"This algorithm offers significant reduction in processing time and in memory requirements . . . "

A Logic Function Reduction and Minimization Algorithm for Microcomputers

by

Rafael S. Ramirez and Luis M. Alarilla, Jr., Ph.D.

ABSTRACT

Combinational logic forms the basis of any digital system when decomposed into its basic elements. Of particular significance in this area is the reduction and minimization of a combinational logic function. Besides being a necessary educational stepping stone towards more complex sequential logic systems, it still bears its importance today in fault analysis of digital systems, applications involving Field Programmable Logic Arrays (FPLAs), or chip-level design requirements for LSI and VLSI technology.

This paper proposes and describes a departure from the more popular Quine-McCluskey computer solution in the form of an algorithm based on variable partitioning, combination generation and searching techniques for the prime implicants. Minimization is done by using a Zero-One Integer Programming model of the problem where the implicants and the logic function minterms or maxterms form a set of constraints to the problem and an objective function is used based on a pseudo-cost coefficient for each implicant requirement on fan-in and number of inverters.

The algorithm was implemented in a computer program called BOZER (acronym for Boolean Zero-One Reduction) in a 64K microcomputer system using Microsoft's MBASIC interpreter under a CDOS (CPM-enhanced) operating system. The present program can handle up to 13 state variables. Even if the implementation is only in a microcomputer system the processing time is still reasonable. It is predicted that for larger problems run on larger computer systems, the algorithm will show a significant reduction of processing time and memory requirement when compared to other algorithms written in the same language and run on the same machine. Finally, an algorithm expansion for the multi-output case is described.

INTRODUCTION

Minimization and Combinational Logic Design

The processes by which a combinational logic function is converted into a workable and practical logic gate implementation may be summarized as follows:

1. Reduction – is the elimination of unnecessary terms which will result in redundant logic gates, rather than a simpler and cost effective implementation. The bases of this process are the absorption and logic adjacency theorems of Boolean Algebra which are successively applied until irreducible expressions or terms are reached (known as prime implicants).

2. Minimization – is the selection of a subset of the prime implicants that satisfy the truth table of the function and chosen with the following objectives:
   a) Minimum number of inputs per gate (i.e., minimum fan-in) to reduce the board - or chip-level electronics, power consumption and size.
   b) Minimum overall gate count to maintain the simplicity and minimal cost of the implementation.
   c) Optionally, to also minimize the use of outboard inverter gates, thus allowing the maximum utilization of NOR/NAND internal inverters.
3. Realization – the selection of currently available devices to implement the chosen subset.

These three processes represent a practical engineering approach towards the design of the circuit implementation of a Boolean Logic Function. It is important to note that the first two processes assume a gate-level implementation that is relatively biased to Small Scale Integration or discrete technology of the past era.

Recently, however, the introduction of LSI chips has provided the opportunity of implementation without regard to minimization or reduction processes at a lower cost and substantially shorter design time. For instance, the current state-of-the-art allows the use of an EPROM that can simply be burned-in or re-used to generate multiple combinational functions without any regard to minimization, although the on-chip logic wasted is typically enormous in magnitude.

Yet, this practice is widely accepted because it is both cost-effective and reliable in such random logic, multi-purpose production applications. Hence, the relative need for absolute minimization has been reduced somewhat by the advances in LSI technology.

On the other hand, large scale minimization processes are still a fundamental necessity in areas where on-chip logic utilization is of primary concern. In VLSI design, where the silicon real estate conservation is the most important, any reduction or minimization possibilities must always be explored and considered. Users of Field Programmable Logic Arrays (FPLAs) also need an optimal utilization of its limited on-chip logic capabilities. Current interest in FPLAs has arisen due to the inherent limitations of the ROM when the number of function inputs or state variables approach a large magnitude. The fundamental limitation of the ROM lies in its state variable to canonical form encoder. This encoder accepts N inputs and generates the 2^N canonical terms which are coupled by fusible links to a large fan-in OR/AND gate, as required. Of course, this is a consequence of a general-purpose device design direction but, say for 10 state variables, the encoder needed should be 10 to 1024 – even worse, the final gate in the ROM will have to have this large fan-in capability. In this extreme case, the trade-off from a general purpose device as against its chip logic waste is no longer practical nor acceptable – in other words, there comes a time when there is neither enough chip area on a ROM to accommodate such requirements nor to accept the wasted space.

There is also the case of a logic designer who may use large scale minimization processes on a regular basis, say in research or for custom-made applications, low volume one-of-a-kind applications. Another application of particular importance is the use of the list of the prime implicants in transient hazard analysis of sequential systems, or fault analysis of digital systems in general.

Finally, in the area of education, the availability of a useful teaching aid on prime implicant extraction and minimization is a practical education tool, especially since the logic gate level is used as the logical starting point in any digital systems course.

The main thrust in algorithm development for reduction and minimization has been geared towards solving large scale problems efficiently at a low cost and short processing time.

This automatically outcasts the hand-reduction methods which are more or less limited to six state variable problems and even that, with a great degree of difficulty. Hence, computer based algorithms are used to meet the requirements mentioned.

Computer Solutions and Desirable Attributes

Some of the basic considerations involved in the development of a computer algorithm for reduction and minimization may be identified as follows:

1. Memory requirements and Data representation which will place an upper bound on the state-variable size and state term size that can be handled by the algorithm.
2. Computer language requirements, particularly on the type of mathematical or bit-by-bit logical operations available and I/O capability which will also have bearing on the processing time required.
3. Algorithm control structures, which will have to be designed such that specific routines may be repeated continuously through block or modular program structures.
4. Generality, which is to prevent extensive modifications when the algorithm or its program implementation is used for different computer types using an assumed minimal system requirement.

The relatively tight interrelationship of these considerations with one another may be understood in the sense that the practical, overall implementation of the algorithm should be designed to be generally user-friendly and not conform to a privileged user group.

One of the most widely publicized computer algorithms for minimization and reduction of a Logic Function is the Quine-McCluskey (Tabular method) or variations of this. Numerous other algorithms also exist which use topological or mathematical approaches to these two processes.

To date, there is no real quantitative comparison of all the different computer algorithms even though extensive research has been done in this area. However, Fletcher [1] has proposed a set of desirable attributes for these algorithms and they are:

1. Speed, i.e., it should be faster compared to the currently used system
2. Efficiency
3. Guaranteed minimal cover subset of the prime implicants
4. Flexible data input formatting to allow for:
   a) Standard minterm/maxterm lists
   b) Map entered variable notation (from K-maps)
   c) Partially simplified expressions
   d) Free use of don't cares
5. Processing time mainly related to problem complexity rather than the number of state variables or state of the given Logic Function
6. Selectable output options such as:
   a) All or any Sum of Products (SOP) or Product of Sums (POS) minimal cover listings.
   b) A complete list of prime implicants for transient hazard analysis.
   c) All or any optimal solutions based on a given set of prime implicant constraints.
   d) All or any optimal solutions based on the minimal use of inverters.
   e) All or any solutions for a multiple-output problem.

Practical algorithms that have been developed satisfy only a few of these desirable attributes for many reasons. The usual trade-off is in problem size capability as against memory availability. The minimization process will need to tag the minterms or maxterms covered by the implicants to allow constraint equations to satisfy the truth table requirements

of the function, and as the number of these canonical terms increases, this tag array size will also increase (it is possible to eliminate it, at the cost of additional backtrack processing time). Another factor involved in memory allocation will be the array used to store the minterms or maxterms. Some algorithms will use a size equal to 2^N number of state variables to store the logic function states (i.e., the entire truth table is actually stored with the subscript equal to the row number) and the processing time will be significantly shorter. Others use only the needed minterms or maxterms, with the consequence of using searching techniques and sorting routines.

Secondly, most algorithms are geared towards single-output function problems, with extension possibilities for the multi-output case. The main change in the processing for a multi-output problem is that the prime implicants for each output must be available, and for all outputs some implicants may be unique to itself (disjoint) or may be shared by another output (joint). The minimization process will now be aimed at using the joint implicants as much as possible, since the consideration is now for the minimum total system-cost for all output implementations.

Finally, complete prime implicant lists are usually traded off for partial listings to save on processing time. Again, this is due to the memory space problem, even though the complete list is necessary for each output in a multi-output problem.

Algorithm Proposal

Development and History

The computer algorithm for the minimization and reduction of a Boolean Logic Function proposed here is implemented in a program called BOZER for Boolean Zero-one Reduction. Originally described in Fletcher (program implementation was called BOOZER), it was developed and implemented at the Electrical Engineering Department of the University of Utah in 1979. Written in extended ALGOL-60 for a Burroughs 7600 stack mainframe computer, the algorithm introduced new approaches and concepts toward the minimization and reduction problem.Its use and conceptual development are introduced by William Fletcher [1] in his chapter on Combinational Logic in the light of the concepts of the Karnaugh map method of reduction. Foremost among its features are the use of Karnaugh map concepts on variable partitioning, the use of zero-one integer programming approach to minimization and a walking search routine for the prime implicants.

