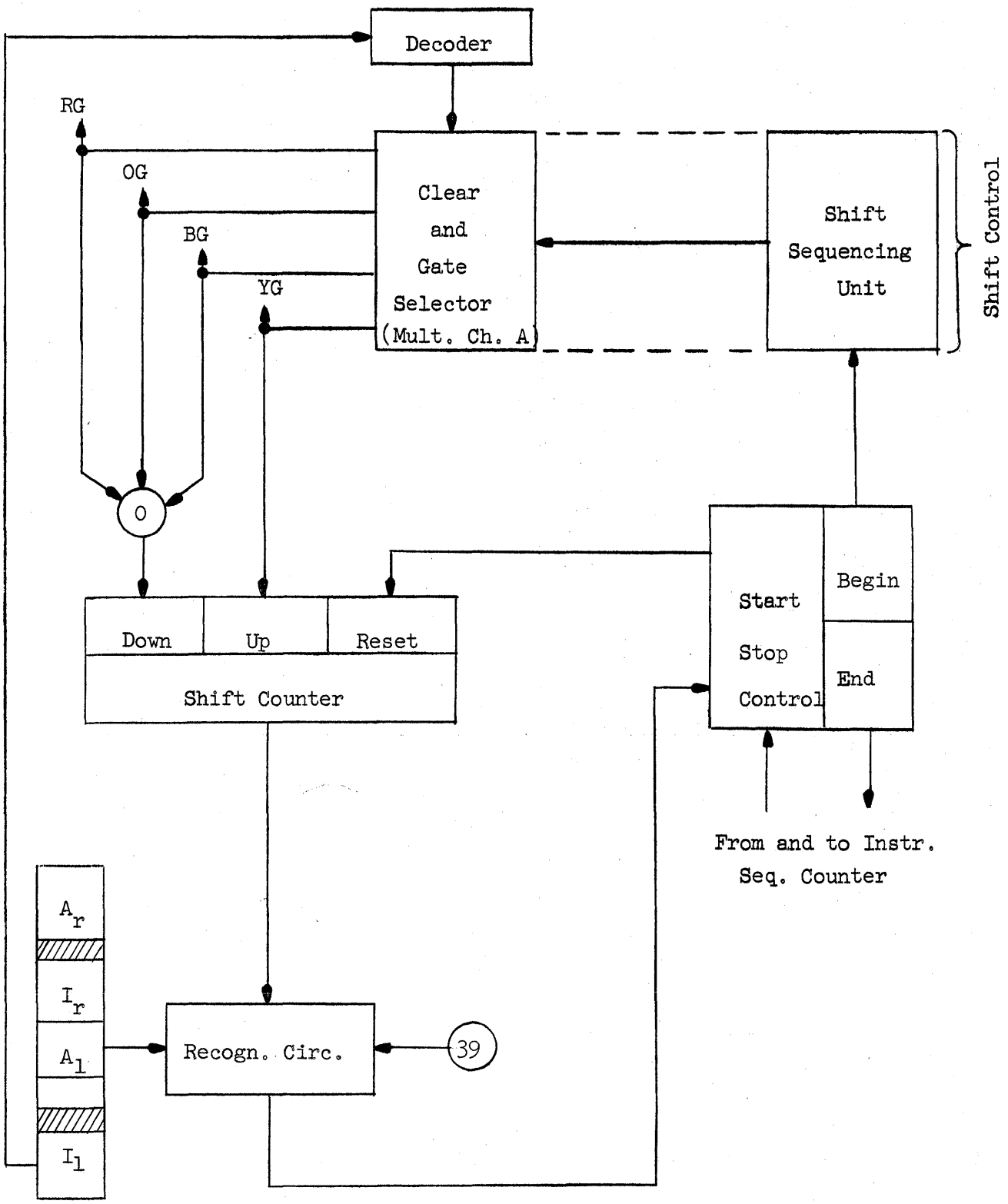


Figure 4-5

Illiac I Control for the Arithmetic Unit

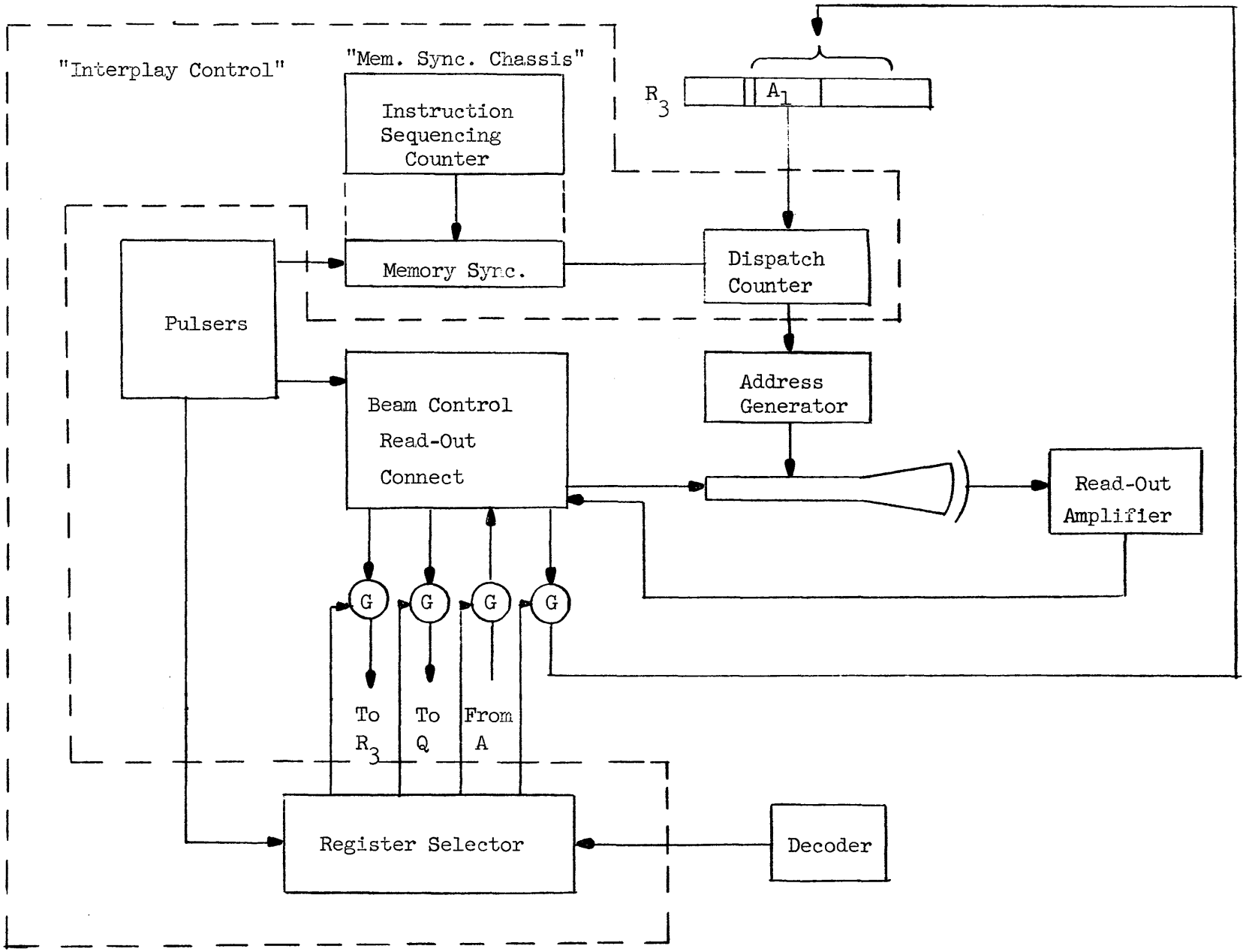


Figure 4-6

Illiac Memory Circuits and Interplay Control

The Illiac memory uses cathode ray tubes as storage elements, each one of the 40 digits in a word coming from a different storage tube. The storage of a "1" is obtained by brightening up any one of the sports in a 32 x 32 raster: there are therefore memory locations numbered from 0 through 1023. By measuring the beam current when the beam is directed to a given location by the address generator (all beams are controlled in parallel!), one can determine whether a "0" or a "1" has been stored. A read-out amplifier detects the signal when the beam-control turns on the beam: a pulse corresponding to a "1" sets a flipflop (previously cleared) in the beam-control. Depending upon the state of this flipflop, a "1" or a "0" is written back immediately to eliminate loss of information.

As mentioned before, the memory can be in either one of two modes: an action-cycle (read or write) or a regeneration-cycle (transformation of "fair" charge distributions in the cathode ray tubes to "good" distributions). In order to read out of the memory, one only has to examine the state of the 40 output flipflops mentioned above. For writing into the memory the restoring process described above is used: a "1" is always written back and writing a "0" involves an additional motion of the beam. This additional motion is suppressed if a "1" is to be written. The regeneration cycle is fundamentally the same as the "read-and-restore" process.

The decoder acts on a register selector which in turn establishes the connections for the transfer of signals to the registers. It is, however, important to remember that the memory is synchronous and contains a clock and a pulser chain. These pulses control the moment of transfer of information through the register selector chassis. They also control the action of the dispatch counter (control counter and regeneration counter) e.g. when regeneration occurs the address is stepped up by one unit by means of a clock pulse; the other input of the memory synchronization comes from the instruction sequencing counter. Often the latter two units are called the "memory synchronization chassis".

4.4 Illiac Input-Output Circuits

The input-output circuitry of Illiac I is given in block-form in Figure 4-7. As mentioned before, the right hand side of A (actually the 4

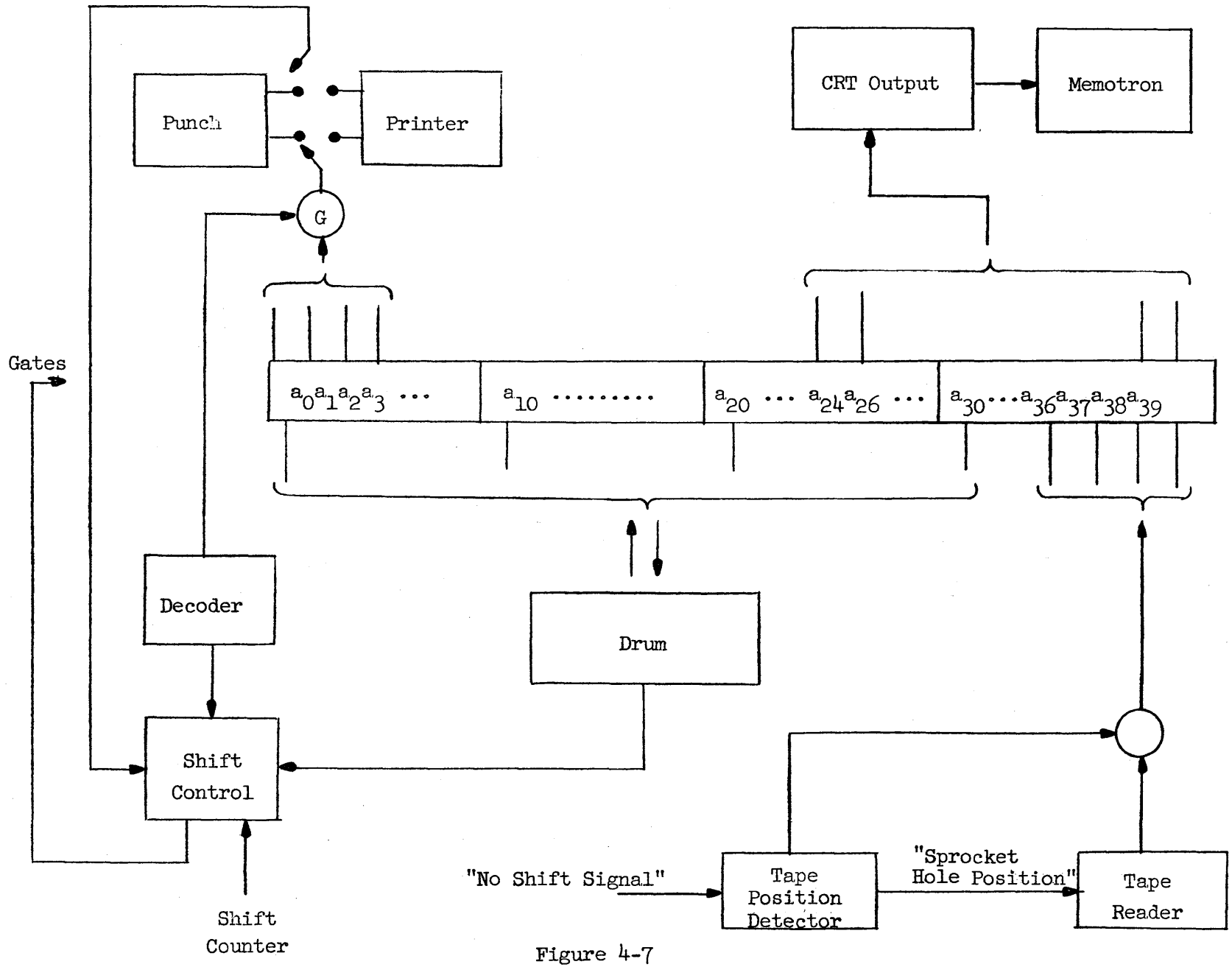


Figure 4-7

Illiac Input-Output Circuitry

rightmost digits) is tied to the tape input equipment i.e. the reader, while the left hand side of A (actually the 4 left-most digits) are tied to the tape output equipment, i.e. the punch or the printer.

When tape is used, digits are read or printed four at a time and after each read or print cycle the shift control (under the action of the shift counter and the decoder) executes four left shifts. As an example let us take the read-in process. The reader uses photo-electric cells which sense holes in paper tape. This tape (see Figure 4-8) has 5 hole-positions plus a row of sprocket holes. The first four holes correspond to binary digits having the weights 2^0, 2^1, 2^2 and 2^3 respectively. This means that one column, by different combinations, can represent any one of the values 0 to 15. In order to simplify the translation from holes to numbers, it is advantageous to use the sexadecimal system with numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, K, S, N, J, F and L. The table below gives the equivalences. For technical reasons K and S are printed + and - respectively.

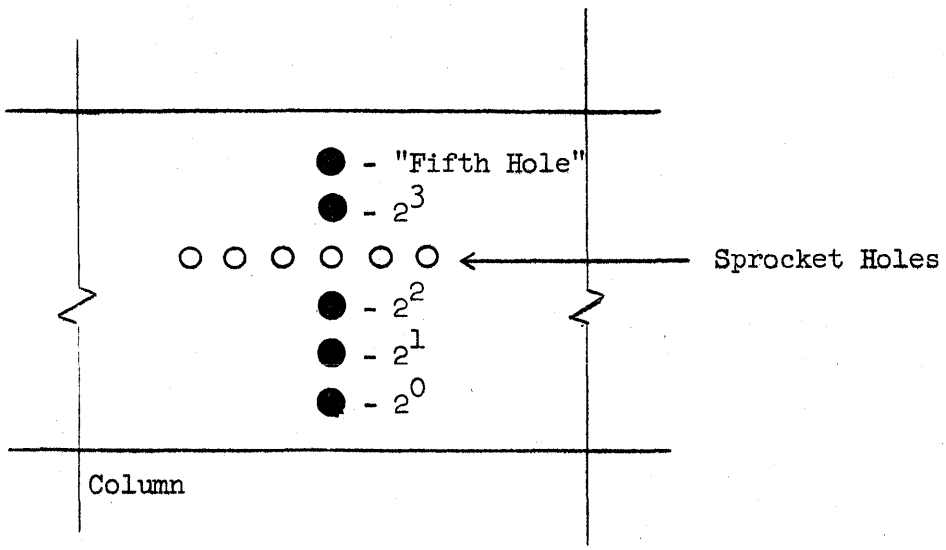


Figure 4-8

Punched Paper Tape

Table for the Sexadecimal System

				o											1		
					o										2		
				o	o										3		
			o												4		
			o			o									5		
			o	o											6		
			o	o	o										7		
		o													8		
		o				o									9		
		o		o											10	K (+)	
		o		o	o										11	S (-)	
		o	o												12	N	
		o	o			o									13	J	
		o	o	o											14	F	
		o	o	o	o										15	L	

If the fifth hole is punched, the interpretation of the other holes changes: they can represent letters (different from K, S etc.), number shifts, letter shifts, spaces, carriage returns etc. If the computer is to take account of these instructions, the fifth hole information must be stored in A: the sign digit of A is assigned to the fifth hole.

The tape reading process can only be carried out when the tape has come to a complete stop in the reader: a tape position detector senses the sprocket-hole position. When its second input is a "no shift signal" it opens a gate on the path between the reader and A. Notice that in the case of the output equipment the shift control must be synchronized with the mechanical movements in the punch or the printer.

The connections between the drum and the A register were already mentioned in Section 4.1. Since the drum is a synchronous device, switching operations (shift control!) and the times of transfer are controlled by a "timing track" on the drum.

A cathode-ray-tube output with a 256 x 256 raster (set up through an address generator connected to a_{32} ... a_{39} for horizontal positioning and a_{24} ... a_{31} for vertical positioning) allows the display of curves etc. To this purpose the computer is programmed to light up specific points of the raster. A memotron, connected to the same address generator, gives a persistent image of the CRT curves.

4.5 The Relative Size of Memories

Problems Involving Large Amounts of Storage

By definition data processing involves simple computations on a large number of words (non-cumulative errors) while scientific calculations involve complicated computations on a small number of words (cumulative errors). Modern scientific calculations tend towards data processing because complicated tabulations and searches are involved while the fundamental iterative loop is simple.

Let us estimate the storage space required for some typical scientific problems.

Hyperbolic Differential Equations. These often occur in hydrodynamic problems in m dimensions. Usually the values of $2m + 1$ variables x_i (i.e. $q_1 \ldots q_m$, $p_1 \ldots p_m$, S) at time $t + \Delta t$ are deduced from their values at time t in a region with $\Delta x_i > c\Delta t$. The number M of mesh points per x_i ($10 \leq M \leq 100$ usually) determines the amount of data to be stored at any one time: approximately M^{2m+1} words are necessary.

Interactions Between Atoms in a Lattice. Here it is customary to employ a tabulation method in which all atoms (M^3 where M is the number in each dimension, again $10 \leq M \leq 100$) are listed and successive approximations change the values of m parameters of interest. Visibly mM^3 words have to be stored.

Solution of Nonlinear Network Problems. It is easy to see that in a network defined by N voltage vs current curves of n points each and formed of K nodes with an average of B branches coming into each node, the number of words to be stored is 2Nn + 2KB. When this nonlinear network solver is a subroutine of another (e.g. optimizing -) program, the number cited can be multiplied by a factor between 2 and 3.

Solution of Systems of Nonlinear Algebraic Equations. Solving n equations in m variables by minimizing a function formed with them and using the method of steepest descents, at least knm words of storage are required and $2 \leq k \leq 10$.

Average Computation Time as a Function of Access Time

The average computation time t can be calculated for certain types of problems by assuming that on the average each multiplication (time t_m) is associated with A non-multiplicative instructions (time t_a each). Let a_i and a_0 be the access times of the random access memory for instructions and operands respectively. Then

$$t = \frac{1}{A+1} [t_m + At_a + (A + 1) (a_0 + a_i)]. \qquad (4\text{-}1)$$

This shows that speeding up multiplication or addition without decreasing a_0 and a_i is inefficient. Unhappily $a_0 = 2a_i \sim 1.5$ μs in the latest core memories. The table shows t for Illiac I and Illiac II for A = 10. Visibly a 200 fold decrease of t_m and t_a produces only a 40 fold decrease in t.

	t_m	t_a	$a_i + a_0$	t	
Illiac I	600	40	27	118	in μs
Illiac II	3	.3	2.25	3	

Paralleling of Operations. Buffer Memories

In the calculations above it was assumed that all operations are done sequentially: if n operations with individual times T_i are required in a process, the time for the process is $\sum_n T_i$. Evidently it is theoretically possible to reduce this time to Max (T_i) when all operations are paralleled. In practice it must be realized that the ideal time, Max (T_i), cannot be attained because of supplementary control times necessitated by the more complex nature of the control problem.

The bottleneck described is alleviated in Illiac II by using a very fast access memory directly connected to the arithmetic unit and called the "buffer". This serves a double purpose: 1) the factor A is reduced since short loops do not have to go outside the "buffer"; and 2) paralleling of information transfer into the "buffer" and actual calculation becomes possible. The usefulness of the 10 word "buffer" in Illiac II is guaranteed by its .2 μs access time. A special design makes the realization in the form of flipflop registers economically feasible.

Optimum Size of the Random Access Memory

For problems of the kind discussed at the beginning the main random access memory must be connected to a back-up memory (drum, tapes), since a 20-30,000 word core memory is too expensive. The cost of such a back-up memory - per bit stored - is reduced by serializing information transfer: blocks are transferred with an access time α per block (random address) and then a time β per word read serially within the block. Always $\alpha \gg \beta$.

The question as to the relative sizes of these memories can only be partially answered, even when the type of problem to be solved is known. The back-up store can be assumed to be of a size sufficient to contain all the information encountered in a problem; it is sometimes useful to assume that it is actually part of the input-output equipment. We shall calculate the number of words (n) in the random access memory for which any further increase in n no longer corresponds to an appreciable decrease of the total calculation time.

Consider problems in which N initial words are reduced to λN words ($\lambda \sim .5$ or less) and an average calculation time t is required per operation. If the random access memory holds all N words, the total calculation time is evidently $T_N = Nt$. If, however, only n words are available in the random access memory, to and fro transfers are necessary. It is easily seen that

$$T_n = T_N + \frac{2\alpha N}{n} + \beta N(2-\lambda). \qquad (4\text{-}2)$$

Suppose that we do not want to increase the calculation time by more than a factor of five by choosing $n < N$. Then

$$5 = \frac{T_n}{T_N} = 1 + \frac{2\alpha N}{nNt} + \frac{\beta N(2-\lambda)}{Nt}$$

For $t = 3\ \mu s$, $\lambda = .5$, $\alpha = 18000\ \mu s$, $\beta = 6\ \mu s$ we then have $n = 12000$. Note that T_n/T_N has a lower bound of $1 + \beta(2-\lambda)/t$: This means that is is very time consuming to transfer information as soon as the calculation time is of the order of the word-time β.

General Design Criterion. Reliability

For the memory - as for any other equipment involving a great number of similar elements (transistors, cores, etc.) - the relationship between the decrease in reliability with increasing complexity and the decrease in total calculation time can be established. Idealizing by assuming that there are N elements with an average life of T hours (meaning that the machine breaks down every T/N hours on the average) and that the computation time lost per breakdown is L(N) hours, a problem needing Θ_N hours of faultless time with this N-element machine will in reality necessitate

$$\Theta'_N = \Theta_N \left[1 + \frac{NL(N)}{T}\right] \text{ hours.}$$

For $M > N$ elements

$$\Theta'_M = \Theta_M \left[1 + \frac{ML(M)}{T}\right] \text{ hours.}$$

But obviously M/N computers of the N-element type can solve the problem in $N\Theta'_N/M$ hours; the increase in hardware is therefore only justified if

$$\Theta_M \left[1 + \frac{ML(M)}{T}\right] < \frac{N\Theta_N}{M} \left[1 + \frac{NL(N)}{T}\right]. \qquad (4\text{-}3)$$

The best possible design therefore minimizes

$$N\Theta_N \left[1 + \frac{NL(N)}{T}\right].$$

Neglecting $N\frac{dL}{dN}$ with respect to L (i.e. repair time $\sim$ indep. of N!), this means that

$$\frac{d\Theta_N}{\Theta_N} = - \left[1 + \frac{NL(N)}{T}\right] \frac{dN}{N} .$$

It seems not unreasonable to put N = 20,000, T = 60,000 hours and $L(N) \sim .5$ hours. Then the optimum memory has the property that a 1% increase in N produces a 1.2% increase in speed.

4.6 Addition, Subtraction, Multiplication and Division in Illiac I

Addition and Subtraction (Order Type L)

The only difference between addition and subtraction is the setting of the complementing circuit:

augend x + addend y = sum z

minuend x - subtrahend y = difference z

During the execution of an add or subtract instruction y is transferred from a specified memory location to R^3 and then to the adder via the complementing circuit. The augend or minuend lies in A and forms the second adder input. When sufficient time has elasped for the sum or difference z to be formed, z is transferred to $\overline{A}$ and by a straight-down shift to A.

Two important variants are used in addition and subtraction: "hold" add and "hold" subtract leave the result of the previous operation in A (x unchanged), while "clear" add and "clear" subtract sets the augend or minuend to zero initially (x = 0). The latter variant is therefore used to bring a number or its negative from the memory to A.

The Illiac orders which interest us in this category are

L 0 n:	(A) - (n)	A
L 1 n:	- (n)	A
L 4 n:	(A) + (n)	A
L 5 n:	(n)	A

where n is a memory location and (n) and (A) means contents of n or A: (A) = x, (n) = y.

Absolute Value Addition and Subtraction. Increment Add Orders and Add From Q Orders (Order Types L, F, S and K)

The absolute value of a number to be subtracted or added can be formed by sensing its sign digit and reversing the setting of the complementing circuits with respect to those discussed in the last section if the sign digit is a one.

Since a one can be added to the least significant digit of A in order to form the two's complement after forming the one's complement of each digit, this facility can be used to create orders in which the relationship between the setting of the complement gate and the insertion of the least significant digit is reversed: this means that $[(n) + 2^{-39}]$ is involved instead of (n).

We have seen that the contents of Q can be transferred to R^3: This allows us to add or subtract (Q) from (A). All variants (absolute value, increment) are available.

The Illiac orders in this category are:

L 2 n	$(A) - \lvert(n)\rvert \rightarrow A$
L 3 n	$- \lvert(n)\rvert \rightarrow A$
L 6 n	$(A) + \lvert(n)\rvert \rightarrow A$
L 7 n	$\lvert(n)\rvert \rightarrow A$
F 0 n	$(A) - (n) - 2^{-39} \rightarrow A$
F 1 n	$- (n) - 2^{-39} \rightarrow A$
F 4 n	$(A) + (n) + 2^{-39} \rightarrow A$
F 5 n	$(n) + 2^{-39} \rightarrow A$
K 0	$(A) - (Q) - 2^{-39} \rightarrow A$
K 1	$- (Q) - 2^{-39} \rightarrow A$
K 4	$(A) + (Q) + 2^{-39} \rightarrow A$
K 5	$(Q) + 2^{-39} \rightarrow A$
S 0	$(A) - (Q) \rightarrow A$
S 1	$- (Q) \rightarrow A$
S 2	$(A) - \lvert(Q)\rvert \rightarrow A$
S 3	$- \lvert(Q)\rvert \rightarrow A$
S 4	$(A) + (Q) \rightarrow A$
S 5	$(Q) \rightarrow A$
S 6	$(A) + \lvert(Q)\rvert \rightarrow A$
S 7	$\lvert(Q)\rvert \rightarrow A$

Multiplication (Order Type 7)

Initially the multiplier y lies in Q (name!) while the multiplicand x is transferred from the specified memory location to R^3 where it remains throughout the multiplication. Multiplication then is a series of additions and right shifts: at each step a partial product is held in A. A multiplier digit in the least significant position q_{39} of Q is sensed. If this digit is

1, the sum of the partial product p_i in A and x (in R^3) is transferred to $\bar{A}$; if this digit is 0, the partial product in A is transferred to $\bar{A}$. In either case, a right shift from $\bar{A}$ to A follows. Simultaneously, a right shift occurs in Q, bringing the next multiplier digit into the least significant place. Notice that a double-length product is formed.

The partial product p_i as well as x are (as all numbers in the computer) in the range -1, +1 (+1 being excluded). The sum in $\bar{A}$ is therefore in the range -2, +2, meaning that the sign digit in $\bar{A}$ is not a true indication of the sign of $p_i + x$: in transferring $p_i + x$ to A with a right shift, the range is reduced to its allowed value: $-1 < 1/2\ (p_i + x) < +1$, but we have to insert the proper sign digit. It is easily seen that the sign of $1/2\ (p_i + x)$ should be that of p_i and x when their signs are equal, or equal to that obtained in $\bar{A}$ for $p_i + x$ if their signs are different.

Let us now consider the 39th partial product formed as described above. The recursion relationship for partial products is

$$p_{i+1} = 1/2\ [p_i + y_{39-i}\ x] \tag{4-4}$$

which shows that

$$p_{39} = 2^{-39}\ p_0 + x \sum_{i=1}^{39} y_i\ 2^{-i} = 2^{-39}\ p_0 + (y + y_0)\,x \tag{4-5}$$

p_0 being the initial contents of A. We see that in case of a negative multiplier ($y_0 = 1$), p_{39} contains a "false term" $y_0\ x = x$: in this case Illiac automatically subtracts the multiplicand x and sets $q_0 = 0$.

According to the value of p_0 we distinguish three types of multiplication:

74 n (n) (Q) + $2^{-39}\ p_0 \longrightarrow$AQ : "hold" multiply

75 n (n) (Q) $\longrightarrow$AQ : "clear" multiply

7J n (n) (Q) + 2^{-40} $\longrightarrow$AQ : "round-off" multiply

Division (Order Type 6)

As mentioned in Section 4.1, registers A and Q can be combined to form a double length register AQ (or $\overline{AQ}$) such that the sign digit q_0 of Q is left out in the shifting process. The contents (AQ) are therefore $a_0 \cdots a_{39}\ q_1 \cdots q_{39}$. In division we start out with a double length dividend* called r_0 having a sign digit p_0 ($p_0 = a_0$; $a_1 \cdots a_{39}\ q_1 \cdots q_{39}$ therefore represent $r_0 + p_0$!); this divident lies in AQ. The divisor y (with a sign digit y_0) is transferred to R^3 and it is supposed that $|r_0| < |y| < 1$, i.e. we will have for the quotient q, $|q| < 1$.

For $p_0 = y_0 = 0$, i.e. positive dividend and divisor, the division process in Illiac I is analogous to long division: the divisor is subtracted from a partial remainder r_n (with sign digit p_n) in A and the sign of the difference (in the adder) is sensed. If the sign is negative, 0 is inserted in q_{39} as quotient digit and AQ is shifted left (doubled) to form a new partial remainder. If the sign is positive, 1 is inserted in q_{39} and the difference in the adder is placed in A; again AQ is shifted left. At each left shift, q_1 is shifted into a_{39} (as usual) but also into q_0: this is to give to the contents of Q the right sign after 39 quotient digits have been created.

In order to understand the division process more fully, especially in the case of negative dividends and divisors, we shall formulate the rules to be obeyed by the computer at each step.

1. At the beginning, the sign y_0 of the divisor is compared to the sign p_0 of the dividend. If they agree, the complementing circuit is set to subtract throughout the division; if they disagree, the complementing circuit is set to add. The setting is thus given by

---

* A single length divident means that the non-sign digits of Q are either left over from a preceding calculation or that they have been set to zero initially.

$$(-1)^{p_0+y_0} = \sigma \text{ (say)}. \tag{4-6}$$

2. A tentative partial remainder s_n (with sign digit t_n) is obtained by forming

$$s_n = r_n - \sigma y. \tag{4-7}$$

3. If the sign t_n of s_n agrees with the sign p_0 of the dividend, the tentative partial remainder is transferred from the adder to $\overline{A}$. If they disagree, the partial remainder in A is transferred to $\overline{A}$. (This choice between a tentative partial remainder and the old partial remainder is a special feature of Illiac, made possible by the fact that the partial remainder in A is <u>not</u> destroyed when the tentative partial remainder is formed). The new partial remainder - after a left shift - is then given by

$$r_{n+1} = 2r_n - \sigma [1 + (-1)^{p_0+t_n}] y \tag{4-8}$$

4. If the sign t_n of the tentative partial remainder agrees with the sign y_0 of the divisor, 1 is inserted in $\overline{q}_{39}$ (39th position of $\overline{Q}$). If they disagree, 0 is inserted. Call the quotient digit thus obtained q_n. Then

$$q_n = 1/2 \, [1 + (-1)^{y_0+t_n}] \tag{4-9}$$

5. $(\overline{A})$ and $(\overline{Q})$ are transferred to A and Q with a left shift.

6. At the end q_{39} is set to 1.

By using (4-8), we find that

$$2^{-39} r_{39} = r_0 - \sigma y \, [(1-2^{-39}) + 2^{-1} (-1)^{p_0} \sum_{0}^{38} (-1)^{t_n} 2^{-n}] \tag{4-10}$$

(4-9) gives the non-sign part of the quotient q (using rule 6)

$$\sum_{1}^{38} q_n 2^{-n} + 2^{-39}$$

i.e. the arithmetical value of the quotient is

$$q = -q_0 + \sum_{1}^{38} q_n 2^{-n} + 2^{-39} = -2q_0 + \sum_{0}^{38} q_n 2^{-n} + 2^{-39} \qquad (4\text{-}11)$$

Now we define a remainder r by

$$r = 2^{39} [r_0 - qy] \qquad (4\text{-}12)$$

In order to show that q is the quotient, we must show that r is of order unity. Using (4-11) and (4-12)

$$r = (r_0 - qy)\, 2^{39}$$

$$= 2^{39} r_0 - 2^{39} y \left[-2q_0 + 2^{-1}(-1)^{y_0} \sum_{0}^{38} (-1)^{t_n} 2^{-n} + 1\right]$$

since $$2^{-1} \sum_{0}^{38} 2^{-n} + 2^{-39} = 1$$

But $$r_{39} = 2^{39} r_0 - 2^{39} \sigma y \left[(1 - 2^{-39}) + 2^{-1} (-1)^{p_0} \sum_{0}^{38} (-1)^{t_n} 2^{-n}\right]$$

and $$q_0 = 2^{-1} \left[1 + (-1)^{y_0 + t_0}\right]$$

$$= 2^{-1} \left[1 - (-1)^{y_0 + p_0}\right] \quad \text{since } p_0 \neq t_0 \text{ by our hypothesis}$$

$|r_0| < |y| < 1$.

$$= 2^{-1} [1 - \sigma] \quad .$$

Therefore

$$(r - r_{39})\, 2^{-39} = y\,[(1-2^{-39})\,\sigma + 1 - \sigma - 1 + \left\{\sigma(-1)^{p_0} - (-1)^{y_0}\right\} \sum_{0}^{38} (-1)^{t_n} 2^{-n}]$$

$$= y\,[-2^{39}\sigma - \{\}\,\Sigma]$$

Now

$$\{\} = (-1)^{y_0}\,[(-1)^{2p_0} - 1] = 0 \qquad \text{since by definition } \sigma = (-1)^{p_0+y_0}$$

and it follows that

$$r = r_{39} - \sigma\, y \tag{4-13}$$

Since r_{39} and y both have absolute values less than one and since $\sigma = \pm 1$, the absolute value of r cannot exceed 2, meaning that

$$|r_0 - qy| \leq 2^{-39} \cdot 2 = 2^{-38}$$

(Actually it can be shown by a more detailed examination that $|r_0 - qy| \leq 2^{-39}$!)

The Illiac division order is written 66n and as we have seen its effect is to put

$$\frac{(AQ)}{(n)} \longrightarrow Q$$

4.7 Other Methods of Multiplication and Division

The IAS Method of Multiplication

This method, used in the Princeton machine, multiplies only the non-sign digits of the multiplier y and the multiplicand x: suitable corrections have to be made. Let x_0 and x_1 be the sign and non-sign digits of x: $x = -x_0 + x_1$.

In the same way $y = -y_0 + y_1$ and

$$x_1 y_1 = (x + x_0)(y + y_0) = xy + x_0 y + y_0 (x + x_0). \qquad (4\text{-}14)$$

This shows that a correction is required in two cases:

1. If $y_0 = 1$, $x + x_0$ must be subtracted at the end.
2. If $x_0 = 1$, y must be subtracted at the end.

The latter operation is quite difficult, since the digits of the multiplier are destroyed in the multiplication process. In order to circumvent this difficulty, a piecewise insertion of the digitwise complement of the non-sign digits of y is used; at the end $-1 + 2^{-39}$ is added to the final product, in order to obtain the two's complement. The following rules describe the operation:

1. At each step if $q_{39} = 0$ add x_0 to the partial product p_i and if $q_{39} = 1$ add the non-sign digits of the multiplicand, i.e. $(x + x_0)$ to p_i. In case x_0 is 1, add the digitwise complement of y_{39-i} i.e. $(1-y_{39-i})$. Transfer the results to $\overline{A}$ and the non-sign digits of Q to $\overline{Q}$.
2. Shift right from $\overline{A}$ to A and from $\overline{Q}$ to Q, inserting 0 as sign digit in A.
3. If the multiplier was negative, subtract x at the end and if the multiplicand was negative, add $-1 + 2^{-39}$.

Formulating these rules mathematically

$$p_{i+1} = 1/2 \, [p_i + y_{39-i} (x + x_0) + (1 - y_{39-i}) x_0]$$

$$= 1/2 \, [p_i + y_{39-i} x + x_0].$$

It follows that

$$p_{39} = 2^{-39}\, p_0 + (y + y_0)\, x + (1 - 2^{-39})\, x_0$$

Correcting by rule 3, we find that the final product p is given by

$$p = p_{39} - y_0 x - (1 - 2^{-39})\, x_0 = xy + 2^{-39}\, p_0$$

It is easily shown by induction that all the partial products p_i lie in the range 0, 1.

Non-Restoring Division

Decimal desk calculators use two different systems. The first category imitates long division by subtracting the divisor from the partial remainder (assuming that both are positive) until a negative number is obtained; then the divisor is added once: this "restoring" of the divisor gives the process its name.

In non-restoring division we subtract the divisor until the sign changes, or until nine subtractions have occurred. The partial remainder (negative this time) is shifted and the divisor added until the sign again changes, or until nine additions have occurred. In the first case put down a positive quotient digit equal to the number of additions. For the decimal system the possible quotient digits are thus -9, -8, ..., -1, +1, +2, ..., +9: no zero is required. It is easy to see that we can convert such a quotient into one using digits 0 through 9.

Let us examine the non-restoring division scheme more closely in the binary system: only two quotient digits, -1 and +1, can be created as at most one addition or subtraction is required at each step. Suppose that we have formed such a "+1, -1 - quotient" and that we wish to find the normal "+1, 0 - quotient", i.e. given

$$x = \sum_{1}^{39} b_i\, 2^{-i} \quad \text{with } b_i = +1,\ -1 \tag{4-15}$$

find x_0 and x_i (having values 0 and 1) such that

$$x = -x_0 + \sum_{1}^{39} x_i \, 2^{-i} \tag{4-16}$$

Introduce

$$a_{i-1} = \frac{b_i + 1}{2}, \text{ then } a_{i-1} = 0, 1 \tag{4-17}$$

then

$$x = \sum_{1}^{39} a_{i-1} \, 2^{-(i-1)} - \sum_{1}^{39} 2^{-i}$$

$$= (a_0 - 1) + \sum_{1}^{38} a_i \, 2^{-i} + 2^{-39}$$

i.e.

$$x_0 = 1 - a_0$$

$$x_i = a_i \qquad i = 1, \ldots, 38 \tag{4-18}$$

$$x_{39} = 1$$

We therefore have the conversion rule: replace -1's by zeros, shift left, insert 1 in the least significant digit and complement the sign digit (obtained after the shift).

Using the same notation as in Section 4.6, we can now discuss non-restoring binary division. In this system a part of the +1, -1 → 0, 1 conversion is made at each step. The initial conditions and the conditions on the absolute value of dividend and divisor are the same as before. As a preliminary step the divident r_0 is transferred to $\overline{AQ}$. Then the rules are

1. Transfer ($\bar{A}$) and ($\bar{Q}$) to A and Q with a left shift, $\bar{q}_1$ being transferred to a_{39}.
2. Sense the sign y_0 of the divisor y in R^3 and the sign p_n of the partial remainder in $\bar{A}$. If these signs agree, subtract y from $2r_n$ and insert 1 in q_{39}. If these signs disagree, add y to $2r_n$ and insert 0 to q_{39}.
3. In either case transfer the difference or the sum to $\bar{A}$.
4. After 39 steps transfer the remainder from $\bar{A}$ to A (without shift), the quotient from $\bar{Q}$ to Q, convert to complementary form (-1→ 0 and left shift are done already) by complementing q_0 and inserting 1 in q_{39}.

Mathematically these rules correspond to

$$r_{n+1} = 2r_n = (-1)^{p_n+y_0}\, y \tag{4-19}$$

$$q_n = 2^{-1}\,[1 + z_{n+1}], \text{ where } z_n = (-1)^{p_{n-1}+y_0} \tag{4-20}$$

The arithmetic value of the quotient q is then

$$q = \sum_{1}^{39} z_n\, 2^{-n} = \sum_{1}^{39} (-1)^{p_{n-1}+y_0}\, 2^{-n}$$

Defining the remainder r as before by

$$r = 2^{39}\,(r_o - qy)$$

we find that

$$r = r_{39} \tag{4-21}$$

where

$$2^{-39} r_{39} = r_0 - y \sum_{1}^{39} (-1)^{p_{n-1}+y_0} 2^{-n} \qquad (4\text{-}22)$$

This shows that q is indeed the quotient, for r_{39} is less than 1 in absolute value.

4.8 Illiac Shift Orders, Transfer Orders, Store Orders and Input-Output Orders

Without going into further details, we give below a list of common orders used in Illiac as an illustration of the facilities which it offers and as a preparation for the next section. All of these orders are of the 20-digit variety, except for the 40-digit drum orders.

