2-5-1987

An expert system to optimize combinational logic

Gu-Ching Lin

Rochester Institute of Technology

School of Computer Science and Technology

An Expert System To Optimize Combinational Logic

by

Gu-Ching Lin

A thesis, submitted to

The Faculty of the School of Computer Science and Technology,

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Approved by:

John L. Ellis

Professor John L. Ellis (Chairman)

John A. Biles

Professor John A. Biles

George A. Brown

Professor George A. Brown

February 5, 1987

Title of Thesis: An Expert System To Optimize Combinational Logic

I Gu-Ching Lin prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

Date: May 22, 1987

ABSTRACT

Twenty to fifty percent of the active area of most semicustom integrated circuits is devoted to combinational logic. Automating the synthesis and optimization of combinational circuitry can result in significant improvements in both the design cycle time and the overall area of the implementation. This thesis presents a rule-based system that optimizes combinational logic for a given technology. By performing Boolean function minimization, decomposition, logic synthesis and a series of local transformations[4], the system achieves area reductions and saves valuable design time.

1. Introduction and Background

Optimization of combinatorial logic is a lengthy and difficult task for circuit designers. Since a significant percentage of most chips consists of combinatorial logic, this optimization can take much effort, increasing the overall turnaround time of the design. In addition, when an existing design is converted from one technology to another, circuit designers have to reoptimize the existing implementation to take full advantage of the target technology. Often this optimization is not even performed, resulting in unnecessarily large and slow chips.

This thesis presents a rule-based expert system that optimizes combinational logic for a specific target technology. The system consists of four modules, the minimization, decomposition, and synthesis module are used to generate a minimized, multilevel netlist describing the target technology for optimization module, which performs subsitiutions of equivalent gate configurations, thereby reducing the overall area of the implementation and improving the speed of the design.

1.1 Problem Statement

One concept of automated logic design is the conversion of a functional description to a logic implementation. While such logic design involves several distinct but interrelated problems

(data flow design, control logic design, physical layout, etc.), the thesis focus here is on methods primarily applicable to logic optimization.

Current techniques for automatic design of control logic fall into two broad categories. The first, which we refer to as two-level design, is characterized by the use of a two-level disjunctive form (DF) as an intermediate stage in synthesis. Roughly speaking, these techniques produce a two-level representation for a function which is to be implemented and minimize or optimize it further. The resulting representation is then factored into a network of gates, transistors, PLA's or other functional units. The key observation concerning this style of design is that there is an intermediate stage at which all information about possible algorithms which might have been contained in the original specification of the function is discarded. Only information about the function to be computed is retained.

This approach to automatic design has several advantages. First, minimization of the DF corresponds to an optimization in the space of all algorithms for the given function. This method therefore has the potential to uncover extremely clever ways to implement the function which may have been overlooked by the designer. Second, there is a firm theoretical foundation underlying this methodology. There has been much work on Boolean algebra and Boolean functions and much

of this theory relates to a two-level representation. For example, there are well understood techniques for taking advantage of "don't care" information during minimization.

There are also problems with the use of two-level minimization in the automatic generation of logic. True two-level minimization algorithms require exponential time; however, recently a technique has been discovered, ESPRESSO-IIC[1], which in actual examples, comes close to the true minimum and has an acceptable running time. A more serious difficulty is that the result of two-level minimization is a network of gates with unlimited fan-in. Current technologies have fan-in limitations, so the result of Boolean minimization cannot be directly realized in any actual technology. The original design may have had implicit information about the sharing of intermediate results which would be useful in altering the two-level design to meet technology requirements, but, in the process of putting the design into DF and minimizing the DF, this information has been lost. Rediscovering this information in order to construct a good multilevel design is the factoring problem, and currently there is a technique known as weak division[6], which takes a two-level function and creates a multilevel function based on small subexpressions.

This second category of approaches to automatic design is more closely related to the strategy used in compiling programming languages and for this reason we will refer to this

type of design methodology as compiler-like design. In this type of design, the specification is thought of as both a description of the function to be implemented and as a high-level description of an algorithm for implementing it, and throughout the compilation, information concerning the implied algorithm is retained whenever possible.

This class of design methodologies has advantages and disadvantages, which are almost complementary to those of the two-level approach. Their primary advantage is that information implicit in the specification is retained throughout the design process. This permits the process to use insights which the designer may have included in the description while also permitting computational efficiency since the factoring problem is largely avoided.

There are several disadvantages to this type of approach. First, since no global optimization (in the sense of Boolean minimization) is performed, the efficiency of the eventual implementation is limited by the form of the specification. For example, it is unlikely that a compiler-like method can accept a specification for a carry-chain adder and produce the implementation of a carry-lookahead adder. Second, the available techniques for dealing with redundancy and "don't care" are in the early stages of development. Finally, there is no firm mathematical foundation underlying these approaches in the way that there is a theory relating to Boolean algebra.

1.2 Previous Work

The goal of logic synthesis is to accept functional specifications for a hardware unit and to generate automatically a detailed, technology-specific implementation comparable in quality to that of an experienced engineer. There has been much work on automating logic design and many effective tools have been developed to aid the designer. Early work centered on developing algorithms for translating a boolean function into a minimum two-level network of boolean primitives[14]. Latter efforts attempted to raise the level of specification[15]. The results were usually more expensive than manual implementations and did not take advantage of the target technology. For example, the ALERT[16] system was validated on an existing design, the IBM 1800, and the implementation produced required 160% more circuits than the manual design.

In attempts to generate more efficient logic and to give the user more control over the implementation, other strategies were tried; computer design language simulation and boolean translation. These constrain the specification language so that there is nearly a one-to-one correspondence between the specification and the implementation. Of course, this constraint also decreases the advantage provided by the system.

Recently, interest has grown significantly in AI applications of digital system design. DAS/Logic[9] (Design Assistant Series) is a tool being developed at Carnegie Group Inc. to aid in the design of integrated circuits. DAS/Logic is a rule-based system written in OPS5[2] which refines a textual behavioral description to a circuit schematic. The system's input is a high level language description of the target IC's behavior; the output consists of a set of standard cells and an interconnection list. The system is separated into four levels. The first level is the Behavioral level ,which describes the input/output behavior of a digital system. The next level is the Generic Logic level. Here, the Behavioral description is translated into a logic representation. In the third level, the Committed Logic level, the Generic Logic representation is cast into the appropriate primitive gates for the implementation technology. For example, the Generic Logic structure is composed of AND and OR gates that correspond to the logical form of the Behavioral description. At the Committed Logic level, the AND and OR gate structures are changed into NAND or NOR gates. The final level is the Standard Cell level, where the transistor circuits required to implement a particular logic function are specified.

A number of research groups are currently exploring knowledge based approaches to various aspects of VLSI. Here we describe a knowledge based system called REDESIGN[10], which assists engineers in the redesign of digital circuits to meet

altered functional specifications. Given the redesign goal, the system generates plausible local changes to make within the circuit, ranks the changes based on implementation difficulty and goal satisfaction, and checks for undesirable side effects associated with the changes. The system provides design assistance by combining casual reasoning, analyzing the cause-effect relations of the circuits operation, with functional reasoning, and analyzing the purposes or roles of circuit components. Circuit knowledge in REDESIGN is represented as a network of modules and data paths. The system was developed at Rutgers University and reached the stage of a research prototype.

1.3 System Approach

Automating the synthesis and optimization of combinational circuitry can result in significant improvements in both the design cycle time and the overall quality of the implementation. Standard techniques, such as two-level minimization tools, for performing logic level reduction are a major step in this direction, but they fail to address the actual circuit-level implementation. Such minimizers will find an optimal implementa-tion using AND/OR gate, for example, but cannot easily take advantage of other logic circuits.

This thesis takes a four step approach to the synthesis and optimization problem for combinational logic. These steps are (Figure 1.1) :

- minimizing the Boolean equations,
- factoring the two-level functions into multilevel functions,
- synthesizing an initial network, and
- optimizing the network for a given technology.

During the minimization phase, the set of Boolean equations describing the desired functions is reduced using mathematical methods that take advantage of the "don't care" set. The ESPRESSO-IIC program performs the reductions. In the second phase, the equations are factored using a technique known as weak division, which takes a two-level function and creates a multi-level function based on small subexpressions that occur often in the original function. This tool detects and eliminates multiple occurrences of the same subexpression, otherwise it would result in duplicate logic in the synthesized circuit.

In the synthesis phase, an initial network is created by using a NAND or NOR implementation for target technology. Finally, this network is optimized for area by performing a series of local transformations[4,5] on the circuit. These transformations are formulated as rules to be applied to the circuit by a rule-based system.

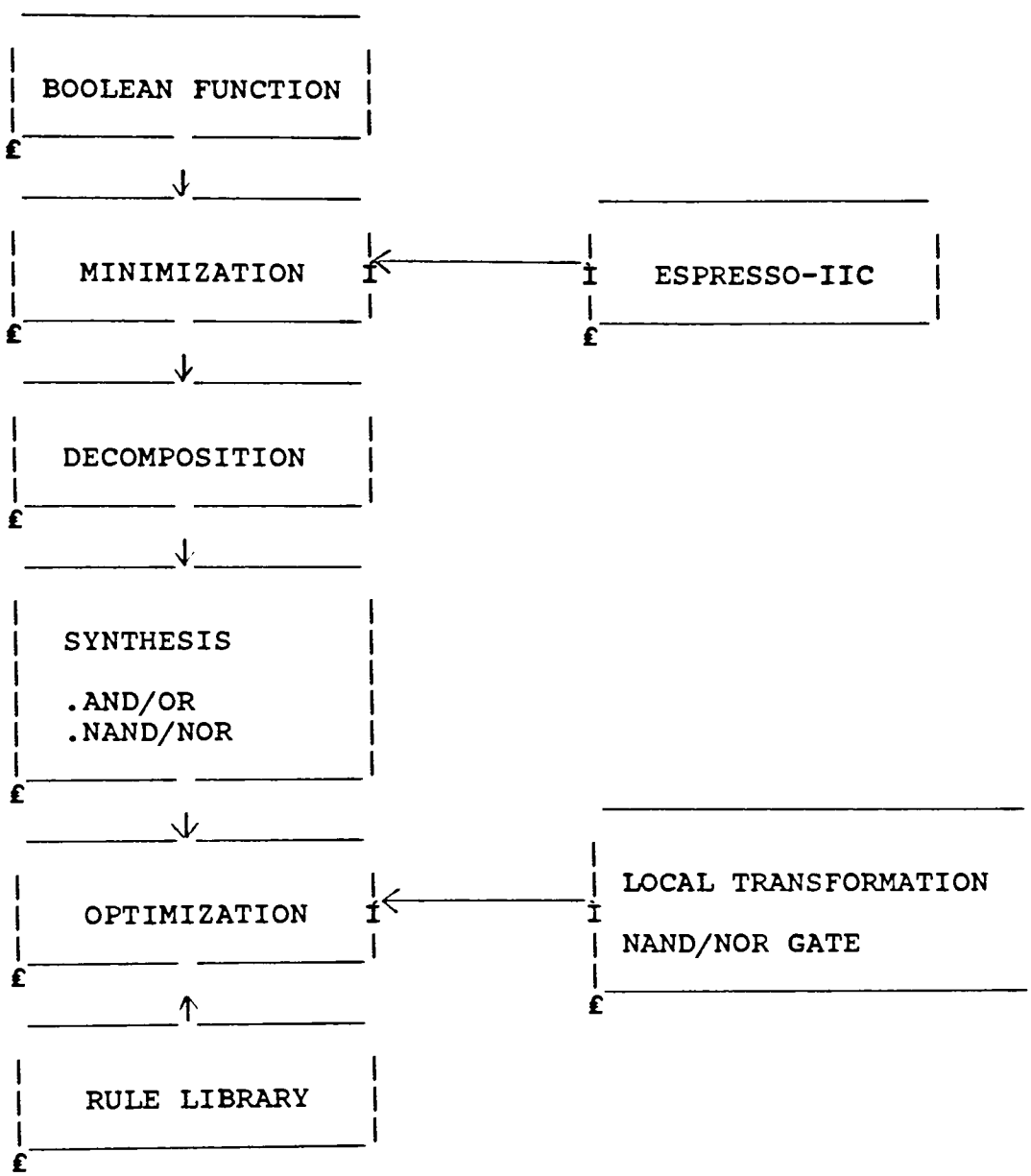


Figure 1.1 System description

2. Knowledge-Based Expert Systems

In recent years, research in the field of artificial intelligence has had many important successes. Among the most significant of these has been the development of powerful new computer systems known as "expert" or "knowledge-based" systems. These programs are designed to represent and apply factual knowledge in specific areas of expertise to solve problems. For example, collaborative efforts by human experts and system developers have resulted in systems that diagnose diseases, configure computer systems, and prospect for minerals at performance levels equal to or surpassing human expertise. The potential power of systems that can replicate expensive or rare human knowledge has led to a worldwide effort to extend and apply this technology.

An expert system essentially consists of a knowledge base and an inference engine. The knowledge base contains facts and rules that use those facts as the basis for decision making. The inference engine contains an interpreter that decides how to apply the rules to infer new knowledge and a scheduler that decides the order in which the rules should be applied. This organization is shown in Figure 2.1.

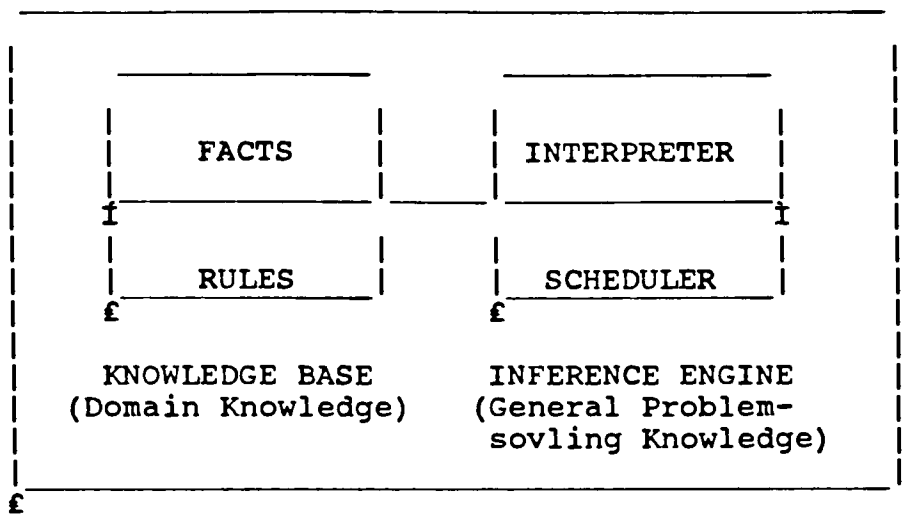


Figure 2.1 The structure of an expert system

In expert system, knowledge is used to slove problem and determine new facts based upon what is already known. The knowledge should be efficiently usable and easily expandable. Knowledge, therefore, has to be represented for quick and easy retrieval, for ease in further expansion and modification, for use in reasoning or solving a specific problem. In order to satisfy the various requirements for the knowledge representation, different techniques have to be used for different types of knowledge. There are three most widely used in current expert systems are rules (the most popular), semantic nets and frames.[24] Each technique provides the program with certain benefits, such as making it more efficient, more easily understood, or more easily modified.

The inference engine uses knowledge in the knowledge base to solve a specific problem by emulating the reasoning process of a humnan expert. The approach to solving a problem consists of searching a solution from a search space. In AI terminology, the set of all possible solutions is known as the search space. The inference engine contains problem-solving strategies that use knowledge in the knowledge base to serch for a solution.