Although the claim to processing speed reduction is biased by the intermediate code Burroughs machine, which is specifically hardware enhanced for the ALGOL-60 language, the features of the Boozer algorithm truly bound the problem so as to eliminate unnecessary searches and processing requirements as compared to the Quine-McCluskey method. Unfortunately, the memory requirements involved are rather large. Specifically, the problem is represented in its entirety by using an array of size 2^N for a problem of N state variables (seemingly, the program could handle up to a 30 variable problem, but this would require a 2^{30} or 2M size array! – even worse, each integer is 2 bytes of 16 bits each, so a directly addressable main memory of 4Mbytes is necessary).

The algorithm presented in this paper is a modification of Fletcher's algorithm to particularly suit the limitations of a minimal 64K microcomputer system, so that much of the original concepts were either revised or enhanced as necessary. Inasmuch as the BOOZER program in Fletcher is described only in a general sense, and more details are not directly available even from decoding the ALGOL-60 listing, only those parts which significantly contribute to the speed and efficiency of the reduction/minimization processes have been adopted, while the rest are original in nature. Of particular difference is the use of extensive searching techniques, which are non-existent in Fletcher's algorithm, to reduce the memory requirements without the expense of prolonged processing time. In essence, the proposed algorithm has been developed with the aim of satisfying the desirable attributes for computer algorithms as discussed in the previous section.

The chosen programming language was necessarily restricted by the requirements of the available floppy-disk based microcomputer system. To allow for portability from different machine brands, the program BOZER is implemented in Microsoft's MBASIC language which is a standard interpreter that is available under a CP/M operating system (the program was developed under the CDOS operating system for a Cromemco System Three Microcomputer, but this is simply an enhanced CP/M). The language was chosen for the following reasons: a) the availability of its compiler, so that the program was entered, edited and tested under the interpreter and then compiled and linked to convert it into object code for faster execution; b) the availability of logical bit-by-bit operations, string functions and 40 character variable names to allow for ease of documentation, all-integer operations and ease in input/output; c) the ease of input/output allowed by the language, to give more concern to the main routines and; d) the capability of program chaining to permit a modular program structure and to remove the restrictions in program memory space – this allows BOZER to be composed of 4 disk files, chained together sequentially with only the necessary variables passed among them in memory.

Overview

The increased processing speed and reduced memory requirements of the proposed algorithm are mostly the result of the use of a walking search routine for the prime implicants, combined with the elimination of unnecessary implicants, the compact data representation and the labelling of the status of prime implicants extracted to eliminate unnecessary implicants in the minimization process. Information is derived from the minterms or maxterms of the function which bound the problem according to its complexity rather than its size in terms of state variables and number of terms, on both the minimization and reduction processes. The three main processes of the algorithm are:

1. Generation of information for:
   a) All the ways each minterm may be involved in any grouping of a given size.
   b) The largest group size to be searched for in the problem.
   c) Identification of the type of implicant that results from the set of minterms the implicant covers.
2. Walking search for all the prime implicants that are feasible, bounded by the information in (1).
3. Minimization process that uses an integer model of the problem's implicants, bound by constraints and an objective function.

In the process of implicant extraction, groupings which are not feasible are quickly eliminated and those extracted are tagged to eliminate them in further searches, to know their type whether essential; necessary; redundant or don't care and to generate the constraint equations which indicate the minterms covered by the implicant. No two searches are ever conducted which will result in the same implicant, and the complexity of group combinations available in the problem determines the speed by which the total prime implicant list is generated. This reduction process is therefore akin to the manner a person scans and identifies all the geometric groupings in a function represented on a Karnaugh Map.

All the essential information for reduction are generated by taking the first partition on all state variable boundaries for all minterms or maxterms. This may be viewed as a "folding" of a K-map about itself on a variable boundary so that all the adjacent minterms or maxterms in the problem are made obvious and logically available for groups searches of implicants. It is from the resulting partition byte formed by the total partitions on all variable boundaries that: a) the row sum vector is derived to indicate the group size available for any minterm; b) the largest grouping size to be searched for in the problem; and c) the implicant type that may be extracted if it is found to be feasible, including the variables eliminated in the grouping. Since all reduction is based on the partition array for all minterms, this is radically different from the Quine-McCluskey reduction process since a) it does not generate any partial results; b) the implicant type is at once known once it is extracted; c) no reductions are performed

which will result in the same prime implicant; and d) there is only one search pass through all the minterms and all the prime implicants are at once generated in the search. The partitioning of the problem, analogous to the first cycle in the Quine-McCluskey algorithm, therefore bounds the reduction process and the true reduction speed is limited by the complexity of the groupings available for the problem.

The minimization process is a selection or assignment problem wherein the implicants are assigned pseudo-cost coefficients to form an objective cost function and the minterms or maxterms of the logic function, which are covered by each implicant, form a set of constraint equations. The representation allows the zero-one integer programming algorithm developed by Egon Balas [2] to be used to minimize the cost objective function without violating any of the constraints. All the coefficients are integers and an implicant is either chosen or not chosen to be included in the minimal subset (hence the zero-one nature of the minimization process). The satisfaction of the constraints is the same as saying that the minterms or maxterms of the problem must at least be covered by one of the chosen prime implicants. The cost of each implicant is based on the number of variables in the expression and the number of inverters required to generate the expression. Hence, the minimization process will choose the implicants which use less number of terms, the one with less inverters. The representation allows the choice of the larger groupings first, much like the visual location of larger groupings on a K-map.

The choice of the algorithm for minimization developed by Egon Balas is due mainly to its processing speed and minimal memory requirements. The number of constraints do not grow, unlike in algorithms such as the Simplex method. Also, only addition and subtraction are used throughout the minimization process and since all the involved terms are integers, the processing speed will be much faster compared to any other. The drawback will be that as the number of trees to prune in the solution tree grows, the algorithm takes a longer time and may yield the optimal solution only after numerous backtracks. However, this is offset by the processing speed advantage and the algorithm assures the minimal cost subset to be chosen in any case.

As a final note, the algorithm proposed has been developed for only single output problems, although expansion to problems of multiple output can also be tackled by the algorithm with additional memory and processing requirements. The multi-output problem will require the prime implicant list for each output and the constraint equations and cost coefficients are modified so that shared implicants are used as much as possible in the minimal cover subset, since the total system cost is now the objective.

Data Representation and Notations

1. Function Specification and Canonical Terms

The algorithm uses integers to represent the Boolean Function's states, minterms and prime implicants. Each bit, from the least significant to the N-1th, is used to represent the assertion levels of each variable for N state variables in the minterm or maxterm list. Hence, any canonical form that may be stored is limited by the integer bit length as implemented by the high level language used to implement the algorithm.

A canonical term is represented by N state variable symbols with a ' for non-asserted variables and the symbol alone for the asserted variables. The corresponding integer representation for the canonical terms is implemented by using the set bits or bits with the 1 level for asserted variables and 0s for the non-asserted. For any integer used to implement a canonical term, only the 0 to N-1 bit positions are used throughout the reduction and minimization processes. The advantage of this is that row numbers corresponding to the canonical term row number is the decimal equivalent of this integer representation.

For example, a canonical term A'BCD' has B & C asserted and A & D as non-asserted. If the agreed upon bit positions are such that A corresponds to the 3rd significant bit and D to

the least significant or 0th bit position, then the binary integer representation is 0110. Since the integer width is usually fixed to eight bit bytes, then the internal representation is actually 00000110 which is 6 in decimal. Note that if the truth table were constructed, the canonical terms corresponding to A'BCD' will be located at row number 6 (rows are from 0 to 2^{N-1} or 15). Hence, the representation allows an easy input format for the canonical terms of the function. Another advantage will be that bit-by- bit logical operations are usually available in any language since this is the manner by which logical branches are evaluated.

The minterms or maxterms are kept in a vector whose length varies as the number of actual canonical terms required by the function. This means that the physical and logical locations of the canonical terms are generally not the same. Actually, a large advantage can be gained if we could use the subscripts of the array or vector to represent each canonical term and using the entire vector as a logical variable. This would speed up processing time since any term can be accessed simply by using the row number which is the subscript of the term. The disadvantage will be that the memory requirements will grow up by 2^N for the case of N state variables. Also, the vector itself is usually sparse for most problems, so the wasted memory space is substantial. The matter becomes more critical when applied to small computer systems, since the address space for arrays is usually limited, aside from the other arrays required for processing the problem.