20-Digit Orders

00 n	Shift AQ left n places
0F	Stop
10 n	Shift AQ right n places
20 n	Stop. After (manual) restarting go to r.h. order of location n
22 n	Transfer control to r.h. order of location n
24 n	Stop. After (manual) restarting go to l.h. order of location n
26 n	Transfer control to l.h. order of location n
30 n 32 n	If (A) $\geq$ 0 do as in corresp. 2 ... order
34 n 36 n	If (A) < 0 go on to next order
40 n	Store (A) in n, leaving A unchanged
41 n	Store (A) in n, clearing A beforehand
42 n	Store digits → to address of r.h. order from A
46 n	Store digits → to address of l.h. order from A
50 n	Put (n) in Q
80 n	Input n/4 sexadecimal characters from tape
81 n	Clear A and then proceed as in 80
82 n	Punch n/4 sexadecimal characters on tape

40-Digit (Drum) Orders

85 11 TV n	For T ≠ 0, 1, 8, 9 transfer drum location (n) to A and then execute the order TV n. For T = 0, 1, 8, 9 do as before, but skip TV n.
86 11 TV n	For T ≠ 0, 1, 8, 9 transfer (A) to drum location n and then execute the order TV n. For T = 0, 1, 8, 9 do as before, but skip TV n.

Note that out of addresses 0-12799 on the drum the first 2600 are "locked out" and contain often used routines.

Example of a Complete Program

As an example of how a complete program is put inside the computer and how coding tricks permit to shorten codes quite considerably, we are going to consider a program which fills the memory full of K4 orders. Looking at the orders given in Section 4.6, we see that this makes the computer count indefinitely: after having read out all order pairs up to 1023, the control counter goes back to zero and another cycle begins.

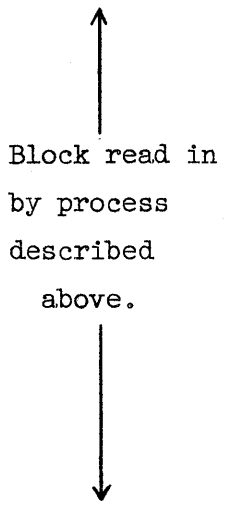
Before discussing the program, it should be mentioned that addresses on the input tape must be written in the sexadecimal system. There is a conversion routine, called SADOI (symbolic address decimal order input), which allows the programmer to use the decimal system for addresses, but we shall not assume its use here.

The program starts as follows:

↑ Miniature	A	(80 028	40 000)	Set by pushbutton, (or by input routine) does not advance control counter.
Bootstrap	B	80 028	40 001	Read in and store
	C	80 028	40 002	Read in and store
↓	D	26 000	(00 000)*	Go back to order pair stored in 0
Block to be read into memory	E	81 004	42 000	Read in one character and modify address
↓	F 3	L5 00K	40 00S	
	G 4	.....	.....	

* "waste order"

A places B in 0. Now B is obeyed, placing C in 1 - which then puts D in 2. Next D is followed; we go back and obey B: this places E in 1 (overwriting C) and this is obeyed next. E clears the accumulator, reads into the accumulator one character of F (i.e. 3) and then modifies the address of the right hand order in 0. Location 0 now reads 80 028 40 003. Going to location 2 we are thrown back on this modified order: the rest of F is read in and stored in 3. Location 1 again modifies location 0 to read 80 028 40 004 (the address being given by the character at the beginning of G). This then reads in the rest of G. This process continues until the r.h. address in 0 has gone up to K. The order after K is preceded by 1, this then modifies location 0 to read again 80 028 40 001: this is obeyed after the transfer of control and places 26 003 00 000 in 1. Now the contents of locations 0 through K look as follows:



0	80 028	40 001	
1	26 003	00 000	Transfer control to location 3
2	26 000	(00 000)	
3	L5 00K	40 00S	(N,J...) Store K4, K4 in location S
4	F5 003	40 003	
5	L5 002	36 003	
6	40 003	40 004	
7	40 005	40 006	
8	40 007	40 008	
9	40 009	25 000	
K	K4 000	K4 000	
S	26 003	00 000	

The next order pair comes from location 1 (which was just overwritten) and transfers control to location 3. This brings K4 000 K4 000 into the accumulator and stores it in location S. Order pair 4 brings L5 00k 40 00S down into the accumulator, adds 1 (i.e. adds 1 to the r.h. address, making it N) and stores it back in 3. So 3 reads successively ... 40 00S, ... 40 00N, ... 40 00J etc. Now order pair 5 brings down 26 000 00 000 and tests if this is positive or zero. Written in binary this pair starts with 0010 ... and is therefore positive: the 36 order transfers control to the

(modified) location 3. Now we store K4 000 K4 000 in N. This process cycles again until all locations between 8 and 3LL (→ 1023) have been filled up with K4 000 K4 000 order pairs.

Now we make use of the fact that after address 3LL the next address 3LL + 1 has the same result as address 0: addresses are interpreted mod 1024 This means that locations 0, 1, 2 will receive the "standard" order pair. Going through locations 3, 4, 5 the cycle is now modified: the address in 3 is stepped up again (order pair L5 00K 40 003) but order pair 5 brings this time K4 000 K4 000 into the accumulator, for this has overwritten the original 26 000 00 000. Upon testing, this reveals itself as negative (K4 ... in binary starts 1010 ...) and this time we go on to obey location 6, 7, 8, ... This overwrites 3, 4 and then the order pairs which have just been obeyed with the fixed contents of the accumulator, i.e. with K4 000 K4 000.

The final step occurs when location 9 is obeyed; 40 009 25 000 is placed in R_3. First the left band order is followed: this places the standard K4 000 K4 000 in location 9, thus drawing the whole program out of the memory and leaving all locations with the same contents! The 25 000 order is a "black switch order": the machine stops and clears A. After being restarted with the "black switch", it goes to location 000 and starts with the left hand order. This initiates the counting process.

References for Math. and EE 294

(Articles excluded, in chronological order)

1. A. W. Burks, H. H. Goldstine and J. von Neumann: "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument". Institute for Advanced Study. (1947)

2. D. R. Hartree: "Calculating Instruments and Machines". The University of Illinois Press. (1949)

3. Staff of Engineering Research Associates: "High-Speed Computing Devices". McGraw-Hill. (1950)

4. W. Keister, A. E. Ritchie and S. H. Washburn: "The Design of Switching Circuits". D. Van Nostrand. (1951)

5. Staff of Harvard Computation Laboratory: "The Synthesis of Electronic Computing and Control Circuits". Harvard University Press. (1951)

6. A. D. Booth and K. H. V. Booth: "Automatic Digital Calculators". Butterworths Scientific Publications. (1953)

7. R. K. Richards: "Arithmetic Operations in Digital Computers". D. Van Nostrand. (1955)

8. R. K. Richards: "Digital Computer Components and Circuits". D. Van Nostrand. (1957)

9. M. Phister: "Logical Design of Digital Computers". John Wiley. (1958)

10. J. T. Culbertson: "Mathematics and Logic for Digital Devices". D. Van Nostrand. (1958)

11. S. H. Caldwell: "Switching Circuits and Logical Design". John Wiley. (1958)

12. W. S. Humphrey: "Switching Circuits". McGraw-Hill. (1958)

13. R. A. Higonnet and R. A. Grea: "Logical Design of Electrical Circuits". McGraw-Hill. (1958)

14. C. V. L. Smith: "Electronic Digital Computers". McGraw-Hill. (1959)

15. F. E. Hohn: "Applied Boolean Algebra". Macmillan (1960)

CHAPTER V

ABSTRACT METHODS

5.1 Groups, Rings, Fields and B. A.

1. SEMIGROUP

Let some common property of the elements a, b, c... define a set $S = \{a, b, c...\}$, also let us define a binary operation * on the members of the set, "binary" meaning here "involving two elements." Now a * b = x may or may not belong to the set S. If for any two elements (a, b) of the set a * b does belong to S, we say that S is closed with respect to the operation *.

Also, if for all a, b, c $\in$ S ($\in$ meaning: belonging to)

$$(a * b) * c = a * (b * c), \tag{5-0}$$

the operation * is called associative in S.

Definition: A semigroup is an associative, closed set S with respect to an operation *.

Remark: A set can be:

discrete finite : {+1, -1}
discrete infinite : {all integers}
continuous : {all numbers}

2. GROUP

Definition: A group G is a semigroup with a unit and inverses where the unit and the inverse are defined as follows:

The unit e satisfies $a * e = e * a = a$ (5-1)

The inverse a^{-1} to a satisfies $a^{-1} * a = a * a^{-1} = e$ (5-2)

Theorem 1. The unit is unique.

Proof: Let e_1, e_2 both be units

then

$$e_1 * e_2 = e_1 \qquad \text{also} \quad e_1 * e_2 = e_2$$

hence

$$e_1 = e_2 \quad \text{QED.}$$

Theorem 2. The inverse is unique.

Proof: Assume that there exist inverses a^{-1}, b.

Then by definition $a * b = e$

$$a * a^{-1} = e$$

$$a * b = a * a^{-1}$$

i.e., operating on the left by a^{-1} we have

$$(a^{-1} * a) * b = (a^{-1} * a) * a^{-1}$$

hence $e * b = e * a^{-1}$

or $b = a^{-1}$

Definition: A group is said to be commutative or Abelian if $a * b = b * a$. The order of a group is the number of elements in it.

3. RING

A ring R is a set which is a commutative group with respect to one binary operation (say +) and a semigroup with respect to a second binary operation (say .). Also, the following distribution relations hold for all a, b, c $\subset$R

$$a \cdot (b + c) = ab + ac \qquad \text{(where ab means } a \cdot b, \text{ etc.)}$$

and (5-3)

$$(a + b) \cdot c = ac + bc$$

Definition: We shall call z the unit of the operation +, and we shall call a^1 the inverse of a with respect to z:

$$a^1 + a = a + a^1 = z$$

$$a + z = z + a = a \qquad (5\text{-}4)$$

Example: {all integers} forms a ring.

Special Rings:

(1) Ring with a unit (unit e for the second operation).

(2) Commutative ring: ab = ba for the second operation.

4. FIELD

A field F is a ring with a unit e and inverses a^{-1} for the second binary operation (called $\cdot$ above). The existence of a unit and an inverse with respect to + is guaranteed by the fact that the field is a ring. We could say that a field is a "double group."

Example: {real numbers}

5. BOOLEAN RING

A Boolean ring BR is a ring with a unit (w.r.t. $\cdot$) in which the idempotency law holds:

$$\text{for every } a \subset BR \qquad a \cdot a = a \tag{5-5}$$

Theorem 3. $a + a = z$ in a BR. (5-6)

Proof: $(a + b) \cdot (a + b) = aa + ba + ab + bb$

LHS = $(a + b)$ by the idempot. law

RHS = $a + ba + ab + b$ by the same law

$\therefore a + b = a + ba + ab + b$

but

$$a^1 + a = z, \quad b^1 + b = z$$

therefore

$$a^1 + b^1 + (a + b) = a^1 + b^1 + a + ba + ab + b$$

or

$$z = ba + ab$$

Now let a = b, then

$$z = aa + aa,$$

or by the idempot. law

$$z = a + a \qquad \text{QED.}$$

Theorem 4. $a^1 = a$ in a BR. (5-7)

Proof: $z = a + a$ and $z = a^1 + a$

Thus

$$a^1 = a^1 + z = a^1 + a + a = a \qquad \text{QED.}$$

Theorem 5. $ab = ba$ in a BR (5-8)

Proof: By theorem 4 we have $(ab)^1 = ab$. Also from the proof of Theorem 3

$z = ab + ba$ or $(ab)^1 = ba$ (unique inverse!)

$\therefore ab = (ab)^1 = ba$ QED.

Theorem 6. $az = z$ in a BR. (5-9)

Proof: $az = a(a + a) = aa + aa = a + a = z$ QED

6. BOOLEAN FIELD

The elements z and e of a Boolean ring form a **Boolean Field BF.**

Theorem 7. A Boolean field has only two elements.

Proof: Let $a \subset BF$, $a \neq z$.

Then

$$a = ae = a(aa^{-1}) = (aa)a^{-1} = (a)a^{-1} = e$$

Thus a BF can have only two elements: z, e.

The distinguishing properties of Rings, Fields, etc., are summed up in the following table:

Table 5-1

Name	e	a^{-1}	$aa = a$	$a^1 = a$	$ab = ba$	$az = z$	$a = (\frac{e}{z})$
Ring							
Field	x	x					
Boolean Ring	x		x	x	x	x	
Boolean Field	x	x	x	x	x	x	x
	postulates			theorems			

7. BOOLEAN ALGEBRA

Two more operations are introduced, called

$$\left.\begin{array}{ll} \text{Complementation:} & \bar{a} = a + e \\ \text{Conjunction:} & a \vee b = a + b + ab \end{array}\right\} \quad \text{and} \tag{5-10}$$

Remark: It is evident from the definitions that we have

$$a \vee b = b \vee a \qquad \text{(commutativity)} \tag{5-11}$$

and

$$(a \vee b) \vee c = a \vee (b \vee c) \qquad \text{(associativity)} \tag{5-12}$$

Theorem 8. $a \vee a = a$ (5-13)

Proof: $a \vee a = a + a + aa = (a + a) + a = z + a = a$

Theorem 9. $a(b \vee c) = ab \vee ac$ (5-14)

Proof: $a(b \vee c) = a(b + c + bc) = (ab) + (ac) + (ab)(ac) = ab \vee ac$

Theorem 10. $a \vee bc = (a \vee b)(a \vee c)$ (5-15)

Proof:
$$\begin{aligned} \text{RHS} &= (a + b + ab)(a + c + ac) \\ &= a + \underline{ac + ac} + \underline{ba} + bc + \underline{abc} + \underline{ab} + \underline{abc} + abc \\ &= a + bc + abc + (ac + ac) + (ab + ba) + (abc + abc) \\ &= (a) + (bc) + (a)(bc) \\ &= a \vee bc = \text{LHS} \end{aligned}$$

QED.

Intersection and Union.

$z \cdot a = z$ (5-9)

as proved above ,

$z \vee a = a$ (5-16)

since $z \vee a = z + a + az = a$,

$e \cdot a = a$ (5-17)

by definition of e ,

$e \vee a = e$ (5-18)

since $e \vee a = e + a + ae = e + a + a = e + z = e$.

Complementation and Duality.

$$a\bar{a} = z \tag{5-19}$$

since $a\bar{a} = a(a + e) = aa + ae = a + a = z$

$$\bar{a} \vee \bar{b} = \bar{a}\bar{b} \tag{5-20}$$

since $\bar{a} \vee \bar{b} = (a + e) + (b + e) + (a + e)(b + e)$

$$= \underline{a} + e + \underline{b} + \underline{e} + ab + \underline{ae} + \underline{eb} + \underline{e}$$

$$= ab + e + (a + a) + (b + b) + (e + e)$$

$$= ab + e = (\overline{ab})$$

also

$$\overline{a \vee b} = \bar{a} \cdot \bar{b} \tag{5-21}$$

since $\overline{a \vee b} = e + a + b + ab = (e + a)(e + b) = \bar{a} \cdot \bar{b}$

We now see that the algebra of a ring with a unit, satisfying idempotemcy and including the operations of complementation and conjunction is formally the same as a Boolean algebra:

$$e \longleftrightarrow 1$$

$$z \longleftrightarrow 0$$

+ is simply the operation $\oplus$ defined in Chapter III. Again it should be emphasized that Boolean algebra is not restricted to values 1 and 0 for the variables as is shown by the following example:

8. EXAMPLE OF A BOOLEAN ALGEBRA OF MORE THAN TWO VARIABLES

Take the most general function $f(x_1, x_2)$ of two (two-valued) Boolean variables. There must be a canonical expansion (see Chapter III) and therefore

$$f = a(\bar{x}_1, \bar{x}_2) \vee b(\bar{x}_1, x_2) \vee c(x_1, \bar{x}_2) \vee c(x_1, x_2)$$

where a, b, c, d are also Boolean variables with the values 0 and 1. Visibly any combination of four zeros and ones corresponds to a different f: There are 16 different functions of two variables.

Now take as the elements of a new, multi-valued algebra the sixteen types of f, setting

$$f(0000) = 0 \quad , \quad f(1111) = 1$$

$$f(0001) = A \ldots . \quad f(1110) = L$$

and call x a variable that can take any one of these 16 values. It is then quite clear that all postulates of Boolean algebra are satisfied, e.g.,

$$x_1 \cdot x_2 = x_2 \cdot x_1 \qquad\qquad x_1 \vee x_2 = x_2 \vee x_1$$

$$\bar{x} \cdot x = 0, \qquad\qquad \bar{x} \vee x = 1,$$

the latter two simply expressing that the minterms in x and those in $\bar{x}$ are mutually exclusive and that the product of any one in $\bar{x}$ with any one in x is zero since they are orthogonal (see Chapter III).

5.2 Cubical Representation of Minterms

5.2.1 GENERALIZED CUBES

In a cartesian coordinate system the vertices of a suitably scaled and rotated 3-dimensional cube (or 3-cube for short) can be represented by the eight possible binary triplets (000), (001), (010), (011), (100), (101), (110) and (111). We shall call the figure in a space of n dimensions whose vertices are represented by all possible multiplets of n binary digits an n-cube and denote it by c^n. A 2-cube is a "square," a 1-cube a "line segment" and a 0-cube simply a "point." A 4-cube is called a "tesseract": It is shown in a (non-unique) projection in Fig. 5-1.

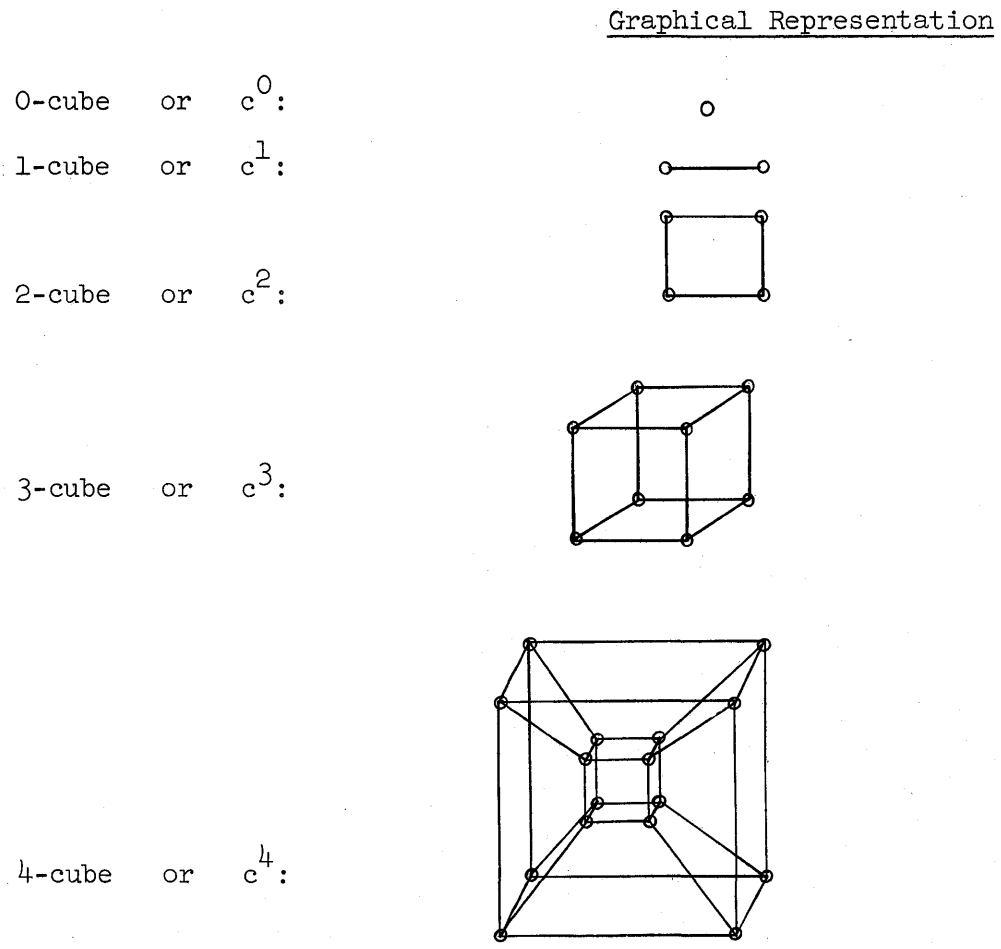


Figure 5-1. Cubes

By definition the vertices (or 0-cubes) of an n-cube correspond to the possible multiplets of the form $(a_1 \ldots a_i \ldots a_n)$ with $a_i = 0$ or 1. This shows that an n-cube has 2^n vertices. Two vertices will be called complementary if they differ in one digit position only. We shall represent the line segment (or 1-cube) joining two complementary vertices by the multiplet representing these vertices with an x in the digit position in which they differ. The line joining (1101) and (1001) is thus (1x01). We shall call two 1-cubes complementary if their multiplets have the x in the same position and coincide in all remaining digits except one. We shall represent the 2-cube joining two complementary 1-cubes by the multiplet representing these 1-cubes with an x in the digit position in which they differ. The generalization of this procedure is evident. Note that in the case of n = 1, 2 or 3 "complementary" has the geometrical significance of "adjacent" as far as vertices are concerned, but means "opposite" when it comes to edges or sides. It is obvious that we can always build up the whole cube by judiciously forming combinations of complementary 0-cubes, then of complementary 1-cubes, etc. This will lead ultimately (in whatever order we synthesize the n-cube) to a multiplet containing x's only: the 1-cube in 1-dimensional space is represented by (x); the 2-cube in 2-dimensional space by (xx); the 3-cube in 3-dimensional space by (xxx).

Suppose now that we work once and for all in a space of a fixed number--n--of dimensions, i.e., that all multiplets are of the form $(a_1 \ldots a_i \ldots a_n)$ with n digits. Then our synthesis of an r-cube from the representation of two complementary (r - 1)-cubes leads to the rule that all r-cubes have exactly r digits a_i equal to x.

Given any r-cube, we shall say that an s-cube with $s < r$ is a subcube of this c^r (or that c^r contains c^s) if its representation can be obtained from that of c^r by particularizing one or more of the x's. If s = r - 1 we shall

call the possible s-cubes faces of the r-cube: this definition shows that a c^r has 2r faces, for any one of the r x's can be given the value 0 and then 1 to obtain r pairs of complementary faces. Similarly a cube c^r containing a c^s is called a supercube of c^s.

5.2.2 SUBSETS OF VERTICES. FACE AND COFACE OPERATORS. CUBICAL COMPLEXES

We shall discuss below the geometry of subsets of vertices of the n-cube. Let f be such a subset. Then one of the possible problems is to group complementary vertices in f into 1-cubes, then complementary 1-cubes in f into 2-cubes, etc. In particular we might be interested in how big the biggest cube (i.e., the cube having most x's) is that uses vertices in f only. Or we might want to construct a set C of cubes (of maximum dimensions) containing all vertices in f: this is the so-called covering problem. In general any set of cubes containing all vertices in f is called a cover of f: we usually want as simple a cover as possible (see below).

The representation of the faces of $c^r = (a_1 \ldots a_i \ldots a_n)$ can be obtained by applying to $(a_1 \ldots a_i \ldots a_n)$ a face operator δ_i^0 or δ_i^1 where δ_i^0 means: replace the i^{th} digit a_i in $(a_1 \ldots a_i \ldots a_n)$ by a 0 if $a_i = x$. If $a_i \neq x$ the operator is zero by definition. Summarizing:

$$\left.\begin{aligned} &\left.\begin{aligned} \delta_i^0(a_1 \ldots a_i \ldots a_n) &= (a_1 \ldots 0 \ldots a_n) \\ \delta_i^1(a_1 \ldots a_i \ldots a_n) &= (a_1 \ldots 1 \ldots a_n) \end{aligned}\right\} \quad \text{if } a_i = x \\ &\delta_i^0(a_1 \ldots a_i \ldots a_n) = \delta_i^1(a_1 \ldots a_i \ldots a_n) = 0 \quad \text{if } a_i \neq x \end{aligned}\right\} \qquad (5\text{-}22)$$

By choosing i equal to the digit position in which the x's occur, we visibly obtain the 2r faces of the c^r.

Given a set f and a certain r-cube $(a_1 \ldots a_i \ldots a_n)$ one of the important questions is: can we find a second r-cube using vertices in f only and

complementary to the first? This is answered by examining the result of the application of the coface operator ϵ_i defined by the property that $c^r = (a_1 \ldots a_i \ldots a_n)$ being a given cube using f-vertices only.

$$\left.\begin{array}{l} \epsilon_i(a_1 \ldots a_i \ldots a_n) = (a_1 \ldots x \ldots a_n) \\ \text{if both } (a_1 \ldots 1 \ldots a_n) \text{ and } (a_1 \ldots 0 \ldots a_n) \text{ use f-vertices only} \end{array}\right\} \quad (5\text{-}23)$$

If the i^{th} digit is already an x, the result is zero by definition.

It is important to note that ϵ_i does not form a supercube of one more dimension: it forms this supercube only if it can do so using f-vertices only. Using coface operators it is now possible to build up all cubes which remain within the bounds of the subset f. Any supercube of a cube c^r which remains within these bounds is called a coface of c^r, All cubes using the f-vertices (which one can obtain by applying ϵ_i on a trial and error basis first to 0-cubes, then to 1-cubes thus formed whenever possible, etc.) form the cubical complex corresponding to f. This complex is denoted by F = K(f), K meaning "form the complex of." F consists possibly of a set of 0-cubes K^0 plus a set of 1-cubes K^1, etc. Obviously K^0 = f and

$$F = K(f) = f \cup K^1 \cup K^2 \cup \ldots \quad (5\text{-}24)$$

If f is given by a cover consisting of the set of cubes (not necessarily minimal) {a,b,c...} it is customary to write F = K{a,b,c...} with the understanding that {} would actually allow us to determine f.

Example: Let f be defined by {(0000), (0001), (0100), (0101), (0110), (1000), (1010), (1110)}. This is also the set K^0. To calculate K^1 we must apply the coface operator to each digit of each vertex, i.e., we must see whether there are in f pairs of complementary vertices which can be combined into 1-cubes. This is done systematically in Table 5-1.

Vertex	Compl. 1st digit	Compl. 2nd digit	Compl. 3rd digit	Compl. 4th digit
(0000)	(1000) ✓	(0100) ✓	(0010)	(0001) ✓
(0001)	(1001)	(0101) ✓	(0011)	(0000) ✓
(0100)	(1100)	(0000) ✓	(0110) ✓	(0101) ✓
(0101)	(1101)	(0001) ✓	(0111)	(0100) ✓
(0110)	(1110) ✓	(1010) ✓	(0100) ✓	(0111)
(1000)	(0000) ✓	(0100) ✓	(1010) ✓	(1001)
(1010)	(0010)	(1110) ✓	(1000) ✓	(1011)
(1110)	(0110) ✓	(1010) ✓	(1100)	(1111)

Table 5-1 Calculation of Complementary Vertices

Whenever the complement is in f (denoted by a check-mark), we can form a 1-cube: (0000) and (1000) give (x000), (0000) and (0100) give (0x00), etc. Replacing 1-cubes which occur several times by a single mention, we obtain

$$K^1 = \{(0x00),\ (0x01),\ (010x),\ (000x),\ (10x0),\ (x110),\ (x000),\ (0x10),\ (01x0)\}$$

We can continue the process, examining only pairs of 1-cubes having the x in the same position. This leads to

$$K^2 = \{(0x0x)\}$$

This terminates the process. Note that δ_2^0 (0x0x) = (000x) for example, while δ_1^0(0x0x) = 0 since the first digit of (0x0x) is not an x. ϵ_4(0x00) = (0x0x) since (0x01) belongs to f, while ϵ_4(000x) = 0 because the 4^{th} digit is already an x. Figure 5-2 shows all the cubes of F = K(f).

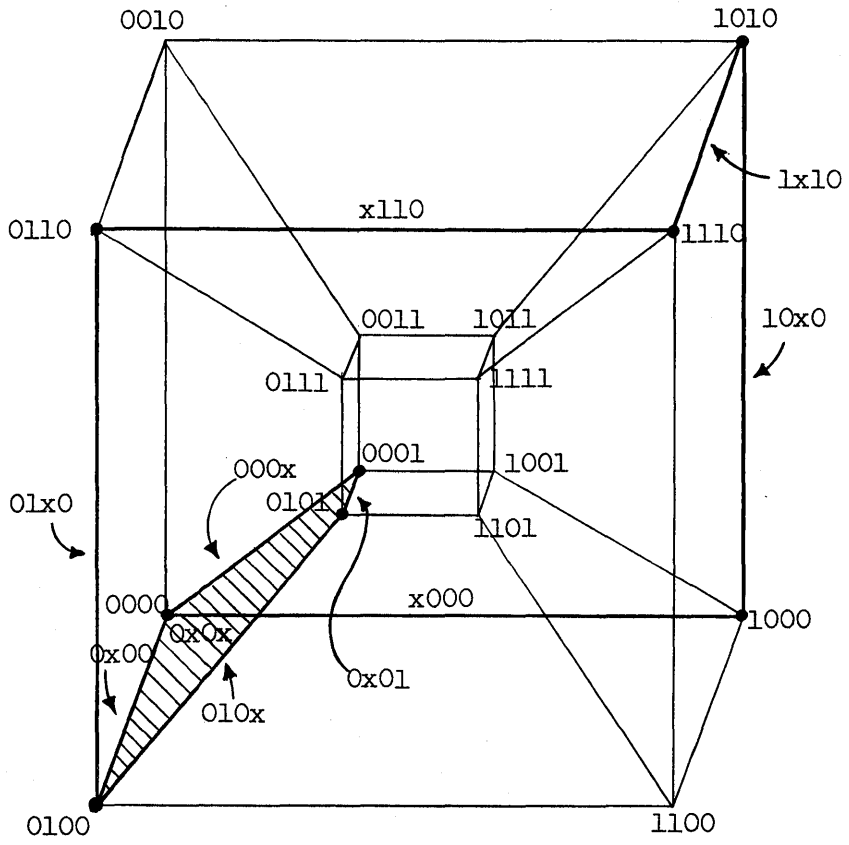


Figure 5-2. Cubical Complex of f = {(0000), (0001), (0100), (0101), (0110), (1000), (1010), (1110)}

5.2.3 MAPPING OF MINTERMS. MINIMUM COST COVERS

The action of the coface operator, which combines two complementary r-cubes into an (r + 1)-cube, is very similar to the operation we called "reduction" in Quine's Method (see Chapter III). This is, of course, no accident because the geometrical language we have just developed (due to Roth and Miller of IBM) is calculated to generalize Quine's Method, using the more elegant wording of geometry. The hyphen "-" used by Quine is the equivalent of the "x" used in the preceding sections.

Before using the Roth-Miller method of minimization, we first note that any function of n Boolean variables $x_1 \ldots x_n$ has a unique cononical expansion, i.e., that it is a unique sum of minterms. Via the binary correspondence

introduced in 3.4 each minterm corresponds to a multiplet of n binary digits ($\overline{x_i} \rightarrow 0$, $x_i \rightarrow 1$) and thus to a well defined vertex of an n-cube. This means that the set of vertices f [and F = K(f)!] is known as soon as the Boolean function is given. Because of the complete equivalence of the set of vertices and the function, we shall denote both by the same letter f.

The fundamental problem of simplifying a Boolean function now becomes equivalent to finding a set C of cubes covering f and causing (for the equivalent physical circuit) minimum cost. If C contains a number g_0 of 0-cubes (corresponding to AND's with n inputs), g_1 1-cubes (corresponding to AND's with n-1 inputs), etc., the criterion for minimizing the cost is usually

$$C_{AO} = \sum_{r=0}^{n} g_r(n - r) + \sum_{r=0}^{n} g_r = \sum_{r=0}^{n} g_r(n - r + 1) = \min. \qquad (5\text{-}25)$$

where the first term gives the total number of AND-inputs and the second term the number of inputs of the "collecting" OR: C_{AO} gives the number of arrowheads in the sense of Chapter III. We are thus led to a search for <u>as few cubes of F as possible, each having the maximum dimensions</u>.

In case we <u>have</u> to cover f, but <u>may</u> cover $f \vee g$, i.e., in case f gives the "care" conditions and g the "don't care" conditions (see Chapter III), one problem is to find a minimum cost subset of $K(f \vee g)$ which covers K(f) only. Let $\dot{K}(C)$ denote the complex of cubes using the vertices in the cover C only and let $\subset$ be the set-theoretical inclusion; then obviously

$$F = K(f) \subset K(C) \subset K(f \vee g) = M(\text{say}) \qquad (5\text{-}26)$$

5.3 Cocycles and Extremals

5.3.1 THE EXPANSION AND INTERSECTION OPERATORS

Let us take two cubes in n-space

$$c^r = (a_1 \ldots a_i \ldots a_n)$$

$$c^s = (b_1 \ldots b_i \ldots b_n)$$

where the number r of x's in c^r is not necessarily equal to the number s of x's in c^s. We shall then define two commutative operators, the expansion operator * and the intersection operator $\cap$ such that $c^r * c^s (= c^s * c^r)$ or $c^r \cap c^s$ $(= c^s \cap c^r)$ is a cube with digits $a_i * b_i$ or $a_i \cap b_i$ defined by Table 5-2.

Digit Combination a_i, b_i	$a_i * b_i$ or $a_i \cap b_i$
0,0	0
1,1	1
0,1	x
0,x	0
1,x	1
x,x	x

Table 5-2. Expansion and Intersection Operator Table

The difference in the two operators is that for the expansion operator $c^r * c^s$ is defined to be equal to zero if the combination 0,1 arises more than once, while $c^r \cap c^s$ is defined to be equal to zero if the combination 0,1 arises at all.

Theorem 1. $c^r * c^s$ is the largest cube containing subcubes of c^r and c^s as complementary faces.

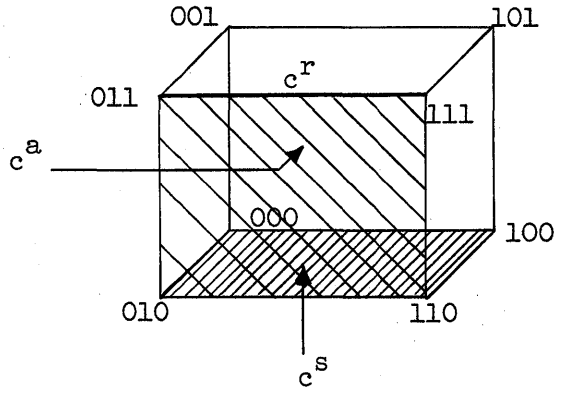
Proof: By putting 0 or 1 into the position of the newly obtained x (if there is such an x) the modified $c^r * c^s$ can be made to look like a subcube of either c^r or c^s. Since these subcubes are obtained by particularizing the same x to 0 or 1, they are complementary. No

more x's (i.e., no larger cubes with the property of the theorem) are possible because all those common to c^r and c^s occur automatically since x * x = x.

Theorem 2. $c^r * c^s$ has at the most one more x than Min(r,s).

This theorem is clearly a consequence of the method of formation of $c^r * c^s$. An interesting case arises when (for an arbitrary s) we consider successively cubes with r = 0 (then $c^r * c^s$ gives at the most 1-cubes), then cubes with r = 1 (giving at the most 2-cubes), etc.

Example.



Let c^r and c^s be the cubes

c^r = (x11)

c^s = (xx0)

shown in Fig. 5-3. Then $c^r * c^s$ = (x1x) = c^a, where c^a is also shown in the figure.

Figure 5-3. Action of the Expansion Operator

The expansion operator has the following properties:

$$(c^r * c^s) * c^t \neq c^r * (c^s * c^t) \qquad \text{(non-associativity)} \qquad (5\text{-}27)$$

$$\text{If } c^r \text{ is a subcube of } c^s,\ c^r * c^s = c^r \qquad (5\text{-}28)$$

Theorem 3. $c^r \cap c^s$ is the largest cube which is entirely contained in (i.e., is a subcube of) both c^r and c^s.

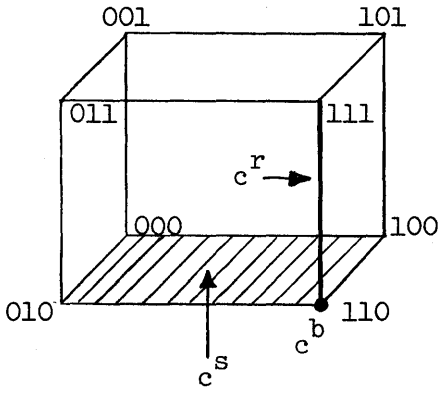
Proof: $c^r \cap c^s$ has zeros where both c^r and c^s had them, ones where both c^r and c^s had them. In cases where c^r or c^s had an x in a digit position the x has been particularized and thus a subcube formed.

The intersection operator has the following properties:

$$(c^r \cap c^s) \cap c^t = c^r \cap (c^s \cap c^t) \qquad \text{(associativity)} \qquad (5\text{-}29)$$

$$\text{If } c^r \text{ is a subcube of } c^s,\ c^r \cap c^s = c^r \qquad (5\text{-}30)$$

Example:



Let c^r and c^s be the cubes

$c^r = (11x)$

$c^s = (xx0)$

shown in Fig. 5-4. Then $c^r \cap c^s = (110) = c^b$, where the vertex c^b is also shown in the figure.

Figure 5-4. Action of the Intersection Operator

The operators * and $\cap$ can, by extension, be applied to sets of cubes rather than single cubes. If A and B are two such sets and c is any specific cube, we define

c * A = {all cubes obtained by applying the expansion operator to c and all cubes of A}

A * B = {all cubes obtained by applying the expansion operator to all possible combinations of one cube in A and one cube in B}

The definition of $c \cap A$ and $A \cap B$ is analogous.

5.3.2 COCYCLES

Suppose that we have a covering problem with an initial set of vertices f giving rise to a cubical complex F. We found F in an example in 5.2.2 using coface operators, but it is also apparent that one can use the expansion operator, applying it first to all pairs of vertices (and the result being zero if the pairs are not complementary!), then to pairs of 1-cubes having the x in the same position, etc. The reason for the success of the first step of this process is, of course, that in our case Theorem 1 states that the result of the expansion operator is "the largest cube containing the vertices as complementary faces." Whatever our procedure (i.e., ϵ_i or *) we shall end up with a great number of combinations and a great number of cubes, namely all cubes in F. It is, however, quite useless in a covering problem to have all cubes in F available: we only want those which are not contained in larger cubes. A cube of F which is not a subcube of a larger cube of F is called a cocyle. It is clear that the minimum cost cover is a combination of cocycles.

We shall now indicate how the set Z of cocycles (consisting of the set Z^0 of vertices not contained in 1-cubes of F, the set Z^1 of 1-cubes not contained in 2-cubes of F, etc.) can be found. It has become customary to generalize the problem slightly by not giving f but an arbitrary initial cover $\textcircled{D_0}$ of f, not necessarily minimal: $\textcircled{D_0}$ is thus a collection of cubes covering F without regard to cost.

The first step is to subtract from $\textcircled{D_0}$ all of those cubes which are contained in bigger cubes of D_0 (i.e., not bigger cubes of the complex formed with the vertices of $\textcircled{D_0}$ but cubes actually present in $\textcircled{D_0}$). Let D_0^* be the set of these cubes contained in bigger cubes. We then form

$$D_0 = \textcircled{D_0} - D_0^*$$

Let d^0 be any 0-cube left in D_0.

Remark: $D_0^* = \{c / c \subset d;\ c,d \in (D_0)\}$ in more abstract notation. This is read: D_0^* is the set of cubes c such that (symbol: /) c is a subcube of d (symbol: $c \subset d$) and both c and d belong (symbol: ϵ) to (D_0).

Theorem 4. The 0-cocycles are those 0-cubes of D_0 which cannot be combined with parts of higher order cubes of D_0 to form 1-cubes.

$$Z^0 = \{d^0 / d^0 * D_0 \not\supset \text{any 1-cube}\}$$

Proof: $d^0 * D_0$ forms at most 1-cubes by Theorem 2. The set $\{\}$ above rules out explicitly those d^0's which actually succeed in forming a 1-cube, i.e., none of the d^0's above have cofaces. This means that $Z^0 \subset \{\}$ since D_0 is certainly a cover and must therefore include all necessary 0-cubes to cover f and, in particular, those in Z^0 which are "unexpandable." Now assume that there is a d^0 in $\{\}$--say d for short--which is not a cocycle. Then there is an ϵ_i-operator such that $\epsilon_i d = e$, where e is a 1-cube covered by D_0. The complement of d, which has been used to form e, must also be covered by D_0, i.e., $d * D_0$ must contain e: d should have been eliminated in the first place! Hence all elements of $\{\}$ are cocycles.

We now form the union (or sum) of the set D_0 and $D_0 * D_0$:

$$(D_1) = D_0 \cup D_0 * D_0$$

We again take away the set D_1^* of cubes of (D_1) contained in larger cubes of (D_1) and consider

$$D_1 = (D_1) = D_1^* - \{\text{all 0-cubes of } (D_1)\}$$

It is clear that D_1 is not a cover of $F = K(f)$ but that $D_1 \cup Z^0$ is. The question now arises whether D_1 actually contains all 1-cubes of

F (or their cofaces), so that we can search for Z^1, i.e., all 1-cubes in F not contained in larger ones <u>using D_1 only</u>. The answer is given by

<u>Theorem 5</u>. D_1 contains all the 1-cubes of F or the cofaces of these cubes.

<u>Proof</u>: Suppose that a certain 1-cube d^1 or its coface is in D_0. Then by definition of D_1 d^1 must also be in D_1 because it was not taken from $D_0 \cup D_0 * D_0$ as a 0-cube and if it was taken away in D_1^* it was a subcube of a larger cube which is still in D_1.