Recently, the use of knowledge-based expert systems in digital system design has grown. One of the most successful applications of such systems is the R1[7] system used at DEC to configure large computer systems. The rest of this chapter focuses on rule base systems and describes the applicability of knowledge-based expert systems to logic design.

2.1 Productions systems

A production system consists of a rule base and a control structure. The rule base is composed of a list of production rules which are checked repeatedly until a condition is achieved or rejected. The controls structure determines which rules should be executed next and executes the actions specified by the rules.

A production rule is a statement cast in the form "If this condition holds, then this action is appropriate." The Figure 2.2 and 2.3 show a transformation rule is encoded as a production rule in this system.

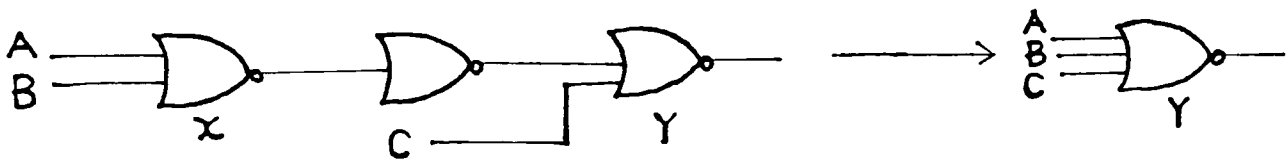


Figure 2.2 A transformation rule

IF
a NOR inverter is connected to the output of a NOR gate X and input to another NOR gate Y.

THEN
remove the NOR inverter and NOR gate X from netlist and connect input A,B to NOR gate Y.

Figure 2.3 A production rule

In the Figure 2-3, the IF part of the productions, called the condition part, states the conditions that must be present for the production to be applicable, and the THEN part, called the action part, is the appropriate action to take.

The control structure uses a rule interpreter, sometimes encoded in terms of "metarules" to find the enabled rules and to decide which rule to apply. One basic control strategy used is data driven or event driven and starts from the available information as it comes in, trying to draw conclusions that are appropriate to the goals. This is how the system in this thesis works. In production systems this is called a forward chaining method of inference. We sometimes work the other way, however, starting from a goal or expectation of what is to happen and working backwards, looking for evidence that supports or contradicts our expectation. This is called goal driven or expectation driven and in production systems it is referred to as backward chaining, since it requires looking at the action parts of rules to find ones that would conclude the current goal, then looking at the condition sides of those rules to find out what conditions would make them execute, then finding other rules whose action parts conclude these conditions, and so on.

Data-driven approaches sometimes have the disadvantage of generating many hypotheses not directly related to the problem under consideration, while goal-driven approaches have the disadvantage of perhaps becoming fixed on an initial set of hypotheses and having difficulty shifting focus when the data available do not support them.

Not all expert system are rule-based. Rule-based systems are particularly attractive when much of the expert knowledge in the field comes from empirical associations acquired as a result of experience.

2.2 Applicability of Knowledge-Based System to Logic Optimization

Knowledge-based expert systems are very costly to implement at the present time. This is mainly because: the lack of knowledge engineers and adequate sophisticated support tools, unfamiliarity of knowledge engineers with the application area, and unfamiliarity of the experts in the application area with knowledge-based expert system. These characteristics makes it necessary to evaluate the candidate application areas very carefully in terms of the applicability of the knowledge-based expert system approach. Two key ingredients for successful application of knowledge-based expert systems has been suggested by the Stanford AI group[8]: attack problems amenable to the techniques of applied AI, and consider only important, difficult, and high-value problems. We look at the above two requirements in the logic design area.

1. Logic optimization is amenable to the techniques of applied of AI.
    - By accumulating design experience, knowledge-based expert systems can imitate human problem solving capabilities more accurately than traditional tools.

- Since optimization techniques depend heavily on the target technology, by using a rule-based system, optimization for different technologies involves only changing the rule library.

2. Logic optimization is an important, difficult and high-value problem.
   - The large number of conferences organized in computer aided design of digital system, and the enormous number of papers published in logic circuit design are testimony to the importance and difficulty of the problem.
   - Logic optimization is a high-value problem because auotmatic optimization of logic circuits can improve both the logic area and design time.

2.3 Advantages and Disadvantages of Knowledge-Based Expert System

In summary, the advantages of knowledge-based expert systems include: the ease with which human knowledge can be encoded, the modularity and incremental development of knowledge-based expert systems, the ease of modification, and the capability of knowledge-based expert systems to explain their decisions. Disadvantages include: their cost of development, the slow execution speed with present technology, the difficulty of extracting knowledge from human designers, and the inefficiencies with modularity maintenance in large systems.

Despite their disadvantages, knowledge-based expert systems have proven to be a valuable approach and their capabilities increase as more applications are attempted, as more people understand the nature of these systems, and as more suitable hardware and software tools are developed.

3. System Implementation

Numerous tools are available that optimize and implement combinational logic. Most of these tools apply at the Boolean level, are technology-independent, and generally assume an AND/OR implementation. Such tools fail to take advantage of the various types of gates available in a semicustom library. The need for more flexible and technology-oriented tools was recognized by Darringer, et al[5]., who implemented a design system to perform local transformations at various levels of abstraction. In the system built for this thesis, the local transformations were formulated as rules to optimize gate-level circuits for area in a given technology, and Prolog[3] was selected as a formalism to represent these rules for a rule-based system.

The system is divided into four main parts: a minimization, mathematical reduction module; a decomposition, multilevel function creation module; a synthesis, gate-level implementation module; and an optimization, local transformation module. This chapter discusses these modules in detail and describes the rules that the system uses to perform optimization. The control structure that applies local transformations to the circuits will also be described in this chapter.

3.1 Minimization

The goal of building an optimizing digital circuit will require the efficient manipulation of Boolean logic functions. The minimization module reduces the set of Boolean equations describing the logic by using heuristics that find a minimal set of prime implicants. In finding this minimal set, the module takes advantage of the "don't care" set of the function. The ESPRESSO-IIC program cteated by Brayton et al. in 1982, performs the reductions. The goals in the design of ESPRESSO-II were to build a logic minimization tool such that in most cases

- the problem submitted by a logic designer could be solved with the use of limited computing resources;
- the final results would be close to a global optimum.

Although ESPRESSO-II follows the basic techniques used in most minimization tools, generation of all prime implicants and extraction of a minimum prime cover, the algorithms employed in ESPRESSO-II are new and quite different. Efficient Boolean manipulation is achieved through the "unate recursive paradigm"[27], which is employed in complementation, tautology and other algorithms. All of these algorithms make ESPRESSO-II as an efficient minimization tool for logic functions with more than 30 inputs and outputs and with more than 100 products terms.

3.2 Decomposition

One of the problems with using two-level minimization is that the result of two-level minimization is a network of gates with unlimited fan-in. This part describes a technique that reconstructs a multilevel function for logic design by identifying subexpressions common to two or more functions. By creating a new variable to represent such subexpressions, we also reduce the complexity of the original function at the cost of adding a new intermediate function. In general, this reduces the number of logical components required to implement the set of functions. The decomposition approach is algebraic as opposed to Boolean. The result is an algorithm, which, by successive substitution of new variable for common subexpression, simplifies a set of functions until they are "relative prime".

Given a set of Boolean expressions, our aim is to pull out common subexpressions, consisting of two or more cubes (a set of variables), until the expression becomes relatively kernal (subexpression) free. It is then easy to locate single cubes dividing two or more functions. By pulling these out as well, we can reduce our expression to a set whose only common divisors are single variables. At this point the expression can be implemented independently with no loss of efficiency; all global commonality has been identified. Let f and g be expressions, and let v be a variable appearing in neither. Let r(f,g) denote the set f - (f/g)g, the remainder resulting from the division of f by g.

Then the substitution s(v,g,f) of v for g in f is the expression (f/g)v + r(f,g). If g is A + B, f is C(A + B) + D, then s(x,g,f) = Cx + D. The kernal (common subexpression) A + B is pulled out from f and g, and replaced by a new variable x. The algorithm for computing f/g and kernal are shown below.

To compute f/g:

1. Let g = (a_i) and for each i set $h_i = (b_j \mid a_i b_j \in f)$.
2. Set f/g = h_i.

To compute kernal(f):

We number the literals appearing in f as $l_1, l_2, \ldots, l_n$.
Let kernal(0,f) = kernal(f). Set kernal(f) = $\emptyset$.
|c| = number of cubes in f.
kernal(k,f):
For i = k + 1 to n
  Let c = f/l_i
  if |c| $\geq$ 2 and i = n then
    kernal(l_i,f) = c
  else if |c| $\geq$ 2 then
    kernal(l_i,f) = kernal(i,c)
  if kernal(f) $\cap$ kernal(l_i,f) = $\emptyset$ then
    kernal(f) = kernal(f) $\cap$ kernal(l_i,f)
  else
    kernal(f) = kernal(f) U kernal(l_i,f)
next i

We now describe the first step of decomposition, which identifies expressions consisting of two or more cubes that occur in several functions. This process is called "distillation"[6].

The Distill Algorithm:

1. Generate all kernals for each function.
2. Select a pair kernals (k,k'), where $k \in f_i$, $k' \in f_j$, $i \neq j$, such that $|k \cap k'| \geq 2$. if no such pair exists, stop.
3. Record $(v, K \cap K')$ for some new variable v.
4. Set $f_i = s(v, k \cap k', f_i)$ for each function.
5. Go to 1.

The total number of variables in the function decreases with each pass, hence the algorithm terminates. The particular pair $k \cap k'$ selected in step 2 can influence the quality of the resulting decomposition. In the program, a useful heuristic is to select the pair whose substitution most reduces the number of variables appearing in the functions. This heuristic was implemented by recording each subexpression produced from the intersection of kernals, the subexpression appearing most often in the record is chosen for the step 2.

To complete the decomposition process, the next step is to pull out those cubes consisting of more than two variables that exist in several functions. This process is referred to as "condensation"[6].

The Condense Algorithm:

1. Select cubes, $c \in f_i$ and $c' \in f_j$, $i \neq j$, such that $|c \cap c'| \geq 2$. if no such pair exists, stop.
2. Record $(v, c \cap c')$ for some new variable v.
3. Set $f_i = s(v, c \cap c', f_i)$ for each function.
4. Go to 1.

The total decomposition process consists of distillation followed immediately by condensation. The pairs $(v, c \cap c')$ generated by condensation are added to the list of $(v, k \cap k')$ produced by distillation. As before, a selection heuristic can be applied in step 1.

Example: Let $f = AB(C(D + E) + F + G) + H$ and

$g = AI(C(D + E) + F + J) + K$

Distillation:

Pass 1: kernal(f) = D + E; kernal(g) = D + E

$|k \cap k'| = D + E \geq 2$

set L = D + E

then $f = AB(CL + F + G) + H$

$g = AI(CL + F + J) + K$

Pass 2: kernal(f) = CL + F + G

kernal(g) = CL + F + J

set $M = |k \cap k'| = CL + F \geq 2$

then $f = AB(M + G) + H$

$g = AI(M + J) + K$

now f and g are relatively kernal free.

Condensation:

Pass 1: set $N = ABM \cap AIM = AM \geq 2$

then $f = B(N + AG) + H$

then $g = I(N + AJ) + K$

the substitution list is:

$L = D + E$

$M = CL + F$

$N = AM$

The decomposition takes about 14 minutes for a combinational logic with 12 outputs, 37 inputs and 140 product terms, but the running time is mostly spent in doing the intersection for each kernal. Since we try to select a common subexpression that appears most often in the functions, we need to generate all common subexpressions and compare each of them in reducing the number of variables in the functions. If we organize these common subexpressions according to their appearance times and let the process go back to step 2 to select the second order of subexpression after substituting a new variable for the functions, then the running time is largely reduced by skipping a lot of time spent in doing the intersection. When this algorithm is applied to a sample with 29 functions having 23 input variables, it takes about 5 hours compared with the first algorithm which takes 13 hours to decompose this sample. It saves up to 70% of the CPU time in the Pyramid 90/X. The savings of time depends on the size of the functions. Of course, since this algorithm can not always select the best common subexpression, it

results in many more logical components than the first algorithm. The algorithm listed below is called "fast decomposition" and the first algorithm is called "optimal decomposition".

The Distill Algorithm:

1. Generate all kernals for each function.
2. Select a pair, where $k \in f_i$, $k' \in f_j$, such that $|k \cap k'| \geq 2$. This process proceeds repeatedly until every kernal has been selected, then these subexpressions ($|k \cap k'|$) are ordered according to their appearance times. If no subexpression exists, stop.
3. Select first common ubexpression, if no such common subexpression exists, go to 1.
4. Record (v, $k \cap k'$) for some new variable v.
5. Set $f_i = s(v, k \cap k', fi)$ for each function.
6. go to 2.

The first step of Condense algorithm is also modified.

1. Select cubes, $c \in f_i$, $c' \in f_j$, such that $|c \cap c'| \geq 2$. This process proceeds repeatedly until every cube has been selected, then these subexpressions ($|c \cap c'|$) are ordered according to their appearance times. if no subexpression exists, stop.

Now we look another example that has more structure than the first one. The incoming data and resulting design for part of a 16-bit bus structure are shown below. The var1 through var8 are new variables and the common subexpressions associated with them.

the incoming data:

```
f3 = h'i'j'k'r + e'f'ar + c'ap'r + b'ap'r + d'ar + aps + h'i'jkr
     + h'ikl'r + d'hjkl'r + b'hjk'l'p'r + e'hjkl'f'r + c'hjkl'p'r
     + hjkl'ps

f4 = h'i'j'k't + c'ap't + b'ap't + e'f'at + d'at + apu + h'i'jkt
     + h'ikl't + e'hjkl'f't + d'hjkl't + c'hjkl'p't + b'hjkl'p't
     + hjkl'pu

f5 = h'i'j'k'v + d'av + b'ap'v + c'a p'v + e'f'av + apw + h'i'jkv
     + h'ikl'v + c'hjkl'p'v + e'hjkl'f'v + b'hjkl'p'v + d'hjkl'v
     + jhkl'pw

f6 = h'i'j'k'x + d'ax + c'ap'x + b'ap'x + e'f'ax + apy + h'i'jkx
     + h'ikl'x + c'hjkl'p'x + b'hjkl'p'x + d'hjkl'x + e'hjkl'f'x
     + hjkl'py

f7 = h'i'j'kz + d'az + b'ap'z + e'f'az + c'ap'z + apa1 + h'i'jkz
     + h'ikl'z + c'hjkl'p'z + b'hjkl'p'z + e'hjkl'f'z + d'hjkl'z
     + hjkl'pa1

f8 = h'i'j'k'b1 + e'f'ab1 + d'ab1 + c'ap'b1 + apc1 + h'i'jkb1
     + h'ikl'b1 + e'hjkl'f'b1 + d'hjkl'b1 + b'hjkl'p'b1
     + c'hjkl'p'b1 + hjkl'pc1

f9 = h'i'j'k'x1 + ap'x1 + apd1 + h'i'jkx1 + h'ikl'x1 + hjkl'p'x1
     hjkl'pd1

f10 = h'i'j'k'e1 + ap'e1 + apy1 + h'i'jke1 + hikl'e1 + hjkl'p'e1
      + hjkl'py1

f11 = h'i'j'k'g1 + ap'g1 + apf1 + h'i'jkg1 + hikl'g1 + hjkl'p'g1
      + hjkl'pf1

f12 = h'i'j'k'i1 + ap'i1 + aph1 + h'i'jji1 + h'ikl'i1 + hjkl'p'i1
      hjkl'ph1
```

The results of decomposition are shown below. Each of the lines numbered 1 throgh 8 gives a subexpression and the new variable associated with it. For example, the line numbered 1 associates the new variable var1 with the expression j'k'+jk. The variable var1 is used in place of the expression j'k'+jk for the functions that contain the expression j'k+jk.

```
1. var1 = j'k' + jk
2. var2 = a + hjkl'
3. var3 = ikl' + var1i'
4. var4 = b' + c'
5. var5 = e'f' + d' + var4p'
6. var6 = var3h' + var5var2
7. var7 = var2p' + var3h'
8. var8 = var2p


f3 = var6r + var8s
f4 = var6t + var8u
f5 = var6v + var8w
f6 = var6x + var8y
f7 = var6z + var8a1
f8 = var6b1 + var8c1
f9 = var7x1 + var8d1
f10 = var7e1 + var8y1
f11 = var7g1 + var8f1
f12 = var7i1 + var8h1
```

3.3 Synthesis

The synthesis module translates a Boolean function into a gate-level implementation. Two synthesis modules were built, one that generates an AND/OR implementation of the function derived from the previous module and one that generates a NAND or NOR implementation for particular target technology from the AND/OR implementation.