The minterm vector is therefore chosen to be a variable length vector, wherein the minterms are sorted in ascending order to facilitate the searches in the reduction process, rather than allow the unnecessarily large memory requirements if the minterms were physically located as they are logically in a truth table.

The choice of minterms or maxterms to specify the problem is left to the user, and the only essential difference will be in the retranslation of the integers into their equivalent Boolean expressions. In any problem, if minterms are chosen for reduction and minimization, then the maxterms are simply excluded from the canonical vector. However, the don't cares should be included in the vector to allow their possible groupings with other canonical terms available in the generation of the prime implicant list. To allow for this, a Function status vector is used to indicate whether each term in the canonical vector is a don't care or a canonical term. The algorithm uses a 1 in the Function vector to indicate Canonical terms and 0 for don't cares. This is again a variable length vector of the same length as the Canonical Vector.

Finally, the variable symbol table is also left to the specification of the user by storing this in a string vector of length equal to the number of state variables. The vector will be used in the retranslation of the prime implicants to their Boolean expressions using string catenation operations.

The following notations are now introduced to facilitate further discussions on the proposed algorithm:

let
- m = number of canonical terms (don't cares included)
- n = number of state variables for the problem
- i = variable denoting the physical term location where i has any value from 1 to m
- j = variable denoting the jth bit position of a term where j has any value from 0 to n-1

The problem is therefore specified by the following vectors:

- Mi = integer canonical term vector, where only bits 0 to n-1 are significant for the problem and Mij denotes the jth bit of the term Mi
- Fi = logical terms status vector, may be a single bit integer
- Vtj = string variable symbol table for each bit position from 0 to n-1

2. Reduced terms and prime implicants

The end product of the reduction process is a set of reduced expressions called prime implicants. The algorithm does not generate partially reduced expressions in reduction steps since only one sequential sweep of the canonical vector's partitions is used to find the prime implicants.

A reduced term will contain variables less than or equal to the total number of variables in the problem, so that some variables which have been eliminated at arbitrary bit positions will not be found in the corresponding expression. To allow for this, a masking integer is provided together with one of the canonical terms involved in the production of the reduced terms such that the 1 bits in the mask designate the "used" variables and the O's the "unused" or eliminated variables. Though the mask itself is used to determine the existence of a given variable corresponding to its bit position, the assertion levels of the reduced term's variables are specified by the canonical term. We shall designate the given canonical term as the Implicate and the mask as the Impmask. Notice that the Implicate for a given reduced expression is not unique, since it may be any one of the canonical term integers involved in producing the reduced term. Also, any two reduced terms or prime implicants may have the same Impmask. However, any implicant will have a unique Implicate/Impmask pair, or else the resulting implicant is not yet prime.

For example, the reduced term B'C for a problem of 4 state variables using the symbol table set (A,B,C,D) from MSB=3 to 0 may be represented as follows: the Implicate is x01x and the Impmask is 0110, where x represents any bit state, 0 or 1. The Implicate may therefore be one of 0010 (m2), 0011 (m3), 1010 (m10), or 1011 (m11). The Implicates may be generated by permuting the free or x bit positions among the states 0 and 1. Internally, the algorithm will select the first canonical term involved in the prime implicant search as the Implicate and the Impmask is derived from combinations of the Partition byte.

Finally, an Implicant Tag array and Implicant status vector are used for each prime implicant to determine the canonical terms "covered" by each implicant and its status. Necessarily, the Tag array will be two dimensional to allow for the general case of more than one prime implicant in any problem and the elements will contain the subscript location of the prime implicant that covers the canonical term's row. The Implicant status vector is a quad state array to allow for the 4 prime implicant types ESSENTIAL, NECESSARY, REDUNDANT and DON'T CARE. Upon minimization this status vector becomes Bi-state, meaning that the minimal subset chosen by the minimization will be indicated by a 1, while non-selected by a 0. Obviously, the chosen subset will be composed of ALL the ESSENTIALS and SOME or ALL of the NECESSARY prime implicants. The following notations are again introduced to facilitate further discussions:

let q = variable to denote the implicant number
k = grouping size, equal to the number of bits set to 1 in any Impmask
h = tag array pointer variable, from 1 to 2^k

The reduction process yields the following:

Iq = Implicate vector corresponding to an IMq
IMq = Impmask vector corresponding to an Iq
Tqh = Implicant tag array (two dimensional)
STq = Implicant status vector (quad then bi-state)

3. Work Vectors and Functions

The following items involve separate sections for discussion and only the notations are introduced here with summarized descriptions. Again, these are all for the facilitation of succeeding discussions on the algorithm processes.

a. Work Vectors

Pi = partition byte for each Mi, where i is from 1 to m Pij corresponds to the first partition of Mi about the jth variable, j from O to n-1
Si = partition sum, equal to the arithmetic sum of the bits set to 1 in Pi

CVi = cover byte for a given Mi, used to facilitate the reduction process by tagging extracted group combinations from Pi. When a Mi is completely utilized then CVi=Pi.

b. Summing Functions

The general symbol Σ will be used to denote the summing operation in the sense described below, with the limits and variable of summation indicated above and below the symbol.

{..expression..}= arithmetic sum operation
L {...}= logical sum, using bit-by-bit OR
X {...}= mod-2 sum, using bit-by-bit EXOR
$ {...}= catenation sum for string expressions

c. Logical bit-by-bit Functions

or = logical or, for each bit
xor = mod 2 sum, for each bit
and = logical product, for each bit
not = logical complement (ls complement)

d. Searching/Existence Function

let M represent an integer or bit item

E[M,l,m] = 1 or O, representing the existence of the item M within a search space defined by the limits l to m. For bit items, the search space is assumed to be the bit O to n-l.

Eᵬ [bit] = string nulling function, so that E [bit] is a transparent function if [bit] is l, else E [bit] nulls the entire string

e. Combination/Permutation Generating Functions

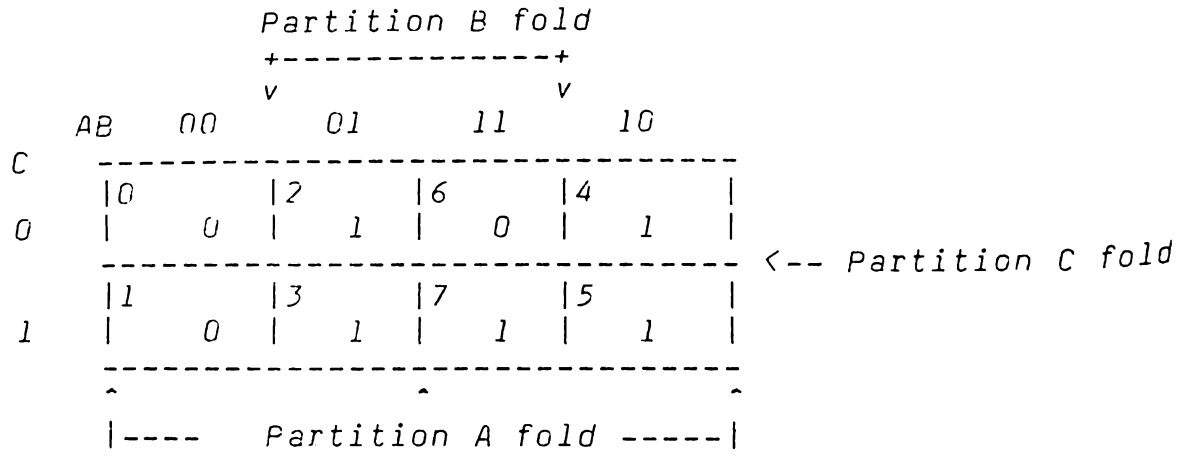
let M represent an integer where N bits are set to l

r = combination size, r less than or equal to N
Cc[M, r] = combinations of the N bits of M set to l taken r at a time
c = combination number, from l to N!/r!(N-r)!
PRp [M] = permutation of the N bits of M set to l
p = permutation number, from O to $2^{(N-l)}$

Variable Boundary Partitioning

The algorithm uses the first partition among all variable boundaries for each minterm to form a partition byte. Each partition corresponds to a bit position in the partition byte for each variable and represents the existence or non-existence of an adjacent minterm along that variable boundary. Hence, the partition byte contains the equivalent information about the grouping possibilities available for any minterm in a problem in the same way as the Karnaugh Map. It is the combinations of the set bits of the partition byte which represent the grouping possibilities and eventually, feasible combinations become the prime implicant Impmasks, where the bits set to 1 represent eliminated variables.

Partitioning is better illustrated than explained, and the problem shown below on a 3-variable K-map is to be partitioned. The equivalent "folding" process of partitioning along the boundaries is illustrated.