Now suppose that d^1 was <u>not</u> in D_0; then there are two complementary faces a and b which, together, form d^1 and which are both covered by D_0. This means that D_0 contains two cubes $\alpha \supset a$ and $\beta \supset b$: $D_0 * D_0$ will then contain $\alpha * \beta$ which is a coface of d^1.

The procedure for finding Z^0 can now be extended to Z^1, Z^2, etc., as can Theorems 4 and 5. The iterative procedure is as follows: from D_r we form

$$\textcircled{D_{r+1}} = D_r \cup D_r * D_r \tag{5-31}$$

and

$$D_{r+1} = \textcircled{D_{r+1}} - D^*_{r+1} - \{\text{all } 0\text{-},1\text{-},\ldots,(r-1)\text{- and } r\text{-cubes of } \textcircled{D_{r+1}}\} \tag{5-32}$$

where D^*_{r+1} denotes all (r+1)-cubes contained in larger cubes of $\textcircled{D_{r+1}}$. The (r+1)-cocycles are obtained from

$$Z^{r+1} = \{d^{r+1}/d^{r+1} * D_{r+1} \neq \text{any } (r+2)\text{-cube}\} \tag{5-33}$$

where d^{r+1} is a cube of D_{r+1}. This iteration is followed until D_{r+1} is empty. It is usual to arrange the calculations of $D_0 * D_0$, etc., in the form of triangular arrays as shown in the example below.

Example. Let the initial cover of a certain f be given by

$$\textcircled{D_0} = \{(1x0),\ (x00),\ (01x)\}$$

since D_0^* is empty (none of the cubes of {} dontains any of the other two!), $D_0 = \textcircled{D_0}$. Since D_0 does not contain any 0-cubes, the set of 0-cocycles is also empty: $Z^0 = 0$. We now form $D_0 * D_0$ by the triangular array

	(1x0)	(x00)	(01x)
(1x0)	-----	(100)	(x10)
(x00)	-----	-----	(0x0)
(01x)	-----	-----	-----

where the dash indicates that the calculation of () * () is either without interest because the cubes in the operation are identical or that the result may be found elsewhere in the table. We now have

$$\textcircled{D_1} = \{(1x0),\ (x00),\ (01x),\ (100),\ (x10),\ (0x0)\}$$

and $D_1^* = \{(100)\}$. Since there are no 0-cubes to subtract

$$D_1 = \{(1x0),\ (x00),\ (01x),\ (x10),\ (0x0)\}$$

We now search for 1-cubes in D_1 (here actually all of them are 1-cubes) which cannot be combined to form higher order cubes. This is most expediently done by examining $D_1 * D_1$: this table will be needed anyway in the formation of D_2. This gives

	(1x0)	(x00)	(01x)	(x10)	(0x0)
(1x0)	-----	(100)	(x10)	(110)	(xx0)
(x00)	-----	-----	(0x0)	(xx0)	(000)
(01x)	-----	-----	-----	(010)	(010)
(x10)	-----	-----	-----	-----	(010)
(0x0)	-----	-----	-----	-----	-----

We see that (01x) is the only 1-cube which cannot be combined to give a larger cube [here (xx0)!] and that therefore $Z^1 = \{(01x)\}$.

Now we form

$$\textcircled{D_2} = \{(1x0), (x00), (01x), (x10), (0x0)\}$$
$$\cup \{(000), (100), (010), (110), (0x0), (x10), (xx0)\}$$

where the second set is formed of the cubes resulting from our table above, leaving out cubes which occur several times. When we take away cubes contained in larger ones, as well as all 0-cubes and all 1-cubes, we are left with

$$D_2 = \{(xx0)\}$$

The 2-cube in D_2 cannot be combined with anything else (to form a 3-cube (xxx), which would imply that the output is not connected to the input!) and therefore Z^2 contains just this cube and nothing else: $Z^2 = \{(xx0)\}$. Thus the set of cocycles of f is

$$Z = \{Z^0 \cup Z^1 \cup Z^2\} = \{(01x), (xx0)\}$$

It is essential to note that at no point in our calculation we had to calculate all the minterms of f. Figure 5-5 shows the cocycles in our example.

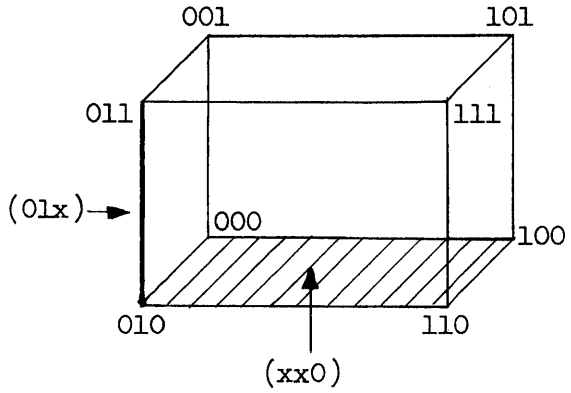


Figure 5-5. Cocycles of the Initial Cover {(1x0), (x00), (01x)}

5.3.3 EXTREMALS

Let us consider once more a problem in which we have f → "care conditions" and g → "don't care conditions." This means that we have to cover F = K(f) but that instead of using the cocycles of F only we may use those of M = K(f ∨ g). The problem is then to cover F with a subset of cocycles of M and at minimum cost.

We shall now introduce a subset E of the set Z of cocycles of M called extremals: these are cocycles covering vertices covered by no other cocycle or so-called distinguished vertices. It is customary to call such an extremal an "F-extremal of M" and to refer to the set E of all such extremals as E(M,F).

Theorem 6. Any minimal cover C of F contains E(M,R): $E(M,F) \subset C \subset Z$.

Proof: C must contain all distinguished vertices; therefore all extremals must be used: C must contain E(M,F). That any cover can be made out of cocycles, has been discussed before.

Theorem 7. If the set of extremals $E(M,F)$ is a cover, it is the minimal cover.

Proof: Since $E(M,F) \subset C$, the fact that $E(M,F) = C$ shows that it is the minimal cover.

We shall now introduce the neighborhood $U(z,Z)$ of a cocycle z as the set of cocycles s in Z which have at least one vertex in common with z or--using the intersection operator--for which $s \cap z \neq 0$:

$$U(z,Z) = \{s/s \in Z,\ s \cap z \neq 0\} \tag{5-34}$$

Since z itself is in U (because $z \cap z \neq 0$!), it is often useful to define the deleted neighborhood $U^-(z,Z)$ as the set $U(z,Z)$ minus z itself:

$$U^-(z,Z) = U(z,Z) \text{ minus } z \tag{5-35}$$

It will now be necessary to find $E(M,F)$ from F and the cocycles of M. First we shall establish a criterion to decide whether or not a cocycle is an extremal e.

Theorem 8. If e is an F-extremal of M, we have

$$\left.\begin{aligned} &K(e \cap F) \neq 0 \\ &K(e \cap F) \neq K[e \cap F \cap U^-(e,Z)] \end{aligned}\right\} \tag{5-36}$$

Conversely if (5-36) is satisfied, e is an F-extremal of M.

Proof: Suppose that e is an extremal, then there is at least one vertex d of F covered by e and by e only. This means that d is in e and also in F. Therefore $e \cap F \neq 0$ and the cubical complex $K(e \cap F) \neq 0$ for it must at least contain d. But d is not in any other cocycle z and in particular not in $U^-(e,Z)$: this means that $e \cap F \cap U^-(e,Z)$ cannot contain d ($e \cap F$ contains it, $U^-(e,Z)$ does not) and therefore $K(e \cap F) = K(e \cap F \cap U^-(e,Z))$.

Now suppose that we have found an e satisfying (5-36). Let us try to assume that $K(e \cap F) \subset K(U^-)$ where $K(U^-)$ is the complex of cubes formed with the vertices in U^-. Then it follows that $K(e \cap F \cap U^-) = K(e \cap F)$ because the supplementary condition $\cap U^-$ does not restrict us for a subset of U^-. This contradicts the second equation and we must therefore have $K(e \cap F) \not\subset K(U^-)$. Then there must be at least one vertex d in $K(e \cap F)$ which is not in $K(U^-)$. Now d must be in e (we formed $e \cap F$) but it is in no other cocycle: neither in those encompassed by U^- nor in those which do not even touch e, i.e., the others. Hence e is an extremal.

Example. Let us take a problem with F = M as shown in Fig. 5-6. It can be seen by inspection of the figure (note that a cube like (1xx1) has two possible complementary cubes, i.e., (0xx1) and (1xx0) with which it could form a larger cube!) that the cocycles are

$$Z = \{(1xx1), (x1x0), (000x), (11xx), (0x00), (x001)\}$$

Let us take Z = (x1x0) and consider its neighborhood: there must be cocycles having a 1 or an x in the second digit position and a 0 or an x in the fourth position. (11xx) and (0x00)--plus (x1x0) itself--form the neighborhood. Figure 5-6 shows that indeed the former two cocycles have common parts with (x1x0):

$$(x1x0) \cap (11xx) = (11x0)$$

$$(x1x0) \cap (0x00) = (0100)$$

Here, therefore

$$U(z,Z) = \{(x1x0), (11xx), (0x00)\}$$

$$U^-(z,Z) = \{(11xx), (0x00)\}$$

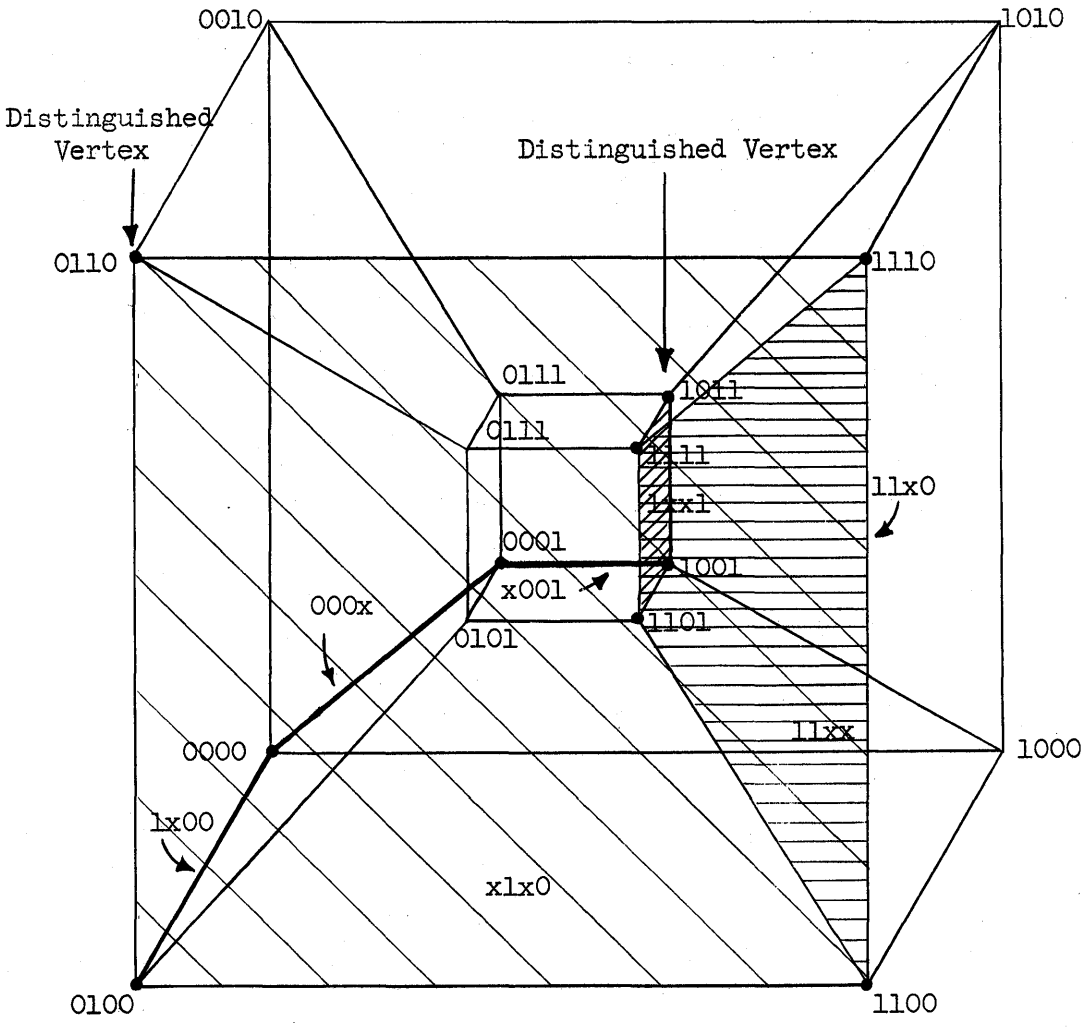


Figure 5-6. Cocycles and Distinguished Vertices for a Complex Defined by {(0100), (0000), (0001), (1001), (1011), (1111), (1101), (1100), (1110)}

5.4 The Roth-Miller Extraction Algorithm

5.4.1 ITERATIVE COVERING. BRANCHING

Suppose that we start out with $M = K(f \vee g)$ and $F = K(f)$. We can now find $Z(M)$ and also $E(M,F)$ by the procedures described in 5.3. If E is a cover of F, the problem is solved. If it is not, we proceed as follows: we set

$$M_1 = M, \quad F_1 = F, \quad Z_1 = Z \quad \text{and} \quad E_1 = E \tag{5-37}$$

and form

$$\textcircled{Z_2} = Z_1 - E_1 \qquad \text{(these are the left-over cocycles)} \tag{5-38}$$

and
$$\begin{aligned} F_2 &= F_1 - \text{subcomplex covered by } E_1 \\ &= K\{\text{left-over vertices}\} \end{aligned} \tag{5-39}$$

Let u and v be cubes in $\textcircled{Z_2}$ and consider $u \cap F_2$ and $v \cap F_2$. Suppose that $u \cap F_2 \subset v \cap F_2$, i.e., that as far as F_2 is concerned, v covers all that is covered by u. Furthermore suppose that cost u > cost v: then v is called a nonmaximal cube and eliminated. In case the costs are equal, we shall still retain the cube covering more of F_2.

Now we continue our process, setting

$$Z_2 = \textcircled{Z_2} - \text{nonmaximal cubes} \tag{5-40}$$

$$M_2 = K(Z_2) \tag{5-41}$$

$$E_2 = E(M_2, F_2) \tag{5-42}$$

Continuing this operation we find E_1, E_2, ..., until there are no further extremals. If $E_1 \cup E_2 \cup \ldots$ forms a cover of F, the problem is solved. Very often, however, we do not attain a cover and yet there are no distinguished vertices left: this is the so-called irreducible case. In such a case one examines the two covers obtained by branching: the first branch assumes that one particular cocycle of the remaining cocycles is in the cover, while the second branch assumes that it is not. The cost of the two branches is then compared and the lower cost one chosen. It is, of course, quite possible to have multiple branching, i.e., branching within each branch.

Example 1. Using Fig. 5-6 we find that there are two distinguished vertices: (1011) and (0101). We have seen that

$$Z_1 = Z(F) = \{(1xx1),\ (x1x0),\ (000x),\ (11xx),\ (0x00),\ (x001)\}$$

$$F_1 = M_1 = K\{(1xx1),\ (x1x0),\ (000x)\}$$

In order to cover the distinguished vertices we need

$$E_1 = \{(x1x0),\ (1xx1)\}$$

Therefore

$$(Z_2) = \{(11xx),\ (x011),\ (000x),\ (0x00)\}$$

$$F_2 = K\{(000x)\}$$

As far as F_2 is concerned, (11xx), (x011) and (0x00) are nonmaximal cubes, giving

$$Z_2 = \{(000x)\}$$

$$E_2 = \{(000x)\}$$

$E_1 \cup E_2$ visibly forms a complete cover: this cover is minimal by Theorem 7.

Example 2.

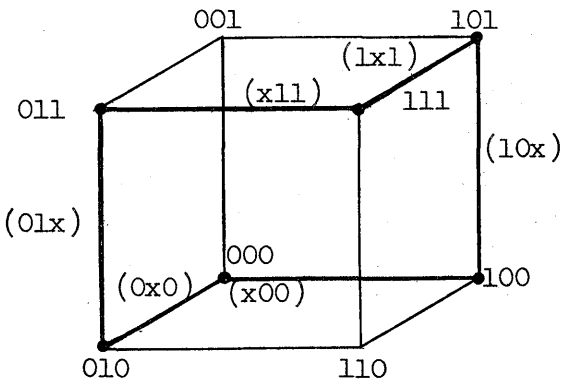


Figure 5-7. Irreducible Case

Let us consider the cubical complex F, defined by the vertices (000), (100), (101), (111), (011) and (010) in Fig. 5-7. It is easily seen that the cocycles (shown in heavy lines) are the 1-cubes of the set

$$Z_1 = \{(x00),\ (10x),\ (1x1),\ (x11),\ (01x),\ (0x0)\}$$

Visibly there are no distinguished vertices: each one of them is included in two cocycles. Starting from Z_1 we now branch out in two possible ways (actually there are 12 ways, but the other ten are equivalent by symmetry!):

Branch 1. We suppose that (x00) is included in the cover and even an extremal:

$$E_1 = \{(x00)\}$$

$$\textcircled{Z_2} = \{(10x), (1x1), (x11), (01x), (0x0)\}$$

We see moreover that what is left to cover of F_1 is simply

$$F_2 = K\{(1x1), (x11), (01x)\}$$

It is easily seen that as far as the covering of F_2 is concerned $(10x) < (1x1)$ and $(0x0) < (01x)$. (Actually one should examine the intersection of $\textcircled{Z_2}$ with F_2 by writing down all the cubes of F_2, i.e., {(1x1), (x11), (01x), (010), (011), (001), (101), (100)}. This is what a machine would do!) Now

$$Z_2 = \{(1x1), (x11), (01x)\}$$

and visibly

$$E_2 = \{(1x1), (01x)\}$$

Since $E_1 \cup E_2$ forms a cover C' of F_1, we reach the end of our problem with

$$C' = \{(x00), (1x1), (01x)\}$$

Branch 2. Now suppose that the cover does <u>not</u> contain (x00) and set

$$E_1 = 0$$

As before

$$\textcircled{Z_2} = \{(10x),\ (1x1),\ (x11),\ (01x),\ (0x0)\}$$

but this time <u>all</u> the vertices of F_1 remain to be covered (those of (x00) had been eliminated above):

$$F_2 = K\{(10x),\ (1x1),\ (x11),\ (01x),\ (0x0)\}$$

Clearly

$$Z_2 = \{(10x),\ (1x1),\ (x11),\ (01x),\ (0x0)\}$$

and

$$E_2 = \{(10x),\ (0x0)\}$$

i.e.,

$$\textcircled{Z_3} = \{(1x1),\ (x11),\ (01x)\}$$

But here

$$F_3 = K\{(x11)\}$$

Therefore

$$(1x1) < (x11) \quad \text{and} \quad (01x) < (x11)$$

i.e.,

$$Z_3 = (1x1)$$

$$E_3 = (x11)$$

This gives us the alternate cover

$$C^{11} = \{(10x), (x11), (0x0)\}$$

Since both covers consist of three 1-cubes, their cost is identical and we may choose either one.

5.4.2 SYMBOLIC NOTATION. TOPOLOGICAL EQUIVALENTS

It is clear after inspecting the second example of the last section that it is by no means necessary to write down the cubical form for each cocycle as long as we deduce all relationships by direct inspection of a figure. If we read off the adjacencies on such a figures, we can replace the cubical notation of the cocycles by--arbitrarily chosen--symbols such as a, b, c, etc., and write down our iterative steps in symbolic form. This aids clarity enormously. It should be remarked, however, that the "blindfolded" calculation a machine would go through must use the full cubical expression of each cube.

Example 1. Let us introduce in the second example of 5.4.1 the following symbolic representation:

$$(x00) \rightarrow a,\ (10x) \rightarrow b,\ (1x1) \rightarrow c$$

$$(x11) \rightarrow d,\ (01x) \rightarrow e,\ (0x0) \rightarrow f$$

Then we can write for branch 1

$$E_1 = a$$

$$\textcircled{Z_2} = \{b,c,d,e,f\}$$

$$F_2 = K\{c,d,e\}, \text{ etc.}$$

The very fact that symbols can be used to denote cubes and that in figures only the adjacencies of cubes count, show that in multidimensional problems it is possible to extract those cubes which interest us in a

minimization problem and to "lay them out" in a space of fewer dimensions--if possible a plane. As long as the figures in the subspace is the topological equivalent, all relationships necessary to calculate a cover can be read directly from it.

Example.

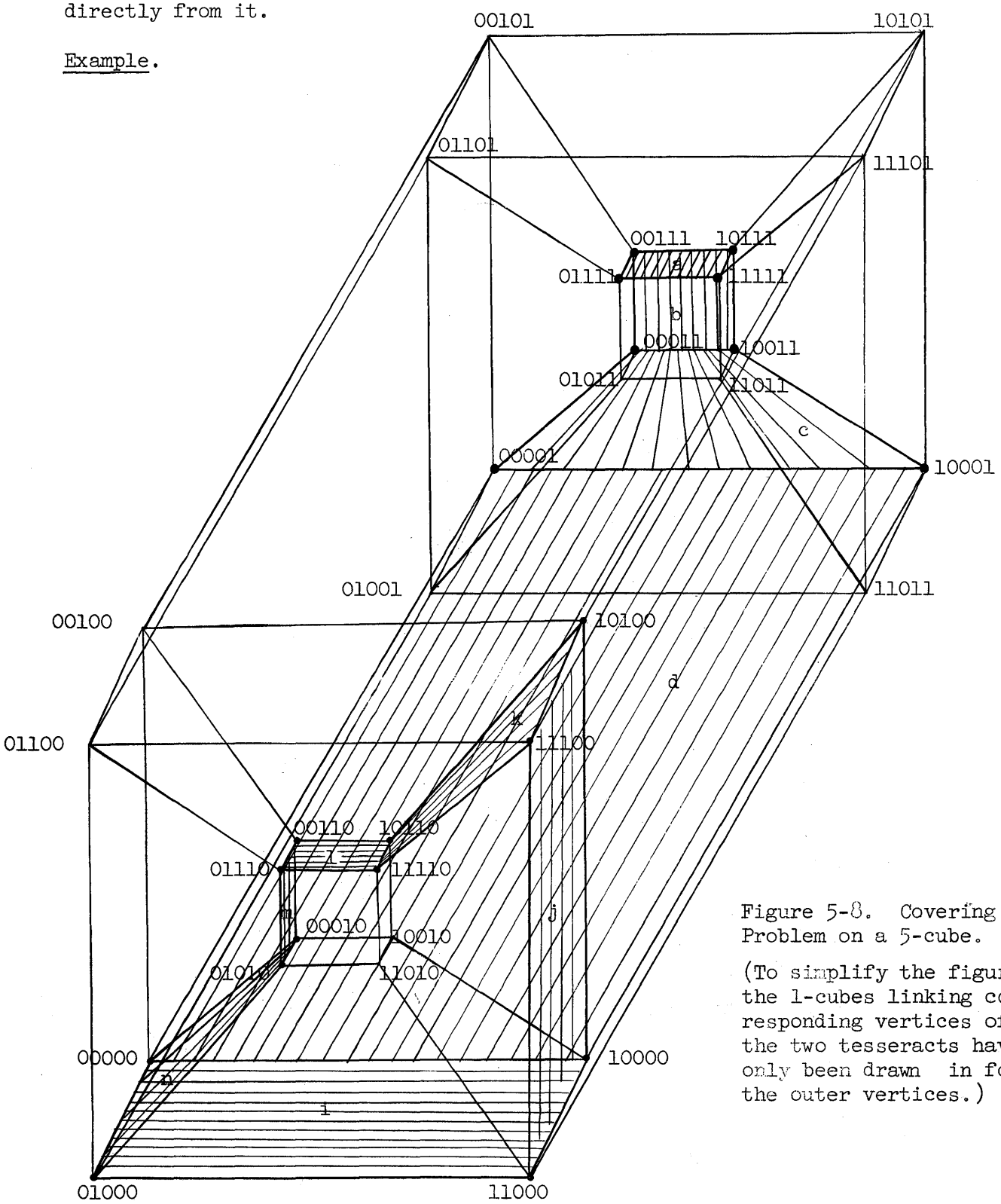


Figure 5-8. Covering Problem on a 5-cube.

(To simplify the figure, the 1-cubes linking corresponding vertices of the two tesseracts have only been drawn in for the outer vertices.)

The complex indicate on the 5-cube in Fig. 5-8 is topologically equivalent to the one illustrated in Fig. 5-9. The squares denoted by a, b, c, d, i, j, k, l, m, n are all cocycles.

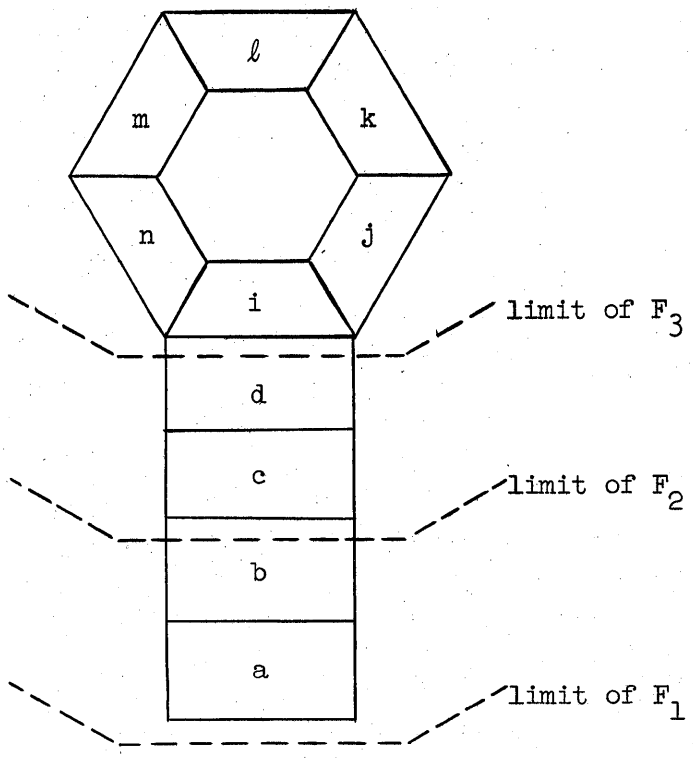


Figure 5-9. Plane Topological Equivalent of Figure 5-8

Thus we start from

$$M_1 = F_1 = K\{a,b,c,d,i,j,k,l,m,n\}$$

Since there are all cocycles

$$Z_1 = \{a,b,c,d,i,j,k,l,m,n\}$$

Now clearly a is an extremal (since it is at the end):

$$E_1 = a$$

Then

$$\textcircled{Z_2} = \{b,c,d,i,j,k,l,m,n\}$$

and

$$F_2 = K\{c,d,i,j,k,l,m,n\}$$

As far as F_2 is concerned $b < c$ and

$$Z_2 = \{c,d,i,j,k,l,m,n\}$$

But this makes c an extremal:

$$E_2 = c$$

Now

$$\textcircled{Z_3} = \{d,i,j,k,l,m,n\}$$

$$F_3 = K\{i,j,k,l,m,n\}$$

But again, as far as F_3 is concerned, $d < c$ and

$$Z_3 = \{i,j,k,l,m,n\}$$

This is now an irreducible case: the remaining cocycles form a sort of ring and we must branch. Following the general branching procedure, we find

Branch 1. Assume that i is in the cover, i.e.,

$$E_3 = i$$

$$\textcircled{Z_4} = Z_3 - i = \{j,k,l,m,n\}$$

$$F_4 = K\{k,l,m\}$$

Clearly

$$j < k,\ n < m$$

and

$$Z_4 = \{k,l,m\}$$

Now

$$E_4 = \{k,m\}$$

$$\textcircled{Z_5} = \{l\}$$

But $F_5 = 0$ since all is covered, and we obtain a cover

$$C^1 = \{z,c,i,k,m\}$$

Branch 2. Here we assume that i is not in the cover. Then

$$E_3 = 0$$

Consequently

$$\textcircled{Z_4} = \{j,k,l,m,n\}$$

$$F_4 = K\{j,k,l,m,n\}$$

i.e.,

$$Z_4 = \{j,k,l,m,n\}$$

Since j and n are now at the end

$$E_4 = \{n,j\}$$

Removing n,j from Z_4, we find

$$\textcircled{Z_5} = \{k,l,m\}$$

$$F_5 = K\{l\}$$

This means that $k < l$ and $m < l$ and

$$Z_5 = l$$

and

$$E_5 = l$$

It is also clear that the sum of all extremals gives a cover

$$C^{II} = \{a,c,j,l,n\}$$

Again the cost of the two covers C^{I} and C^{II} is identical. We can choose either one.

5.4.3. THE ROTH-MILLER EXTRACTION ALGORITHM

Whether we program a machine in order to perform the iterative steps or whether we examine by inspection a topologically equivalent figure using a symbolic notation, the steps we have to perform always follow the same pattern. This is described by Roth and Miller as follows:

We start with (M_1,F_1) and form $Z_1(M_1)$, $E_1(M_1,F_1)$. Then $\textcircled{Z_2} = Z_1 - E_1$ is formed as well as $F_2 = F_1$ - complex covered by E_1 and after the elimination of nonmaximal cubes $Z_2 = \textcircled{Z_2}$ - nonmaximal cubes is formed. This process is iterated until either a complete cover is obtained or until branching is necessary. Formally the step $r \rightarrow r + 1$ is as follows:

1. Z_r is known as well as F_r. In case extremals exist, we find E_r.
2. We form

$$\textcircled{Z_{r+1}} = Z_r - E_r \qquad (5\text{-}43)$$

$$F_{r+1} = F_r - \text{cubes covered by } E_r = K \{\text{vertices in } F_r - \text{vertices covered by } E_r\} \quad (5\text{-}44)$$

$$Z_{r+1} = \textcircled{Z_{r+1}} - \text{nonmaximal cubes with respect to } F_{r+1} \quad (5\text{-}45)$$

$$E_{r+1} = \text{extremals of } Z_{r+1} \quad (5\text{-}46)$$

3. If there are no extremals, we branch by comparing:

3a. Assume a given cocycle, a, of Z_r is part of the cover. We set $E_r = a$ and form

$$\textcircled{Z_{r+1}} = Z_r - E_r \quad (5\text{-}47)$$

We have to cover the complex F_r minus the cubes covered by a; call this F^1_{r+1}. We eliminate nonmaximal cubes from Z_{r+1} and examine Z^1_{r+1} for extremals. This brings us back to a step like 2 or 3.

3b. Assume cocycle a above is <u>not</u> part of the cover. We set $E_r = 0$ and form (<u>as above</u>)

$$\textcircled{Z_{r+1}} = Z_r - a$$

This time, however, we have to cover $F^{11}_{r+1} = F_r$ since no simplification has been obtained in the preceeding step. This will give us a Z^{11}_{r+1} which differs from Z^1_{r+1} above, but we are also back to steps like 2 or 3.

5.4.4 <u>THE SHARP OPERATOR (SUBTRACTION OPERATOR)</u>

The reader may have noted that in the formation of F_{r+1} from F_r by the use of (5-44) we had to fall back on an explicit enumeration of "left-over vertices" in order to form the new complex. This is highly undesirable: in the calculation of cocycles we already formulated a method which starts with

a nonmaximal cover as the basis of all calculations. The same thing is possible for the passage from $F_r \rightarrow F_{r+1}$ if we use the Sharp Operator. This operator is defined as follows:

Let c^r and c^s be two cubes of a cubical complex:

$$c^r = (a_1 a_2 \ldots a_n)$$

$$c^s = (b_1 b_2 \ldots b_n)$$

Let us define the sharp (#) or subtraction operation on the ith digits of the two cubes by Table 5-3.

	#	b_i: 0	1	x
a_i	0	z	y	z
	1	y	z	z
	x	1	0	z

Table 5-3. Sharp Operator

Then the # operation on two cubes is defined by the following rules:

1. If for all i $(1 \leq i \leq n)$ $a_i \# b_i = z$ (i.e., a_i and b_i identical or b_i an x!) then

$$c^r \# c^s = 0 \tag{5-48}$$

2. If for some i $a_i \# b_i = y$ (i.e., a_i the opposite of b_i), then

$$c^r \# c^s = c^r \tag{5-49}$$

3. If $a_i \# b_i =$ (0 or 1) (i.e., a_i is an x but b_i is not) for some i's, viz., $i = i^1$, $i = i^{11}$, etc., then

$$c^r \# c^s = \sum_j (a_1 \ldots a_{j-1} \bar{b}_j a_{j+1} \ldots a_n) \qquad (5\text{-}50)$$

where $j = i^1, i^{11}, \ldots$

and where the sum should be understood in the $\cup$ sense. $c^r \# c^s$ is thus the sum of a certain number of cubes complementary to cofaces of the minuend.

Example. On a 3-cube (xxx) together with one of its 1-cubes (1x0) we have for instance (xxx) # (1x0) = (0xx) $\vee$ (xx1), i.e., when we take the left lower edge away from the cube, we are left with the sum of the left face and the upper face.

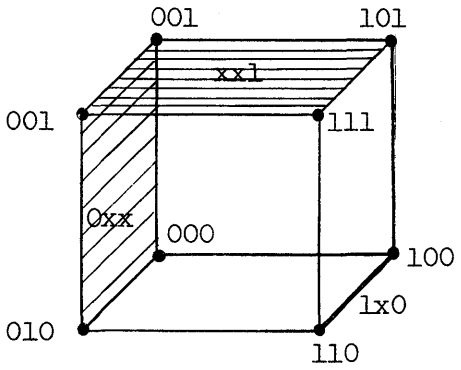


Figure 5-10. Sharp Operation on a 3-cube (xxx)

Theorem 9. $c^r \# c^s$ forms all subcubes of c^r which are not included in c^s (i.e., we are left with the biggest cubes one can build out of the vertices of c^r after those in $c^r \cap c^s$ have been taken away).

Proof: Let us take the three cases of the definition separately.

1. If the subtrahend cube has the same digits (or x) as the minuend cube, all vertices of the minuend will be taken away: the occurrence of z in all positions indicates precisely this.
2. If $a_i \# b_i = y$ for a given i, the minuend and subtrahend are opposite faces of a bigger cube, obtained by replacing the 0 and 1 in digit i by x. Such opposite faces cannot intersect: the minuend is therefore not affected by the operation.

3. If a 0 or 1 occurs, there was an x-digit in the minuend: the operation forms the other face (complement in the digit position!) of the bigger cube (→ to x in position i) still left over. For several x's, we take the sum of all complementary cubes.

The sharp operator has the following properties:

$$c^r \# c^s = c^r \quad \text{if} \quad c^r \cap c^s = 0 \tag{5-51}$$

$$c^r \# c^s \subset c^r \tag{5-52}$$

$$c^r \# c^s \neq c^s \# c^r \tag{5-53}$$

$$(c^r \# c^s) \# c^t = (c^r \# c^s) \cup (c^s \# c^t) \tag{5-54}$$

$$(c^r \# c^s) \# c^t \neq c^r \# (c^s \# c^t) \tag{5-55}$$

$$(c^r \# c^s) \# c^t = (c^r \# c^t) \# c^s \tag{5-56}$$

The proof of these properties follows more or less immediately from the definition of the sharp operator. Note that (5-56) can be generalized by saying: it is allowed to subtract a set of cubes <u>in any order</u> from a given cube--no brackets have to be used and $(c_1 \# c_2) \# c_3 \ldots$ can be written $c_1 \# c_2 \# c_3$.

The $F_r \rightarrow F_{r+1}$ step is now described by

<u>Theorem 10</u>. If $F_r = K\{c_1, \ldots, c_n\}$

and $E_r = \{e_1, \ldots, e_m\}$ then

$$F_{r+1} = K\left\{\begin{matrix} c_1 \# e_1 \# e_2 \cdots e_m \\ c_2 \# e_1 \# e_2 \cdots \# e_m \\ \vdots \end{matrix}\right\} \tag{5-57}$$

Proof: $c_1 \# e_1 \# e_2 \ldots \# e_m$ contains all vertices of c_1 not contained in the extremals $e_1 \ldots e_m$. Hence by the definition of K the right-hand side equals F_{r+1}.

Let us now introduce some further definitions:

1. Let $C = \{c_1, c_2, \ldots, c_m\}$. Then by definition

$$c \# C = c \# c_1 \# c_2 \ldots \# c_m \tag{5-58}$$

2. Let $C_0 = \{c_{01}, c_{02}, \ldots, c_{0n}\}$

$$C_1 = \{c_{11}, c_{12}, \ldots, c_{1m}\}$$

Then by definition

$$C_0 \# C_1 = \left\{ \begin{array}{c} c_{01} \# C_1 \\ c_{02} \# C_1 \\ \vdots \\ c_{0n} \# C_1 \end{array} \right\} \tag{5-59}$$

Now we simply write

$$F_{r+1} = K\{F_r \# E_r\}$$

The sharp operator is useful too when we want to decide on the equivalence of two covers because we have

Theorem 11. Two covers C_0 and C_1 cover the same complex if and only if

$$C_0 \# C_1 = C_1 \# C_0 = 0$$

Proof: $C_0 \# C_1$ contains all vertices of C_0 not in C_1 and $C_1 \# C_0$ all vertices of C_1 not in C_0.

5.5 Partially Ordered Sets

1. DEFINITION OF POSETS

Sets can be totally ordered ("toset"), partially ordered ("poset") or unordered: it is, of course, necessary to define the relationship with respect to which ordering occurs. Examples will illustrate the three cases.

Example 1. The heights of mountains inside the continental confines of the U.S.A. can be totally ordered by the relationship "higher than" or "lower than."

Example 2. The successive generations of a family can be ordered by the relationship "is a descendant of." Figure 5-11 shows a family tree: it is what is called later on a "Hasse diagram" of the poset in question.

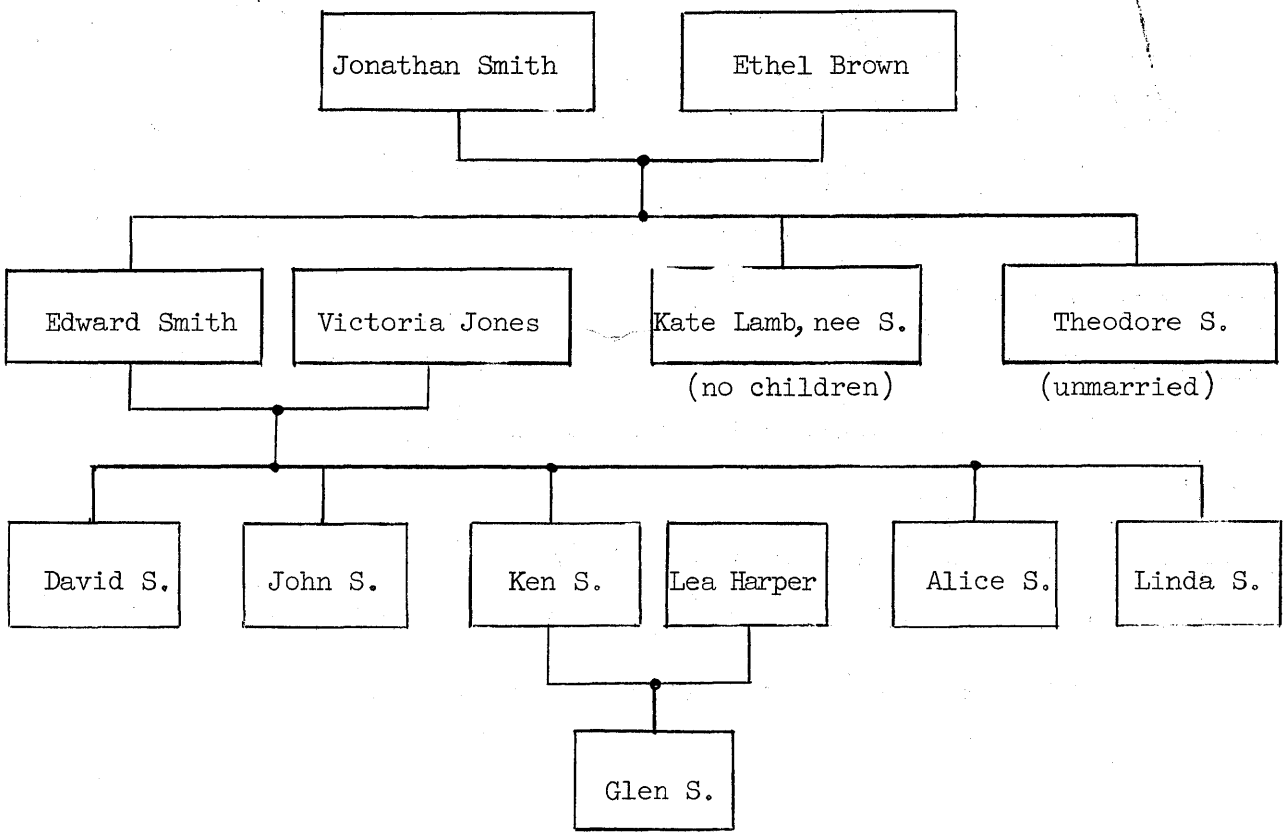


Figure 5-11

A Family Tree as a Graphical Representation of a Partially Ordered Set

Example 3. The set of complex numbers cannot be ordered by the relationship "argument less than" because for a given argument there are many complex numbers.