The first synthesis module is relatively straightforward and implements the function as an interconnected netlist. The second synthesis module converts the netlist to a network composed of NAND gates or NOR gates. This conversion is carried out by using F = (F')' and then applying DeMorgan's laws:

$$(X_1 + X_2 + \ldots + X_n)' = X_1' X_2' \ldots X_n'$$

$$(X_1 X_2 \ldots X_n)' = X_1' + X_2' + \ldots + X_n'$$

The Figure 3.1 illustrates conversion of two-level forms.

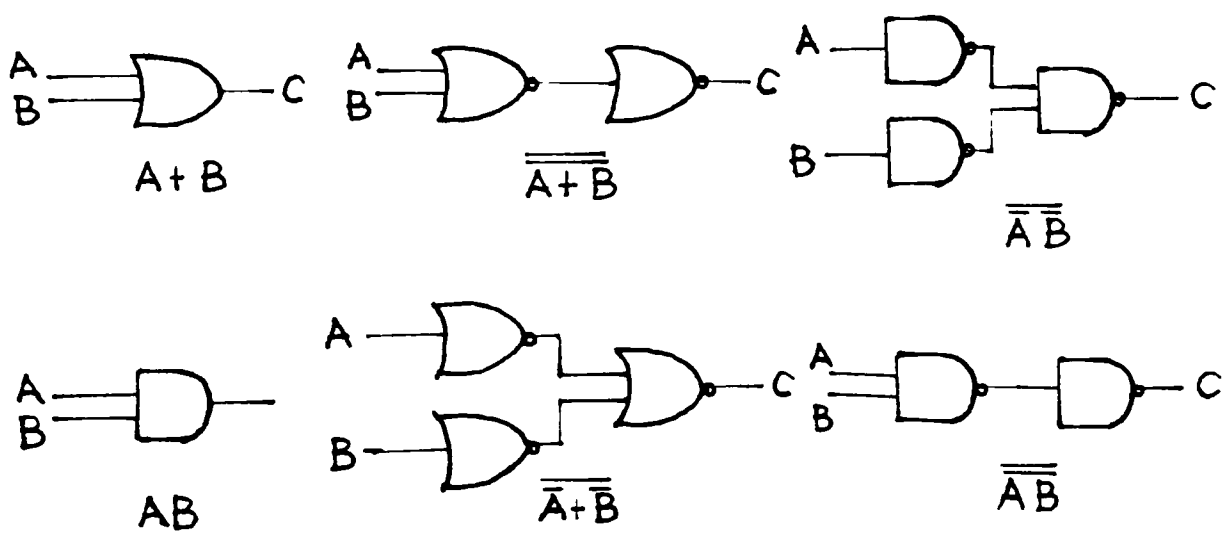


Figure 3.1 NAND/NOR Implementation

The above conversion produces a lot of cascaded inverters in a multilevel netlist, but since the double inversion does not alter a logic function, it should not appear in network. In this module, the conversion is performed by using rules and by passing an output signal to the next lower level, so that the cascaded inverters are not found in the NAND/NOR implementation. The output signal tells a gate that its output is logic 0 or logic 1 when it is converted from AND/OR logic to NAND/NOR logic. The number of gates cascaded in series between a network input and the output is referred to as the number of 'levels' of gates. The highest level is the network output. As shown in Figure 3.2, the network has 4 levels, the first level is gate5, the fouth level is gate1. The logic conversion is started from the gates that are in the highest level. The rules for obtaining the NOR network from a AND/OR netlist are as follow:

1. If the output signal is 1 and the gate is an AND gate, then change the AND gate to NOR gate and pass a 0 signal to the gates that are in the next lower level.
2. If the output signal is 0 and the gate is an AND gate, then change the AND gate to NOR gate followed by a NOR inverter, and pass a 0 signal to the gates that are in the next lower level.
3. If the output signal is 1 and the gate is an OR gate, then change the OR gate to NOR gate followed by a NOR, inverter, and pass a 1 signal to the gates that are in the next lower level.

4. If the output signal is 0 and the gate is an OR gate, then change the OR gate to a NOR gate and pass a 1 signal to the gates that are in the next lower level.

The rules for obtaining the NAND network from AND/OR implementation are exactly the same as for NOR network except the signal is as opposed as NOR logic. A fan-in constraint has been added for the NAND implementation so that a NAND gate can have no more than four inputs in a NAND gate. An example to illustrate these rules is provided below.

Figure 3.2 is traced from gate 1 which is an OR gate with an output signal 1. After applying rule 3, gate 1 is changed to a NOR gate followed by a NOR inverter, and a signal 1 is passed to level 3 which includes gate 2 and gate 3. Since gate 2 is an AND gate with an output signal 1, rule 1 is selected and applied in netlist resulting in a NOR gate and a signal 0 for next lower level (level 2). Gate 4 is also an AND gate but with an output signal 0, by applying rule 2, the AND gate is replaced by a NOR gate with an inverter and a signal 0 for level 1. Rule 4 translates gate 5 from an OR gate to a NOR gate and passes signal 1 to input variable A. The process backtracks to input variable B and finishes conversion in level 1. Since D and F are input variables, there is no gate connected to gate 2 and gate 4, the process goes back to gate 1. Before going to gate 3 it should be mentioned that the AND/OR implementation still exists in the data base. Gate 3 is replaced by a NOR gate, and an inverter, and a

signal 1 for gate 5 that we mentioned before still exists. Gate 5 is translated to a NOR gate and an inverter but the NOR gate with input A, B can be found in the network, so only the inverter is added to the netlist. The process is stopped until the NOR implementation has been completed.

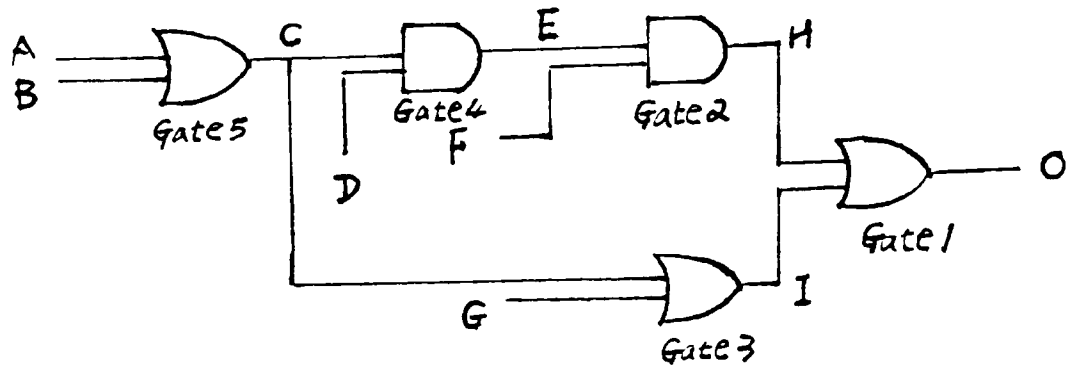


Figure 3.2 before synthesis

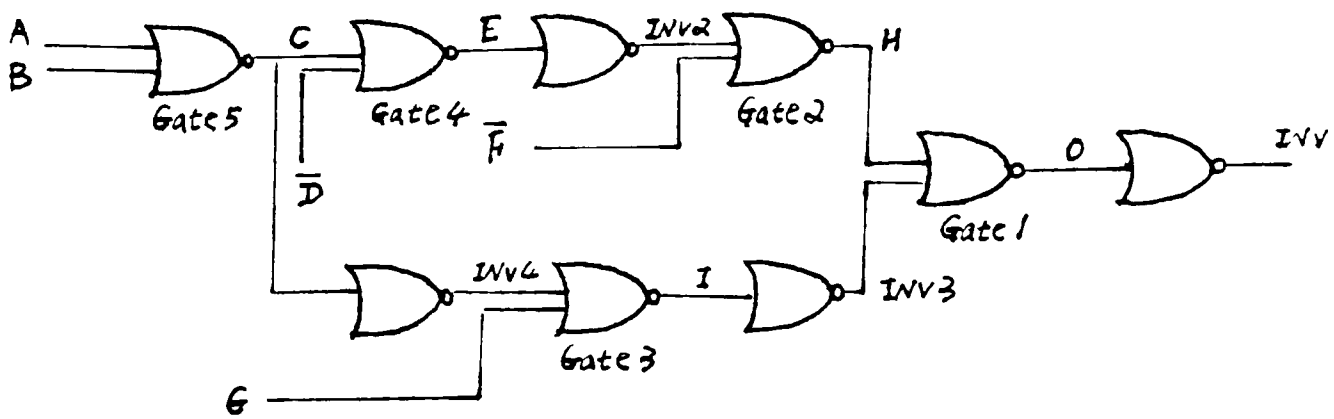


Figure 3.3 after synthesis

3.4 Optimization

The optimization module performs a succession of substitutions on an existing netlist, similar to the way an experienced designer manipulates a design to achieve greater efficiency. The module consists of a knowledge base and a control structure.

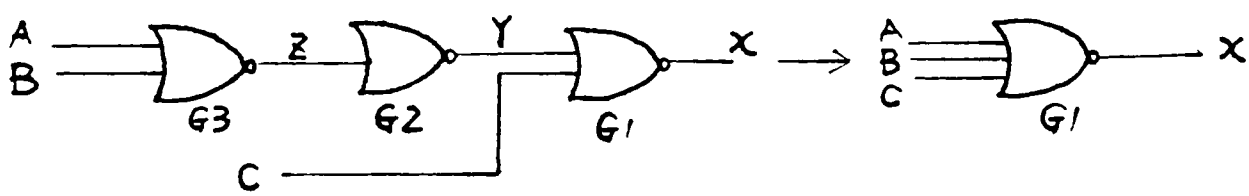
The system optimizes a circuit by performing a series of local transformations to that circuit. In performing each transformation, the program replaces a given configuration of gates by another functionally equivalent configuration of gates. These transformations are always applied in such a way as to reduce circuit areas and produce more optimal circuits. The example of such a transformation rule was shown in chapter 2. The control structure of a rule-based system directs the application of the rules. During the application, a meta-rule determines what rules or sequences of rules are applicable to the circuit.

3.4.1 Knowledge-Base

In the system, a rule is a mechanism to replace a portion of a circuit by a functionally equivalent but more desirable circuit portion. Rules are sorted as a netlist describing a target configuration to be recognized in the circuit and an associated action detailing how to build the replacement

configuration. Substituting the replacement configuration for the target configuration is to reduce the overall area of the circuit.

During the building of the rule library, it became apparent that a large numbers of rules differing from each other by the number of input variables existed. To reduce the number of these equivalent rules, we incorporated these rules to a general rule. A rule represented in Prolog is shown in Figure 3.4. The "adjust_netlist" clause removes the NOR inverter G2 and NOR gate G3 from netlist and connects the input variable A and B to NOR gate G1.




```
nor_rule_1(X,Y) :-
        node_(nor,Y,Z),
        inverter(nor,Y,Z),
        gate_(nor,Z,Vars),
        adjust_netlist(nor,X,Y,Vars).
```

Figure 3.4 A rule represented in Prolog

The rules within rule library are ordered by their desirability. A meta-rule in the system determines the appropriate order of rules based on how many times a rule has been used in the circuit. There are sixteen rules in the system now (see Appendix C), half for NOR implementation half for NAND. The control structure will use rules from the first rule until there is no applicable rule in the knowledge base. Although these rules are specific to a given technology, optimization for different technologies involves only changing the library containing the rules.

3.4.2 Control Structure

A problem-solving that uses forward reasoning and whose operators each work by producting a single new object -- a new state -- in the database is said to represent problems in a state space representation.[24] The problem of producing a state that satisfies a goal condition can now be formulated as the problem of searching a graph to find a node whose associated state description satisfies the goal. The graph, which grows as the search proceeds, will be referred to as a search graph or search tree.

Optimization of combinational logic through successive transformations can be translated to the problem of optimally traversing a state space. The nodes of this graph are the implementations of the circuit, and the arcs represent rule applications. The root of the tree corresponds to the current

implementation. Optimization is equivalent to finding a path from the initial circuit configuration to an optimal configuration. The process of finding this path is refered to as a state space search. The state space search strategy used in this system is presented below. The strategy uses heuristic information to decide which node to expand next, the information is provided by a meta-rule.

```
select rule
for each rule R in some class C
  for each gate G in the circuit
    if the target for R matches at gate G
    then apply rule go back to select rule
```

The value of C controls what a given technology is optimized. For instance, when optimzing for NOR gate, the class of NOR rules are used. The first step of the system selectes rules based on the ordering of the knowledge base, always applying the first applicable rule in the knowledge. The above search strategy is called a best-first search.

We use an example to explain how a state changes to another, and how the state describing a target configuration matches the condition part of a selected rule. The forward chaining inference method will be used to implement the matching operation; it starts from the available information and try to infer the conclusions that are appropriate to the goal. When the

search function is called, it is given two arguments: a rule, and a gate at which the search function should begin. Figure 3.5 shows a sample netlist, a portion of a circuit, and a rule to be applied in that circuit. In searching, the controller first checks if gate G1 is a NOR gate. If so, then the controller selects one input from the input variables of gate G1. If P2 is an input variable for gate G1 and connects to another NOR gate G2, then the controller checks to see if gate G2 is an inverter. Since the G2 is not an inverter, the searching activity fails. At this point a condition cannot be met, so the controller backtracks by returning to the last selection it made and making a different choice. Backtracking continues until the rule's condition part has been satisfied, or until all possible choices have been rejected. As this example illustrates, the controller goes back to G1 and takes another choice. P3 is an another input variable for gate G1 and connects to a NOR gate G3. Since the gate G3 is an inverter between gates G4 and G1, the rule is executed and a replacement function is called to perform a transformation for the circuit. The replacement function is contained in each transformation rule.

In order to make the control structure more flexible, a meta-rule was implemented in the system. As noted above, the meta-rule determines appropriate order of the transformation rules. By adjusting these rules, control structure can be used perhaps with a better result.