Partition A map

P(A)

C \ AB	00	01	11	10
0	0	0	0	0
1	0	1	1	0

Partition B map

P(B)

C \ AB	00	01	11	10
0	0	0	0	0
1	0	1	1	0

Partition C map

P(C)

C \ AB	00	01	11	10
0	0	1	0	1
1	0	1	0	1

Summarized Results: (minterms)

i	minterm	P(A)	P(B)	P(C)	Pi (byte)	Si (row sum)
1	m2 010	0	0	1	001	1
2	m3 011	1	0	1	101	2
3	m4 100	0	0	1	001	1
4	m5 101	0	1	1	011	2
5	m7 111	1	1	0	110	2

Notice that the partitions among the variable boundaries or bit positions have produced some very useful data about the problem. Each partition set to 1 identifies a possible grouping involving the minterm at that position and at the same time indicates the variable that is to be eliminated. For instance, in m2, the P(C) bit is set to 1 so that a group of two is defined by implicate 010 and impmask 110, or translating, this is the reduced term A'B as can be seen from the K-map cited previously. Even larger grouping possibilities can be taken from the partition bytes depending on the number of bits set to 1 as given by Si for the Pi. The following summarize the information available from partitioning:

1. From the row sum data, any minterm with a given Si may group into a size of 2^{Si}, with the variables of the bits set to 1 in Pi being eliminated. Any islands of prime implicants which group by themselves are those with Si = O and these are at once extracted after this step.

2. If we can find the largest Si, call this G, such that there are at least 2^G minterms that have their Si $\geq$ G, then G defines the largest possible grouping in the problem. This is because any grouping must cover a set of minterms which is at least two raised to the number of eliminated variables. If we observed the summarized results in the previous page, we see that the largest row sum is 2. However, there are only 3 minterms in the problem which have their Si ≥ 2, so that there is an insufficient number of terms to support a group of 2^2 or 4. Therefore, the largest possible grouping in the problem is a group of 2^1 or 2. As can be seen in the K-map, the largest grouping in the problem are indeed only duals.

3. Since each bit set to 1 in any Pi defines a dual (i.e., this is the first partition), then any "walk" along the set bits of Pi can be used to define larger groupings. Starting from a given minterm, the corresponding terms involved may be found by XORing the partition bit (in its proper bit position) with the minterm integer in binary. This will be used in the succeeding section on Prime Implicant Extraction.

It can be concluded that variable boundary partitioning has resulted in the availability of information equivalent to that of the K-map, but in a compact, integer form which is the partition byte. In retrospect, this is equivalent to the first cycle in the Quine-McCluskey Algorithm, except that the algorithm uses only the partition bytes to search out all the prime implicants.

In the computer implementation, the folding process is accomplished by generating the set of adjacent terms to a given minterm and searching for these other minterms in the minterm list by a searching technique to be discussed later. The partition byte generation may be expressed as follows:

let Mi = minterm whose partition byte is to be formed
Pi = partition byte of Mi, initially set to O

The set of adjacent minterms may be found for each variable boundary j (j = O to n-1, n = no. of state variables) by:

$$Mi^* = Mi \text{ xor } 2^j$$

A partition exists if Mi* is found in the minterm list by the searching function, which returns a 1 or O, i.e.:

$$Pij = E[Mi^*] = E[Mi \text{ xor } 2^j] = 0 \text{ or } 1$$

The complete partition byte is therefore found by logically ORing all the Pij in their respective bit positions, or:

$$Pi = L \sum_{j=0}^{n-1} \{ E[Mi \text{ xor } 2^j] (2^j) \}$$

Note that Pi is actually generated concurrently for all minterms Mi* as well as Mi, since both will have the same partition bit set. Hence, the complete partition byte is generated rapidly because any verification of a partition will affect the partition bytes of two minterms. Also, problems with more partitions (i.e., those with greater grouping possibilities and therefore more complex) have their partition bytes generated with backtrack or redundant searches avoided and therefore, approximately as fast as for simpler problems. The worst case in partitioning will be if all partitions are zero (a problem involving all islands, or the minterms are the prime implicants themselves), so that all searches fail. However, the algorithm will make up for lost time in the minimization process, since it will see that the problem is one of all ESSENTIAL prime implicants.

Prime Implicant Extraction

1. Prime Implicants from the First Partitions

The walking search for the prime implicants will now be explained and illustrated. Consider the problem below which has been partitioned and with the corresponding row sums. The number of state variables is 4.

i	Mi		Pi		Si
1	0000	(0)	1101	(13)	3
2	0001	(1)	1001	(9)	2
3	0100	(4)	1110	(14)	3
4	0110	(6)	1010	(10)	2
5	1000	(8)	1101	(13)	3
6	1001	(9)	1011	(11)	3
7	1011	(11)	0110	(6)	2
8	1100	(12)	1110	(14)	3
9	1110	(14)	1011	(11)	3
10	1111	(15)	0101	(5)	2

The largest possible grouping for the problem is a group that eliminates 2 variables, or involving 4 minterms. This is because there is an insufficient number of terms with Si=3 to hold an octet or 8 minterm grouping.

The prime implicants will be extracted from a certain starting point minterm, which will be the "walk" search basis. Any large grouping should contain terms that have their partition bits set to 1 in the variables eliminated and they should also be compatible by the XOR at the variable partitions. The general procedure is a straight forward single scan over the minterm list, and appropriate tags are created to enable the extracted implicants to be excluded in further searches.

The prime implicants are extracted from a given minterm starting point by the following procedure:

a. Consider the grouping size to be the largest possible grouping. This ensures prime implicants are extracted since the size searched is the largest for the problem, and any failure

will result in a smaller size, which is still prime. Call the minterm starting point Mi and its partition byte Pi. Let the group size to be searched be denoted by G, where 2^G minterms are covered. If the Si of the minterm is less than G, then set G=Si.

b. Generate a combination of Pi such that there are G bits set to one in the combination. If Si=G then there is only one combination, or else there are Si!/G!(Si-G)! combinations. If all combinations have been generated, then go to i.

c. Generate the permutations of the combination in b. This means to permute the bits set to 1 in the generated combinations, which is similar to counting in binary except at different bit positions as determined by the set bits in the combination. Each permutation generated and XORed to Mi will create the list of minterms involved in the grouping.

d. Search for all these terms in the minterm list. If any one term does not exist, then the grouping is infeasible, so generate another combination (go to b).

e. If a term is found in d, then its Pi must have the same boundaries as the bit positions in the combination. This is done by taking the AND of the combination with the term's Pi. If the result is still the combination, then they are combinable. Otherwise, the grouping is again infeasible so go to b.

f. Repeat c and d until all the terms in the permutation are found.

g. The prime implicant is found and specified by its implicate = Mi and its impmask = the NOT of the combination byte.

h. Repeat b until all the combinations are exhausted. Go to b for another combination.

i. If no implicant has been extracted with the size G, then reduce G by 1 and go to b. Else, all the prime implicants involving the chosen search basis Mi of size G have been found and a new search basis is chosen. Note that it is not necessarily true that all the prime implicants involving Mi have been found, but a successive change of the search basis over all the minterms of the problem assures that the complete prime implicant list has been generated.

For the problem in the previous page, let us choose mO (0000) for the first search basis. Its Pi is 13 which has 3 bits set to 1, meaning it can group three ways as a dual. Since the largest possible group size is a group of 4, we take the combinations of the 3 bits set in 13 or 1101 two at a time. These combinations are C1=1100, C2=1001 and C3=0101 since there are 3!/2!(3-2)! = 3 combinations. We will work these out separately.

Choosing C1 = 1100, the permutations of the set bits are PRI = 0000, PR2 = 0100, PR3 = 1000 and PR4 =1100. The corresponding minterms to be searched for to make the group feasible are the XOR of each PR to mO or 0000 (mO), 0100(m4), 1000(mS) and 1100(m12).

The terms m0, m4, m8 and m12 are found to exist, as can be scanned from the minterm list. It now remains to be checked if the Pi of each ANDed with the combination C1 yields C1. For m4, m8 and m12, the corresponding Pi are 1110 (14), 1101 (13) and 1110 (14) for which the AND of C1 = 1100 all result in C1. Hence a prime implicant is defined by implicate =0000 snf impmask = 0011 (NOT C1) and translating, the expression is C'D'.

For C2=1001, following the same procedure, the required minterms are m0, m1, m8 and m9 and each one exists and has its Pi AND C2 = C2, so that the implicant implicate=0000, impmask=0110 or the expression B'C' has been extracted. For C3=0101, the required terms are m0,m1,m4 and m5 but since m5 does not exist, the grouping is infeasible. This now terminates the use of m0 as the search basis and another minterm is now chosen.