Definition of a Poset

A set $X = \{x_1, \ldots, x_n\}$ is partially ordered (a poset) if and only if for some pairs x_i, x_j there is a relationship

$$x_i \leq x_j \qquad \text{(inclusion)}$$

One then says: x_j includes x_i, x_j follows x_i or x_j is greater than x_i. Note that this relationship is not given for all pairs: otherwise the poset would be a toset.

The $\leq$ relation has the following properties:

1. $x_i \leq x_i$ (reflexivity)
2. if $x_i \leq x_j$ and if $x_j \leq x_i$ then $x_i = x_j$ (antisymmetry)
3. if $x_i \leq x_j$ and $x_j \leq x_k$ then $x_i \leq x_k$ (transitivity)

Theorem 1. Cyclical lists cannot occur in a poset.

Proof: Suppose $x_1 \leq x_2 \leq \ldots \leq x_n \leq x_1$

Then by the antisymmetric and transitive laws we have

$$x_1 = x_2 = \ldots = x_n \text{ , QED.}$$

Remark: If $x_1 \leq x_2$, then clearly $x_2 \geq x_1$. But the ordering relationship could also be turned around, i.e., we could again write $x_2 \leq x_1$, because the ordering symbol can correspond to "bigger than" or "less than". In other words: in any theorem about lattices we can always substitute $\geq$ for $\leq$.

Definition: If $x_i \leq x_j$ and $x_i \neq x_j$ we write

$$x_i < x_j$$

and if furthermore there exists no x_k between x_i and x_j such that

$$x_i \leq x_k \leq x_j$$

and we say x_j covers x_i.

2. HASSE DIAGRAMS

The partial ordering relationship can be illustrated graphically if we adopt the following rules of correspondence:

The elements of the set, x_i, correspond to: points or circles

$x_i < x_j$ corresponds to: point x_j is above point x_i

x_i covers x_j corresponds to: a segment leads from x_i to x_j without interruption (points x_i and x_j are directly connected)

The resulting diagram is called the Hasse diagram.

Remark: Note that if an element x_i covers x_j, they cannot both be covered by an element k nor can they both cover such an element. Suppose that k covers x_i (which covers x_j): Then $k > x_i > x_j$, i.e., x_i is between k and x_j and consequently k cannot cover x_j. In the Hasse Diagram this means that no triangles can occur. More generally: if x_i covers x_j there cannot be any side-branch (passing through more than one element) leading from x_i to x_j.

Example. Take all functions of two Boolean variables x_1, x_2 as was shown in Section 5.1. They can be written in the form

$$f = a\,\bar{x}_1\,\bar{x}_2 \vee b\,\bar{x}_1\,x_2 \vee c\,x_1\,\bar{x}_2 \vee d\,x_1\,x_2$$

Let $f_1 = f(a_1, b_1, c_1, d_1)$

$f_2 = f(a_2, b_2, c_2, d_2)$

then $f_1 \leq f_2$ is defined as meaning that

$$1.\quad f_1 \cdot f_2 = f_1$$

$$\text{or } 2.\quad f_1 \vee f_2 = f_2$$

As was shown in Chapter 3 these two definitions are mutually consistent and one follows from the other.

The corresponding Hasse diagram is then as shown in Figure 5-12.

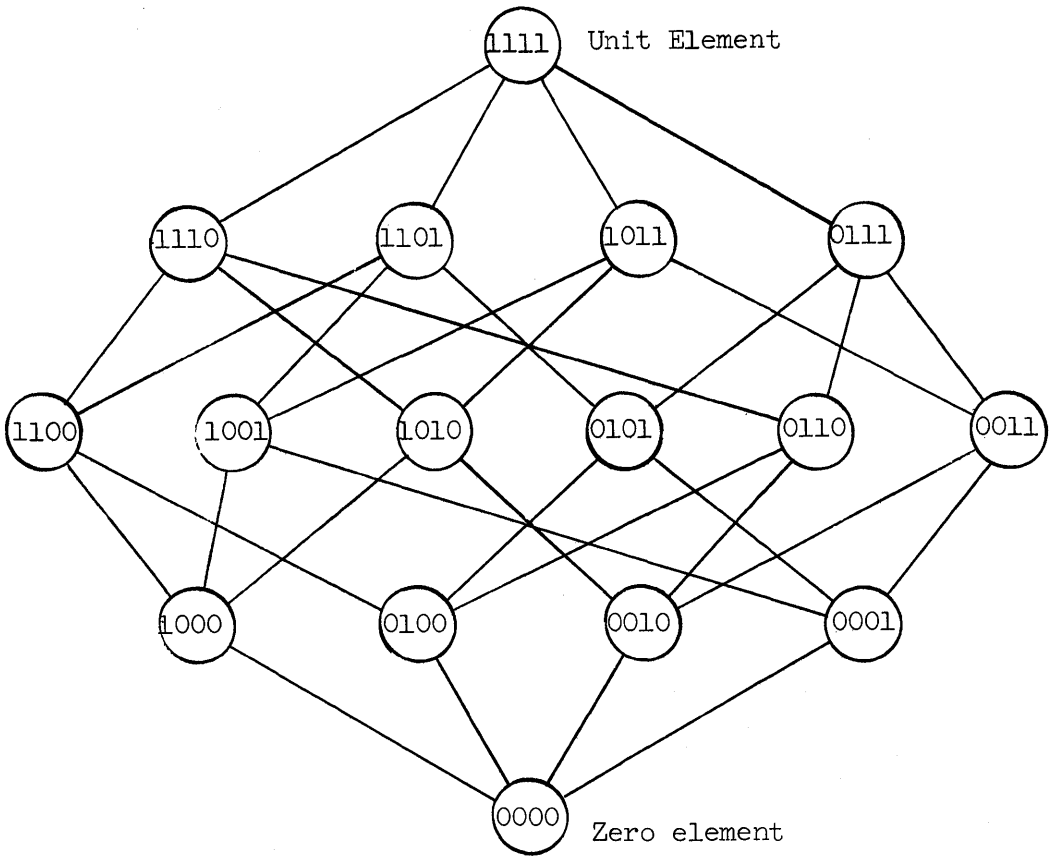


Figure 5-12

Hasse Diagram for the Boolean Functions of Two Variables

3. LUBS AND GLOBS

Definition 1. A maximum element of a poset is the one which is under no other elements.

Definition 2. A minimum element of a poset is above no other elements.

Remark: Obviously each poset has at least one maximum element and one minimum element.

Definition 3. The unit element is a unique maximum element (if it exists).

Definition 4. The zero element is a unique minimum element (if it exists).

Definition of lub (lowest upper bound)

a. Let Y be a subset of a poset X and let there be an $x_i \in X$ such that for every $y_j \in Y$ we have $y_j \leq x_i$. Then x_i is called an ub (upper bound) of the set Y.

   Remark: x_i may or may not belong to Y.

b. If there exists an x_k lower than any x_i (the upper bounds of Y) the x_k is called the lub (lowest upper bound) of Y; x_k = lub (Y).

Definition of glob (greatest lower band)

a. Let Y be a subset of a poset X and let there be an $x_i \in X$ such that for every $y_j \in Y$ we have $x_i \leq y_j$, then x_i is called a lob (lower bound) of Y.

   Remark: x_i may or may not belong to Y.

b. If there is an x_k above all lob's, then this x_k is called the glob (greatest lower bound); x_k = glob (Y)

Example: Let $X = \{x_1, x_2, x_3, x_4, x_5\}$ in Figure 5-11 and let Y be the set

$Y = \{x_3, x_4, x_5\}$

Visibly $Y \subset X$

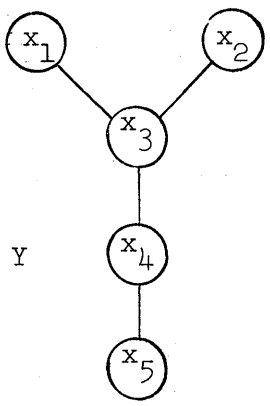


Figure 5-13

Here x_1 x_2 and x_3 are upper bounds of Y. The lub is x_3 and here belongs to Y. The glob of Y is clearly x_5. Note that this poset has a zero (namely x_5) but no unit: There are two maximum elements.

Remark: The lub and glob of a subset Y of a poset X are unique if they exist. This stems from the fact that if x_1 and x_2 for example are both lubs, $x_1 \leq x_2$ and $x_2 \leq x_1$, i.e., $x_1 = x_2$.

5.6 Lattices

1. DEFINITION OF LATTICES

Definition 1. A lattice is a poset in which every arbitrary pair x_i, x_j has a lub and a glob, i.e., there are two other elements of the lattice x_u and x_ℓ such that

$$\left.\begin{aligned} x_i &\leq x_u \\ x_i &\leq x_u \end{aligned}\right\} \quad \text{and} \quad \left.\begin{aligned} x_\ell &\leq x_i \\ x_\ell &\leq x_j \end{aligned}\right\}$$

and there are no lower x_u's and no higher x_ℓ's.

This means, of course, that "going up" or "going down" in a Hasse diagram for a lattice, we shall converge, for any pair of points, on a "single nearest common point" both above and below. An example is visibly furnished by all switching functions of two variables as shown in Figure 5-12.

We shall introduce the symbols $\cup$ and $\cap$ (not to be confused at this stage with v and $\wedge$) by the following

Definition 2.

$$\left.\begin{aligned} \text{lub}\ (x_i, x_j) &= x_i \cup x_j \\ \text{glob}\ (x_i, x_j) &= x_i \cap x_j \end{aligned}\right\} \tag{5-60}$$

Theorem 1. If $x_i \cup x_j$ (or $x_i \cap x_j$) equals one of the factors for all i, j, the poset is a totally ordered set (toset).

Proof: Suppose for instance that

$$x_i = x_i \cup x_j = \text{lub}\ (x_i, x_j)$$

This implies that $x_i \leq x_j$ by the definition of lub; thus for all i, j either $x_i \geq x_j$ or $x_i \leq x_j$, hence the elements are totally ordered.

Theorem 2. In a lattice L each finite subset Y has a lub and a glob.

Proof: The proof can be given by induction:

1. Suppose $Y_1 = \{x_k\}$; then visibly glob Y_1 = lub $Y_1 = x_k$, i.e., the theorem holds.

2. Suppose $Y_2 = \{x_k, x_{k+1}\}$, then by the definition of a lattice, the theorem is satisfied.

3. Suppose $Y_n = \{x_k, \ldots, x_{k+n}\}$ and suppose that the theorem holds for this Y_n, i.e., that lub $(Y_n) = x_{k+i}$ and glob $(Y_n) = x_{k+j}$ exist. Let

$$Y_{n+1} = \{Y_n, x_{k+n+1}\}$$

then clearly

$$\text{lub } (Y_{n+1}) = \text{lub } (\text{lub } Y_n, x_{k+n+1})$$

$$= \text{lub } (x_{k+i}, x_{k+n+1})$$

Since x_{k+i} and x_{k+n+1} are elements of L, the lub exists by the definition of a lattice. Similarly glob (Y_{n+1}) = glob (glob Y_n, x_{k+n+1}) exists and thus the theorem is true for all n, hence it is true for all lattices.

Theorem 3. Every finite lattice contains a unit and a zero.

Proof: By the definition of L we have a lub for all pairs of elements of L, and by an iterative process we clearly can find an element which is a lub of all elements of L. This element satisfies the requirements of a unit. By similar reasoning, one can show that a zero element exists.

It is quite obvious from the definition of lub and glob that we have

$$\left.\begin{aligned} 0 \cap x_i &= 0 \\ 0 \cup x_i &= x_i \end{aligned}\right\} \qquad (5\text{-}61)$$

$$\left.\begin{aligned} 1 \cap x_i &= x_i \\ 1 \cup x_i &= 1 \end{aligned}\right\} \qquad (5\text{-}62)$$

2. PROPERTIES OF LATTICES

In every lattice the following identities hold for the glob and lub operations:

$$\left.\begin{aligned} &1.\ x_i \cup x_j = x_j \cup x_i \\ &\text{and } x_i \cap x_j = x_i \cap x_i \end{aligned}\right\} \tag{5-63}$$

$$\left.\begin{aligned} &2.\ (x_i \cup x_j) \cup x_k = x_i \cup (x_j \cup x_k) \\ &\text{and } x_i \cap (x_j \cap x_k) = (x_i \cap x_j) \cap x_k \end{aligned}\right\} \tag{5-64}$$

$$3.\ x_i \cap x_i = x_i \cup x_i = x_i \tag{5-65}$$

$$4.\ (x_i \cap x_j) \cup x_i = x_i = (x_i \cup x_j) \cap x_i \tag{5-66}$$

All of these properties are nearly self-evident except (5-66): from the definition of a glob we have

$$x_i \cap x_j \leq x_i$$

hence $$(x_i \cap x_j) \cup x_i = \text{lub}\,[\text{glob}\,(x_i, x_j), x_i] = \text{lub}\,(x_i, x_i) = x_i$$

Remark: It will be shown later that a Boolean Algebra is a special type of lattice if $\cup$ and $\cap$ operations correspond to the OR and AND operations respectively.

Theorem 4. Any set in which two operations $\cup$ and $\cap$ are defined and satisfy properties (5-63) to (5-66) is a lattice.

Proof: Let us define the "$\leq$" relation by

$x_i \leq x_j$ if and only if $x_i \cup x_j = x_j$

Now we have to show that the poset postulates hold for the $\leq$ relation as defined and furthermore that glob $(x_i, x_j) = x_i \cap x_j$ and lub $(x_i, x_j) = x_i \cup x_j$ with respect to this relation.

Effectively we have

1. Reflexitivity: $x_i \leq x_i$ since $x_i \cup x_i = x_i$

2. Antisymmetry: $x_i \leq x_j$ and $x_j \leq x_i$ imply that

$$x_i \cup x_j = x_j \quad \text{and} \quad x_j \cup x_i = x_i$$

But from property (5-63)

$$x_i \cup x_j = x_j \cup x_i, \text{ hence } x_i = x_j$$

3. Transitivity:

Let $x_i \leq x_j$ and $x_j \leq x_k$ implying that $x_i \cup x_j = x_j$ and that $x_j \cup x_k = x_k$. By property (5-64) we can write

$$x_i \cup x_k = x_i \cup (x_j \cup x_k) = (x_i \cup x_j) \cup x_k = x_j \cup x_k = x_k$$

but this implies that $x_i \leq x_k$, thus transitivity holds for the relation.

Now consider that

$$x_i \cup (x_i \cup x_j) = (x_i \cup x_i) \cup x_j = x_i \cup x_j$$

but this implies that $x_i \leq x_i \cup x_j$

and $x_j \leq x_i \cup x_j$

hence $x_i \cup x_j$ is an upper bound of (x_i, x_j). It remains to be shown that $x_i \cup x_j$ is actually the lub; let x_k be another upper bound of x_i, x_j, i.e.,

$$x_i \leq x_k, \; x_j \leq x_k$$

implying that

$$x_i \cup x_k = x_k \quad \text{and} \quad x_j \cup x_k = x_k.$$

Then we have

$$(x_i \cup x_j) \cup x_k = x_i \cup (x_j \cup x_k) = (x_i \cup x_k) = x_k$$

hence from the definition of "$\leq$"

$$x_i \cup x_j \leq x_k$$

i.e., $x_i \cup x_j$ is lower than any other upper bound x_k, hence it is the lub of $\{x_i, x_j\}$. The proof of the theorem can be completed by using similar arguments to show that

$$x_i \cap x_j = \text{glob } (x_i, x_j)$$

Duality: Since properties (5-63) ... (5-66) define a lattice by Theorem 4, and since these expressions are perfectly symmetrical with respect to $\cup$ and $\cap$, it follows that each theorem about lattices has it dual, obtained by interchanging $\cup$ and $\cap$.

3. SPECIAL LATTICES

We shall now consider briefly several other types of lattices which are of interest either from a theoretical point of view because they form a link to Boolean Algebra or from the point of the theory of asynchronous circuits to be presented in Chapter VI.

Semi Modular Lattice:

Definition: A lattice is a semi modular if either one (but not both) of the following conditions is satisfied.

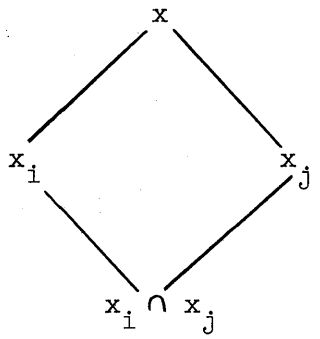


Figure 5-14a
Case 1 of a Semi Modular Lattice

Case 1

Here x covers both x_i and x_j (i.e., there exists no element x_k such that $x_i \leq x_k \leq x$ or $x_j \leq x_k \leq x$) and in a semi modular lattice of the first type this implies that x_i and x_j both cover $x_i \cap x_j$.

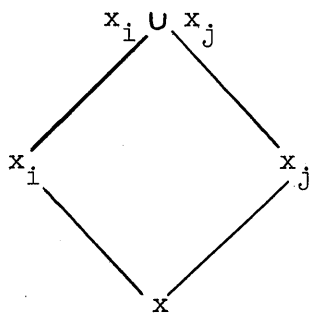


Figure 5-14b
Case 2 of a Semi Modular Lattice

Case 2

Here x is covered by both x_i and x_j (i.e., there exists no element x_k such that $x \leq x_k \leq x_i$ or $x \leq x_k \leq x_j$) and in a semi modular lattice of the second type this implies that $x_i \cup x_j$ covers both x_i and x_j.

Modular Lattice:

Definition: A lattice is modular if

$$x_i \leq x_j$$

implies that

$$x_i \cup (x \cap x_j) = (x_i \cup x) \cap x_j \tag{5-67}$$

Theorem 5. A modular lattice is doubly semi modular, i.e., it satisfied the properties of both case 1 and case 2 above.

Proof: Suppose that properly 2 is not true, i.e., that there exists an element k such that

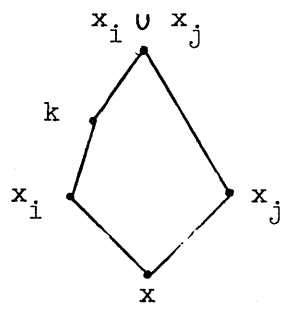


Figure 5-15
Proof of Double Semi Modular Property of Modular Lattices

$$x_i < k \leq x_i \cup x_j$$

i.e., $k \neq x_i$! Clearly x is a lower bound of x_j and k and x_j is greater than $x_j \cap k$ by definition of the $\cap$ operation:

$$x \leq x_j \cap k \leq x_j$$

There is nothing between x_j and x; i.e., x_j covers x. This means that $x_j \cap k$ must be equal to either x or to x_j.

1. Suppose that $x_j \cap k = x_j$, implying that $x_j \leq k$. But by hypothesis $x_i < k$; hence k would be an upper bound of x_i and x_j lower than $x_i \cup x_j$: this is clearly impossible by definition of $x_i \cup x_j$.

2. Now suppose that $x_j \cap k = x$. Since the lattice is modular, we have $x_i \cup (x_j \cap k) = (x_i \cup x_j) \cap k$

   i.e., $x_i \cup x = k$

   or $x_i = k$,

   but this clearly contradicts the hypothesis that $x_i \neq k$. We conclude that there cannot be any element k between x_i and $x_i \cup x_j$: The latter covers x_i. Similarly it covers x_j.

Distributive Lattice:

Definition: For a distributive lattice we have be definition

$$x_i \cap (x_j \cup x_k) = (x_i \cap x_j) \cup (x_i \cap x_k) \tag{5-68}$$

(Note similarity of this equation and the distributive property of a Boolean Algebra when $\cap \longrightarrow \cdot$ and $\cup \longrightarrow v$).

Theorem 6. In a distributive lattice

$$x_i \cup (x_j \cap x_k) = (x_i \cup x_j) \cap (x_i \cup x_k) \tag{5-69}$$

and conversely: if a lattice has this property, then it is a distributive lattice.

Proof:
$$x_i \cup (x_j \cap x_k) = [x_i \cup (x_i \cap x_k)] \cup (x_j \cap x_k) \quad \text{by (5-66)}$$

$$= x_i \cup [(x_i \cap x_k) \cup (x_j \cap x_k)] \quad \text{by (5-64)}$$

$$= x_i \cup [x_k \cap (x_i \cup x_j)] \quad \text{by (5-69)}$$

$$= [(x_i \cup x_j) \cap x_i] \cup [x_k \cap (x_i \cup x_j)] \quad \text{by (5-66)}$$

$$= (x_i \cup x_j) \cap (x_i \cup x_k) \quad \text{by (5-69)}$$

QED.

4. COMPLEMENTED LATTICES AND BOOLEAN ALGEBRA

Definition: In a lattice with 0 and 1 (e.g., every finite lattice) a complement $\bar{x}_i$ of x_i is defined by

$$\left.\begin{aligned} \bar{x}_i \cup x_i &= 1 \\ \overline{x_i} \cap x_i &= 0 \end{aligned}\right\} \tag{5-68}$$

Example: The lattice of all subsets of a set is a complemented lattice as we have seen in Chapter 3 in the discussion of Venn Diagrams.

Theorem 7. In a distributive lattice the complement is unique.

Proof: Suppose that $\bar{x}_i$ is a complement and also $\tilde{x}_i$.

Then

$$x_i \cup \bar{x}_i = 1 \qquad x_i \cup \tilde{x}_i = 1$$

$$x_i \cap \bar{x}_i = 0 \qquad x_i \cap \tilde{x}_i = 0$$

and

$$\bar{x}_i = \bar{x}_i \cap 1 = \bar{x}_i \cap (x_i \cup \tilde{x}_i)$$

$$= (\bar{x}_i \cap x_i) \cup (\bar{x}_i \cap \tilde{x}_i)$$

$$= 0 \cup (\bar{x}_i \cap \tilde{x}_i) = \bar{x}_i \cap \tilde{x}_i$$

By symmetry $\tilde{x}_i = \bar{x}_i \cap \tilde{x}_i$

therefore $\tilde{x}_i = x_i$

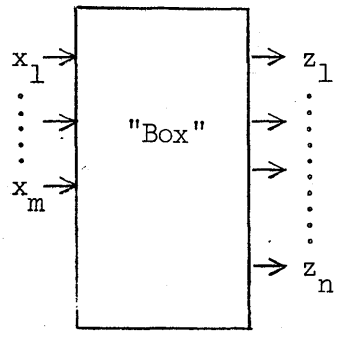
Theorem 8. A distributive complemented lattice follows the rules of Boolean Algebra.

Proof: If we replace $\cap$ by AND and $\cup$ by OR in the arguments, the theorems and remarks contain the postulates of Boolean Algebra.

5.7 Combinational Circuits, Feedback, Sequential Circuits and Timing

1. DESCRIPTION OF INTERCONNECTIONS

Definition: If the outputs of a "box" are functions (Boolean Functions) of the inputs only, then the "box" contains a combinational circuit. The relationship between inputs and outputs can be written



$$z_i = \underbrace{f_i(x_1 \ldots x_m)}_{\text{Boolean expressions}} \qquad i = 1 \ldots n$$

Figure 5-16
Circuit Notation

Take any arbitrary interconnections of AND, OR and NOT elements and possible other one-output elements as shown in Figure 5-17. Suppose, for the moment, that elements like flipflops are replaced by appropriate AND-NOT or OR-NOT combinations according to Chapter 2.

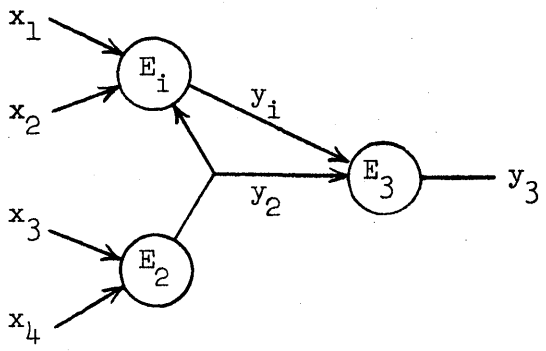


Figure 5-17
Internal Nodes

Definition: All output points (with signals y_i) are internal nodes. They can be connected to any number of inputs, but controlled by one output only. Neglecting any consideration of timing, we can write

$$y_i' = f_i(x_1, \cdots x_m, y_1, y_2, \cdots y_{i-1}, y_{i+1}, \cdots y_s) \quad (5\text{-}69)$$

where all variables of f_i are assumed to have a fixed value while y_i is computed. In order to indicate this, we write y_i' rather than y_i. Note that y_i is excluded from f_i for technical reasons: no output of an element is ordinarily supposed to be directly connected back to its input. This rule may, however, be violated and in such a case we shall simply include y_i on the right-hand side.

2. PARTIAL ORDERING OF ELEMENTS, FEEDBACK

Let $E_i \leq E_j$ mean that element E_j receives (besides direct inputs from $x_1 \ldots x_m$) only inputs from elements with $i \leq j$. It is easily seen that this convention gives a partial ordering: The laws of reflexitivity ($E_j \leq E_j$), antisymmetry ($E_j \leq E_i$, $E_i \leq E_j \longrightarrow E_i = E_j$) and transitivity ($E_i \leq E_j$, $E_j \leq E_k \longrightarrow E_i \leq E_k$) are visibly satisfied.

Definition of Feedback

A network of elements has feedback if it is possible to trace a complete loop through some sequence of elements by following the input to output direction. (Note: A circuit with feedback can still be combinational, although these circuits can always be reduced to an equivalent form not having feedback.)

Theorem 1. A circuit without feedback can be partially ordered and conversely a circuit which can be partially ordered does not have feedback.

Proof: We can describe the partial ordering as a numbering process:

1. Number 1 to j all elements having direct inputs only (x's).
2. Number j+n (n = 1, 2 ...) n elements having (besides direct inputs) as inputs only the outputs of previously numbered elements (not necessarily all of them!)
3. Suppose that at step k we find that the further numbering is impossible. Now this means that there is no element outside the set 1 to k which has inputs from one to k only. Therefore in the "non-k" set every element must have at least one input from another element in the "non-k" set. Let us start in an arbitrary node p of the "non-k" set and proceed "backwards" in the out-in direction. Because we can find always an input coming from a "non-k" set element we can trace a path step by step inside this "non-k" set. Since this set has a finite number of elements, we must come back to a node covered previously, i.e., the circuit has feedback. This contradicts the hypothesis of the theorem: therefore, the "non-k" set must be empty, i.e., we must have numbered all elements: a circuit without feedback can be partially ordered by this method; conversely, if we have partial ordering, cyclic lists are excluded (Theorem 1 of section 5.5) and we cannot have feedback.

Theorem 2. A circuit which can be partially ordered is a combinational circuit. (The converse is not true.)

Proof: Suppose that we have partially ordered the elements as above; then

$$y_1 = f_1(x_1 \ldots x_m)$$

$$y_2 = f_2(x_1 \ldots x_m, y_1)$$

$$y_3 = f_3(x_1 \ldots x_m, y_1, y_2)$$

i.e., by substitution

$$y_2 = f_2[x_1, \ldots, x_m, f_1(x_1 \ldots x_m)] = f_2'(x_1 \ldots x_m) \quad \text{etc.}$$

By continuation of this process, all node signals can be expressed as functions of the inputs only. Hence, the circuit is combinational.

Remark: A circuit may have feedback and yet be combinational.

Example: Consider the circuit in Figure 5-18. Its output is $z = x_1 x_2 \vee x_2 x_3 \vee x_1 x_3 = M(x_1 x_2 x_3)$. Such a "majority function" can actually be performed by the circuit shown in Figure 5-19. In Figure 5-18 we simply apply x_3 to two inputs of the lower majority element and it is evident that its output y simply follows x_3 quite independently of the feedback from z. But this does not contradict the fact that we have a perfectly good feedback circuit.

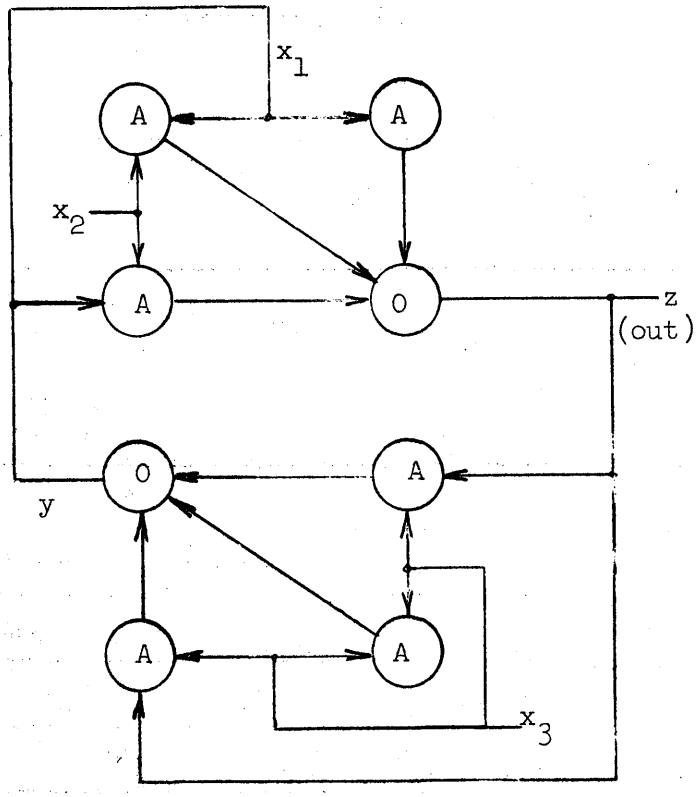


Figure 5-18
Combinational Circuit with Feedback

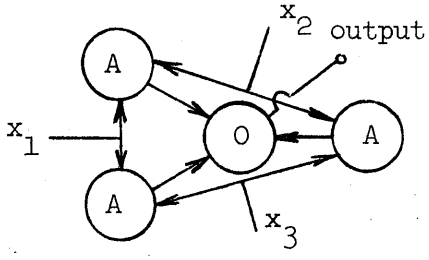


Figure 5-19
Majority Element

Sequential Circuits

There are feedback circuits in which the outputs depend on the history of the inputs: such feedback circuits are called sequential circuits.

As an example consider the circuit of a FF (see Figure 5-20): if x_1 and x_2 are both zero, this circuit still can exist in two "states": either $y_1 = 1$, $y_2 = 0$ or $y_1 = 0$ and $y_2 = 1$. Which one is "held" depends on which x was last made a one.

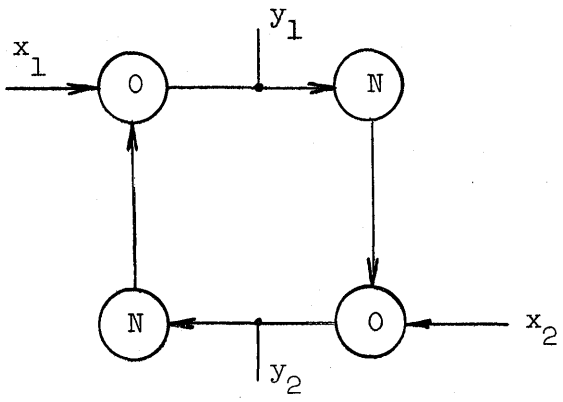


Figure 5-20
Flipflop as a Sequential Circuit

3. RACE CONDITIONS

Up to now we have not given any consideration to all the difficulties arising from the fact that logical elements produce signal-delays and that, if a circuit contains many paths, it is often very important to know which one of the possible paths reacts first. Some typical values for delays of individual elements are listed below in Table 5-5.

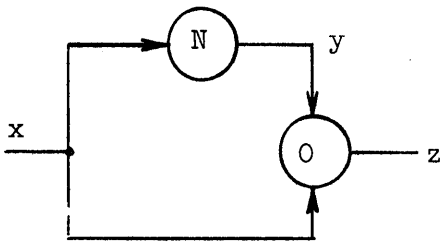
TYPICAL DELAYS IN DIFFERENT COMPUTERS

(times given in millimicroseconds)

	δ_{AND}	δ_{OR}	δ_{NOT}	δ_{FF}
	= 10^{-9} sec.			
Illiac I	250	250	700	1500
New Illiac	3	3	15	30
Fastest Known	0.3	0.3	1	2

Table 5-5

Consider as an example for timing difficulties the circuit in Figure 5-21:



assume that the delays are

$$\delta_{NOT} = \delta \neq 0.$$

$$\delta_{OR} = 0$$

and that the input goes from 1 $\longrightarrow$ 0.

Figure 5-21
Circuit with Race Conditions

If we had instantenously acting elements, we would conclude that z is identically equal to one, quite independently of the input changes. In reality we have (as illustrated in Figure 5-22 below) a critical time t_o where the upper path is not yet able to furnish a 1 (or an inverted 0) while the lower path has already taken away its (directly transmitted) 1. Consequently, the output is actually going to drop momentarily to 0 and then come up again. Such intermediate false signals called "transients" can obviously totally upset the operation of a logical circuit connected to z.

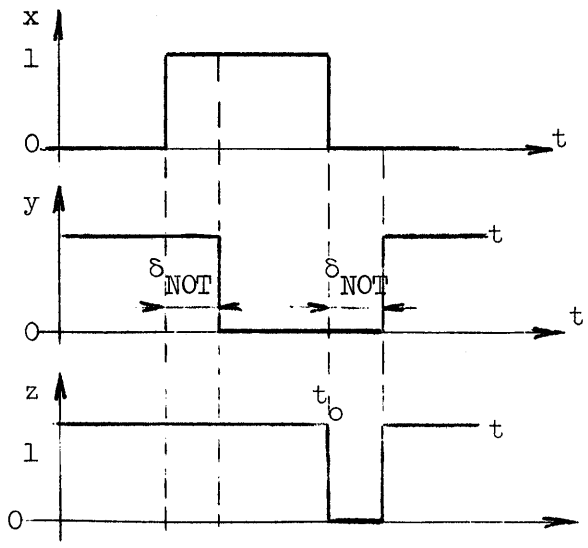


Figure 5-22
Timing in the Circuit of Figure 5-21

Theorem 3. In a combinational circuit all transients die out after a time greater than the sum of all delay times.

Proof: The elements clearly can be ordered 1 to k. Then

$$y_1 = f_1(x_1 \ldots x_m) \qquad \text{is settled after some } \delta_1$$

also

$$y_2 = f_2(x_1 \ldots x_m, y_1) \quad \text{is settled after } \delta_2' = (\delta_1 + \delta_2).$$

Generalizing all y_i's are fixed after $\delta_1 + \delta_2 \ldots \delta_m$.

4. TRANSIENT ANALYSIS BY POST ALGEBRA

Post Algebra will give us a systematic procedure to investigate dangerous race conditions in a logical circuit.

Let us define the following symbols:

"1" ... means that the signal is at the one level

"δ" ... indicates the transition from the one level to the zero level, i.e., "the signal tends to '0"

"0" ... means that the signal is permanently at the zero level

"ϵ" ... indicates the transition from the zero to the one level, i.e., "the signal tends to 1"

Actually we have, as usual, bands for the 0 and 1 signal states in practial circuits. Our definitions are shown in Figure 5-23.

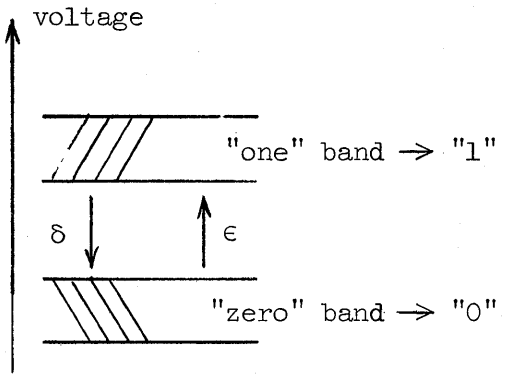


Figure 5-23
Illustration of the Meaning of 0, δ, ϵ, 1

The idea is now to examine the behavior of nodes in a network when some input is made to change. Such an input change would be the succession $0 \rightarrow \epsilon \rightarrow 1$ or $1 \rightarrow \delta \rightarrow 0$. If all nodes show allowed sequences, i.e., $0\epsilon\epsilon1$ $\delta000$ or $111\ \delta\delta0$, the circuit is safe. If, however, disallowed sequences like $1\delta1$, $0\epsilon0$ $0\delta\epsilon1$ appear, the circuit may be unsafe. Visibly disallowed sequences are those in which a rising signal is not followed by 1 and a falling signal not by a 0.

Next we attempt to establish some rules for this special algebra: Consider an AND circuit (see Figure 5-24). Its performance is described by Table 5-6.

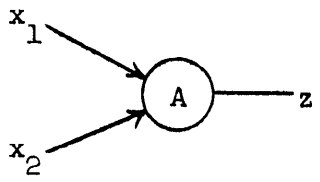


Figure 5-24 AND

x_1 \ x_2	0	δ	ϵ	1
0	0	0	0	0
δ	0	δ	δ	δ
ϵ	0	δ	ϵ	ϵ
1	0	δ	ϵ	1

Table 5-6

The justification of the table is the physical behavior of AND's, e.g., if one input rises and the other one falls, the output tends towards zero.

If we suppose that we have the ordering

$$0 < \delta < \epsilon < 1 \tag{5-70}$$

the AND Table 5-6 says: take the "smaller" one of x_1, x_2, i.e., $z = \min(x_1\ x_2)$. For example, if $x_1 = \delta$, $x_2 = \epsilon$, then $z = \delta$ from table; also $\delta < \epsilon$, hence the rule checks out.

For an OR circuit (see Figure 5-25) we have table 5-7.

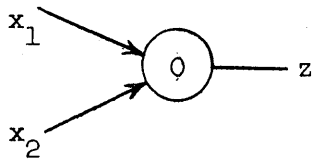


Figure 5-25 OR

x_1 \ x_2	0	δ	ϵ	1
0	0	δ	ϵ	1
δ	δ	δ	ϵ	1
ϵ	ϵ	ϵ	ϵ	1
1	1	1	1	1

Table 5-7

Again the justification is the physical behavior. With the same ordering of symbols ($0 < \delta < \epsilon < 1$) we have the rule for the OR circuit that $z = \max(x_1\ x_2)$, i.e., if $x_1 = \epsilon$, $x_2 = 1$ we have $z = \max(\epsilon, 1) = 1$.

The following rules can easily be verified by the tables.

$$
\left.
\begin{aligned}
x \vee x &= x \\
x_1 \vee x_2 &= x_2 \vee x_1 \\
(x_1 \vee x_2) \vee x_3 &= x_1 \vee (x_2 \vee x_3) \\
x_1 (x_2 \vee x_3) &= x_1 x_2 \vee x_2 x_3 \\
0 \cdot x &= 0 \\
0 \vee x &= x
\end{aligned}
\right\}
$$

NOTE: The duals of these equations are also true. (5-71)

Instead of complementation we introduce an operation called negation. Consider the operation of a NOT circuit (see Figure 5-26): The truth table for y is then given by Table 5-8.

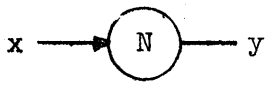


Figure 5-26 NOT

x	y
0	1
δ	ϵ
ϵ	δ
1	0

Table 5-8

As before, the physical behavior of NOT's suggests the table.

Definition of Negation by Cycling

We define cycling by the stepping forward by one unit in Figure 5-27.

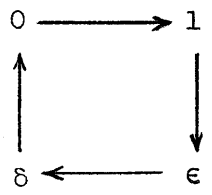


Figure 5-27 Cycling

$y = x'$ means that x is "cycled" one step. We have the rule $x'' = (x')'$, etc., and visibly $x'''' = x$.

Now we define the functions

$$\left.\begin{aligned} \varphi_1(x) &= [x' \vee x'' \vee x''']''' \\ \varphi_2(x) &= [(x \vee \epsilon)''' \vee x'']''' \\ \varphi_3(x) &= [(x \vee \epsilon)''' \vee x'']''' \end{aligned}\right\} \qquad (5\text{-}72)$$

and $y = \tilde{x}$ by

$$y = \tilde{x} = \varphi_1(x') \vee \varphi_2(x'') \vee \varphi_3(x''') \qquad (5\text{-}73)$$

Although we certainly cannot write $x\tilde{x} = 0$ and $x \vee \tilde{x} = 1$ in the general case, we still have involution and DeMorgan's theorems:

$$\left.\begin{aligned} \widetilde{(x_1 \cdot x_2)} &= \tilde{x}_1 \vee \tilde{x}_2 \\ \widetilde{(x_1 \vee x_2)} &= \tilde{x}_1 \cdot \tilde{x}_2 \\ \widetilde{(\tilde{x})} &= x \end{aligned}\right\} \qquad (5\text{-}74)$$

Example of the Use of P. A.

Consider a FF of the kind discussed in Chapter II and shown in Figure 5-28.

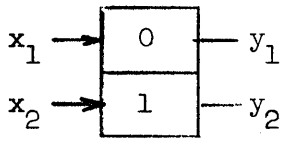


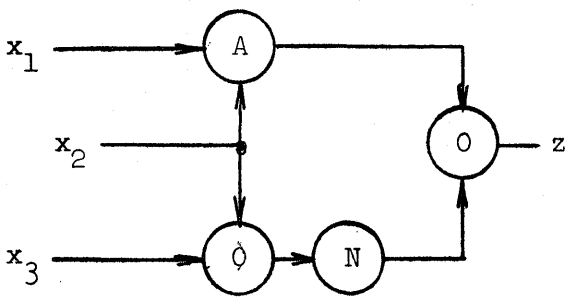
Figure 5-28 Flipflop

From the equivalent OR-NOT combination FF we find that the following sequencing table is true.

x_1	x_2	y_1	y_2	State of FF
0	0	0	1	"1"
ϵ	0	ϵ	δ	transition
1	0	1	0	"0"
δ	0	1	0	
0	0	1	0	
0	ϵ	δ	ϵ	transition
0	1	0	1	"1"
0	δ	0	1	
0	0	0	1	

Example 2

Consider the circuit of Figure 5-29 and let us examine the transition 110 → 1δ0 → 100 at the input. Assume $\delta_A = \delta_0 = 0 \neq \delta_N > 0$



x_1	$\tilde{x}_1$	x_2	$\tilde{x}_2$	x_3	$\tilde{x}_3$	$x_1 x_2$	$\tilde{x}_2 \tilde{x}_3$	z
1	0	1	0	0	1	1	0	1
1	0	δ	ϵ	0	1	δ	ϵ	ϵ
1	0	0	1	0	1	0	0	0

Figure 5-29
Circuit to Form $z = x_1 x_2 \vee \bar{x}_2 \bar{x}_3$

Since the sequence 1ϵ1 at the output is an unsafe sequence, Post Algebra warns us about a potential danger.