A sample netlist:

```
(nor,p1,p2,p3)
(nor,p2,p4,p5)
(nor,p3,p6)
(nor,p6,p7,p8)
```

A rule selected by control structure:

```
nor_rule_1(X,Y) :-
    node_(nor,Y,Z),
    inverter(nor,Y,Z),
    gate_(nor,Z,Vars)
    adjust_netlist(nor,X,Y,Vars)
```

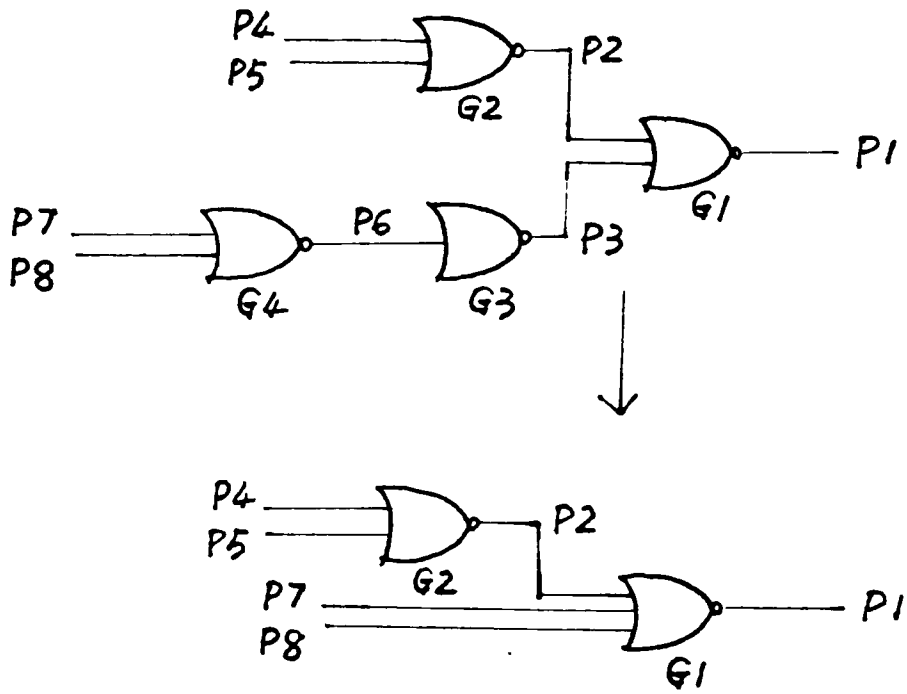


Figure 3.5 Logic optimization represented in state space search, each circuit configuration is a state.

3.5 Program Organization

The minimization module is written in C, and the other modules are written in Prolog. The reason for this division is that the minimization module uses ESPRESSO-IIC to perform reduction, whcih manipulates matrices during minimization, and are much easier to implement in a conventional language. The other modules are implemented in Prolog based on following observations: In Prolog it is easy to represent the functional behavior of gates, all the transformations needed for the system can be entered as rules and implemented in Prolog, and Prolog provides an efficient pattern-directed inference tool.

3.6 Input and Output

The input of the system is in sum of products function set or in a truth table including input and output variables. The system output is in form of a netlist which describes target logic type, output variables and input variables. The following netlist is a sample output (Figure 3.6).

```
(nor,f12,nor1,nor2)
(nor,nor1,gate22,i1)
(nor,gate22,/p,var2,/h1)
(nor,/h1,h1)
(nor,/p,p)
(nor,var2,gate39,a)
(nor,gate39,1,/k,/j,/h)
(nor,/k,k)
(nor,/j,j)
(nor,/h,h)
(nor,nor2,nor21,gate47,gate46)
(nor,gate47,h,var3)
(nor,var3,gate41,gate40)
(nor,gate41,i,var1)
(nor,var1,gate38,gate37)
(nor,gate38,/k,/j)
(nor,gate37,k,j)
(nor,gate40,1,/k,/i)
(nor,/i,i)
(nor,gate46,p,var2)
(nor,nor21,var2,/h1)
```

Figure 3.6 A sample output

4. Conclusion and Future Work

This thesis has described the development of a system that is capable of automatically synthesizing and transforming functional specifications into gate level implementations. The system is implemented as a rule-based system because it works without an established algorithm and is easy to modify according to the target technology. The system progresses through four distinct modules during the design process: Minimization, Decomposition, Synthesis and Optimization. For larger examples above 100 gates, the system achieved area reductions ranging from 20% to 30% from unoptimized circuits. these results are comparable to the result of manual optimization. The flexibility of the system is largely due to its separation into independently useful modules. It may be use for translating two-level functions to a multilevel implementation, generating a circuit, or optimizing an existing circuit. In any of these applications, the system saves valuable time and space.

Future work can be classified into two categories: enhancements to the system to improve its present performance; and new approaches to logic design.

1. An Additional feature to improve system performance includes building a rule entry module to help users easily extend the knowledge base. The rule entry module would automatically generate a pattern describing the target configuration and an

action describing how to replace it with the replacement configuration.

2. New approaches include extension of the current NAND/NOR implementation to other gates available in gate-array or standard cell libraries, and consideration of the fan-in and timing constraints.

Appendix A. User's Manual

NAME

preopt - generate Prolog accepted form from truth table

SYNOPSIS

preopt [file]

DESCRIPTION

preopt generates a form suitable for Prolog from a truth table which defines a set of Boolean functions. If no output file is specified, the default output file "_opt " is generated (e.g. test_opt). Since Prolog does not accept a period within file name, all the file name including a period will be changed to "_".

Input to preopt is in the form of truth table, that is the output from Espresso-IIC program. Comments are allowed within the input by placing a pound sign (#) as the first character on a line. Comments and unrecognized keywords are passed directly from the input file to output file. Any white-space (blanks, tab, etc.) is ignored. Output is used as input for logopt (see Figure A.1). The example in appendix B will show how the program progresses.

The following keywords are reserved for the system use, they should not appear in the input file for input variable or any other variable.

key words: gate(n), var(n), inv, in, out, nor(n), nand(n), norgate(n), nandgate(n).

the (n) means 1, 2, 3.....n. (e.g. var1, var2, var3....var10)

SEE ALSO

Eqntott,Espresso

NAME

logopt - combinational logic optimization

SYNOPSIS

logopt [option] [type] [file]

DESCRIPTION

logopt takes as input a two-level Boolean functions optimized functional equivalent netlist. The system consists of four distinct modules: minimization, decomposition, synthesis and optimization. The optimization module is implemented by a rule-based system.

logopt reads the file provided, performs the optimization for logic area, and writes an optimized netlist to a default output file "_out" (e.g. test_out) if no output file is specified. The system generates a command file "_com" (e.g. test_com) for Prolog programming, it can be deleted after running.

"type" specifies the target technology for the system. The allowed types are -nand for NAND gate, and -nor for NOR gate. The output netlist only includes NOR gate if -nor type is specified.

"option" specifies boolean function decomposition which creates multilevel function by identifying common sub-expression from a two-level function. The allowed options are -f for " fast decomposition", and -o for "optimal decomposition". Although "fast decomposition" saves a lot of time, the logic area is its tradeoff.

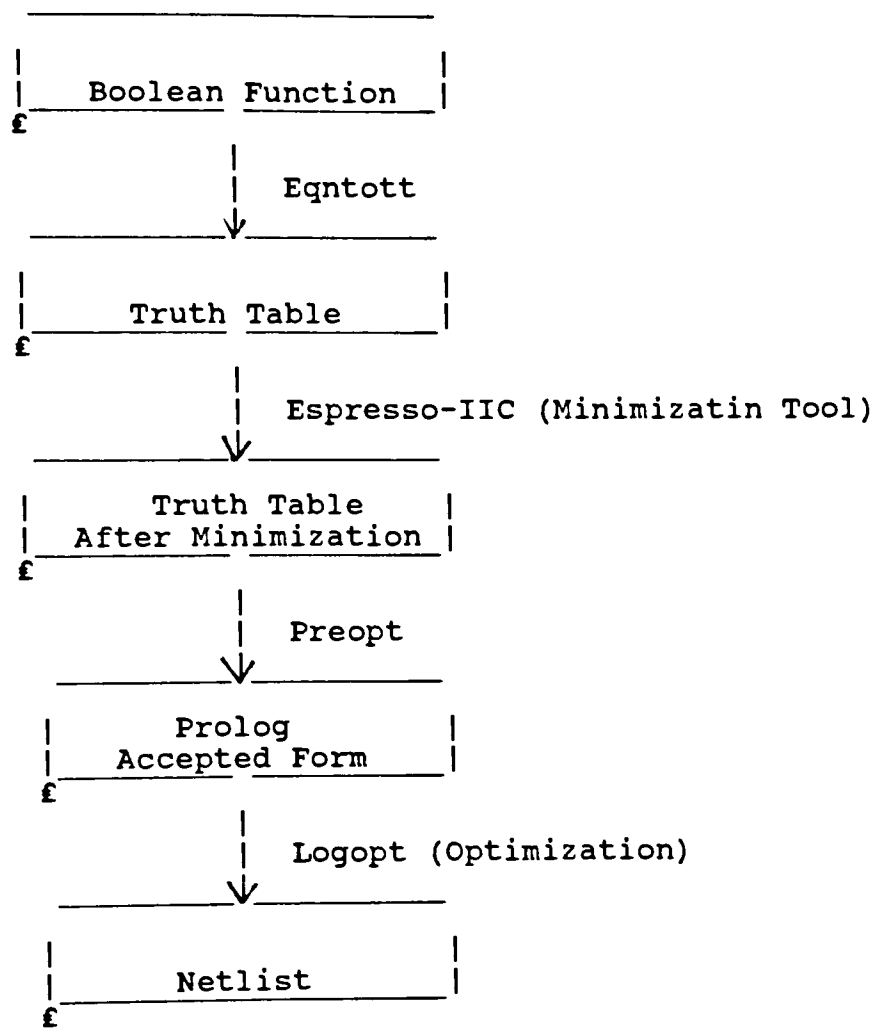


Figure A.1 Program Organization

Appendix B. Example

Example 1

```
/* input data (sample.bol) */

f0 = b & c & d & e & h & !i & j & k & !l ;
f0 = b & c & d & f & h & !i & j & k & !l ;
f0 = d & e & h & i & j & k & !l & o ;
f0 = d & f & h & i & j & k & !l & o ;
f0 = !d & e & h & i & j & k & !l & m & n ;
f0 = !d & f & h & i & j & k & !l & m & n ;
f0 = a & b & c & d & e ;
f0 = a & b & c & d & f ;
f0 = a & !d & g ;
f0 = a & !e & !f & g ;
f0 = !d & g & !i & j & k & !l ;
f0 = !e & !f & g & !i & j & k & !l ;
f0 = g & !h & i & k & !l ;
f0 = g & !h & j & k & !l ;
f0 = g & !h & !i & !j & !k ;
f0 = !h & !i & j & k & l & !p ;
f0 = d & !e & !f & h & i & j & k & !l & m ;
f2 = !d & e & !h & !i & j & k & l & m & n & !p ;
f2 = !d & f & !h & !i & j & k & l & m & n & !p ;
f2 = d & e & !h & !i & j & k & l & o & !p ;
f2 = d & f & !h & !i & j & k & l & o & !p ;
f2 = d & e & h & !i & j & k & !l & o ;
f2 = d & f & h & !i & j & k & !l & o ;
f2 = a & !d & e & m & n ;
f2 = a & !d & f & m & n ;
f2 = a & d & e & o ;
f2 = a & d & f & o ;
f2 = !d & !i & j & k & !l & q ;
f2 = !e & !f & !i & j & k & !l & q ;
f2 = d & !e & !f & !h & !i & j & k & l & m & !p ;
f2 = !h & !i & j & k & q ;
f2 = !h & !i & !j & !k & q ;
f2 = !h & i & k & !l & q ;
f2 = a & d & !e & !f & m ;
f3 = a & !b & !p & r ;
f3 = a & !c & !p & r ;
f3 = !b & h & j & k & !l & !p & r ;
```

```
f3 = !c & h & j & k & !l & !p & r ;
f3 = a & !d & r ;
f3 = a & !e & !f & r ;
f3 = !d & h & j & k & !l & r ;
f3 = !e & !f & h & j & k & !l & r ;
f3 = !h & !i & !j & !k & r ;
f3 = !h & !i & j & k & r ;
f3 = !h & i & k & !l & r ;
f3 = a & p & s ;
f3 = h & j & k & !l & p & s ;
f4 = a & !b & !p & t ;
f4 = a & !c & !p & t ;
f4 = !b & h & j & k & !l & !p & t ;
f4 = !c & h & j & k & !l & !p & t ;
f4 = a & !d & t ;
f4 = a & !e & !f & t ;
f4 = !d & h & j & k & !l & t ;
f4 = !e & !f & h & j & k & !l & t ;
f4 = !h & !i & !j & !k & t ;
f4 = !h & !i & j & k & t ;
f4 = !h & i & k & !l & t ;
f4 = a & p & u ;
f4 = h & j & k & !l & p & u ;
f5 = a & !b & !p & v ;
f5 = a & !c & !p & v ;
f5 = !b & h & j & k & !l & !p & v ;
f5 = !c & h & j & k & !l & !p & v ;
f5 = a & !d & v ;
f5 = a & !e & !f & v ;
f5 = !d & h & j & k & !l & v ;
f5 = !e & !f & h & j & k & !l & v ;
f5 = !h & !i & !j & !k & v ;
f5 = !h & !i & j & k & v ;
f5 = !h & i & k & !l & v ;
f5 = a & p & w ;
f5 = h & j & k & !l & p & w ;
f6 = a & !b & !p & x ;
f6 = a & !c & !p & x ;
f6 = !b & h & j & k & !l & !p & x ;
f6 = !c & h & j & k & !l & !p & x ;
f6 = a & !d & x ;
f6 = a & !e & !f & x ;
f6 = !d & h & j & k & !l & x ;
f6 = !e & !f & h & j & k & !l & x ;
f6 = !h & !i & !j & !k & x ;
f6 = !h & !i & j & k & x ;
f6 = !h & i & k & !l & x ;
f6 = a & p & y ;
f6 = h & j & k & !l & p & y ;
f7 = a & !b & !p & z ;
f7 = a & !c & !p & z ;
f7 = !b & h & j & k & !l & !p & z ;
```