If no combination yields a feasible group, then the size to search for is reduced by 1, so that the prime implicants may be extracted using the largest grouping available for the minterm. For instance, if minterm m15 is chosen as the search basis, its Pi=0101 (5) which has two bits set to 1, and since the largest possible grouping eliminates 2 variables, the only combination of Pi with 2 bits set is Pi itself or C1=0101. Now the terms required are the XOR of m15=1111 with the permutations of C1, which yield the minterms 1111 (m15), 1110 (m14), 1011 (m11) and 1010 (m10). However, m10=1010 does not exist and since C1 is the only combination, then there will be no prime implicant extracted which eliminates 2 variables for m15. The group size is reduced to 1, so that the combinations to satisfy are now C1=0100 and C2=0001 which both yield feasible groupings. Hence, the prime implicants extracted from m15 are 2 duals which yield implicate=1111, impmask=1011 or ACD and implicate=1111, impmask = 1110 or ABC.

It has been shown that a simulated walk as generated by the permutations of a given grouping combination may be done among the bits set to 1 of the partition byte of a minterm. The combinations generated are dependent on the largest possible grouping or the largest grouping available for the minterm. The algorithm actually tests for the worst case of the last permutation to test if the minterm Cc XOR Mi exists, and hence the search test is maximized. Also, each combination is actually compared to a tag called the minterm cover CVi to eliminate combinations which have already been found for the given search basis minterm and since the covers approach Pi as the prime implicants are extracted, the prime implicant extraction is actually accelerating in speed as the search basis is changed. The tagging process and the determination of the implicant type is now to be discussed.

2. Prime Implicant Type Determination and Minterm Tagging

Any prime implicant extracted as described in the previous section involves a starting minterm as a search basis and involves or "cover" a set of minterms that have been searched to make the grouping possible.

However, when the search basis is changed to be one of the minterms involved in the original search basis' prime implicants, there is the possibility of a redundant search resulting in a prime implicant that has been extracted previously. Such is the case, for example, in the Quine McCluskey algorithm where any set of combinable terms are always reduced by eliminating one variable, although the end result will be the same prime implicant.

The algorithm uses a tagging array to eliminate this possibility. Note that any prime implicant extracted will simply mean that a combination of the set bits of Pi has been used for all the minterms involved in the prime implicant. Hence, if a cover is used such that if it is involved in a verified prime implicant, then the combination used to give that implicant is ORed to its cover, then we have effectively tagged the minterms so that if it becomes the search basis, any combination Cc is removed from further consideration since this would have resulted in a prime implicant that has already been extracted.

Necessarily, the cover array or cover vector will have the same length as the minterm list vector so that this again represents a memory overhead for the algorithm computer implementation. If a minterm has been totally used, i.e., all its partitions have been worked out for prime implicants, then its cover will be equal to the partition byte. Note that the cover for any minterm is also a n-bit integer, where n=no. of state variables, since we need to know the partitions of the minterm that have been used already.

In the previous examples then, any time a prime implicant is extracted (i.e., verified and all minterms searched), all the minterm covers are ORed with the corresponding combination. For instance, in using minterm m0 as the search basis, for which the combinations 1100 and 1001 have been used, all other minterms involved must have their covers ORed with 1100 and 1001. Minterm m1 for instance, which has Pi=1001 has its cover as 1001 also, so that m1 is not used as a further search basis since all its partitions have been used for the prime implicant involved with combination 1001 when m0 was the search basis. This implies that the

algorithm reduction speed is actually accelerating as the prime implicants result from previous search basis minterms.

It now remains to determine the type of implicant that was extracted, and this is a feature that is never available in the Quine-McCluskey algorithm as the reduction progresses. The algorithm is able to do this using the partition bytes and the Function vector or status array. Recall that the latter is simply an integer that is either 0 or 1, 1 being for canonical terms (minterms or maxterms) and 0 for don't care terms.

A prime implicant may be either of four types: ESSENTIAL, NECESSARY, REDUNDANT or DON'T CARE.

The easiest of all to determine will be a DON'T CARE prime implicant, since this is simply a prime implicant which involves all don't cares. By virtue of the fact that the Function array is 1 for minterms and 0 for don't cares, if a counter is used to sum the Function integers of all the minterms involved in a prime implicant, then if the counter yields 0, this indicates that all don't cares have been used and hence a DON'T CARE implicant is extracted. The counter, however, may also be used to determine other prime implicant types. Note that in any case that the counter is greater than one, then the prime implicant is never a DON'T CARE prime implicant. This counter is cleared once a new combination is being used.

The next easier type to determine will be an ESSENTIAL prime implicant. This prime implicant is defined when at least one minterm groups in only one way on a K-map. In terms of the partition type, this is translated to mean that at least one minterm in a prime implicant group combination has all its partitions used or that one minterm has Cc=Pi, where Pi is the partition byte of the minterm. Note that this has to be a minterm, i.e., its function integer must be equal to 1, since we do not desire an "essential" don't care, or in other words, a don't care term must not decide if the grouping is essential.

The most complicated to determine is a NECESSARY prime implicant. Such a prime implicant involves at least one minterm which can group to eliminate say L variables, but can only group in a prime implicant to eliminate L-1 variables with the additional restriction that the minterm mentioned must not group in any ESSENTIAL grouping. Other than this, the prime implicant is a REDUNDANT term. The difficulty is that all the ESSENTIALS are extracted at the same time as the possible NECESSARY prime implicants, but the final status is determined only after all the ESSENTIALS have been extracted. A REDUNDANT prime implicant is at once generated if not one minterm in a prime implicant cover satisfies the L-1 criteria, i.e., all the terms group in less than the number of possible ways by more than 1. Hence, if no minterm groups in L-1 ways exactly (this must of course be a minterm and not a don't care), then a REDUNDANT is at once determined.

For the possible NECESSARY implicants that have been found, a "clean-up" is therefore performed after all the prime implicants of the problem have been extracted. This means that all the possible NECESSARY implicants or those which group with at least one minterm which has L ways of grouping, but with the prime implicant grouping using only L-1, are then tested to see if those minterms are now covered by some ESSENTIAL groupings. The backtrack is not really time wasteful, since if we recall, only the NECESSARY prime implicants are used for minimization and the final minimal cover subset will be composed of ALL the ESSENTIALS plus the NECESSARY implicants chosen by the minimization algorithm. Hence, the "clean-up" process allows the final build-up of the minimization constraint equations and only a few prime implicants are involved in the final steps.

3. Constraint Equations for Minimization

The constraint equations for minimization are generated once the prime implicant cover, i.e., the minterms involved in the prime implicant, are found. This criterion in minimization may be translated into the following: if we represent each prime implicant as a "variable" in minimization, with each prime implicant assigned a cost coefficient, then the

constraint of the problem is that any set of prime implicants chosen as the minimal subset must have the constraint equation greater than or equal to one. The prime implicant as a "variable" in the minimization process is called a zero-one variable, since we either choose a prime implicant depending on whether it minimizes the total cost for implementation or else satisfies a constraint.

The number of constraint equations are therefore equal to the number of minterms of the problem (don't cares are never included). The constraint equation may be expressed as the sum of the prime implicants as "variables" which cover a given minterm, and the total sum must be $\geq$ than 1. These constraint equations are kept in a two-dimensional array, where the rows are defined by the minterms and the columns contain the coordinates of the prime implicants in their implicate-impmask vectors which cover the minterm. The algorithm uses this to also determine if the possible NECESSARY prime implicants are now REDUNDANTs during the clean-up process. Note that this is again usually the largest array used in the problem, although it may be eliminated at the expense of processing time for backtracking to find the minterm covers.

Of the constraint equations, any minterm that has been covered by an ESSENTIAL is at once satisfied, since any minimal cover subset of prime implicants will include the ESSENTIAL prime implicants. Hence, the only active constraint equations are those which still involve NECESSARY prime implicants, i.e., all the prime implicant coordinates in the columns for a given minterm point to NECESSARY prime implicants only.

4. Summary of Prime Implicant Extraction in Equation Form

The procedures described in this section may be summarized using the notations previously described. Letting G be the largest possible grouping in the problem, the following equations describe the prime implicant extraction processes:

let Mi = search basis with Pi as its partition byte

The combinations to be generated for a prime implicant search are:

Cc [Pi,G] where Si bits are set to 1 in Pi so that there are Si!/G!(Si–G)! combinations and c is the combination number from 1 to this value.