One method of avoiding this danger is the use of a trick(which will be discussed in more detail later): From Boolean Algebra we know that

$$x_1 x_2 \vee \bar{x}_2 \bar{x}_3 = x_1 x_2 \vee \bar{x}_2 \bar{x}_3 \vee x_1 \bar{x}_3$$

The apparently redundant $x_1 \bar{x}_3$ term actually eliminates the unsafe sequence.

Bibliography for Chapter V

(in chronological order)

1. E. L. Post: "Introduction to a General Theory of Elementary Propositions". AJM 43 (163-185) (1921)

2. B. L. Vander Waerden: "Modern Algebra". Springer. (1931)

3. G. Birkhoff: "Lattice Theory", American Mathematical Society Colloquium Publication, Vol. 25 (1948)

4. G. Birkhoff and S. MacLane: "A Survey of Modern Algebra". Macmillan. (1948)

5. M. L. Dubreil-Jacotin, L. Lesieur, R. Croisot: "Théorie des Treillis des Structures Algébriques Ordonnées et des Treillis Géométriques". Gauthier-Villars. (1953)

6. H. Hermes: "Einführung in die Verbandstheorie". Springer. (1955)

7. G. A. Metze: "Many-Valued Logic and the Design of Switching Circuits". Master's Thesis at University of Illinois. (1955)

8. E. J. McCluskey, Jr.: "Minimization of Boolean Functions". BSTJ 35/6 (Nov. 1956)

9. S. H. Caldwell: " Switching Circuits and Logical Design". Wiley. (1958)

10. D. E. Muller. Lecture Notes for "University of Illinois Math-EE 391". ("Boolean Algebras with Applications to Computer Circuits I") (1958)

11. J. P. Roth: "Algebraic Topological Methods for the Synthesis of Switching Systems, I". Transactions of the American Mathematical Society, 88/2 (July 1958)

12. J. P. Roth: "Algebraic Topological Methods in Synthesis". Proceedings of an Intern. Symposium on the Theory of Switching" Vol. 29, Annals of Computation Laboratory of Harvard University (1959)

13. T. H. Mott, Jr.: "Determination of the Irredundant Normal Forms of a Truth Function by Iterated Consensus of the Prime Implicants". IRE Transactions on EC, Vol. 9. (1960)

14. R. E. Miller: "Switching Theory and Logical Design of Automatic Digital Computer Circuits". IBM Report RC-473 (1961) Also, equivalent "University of Illinois Math-EE 394 Lecture Notes". (1960)

15. F. E. Hohn. Lecture Notes for "University of Illinois Math-EE 391". (Boolean Algebras with Applications to Computer Circuits I") (1958)

CHAPTER VI

SEQUENCING

6.1 Asynchronous, Speed Independent and Synchronous Circuits

There are three well-known ways of eliminating the unwanted effects of transients in logical circuits: asynchronous design, speed independent design and synchronous design. We shall discuss the circuit modifications to be made to the circuit of Figure 5-27 in the three methods.

1. ASYNCHRONOUS DESIGN

Principle: We introduce artificial delays which ensure the proper signal timing by creating delay asymmetry in the circuit. In the example of the last section we choose a delay Δ in the upper path such that $\Delta \gg \delta_{NOT}$. This will certainly eliminate the transient in the 1,1,0 → 1,0,0 → 1,0,0 input transition. Unhappily this very modification now introduces a transient in the 1,0,0 → 1,0,0 → 1,1,0 input transition (all other input transitions are safe!).

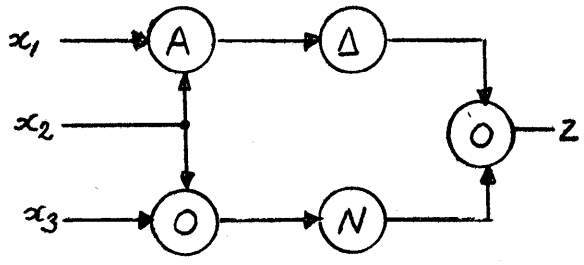


Figure 6-1
Asynchronous Modification of the Circuit in Figure 5-27

2. SPEED INDEPENDENT DESIGN (MULLER THEORY)

Principle: We provide additional internal paths and interlocks which make the output transition independent of the relative speeds of the elements inside.

As noted before $x_1x_2 \vee \bar{x}_2\bar{x}_3 = x_1x_2 \vee \bar{x}_2\bar{x}_3 \vee x_1\bar{x}_3$; hence the circuit is modified to have the configuration of Figure 6-2. It can be proved that no transients will appear for any input transition starting with a steady value.

Note: Depending on the input sequence, different subsets of the circuit control the output, and this control path operates in a

sequential fashion, i.e., one can often say that in a speed independent circuit the effective topology varies with the input. In our example only "box 1" is used for the 110 → 100 transition, only "box 2" for the 111 → 011 transition, etc.

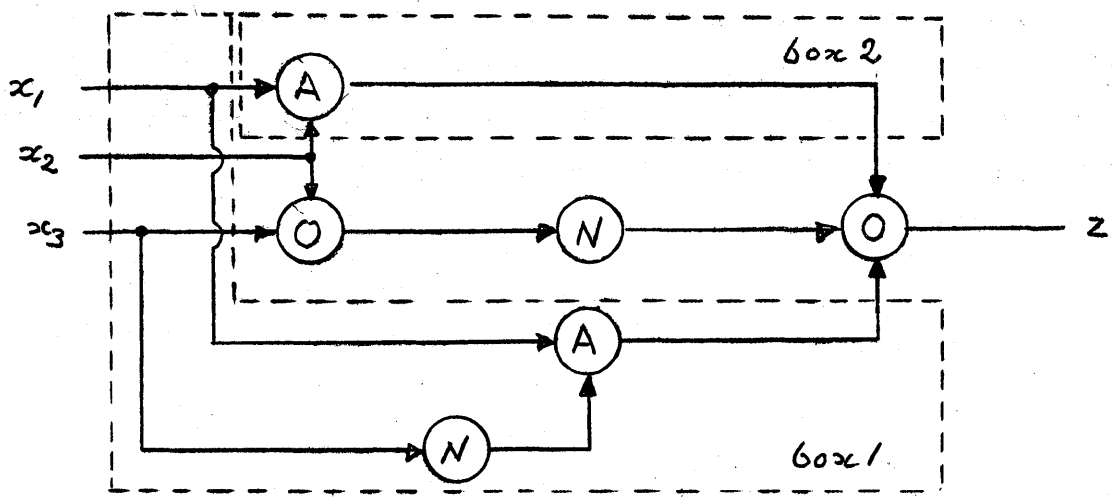


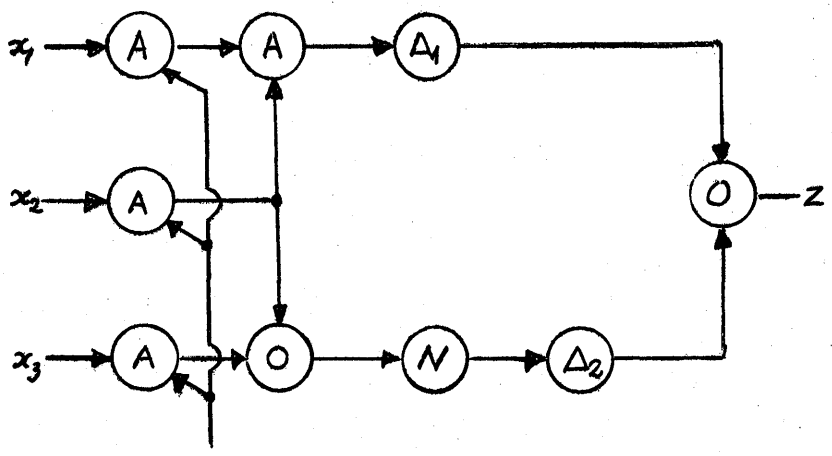
Figure 6-2

Speed Independent Modification of the Circuit in Figure 5-27

3. SYNCHRONOUS DESIGN

Principle: Inputs are periodically strobed by a clock signal c and internal paths are provided with compensating delays so that each subsection of the circuit produces exactly one clock period (T_{clock}) delay.

Note: This method in general presupposes very close time tolerances. It might actually be necessary to strobe the outputs by a second set of AND's.



In this design we choose therefore

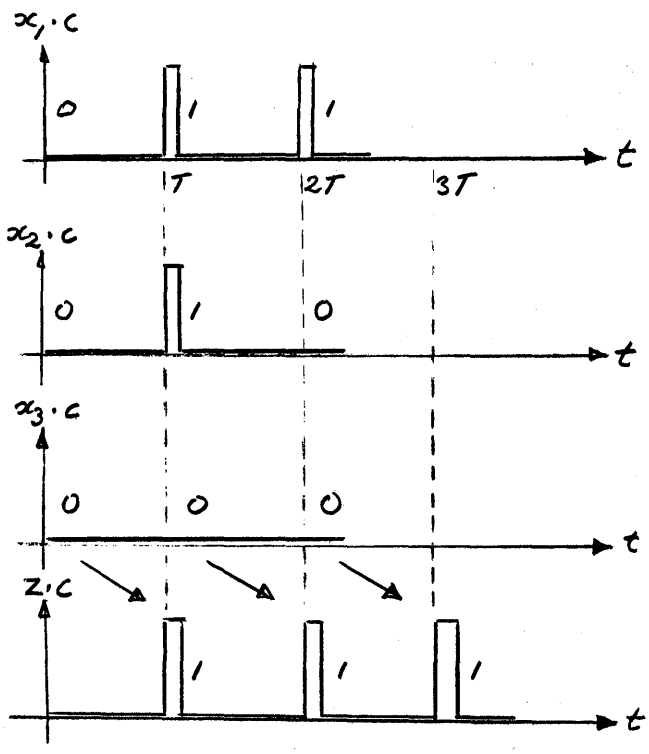
$\Delta_1 = \delta_{NOT} + \Delta_2 = T_{clock}$

(assuming that $\delta_{AND} = \delta_{OR} = 0$).

Figure 6-3

Synchronous Modification of the Circuit in Figure 5-27

As an example consider the $1,1,0 \rightarrow 1,0,0$ transition:



$z(1,1,0) = 1 \cdot 1 \vee 0 \cdot 1 = 1$

$z(1,0,0) = 1 \cdot 0 \vee 1 \cdot 1 = 1$

when the output appears, i.e., one clock period after the input was applied:

$z[(\nu+1)T] = f[x_1(\nu T), x_2(\nu T), x_3(\nu T)]$.

Figure 6-4
Timing in the Circuit of Figure 6-3

Remark: The example above shows that a synchronous combinational circuit behaves in many ways like a sequential circuit. The present output depends on previous inputs, i.e., there is memory. There is, however, one important difference: the initial internal state is completely "flushed out" after ν clock pulses, where ν is the maximum number of layers of logic between input and output. The initial internal state of a synchronous sequential circuit can influence the output for any length of time.

4. THEOREMS ABOUT SYNCHRONOUS CIRCUITS

Theorem 1. Any logical circuit can be converted into an equivalent synchronous circuit.

Proof. We shall show that an arbitrary subsection can be converted. Then the theorem follows by converting the finite number of subsections of the original circuit one by one.

Consider the subsection shown in Figure 6-5.

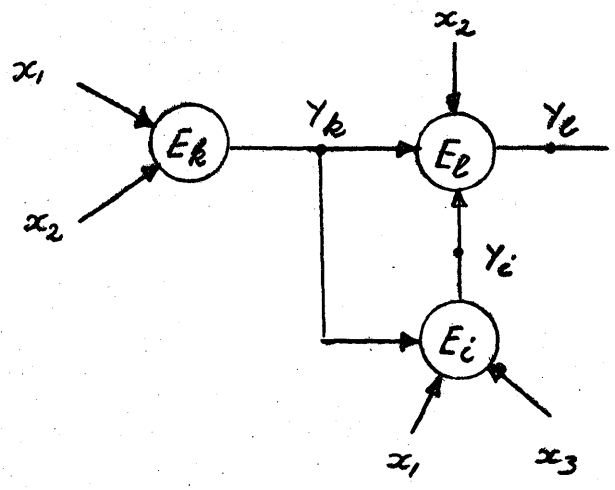


Figure 6-5 Subsection of a Logical Circuit

The equivalent synchronous circuit is shown in Figure 6-6.

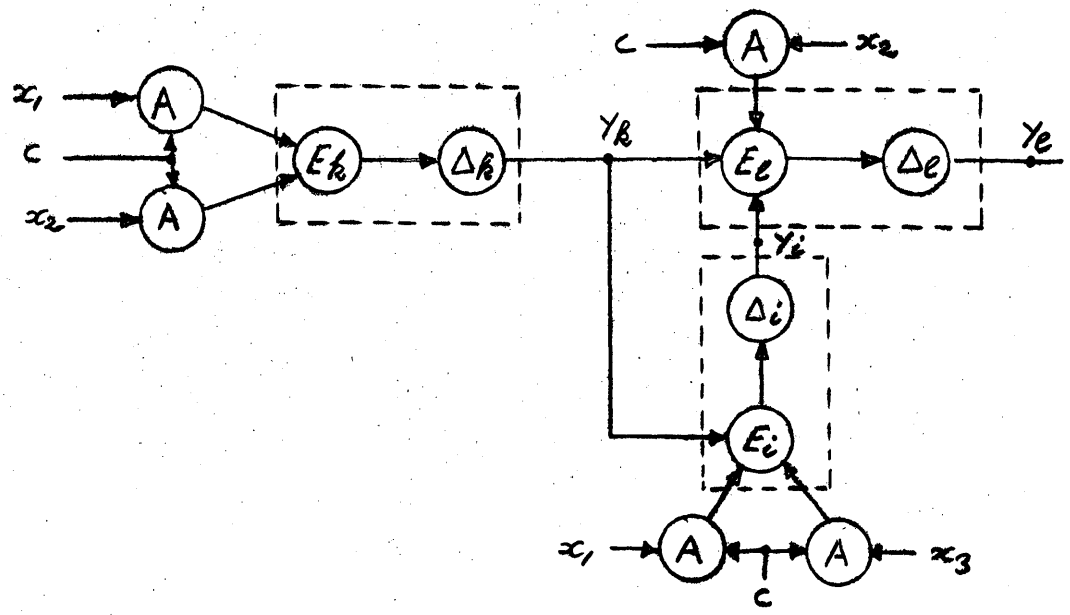


Figure 6-6 Synchronous Equivalent of the Subsection in Figure 6-5

The additional delays Δ are chosen such that (δ_k being the inherent delay of E_k, etc.)

$$\delta_k + \Delta_k = \delta_i + \Delta_i = \delta_\ell + \Delta_\ell = T = \text{clock period} \qquad (6\text{-}1)$$

Furthermore, we introduce strobing AND circuit whenever direct inputs occur.

New Definition of y'

In synchronous circuits we shall designate by y' the value one clock period later rather than the "value if all arguments were fixed" as we did in the last chapter:

$$y_i'(\nu T) = y[(\nu+1)T] = f_i[x_1(\nu T), \ldots, y_s(\nu T)] \tag{6-2}$$

In the example we have

$$y_i'(T) = f_i[x_1(0), x_3(0), y_k(0)]$$

$$y_k'(T) = f_k[x_1(0), x_2(0)]$$

$$y_\ell'(T) = f_\ell[x_2(0), y_k(0), y_i(0)]$$

The clock period will be dropped from the argument in what follows, i.e., we shall write

$$y(\nu T) \equiv y(\nu) \tag{6-3}$$

If now we assume that the initial state is known, i.e., that all signal values of the internal nodes are given at t = 0 (as well as the inputs), we have

$$y_k(1) = y_k'(0) = f_k[x_1(0), x_2(0)]$$

$$y_i(1) = y_i'(0) = f_i[x_1(0), x_3(0), y_k(0)]$$

and $$y_\ell(1) = y_\ell'(0) = f_\ell[x_2(0), y_k(0), y_i(0)]$$

From the y's at time T and the input at T (i.e., the set $y_k(1)$, $y_i(1)$, $y_\ell(1)$ and $x_1(1)$, $x_2(1)$, etc.) we can calculate the next motions of the circuit, i.e., the internal states at time 2T, etc.

At this point it becomes convenient to introduce vectors to represent input combinations, the set of signals at the internal nodes and finally the output combinations. These vectors may be thought of as column vectors, although other interpretations may be useful.

Definition: Let $S = 2^s$, $M = 2^m$ and $N = 2^n$ where s is the number of internal nodes, m the number of input lines and n the number of output lines. If $x_{ii} \ldots x_{mi}$ is a given input, $y_{ij} \ldots y_{sj}$ a given internal state

and $z_{1k} \ldots z_{nk}$ a given output, we define <u>input vectors</u> X_i, <u>state vectors</u> Y_j, and <u>output vectors</u> Z_k by

$$\left.\begin{aligned} X_i &= \{x_{1i}, \ldots, x_{mi}\} \qquad i = 1 \ldots M \\ Y_j &= \{y_{1j}, \ldots, y_{sj}\} \qquad j = 1 \ldots S \\ Z_k &= \{z_{1k}, \ldots, z_{nk}\} \qquad k = 1 \ldots N \end{aligned}\right\} \tag{6-4}$$

If we want to indicate values at clock period ν we may also write

$$\left.\begin{aligned} X(\nu) &= \{x_1(\nu), \ldots, x_m(\nu)\} \\ Y(\nu) &= \{y_1(\nu), \ldots, y_s(\nu)\} \\ Z(\nu) &= \{z_1(\nu), \ldots, z_n(\nu)\} \end{aligned}\right\} \tag{6-5}$$

Symbolically our equations (6-2) or the equivalent (5-69) can now be written

$$Y' = Y(\nu + 1) = F[X(\nu), Y(\nu)] \tag{6-6}$$

We see, then, how successive internal states can be constructed iteratively. It is, however, not clear how the outputs can be obtained. It is sometimes useful to consider the z's as forming together with the y's a new set of nodes, but this can lead to confusion. We shall therefore until further notice make the following

<u>Convention</u>: It will be assumed that we have an instantaneous decoder (i.e., an infinitely fast combinational circuit) which forms the z's either from the y's alone (Moore machine) or from the y's and x's (Mealy machine). Symbolically

$$Z(\nu) = G[Y(\nu)] \qquad \text{(Moore)} \tag{6-7}$$

or

$$Z(\nu) = H[X(\nu), Y(\nu)] \qquad \text{(Mealy)} \tag{6-8}$$

Theorem 2. In any synchronous circuit (combinational or not) the initial condition of all internal nodes and the sequence of all inputs determine uniquely the output behavior.

Proof. (6-6) shows how we can calculate Y(n) recursively:

$$Y(1) = F[X(0),\ Y(0)]$$

$$Y(2) = F\{X(1),\ Y(1)\} = F\{X(1),\ F[X(0),\ Y(0)]\}\ ,\ \text{etc.}$$

Y(0) and the sequence X(0), X(1), etc., therefore determine Y(n). Similarly this sequence also determines Z(n) since Z(n) = G[Y(n)] or H[X(n), Y(n)].

Theorem 3. If the inputs are held fixed in a synchronous circuit of s elements, the outputs show, after not more than $S = 2^s$ clock periods, constant signals or a cyclical behavior of the outputs, the cycle length being $\leq 2^s$ clock periods.

Proof. Assume that we start from Y(0), which is one of the possible internal states $Y_1 \ldots Y_S$. Let us wait S clock periods: we have then gone through S + 1 states, i.e., a certain state Y(ν) must have occurred twice. Since the output X is constantly equal to X(0) we shall therefore have the condition X(0), Y(ν) both at clock period ν and again before S . From this point onwards everything is repeated by virtue of (6-6). It is also obvious that the cycle length is $\leq S = 2^s$ periods. (6-7) or (6-8) then show that the output is cyclic with a cycle length of $\leq 2^s$ periods. The case where Z is constant simply corresponds to a cycle length of one period.

Theorem 4. Any system $y_i' = f_i(x_1, \ldots, x_m, y_1, \ldots, y_s)$, i.e., $Y(\nu + 1) = F[X(\nu), Y(\nu)]$ can be realized when infinitely fast combinational logic is available (or the clock period is made long enough).

Proof: Figure 6-7 below shows how one can simply delay combinational outputs (formed in a time short compared to the clock period) by one clock period before feeding them back into the circuit. The combinational circuit can obviously be made to give instaneously all $f_i(x_1, \ldots, x_m, y_1, \ldots y_s)$ functions. The description of its operation with delays inserted is clearly $y_i' = f_i$.

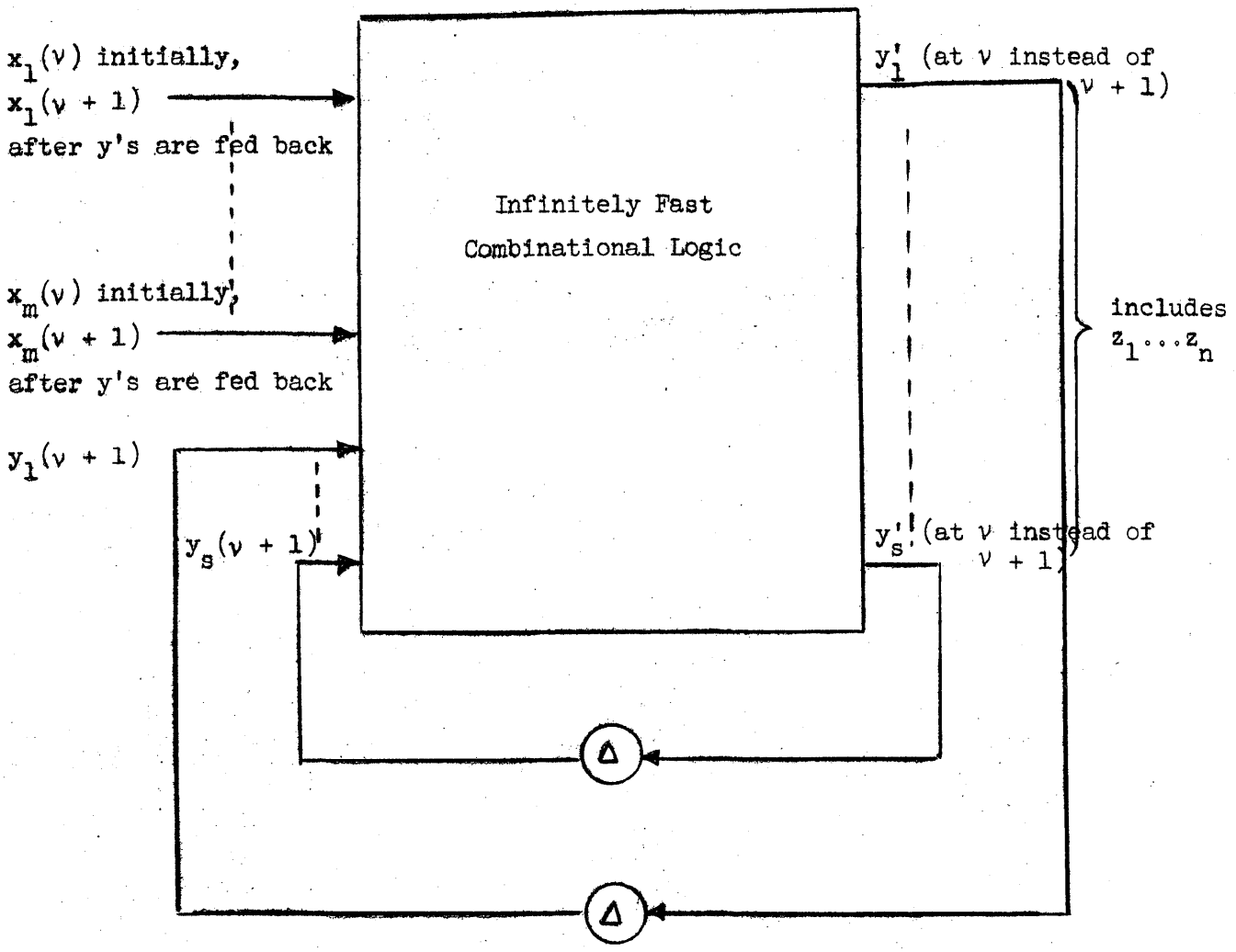


Figure 6-7 Model of a General Synchronous System

6.2 State Diagrams and Matrices for Synchronous Circuits

1. STATE DIAGRAMS

A state diagram is a linear graph in which circles represent the set (or a subset) of states $Y_1,\ldots,Y_S$ and arrows the transitions under given input conditions $X_1,\ldots,X_M$ (usually written next to these arrows). The state diagram may also contain information about the outputs $Z_1,\ldots,Z_N$: in the first two examples, however, we are going to neglect the outputs.

Example 1. Suppose that a simple minded animal has the three states "Eating" (Y_1), "Awake" (Y_2), and "Asleep" (Y_3) and that the transitions are cause by four inputs "Darkness" (X_1), "Light" (X_2), "Stomach full" (X_3), and "Smell of food" (X_4). Then Figure 6-8 shows a possible state diagram.

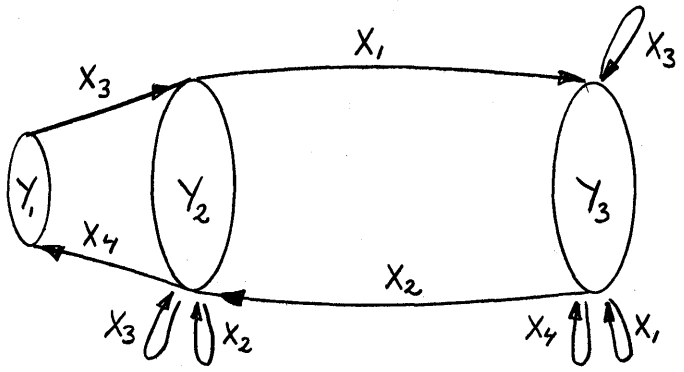


Figure 6-8 State Diagram for a Simple-minded Animal

It is to be noted that transitions only occur periodically (when the "brain" scans the sensory perceptions) and also that as long as it is light, food will be absorbed, except when the stomach is full.

Remark: It happens quite often that the complement of the signals at internal nodes is available. If this complement is formed instantaneously (as in the two outputs of an Eccles Jordan flipflop) it is not necessary to introduce separate nodes for the complement. One sometimes expresses this by saying that we only have to choose the cardinal points, i.e., a subset of outputs inside the circuit which completely define the state.

Example 2. Consider the system shown in Figure 6-8 formed of three flipflops, various gates and an input x_1. The internal nodes are 1, 2 and 3 with signals y_1, y_2, y_3. To simplify the argument we shall suppose that $\bar{y}_1$, $\bar{y}_2$, $\bar{y}_3$ are also available. We shall also assume that each flipflop has a time delay equal to the clock period and that other element act instantaneously.

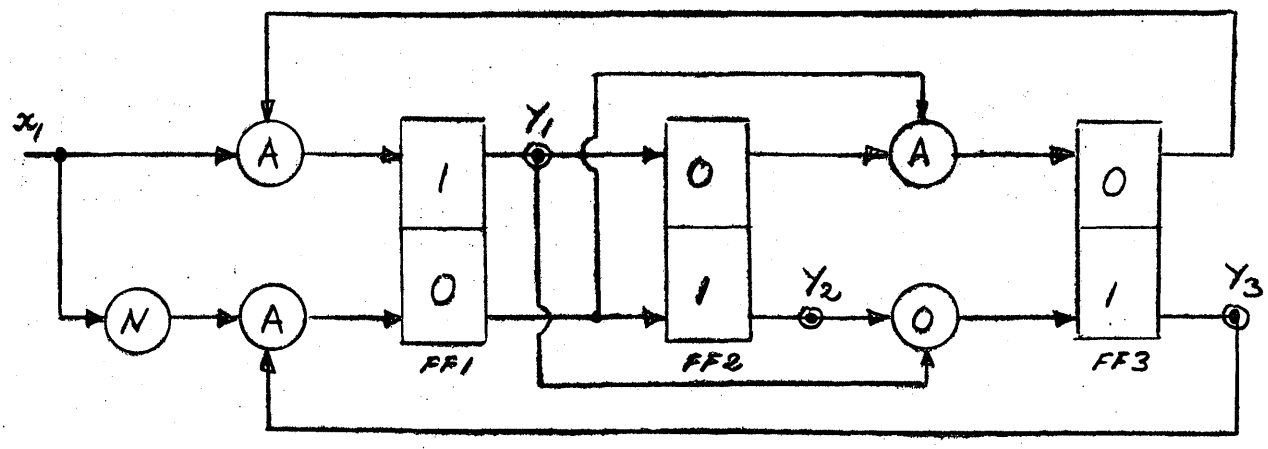


Figure 6-9 Three Flipflop System

The "setting" equations of the FF's (i.e., the combinations setting the flipflops to 1 and to zero respectively) are:

for FF_1 $f_1(y_1) = x_1\bar{y}_3$ $f_0(y_1) = \bar{x}_1 y_3$

FF_2 $g_1(y_2) = \bar{y}_1$ $g(y_2) = y_1$

FF_3 $h_1(y_3) = y_1 \vee y_2$ $h_0(y_2) = \bar{y}_1\bar{y}_2$

Now we are in the position to construct the transition table (see Table 6-1) of the system. The question is: given a state, to which next state will the system go for the condition $x_1 = 0$ and for the condition $x_1 = 1$. It should be noted that care has been taken in the design of the circuit to avoid "1,1" inputs to flipflops.

Although we can simply operate with states designated by (0,0,0), (0,0,1) and inputs 0 and 1 respectively, our example can be converted to more general notation by calling X_1 the input 0 and X_2 the input 1. Similarly we can assign the names $Y_1 \ldots Y_8$ to (0,0,0) through (1,1,1). The relationship between Y and Y' for the conditions X_1 and X_2 are then expressed by the Huffman Flow Table given in Table 6-2. Figure 6-10 gives two equivalent forms of the state diagram.

TABLE 6-1

x_1	y_1	y_2	y_3	$x_1\bar{y}_3$	$\bar{x}_1 y_3$	$\bar{y}_1$	y_1	$y_1 \vee y_2$	$\bar{y}_1\bar{y}_2$	y'_1	y'_2	y'_3
0	0	0	0	0	0	1	0	0	1	0	1	0
1	0	0	0	1	0	1	0	0	1	1	1	0
0	0	0	1	0	1	1	0	0	1	0	1	0
1	0	0	1	0	0	1	0	0	1	0	1	0
0	0	1	0	0	0	1	0	1	0	0	1	1
1	0	1	0	1	0	1	0	1	0	1	1	1
0	0	1	1	0	1	1	0	1	0	0	1	1
1	0	1	1	0	0	1	0	1	0	0	1	1
0	1	0	0	0	0	0	1	1	0	1	0	1
1	1	0	0	1	0	0	1	1	0	1	0	1
0	1	0	1	0	1	0	1	1	0	0	0	1
1	1	0	1	0	0	0	1	1	0	1	0	1
0	1	1	0	0	0	0	1	1	0	1	0	1
1	1	1	0	1	0	0	1	1	0	1	0	1
0	1	1	1	0	1	0	1	1	0	0	0	1
1	1	1	1	0	0	0	1	1	0	1	0	1

Signal Changes in the Circuit of Figure 6-9

TABLE 6-2

Y	Y'	
	for X_1	for X_2
Y_1	Y_3	Y_7
Y_2	Y_3	Y_3
Y_3	Y_4	Y_8
Y_4	Y_4	Y_4
Y_5	Y_6	Y_6
Y_6	Y_6	Y_6
Y_7	Y_6	Y_6
Y_8	Y_2	Y_6

Huffman Flow Table for the Circuit of Figure 6-9

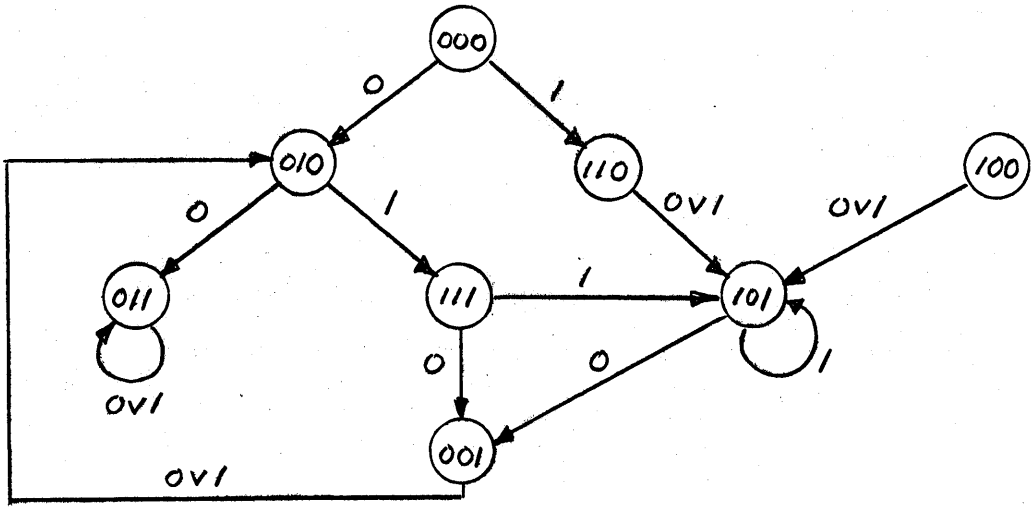


Figure 6-10a Unencoded State Diagram

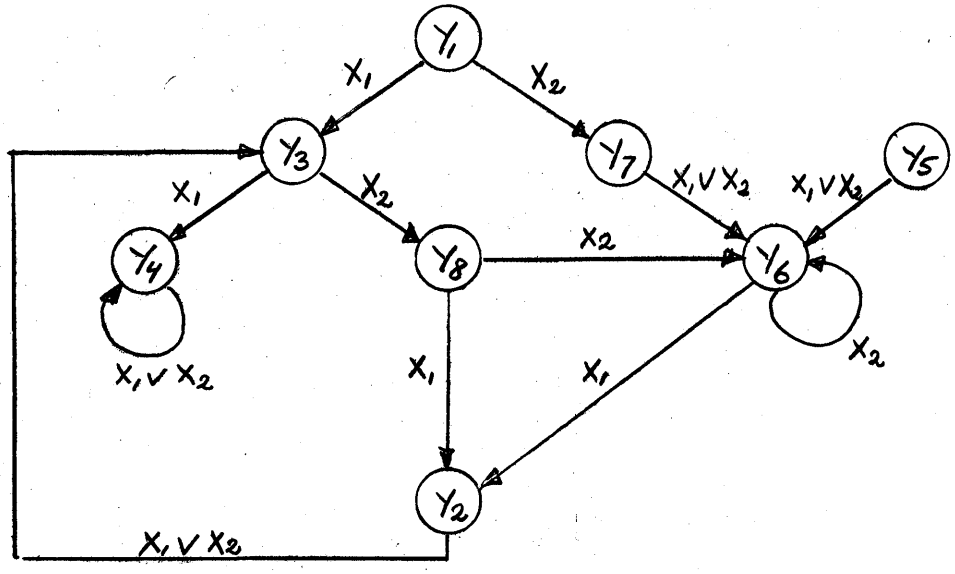


Figure 6-10b Encoded State Diagram

We see now that the circuit admits two types of cycles $(0,1,0) \rightarrow (1,1,1) \rightarrow (0,0,1) \rightarrow (0,1,0)$ ... for the input sequence $1,0,(\overset{1}{\underset{0}{\text{or}}}),1\ldots$ and $(0,1,0) \rightarrow (1,1,1) \rightarrow (1,0,1) \rightarrow \ldots (1,0,1) \rightarrow (0,0,1) \rightarrow (1,0,1)$ for the input sequence $1,1 \ldots 1,0,(\overset{1}{\underset{0}{\text{or}}}),1 \ldots$, ie., the input sequences which produce cycling are $1,0,x,1, \ldots 1,0,x,1, \ldots$ and $1,0,x,1,0, , \ldots$.

Rules for State Diagrams (including output information)

1. From a given state draw arrows to all states which can be attained by the allowed inputs in that state.

2. In case a state does not admit all inputs two cases are possible: either certain inputs are actually prohibited or the transition is simply not defined for these inputs. If the latter is the case, it is allowed to define the transitions for these inputs.

3. Every input must lead to a well defined state, i.e., the machine must be deterministic.

4. In the so-called Moore-model it is assumed that the outputs depend only on the state. In such cases we write inside each circle representing a state the corresponding output, i.e., terms of the form Y_i/Z_k (called state/output pairs).

5. In the so-called Mealy-model it is assumed that the outputs depend on the state and the input. In such cases we write next to each arrow the input and the corresponding output, i.e., terms of the form X_j/Z_k (called input/output pairs). If several inputs cause a given transition, we write an OR-sum.

2. DESCRIPTION OF MACHINE BEHAVIOR

The following items are necessary to describe completely how a machine behaves:

1. A list of
   a) Inputs $X_1, \ldots X_M$
   b) Internal states $Y_1, \ldots Y_S$
   c) Outputs $Z_1, \ldots Z_n$

   where $X = \{x_1, \ldots, x_m\}$
   $Y = \{y_1, \ldots, y_s\}$
   $Z = \{z_1, \ldots, z_n\}$ as discussed in (6-5).

2. A transition map T written symbolically

$$T \rightarrow \{Y_i \ldots X_k \rightarrow Y_\ell\} \tag{6-9}$$

   and usually given in the form of a Hoffman flow table (see Table 6-2). This is a matrix form of the function F in (6-6).

3. An output map Ω.

a) For a Moore machine, Ω is denoted symbolically by

$$\Omega \rightarrow \{Y_i \rightarrow Z_\ell\} \tag{6-11}$$

This map is usually written in the form of a column vector associated with $\{Y_1,\ldots,Y_S\}$. It is the matrix form of G in (6-7).

b) For a Mealy machine, Ω is written symbolically

$$\Omega \rightarrow \{Y_i \cdot X_k \rightarrow Z_\ell\} \tag{6-10}$$

It can take the form of addition columns in the Huffman flow table or of a set of output vectors $\Omega_1 = \{Z_{11}Z_{12} \ldots Z_{1S}\}$, $\Omega_2 = \{Z_{21}Z_{22} \ldots Z_{2S}\}$, etc., associated with the states $Y_1,\ldots,Y_S$ and the possible inputs $X_1,\ldots,X_M$. Here we have the matrix representation of H in (6-8).

Remark 1: In an incompletely specified machine some of the outputs or transitions are not defined. We then write a hyphen in the corresponding position, except if the transition is prohibited: we then write a 0.

We shall only discuss Mealy machines in all that follows. In order to facilitate the discussion, we shall now introduce two further types of matrices.

The Connection Matrix (Hohn)

The general element c_{ij} of the connection matrix

$$C = [c_{ij}] \tag{6-12}$$

is given by

$$c_{ij} = X_a/Z_\alpha \vee X_b/Z_\beta \vee \ldots \tag{6-13}$$

where X_a, X_b ... are the inputs that produce a transition from state Y_i to state Y_j and Z_α ... the corresponding outputs. Note that we are only talking about Mealy machines.

The Transition Matrices

With the input X_k we shall associate a transition matrix

$$T^k = [t^k_{ij}] \tag{6-14}$$

where

$$t^k_{ij} = \begin{cases} 0 \text{ if } X_k \text{ does not produce transition } Y_i \to Y_j \\ \\ 1 \text{ if } X_k \underline{\text{does}} \text{ produce transition } Y_i \to Y_j \end{cases} \tag{6-15}$$

It is to be noted that due to the deterministic behavior of machines a transition matrix has only <u>one</u> 1 in each row.

<u>Example</u>. Take a Mealy machine with a state diagram as shown in Figure 6-11.

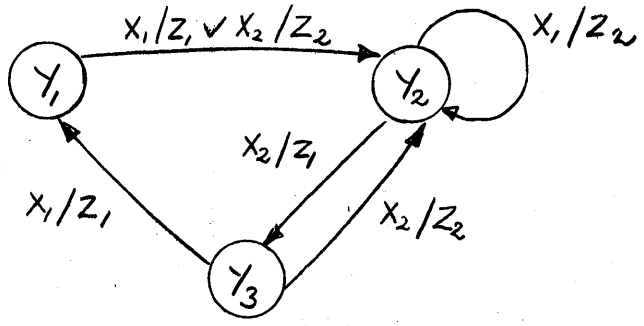


Figure 6-11 Mealy Machine

Here we have the connection matrix

$$C = \begin{bmatrix} 0 & (X_1|Z_1 \vee X_2|Z_2) & 0 \\ 0 & X_1|Z_2 & X_2|Z_1 \\ X_1|Z_1 & X_2|Z_2 & 0 \end{bmatrix}$$

and from the definition

$$T^1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad ; \quad T^2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Furthermore, the output vectors for X_1 and X_2 respectively are

$$\Omega_1 = \begin{bmatrix} Z_1 \\ Z_2 \\ Z_1 \end{bmatrix} \quad ; \quad \Omega_2 = \begin{bmatrix} Z_2 \\ Z_1 \\ Z_2 \end{bmatrix}$$

Theorem 1. Let X_i be an input and T^i the corresponding transition matrix. Let Z_{ij} be the output for input X_i and internal state Y_j (i.e., the Z_{ij}'s are amongst $\{Z_1 \ldots Z_N\}$). Form the diagonal matrix

$$U_i = \begin{bmatrix} X_i/Z_{i1} & & & 0 \\ & X_i/Z_{i2} & & \\ & & \ddots & \\ 0 & & & X_i/Z_{iS} \end{bmatrix} \tag{6-16}$$

Then the connection matrix C is given by

$$C = \sum U_i T^i \tag{6-17}$$

where the sum is to be interpreted as an OR-sum.