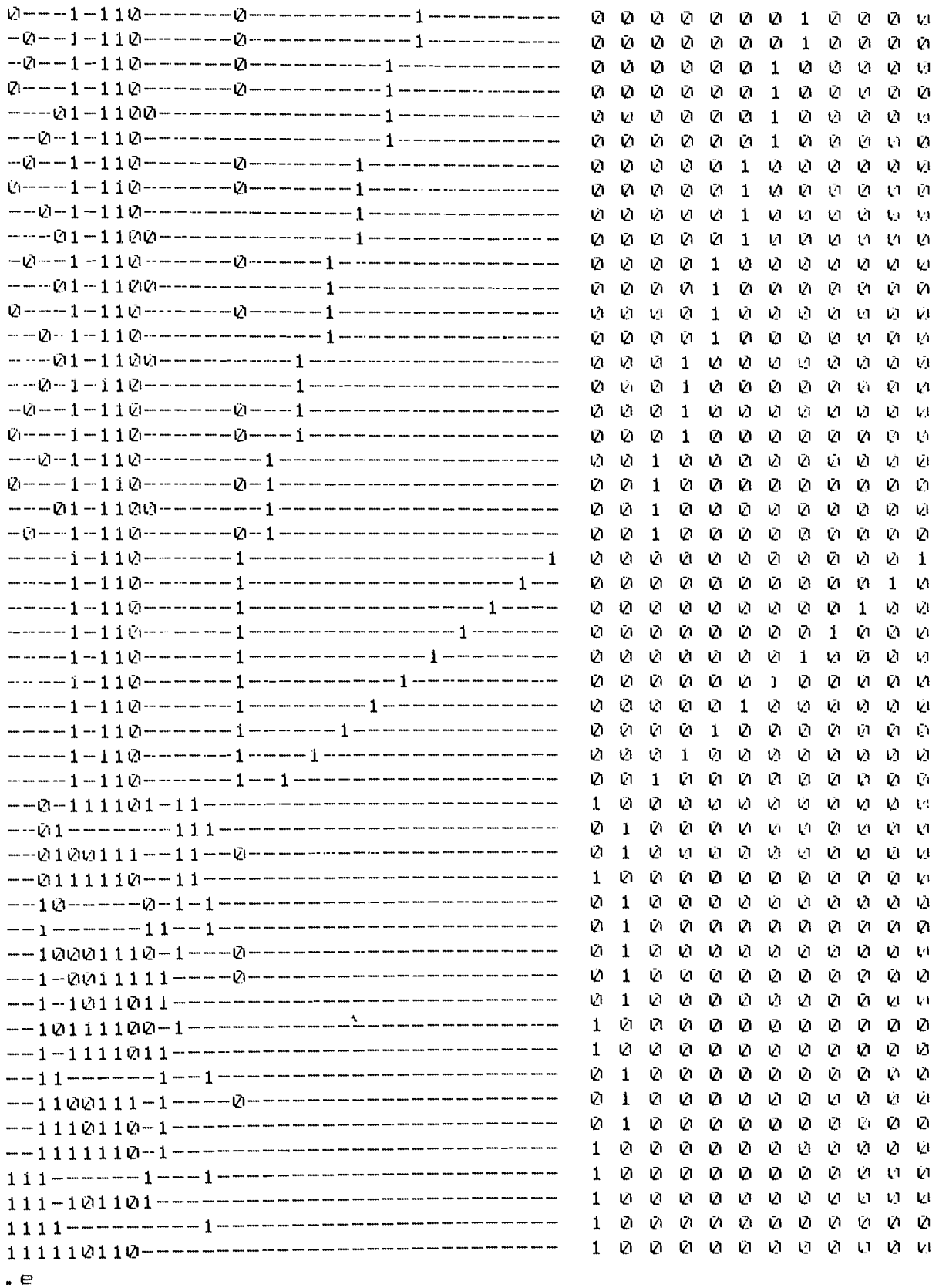
```
f7 = !c & h & j & k & !l & !p & z ;
f7 = a & !d & z ;
f7 = a & !e & !f & z ;
f7 = !d & h & j & k & !l & z ;
f7 = !e & !f & h & j & k & !l & z ;
f7 = !h & !i & !j & !k & z ;
f7 = !h & !i & j & k & z ;
f7 = !h & i & k & !l & z ;
f7 = a & p & a1 ;
f7 = h & j & k & !l & p & a1 ;
f8 = a & !b & !p & b1 ;
f8 = a & !c & !p & b1 ;
f8 = !b & h & j & k & !l & !p & b1 ;
f8 = !c & h & j & k & !l & !p & b1 ;
f8 = a & !d & b1 ;
f8 = a & !e & !f & b1 ;
f8 = !d & h & j & k & !l & b1 ;
f8 = !e & !f & h & j & k & !l & b1 ;
f8 = !h & !i & !j & !k & b1 ;
f8 = !h & !i & j & k & b1 ;
f8 = !h & i & k & !l & b1 ;
f8 = a & p & c1 ;
f8 = h & j & k & !l & p & c1 ;
f9 = a & !p & x1 ;
f9 = h & j & k & !l & !p & x1 ;
f9 = a & p & d1 ;
f9 = h & j & k & !l & p & d1 ;
f9 = !h & !i & !j & !k & x1 ;
f9 = !h & !i & j & k & x1 ;
f9 = !h & i & k & !l & x1 ;
f10 = a & !p & e1 ;
f10 = h & j & k & !l & !p & e1 ;
f10 = a & p & y1 ;
f10 = h & j & k & !l & p & y1 ;
f10 = !h & !i & !j & !k & e1 ;
f10 = !h & !i & j & k & e1 ;
f10 = !h & i & k & !l & e1 ;
f11 = a & !p & g1 ;
f11 = h & j & k & !l & !p & g1 ;
f11 = a & p & f1 ;
f11 = h & j & k & !l & p & f1 ;
f11 = !h & !i & !j & !k & g1 ;
f11 = !h & !i & j & k & g1 ;
f11 = !h & i & k & !l & g1 ;
f12 = a & !p & i1 ;
f12 = h & j & k & !l & !p & i1 ;
f12 = a & p & h1 ;
f12 = h & j & k & !l & p & h1 ;
f12 = !h & !i & !j & !k & i1 ;
f12 = !h & !i & j & k & i1 ;
f12 = !h & i & k & !l & i1 ;
```

```
step 1: eqntott -r -l sample.bol > sample.eqn


/* sample.eqn */


.i 37
.o 12
.ma sample
.ilb  b c d e h i j k l f o m n a g p q r s t u v w x y z a1 b1
 c1 x1 d1          e1 y1 g1 f1 i1 h1
.ob  f0 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12
.p 140
------0000----------------------------1-  0 0 0 0 0 0 0 0 0 0 0 1
-----00000----------------------------1---  0 0 0 0 0 0 0 0 0 0 1 0
-----00000--------------------------1-----  0 0 0 0 0 0 0 0 0 1 0 0
-----00000------------------------1-------  0 0 0 0 0 0 0 0 1 0 0 0
-----00000----------------------1---------  0 0 0 0 0 0 0 1 0 0 0 0
-----00000--------------------1-----------  0 0 0 0 0 0 1 0 0 0 0 0
-----00000------------------1-------------  0 0 0 0 0 1 0 0 0 0 0 0
-----00000----------------1---------------  0 0 0 0 1 0 0 0 0 0 0 0
-----00000--------------1-----------------  0 0 0 1 0 0 0 0 0 0 0 0
-----00000------------1-------------------  0 0 1 0 0 0 0 0 0 0 0 0
-----00000----------1---------------------  0 1 0 0 0 0 0 0 0 0 0 0
-----00000--------1-----------------------  1 0 0 0 0 0 0 0 0 0 0 0
----------------1-0---------------------1-  0 0 0 0 0 0 0 0 0 0 0 1
--------------1-0-------------------1---  0 0 0 0 0 0 0 0 0 0 1 0
------------1-0-----------------1-------  0 0 0 0 0 0 0 0 0 1 0 0
----------1-0---------------1-----------  0 0 0 0 0 0 0 0 1 0 0 0
---0-----0---1-------------1-------------  0 0 0 0 0 0 0 1 0 0 0 0
0-------------1-0-----------1-----------  0 0 0 0 0 0 0 1 0 0 0 0
--0----------1-----------1---------------  0 0 0 0 0 0 0 1 0 0 0 0
-0-----------1-0---------1---------------  0 0 0 0 0 0 0 1 0 0 0 0
--0----------1---------1-----------------  0 0 0 0 0 0 1 0 0 0 0 0
0------------1-0-------1-----------------  0 0 0 0 0 0 1 0 0 0 0 0
---0-----0---1---------1-----------------  0 0 0 0 0 0 1 0 0 0 0 0
--0----------1-------1-------------------  0 0 0 0 0 1 0 0 0 0 0 0
0------------1-0-----1-------------------  0 0 0 0 1 0 0 0 0 0 0 0
-0-----------1-0-----1-------------------  0 0 0 0 1 0 0 0 0 0 0 0
---0-----0---1-------1-------------------  0 0 0 0 1 0 0 0 0 0 0 0
-0-----------1-0---1---------------------  0 0 0 1 0 0 0 0 0 0 0 0
0------------1-0---1---------------------  0 0 0 1 0 0 0 0 0 0 0 0
---0-----0---1-----1---------------------  0 0 0 1 0 0 0 0 0 0 0 0
--0----------1-----1---------------------  0 0 0 1 0 0 0 0 0 0 0 0
---0-----0---1---1-----------------------  0 0 1 0 0 0 0 0 0 0 0 0
```

```
-0----------1-0-1-------------------  0 0 1 0 0 0 0 0 0 0 0 0
0----------1-0-1-------------------  0 0 1 0 0 0 0 0 0 0 0 0
--0----------1---1-------------------  0 0 1 0 0 0 0 0 0 0 0 0
-------------1-1----------------------1  0 0 0 0 0 0 0 0 0 0 0 1
-------------1-1----------------------1--  0 0 0 0 0 0 0 0 0 0 1 0
-------------1-1----------------1-----  0 0 0 0 0 0 0 0 0 1 0 0
-------------1-1-------------1--------  0 0 0 0 0 0 0 0 1 0 0 0
-------------1-1----------1----------  0 0 0 0 0 0 0 1 0 0 0 0
-------------1-1--------1------------  0 0 0 0 0 0 1 0 0 0 0 0
-------------1-1------1--------------  0 0 0 0 0 1 0 0 0 0 0 0
-------------1-1----1----------------  0 0 0 0 1 0 0 0 0 0 0 0
-------------1-1--1------------------  0 0 1 0 0 0 0 0 0 0 0 0
--0----------11--------------------  1 0 0 0 0 0 0 0 0 0 0 0
---0------0---11--------------------  1 0 0 0 0 0 0 0 0 0 0 0
--0-------1-111----------------------  0 1 0 0 0 0 0 0 0 0 0 0
-----00110-------------------------1-  0 0 0 0 0 0 0 0 0 0 0 1
----00110-----------------------1---  0 0 0 0 0 0 0 0 0 0 1 0
----00110---------------------1-----  0 0 0 0 0 0 0 0 0 1 0 0
----00110-------------------1-------  0 0 0 0 0 0 0 0 1 0 0 0
----00110-----------------1---------  0 0 0 0 0 0 0 1 0 0 0 0
----00110---------------1-----------  0 0 0 0 0 0 1 0 0 0 0 0
----00110-------------1-------------  0 0 0 0 0 1 0 0 0 0 0 0
----00110-----------1---------------  0 0 0 0 1 0 0 0 0 0 0 0
----00110---------1-----------------  0 0 0 1 0 0 0 0 0 0 0 0
----00110-------1-------------------  0 0 1 0 0 0 0 0 0 0 0 0
---0-01100-------1-------------------  0 1 0 0 0 0 0 0 0 0 0 0
----0-110------1---------------------  1 0 0 0 0 0 0 0 0 0 0 0
---0-011000----1---------------------  1 0 0 0 0 0 0 0 0 0 0 0
--0---0110-------1---------------------  1 0 0 0 0 0 0 0 0 0 0 0
----00111------0---------------------  1 0 0 0 0 0 0 0 0 0 0 0
--0-00111110--0---------------------  0 1 0 0 0 0 0 0 0 0 0 0
----01-10----------------------------1-  0 0 0 0 0 0 0 0 0 0 0 1
----01-10----------------------------1---  0 0 0 0 0 0 0 0 0 0 1 0
----01-10--------------------------1-----  0 0 0 0 0 0 0 0 0 1 0 0
----01-10----------------------1-------  0 0 0 0 0 0 0 0 1 0 0 0
----01-10--------------------1---------  0 0 0 0 0 0 0 1 0 0 0 0
----01-10------------------1-----------  0 0 0 0 0 0 1 0 0 0 0 0
----01-10----------------1-------------  0 0 0 0 0 1 0 0 0 0 0 0
----01-10--------------1---------------  0 0 0 0 1 0 0 0 0 0 0 0
----01-10------------1-----------------  0 0 0 1 0 0 0 0 0 0 0 0
----01-10----------1-------------------  0 0 1 0 0 0 0 0 0 0 0 0
----01-10--------1---------------------  0 1 0 0 0 0 0 0 0 0 0 0
----01-10------1-----------------------  1 0 0 0 0 0 0 0 0 0 0 0
----1-110------0----------------------1-  0 0 0 0 0 0 0 0 0 0 0 1
----1-110------0--------------------1---  0 0 0 0 0 0 0 0 0 0 1 0
----1-110------0----------------1-------  0 0 0 0 0 0 0 0 0 1 0 0
----1-110------0--------------1---------  0 0 0 0 0 0 0 0 1 0 0 0
---01-1100-------------------1---------  0 0 0 0 0 0 0 1 0 0 0 0
--0-1--110-----------------1---------  0 0 0 0 0 0 0 1 0 0 0 0
```