The permutations of this combination are:

PRp [Cc[Pi,G]] where p is the permutation number 0 to 2^G

The set of minterms to search start from p = 1 to 2^G since p = O yields the search basis minterm Mi. This set is called Mi* and is determined by XORing the permutation with Mi or:

$$Mi^* = Mi \text{ XOR } PRp[Cc[Pi,G]] \text{ for } p=1 \text{ to } 2^G$$

The condition for the feasibility of the group is that all the minterms Mi* exist and that their partitions Pi* ANDed with Cc [pi,G] results in Cc also. Let E denote the searching function. The total number of compatible terms must therefore be 2^{G-1}, the minus one to eliminate the search basis Mi which is of course compatible. Hence, the existence of a prime implicant is:

$$E\{\text{prime}\} = E\{\sum^{2^G} \{E[Mi^*] \text{ and } p=0 \; ((Pi^* \text{ and } Cc[Pi,G]) = Cc[Pi,G])\} = 2^G\}$$

where E {prime} is either O or 1 to indicate non-existence or existence respectively. We will use i* to enumerate the minterms involved.

If all the combinations tried do not yield a prime implicant, then G is set to G-1 until at least 1 prime implicant has been extracted. A general combination to be tried should actually exclude those that have been used, to prevent redundant prime implicant searches. Let this general combination be Cc'. Then Cc' = Cc and E[(Cc and Cvi){ }Cc], so the total number of combinations to try for any search basis Mi is decreased depending on the CVi of that

minterm. This Cc' is what is actually used in the algorithm and the symbol Cc will be used to mean the same thing.

The prime implicant, call this the qth extracted implicant, is expressed by its implicate=Mi and its impmask=NOT Cc, where Cc is the combination of Mi used to find the prime implicant. This may be expressed as:

$$Iq = Mi \text{ (implicate vector)}$$

$$IMq = \text{NOT } (Cc[Pi,G])$$

The implicant tags are simply the set of prime implicants that satisfy or cover a given minterm. Given a prime implicant that has been extracted, the tag is:

Tqh=q, where q = prime implicant number or coordinate and
h = i* the set of minterms covered

The implicant type is determined by the total number of minterms involved in the grouping (i.e., don't cares are assumed to exist in the problem) and the difference between its Pi* and the combination Cc. If we let r=sum of the Fi logical status vectors of the terms of the prime implicant, then:

$$r = \sum_{i^*=1}^{2^G} Fi^*$$

where i* is a number to denote the set of minterms of the prime implicant

A prime implicant is a DON'T CARE if r=O (no minterm)

A prime implicant is ESSENTIAL if there is at least one minterm which has its Pi* = Cc, or

$$\sum_{i^*=1}^{2^G} E[PI^* = CC] \geq 1$$

A possible NECESSARY is defined if there is at least one minterm which has Si*G−1 exactly or:

$$\sum_{i^*=1}^{2^G} E[\,Si^* = G-1] \geq 1$$

A REDUNDANT is at once defined if all the four tests fail or the possible NECESSARY defined above for a given i* is covered by an ESSENTIAL prime implicant in the clean-up process.

Hence, the implicant status vector STq will have 4 values to denote each prime implicant type and then, after minimization, it will be two-valued – conveniently, the algorithm uses 1 to denote chosen implicants and 0 to denote discarded implicants. Necessarily, all ESSENTIALS at once have their STq = 1 and the DON'T CARES and REDUNDANTS have their STq = O. The remaining NECESSARY prime implicants are assigned an STq = 1 to denote that they are free "variables" for minimization and their STq will be converted to 0 or 1 after the selection of the minimal cover subset.

Combination and Permutation Generation

There are two areas other than the problem complexity itself, which will determine the speed of the reduction process of the algorithm. These are the generation of combinations and permutations, and the searching algorithm. The latter is described in the next section.

The problem presented in the generation of combinations of a given Pi, taken G at a time becomes complicated by the fact that these involve only the bits of Pi set to 1. Moreover, the permutations of this combination also involve permutations of the set bits only. Consider, for instance, Pi= 11011 and G = 2. The combinations that need to be generated are 10001, 10010, 11000, 01001, 01010, and 00011. If we translate these to their decimal equivalents, Pi = 27 and the combinations are the numbers 17, 18, 24, 9, 10 and 3. As predicted, since there are 4 bits in Pi set to 1, there will be 4!/2!(4-2)! = 6 combinations of Pi taken two at a time.

However, if the reader takes the effort to research on publications involving combinatorial analysis, the only information usually available is precisely the number of combinations of N items taken say R at a time, but NOT how the combinations themselves are generated. Although the combination "trees" may be done manually to determine these, there are no published computer algorithm for generating these combinations.

For instance, take four items and let them be called item 1, 2, 3 and 4. The combinations taken three at a time are 123, 124 and 234. The combinations taken two at a time are 12, 13, 14, 23, 24 and 34. This sequence follows no particular mathematical formula although their total number is always N!/R!(N-R)! where N items are taken R at a time.

In a computer algorithm, we may keep the items in an N length vector or array and to take the combinations, we will need a nested set of loops that is R deep, wherein the loop counters represent the pointers to the variables and the limits are receeding by N-R-loop number. For instance, four items taken three at a time will have the following implementation:

```
FOR I = 1 TO 4-3 +1
    FOR J = I TO 4-3-2
        FOR K-J TO 4-3-3
            PRINT "COMBINATION="; I;J;K
        NEXT K
    NEXT J
NEXT I
```

Notice that the loop limits are N-R-loop number, where we count the loops from the outermost to the innermost until N-R-loop number=N. The main problem here is that R is desired to be variable in the algorithm, and since this requires a new set of loops, the versatility of the above is restrictive. Besides, N by itself varies from the problem specification, since N = number of state variables in the problem.

One possible solution is to use a programming language that allows a recursive function call. This way, a procedure may be defined that is recursively called with the loop limits changing as stated above. However, this will allow a limited flexibility to computer implementation and instead another solution is used which uses a walking pointer.

The walking pointer is another array whose vector length is equal to N. The combinations are taken by incrementing and decrementing both the subscript pointer to this array and its contents, in a manner to allow a unique set of variable pointers to exist in the first R elements of this pointer array.

To clarify, let us call this array the VPTR array, and the subscript to be SPTR. Is always the first R elements of the VPTR array that define a combination, and SPTR is used only as a secondary pointer to increment or decrement the contents of VPTR at specific locations to make each of the first R elements to represent a unique combination.

For example, let us take four items which are called APPLE, ORANGE, LEMON and GRAPE. Let these be numbered as the 1st, 2nd, 3rd and 4th item respectively, with their names as strings inside a SYMBOL array. If we desire the combinations of these items taken three at a time, we will need the set of number sequences 123, 124, 134 and 234 to represent the different item combinations and the display is simply the corresponding symbols for each number. For instance, 123 means SYMBOL (1); SYMBOL (2) and SYMBOL (3) or APPLE; ORANGE; LEMON is the combination corresponding to 123.

The following program statements generate the combinations of any general N things taken R at a time by the walking pointer technique:

```
5    rem ' minimum array subscript is zero & VPTR(0) =0
10   SPTR = 1
15   rem '  VPTR walk by one from previous
20   VPTR (SPTR) = VPTR (SPTR-1) + 1
30   IF VPTR (SPTR) < = N– R + SPTR THEN 40 ELSE 120
35   rem ' SPTR walk forward
40   If SPTR < R THEN SPTR =SPTR+1 : GOTO 20
50   rem ' a combination has been found
60   FOR I = 1 to R
70   PRINT SYMBOL (VPTR(I)),;
80   NEXT I
90   rem ' stay at current position if VPTR(SPTR) < = N-R + SPTR
100  VPTR (SPTR) = VPTR(SPTR) +1 : GOTO 30
110  rem ' SPTR walk back to the previous SPTR
120  IF SPTR = 1 THEN END rem ' no more walk – done
130  SPTR =SPTR -1: VPTR (SPTR) = VPTR (SPTR) + 1:GOTO 30
```

The above statements represent the incrementing of the VPTR elements until each element is less than or equal to N-R+SPTR. This may be seen in the aforementioned example if we recall the set of combinations of VPTR as 123, 124 and 234. Note that if we regard each number as a location in VPTR, counting one to three from left to right, position one has a maximum value of 2, the second as 3 and the third as 4. This is true even for the case of the four items taken two at a time, with position one having a maximum value of $4 - 2 + 1 = 3$ and the second as $4-2 + 2 = 4$, since we need the numbers, 12, 13, 14, 23, 24 and 34. The SPTR pointer then takes care of incrementing each VPTR location by one until the $N - R + SPTR$ value is exceeded or until the last location (the Rth element) has been reached, whereupon SPTR decrements by 1 and increments the new location until the last again. The combinations are therefore generated sequentially and in a simple yet efficient manner.

For use in generating the combinations of the set bits of Pi, the N items are simply the bit positions where Pi=1 and the total combination may be taken as the sum of two raised to the bit positions of those chosen by the combination generator.