Proof. The process described above takes T^i and multiplies its successive rows by X_i/Z_{i1}, X_i/Z_{i2}, etc. Suppose that T^i has a 1 in position mn, then the product will give X_i/Z_{im}. If any other input X_k also has a 1 in position mn, we add X_k/Z_{km}, i.e., if inputs X_i, X_k ... give the transition $Y_n \rightarrow Y_m$ we shall find in $\sum X_i/Z_{im} \vee X_k/Z_{km}$ .... Visibly the outputs are precisely those that correspond to the inputs and state Y_m, i.e., those we would expect in the connection matrix.

Example. Take a machine with a state diagram as shown in Figure 6-12. Form C according to (6-17).

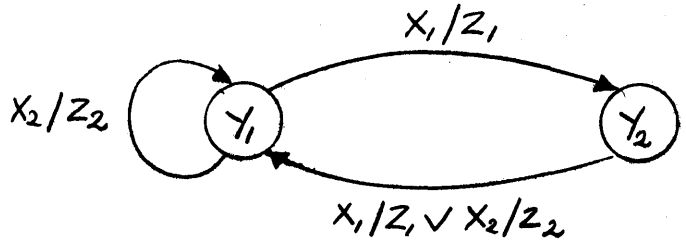


Figure 6-12 Example of a Mealy Machine

$$C = \begin{bmatrix} X_1/Z_{11} & 0 \\ 0 & X_1/Z_{12} \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \vee \begin{bmatrix} X_2/Z_{21} & 0 \\ 0 & X_2/Z_{22} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

where the second factors are simply the transition matrices T^1 and T^2 corresponding to X_1 and X_2. Since

$$Z_{11} = (\text{output for } X_1 \text{ in state } Y_1) = Z_1$$
$$Z_{12} = (\text{output for } X_1 \text{ in state } Y_2) = Z_1$$
$$Z_{21} = (\text{output for } X_2 \text{ in state } Y_1) = Z_2$$
$$Z_{22} = (\text{output for } X_2 \text{ in state } Y_1) = Z_2$$

as is seen from the state diagram, we can write

$$C = \begin{bmatrix} X_2/Z_2 & X_1/Z_1 \\ X_1/Z_1 \vee X_2/Z_2 & 0 \end{bmatrix}$$

This is indeed the connection matrix as shown by direct inspection.

Remark: If a row of zeros occurs in C in row m it simply means that we cannot leave the corresponding state Y_m.

Theorem 2. The product of two transition matrices is a possible transition matrix itself, i.e., at most a single 1 can occur in a given row.

Proof. By definition of a matrix product, the element mn of the product of two transition matrices T^i and T^j is given by

$$(T^i T^j)_{mn} = \Sigma\, t^i_{m\lambda} t^i_{\lambda n}$$

Now row m of T^i has a single 1 at most: let it occur in column λ (a particular value of λ). Consider now column n of T^j: if and only if it has a 1 in position $t_{\lambda n}$ we shall obtain a 1 in position mn of the product. Now consider an element mp with $p \neq n$ in row m: The element $t_{m\lambda}$ giving a 1 is still the same. Let us look at column p of T^j: if again the 1 occurred in position λp we would have a second 1 in row m of the product. But this would imply that T^j had a 1 in position $t_{\lambda n}$ and $t_{\lambda p}$, i.e., two 1's in the same row. This being impossible, the theorem is proved.

Theorem 3. The resulting state after the input sequence $X_1 \ldots X_k$ is applied to a machine initially in state Y_ν, is given by the νth row of the column vector Y^*.

$$\left.\begin{aligned} Y^* &= T^1 \cdot T^2 \ldots T^k \cdot Y \\ \text{where} \quad Y &= \begin{bmatrix} Y_1 \\ \vdots \\ Y_S \end{bmatrix} \end{aligned}\right\} \qquad (6\text{-}18)$$

Proof. For a single transition we have

$$Y^* = T^1 \cdot Y = \begin{bmatrix} \sum_{j=1}^{S} t^1_{1j} Y_j \\ \vdots \\ \sum_{j=1}^{S} t^1_{Sj} Y_j \end{bmatrix}$$

Suppose now that we start from Y_ν and that $t^1_{\nu\lambda} = 1$, i.e., that under the influence of X_1, $Y_\nu \rightarrow Y_\lambda$. Then our assertion is that

$$\sum_{j=1}^{S} t^1_{\nu j} Y_j = 0 \vee \ldots \vee t^1_{\nu\lambda} Y_\lambda \vee \ldots 0 = Y_\lambda$$

which is correct. Now for two transitions we have

$$Y^* = T^1 \cdot T^2 \cdot Y = \begin{bmatrix} \sum_\lambda \sum_j t^1_{1\lambda} t^2_{\lambda j} Y_j \\ \vdots \\ \sum_\lambda \sum_j t^1_{S\lambda} t^2_{\lambda j} Y_j \end{bmatrix}$$

Suppose that we start in state Y_ν and that under the influence of X_1, $Y_\nu \rightarrow Y_\eta$, i.e., that $t^1_{\nu\eta} = 1$. Also suppose that under the influence of X_2, $Y_\eta \rightarrow Y_\mu$, i.e., that $t^2_{\eta\mu} = 1$. Then obviously $Y_\nu \rightarrow Y_\mu$ under the influence of X_1, followed be X_2.

Consider the νth row of $Y^* = T^1 T^2 Y$, i.e.,

$$\sum_{\lambda=1}^{S} \sum_{j=1}^{S} t^1_{\nu\lambda} t^2_{\lambda j} Y_j = t^1_{\nu\eta} \cdot t^2_{\eta\mu} Y_\mu \vee 0 \vee 0 \vee 0 \ldots$$

$$= 1 \cdot 1 \cdot Y_\mu = Y_\mu$$

Again this agrees with our direct observation. Clearly this method can be generalized to any number of transitions, i.e., the theorem is true.

Remark: If Y^* has a zero, it means that for the given input sequence the result is not specified. One says that such input sequences are not allowed.

Example. Take a state diagram according to Figure 6-13, neglecting outputs.

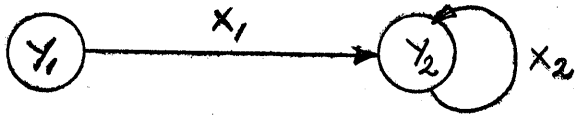


Figure 6-13 Simple Machine

Here $$T^1 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad T^2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$T^1T^2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \text{and therefore}$$

$$T^1T^2Y = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} Y_2 \\ 0 \end{bmatrix}$$

meaning that if we start in state Y_1, input sequence X_1X_2 leads us to state Y_2, while if we start in Y_2 the result is unspecified.

3. INPUT/OUTPUT POLYNOMIALS. SEQUENCES OF GIVEN LENGTH

Since we have inputs $X_1 \ldots X_M$ and outputs $Z_1 \ldots Z_N$ there are $M \cdot N$ possible input/output pairs of the form X_i/Z_k. Let us number them $P_1 \ldots P_\alpha \ldots P_\beta \ldots P_{MN}$.

Definition: An input/output polynomial of degree r is the OR-sum of products having each r factors taken from the set $P_1 \ldots P_{MN}$.

It is natural to interpret the product $P_\alpha P_\beta = (X_i/Z_k)(X_j/Z_\ell)$ (say) as meaning that we consider the input sequence X_iX_j and that the observed output sequence was Z_kZ_ℓ. A product of length r then corresponds to a definite

input sequence involving r-inputs and a definite output sequence involving r-outputs, i.e., each term of the OR-sum in the input/output polynomial represents a certain input/output sequence of "length r."

In a similar way we shall interpret the OR-sum $P_\alpha \vee P_\beta = (X_i/Z_k) \vee (X_j/Z_\ell)$ as meaning that input X_i or input X_j has been applied, the output being Z_k or Z_ℓ depending on the case. More generally an OR-sum of r-factors shall mean that the input/output sequence corresponding to the first term or that corresponding to the second term, etc., has been applied.

It is easily seen that we can operate with these input/output polynomials according to a set of laws not unlike those of Boolean Algebra: There is a 0-element, namely a polynomial "0" corresponding to no further stepping forward of the clock time. There is, however, no analog of a 1-element.

Rules for Input/Output Polynomials

Let f,g and h be possible polynomials. Then

1.	$f \vee g = g \vee f$	(commutativity w.r.t. $\vee$)	(6-19)
2.	$fg \neq gf$	(non-commutativity w.r.t. $\cdot$)	(6-20)
3.	$f \vee f = f$	(idempotency w.r.t. $\vee$)	(6-21)
4.	$f \vee 0 = 0 \vee f = f$	(0 is the unit element for $\vee$)	(6-22)
5.	$f \cdot 0 = 0 \cdot f = 0$	(zero annuls a sequence)	(6-23)
6.	$(f \vee g) \vee h = f \vee (g \vee h)$	(associativity for $\vee$)	(6-24)
7.	$(fg)h = f(gh)$	(associativity for $\cdot$)	(6-25)
8.	$(f \vee g)h = fh \vee gh$	(distributivity)	(6-26)
9.	$f(g \vee h) = fg \vee fh$	(distributivity)	(6-27)

Remark 1: (6-23) may astonish at first sight. All it means is that a sequence represented by f (say of length r) cannot be made into any longer sequence when the clock is stopped.

Remark 2: The set S of all input/output polynomials is (comparing to the definitions in 5.1) a commutative idempotent semi-group (with a unit) with respect to $\vee$. It is a non-commutative, non-idempotent semi-group (with a zero) with respect to $\cdot$.

The rules given above allow us now to calculate powers of the connection matrix. We have

Theorem 4. The entry mn in C^r gives all input/output sequences of length r starting with Y_m and ending with Y_n.

Proof. Let us denote by $(C^r)mn$ the entry mn. By the rules of matrix multiplication (with + replaced by v!) we have

$$(C^r)mn = \sum_{\lambda\mu\ldots\nu=1}^{S} c_{m\lambda}c_{\lambda\mu}\ldots c_{\nu n}$$

where Σ is taken in the OR-sense. All these sequences are of length r since they correspond to polynomials of degree r. They all start with Y_m because the first term is $c_{m\lambda}$ (which may be an OR-sum of several input/output pairs) and all end with Y_n because of $c_{\nu n}$. They also go via all possible intermediate states since λ, μ, ... can take on all values between 1 and S. Of course it is entirely possible that no sequence exists, i.e., that for instance $c_{\lambda\mu} = 0$ for all lambdas which make $c_{m\lambda} \neq 0$: then $(C^r)_{mn} = 0$.

Example. Take the machine shown in Figure 6-14.

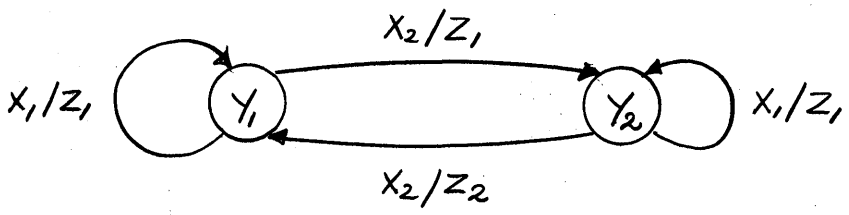


Figure 6-14 Binary Pulse Divider

As is easily seen, this is simply a binary pulse divider.

$$C = \begin{bmatrix} X_1/Z_1 & X_2/Z_1 \\ X_2/Z_2 & X_1/Z_1 \end{bmatrix}$$

$$C^2 = \begin{bmatrix} X_1/Z_1 & X_2/Z_1 \\ X_2/Z_2 & X_1/Z_1 \end{bmatrix} \begin{bmatrix} X_1/Z_1 & X_2/Z_1 \\ X_2/Z_2 & X_1/Z_1 \end{bmatrix}$$

$$= \begin{bmatrix} (X_1/Z_1)(X_1/Z_1) \vee (X_2/Z_1)(X_2/Z_2) & (X_1/Z_1)(X_2/Z_1) \vee (X_2/Z_1)(X_1/Z_1) \\ (X_2/Z_2)(X_1/Z_1) \vee (X_1/Z_1)(X_2/Z_2) & (X_2/Z_2)(X_2/Z_1) \vee (X_1/Z_1)(X_1/Z_1) \end{bmatrix}$$

meaning, for instance, that we can go from state Y_1 back to itself in two steps by the two input sequences X_1X_1 or X_2X_2, the output sequence being in the first case Z_1Z_1, and Z_1Z_2 in the second.

6.3 State Reduction in the Case of Few Input Restrictions (Hohn and Aufenkamp)

1. ALLOWABLE SEQUENCES, EQUIVALENT STATES AND MACHINES

Hohn and Aufenkamp have developed a method which allows the simplification of machines in such a way that the input/output behavior is unchanged but the number of internal states $Y_1 \ldots Y_S$ of the "black box" is reduced. This method is at its maximum efficiency in those cases where all inputs can be applied to all states (no "input restrictions") but it remains useful in cases where there are a few input restrictions. In case there are a great number of input restrictions, an extension of the method--due to Aufenkamp--can be applied: This case will be treated in Section 5.

Definition 1: Given a state Y_i, a sequence is called allowed if it corresponds to a path on the state diagram which does not violate any input restrictions.

It is to be noted that in a machine without input restrictions--also called a "completely specified machine"--all inputs are allowed in all states.

Definition 2: The state Y_i of a machine M is said to be equivalent to a state Y'_j of M' if all their allowed sequences are identical.

Definition 3: A machine M is said to be equivalent to a machine M' if for every Y_i in M there is at least one equivalent state Y'_j of M' and vice versa.

Let us now agree to write the transition of a state Y_i or a set of states $\{\ldots\ Y_i\ \ldots\}$ into a single state Y_k or a set of states $\{\ldots\ Y_k\ \ldots\}$ under the influence of X_j with an output Z_m symbolically

$$Y_i \xrightarrow{X_j/Z_m} Y_k$$

or

$$\{\ldots\ Y_i\ \ldots\} \xrightarrow{X_j/Z_m} \{\ldots\ Y_k\ \ldots\}$$

The general idea of state reduction can then be introduced by the following example.

Example of Machine Reduction

Let M be represented by the state diagram shown in Figure 6-15. Here there are no input restrictions.

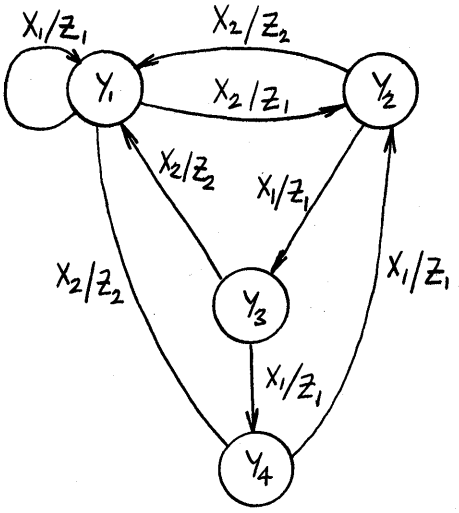


Figure 6-15
Example of a Machine
without Input Restrictions

The connection matrix of M is visibly

$$C = \left[\begin{array}{c|ccc} X_1/Z_1 & X_2/Z_1 & 0 & 0 \\ \hline X_2/Z_1 & 0 & X_1/Z_1 & 0 \\ X_2/Z_1 & 0 & 0 & X_1/Z_1 \\ X_2/Z_1 & X_1/Z_1 & 0 & 0 \end{array}\right]$$

Here we see that

$$\{Y_2, Y_3, Y_4\} \xrightarrow{X_2/Z_2} Y_1$$

$$\{Y_2, Y_3, Y_4\} \xrightarrow{X_1/Z_1} \{Y_2, Y_3, Y_4\}$$

the latter meaning that the states Y_2, Y_3 and Y_4 are permuted by input X_1.

It we "collapse" Y_2, Y_3 and Y_4 into the state Y_2' of a machine M' and call Y_1 now Y_1' of M' (for consistency), it is clear that the input/output behavior of M' (shown in Figure 6-16) cannot be distinguished from that of M: They both respond in the same way to a given input sequence and are therefore equivalent.

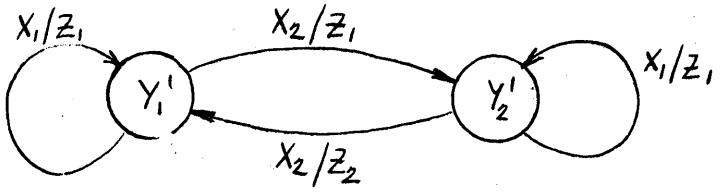


Figure 6-16 Reduced Form of the Machine in Figure 6-15

Note that this example simply shows that reduction is possible. Instead of discussing more cases and deducing rules from them, we shall treat directly the general method developed by Hohn and Aufenkamp. It will turn out that if C can be partitioned (see example) into submatrices having in each row identical entries (obviously, however, in different columns!), all states corresponding to a given partition are equivalent!

2. PERMUTABLE MATRICES

Definition 1: A matrix containing input/output polynomials as elements is called a permutable 1-matrix if:

1) The same entries appear in each row (but perhaps in different columns, perhaps all in the same column!)

2) All non-zero input sequences are different in each row.

3) All non-zero entries are OR-sums of the product of r input/output pairs.

Remark: The second condition simply means that we are talking about a deterministic machine.

Definition 2: A square matrix is symmetrically partitioned if the columns are grouped in the same way the rows are.

Example:

$$A = \left[\begin{array}{c|cc|cc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ \hline a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ \hline a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{array}\right] = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

where $A_{11} = a_{11}$; $A_{12} = [a_{12}\ a_{13}]$; $A_{13} = [a_{14}\ a_{15}]$, etc., is symmetrically partitioned because (going downwards) we have groups of 1, 2 and 2 elements and going across we also have groups of 1, 2 and 2 elements.

Theorem 1. The sum of two permutable r-matrices A and B is a permutable r-matrix provided each row in A has its set of input sequences distinct from the set of input sequences in the corresponding row in B.

Proof. We can directly apply the definition 1 above.

Remark: It is evident that permutable matrices are not necessarily square matrices.

Theorem 2. The product of a permutable r-matrix and a permutable s-matrix is always a permutable (r + s)-matrix, if the product can be formed.

Proof. Property 3 of the definition 1 is evident. Let us consider property 2: all non-zero input sequences in a row must be different. Let

$A = [a_{ij}]$, $B = [b_{ij}]$

Then element ij of the product is given by (Σ -> OR-sum)

$$(AB)_{ij} = \Sigma\, a_{i\lambda}\, b_{\lambda j}$$

(We shall omit the limits for λ in all calculations: they are 1 and S respectively.) We want to show that if $\underline{j} \neq \underline{\underline{j}}$ the input sequences in $(AB)_{i\underline{j}}$ are different from those in $(AB)_{i\underline{\underline{j}}}$ and that for a given $\underline{j}$ all sequences in $(AB)_{i\underline{j}}$ are different for the different values of λ. The latter point is evident since the sequences in $a_{i\underline{\lambda}} \neq$ the sequences in $a_{i\underline{\underline{\lambda}}}$ when $\underline{\lambda} \neq \underline{\underline{\lambda}}$, A being a permutable r-matrix and the interpretation of $a_{i\lambda}b_{\lambda j}$ is "sequences in $a_{i\lambda}$ followed by sequences in $b_{\lambda j}$": The $a_{i\lambda}$-part being different the whole sequences must be different. This argument still holds when we consider $\underline{j} \neq \underline{\underline{j}}$ and different λ's. The only new case is then an identical λ and $\underline{j} \neq \underline{\underline{j}}$: Then the sequences in $b_{\lambda\underline{j}} \neq$ from the sequences in $b_{\lambda\underline{\underline{j}}}$ because B is a permutable s-matrix and because of the above mentioned interpretation of $a_{i\lambda}b_{\lambda j}$.

Lastly we want to show that property 1 holds, i.e., that the set of all entries in a row is the same for each row. To this end consider a certain term $t = a_{\underline{i\lambda}}\, b_{\underline{\lambda j}}$ in row $\underline{i}$. Then the part $a_{\underline{i\lambda}}$ must also occur in another (arbitrarily chosen) row $\underline{\underline{i}}$ as $a_{\underline{\underline{i\lambda}}}$ since A is permutable. Similarly the part $b_{\underline{\lambda j}}$ must occur in row $\underline{\underline{\lambda}}$ of B as $b_{\underline{\underline{\lambda j}}}$ since B is permutable. Hence t appears as $a_{\underline{\underline{i\lambda}}}\, b_{\underline{\underline{\lambda j}}}$ in row $\underline{\underline{i}}$.

Remark: We shall use below an interesting property of two symmetrically partitioned square matrices $A = [a_{ij}]$ and $B = [b_{ij}]$ where the partitioning is identical: one proves easily that in forming the product it is allowed to multiply the submatrices together as if they were single elements. It is also noteworthy that all submatrices multiplied in the operation are conformable (i.e., the number of columns in the first factor equals the number of rows in the second one).

Example. Let

$$A = \left[\begin{array}{c|cc} a_{11} & a_{12} & a_{13} \\ \hline a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array}\right], \qquad B = \left[\begin{array}{c|cc} b_{11} & b_{12} & b_{13} \\ \hline b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{array}\right]$$

and set

$$A_{11} = a_{11} \qquad B_{11} = b_{11}$$

$$A_{12} = [a_{12}\ a_{13}] \qquad B_{12} = [b_{12}\ b_{13}]$$

$$A_{21} = \begin{bmatrix} a_{21} \\ a_{31} \end{bmatrix} \qquad B_{21} = \begin{bmatrix} b_{21} \\ b_{31} \end{bmatrix}$$

$$A_{22} = \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} \qquad B_{22} = \begin{bmatrix} b_{22} & b_{23} \\ b_{32} & b_{33} \end{bmatrix}$$

Then

$$AB = \begin{bmatrix} A_{11}B_{11} \vee A_{12}B_{21} & A_{11}B_{12} \vee A_{12}B_{22} \\ A_{21}B_{11} \vee A_{22}B_{21} & A_{21}B_{12} \vee A_{22}B_{22} \end{bmatrix}$$

since for instance

$$A_{11}B_{11} \vee A_{12}B_{21} = a_{11}b_{11} \vee [a_{12}\ a_{13}] \begin{bmatrix} b_{21} \\ b_{31} \end{bmatrix}$$

$$= (a_{11}b_{11} \vee a_{12}b_{21} \vee a_{13}b_{31})$$

$$A_{11}B_{12} \vee A_{12}B_{22} = a_{11}[b_{12}\ b_{13}] \vee [a_{12}\ a_{13}] \begin{bmatrix} b_{22} & b_{23} \\ b_{32} & b_{33} \end{bmatrix}$$

$$= [a_{11}b_{12} \quad a_{11}b_{13}] \vee [(a_{12}b_{22} \vee a_{13}b_{32})(a_{12}b_{23} \vee a_{13}b_{33})]$$

$$= [(a_{11}b_{12} \vee a_{12}b_{22} \vee a_{13}b_{32})(a_{11}b_{13} \vee a_{12}b_{23} \vee a_{13}b_{33})]$$

$$A_{21}B_{11} \vee A_{22}B_{21} = \begin{bmatrix} a_{21} \\ a_{31} \end{bmatrix} b_{11} \vee \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{21} \\ b_{31} \end{bmatrix}$$

$$= \begin{bmatrix} a_{21}b_{11} \\ a_{31}b_{11} \end{bmatrix} \vee \begin{bmatrix} a_{22}b_{21} \vee a_{23}b_{31} \\ a_{32}b_{21} \vee a_{33}b_{31} \end{bmatrix}$$

$$= \begin{bmatrix} a_{21}b_{11} \vee a_{22}b_{21} \vee a_{23}b_{31} \\ a_{31}b_{11} \vee a_{32}b_{21} \vee a_{33}b_{31} \end{bmatrix} \qquad \text{etc.}$$

Theorem 3. If a given symmetrical partitioning of a connection matrix C (say $C = [C_{ij}]$) results in permutable 1-matrices, then an equal partitioning of the rth power of C (say $C^r = [C^r_{ij}]$) will result in permutable r-matrices.

Proof. The theorem is clearly true for $r = 1$. Now we use induction: suppose that the theorem holds for $r = k$, i.e., that C_{ij} and C^k_{ij} are permutable matrices. Since

$$C^{k+1} = C \cdot C^k = \sum C_{i\lambda} C^k_{\lambda j}$$

by the previous remark, we conclude from Theorem 2 that each product $C_{i\lambda}C^k_{\lambda j}$ is a permutable (k+1)-matrix. Furthermore there are no common input sequences $C_{i\underline{\lambda}}C^k_{\underline{\lambda}j}$ and $C_{i\underline{\underline{\lambda}}}C^k_{\underline{\underline{\lambda}}j}$ where $\underline{\lambda} \neq \underline{\underline{\lambda}}$ because C is a connection matrix and therefore $C_{i\underline{\lambda}}$ and $C_{i\underline{\underline{\lambda}}}$ have no common input sequences. Therefore by Theorem 1 the OR-sum $\sum C_{i\lambda}C^k_{\lambda j}$ is a permutable (k+1)-matrix.

Theorem 4. If a connection matrix C can be symmetrically partitioned into permutable 1-matrices, all states within a submatrix C_{ij} are equivalent.

Proof. Consider a sequence of input/output polynomials of length r given by the terms in C^r. By definition we have partitioned C, and therefore also C^r, into submatrices C^r_{ij} which are permutable r-matrices. Hence the input/output sequences for all states inside a submatrix are identical, i.e., the states in a submatrix are equivalent.

3. THE HOHN-AUFENKAMP ALGORITHM FOR STATE REDUCTION

1) Separate the states in the connection matrix C into groups of maximum size ${}^1Y\ {}^2Y \ldots$ such that there is no overlap, and such that all states in a given group have the same input/output pairs (usually in different columns). If the partitioning is trivial (i.e., each group has one member only), the matrix cannot be reduced.

2) Reorder the connection matrix C by putting ${}^1Y$ first, then ${}^2Y$, etc., and also reorder the column in the same way, i.e., partition the result symmetrically: if all matrices are permutable we have found a reduced equivalent machine.

3) If the submatrices are not permutable, separate the states in ${}^1Y$ into ${}^{11}Y$, ${}^{21}Y$, ${}^{31}Y$, etc., such that the rows in each new subgroup in the first column have identical entries (i.e., repeat essentially step 1).

4) Reorder C according to ${}^{11}Y$, ${}^{21}Y$, ${}^{31}Y$, ..., ${}^{12}Y$, ${}^{22}Y$, ${}^{32}Y$, ... and partition the result again symmetrically (i.e., repeat essentially step 2).

5) Continue steps 3 and 4 until all matrices are permutable (meaning that we succeeded) or have only one element (meaning that M cannot be reduced).

Theorem 5. Let the final partitioning lead to groups of states ${}^1Y$, ${}^2Y$, ... ${}^QY$ ($Q \leq S$, where $Y_1 \ldots Y_S$ are the states of M) and let the submatrices corresponding to this final partitioning be C_{ij} ($i,j = 1, \ldots Q$). Now place all states in ${}^iY$ by one state Y'_i of M'. Describe M' by a matrix obtained by putting in position ij an entry d_{ij} = union of all entries in C_{ij} (see Figure 6-17). Then the reduced machine M' is equivalent to the original machine M.

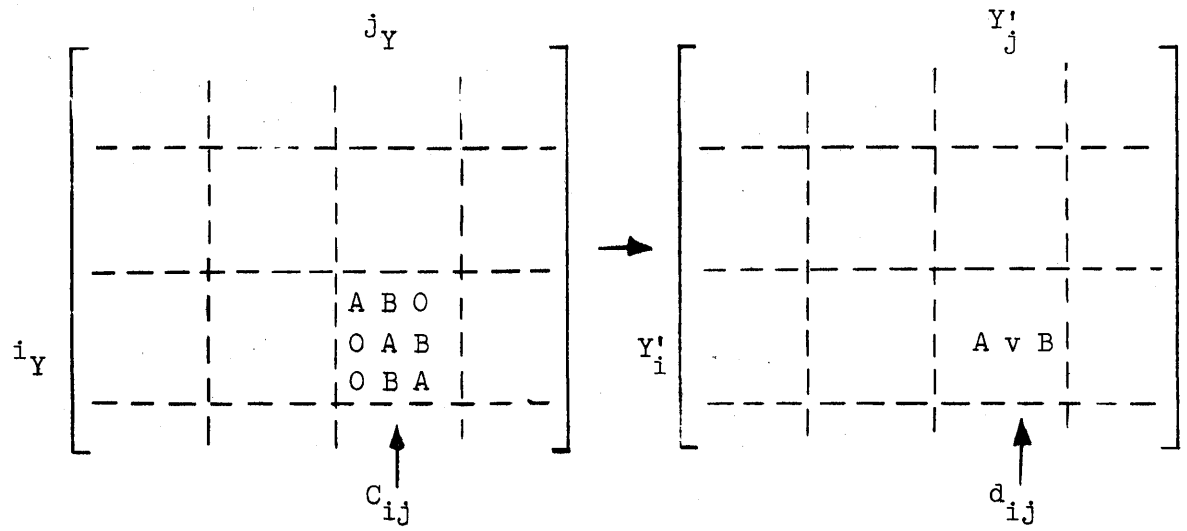


Figure 6-17 Partitioned C-matrix and Matrix of Reduced Machine

Proof. Let $D = [d_{ij}]$. The matrix D has the properties of a connection matrix since $d_{ij} \neq d_{i\underline{j}}$ (since all entries in $C_{i\underline{j}}$ are unequal to entries of the $C_{i\underline{j}}$, because the algorithm leads to groups with non-overlapping input/output sequences).

We want to show that if we have an input/output sequence of length r starting in Y'_i and leading to Y'_j there is an identical sequence starting in some state of ${}^{i}Y$ and ending in some state of ${}^{j}Y$.

All sequences of length r starting in Y'_i and ending in Y'_j of M' are given by the element ij of D^r, i.e., by

$$(D^r)_{ij} = \sum_{\lambda,\mu,\ldots\nu} \overbrace{d_{i\lambda}d_{\lambda\mu}\ldots d_{\nu j}}^{\text{r number of terms}}$$

$$= \Sigma \text{ (all sequences leading from any state in } {}^{i}Y \text{ to any state in } {}^{\lambda}Y) \text{ x (all sequences leading from any state } {}^{\lambda}Y \text{ to any state in } {}^{\mu}Y), \text{ etc.}$$

Any term in $(D^r)_{ij}$ is therefore of the form (input/output sequence from a particular state in ${}^{i}Y$ to a particular state in ${}^{\lambda}Y$) x etc., i.e., there are particular states in ${}^{i}Y$, ${}^{\lambda}Y$ ... which give a sequence of length r and identical to the term of M'.

Conversely it is easily seen that any particular sequence from a particular state in ${}^{i}Y$ to a particular state in ${}^{j}Y$ will occur in $\sum d_{i\lambda} d_{\lambda\mu} \cdots d_{\nu j}$ because d_{ij} is the union of all possible entries; therefore, M is equivalent to M'.

Example. Consider the machine described by Figure 6-18.

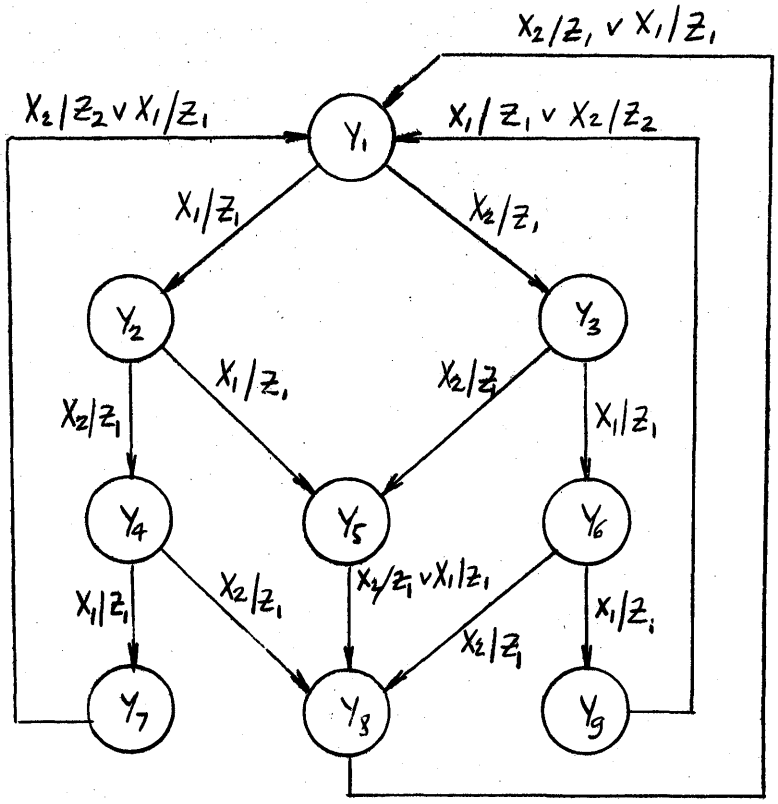


Figure 6-18 Machine to be Reduced

The corresponding connection matrix is

$$C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{bmatrix} 0 & X_1/Z_1 & X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & X_2/Z_1 & X_1/Z_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & X_2/Z_1 & X_1/Z_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & X_1/Z_1 & X_2/Z_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & X_1/Z_1 \vee X_2/Z_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & X_2/Z_1 & X_1/Z_1 \\ X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ X_1/Z_1 \vee X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

We see at once that a first partitioning leads to:

$$^{1}Y = \{Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_8\}$$

$$^{2}Y = \{Y_7, Y_9\}$$

Now C must be reordered:

$$C = \begin{array}{c|ccccccc|cc} & 1 & 2 & 3 & 4 & 5 & 6 & 8 & 7 & 9 \\ \hline 1 & 0 & X_1/Z_1 & X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & X_2/Z_1 & X_1/Z_1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & X_2/Z_1 & X_1/Z_1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & X_2/Z_1 & X_1/Z_1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & X_1/Z_1 \vee X_2/Z_1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & X_2/Z_1 & 0 & X_1/Z_1 \\ \hline 8 & X_1/Z_1 \vee X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

We note that ^{1}Y can be partitioned further:

$$^{11}Y = \{Y_1, Y_2, Y_3, Y_5, Y_8\}$$

$$^{21}Y = \{Y_4, Y_6\}$$

$$^{2}Y = \{Y_7, Y_9\}$$

Again we reorder C as shown below:

$$C = \begin{array}{c|ccccccccc} & 1 & 2 & 3 & 5 & 8 & 4 & 6 & 7 & 9 \\ \hline 1 & 0 & X_1/Z_1 & X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & X_1/Z_1 & 0 & X_2/Z_1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & X_2/Z_1 & 0 & 0 & X_1/Z_1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & X_1/Z_1 \vee X_2/Z_1 & 0 & 0 & 0 & 0 \\ 8 & X_1/Z_1 \vee X_2 Z_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 4 & 0 & 0 & 0 & 0 & X_2/Z_1 & 0 & 0 & X_1/Z_1 & 0 \\ 6 & 0 & 0 & 0 & 0 & X_2/Z_1 & 0 & 0 & 0 & X_1/Z_1 \\ \hline 7 & X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Further separation is possible: we obtain

$$^{111}Y = \{Y_1, Y_5, Y_8\} \qquad ^{21}Y = \{Y_4, Y_6\}$$

$$^{211}Y = \{Y_2\} \qquad ^{2}Y = \{Y_7, Y_9\}$$

$$^{311}Y = \{Y_3\}$$

After reordering, C will have the form:

$$C = \begin{array}{c|ccccccccc} & 1 & 5 & 8 & 2 & 3 & 4 & 6 & 7 & 9 \\ \hline 1 & 0 & 0 & 0 & X_1/Z_1 & X_2/Z_1 & 0 & 0 & 0 & 0 \\ \hline 5 & 0 & 0 & X_1/Z_2 \vee X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 8 & X_1/Z_1 \vee X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 2 & 0 & X_1/Z_1 & 0 & 0 & 0 & X_2/Z_1 & 0 & 0 & 0 \\ \hline 3 & 0 & X_2/Z_1 & 0 & 0 & 0 & 0 & X_1/Z_1 & 0 & 0 \\ \hline 4 & 0 & 0 & X_2/Z_1 & 0 & 0 & 0 & 0 & X_1/Z_1 & 0 \\ 6 & 0 & 0 & X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & X_1/Z_1 \\ \hline 7 & X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The groups are now

$$1111_Y = \{Y_1\}$$

$$2111_Y = \{Y_5, Y_8\}$$

$$211_Y = \{Y_2\}$$

$$311_Y = \{Y_3\}$$

$$21_Y = \{Y_4, Y_6\}$$

$$2_Y = \{Y_7, Y_9\}$$

No reordering is necessary, but we still do not have permutable matrices in all positions. One last partitioning attains our goal: we must split up 2111_Y into

$$12111_Y = \{Y_5\}$$

$$22111_Y = \{Y_8\}$$

Again no reordering is called for and this time all submatrices <u>are</u> permutable. The sets of states that are equivalent are thus $\{Y_4, Y_6\}$ and $\{Y_7, Y_9\}$ giving

$$Y'_1 = Y_1$$

$$Y'_2 = Y_5$$

$$Y'_3 = Y_8$$

$$Y'_4 = Y_2$$

$$Y'_5 = Y_3$$

$$Y'_6 = \{Y_4, Y_6\}$$

$$Y'_7 = \{Y_7, Y_9\}$$

Then we have a reduced connection matrix D

$$D = \begin{array}{c} \\ Y_1' \\ Y_2' \\ Y_3' \\ Y_4' \\ Y_5' \\ Y_6' \\ Y_7' \end{array} \begin{array}{c} \begin{array}{ccccccc} 1' & 2' & 3' & 4' & 5' & 6' & 7' \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & X_1/Z_1 & X_1/Z_1 & 0 & 0 \\ 0 & 0 & X_1/Z_1 \vee X_2/Z_1 & 0 & 0 & 0 & 0 \\ X_1/Z_1 \vee X_2/Z_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X_1/Z_1 & 0 & 0 & 0 & X_2/Z_1 & 0 \\ 0 & X_2/Z_1 & 0 & 0 & 0 & X_1/Z_1 & 0 \\ 0 & 0 & X_2/Z_1 & 0 & 0 & 0 & X_1/Z_1 \\ X_1/Z_1 \vee X_2/Z_2 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

and the corresponding state diagram is that of Figure 6-19.

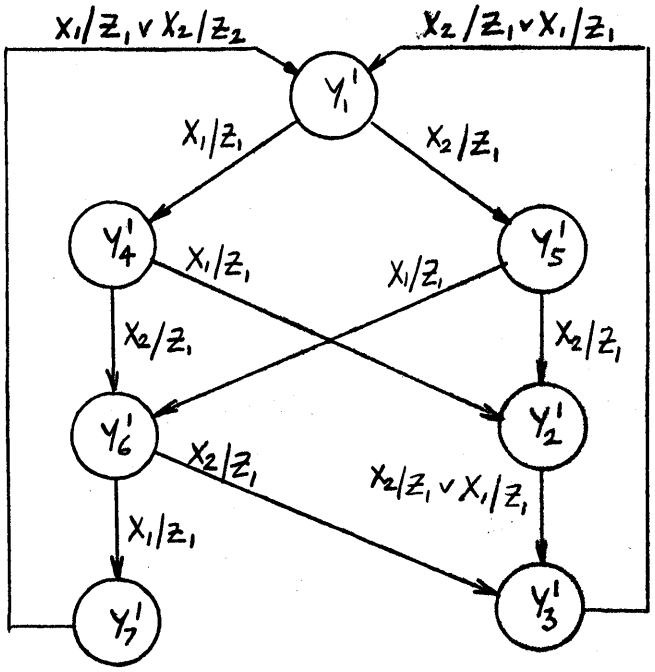


Figure 6-19 Reduced Machine Equivalent to That in Figure 6-18

Remark: It is often customary in discussing state reduction problems to call each "collapsed set" (i.e., the $Y_j' \rightarrow \{Y_a Y_b \ldots\}$) by the name of the state of lowest number in the original set: if states Y_4 and Y_6 and states Y_7 and Y_9 are respectively equivalent, the states of the reduced machines would be called

$Y_1' = Y_1$ $\qquad$ $Y_5' = Y_5$

$Y_2' = Y_2$ $\qquad$ $Y_7' = \{Y_7, Y_9\}$

$Y_3' = Y_3$ $\qquad$ $Y_8' = Y_8$

$Y_4' \rightarrow \{Y_4, Y_6\}$

Note that there is no Y_6' and Y_9'. Often it is even desired to leave off the prime symbol (to avoid confusion with "next states"). We shall follow this practice and redraw Figure 6-19 in the form 6-20. Evidently topologically equivalent state diagrams (the equivalence extending to the input/output pairs) represent the same machine.