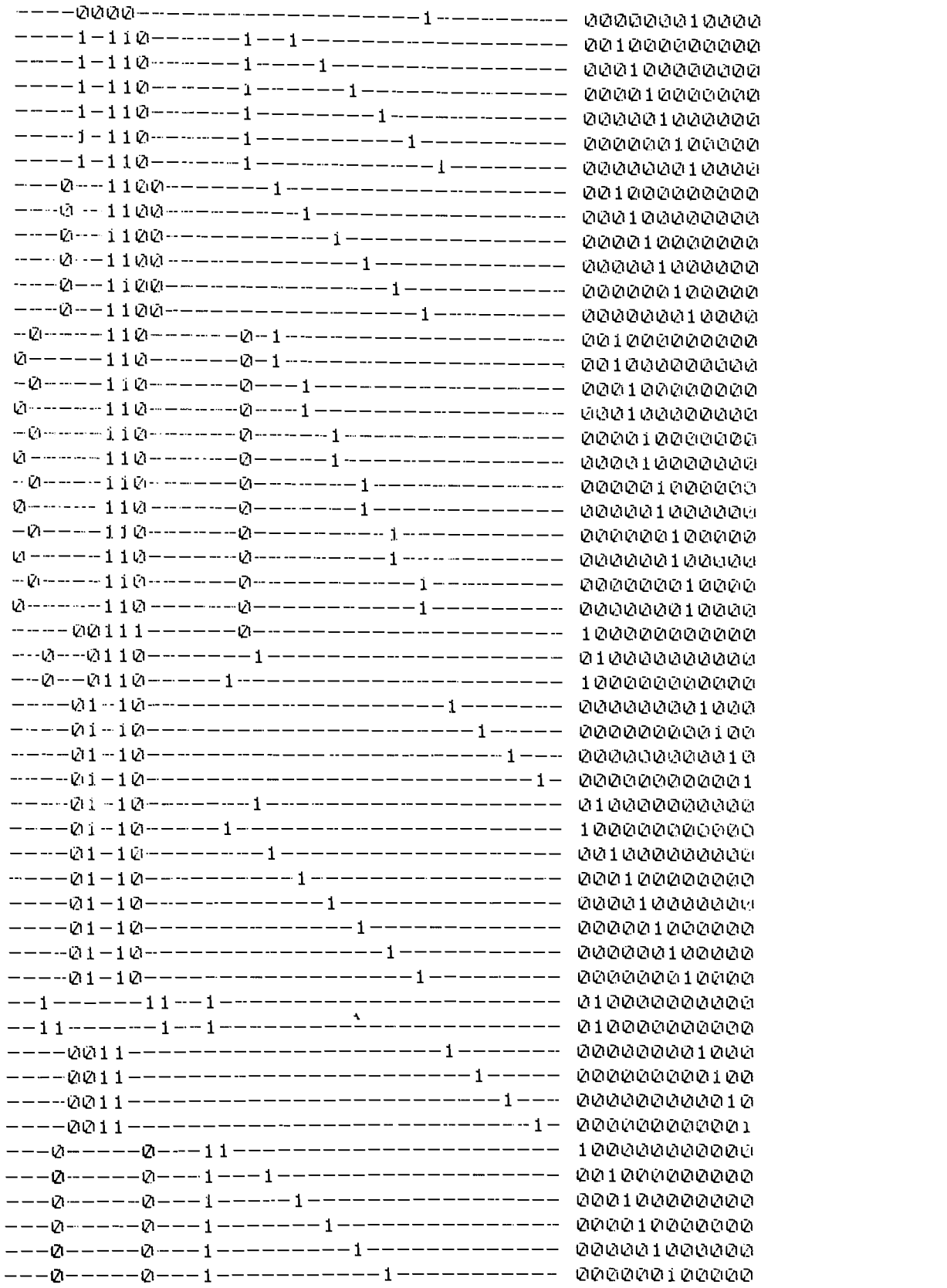


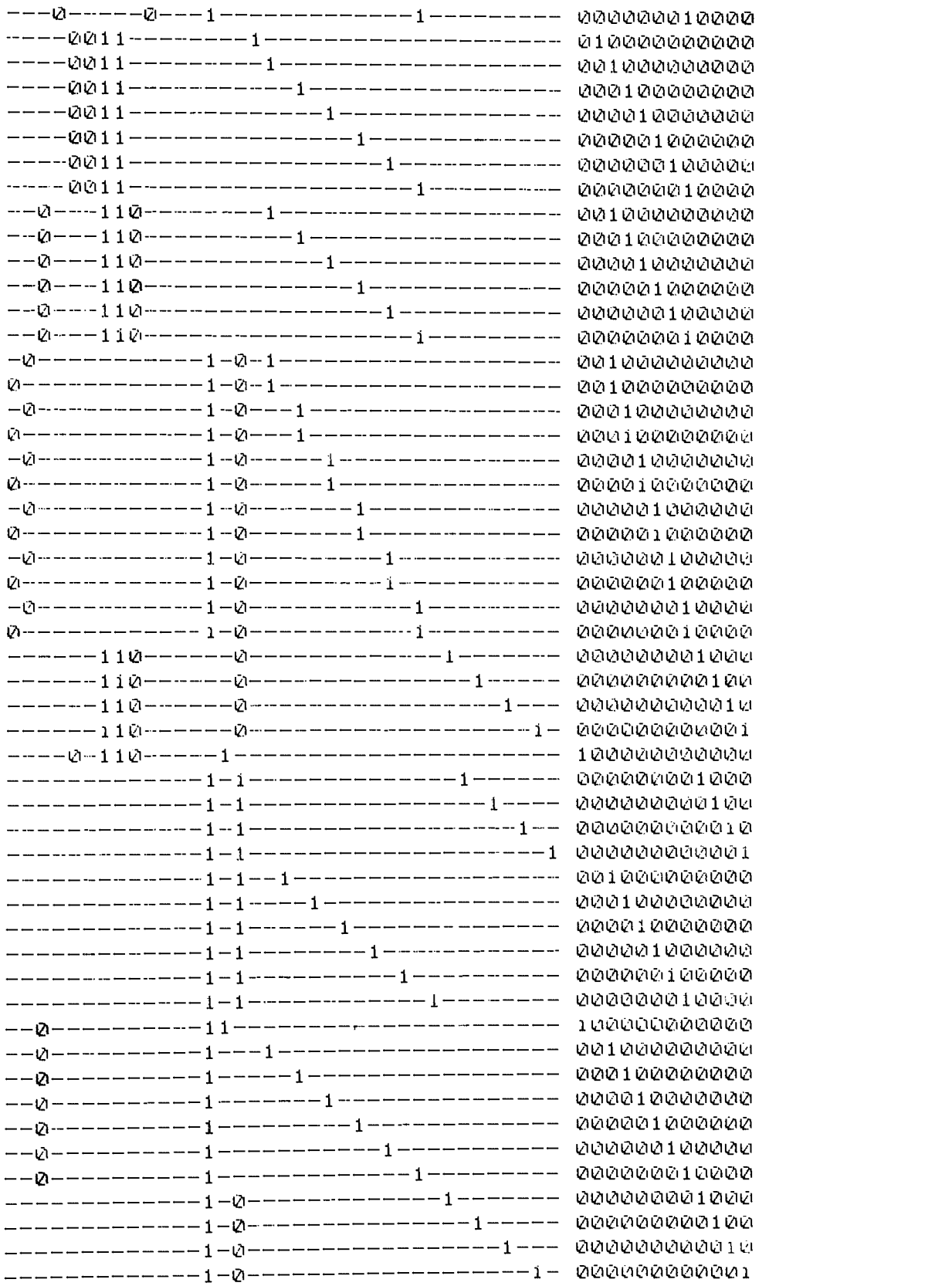
.e

step 2: espresso sample.eqn > sample.esp

```
/* sample.esp */


.na sample
.ilb  b c d e h i j k l f o m n a g p q r s t u v w x y z a1 b1
 c1 x1 d1          e1 y1 g1 f1 i1 h1
.ob  f0 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12
.i 37
.o 12
.p 140
--0-000111-11--0---------------------- 010000000000
---0100111--11--0---------------------- 010000000000
---1000111-0---0---------------------- 010000000000
--0-1111010-11--------------------------- 100000000000
---0111110---11---------------------------- 100000000000
111-1011010-------------------------------- 100000000000
111110110---------------------------------- 100000000000
--1-0001111----0---------------------- 010000000000
---11001111-----0---------------------- 010000000000
--1011110-1---------------------------- 100000000000
---1-1111011---------------------------- 100000000000
---1111110-1---------------------------- 100000000000
---1-1011011---------------------------- 010000000000
---1110110-1---------------------------- 010000000000
----0-0110-------1---------------------- 010000000000
----0-0110----1---------------------- 100000000000
111---------0---1---------------------- 100000000000
1111----------1---------------------- 100000000000
---0------1-111---------------------- 010000000001
--01----------111---------------------- 010000000000
-----00000---------------------1-------- 000000001000
-----00000----------------------1------- 000000000100
-----00000-----------------------1---- 000000000010
-----00000------------------------1- 000000000001
-----1-110------1-----------1------ 000000001000
-----1-110------1-----------------1---- 000000000100
-----1-110------1-----------------1-- 000000000010
-----1-110------1-----------------1 000000000001
--10------0-1-1---------------------- 010000000000
-----00000--------1-------------------- 010000000000
-----00000------1---------------------- 100000000000
-----00000---------1-------------------- 001000000000
-----00000-----------1----------------- 000100000000
-----00000-------------1--------------- 000010000000
-----00000---------------1------------- 000001000000
-----00000-----------------1----------- 000000100000
```





```
step 3: preopt sample.esp

/* sample_opt (default output file) */


/* .na sample2 */
/* .ilb  b c d e h i j k l f o m n a p p q r s t u v w x y z a1
   b1 c1 x1 d1 e1 y1 g1 f1 i1 h1 */
/* .ob  f0 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 */
/* .i 37 */
/* .o 12 */
/* .p 140 */
/* .e */
fn_(f0,['/d',h,i,j,k,'/l',i,m,n]).
fn_(f0,['/d',o,h,i,j,k,'/l',m,n]).
fn_(f0,[b,c,d,h,'/i',j,k,'/l',f]).
fn_(f0,[b,c,d,e,h,'/i',j,k,'/l']).
fn_(f0,[d,'/e',h,i,j,k,'/l','/f',m]).
fn_(f0,[d,h,i,j,k,'/l',f,o]).
fn_(f0,[d,e,h,i,j,k,'/l',o]).
fn_(f0,['/e','/i',j,k,'/l','/f',g]).
fn_(f0,[b,c,d,f,a]).
fn_(f0,[b,c,d,e,a]).
fn_(f0,['/h','/i','/j','/k',g]).
fn_(f0,['/h','/i',j,k,l,'/p']).
fn_(f0,['/d','/i',j,k,'/l',g]).
fn_(f0,['/h',i,k,'/l',g]).
fn_(f0,['/e','/f',a,g]).
fn_(f0,['/h',j,k,'/l',g]).
fn_(f0,['/d',a,g]).
fn_(f2,['/d','/h','/i',j,k,l,f,m,n,'/p']).
fn_(f2,['/d',e,'/h','/i',j,k,l,m,n,'/p']).
fn_(f2,[d,'/e','/h','/i',j,k,l,'/f',m,'/p']).
fn_(f2,[d,'/h','/i',j,k,l,f,o,'/p']).
fn_(f2,[d,e,'/h','/i',j,k,l,o,'/p']).
fn_(f2,[d,h,'/i',j,k,'/l',f,o]).
fn_(f2,[d,e,h,'/i',j,k,'/l',o]).
fn_(f2,['/e','/i',j,k,'/l','/f',g]).
fn_(f2,['/d',f,m,n,a]).
fn_(f2,['/d',e,m,n,a]).
fn_(f2,[d,'/e','/f',m,a]).
fn_(f2,['/h','/i','/j','/k',g]).
fn_(f2,['/d','/i',j,k,'/l',g]).
fn_(f2,['/h',i,k,'/l',g]).
fn_(f2,[d,f,o,a]).
fn_(f2,[d,e,o,a]).
```

```
fn_(f2,['/h','/i',j,k,q]).
fn_(f3,['/h','/i','/j','/k',r]).
fn_(f3,[h,j,k,'/l',p,s]).
fn_(f3,['/e',j,k,'/l','/f',r]).
fn_(f3,['/c',j,k,'/l','/p',r]).
fn_(f3,['/b',j,k,'/l','/p',r]).
fn_(f3,['/h',i,k,'/l',r]).
fn_(f3,['/e','/f',a,r]).
fn_(f3,['/h','/i',j,k,r]).
fn_(f3,['/d',j,k,'/l',r]).
fn_(f3,['/c',a,'/p',r]).
fn_(f3,['/b',a,'/p',r]).
fn_(f3,[a,p,s]).
fn_(f3,['/d',a,r]).
fn_(f4,['/h','/i','/j','/k',t]).
fn_(f4,[h,j,k,'/l',p,u]).
fn_(f4,['/e',j,k,'/l','/f',t]).
fn_(f4,['/c',j,k,'/l','/p',t]).
fn_(f4,['/b',j,k,'/l','/p',t]).
fn_(f4,['/h',i,k,'/l',t]).
fn_(f4,['/e','/f',a,t]).
fn_(f4,['/h','/i',j,k,t]).
fn_(f4,['/d',j,k,'/l',t]).
fn_(f4,['/c',a,'/p',t]).
fn_(f4,['/b',a,'/p',t]).
fn_(f4,[a,p,u]).
fn_(f4,['/d',a,t]).
fn_(f5,['/h','/i','/j','/k',v]).
fn_(f5,[h,j,k,'/l',p,w]).
fn_(f5,['/e',j,k,'/l','/f',v]).
fn_(f5,['/c',j,k,'/l','/p',v]).
fn_(f5,['/b',j,k,'/l','/p',v]).
fn_(f5,['/h',i,k,'/l',v]).
fn_(f5,['/e','/f',a,v]).
fn_(f5,['/h','/i',j,k,v]).
fn_(f5,['/d',j,k,'/l',v]).
fn_(f5,['/c',a,'/p',v]).
fn_(f5,['/b',a,'/p',v]).
fn_(f5,[a,p,w]).
fn_(f5,['/d',a,v]).
fn_(f6,['/h','/i','/j','/k',x]).
fn_(f6,[h,j,k,'/l',p,y]).
fn_(f6,['/e',j,k,'/l','/f',x]).
fn_(f6,['/c',j,k,'/l','/p',x]).
fn_(f6,['/b',j,k,'/l','/p',x]).
fn_(f6,['/h',i,k,'/l',x]).
fn_(f6,['/e','/f',a,x]).
fn_(f6,['/h','/i',j,k,x]).
fn_(f6,['/d',j,k,'/l',x]).
fn_(f6,['/c',a,'/p',x]).
fn_(f6,['/b',a,'/p',x]).
fn_(f6,[a,p,y]).
```

```
fn_(f6,['/d',a,x]).
fn_(f7,['/h','/i','/j','/k',z]).
fn_(f7,[h,j,k,'/l',p,a1]).
fn_(f7,['/e',j,k,'/l','/f',z]).
fn_(f7,['/c',j,k,'/l','/p',z]).
fn_(f7,['/b',j,k,'/l','/p',z]).
fn_(f7,['/h',i,k,'/l',z]).
fn_(f7,['/e','/f',a,z]).
fn_(f7,['/h','/i',j,k,z]).
fn_(f7,['/d',j,k,'/l',z]).
fn_(f7,['/c',a,'/p',z]).
fn_(f7,['/b',a,'/p',z]).
fn_(f7,[a,p,a1]).
fn_(f7,['/d',a,z]).
fn_(f8,['/h','/i','/j','/k',b1]).
fn_(f8,[h,j,k,'/l',p,c1]).
fn_(f8,['/e',j,k,'/l','/f',b1]).
fn_(f8,['/c',j,k,'/l','/p',b1]).
fn_(f8,['/b',j,k,'/l','/p',b1]).
fn_(f8,['/h',i,k,'/l',b1]).
fn_(f8,['/e','/f',a,b1]).
fn_(f8,['/h','/i',j,k,b1]).
fn_(f8,['/d',j,k,'/l',b1]).
fn_(f8,['/c',a,'/p',b1]).
fn_(f8,['/b',a,'/p',b1]).
fn_(f8,[a,p,c1]).
fn_(f8,['/d',a,b1]).
fn_(f9,['/h','/i','/j','/k',x1]).
fn_(f9,[h,j,k,'/l',p,d1]).
fn_(f9,['/h',i,k,'/l',x1]).
fn_(f9,['/h','/i',j,k,x1]).
fn_(f9,[j,k,'/l','/p',x1]).
fn_(f9,[a,p,d1]).
fn_(f9,[a,'/p',x1]).
fn_(f10,['/h','/i','/j','/k',e1]).
fn_(f10,[h,j,k,'/l',p,y1]).
fn_(f10,['/h',i,k,'/l',e1]).
fn_(f10,['/h','/i',j,k,e1]).
fn_(f10,[j,k,'/l','/p',e1]).
fn_(f10,[a,p,y1]).
fn_(f10,[a,'/p',e1]).
fn_(f11,['/h','/i','/j','/k',g1]).
fn_(f11,[h,j,k,'/l',p,f1]).
fn_(f11,['/h',i,k,'/l',g1]).
fn_(f11,['/h','/i',j,k,g1]).
fn_(f11,[j,k,'/l','/p',g1]).
fn_(f11,[a,p,f1]).
fn_(f11,[a,'/p',g1]).
fn_(f12,['/h','/i','/j','/k',i1]).
fn_(f12,[h,j,k,'/l',p,h1]).
fn_(f12,['/h',i,k,'/l',i1]).
fn_(f12,['/h','/i',j,k,i1]).
```

```
fn_(f12,[j,k,'/l','/p',i1]).
fn_(f12,[a,p,h1]).
fn_(f12,[a,'/p',i1]).
fns_([f0,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12]).
fnma_(f0).
fnma_(f2).
fnma_(f3).
fnma_(f4).
fnma_(f5).
fnma_(f6).
fnma_(f7).
fnma_(f8).
fnma_(f9).
fnma_(f10).
fnma_(f11).
fnma_(f12).


step 4: logopt -o -nor sample_opt


/* sample_out (default output file) */


(nor,f0,inv1)
(nor,inv1,gate34,var23,gate31,gate29,gate27,gate26,gate24,
 gate23,gate1)
(nor,gate34,/j,i,l,/k,/d,var10,/h,/c,/b)
(nor,/c,c)
(nor,/b,b)
(nor,/h,h)
(nor,/d,d)
(nor,var10,e,f)
(nor,/k,k)
(nor,/j,j)
(nor,var23,/j,i,/k,h,p,/l)
(nor,/l,l)
(nor,gate31,l,/k,h,/i,/g)
(nor,/g,g)
(nor,/i,i)
(nor,gate29,/j,i,l,/k,var6,/g)
(nor,var6,var10,/d)
(nor,gate27,k,j,i,h,/g)
(nor,gate26,/d,var10,/a,/c,/b)
(nor,/a,a)
(nor,gate24,l,/k,h,/g,/j)
(nor,gate23,l,/k,/j,/i,/h,var12)
(nor,var12,gate51,gate50)
(nor,gate51,/m,var11)
(nor,/m,m)
```

```
(nor, var11, gate49, gate48)
(nor, gate49, /n, d, var10)
(nor, /n, n)
(nor, gate48, f, e, /d)
(nor, gate50, /o, /d, var10)
(nor, /o, o)
(nor, gate1, /g, /a, var6)
(nor, f2, inv12)
(nor, inv12, nor22, nor21, gate35, gate33, gate30, gate12)
(nor, gate35, /j, i, l, /k, /d, var10, /h, /o)
(nor, gate33, /j, i, /k, h, p, /l, var12)
(nor, gate30, /j, i, l, /k, var6, /q)
(nor, /q, q)
(nor, gate12, /a, var12)
(nor, nor21, i, h, /q)
(nor, nor22, l, /k, h, /q)
(nor, f3, nor1, nor2)
(nor, nor1, gate21, r)
(nor, gate21, /p, var2, /s)
(nor, /s, s)
(nor, /p, p)
(nor, var2, a, gate38)
(nor, gate38, l, /k, /j, /h)
(nor, nor2, gate21, gate45, gate44)
(nor, gate45, var7, var3)
(nor, var7, var10, /d, gate43)
(nor, gate43, p, var5)
(nor, var5, /b, /c)
(nor, var3, a, gate39)
(nor, gate39, l, /k, /j)
(nor, gate44, h, var4)
(nor, var4, gate41, gate40)
(nor, gate41, i, var1)
(nor, var1, gate37, gate36)
(nor, gate37, /k, /j)
(nor, gate36, k, j)
(nor, gate40, l, /k, /i)
(nor, f4, nor3, nor4)
(nor, nor3, gate22, t)
(nor, gate22, /p, var2, /u)
(nor, /u, u)
(nor, nor4, gate22, gate45, gate44)
(nor, f5, nor5, nor6)
(nor, nor5, gate20, v)
(nor, gate20, /p, var2, /w)
(nor, /w, w)
(nor, nor6, gate20, gate45, gate44)
(nor, f6, nor7, nor8)
(nor, nor7, gate19, x)
(nor, gate19, /p, var2, /y)
(nor, /y, y)
(nor, nor8, gate19, gate45, gate44)
```