For example, a Pi=1011 means that the items are 1000 (8), 0010 (2) and 0001 (1). The combinations of these taken two at a time are 8 & 2; 8 & 1; and 2 & 1 corresponding to C1 = 1010, C2 = 1001 and C3 = 0011.

Permutations are even easier to generate using the representation of the items by means of the pointer array, since this will mean the continuous decrementing by 1 will always assure the change of at least 1 bit from the last permutation. For instance, let C1=1011 – then the items to permute are again 1000 (8), 0010 (2) and 0001 (1). To permute these, consider the VPTR array as an integer where a 0 means we omit the item, and a 1 means that we use it. The "byte" corresponding to choosing all three items is 111, which is the first permutation. Then next is this value minues 1, if we consider 111 to be a binary number that results in 110. This may be continued to 101, 100,011, 010, 001 and 000. Now, if we translate each bit in this permutation byte to the actual items we are permuting, then 110 will represent 8 and 2, or summing them, 1000 + 0010 = 1010 and continuing we get the sequence of numbers 1000, 0011, 0010, 0001 and 0000. All of these are permutations of the set bits of C1 = 1011 since the OR of all of them result in C1.

Searching Algorithm

Extensive searching is used in the algorithm in two main parts: in the generation of the partition bytes, and in the extraction of the prime implicants. The importance of the algorithm used in searching is due to its direct relationship with the reduction speed or the speed by which the prime implicant list is generated. All minterms are located in physical

locations that have no direct relationship to their logical truth-table positions, so that there is no easy way to verify the existence of a minterm except by a separate searching routine.

The search algorithm makes use of the fact that the minterms are sorted in ascending order prior to any reduction, so that the "alphabetized" list may be used to remove the randomness of the arrangement of terms.

The most attractive search method to use will be the binary searching algorithm, wherein we continuously divide the length of the search space by two, until the term is found. The search space is initialized to be the entire length of the alphabetized list or in this case, the total number of minterms in the minterm vector. Let this length be designated by NTERM. The binary search algorithm tells us to start with the term of location NTERM/2 and the branch condition (whether we search the upper or lower half), will determine whether we take the upper or lower half to again divide by two and to test for a match with the search item.

The algorithm uses what may be called a Scaled Binary Search algorithm, owing to the unique relationship of the minterms and their combinable terms.

The main difference from the conventional binary search algorithm is that the initial value of the search space is not set to NTERM, but instead, the actual difference between the minterm value of the search basis term and the term to be searched. This is because there is always a logical relationship between the physical and logical or row number values of the minterms.

Consider for instance that NTERM = 100 and we are staying at minterm m0. If, for example, we are searching for m8, for the purpose of verifying a prime implicant or a partition bit, then the initial search space chosen by the binary search algorithm would be 100. However, the scaled search would consider the search space from m0's physical location to m8–m0=8. Hence, the search space is chosen to be 8 items downward from the physical location of m0. This is a valid method for assigning the initial search space because even in the case when all the minterms m0 to m8 are present in the problem, then the maximum physical distance between the two minterms will be equal to 8. Therefore, there is now a bound set to the search space because we are sure that m8 cannot be farther than 8 physical locations away from the location of m0. In the same way, say we are at m4 and we are looking for the minterms m6, m12 and m14 to verify the prime implicant BD', then the search spaces will all be downward from m4 (since the minterm list is sorted in ascending order) and the distance will be 2 for m6, 8 for m12 and 10 for m14 – all measured downward from m4.

A quantitative comparison may be made by considering worst case searches. The straight-forward binary search will always start from a search space equal to NTERM. Let the worst case search proceed by successive division of the search space until the search space is reduced to 1, or we do K searches until the condition $NTERM/2^K$ is equal to 1. The number of searches is given by:

$$K = \log NTERM \text{ where the log is to the base 2}$$

The scaled binary search will use and initial search space that is the absolute value of the location of a minterm Mi minus the value of the minterm value to search, call this Mi*. The worst case number of searches is therefore

$$K' = \log ABS(Mi\text{-}Mi^*) + (Mi\text{-}Mi^*)$$

where ABS is the absolute function

Since ABS (Mi-Mi*) is less than or equal to NTERM in any search, then $K' < K$ always. Also, we can at once determine if the minterm Mi* cannot exist, if the distance LOC (Mi) ± ABS (Mi* – Mi) exceeds the minterm list length (i.e., it goes below zero or exceeds NTERM).

Minimization Process

The minimization process involves the representation of the problem as a zero-one integer programming model composed of an objective cost function, which is to be minimized, and a set of constraint equations developed from the prime implicant covers. The algorithm uses two major calculations: Constraint Evaluation and Cost Calculation to determine the next bound rule in the Branch and Bound routine in the Balas Algorithm.

1. Structure and Evaluation of Constraint Equations

The prime implicants are treated as the "variables" of minimization and their coefficients are all one in a constraint equation. A constraint equation is formed as the algebraic sum of these "variables" and an equation is specified for each minterm in the problem. The condition for any equation is that the algebraic sum must be greater than or equal to one, with each "variable" taking only values of zero or one. This may be expressed as:

let X1, X2, X3 represent the prime implicants where any Xi=0 or 1 only

sample constraint equations:

$$X1 + X2 \geq 1$$
$$X2 + X3 \geq 1$$
$$X1 + X3 \geq 1$$

Using the tag array Tqh generated in the reduction step, where q points to the prime implicant, h to the minterm for which the constraint equation is written, the final values of the status array STq of the prime implicants are either zero or one corresponding to choosing and not choosing the prime implicant respectively. Hence, the constraint equations for the problem may be expressed as:

$$\sum_{h=1}^{m} STq(Tqh) \geq 1$$

where h always involves minterms and not don't cares
Tqh = tag array pointer
STq = 0 or 1 only

STq is originally quad-state with values of 1 for ESSENTIAL, 0 for REDUNDANT, -1 for NECESSARY and -2 for DON'T CARE prime implicants. The DON'T CAREs and REDUNDANTs are set to 0 and the ESSENTIALs left with a 1 value, since the minimal cover subset never includes DON'T CAREs and REDUNDANTs and always include the ESSENTIALs. Hence, any STq with a value of -1 is interpreted as a "free variable" in minimization, and the final values are affixed only after minimization.

Because any trial set of solutions of the problem must satisfy the constraint equations first, the dominant factor in the minimization processing time is the evaluation of the constraints. The number of constraints depends on the number of minterms that are covered by NECESSARY prime implicants, so that the speed of the minimization is directly related to problem complexity.

2. Cost Coefficients and Cost Objective Calculation

Each prime implicant is assigned a cost coefficient equal to the number of state variables it uses and the number of inverters it requires. This assures that the minimal subset chosen is first determined by the prime implicants that cover more implicants, or we are choosing the largest grouping available in the problem, similar to the use of the larger geometrical groups on the K-map. The addition of inverter cost then dominates the case of a multiple solution problem that involves prime implicants which can be used in multiple ways that give the same cost if only the FAN-IN were used for the cost coefficients. The cost coefficients are kept in a Weight array Wq, for each NECESSARY prime implicant. As an example, assuming the terms A', BCD, B'CD' and AC' are prime implicants, the cost coefficients are then 2, 3, 5 and 3 respectively.

The objective function is simply the sum of the cost coefficients of the variables multiplied by the variable value, which is either 0 or 1 only (n.b. variable is used here in the sense of a minimization variable, i.e., a prime implicant). This may be expressed as:

$$\text{Cost} = \sum_{q=1}^{Q} Wq * STq$$

where Q = total number of prime implicants and any q is a NECESSARY implicant. STq is either 0 or 1 only.

Any cost calculation is stored in a variable called ZNEW and ZBAR is used to store the cost of the incumbent solution as of the last minimization iteration. Initially, ZBAR is set to

the machine infinity, which is some large value such as 1,000,000,000.

An additional step needed prior to the minimization proper is the sorting of the prime implicants involved in the order of increasing cost coefficients. This is to facilitate the Bound step in the Branch and Bound algorithm which is to be described.

3. Branch and Bound Minimization Algorithm

The original algorithm for solving a zero-one integer programming problem (i.e., a problem involving variables constrained to take only 0 or 1 values, and all integer coefficients in cost and constraint equations) was developed by Egon Balas and presented in 1965 in the Operations Research Journal [2]. It is called an additive algorithm because it uses only add and subtract operations.

The version used here is the modified version by Hillier and Lieberman [4]. The basic steps are the Branch step and the Bound step. Since the variables of minimization of prime implicants are sorted in increasing cost, the Branch step always chooses the next variable only if the Bound step determines that there is a solution emanating from selecting that variable. This means that zeroes are assigned to be one only when it will result in a lower cost and satisfy further the violated constraints. The termination condition is when we find that we have returned to the first variable being set to zero, and finding non further solutions since all the variables are now zero and there are no free variables left.