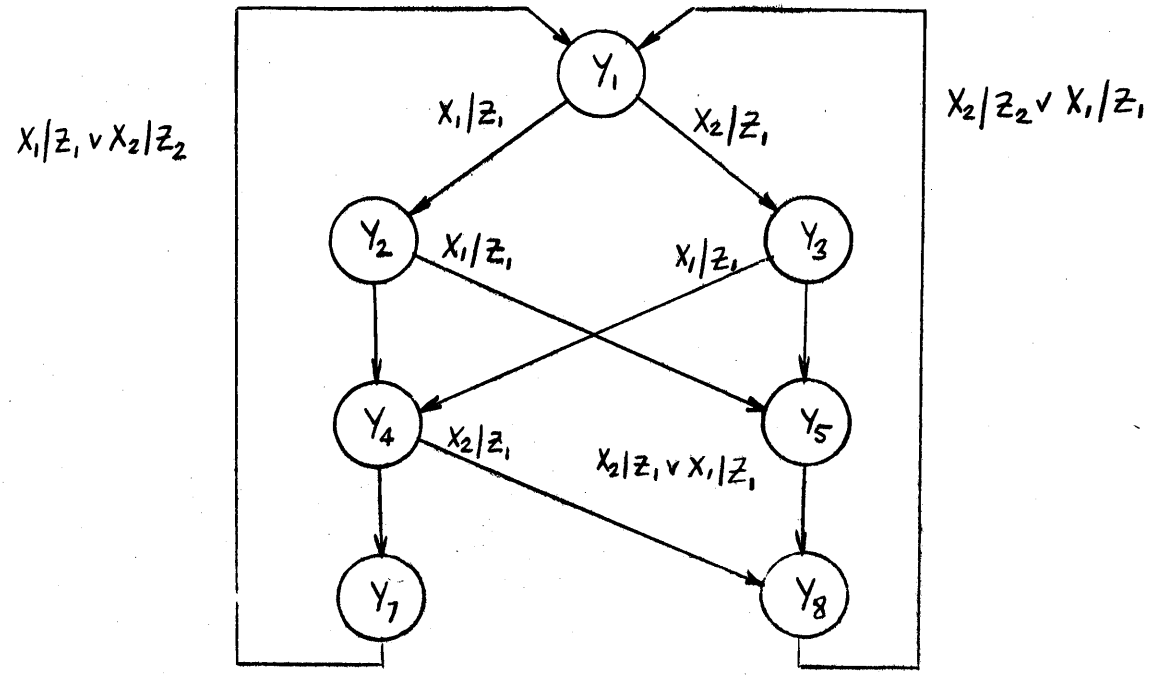


Figure 6-20 Different Notation for the Machine in Figure 6-19

6.4 State Assignment

1. SIMPLIFIED METHOD

In this simplified method we shall assume that no complicated feedback loops exist inside the machine, i.e., that the internal states $Y_1, \ldots, Y_S$ will be determined by the combination of states of an appropriately chosen

number flipflops--say t--and that we only have to design the AND and OR gates combining the inputs $X_1,\ldots,X_M$ with the outputs of these flipflops. We shall furthermore assume that each flipflop is set at each clock period, independently of whether it may actually already be in the right state or not. That such a simplification is possible will be shown by the success of the method.

The first question is: how many secondary variables $y_1,\ldots,y_t$ (representing the flipflop states, or more precisely the signal on the "1"-output side) are necessary. We shall include the answer in the State Assignment Algorithm.

State Assignment Algorithm

1. For a machine with S states take t flipflops where

$$2^t \geq S \quad \text{but} \quad 2^{t-1} < S \tag{6-28}$$

(i.e., the number of combinations of flipflops states must be at least equal to the number of internal states!)

This fixes the number of secondary variables $y_1,\ldots,y_t$. Note that we have chosen t rather than s since there is not necessarily any relationship between the number of nodes (s) and the number of flipflops (t).

2. Associate arbitrarily (we shall improve on this in the next section) the combinations of the y's with the Y's by some coding scheme:

$$Y_i \rightarrow (y_{1i}, y_{2i},\ldots,y_{ti}) \qquad i = 1 \ldots S \tag{6-29}$$

Obviously there may be many combinations of y's which are unused. When they occur in a table we may consider the corresponding Y's to be don't-cares.

3. Repeat steps 1 and 2 for $X_1,\ldots,X_m$ and $Z_1,\ldots,Z_n$, i.e., determine the appropriate number of binary input variables m and the appropriate number of binary output variables n such that we can establish an (arbitrary) code

$$X_k \rightarrow (x_{1k}, x_{2k},\ldots,x_{mk}) \qquad k = 1 \ldots M \tag{6-30}$$

$$Z_j \rightarrow (z_{1j}, z_{2j},\ldots,z_{nj}) \qquad j = 1 \ldots N \tag{6-31}$$

4. From the connection matrix C write down a modified Huffman Flow Table using secondary variables. Besides the usual "next state" columns, write down for each input $X_1,\ldots,X_M$ $y_1,\ldots y_t$ as column headings twice and mark the two parts f_0 and f_1. Mark an * under f_0 and y_i in group X_k if this input corresponds to $y_i = 0$ in the "next state" part of the table. Mark an * under f_1 and y_i if $y_i = 1$ under the influence of X_k.

5. Determine the functions $f_0(y_i)$ and $f_1(y_i)$ which are necessary to set the flipflops from the table in step 4 and the encoding of the inputs (6-30). Note that $f_0(y_i)$ means the "0"-setting function for flipflop i. This function will generally depend on $y_1,\ldots,y_t$ and $x_1,\ldots,x_m$.

6. Simplify the f_0 and f_1 functions by appropriate methods (Roth, Karnaugh). Note that under the assumptions of the present method

$$f_0(y_i) = \overline{f_1(y_i)}$$

so that we can actually just determine the f_1-functions.

7. Design a decoder such that for each combination $(x_1,\ldots,x_m)(y_1,\ldots,y_t)$ we obtain the correct $(z_1,\ldots,z_n)$ when (6-29), (6-30) and (6-31) are used. This is a (in theory simple) combinational problem.

Example. Let us reconsider the machine of the last example, drawn in the form of Figure 6-20.

Steps 1-3: We have only two inputs X_1 and X_2 and two outputs Z_1 and Z_2. Therefore we have m = 1 and n = 1, i.e., one input variable x_1 and one output variable z_1 suffice. Let us encode as follows:

$$X_1 \rightarrow x_1 = 0$$

$$X_2 \rightarrow x_1 = 1$$

$$Z_1 \rightarrow z_1 = 0$$

$$Z_2 \rightarrow z_1 = 1$$

To represent seven states we need three flipflops, i.e., three secondary variables $y_1y_2y_3$. Let us encode as follows

$Y_1 \rightarrow 0,0,0$

$Y_2 \rightarrow 0,0,1$

$Y_3 \rightarrow 0,1,0$

$Y_4 \rightarrow 0,1,1$

$Y_5 \rightarrow 1,0,0$

$Y_7 \rightarrow 1,0,1$

$Y_8 \rightarrow 1,1,0$

The combination 1,1,1 is left over.

Step 4: The table below shows the aspect of the modified Huffman Flow Table:

TABLE 6-3

	$x_1 = 0$	$x_1 = 1$	$x_1 = 0$		$x_1 = 1$	
State	State'	State'	f_1	f_0	f_1	f_0
$y_1\ y_2\ y_3$	$y_1\ y_2\ y_3$	$y_1\ y_2\ y_3$	$y_1\ y_2\ y_3$	$y_1\ y_2\ y_3$	$y_1\ y_2\ y_3$	$y_1\ y_2\ y_3$
0 0 0	0 0 1	0 1 0	*	* *	*	* *
0 0 1	1 0 0	0 1 1	*	* *	* *	*
0 1 0	0 1 1	1 0 0	* *	*	*	* *
0 1 1	1 0 1	1 1 0	* *	*	* *	*
1 0 0	1 1 0	1 1 0	* *	*	* *	*
1 0 1	0 0 0	0 0 0		* * *		* * *
1 1 0	0 0 0	0 0 0		* * *		* * *

Modified Huffman Flow Table

Steps 5 and 6: From the table we can write down the 1-setting conditions for the first flipflop; if we take the order $x_1 y_1 y_2 y_3$ we have:

$$f_1(y_1) = 0001 \vee 0011 \vee 0100 \vee 1010 \vee 1011 \vee 1100$$

$$= -100 \vee 101- \vee 00-1$$

$$= y_1 \bar{y}_2 \bar{y}_3 \vee x_1 \bar{y}_1 y_2 \vee \bar{x}_1 \bar{y}_1 y_3$$

	00	01	11	10
00	0	1	1	0
01	1	0	x	0
11	1	0	x	0
10	0	0	1	1

Figure 6-21 Karnaugh Map for $f_1(y_1)$ of the Machine in Figure 6-20

as can also be seen on the Karnaugh Map in Figure 6-21. Note the presence of x (don't care) for the two squares corresponding to the unused 111 combination of secondary variables. Here obviously we shall choose x = 0.

Similarly

$$f_1(y_2) = (x_1 \vee y_1)\bar{y}_2 y_3$$

$$f_1(y_3) = \bar{x}_1\bar{y}_1(y_2 \vee \bar{y}_3) \vee x_1\bar{y}_1\bar{y}_2 y_3$$

and, as we discussed before:

$$f_0(y_1) = \overline{f_1(y_1)}$$

$$f_0(y_2) = \overline{f_1(y_2)}$$

$$f_0(y_3) = \overline{f_1(y_3)}$$

Step 7: This is rather trivial since Z_2, i.e., $z_1 = 1$ only occurs when we are in state Y_7 ($y_1y_2y_3 \rightarrow 1,0,1$) and when we also have input X_2 ($x_1 = 1$). A simple AND circuit can decode this combination:

$$z_1 = x_1 y_1 \bar{y}_2 y_3$$

2. ELIMINATION OF PERIODIC SETTING OF FF'S

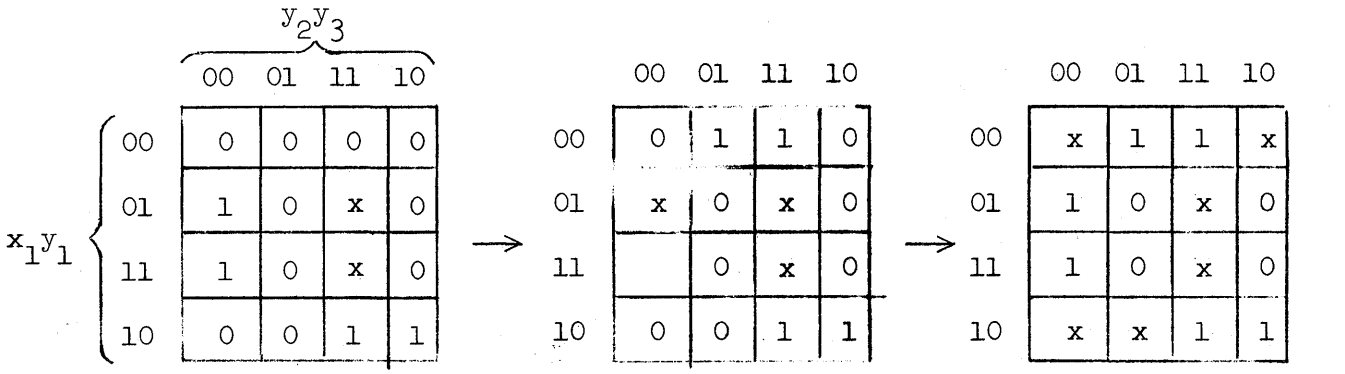
The flipflops will not have to be "adjusted" at every clock pulse if they are already in the correct state; therefore, we use the following principle:

Principle: On the $f_1(y_j)$ map replace all 1- entries in rows having $y_j = 1$ by don't cares. Similarly replace on the $f_0(y_j)$ map by don't cares 1's in rows having $y_j = 0$.

Since the original f_1 and f_0 maps (without the new don't cares introduced above) are complementary in all positions corresponding to assigned secondary variable combinations, we can factor (i.e., simplify) either 1's on f_0 or the corresponding 0's on f_1. If now the new don't cares occur, we would put an x in place of a 1 on f_0; we can therefore also put an x in place of those 0's on f_1 for which y_i is zero. We shall call f_{10} such an f_1 map on which we actually factor 0's and which has x in all positions in which $y_i = 0$ and the square is 0 on the original f_1 map.

It is clear that after choosing values for the don't cares which may differ from one map to the other, the f_0 and f_1 are no longer necessarily complementary: we pay this price to gain greater simplicity.

Example. Let us go back to the map in Figure 6-21.



Original $f_1(y_1)$ without don't cares (factor "ones")

$f_1(y_1)$ with don't cares (factor "ones")

$f_{10}(y_1)$ with don't cares (factor "zeros")

Figure 6-22 Don't Cares in the Karnaugh Map of Figure 6-21

Figure 6-22 shows how we first arrive at an $f_1(y_1)$ map with don't cares: the two leftmost "ones" in the original f_1 map become x's becuase y_1 is actually "one" in their rows. This means that we can simplify the map by setting the two next x's to 0, the old ones equal to 1 and obtain a simplified

$$f_1(y_1) = 00\text{-}1 \vee \text{--}11 \vee 101\text{-}$$

$$= \bar{x}_1\bar{y}_1y_3 \vee y_2y_3 \vee x_1\bar{y}_1y_2$$

We than factor zeros on the $f_{10}(y_1)$ map. Here it is useful to make the x's in the 00 and 10 rows equal to 1 and the x's in the two middle rows equal to 0. This leads to

$$f_0(y_1) = y_1(y_2 \vee y_3)$$

by applying the blocking technique shown in Chapter III: here we block the two leftmost "zeros" by $(y_2 \vee y_3)$.

3. OPTIMIZATION OF STATE ASSIGNMENT

At the beginning of this section we agreed to choose the "code" for the correspondence between $Y_1,\ldots,Y_S$ and the combinations of $y_1,\ldots,y_t$ arbitrarily. We shall now improve our method, i.e., choose the code in such a fashion that the gating circuitry is simplified. We shall judge our success by the ease of factoring of the f_0 and f_1 maps, in particular we would like to make the largest number of maps as simple as possible. We shall, however, completely neglect the output "code": This calls for a separate treatment.

Definition: The assignments

$$Y_i \rightarrow (y_{1i}, y_{2i},\ldots,y_{ti})$$

$$Y_j \rightarrow (y_{ij}, y_{2j},\ldots,y_{tj})$$

are called neighboring if they differ in as few digits as possible.

Since Y_i and Y_j are different states, the two combinations must at least differ in one digit: The optimum for a neighboring assignment is therefore one adjacent assignment in the sense of a Karnaugh map.

The idea is now to consider the $f_0(y_m)$ and $f_1(y_n)$ maps for a given assignment of the form (6-29). We shall also go back to our assumption that the flipflops are set at each cycle,

independently of whether this is actually necessary or not. In order to simplify our reasoning we shall assume that there is only one input variable x_1 (i.e., $x_1 = 0$ or $x_1 = 1$) and that there are only three secondary variables $y_1y_2y_3$. Let us now draw

1. A Present State Map

   This is a Karnaugh map containing in each square one of the state symbols $Y_1,\ldots,Y_S$. This map will be symmetric with respect to a horizontal line through the center, since we do not take account of the input ($x_1 = 0$ or $x_1 = 1$). Let X_k (see Figure 6-23) be a state corresponding to two symmetrically placed sequences.

2. A Next State Map

   This is a map in which a given square contains the state following the state in a similar location on the present state map. Since this new state depends on whether $x_1 = 0$ or $x_1 = 1$, we can no longer expect symmetry. In Figure 6-24 we have Y_i and Y_j respectively, where $Y_k \xrightarrow{x_1 = 0} Y_i$ and $Y_k \xrightarrow{x_1 = 1} Y_j$.

3. An f_0 or f_1 Map for y_n

   Whether we choose f_0 or f_1 depends on whether in the $Y_k \to Y_i$ transition (i.e., the upper half of the map) the variable y_n has to be set to 0 or to 1. (By virtue of our hypothesis that flipflops are set at each cycle one of the maps will contain a 1.)

   The idea is now that if Y_j has an assignment neighboring to that of Y_i the y_i's of Y_j change when those of Y_i do: a 1 or a 0 in the square $\to Y_i$ will give the same symbol in the square $\to Y_j$, i.e., the f_1 (or f_0) maps for most of the y_i's will have symmetrically placed 0's or 1's and will be easy to factorize. This amounts to saying that two next states Y_i and Y_j (symmetrically placed) should have neighboring assignments.

Remark: Since we have only treated the $x_1y_1y_2y_3$ case it is difficult to generalize the geometrical rules. It is, however, not too difficult to find the modified rules in more general cases. In the case of $x_1x_2y_1y_2y_3$ one sees, for instance, that all next states in a vertical column of the five-variable Karnaugh map should be neighboring in their assignments.

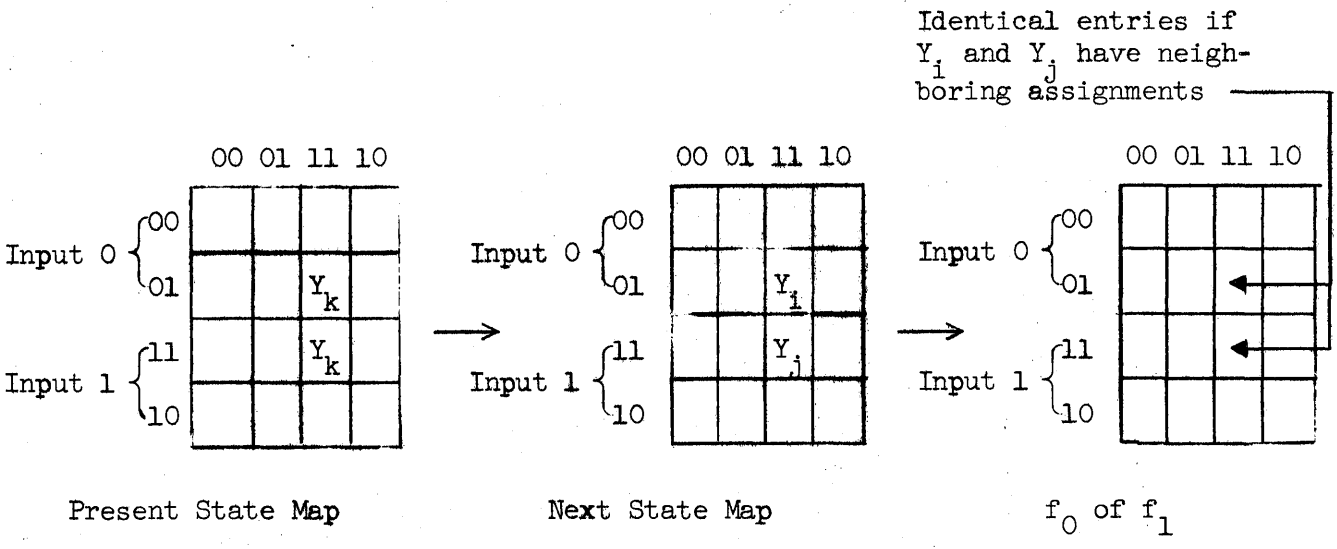


Figure 6-23 Reason for Neighboring State Assignments

It is to be noted that in order to draw Figure 6-23 it is necessary to make a tentative assignment. But it is also clear that the result will not depend on this tentative assignment, since the symmetry properties do not depend on it.

Example. Let Figure 6-24 give the state diagram of a certain machine.

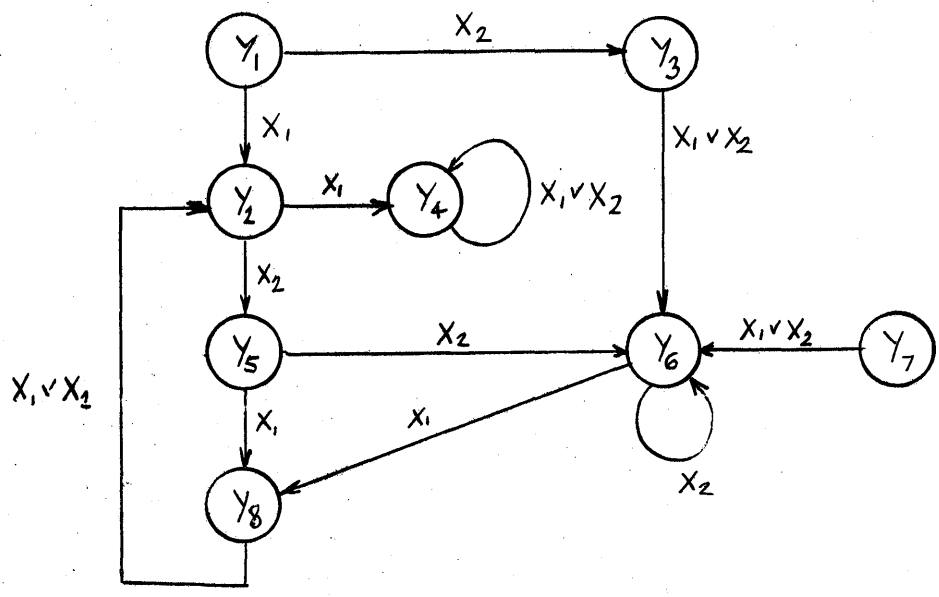


Figure 6-24 Assignment Example

Let us make the following tentative assignment

$Y_1 \rightarrow 000$ $\qquad$ $X_1 \rightarrow x_1 = 0$

$Y_2 \rightarrow 001$ $\qquad$ $X_2 \rightarrow x_1 = 1$

$Y_3 \rightarrow 010$

$Y_4 \rightarrow 011$

$Y_5 \rightarrow 100$

$Y_6 \rightarrow 101$

$Y_7 \rightarrow 110$

$Y_8 \rightarrow 111$

Then the state diagram gives us the Present State and Next State maps shown in Figure 6-25.

	00	01	11	10
00	Y_1	Y_2	Y_4	Y_3
01	Y_5	Y_6	Y_8	Y_7
11	Y_5	Y_6	Y_8	Y_7
10	Y_1	Y_2	Y_4	Y_3

$\longrightarrow$

	00	01	11	10
00	Y_2	Y_4	Y_4	Y_6
01	Y_8	Y_8	Y_2	Y_6
11	Y_6	Y_6	Y_2	Y_6
10	Y_3	Y_5	Y_4	Y_6

Figure 6-25 Maps for the Assignment Example

We now deduce directly from the right-hand map that the following pairs should have neighboring assignments: $\{Y_2, Y_3\}$, $\{Y_4, Y_5\}$ and $\{Y_6, Y_8\}$. Comparing Figure 6-10a in Section 2 to Figure 6-24, we see that the two state diagrams are really the same. In Section 2 the state assignment (dictated by the actual layout of the flipflops and their gates) was

$Y_1 \rightarrow 000$

$Y_2 \rightarrow 010$ ⟵ neighboring

$Y_3 \rightarrow 110$ ⟵ neighboring

$Y_4 \rightarrow 011$ ⟵ neighboring

$Y_5 \rightarrow 111$ ⟵ neighboring

$Y_6 \rightarrow 101$ ⟵ neighboring

$Y_7 \rightarrow 100$ neighboring

$Y_8 \rightarrow 001$ ⟵ neighboring

and actually does satisfy our criterion for optimization.

6.5 Machines with Prescribed Input/Output Behavior. State Reduction in the Case of Many Input Restrictions

1. DEFINITION OF A MACHINE BY SEQUENCES

It is possible to design a machine by specifying its output sequences when given input sequences are applied. Such a specification takes the following form:

$$\left.\begin{aligned} X_{i_1} X_{j_1} \dots X_{k_1} &\rightarrow \underbrace{Z_{i_1} Z_{j_1} \dots Z_{k_1}}_{\text{length } L_1} \\ X_{i_2} X_{j_2} \dots X_{k_2} &\rightarrow \underbrace{Z_{i_2} Z_{j_2} \dots Z_{k_2}}_{\text{length } L_2} \\ X_{i_g} X_{j_g} \dots X_{k_g} &\rightarrow \underbrace{Z_{i_g} Z_{j_g} \dots Z_{k_g}}_{\text{length } L_g} \end{aligned}\right\} \qquad (6\text{-}32)$$

In case the length of any of these sequences is infinite (i.e., infinitely many terms) we shall assume that it is periodic after a finite number of terms. In such a case we shall write down one complete period (on both sides), underlining it and marking it "cycle."

Remark: We are not saying that we can always start the desired machine in the same state to obtain the above correspondence. All we are trying to obtain is a machine M which started in some appropriate state will show the desired input/output behavior. By virtue of Theorem 3 in Section 1 and its proof we can always attain the appropriate starting state by applying a fixed input for a sufficiently long time.

The design procedure is quite elementary: we design separate machines M_1, M_2 ..., etc., for each one of the sequences, i.e., we draw up an appropriate state diagram. We than merge all state diagrams into a single one by re-numbering all states. Although this "merged" diagram is formed of isolated pieces, it is a perfectly acceptable diagram of a machine M.

The next step would be to simplify M and to obtain a reduced machine M' by the Hohn-Aufenkamp method. Unluckily it turns out that the very fact that we have disconnected sub-diagrams means that only very few inputs may be applied to a given state (i.e., we have severe input restrictions). The ordinary partitioning of C leads usually to nothing. Happily there is an extension of the method (due to Aufenkamp) which gives useful results. It will be treated after an example.

Example. Suppose that we are given the following sequence requirements:

$$\underset{\text{cycle}}{X_1, X_1, X_1} \rightarrow \underset{\text{cycle}}{Z_1, Z_2, Z_1}$$

$$\underset{\text{cycle}}{X_1, X_1, X_2} \rightarrow \underset{\text{cycle}}{Z_1, Z_2, Z_2}$$

$$X_2, X_2, X_2 \rightarrow Z_2, Z_1, Z_2$$

and

$$\underset{\text{cycle}}{X_1, X_2, X_1} \rightarrow \underset{\text{cycle}}{Z_2, Z_1, Z_1}$$

where "cycle" means that an infinite succession of the underlined terms on the left gives an infinite succession of the underlined terms on the right.

By the procedure outlined above we find the following partial state diagrams for the four "partial machines":

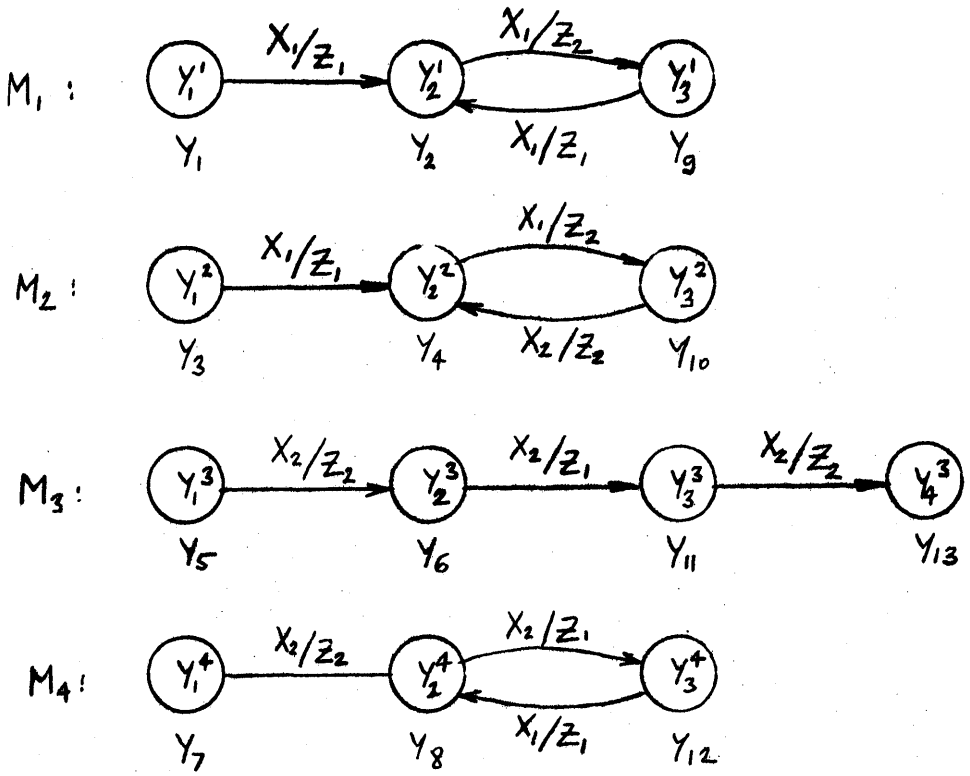


Figure 6-26 Partial State Diagrams

The union of M_1, M_2, M_3, M_4 forms a machine M which is obtained by renumbering the states as shown in Figure 6-26, i.e., Y_1' becomes Y_1, Y_2' becomes Y_2, etc. We can write down the connection matrix for M: due to the disjoint structure of its state diagram and the fact that all states only allow an X_1 or an X_2, input partitioning according to Hohn-Aufenkamp does not lead to any reduction. However it is clear that the machine can be reduced: Figure 6-27 shows a machine M' having exactly the prescribed input/output behavior and only two internal states!

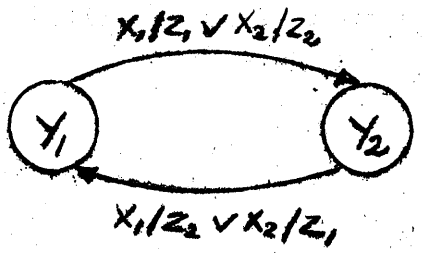


Figure 6-27 Reduced Machine Corresponding to That of Figure 6-26

2. NON-PERMUTABLE MATRICES

The state reduction method discussed in Section 3 can be generalized. Aufenkamp found that if the terms "equivalent" and "permutable" are replaced by "compatible" and "non-permutable" respectively in the statements of that section, most results can be interpreted to have a more general meaning.

The general idea is that if two states do not have to react to the same input, they may be contracted into one, although they are certainly not equivalent. Since the Theorems (as well as their proofs) are very similar to those in Section 3, we shall not give any proofs; they may be left as an exercise for the reader.

Definition 1: A state Y_i of a machine M is compatible with a state Y'_j of a machine M' if for those input sequences they might have in common the output sequences of M and M' are identical.

Remark: Note that this means that two states may be compatible simply because they do not have any inputs in common. In case all allowed sequences are common to both states, the notion of compatibility reverts to that of equivalence.

Definition 2: Two machines M and M' are compatible if and only if for every state Y_i of M there is at least one compatible state Y'_j of M' and vice versa.

Definition 3: A set of states of M is called pseudo-equivalent if they are all compatible.

Definition 4: A matrix containing input/output polynomials as elements is called a non-permutable r-matrix if

1. whenever two rows happen to have the same input sequence, they are associated with the same output sequence. (In the permutable case all input/output sequences would occur in each row.)
2. in a row all non-zero input sequences must be different.
3. all non-zero entries are OR-sums of the product of r input/output pairs.

Using these definitions, the following theorems can be stated (The proofs are analogous to those given in Section 3; hence, will not be given here.):

Theorem 1. The sum of two non-permutable r-matrices A and B is another non-permutable r-matrix if the entries in each row in A are different from those of the corresponding row in B and if furthermore whenever an input appears in different rows in both matrices, it is associated with the same output.

Theorem 2. The product of a non-permutable r-matrix and a non-permutable s matrix is a non-permutable r + s matrix if it can be formed.

Theorem 3. If a given symmetrical partitioning of a connection matrix $C = [C_{ij}]$ gives non-permutable 1-matrices and furthermore all submatrices in a row have different sets of entries, the rth power of C, partitioned in the same way (i.e., C^r_{ij}), has as its submatrices non-permutable r-matrices and the submatrices in a row again have different sets of entries.

Theorem 4. If C can be symmetrically partitioned such that all submatrices are non-permutable 1-matrices and such that all submatrices in a row hav disjoint input sequences, then all states in a submatrix are pseudo-equivalent.

3. THE AUFENKAMP ALGORITHM

1. Parition the states $Y_1 \ldots Y_S$ in the connection matrix C into groups of maximum size ^1Y, ^2Y ... such that there is no overlap and such that the

rows in each set form non-permutable 1-matrices, and such that if two groups are united, the result is no longer a non-permutable 1-matrix. (There usually is more than one solution.) If the partitioning is tri ial, the matrix cannot be reduced.

2. Reorder the connection matrix by putting ^1Y first, then ^2Y, etc., and partition symmetrically: if all submatrices are non-permutable 1-matrices and all submatrices in a row have disjoint input sequences we terminate: all states in a partition are pseudo-equivalent.

3. If the submatrices after Step 2 are not non-permutable 1-matrices, repartition inside of ^1Y, ^2Y, ... If the result is trivial, there are no pseudo-equivalent states.

4. If the partitioning is Step 3 is successful, reorder and partition the matrix symmetrically.

5. Continue Steps 3 and 4 until all matrices are non-permutable 1-matrices (meaning that we succeeded) or have only one element (meaning that M cannot be reduced).

Theorem 5. If the pseudo-equivalent states of M obtained by the Aufenkamp algorithm are replaced by a single state of a machine M' and the connection matrix C' of M' is obtained by forming the union (OR-sum) of the entries in the submatrices of C after the final partitioning, then M' is compatible with M. (This means, of course, that for those input sequences they may have in common, the output sequences will be identical!)

Theorem 6. The reduced machine M' can accept all input sequences of M but not vice versa.

Proof. This rather important fact (the reduction would be without sense otherwise) simply follows from the reduction method: no inputs are lost in the partitioning and the formation of the final OR-sum.

Example 1. Let M be given by the state diagram of Figure 6-28.

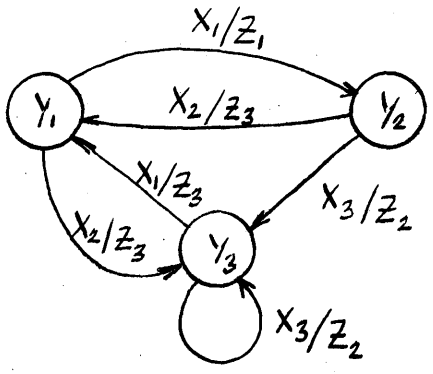


Figure 6-28 Machine to be Reduced

Consequently C is given by

$$C = \left[\begin{array}{c|cc} 0 & X_1/Z_1 & X_2/Z_3 \\ \hline X_2/Z_3 & 0 & X_3/Z_2 \\ X_1/Z_3 & 0 & X_3/Z_2 \end{array}\right] \begin{array}{l} \} \; Y_1' \\ \\ \} \; Y_2' \end{array}$$

Note that by the Hohn-Aufenkamp method, C is irreducible. The Aufenkamp method, however, gives the indicated partitioning: there are two pseudo-equivalent states Y_1' (corresponding to Y_1) and Y_2' (corresponding to Y_2 and Y_3). The reduced state diagram is shown in Figure 6-29.

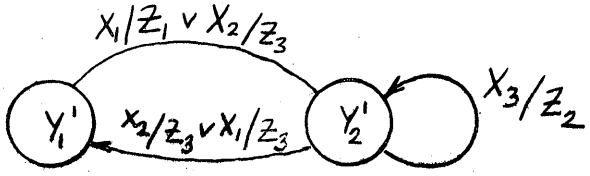


Figure 6-29 Reduction of the Machine in Figure 6-28

Example 2. Let us apply the Aufenkamp algorithm to the machine ($M = \sum_{i=1}^{4} M_i$) discussed at the beginning of this section. Its connection matrix can be symmetrically partitioned and reordered as follows:

	1	3	5	7	9	10	11	12	2	4	6	8	13
1	0	0	0	0	0	0	0	0	X_1/Z_1	0	0	0	0
3	0	0	0	0	0	0	0	0	0	X_1/Z_1	0	0	0
5	0	0	0	0	0	0	0	0	0	0	X_2/Z_2	0	0
7	0	0	0	0	0	0	0	0	0	0	0	X_2/Z_2	0
9	0	0	0	0	0	0	0	0	X_1/Z_1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	X_2/Z_2	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	X_2/Z_2
12	0	0	0	0	0	0	0	0	0	0	0	X_1/Z_1	0
2	0	0	0	0	X_1/Z_2	0	0	0	0	0	0	0	0
4	0	0	0	0	0	X_1/Z_2	0	0	0	0	0	0	0
6	0	0	0	0	0	0	X_2/Z_1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	X_2/Z_1	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0

Hence M has effectively two pseudo-equivalent sets of states:

$$\{Y_1, Y_3, Y_5, Y_7, Y_9, Y_{10}, Y_{11}, Y_{12}\} \rightarrow Y_1'$$

$$\{Y_2, Y_4, Y_6, Y_8, Y_{13}\} \rightarrow Y_2'$$

This agrees with our previous findings.

6.6 Asynchronous Circuit Theory (Muller-Bartky)

1. TARGET STATES, SURROUNDING STATES, → RELATIONSHIP

It will turn out that in the discussion below we will need not only what corresponds to a "next state" for each state Y_1 ... Y_S but also "surrounding states." Furthermore we must often distinguish sequences of states starting with a given initial state and even discriminate between the signals at the circuit nodes for each one of these: if Y_i were a given state,

surrounding states would have to have two indices Y_{ik}, the signals $y_1 \ldots y_s$ three indices, i.e., $Y_{ik} = (y_{1ik}, y_{2ik} \ldots y_{sik})$ and a node signal for a sequence a fourth index. In order to simplify matters we shall often call the states A, B, ... Y ... Z (we will not be concerned with outputs and can use X and Z for states), i.e., $A = Y_1$, $B = Y_2$ ... etc. The "surrounding" relationship will be indicated without using a subscript, a sequence of states starting with A will be written as A(0) A(1) ... A(n) ... and the internal signals of A(n) by $(a_1(n)\ a_2(n) \ldots a_s(n))$.

In synchronous circuit theory we had the equation

$$y_i' = f_i(x_1,\ldots,x_m,\ y_1,\ldots,y_s)$$

In asynchronous circuit theory we assume that the inputs are held constant while we examine the transitions of the machine, i.e., that we actually have

$$y_i' = f_i(y_1,\ldots,y_s) \tag{6-33}$$

The inputs may be thought of as parameters that can only be changed after the machine has settled down. We suppose, as usual, that it is always possible to choose appropriate internal nodes or "cardinal points" such that the state of the (input-independent) machine is completely specified by their signals.

Definition 1: The state $Y' = (y_1',\ldots,y_s')$ defined by (6-33) will be called the target state of $Y = (y_1,\ldots,y_s)$.

Remark: The target state Y' of an asynchronous machine is defined in the same way as the next state Y' of a synchronous machine. In the present case, however, there is no guarantee that the machine will ever attain Y' because of internal races.

Definition 2: A state $W = (w_1,\ldots,w_s)$ "surrounds" state $Y = (y_1,\ldots,y_s)$ with target state $Y' = (y_1',\ldots,y_s')$ if its signals agree with those of Y and Y' whenever the latter agree:

$$w_i = y_i = y_i' \quad \text{if} \quad y_i = y_i'$$

Otherwise we shall allow w_i to have either value, i.e.,

$$w_i = \begin{cases} y_i \\ \text{or } y_i' \end{cases} \quad \text{if } y_i \neq y_i' \tag{6-34}$$

(Note that (6-34) contains the case $y_i = y_i'$.) For such a state W surrounding Y we shall write $Y \to W$ (W surrounds Y or Y is surrounded by W), with the explicit understanding that W may come after Y but must not and that there was no intervening state.

Remark 1: The $\to$ relationship is denoted by $\aleph$ in Muller's original papers. Also his definitions include the case of more than two signal values.

Remark 2: It is clear from the definition that $Y \to Y'$ and $Y \to Y$. However it is usually not true that $Y \to W$ implies $W \to Y$.

Remark 3: If Y' differs from Y in k digit positions (i.e., signals), Y is surrounded by 2^k states (including Y and Y' themselves).

Theorem 1. Any state following Y directly must surround Y.

Proof. It is clear that the next state after Y will correspond to a change in none, some or all the signals, excepting those which remain constant in passing from Y to Y'. A following state is therefore a surrounding state.

Example. Let Y = (0,0,0,0) and Y' = (0,1,1,1). Then the states surrounding Y (and $\neq$ from Y and Y') are (0,0,0,1), (0,0,1,0), (0,1,0,0), (0,0,1,1), (0,1,1,0) and (0,1,0,1), i.e., they are obtained by changing the digits one at the time, two at the time, etc. Figure 6-30 shows this relationship on a tesseract. One can say that all surrounding states lie on a cube passing through the initial state and the target state.