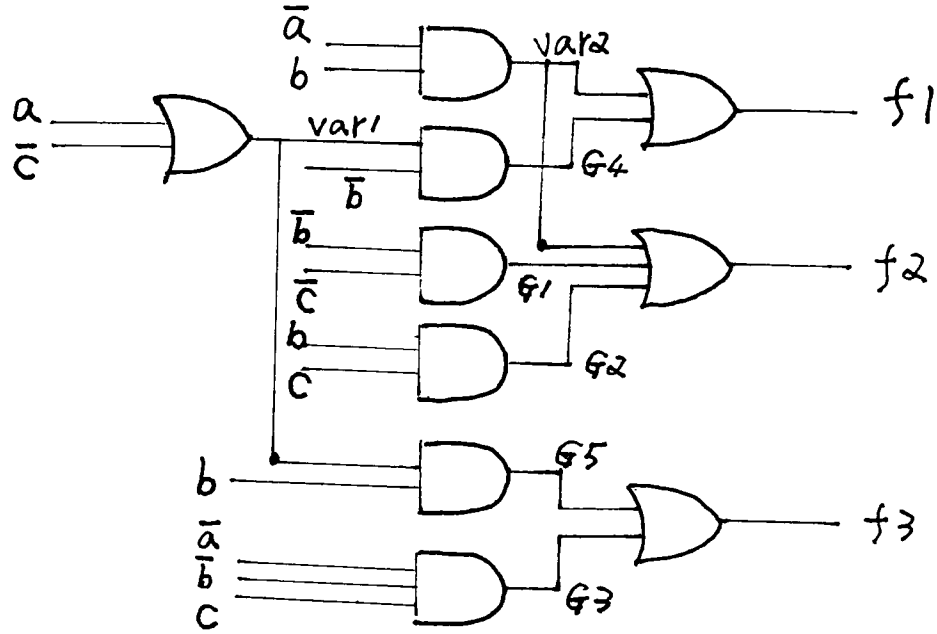
```
(nor, f7, nor9, nor10)
(nor, nor9, gate18, z)
(nor, gate18, /p, var2, /a1)
(nor, /a1, a1)
(nor, nor10, gate18, gate45, gate44)
(nor, f8, nor11, nor12)
(nor, nor11, gate17, b1)
(nor, gate17, /p, var2, /c1)
(nor, /c1, c1)
(nor, nor12, gate17, gate45, gate44)
(nor, f9, nor13, nor14)
(nor, nor13, gate16, x1)
(nor, gate16, /p, var2, /d1)
(nor, /d1, d1)
(nor, nor14, gate44, gate16, gate46)
(nor, gate46, p, var3)
(nor, f10, nor15, nor16)
(nor, nor15, gate15, e1)
(nor, gate15, /p, var2, /y1)
(nor, /y1, y1)
(nor, nor16, gate44, gate15, gate46)
(nor, f11, nor17, nor18)
(nor, nor17, gate14, g1)
(nor, gate14, /p, var2, /f1)
(nor, /f1, f1)
(nor, nor18, gate44, gate14, gate46)
(nor, f12, nor19, nor20)
(nor, nor19, gate13, i1)
(nor, gate13, /p, var2, /h1)
(nor, /h1, h1)
(nor, nor20, gate44, gate13, gate46)
```

Example 2

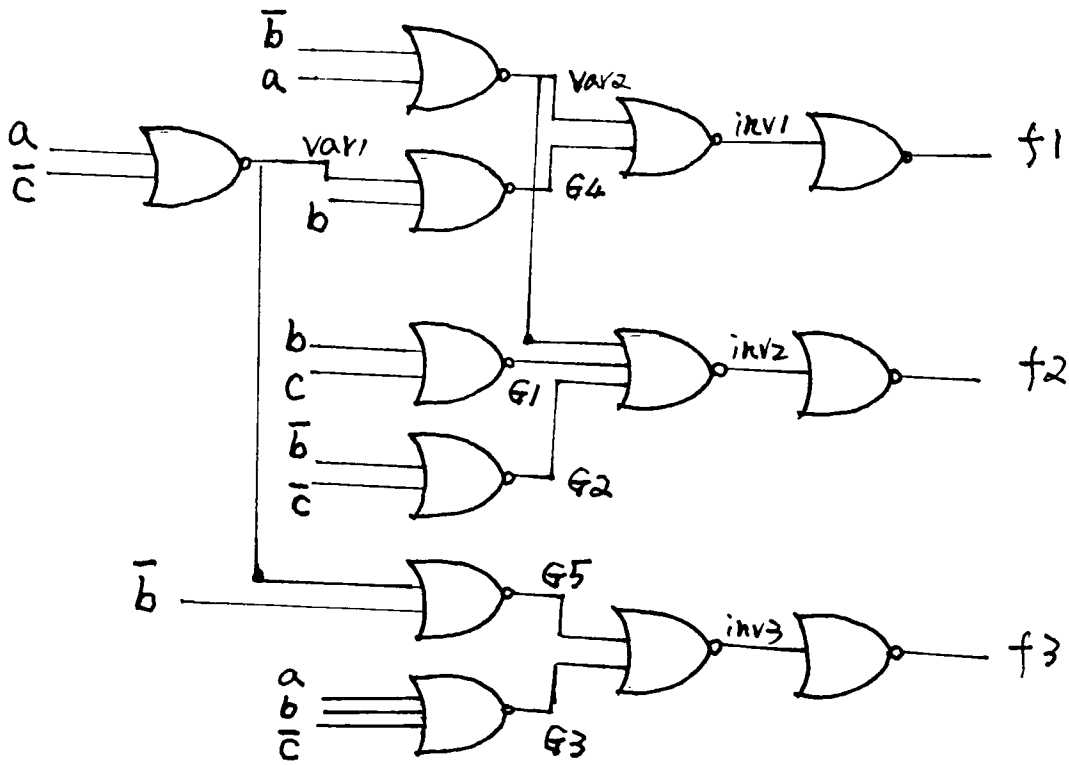
```
/* The following data is for logic synthesis diagrams */

gate_(invert,'/b',[b]).
gate_(in,b,[b]).
gate_(invert,'/c',[c]).
gate_(in,c,[c]).
gate_(and,gate1,['/b','/c']).
gate_(and,gate2,[b,c]).
gate_(invert,'/a',[a]).
gate_(in,a,[a]).
gate_(and,gate3,['/a','/b',c]).
gate_(and,gate4,[var1,'/b']).
gate_(and,gate5,[var1,b]).
gate_(or,f2,[var2,gate2,gate1]).
gate_(or,f3,[gate5,gate3]).
gate_(or,f1,[var2,gate4]).
gate_(or,var1,[a,'/c']).
gate_(and,var2,['/a',b]).
```