The branch step is specified by three discounting tests called Fathoming tests. If upon any partial solution (i.e., there are still violated constraints), a Fathoming step is satisfied, then further consideration of completing the partial solution is scrapped and a new partial solution is considered. This is also termed as choosing the newest bound rule, since we Bound first before branching. The three Fathoming steps are as follows:

a. Fathom Test 1 – No lower cost in completion

This assumes that the current partial solution is already a complete solution, even though the constraint equations have not yet been evaluated. Since the variables are ordered in increasing cost, then if the remaining variables (i.e., those not yet used in the current partial solution) are set to zero, and the cost ZNEW is calculated and we get ZNEW $\geq$ ZBAR, where ZBAR is the lowest cost so far, then we scrap the partial solution and move to another, since any completion of the partial solution (if ever it exists) cannot have a lower cost than ZBAR.

b. Fathom Test 2 – No feasible solutions

This concerns only the satisfaction of the constraint equations. With the current partial solution, composed of some variables set to 1 and others to 0, the remaining variables are set to 1 temporarily and the constraint equations are evaluated. If any constraint is still not satisfied, then further branching using this partial solution will never give us a solution, so again, the partial solution is scrapped.

c. Fathom Test 3 – Best feasible solution found

This is the case when a solution has been reached, meaning that all constraints are satisfied and the current cost ZNEW is lower than the lowest cost ZBAR of the problem. The solution is formed by the values of the variables of the partial solution, and the next variable set to 1 less the value of the current variable, with all the rest that has not yet been considered set to zero.

The Bound step Fathoming tests may be expressed in equation form if we use the following notations:

Xi = the ith variable, from Xl to the last variable Xq

Ci = the cost coefficients in increasing order from 1 to q

Aij = the constraint equation coefficients, either 1 or 0 in value for the jth minterm, using the ith prime implicant

The cost calculation is $ZNEW = \sum_{i=1}^{q} CiXi$

The constraint evaluation seeks to satisfy all constraints, or that any minterm must be covered at least once. This may be expressed in terms of the existence function E as:

$$E\left[\sum_{i=1}^{q} Aij * Xi \geq 1, \text{ for all } j\right] = 1 = E[-\text{constraint}-]$$

The current partial solution is the set of Xi from 1 to h, with the maximum of h equal to q.

The fathoming tests are:

a. ZNEW $\geq$ ZBAR, where ZNEW is evaluated for the Xi from 1 to h and the h+1th variable if Xh=0

b. E [-constraint-]=0 for some constraint equation j, using the variables Xi from 1 to h only

c. ZNEW< ZBAR and all E[-constraint-]=1 for all the variables Xi from 1 to q, with the Xi from 1 to h set to their values, h + 1th to 1 – Xh and the rest set to 0.

The branch step always chooses Xh + 1 = 1, for the new partial solution, and the partitions of the solution tree are considered from the new h set equal to h + 1. This is again because of the arrangement of the costs in increasing order. The following steps summarize the attack procedure for minimization using the above concepts:

a. Initialize h = O and ZBAR = infinity or some large value.

b. Perform the fathoming tests. If Fathoming step 3 (the best feasible solution is found), then store the current values as the incumbent solution and reset ZBAR=ZNEW. Go to d.

c. If the partial solution has not been fathomed, then set Xh + 1 = 1 and set h = h + 1. Go to b. If ever b is not satisfied for h = q then there is no solution to the problem – this will never occur in our case.

d. If all the Xi's are now set to zero, then the solution terminates and the minimal cover subset has been found. Otherwise, proceed to e.

e. Backtrack from h until an Xi is found that is set equal to 1. Let its location be called g. Then reset this Xg=O and set h=g. Go to b. This step is because we are considering the partial solutions emanating from any Xi set to 1 only, since we will only result in Fathomed partial solutions if we consider otherwise.

The Xis used here may be translated in the problem to represent the STq of each NECESSARY prime implicant of the problem.

The Branch and Bound Technique prunes the possible solution combinations in exponential speed, because any non-feasible solutions are quickly fathomed out from further consideration. Also, all operations involved in constraint evaluation or cost calculation are merely additions, so that the computing speed is faster compared to other algorithms such as the Simplex method, which uses multiplication and division. Also, there are no additional constraint equations introduced and hence no growing memory requirements involved, unlike other minimization techniques which introduce slack variables.

System Flowchart

To summarize the processes of the algorithm, the following flow chart is presented which is in a sequential form:

1. Data entry for the Number of State Variables, State Terms and their status, either don't care or minterm/maxterm.
2. Sorting of the terms in Ascending order.
3. Partition Byte-Array Generation and determination of the Largest Possible Grouping (extraction of island implicants).
4. Reduction or Generation of the prime implicant list which includes:

a. Combination and permutation generation, and a Scaled Binary Search Technique to to extract implicants
b. Generation of Constraint equations
c. Determination of Implicant Types
d. Clean up of Redundants from the possible NECESSARY prime implicants

5. If this is an all ESSENTIAL problem, then no minimization is necessary, so go to step 7.

6. Minimization to generate minimal cover subset, which includes:

a. Calculation of cost coefficients for the prime implicants (only NECESSARY prime implicants are used)
b. Sorting of the prime implicants in increasing cost
c. Minimization Proper, using the Branch and Bound technique

7. Output Routine for Translating the internal data representation of the prime implicants into Boolean Expressions. This may be expressed in equation form, using the prime implicate and impmask as:

$$\text{TERM} = \text{⊬} \sum_{j=n-1}^{0} \{ E\,[\,(Iq \text{ and } IMq) \text{ xor } 2^j\,] * [V\text{⊬}\,j\,] + E\text{⊬}\,[\,Iq \text{ and } 2^j\,] * \text{" ' "} \}$$

where E [–] is the string nulling function
" ' " is used to indicate non-asserted
V⊬ J is the variable symbol table
Iq, IMq are the prime implicant implicate and impmasks respectively (n-bit integers)
SUM is a string catenation sum

Conclusion and Discussion

An algorithm for logic function reduction and minimization using a microcomputer system has been presented. This algorithm offers significant reduction in processing time and in memory requirements through the use of a walking search routine for prime implicants and of zero-one integer programming algorithm combined with the elimination of unnecessary prime implicants, compact data representation and the labelling of the status of prime implicants extracted to eliminate unnecessary implicants in the minimization process. As currently implemented, the program finds the solution of a four-variable problem in 30 seconds and a ten-variable problem in 15 minutes. This is definitely much faster than manual solution which is the only other alternative available to us. Because of microcomputer system limitations the maximum number of variables that can be tackled is 13.

The algorithm, as described here, has been developed for a single output function only. This has been done to avoid the use of disk file access techniques and hence using only memory stored variables throughout the reduction and minimization processes. However, an expansion to allow larger problem sizes may be done by using the file access techniques also available in MBASICs interpreter, so that memory area may be used solely for the programs per se. This will also permit the application of the algorithm for multiple output problems by the following steps:

1. Store the prime implicant lists (implicates, impmasks and status) for each output in the disk file.

2. Store the constraint equations for each output also in the disk file.

3. After all the outputs have been used for 1 and 2, add to them the minterms themselves as island prime implicants, and include them in the constraint equations appropriately.

4. Consider the adjoined minterm list in minimization, i.e., the unique minterms for each output and the joint or shared minterms for all outputs.
5. The cost objective function is now modified as the summation of all the cost objective equations for each output.
6. The minimization algorithm now includes as constraint equations, that each individual function must have their minterms completely covered at least once by a prime implicant.

This way, the new cost equation is now a system cost and not a single output equation. Necessarily, the use of memory as storage for the temporary results of each output will limit the problem size capability, but since floppy-disk based microcomputers are also common, the expansion for multiple output problems is also feasible, with modifications only in the minimization portion and using the reduction process successively for each output.

Implementation of the algorithm in a mainframe computer should improve the speed for solution and will allow for larger problems to be solved. This will also allow comparison of this algorithm with other algorithms in terms of processing time and memory space requirements.

References

1. Fletcher, William. *An Engineering Approach To Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1980.
2. Balas, Egon. "An Additive Algorithm for Solving Linear Programs With Zero-One Variables," *Operations Research,* vol. 13, pp. 517-546, 1965..
3. Riordan, John. *An Introduction To Combinatorial Analysis.* John Wiley and Sons, New York, N.Y., 1958, 4th printing 1967.
4. Hillier, Frederick and Lieberman, Gerald: *Operations Research.* 2nd Edition, Holden-Day, San Francisco, 1967, 1974.
5. Gillet, Billy. *Introduction To Operations Research – A Computer Oriented Algorithmic Approach.* McGraw-Hill, New York, 1976.