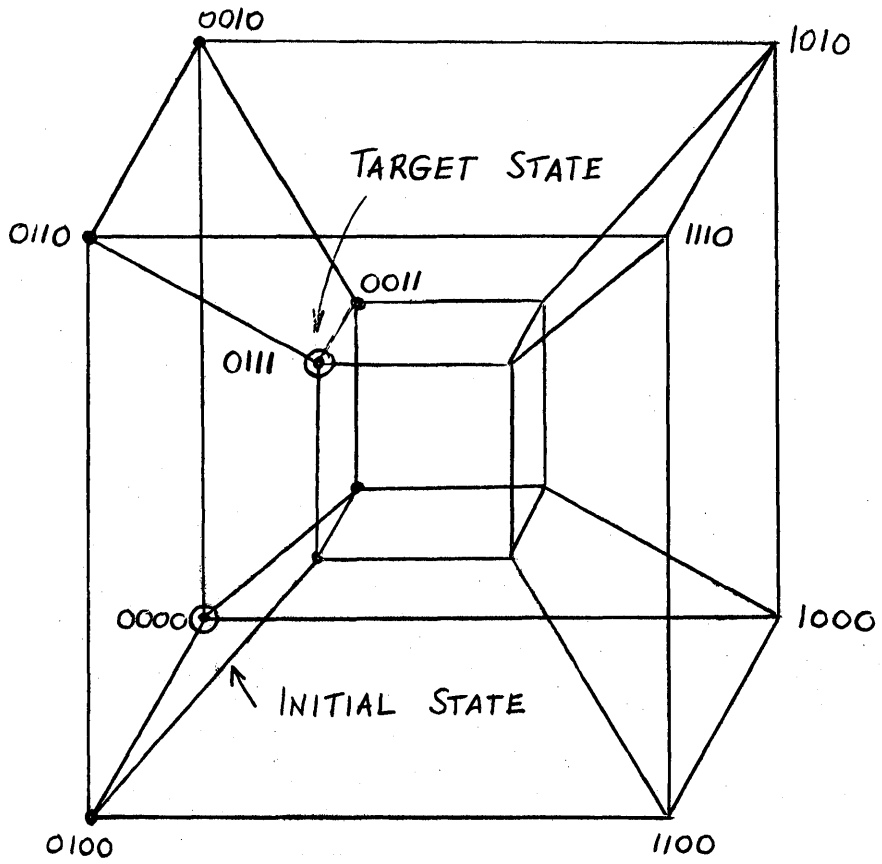


Figure 6-30 States Surrounding State (0,0,0,0) with Target State (0,1,1,1)

Theorem 2. A machine is in equilibrium if and only if $Y = Y'$.

Proof. If $Y = Y'$ there are no surrounding states $\neq Y$: the following state can therefore only be Y and this means equilibrium. If the machine is in equilibrium all surrounding states must be the same; now Y' always surrounds Y, therefore $Y = Y'$.

Definition 3: A sequence of states $Y(0), Y(2), \ldots Y(j), Y(j+1) \ldots$ is an allowed sequence if and only if it satisfied the following conditions:

1. $Y(j) \to Y(j+1)$ (6-35)

   ($Y(j+1)$ surrounds $Y(j)$)

2. $Y(j+1) \neq Y(j)$ (6-36)

   ($Y(j+1)$ differs from $Y(j)$)

3. For no internal node i can we have for all $j > 0$

$$\left.\begin{array}{l} Y_i(j) \text{ constantly} < Y'_i(j) \\ \text{or } Y_i(j) \text{ constantly} > Y'_i(j) \end{array}\right\} \quad (6\text{-}37)$$

(target condition)

(Here > and < are taken in the Boolean sense, which simply reduce to the ordinary numerical $0 < 1$ and $1 > 0$).

Remark 1: The second condition eliminates the trivial case when the machine hangs up in one state.

Remark 2: The third condition simply means this: when for a given node the signal in the target state is different from the signal in the present state and "pulls" constantly in the same direction, the node will finally "give in" and change in the direction of the "force." This excludes by no means the possibility of the target state pulling sometimes in one direction and sometimes in the other. In such a case we shall say that node i is variably forced. A sequence of different states following each other and variably forced for all nodes (or simply "variably forced") is always an allowed sequence.

Remark 3: A cyclic sequence is perfectly allowed if its states are variably forced.

Remark 4: A subsequence of finite length in which (6-37) is not necessarily verified, is called a partial allowed sequence or simply a sequence.

Definition 4: We shall say that a state K "follows" a state A if there is a sequence A = A(0), A(1), ... containing K. We shall then write $A \nrightarrow K$. (There are, usually, many intermediate states.) This sequence does not necessarily satisfy the target condition.

Theorem 3. For any state A there is at least one allowed sequence starting with it except if A = A' (equilibrium!).

Proof: A' surrounds A and we can form A(0), A(1), ... by making A(0) = A, A(1) = A' = A'(0), A(2) = A'(1), etc. It must come to an equilibrium state or go into a cycle: in both cases the target condition is satisfied ($a'_i(j) = a_i(j+1)$!)

Theorem 4. An allowed finite sequence ends with an equilibrium state.

Proof: There would be a continuing allowed sequence from the last state K (say) if $K \neq K'$.

2. EQUIVALENT, TERMINAL, FINAL AND PSEUDO-FINAL SETS

Definition 5: If two states A and B are "reversibly joined", i.e., if $A \nrightarrow B$ and $B \nrightarrow A$ we shall say that they are in the same equivalent set and write $A \mathcal{E} B$.

From the definition it follows that the $\mathcal{E}$-relationship satisfies the following rules:

$$A \mathcal{E} A \tag{6-38}$$

$$A \mathcal{E} B \rightarrow B \mathcal{E} A \tag{6-39}$$

$$A \mathcal{E} B \text{ and } B \mathcal{E} C \rightarrow A \mathcal{E} C \tag{6-40}$$

Let us denote the equivalence sets by Greek letters α, β.... Note that their number is finite since the number of states S is finite.

Definition 6: We shall write $\alpha \mathfrak{I} \beta$ if there is a state A* in α and a state B* in β such that $A^* \mathfrak{I} B^*$.

Theorem 5. If A is any state in α and B any state in β and $\alpha \mathfrak{I} \beta$, then $A \mathfrak{I} B$.

Proof: There is an A* in α and a B* in β with $A^* \mathfrak{I} B^*$. Also by definition $A \varepsilon A^*$ and $B \varepsilon B^*$, There are, therefore, sequences from A to A*, from A* to B* and from B* to B.

Remark: Clearly $\alpha \mathfrak{I} \beta$ does not imply $\beta \mathfrak{I} \alpha$, for then all states in α and β would be reversibly joined and the sets α and β should have been collapsed into a single set.

Theorem 6. The equivalence sets α, β... form a poset.

Proof: $\mathfrak{I}$ in the ordering of α, β... can be replaced by $\leq$ in the rules for a poset in 5.5:

$A \mathfrak{I} A$ (reflexivity)

$A \mathfrak{I} \beta$ and $\beta \mathfrak{I} \alpha \rightarrow \alpha = \beta$ (anti-symmetry, see remark above)

$\alpha \mathfrak{I} \beta$ and $\beta \mathfrak{I} \gamma \rightarrow \alpha \mathfrak{I} \gamma$ (transitivity)

Definition 7: A final set μ is a set such that there is no set μ^* with $\mu \mathfrak{I} \mu^*$.

Theorem 7. For any equivalence set α there is at least one final set μ such that $\alpha \mathfrak{I} \mu$.

Proof: This follows from the partial ordering: any poset has at least one maximum and one minimum element.

Definition 8: A pseudo-final set is an equivalence set of states containing more than one state--which is not final and variably forced. (The last condition means, as usual, that no node i may have p_{in} constantly $< p'_{in}$ or p_{in} constantly $> p'_{in}$ for all n, where $P_n = (p_{1n}, \ldots, p_{sn})!$)

Definition 9: If an allowed sequence Y(0), Y(j), Y(j+1)... has the property that for $j \geq m$ all states are in the same equivalence set τ, this equivalence set is called the terminal set of the sequence.

Theorem 8. Any allowed sequence attains a terminal set.

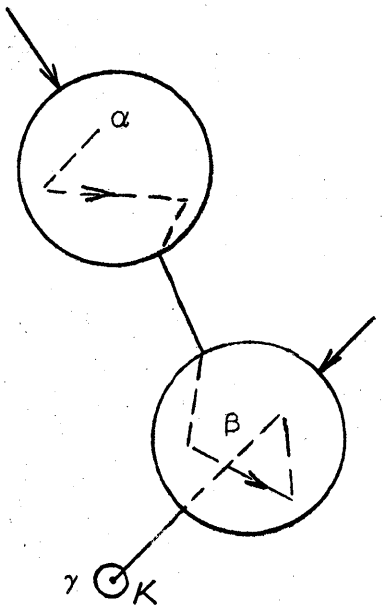
Proof: The number of equivalence sets is finite and they are partially ordered: after having left a certain set as we go along in one sequence we are never allowed to go back to it. So the sequence slowly exhausts all equivalence sets and must, after some time, be trapped in a last one.

Theorem 9. The terminal set of an allowed sequence is either pseudo-final (in this case we have a cycle in it) or final (in this case we have several states and a cycle or just one state and equilibrium). Figure 6-31 shows all these possibilities.

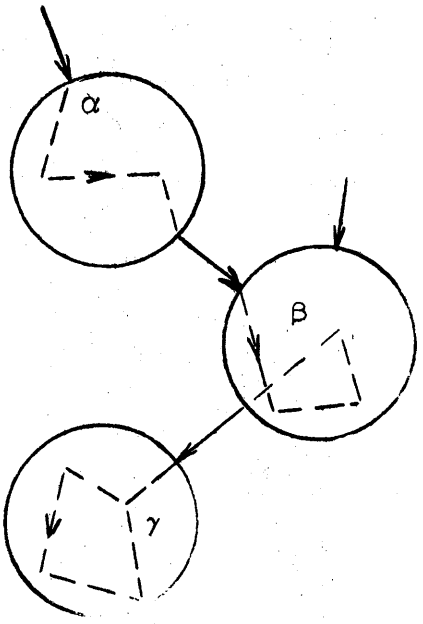
Proof: Suppose that τ is final. If it has one state K, this means that we cannot go anywhere from K. But we always have $K \to K'$: we must have $K = K'$, i.e., equilibrium. Conversely if we go to equilibrium in a state K of τ, K must be the only state in τ: any other state M preceding K (and in τ) as we go towards K must be reversibly joined to K (since both are in τ). K being an equilibrium state, we cannot go anywhere from K, in particular not to M. Therefore M does not exist. The target condition is satisfied since we have equilibrium.

If we still suppose that τ is final but contains more than one state, we can evidently have a cycle. Then the variable forcing condition must be satisfied for the states of the cycle since we have an allowed sequence.

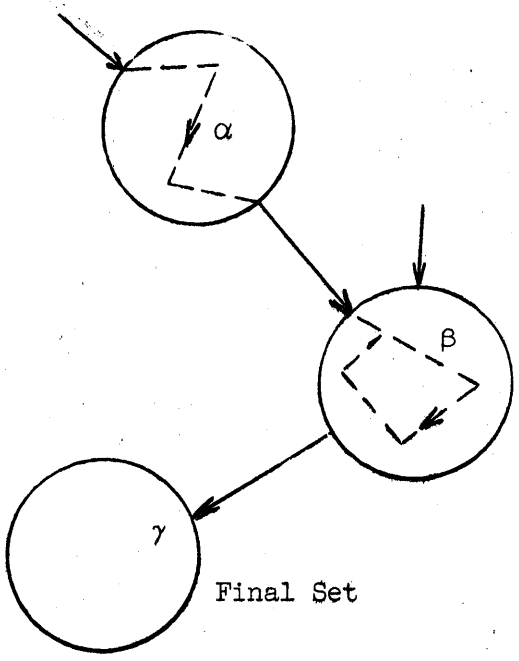
Now we shall suppose that τ is not final: then it is an intermediate set with more than one state. (One state K would mean equilibrium--since τ is terminal--and then $K = K'$ means that there are no states surrounding K: we cannot leave K and τ would be final.) Since it contains an allowed sequence the target condition becomes the variable forcing condition and the set must be pseudo-final.



Final Set with
One State (Equilibrium)



Final Set with
Several States



Pseudo-Final Set (Several States)

Figure 6-31 Ultimate Behavior of an Asynchronous Circuit

Theorem 10. If A is a state in α and φ is any final or pseudo-final set following α (i.e., $\alpha \mathcal{F} \varphi$), there is an allowed sequence A = A(0), A(1),... whose terminal set τ is φ.

Proof: If φ is final, this is evident, for we can go from A to a certain F* in φ. From F* onwards we can take the target state sequence: this is trapped in φ since it is final and it is allowed (as are all target state sequences!). If φ is pseudo-final we can still go from A to F* in φ. Let F* = P_n of φ. From P_n we can go to P_{n+1} (since they are in the same equivalence set), from P_{n+1} to P_{n+2}, etc., ... up to P_r. From P_r we go back to P_0. This sequence P(0) = P_n, P(1) = next state on path from P_n to P_{n+1}, etc., is cyclic, has all different adjacent terms, is entirely in φ and satisfies the target condition because the states in φ satisfy the variable forcing condition. φ is therefore a possible terminal set τ of this allowed sequence.

3. METHODS FOR FINDING EQUIVALENT STATES

The discussion of an asynchronous machine amounts essentially to finding the equivalence sets of all its states. This can be done by the following algorithm.

Equivalence Algorithm

1. Choose the necessary number of cardinal points (say s) inside the logical diagram and establish (for a given fixed input) the relationships between states and target states, i.e., determine

$$y_i' = f_i(y_i,\ldots,y_s)$$

2. Assign states $Y_1,\ldots,Y_s$ to all the possible combinations of y's.

3. Draw up a table of target states by listing alongside each present state Y_j the corresponding $(y_1,\ldots,y_s)$ combination and calculating from it the y_i's. List the y_i's in order on the same line and via the assignment of Step 2 determine the target state corresponding to Y_j, i.e., Y_j'.

4. Varying all signals differing in Y_j and Y_j' one at the time, two at the time, etc., calculate all other states surrounding Y_j and draw up a table of surrounding states.

5. Take a state, say Y_j, and investigate how it is connected to its surrounding states, i.e., if $Y_j \to Y_k$, investigate whether there is a sequence from Y_k back to Y_j (meaning that $Y_j \mathcal{F} Y_k$ and $Y_k \mathcal{F} Y_j$, that is $Y_j \mathcal{E} Y_k$). To this effect draw up a stepping diagram as follows:

   a. The first column contains Y_j.

   b. The second column contains all states surrounding Y_j except Y_j itself.

   c. Examine this column and strike out all equilibrium states or states leading solely to equilibrium states in a few steps (scan the table of surrounding states for this!).

   d. Next strike out in this same column all states leading to other states in the column or to the left of it in very few steps.

   e. Finally strike out in this column all states leading to the same states as another entry in the column in very few steps. Do not strike out this other entry.

   f. Iterate steps c. through e. after having formed a third column containing all states surrounding the states in the second column (except for these states themselves!).

6. The process in Step 5 will reduce the possibilities for a path back to Y_j. As soon as we find such a path, we terminate the process and we know that $Y_j \mathcal{E} Y_k$. If, however, we find only paths that avoid Y_j (in particular if we only find a closed cycle leading back to Y_k without touching Y_j) we know that Y_k is in another equivalence set.

Example. Take the circuit shown in Figure 6-32 in which the element Γ is defined by the fact that for it

$$\text{output} = (\text{input 1} \vee \overline{\text{input 2}})(\overline{\text{input 3}})$$

We clearly need four cardinal points (namely the outputs of the four elements) and the circuit equations are

$$y_1' = y_2 \vee y_3$$

$$y_2' = \bar{y}_1$$

$$y_3' = \bar{y}_4(y_1 \vee \bar{y}_2)$$

$$y_4' = \bar{y}_3$$

We now assign to all combinations (0,0,0,0) through (1,1,1,1) the states $Y_0 \ldots Y_{15}$ when the index is simply the decimal equivalent of the binary combination. The target state table is shown below.

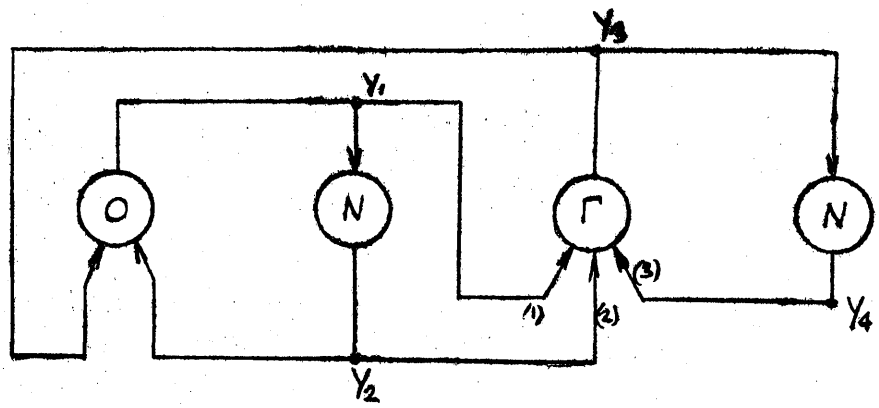


Figure 6-32 Example of an Asynchronous Circuit

TABLE 6-4

Table of Target States

State	y_1	y_2	y_3	y_4	y_1'	y_2'	y_3'	y_4'	Target State
Y_0	0	0	0	0	0	1	1	1	Y_7
Y_1	0	0	0	1	0	1	0	1	Y_5
Y_2	0	0	1	0	1	1	1	0	Y_{14}
Y_3	0	0	1	1	1	1	0	0	Y_{12}
Y_4	0	1	0	0	1	1	0	1	Y_{13}
Y_5	0	1	0	1	1	1	0	1	Y_{13}
Y_6	0	1	0	0	1	1	0	0	Y_{12}
Y_7	0	1	1	1	1	1	0	0	Y_{12}
Y_8	1	0	0	0	0	0	1	1	Y_3
Y_9	1	0	0	1	0	0	0	1	Y_1
Y_{10}	1	0	1	0	1	0	1	0	Y_{10}
Y_{11}	1	0	1	1	1	0	0	0	Y_8
Y_{12}	1	1	0	0	1	0	1	1	Y_{11}
Y_{13}	1	1	0	1	1	0	0	1	Y_9
Y_{14}	1	1	1	0	1	0	1	0	Y_{10}
Y_{15}	1	1	1	1	1	0	0	0	Y_8

The next step is to draw up the surrounding state table:

TABLE 6-5

Surrounding State Table

State	Target State	Other Surrounding States
Y_0	Y_4	Y_1 Y_2 Y_3 Y_4 Y_5 Y_6
Y_1	Y_5	--
Y_2	Y_{14}	Y_6 Y_{10}
Y_3	Y_{12}	Y_0 Y_1 Y_2 Y_4 Y_5 Y_6 Y_7 Y_8 Y_9 Y_{10} Y_{11} Y_{13} Y_{14} Y_{15}
Y_4	Y_{13}	Y_5 Y_{12}
Y_5	Y_{13}	--
Y_6	Y_{12}	Y_4 Y_{14}
Y_7	Y_{12}	Y_4 Y_5 Y_6 Y_{13} Y_{14} Y_{15}
Y_8	Y_3	Y_0 Y_1 Y_2 Y_8 Y_9 Y_{10}
Y_9	Y_1	--
Y_{10}	Y_{10}	--
Y_{11}	Y_8	Y_9 Y_{10}
Y_{12}	Y_{11}	Y_8 Y_9 Y_{10} Y_{13} Y_{14} Y_{15}
Y_{13}	Y_9	--
Y_{14}	Y_{10}	--
Y_{15}	Y_8	Y_9 Y_{10} Y_{11} Y_{12} Y_{13} Y_{14}

Let us now take a state, say Y_0. We see that it is surrounded by Y_1 and the question is: can we go back from Y_1 to Y_0 by some path? Here we need not even draw up a stepping diagram since the only sequence starting with Y_1 is

$$Y_1 \rightarrow Y_5 \rightarrow Y_{13} \rightarrow Y_9 \rightarrow Y_1$$

This sequence avoids Y_0 and Y_1 is therefore not in the same equivalence set as Y_0.

Let us now try the next state that surrounds Y_0, namely Y_2. Here we use a stepping diagram:

TABLE 6-6

Stepping Diagram for Y_2 to Y_0

Y_2	Y_6	Y_4	~~Y_5~~
	~~Y_{10}~~	Y_{12}	Y_8
	~~Y_{14}~~	~~Y_{14}~~	Y_9
			Y_{10}
			Y_{11}
			Y_{12}
			Y_{13}
			Y_{14}

We note that in the fourth column (after striking out Y_5 because we know that we can only have $Y_5 \rightarrow Y_{13} \rightarrow Y_9 \rightarrow Y_1 \rightarrow Y_5$!) Y_8 will lead back to Y_0. Continuing this process for all states and all their surrounding states, it turns out that there are four equivalence sets:

α: $\{Y_0 \quad Y_2 \quad Y_3 \quad Y_4 \quad Y_6 \quad Y_7 \quad Y_8 \quad Y_{11} \quad Y_{12}\}$

(Note that not all surround Y_0; some surround other states in the set.)

β: $\{Y_1 \quad Y_5 \quad Y_9 \quad Y_{13}\}$

γ: Y_{14}

δ: Y_{10}

It is to be remarked that set γ is not final, although it contains just one state. This state cannot be an equilibrium state by a preceding theorem. The partial ordering of the sets is shown in Figure 6-33. Note that it can be seen that α itself is pseudo-final: it is variably forced.

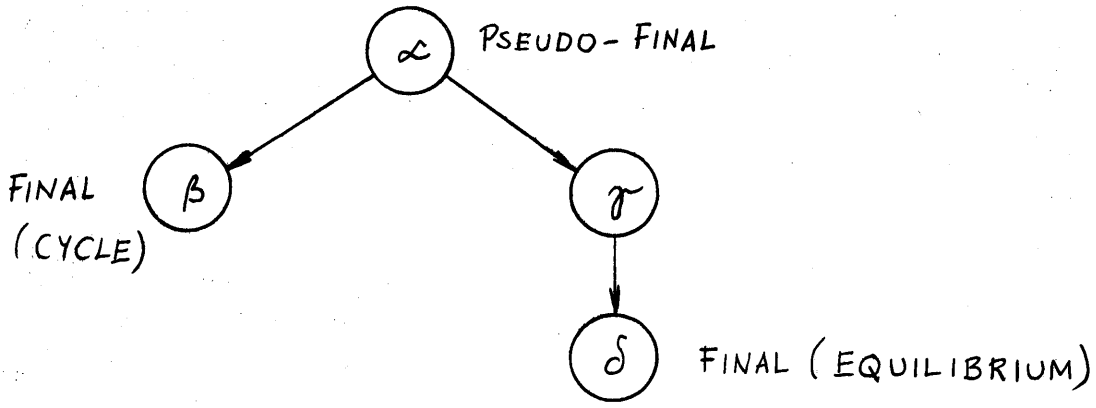


Figure 6-33 Equivalence Sets for the Circuit Shown in Figure 6-32

6.7 Speed Independent Circuits

1. SPEED INDEPENDENT AND TOTALLY SEQUENTIAL CIRCUITS

We have proved that an asynchronous circuit attains either an equilibrium state or that it cycles in a final or pseudo-final set. In the first case we have true static equilibrium, in the second case a sort of "dynamic" equilibrium. However, we must realize that for a given initial state A there are many possible pseudo-final and final sets following the equivalence set of A. Each one of

them is a potential candidate for the terminal set of an allowed sequence starting with A. This means that in general a given initial state can lead to almost any machine behavior. We shall now introduce "speed independence," i.e., a foreseeable machine behavior by a new definition.

Definition 10: A circuit is "speed independent" with respect to an initial state A (we shall then write si(A)) if every allowed sequence starting with A ends up in the same terminal set τ.

Theorem 11. A circuit is si(A) if and only if the equivalence set α of A is followed by a single final set and no pseudo-final set.

Proof: Let φ be the single final set. Then there is an allowed sequence beginning in A which is ultimately trapped in φ. Since there are no pseudo-final sets it cannot get trapped on its way to φ and there can be no other sequences leaving A and not attaining φ because they would have to end in a pseudo-final set (and there is none which follows α) or a final set (and there is nonedifferent from φ). It is seen that the condition is not only sufficient but that it is necessary by a similar reasoning.

Definition 11: A circuit is "totally sequential" with respect to an initial state A (we shall then write ts(A)) if there is only one allowed sequence starting with A.

Theorem 12. A circuit which is ts(A) is also si(A).

Proof: The only allowed sequence leaving A will (as any allowed sequence be trapped after some time in a terminal set. This must be a unique final set (for if there had been another one, there would have been another allowed sequence leading into it) and there cannot be any pseudo-final sets in between because then there would be two allowed sequences: the one trapped in the pseudo-final set and the on trapped in the final set.

Theorem 13. In a totally sequential circuit only one signal changes at a time (i.e., parallel action is excluded!).

Proof: Since the target state sequence can always be constructed, it is clear that in a totally sequential circuit the one and only

allowed sequence is precisely the target state sequence, i.e., $A(j+1) = A'(j)$. If two signals were to change we would by the construction of surrounding states have more than $A'(j)$ following $A(j)$ and there would be other sequences. This not being the case, only one signal can have changed.

2. SEMI-MODULAR CIRCUITS

It becomes apparent that although a totally sequential circuit is safe in the sense that it is actually speed independent, the advantages of parallel operations cannot be reaped. Happily there are speed independent circuits which are not totally sequential, the prime example being semi-modular circuits.

Definition 12: A circuit is "semi-modular" with respect to an initial state A (we shall then write sm(A)) if for a state C surrounding a state B in a sequence starting with A we can establish that B' surrounds C, i.e.,

$$\left.\begin{array}{l} \text{if} \quad B \rightarrow C \\ \qquad C \rightarrow B' \end{array}\right\} \tag{6-41}$$

Theorem 14. In a semi-modular circuit a node which is excited remains excited or acts as we go to the next state but its excitation does not disappear before it has acted.

Proof: Suppose that in state B (surrounded by C) node i is excited, i.e., that $b_i' \neq b_i$. Then the semi-modularity conditions show that we must have simultaneously:

$$B \rightarrow C \text{ meaning } \left.\begin{array}{l} \textcircled{1}\ c_i = b_i' \\ \text{or } \textcircled{2}\ c_i = b_i \end{array}\right\}$$

$$C \rightarrow B' \text{ meaning } \left.\begin{array}{l} \textcircled{3}\ b_i' = c_i \\ \text{or } \textcircled{4}\ b_i' = c_i' \end{array}\right\}$$

If (1) is true, node i has effectively changed and condition (3) is automatically satisfied. If (2) is true, (3) is untrue and therefore (4) must be true: $c_i' = b_i'$; here we have thus $c_i' = b_i' \neq b_i = c_i$, i.e., in state C we still have this node excited.

We must now establish that semi-modularity does indeed mean speed independence. This will necessitate the introduction of the notion of min-max state and of parallel sequence:

Definition 13: In a circuit which is sm(A) let K be a state following A and let B and C be states surrounding K. We then define the "min-max state" M of B and C with respect to K (written as M = Mm K[B,C]) by its components $(m_1, \ldots, m_s)$:

$$m_i = \begin{cases} \max (b_i, c_i) & \text{if } k_i < k_i', \text{ i.e., } k_i' = 1 \\ \min (b_i, c_i) & \text{if } k_i > k_i', \text{ i.e., } k_i' = 0 \\ k_i & \text{if } k_i = k_i' \end{cases}$$

or symbolically

$$m_i = \text{min-max } k_i(b_i, c_i) \qquad (6\text{-}42)$$

Theorem 15. M surrounds K, B and C.

Proof: First we show that M surrounds K, i.e., that $m_i = k_i = k_i'$ whenever $k_i = k_i'$. This is evident because $K \rightarrow B$ and $K \rightarrow C$ means that $k_i = k_i'$ implies $b_i = c_i = k_i = k_i'$ and therefore $m_i = \text{min-max } k_i(b_i, c_i) = k_i = k_i'$.

Now let us show that M surrounds B, i.e., that $m_i = b_i = b_i'$ whenever $b_i = b_i'$. Because of semi-modularity we have $B \rightarrow K'$ or $k_i' = b_i = b_i'$. There are three sub-cases:

(1) If $k_i = k_i'$ we have the case above, i.e., $m_i = \text{min-max } k_i(b_i, c_i)$ with $b_i = c_i = k_i = k_i'$ and therefore $m_i = k_i = b_i = b_i'$ because of our hypothesis that $b_i = b_i'$.

(2) If $k_i < k_i'$, this means that $k_i' = 1$ and since $k_i' = b_i = b_i'$ we see that $b_i = 1$ and $m_i = \text{min-max } k_i(b_i, c_i)$ becomes $m_i = \max (b_i, c_i) = 1 = b_i = b_i'$.

(3) If $k_i > k_i'$, this means that $k_i' = 0$ and since $k_i' = b_i = b_i'$ we see that $b_i = 0$ and $m_i = \text{min-max } k_i(b_i, c_i)$ becomes $m_i = \min(b_i, c_i) = 0 = b_i = b_i'$.

Definition 14: Suppose that we have a sequence (not necessarily satisfying the target condition and therefore not necessarily allowed!) B(0), B(1), B(2), ... B(j), B(j+1) ... and that B(0) is surrounded by a state $C(0) \neq B(0)$. We can then construct iteratively a "parallel sequence" as follows: take B(0), B(1) and C(0) and take

$$\begin{aligned} C(1) &= Mm\ B(0)[B(1), C(0)] \\ C(2) &= Mm\ B(1)[B(2), C(1)] \\ &\vdots \\ C(j+1) &= Mm\ B(j)[B(j+1), C(j)] \end{aligned} \tag{6-43}$$

obtaining

$$\begin{array}{ccccccccccc} B(0) & \rightarrow & B(1) & \rightarrow & B(2) & \ldots \rightarrow & B(j) & \rightarrow & B(j+1) & \rightarrow & \ldots \\ \downarrow & \searrow & \downarrow & \searrow & \downarrow & & \downarrow & \searrow & \downarrow & & \\ C(0) & \rightarrow & C(1) & \rightarrow & C(2) & \ldots \rightarrow & C(j) & \rightarrow & C(j+1) & \rightarrow & \ldots \end{array}$$

where the arrows have their usual significance of "surrounded by" by virtue of Theorem 15.

We see that the parallel sequence we have formed has the property that each of its terms surrounds the preceding term of the new sequence and also the two corresponding terms in the original sequence.

Theorem 16. Let P(0), ... P(r), P(0) be a cyclic sequence (not necessarily allowed, i.e., not necessarily fulfilling the target condition) and Q(0), ... Q(r), Q(0) the parallel sequence constructed by (6-43). Then for any node i for which

1. $p_i(j) = q_i(j)$ we also have $p_i(j+1) = q_i(j+1)$
2. $p_i(j) < q_i(j)$ we also have $p_i(j+1) < q_i(j+1)$ (in this case $p_i(j) \leq p'(j)$!)

3. $p_i(j) > q_i(j)$ we also have $p_i(j+1) > q_i(j+1)$ (in this case $p_i(j) > p_i'(j)$!)

i.e., <u>all inequalities and equalities between pairs in parallel cycles are propagated through the whole cycle</u>.

<u>Proof</u>: Let us first discuss the case <u>$p_i(j) = q_i(j)$</u>. There are three sub-cases: (1) $p_i'(j) = p_i(j)$, (2) $p_i'(j) > p_i(j)$ and (3) $p_i'(j) < p_i(j)$.

<u>Case (1)</u>. Remembering that

$$\begin{array}{ccccc} \rightarrow P(j) & \rightarrow & P(j+1) & \rightarrow & \\ \searrow \downarrow & & \searrow \quad \downarrow & & \searrow \\ \rightarrow Q(j) & \rightarrow & Q(j+1) & \rightarrow & \end{array}$$

where $Q(j+1) = Mm\ P(j)[P(j+1), Q(j)]$ we see that $p_i'(j) = p_i(j)$ implies that $q_i(j) = p_i(j+1) = p_i(j) = p_i'(j)$ and $q_i(j+1) =$ min-max $p_i(j)[q_i(j), p_i(j+1)] = p_i(j)$ which here is $= p_i(j+1)$, i.e., $q_i(j+1) = p_i(j+1)$. Note that we did not even have to use the hypothesis that $p_i(j) = q_i(j)$!

<u>Case (2)</u>. Here clearly $p_i(j) = 0$ and $p_i'(j) = 1$. By our hypothesis $q_i(j) = 0$. Therefore $q_i(j+1) =$ min-max $p_i(j)[q_i(j), p_i(j+1)]$ $= \max [0, p_i(j+1)] = p_i(j+1)$.

<u>Case (3)</u>. Here clearly $p_i(j) = 1$ and $p_i'(j) = 0$. By our hypothesis $q_i(j) = 1$. Therefore $q_i(j+1) =$ min-max $p_i(j)[q_i(j), p_i(j+1)]$ $= \min [1, p_i(j+1)] = p_i(j+1)$.

We see thus that equalities are effectively carried forward in all cases.

<u>Part 2</u>. Now we have to discuss the case <u>$p_i(j) < q_i(j)$</u>, implying $p_i(j) = 0$ and $q_i(j) = 1$. Let us again split up the discussion into the three sub-cases above:

<u>Case (1)</u>. This case is clearly impossible, since it implies (as shown above) that $p_i(j) = q_i(j)$ quich contradicts $p_i(j) < q_i(j)$.

<u>Case (3)</u>. This case is also excluded since $p_i(j) = 0$ and $p_i'(j) < p_i(j)$ contradict each other.

Case (2). We must therefore have case (2), i.e., $p_i(j) < p_i'(j)$ whenever $p_i(j) < q_i(j)$. This means that $p_i'(j) = 1$ and $q_i(j+1) = \text{min-max } p_i(j)[q_i(j), p_i(j+1)] = \max [1, p_i(j+1)] = 1$. We have yet no proof that $p_i(j+1) = 0$ so as to give $p_i(j+1) < q_i(j+1)$. We do know, however, that $p_i(j+1) \neq q_i(j+1)$ because otherwise by Part 1 we would have all successive pairs equal--coming around in the cycle we would have $p_i(j) = q_i(j)$ which is contrary to our hypothesis in Part 2. Therefore, $p_i(j+1) = 0$ while, as shown, $q_i(j+1) = 1$: this carries the inequality one step forward.

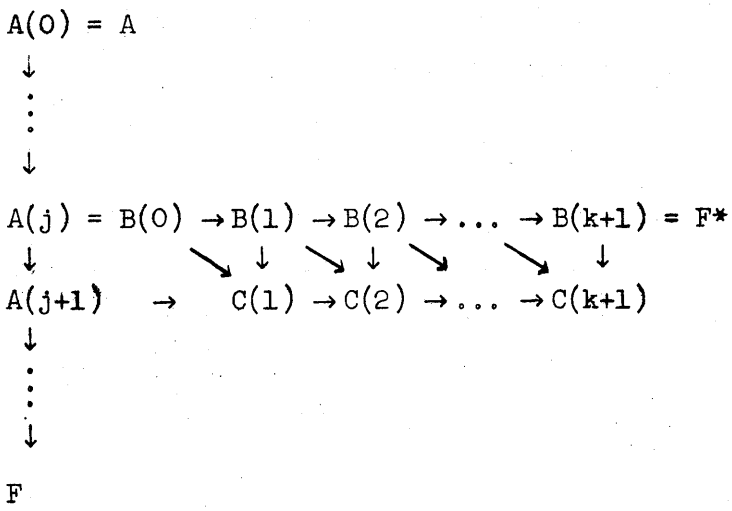
Part 3. Here we suppose that $p_i(j) > q_i(j)$. The reasoning being symmetric in $p_i(j)$ and $q_i(j)$, it is evident that the proof of Part 2 is sufficient.

We now come to the central and final theorem of our discussion.

Theorem 17. A circuit sm(A) is si(A).

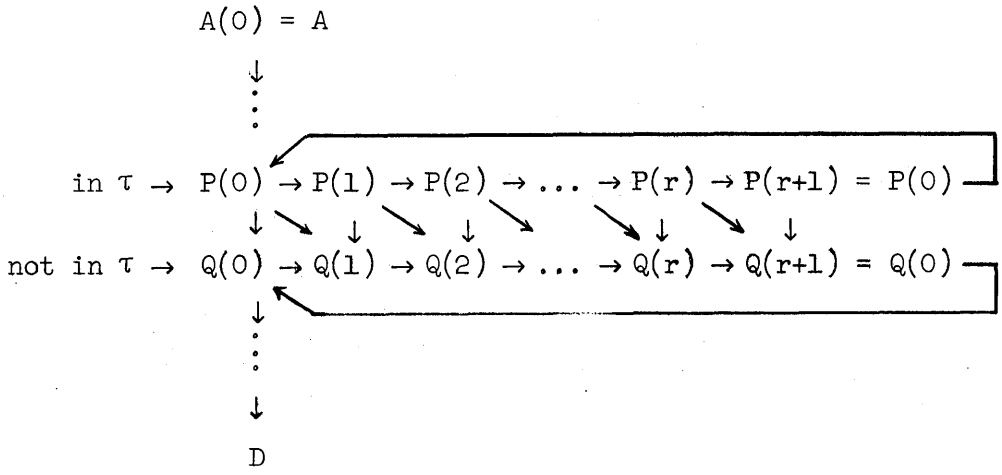
Proof: We shall show that the equivalence set α of A is followed by a single final set φ and no pseudo-final set. Using Theorem 11 we obtain the desired proof.

Part 1. Let us first show that α cannot be followed by two final sets φ and φ^* (which a priori does not exclude that it is followed by a pseudo-final set). Let F be in φ and F* in φ^*. Then we know that we can form sequences A(0), ...,F and A(0), ...,F* where A(0) = A. Evidently $F \neq F^*$, but some states in the sequences may be common to both. Let A(j+1) be the first state in the first sequence from which we can no longer go to F*. Relabel A(j) (from which we can go to F*) simply B(0). Then there is a sequence B(0), B(1),...B(k), F*. Call F* now B(k+1). Then we can construct the parallel sequence C(1)...C(k+1) to B(1) ... B(k+1) by the min-max process, obtaining



Now consider C(k+1): this state cannot be in φ^* for if it were, we could go from it to F* and that would imply that, contrary to our hypothesis, we could go from A(j+1) to F*. Therefore, C(k+1) is outside φ^*. But by the construction of the parallel chain we can go from F* to C(k+1) outside: our assumption that φ^* is final is, therefore, wrong. There can, then, be only a single final set φ following α.

Part 2. Now we must show that α cannot be followed by a pseudo-final set (say τ) composed of states T(0), ..., T(r). Since τ is not final, there must be a set $\delta \neq \tau$ following τ. Let D be a state in δ. We can then construct a sequence T(0), ..., D: in it is a first state--say Q(0) which is not in τ. We can assume that the states in τ can be labelled such that Q(0) surrounds T(0). Now let us construct a cycle T(0) ... T(1), T(1) ... T(2), ..., T(r) ... T(0) containing all states in τ and let us show that this cannot be an allowed sequence, i.e., that the target condition is not satisfied. Note that if we chose a subset of states in τ as our cycle, the hope of satisfying the target condition would even diminish. Let us rename our cycle P(0), ..., P(r); then P(r+1) = P(0). Finally, let us construct the parallel cycle Q(0), ..., Q(r) to P(0), ..., P(r). We then obtain:



Since $Q(0) \neq P(0)$ (They are even in different equivalence sets!), they must differ in at least one signal, say $p_i(0) \neq q_i(0)$. Therefore, we can only have $p_i(0) > q_i(0)$--Case (1)-- or $p_i(0) < q_i(0)$--Case (2).

Case (1). By the proof of Theorem 16 this must imply that $p_i(0) > p_i'(0)$. Since $p_i(0) > q_i(0)$ is propagated and gives $p_i(j) > q_i(j)$ for all j. This also means that for all states of our cycle $p_i(j) > p_i'(j)$, this visibly violates the target condition: our sequence is not an allowed sequence and τ cannot be pseudo-final.

Case (2). The assumption $p_i(0) < q_i(0)$ leads to $p_i(0) < p_i'(0)$ and by iteration to $p_i(j) < p_i'(j)$ for all j. Again the target condition is violated.

Thus there is no pseudo-final set following α but there is a unique final set following it: the circuit (started in state A) is speed independent!

Bibliography for Chapter VI

(in chronological order)

1. D. A. Hoffman: "The Synthesis of Sequential Switching Circuits", J. Franklin Institute, Vol. 257. (1954)

2. G. H. Mealy: "A Method for Synthesizing Sequential Circuits", BSTJ, Vol. 34. (1955)

3. E. F. Moore: "Gedanken-Experiments on Sequential Machines", Automata Studies, Princeton University Press (1956)

4. D. E. Muller and W. Scott Bartky: "A Theory of Asynchronous Circuits 1". Digital Computer Laboratory of the University of Illinois, Report 75. (1956)

5. D. E. Muller and W. Scott Bartky: "A Theory of Asynchronous Circuits II". Digital Computer Laboratory of the University of Illinois, Report 78. (1957)

6. D. D. Aufenkamp and F. E. Hohn: "Analysis of Sequential Machines". IRE Transactions on EC, Vol. 6. (1957)

7. F. E. Hohn, S. Seshu, and D. D. Aufenkamp: "The Theory of Nets". IRE Transactions on EC, Vol. 6. (1957)

8. W. S. Humphrey, Jr.: "Switching Circuits with Computer Applications". McGraw-Hill. (1958)

9. D. D. Aufenkamp: "Analysis of Sequential Machines II", IRE Transactions on EC, Vol. 7. (1958)

10. S. H. Caldwell: "Switching Circuits and Logical Design". Wiley. (1958)

11. D. E. Muller. Lecture Notes for "University of Illinois Math-EE 391". ("Boolean Algebras with Applications to Computer Circuits I") (1958)

12. F. E. Hohn. Lecture Notes for "University of Illinois Math-EE391". ("Boolean Algebras with Applications to Computer Circuits I") (1958)

13. R. E. Miller: "Switching Theory and Logical Design of Automatic Digital Computer Circuits". IBM Report RC-473 (1961) Also, equivalent "University of Illinois Math-EE 394 Lecture Notes". (1960)

14. D. E. Muller. Lecture Notes for "University of Illinois Math 489". ("Asynchronous Circuit Theory") (1961)