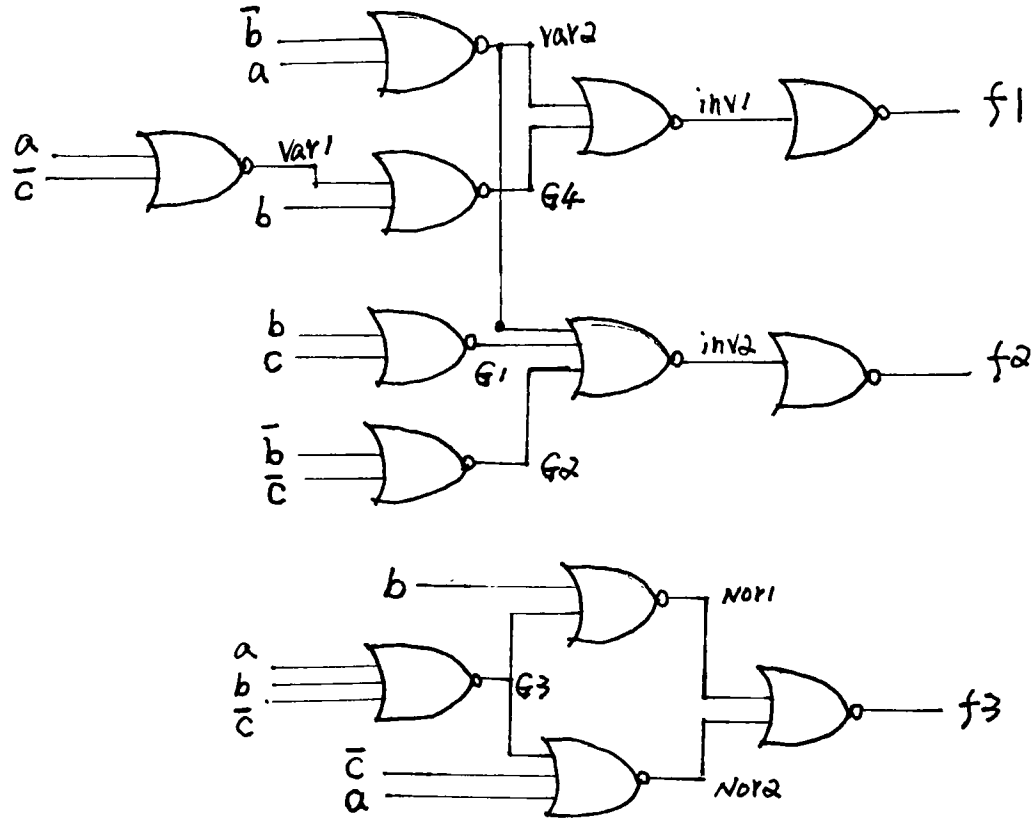
AND-OR SYNTHESIS



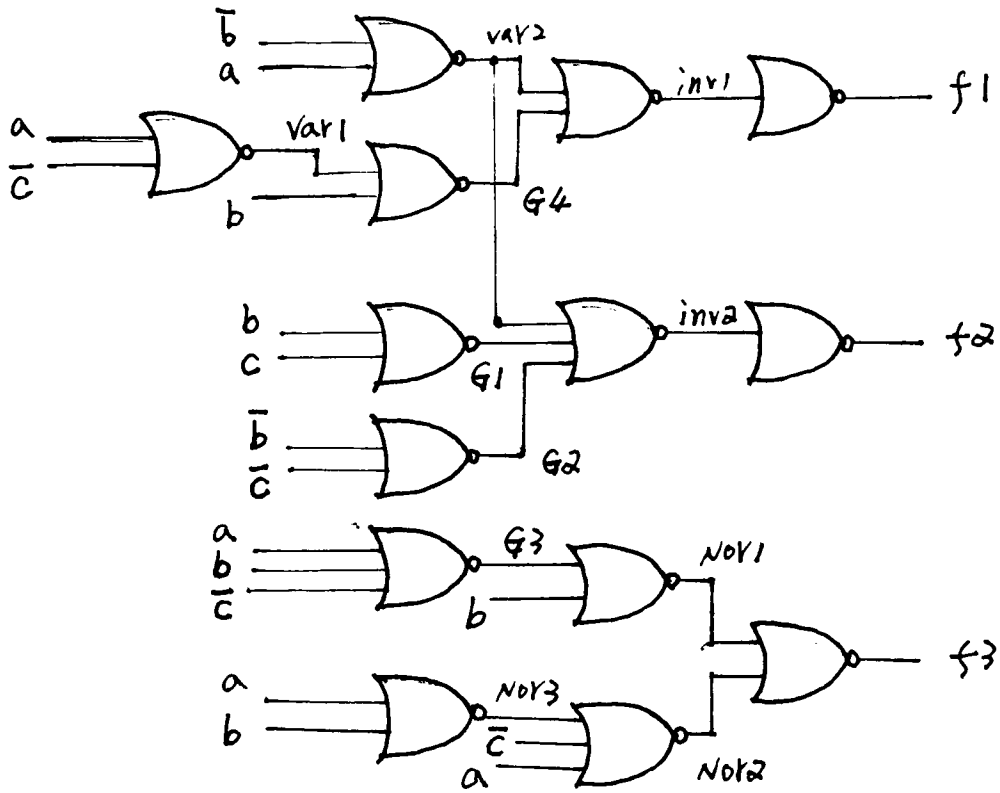
NOR SYNTHESIS



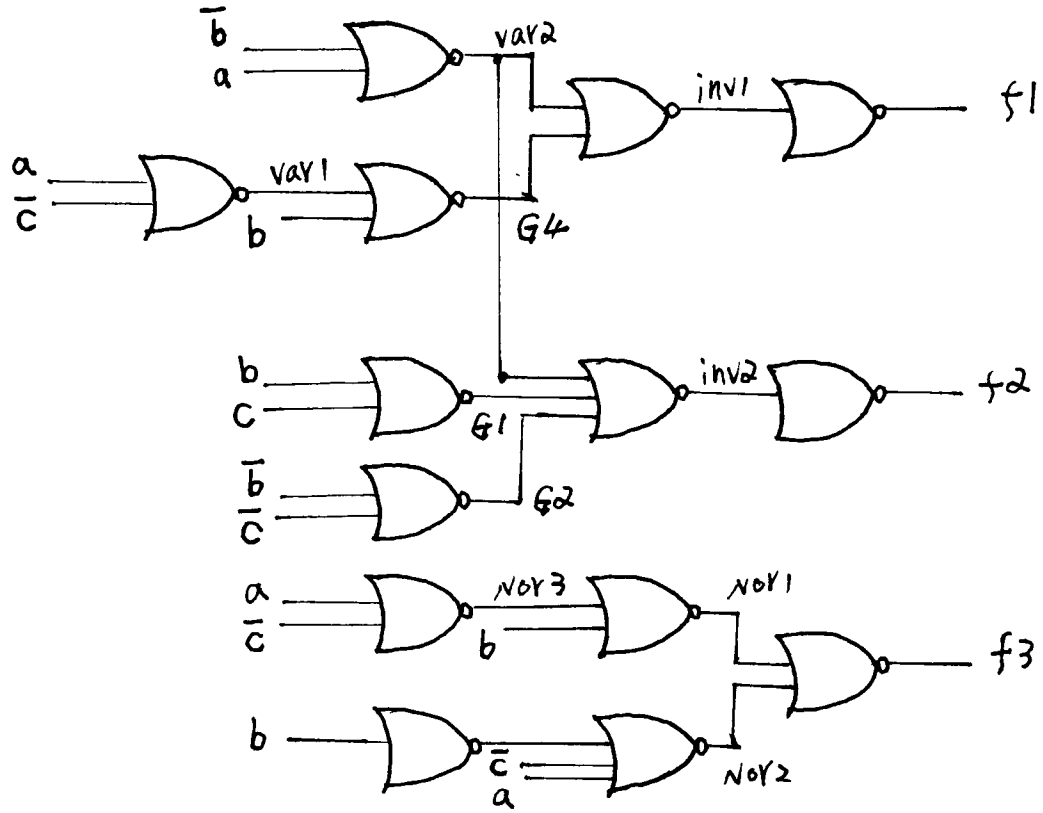
Apply Rule 2



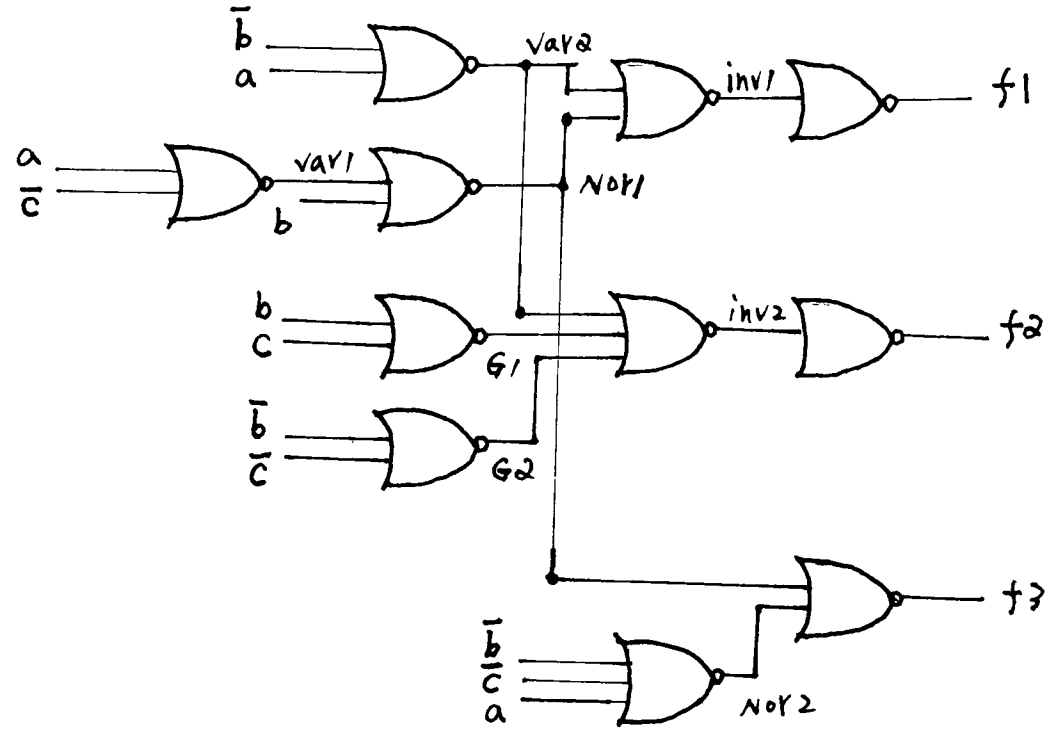
Apply Rule 6



Apply Rule 8



Apply Rule 3



Appendix C. Rule Base

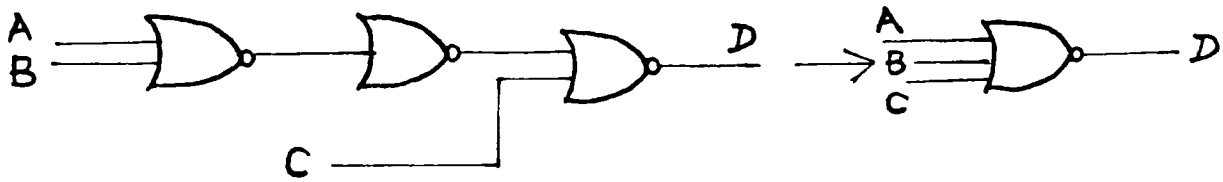


Figure C.1 Nor_Rule 1

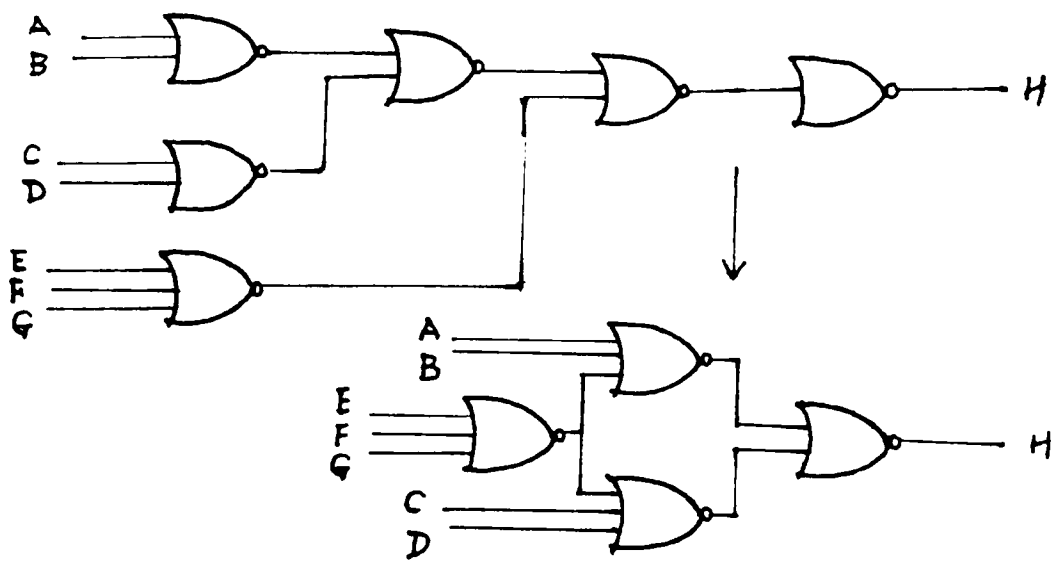


Figure C.2 Nor_Rule 2

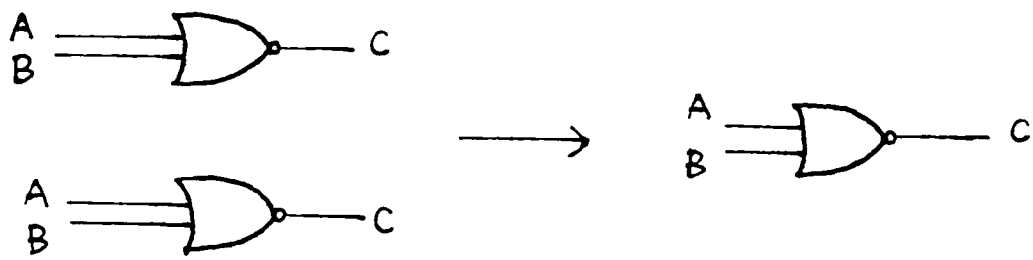


Figure C.3 Nor_Rule 3

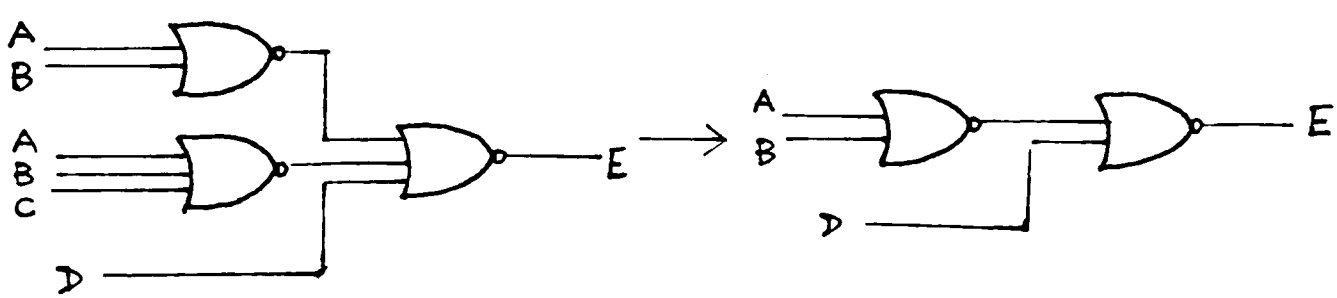


Figure C.4 Nor_Rule 4

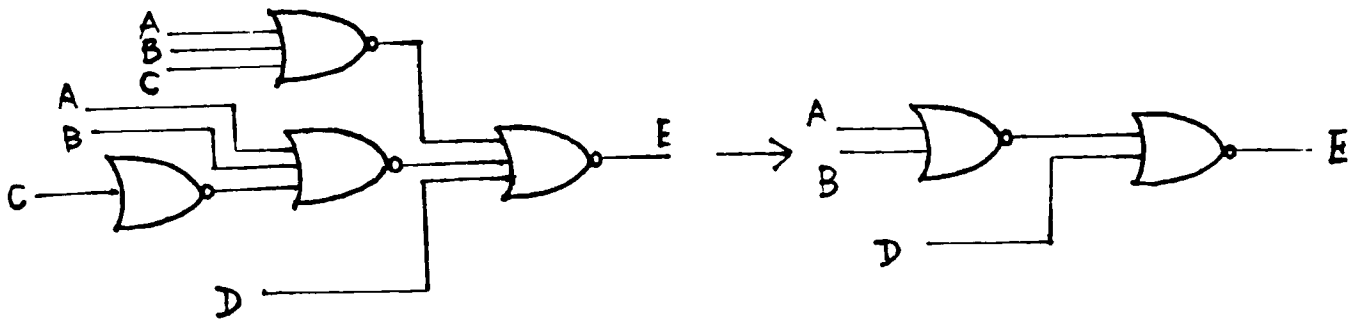


Figure C.5 Nor_Rule 5

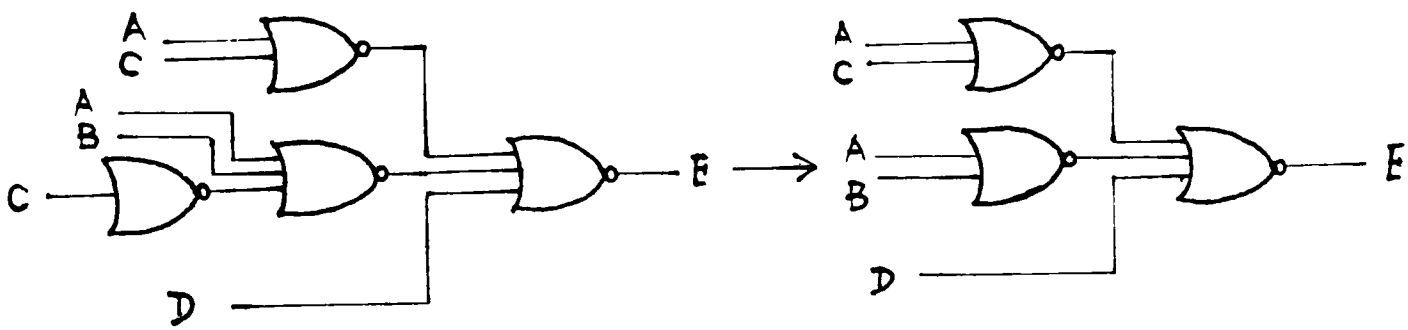


Figure C.6 Nor_Rule 6

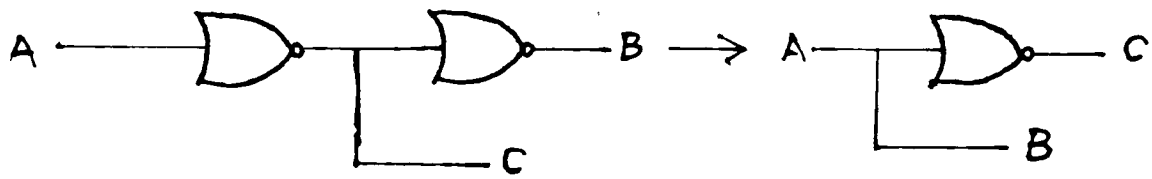


Figure C.7 Nor_Rule 7

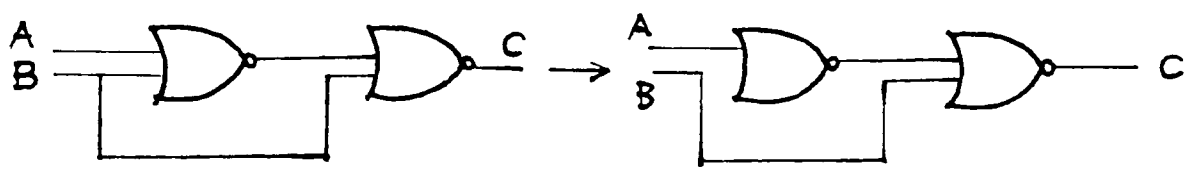


Figure C.8 Nor_Rule 8

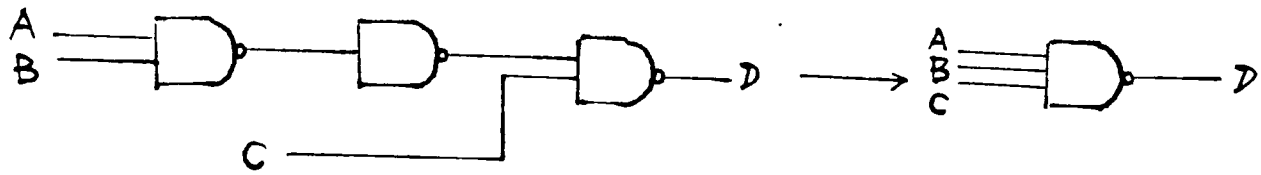


Figure C.9 Nand_Rule 1

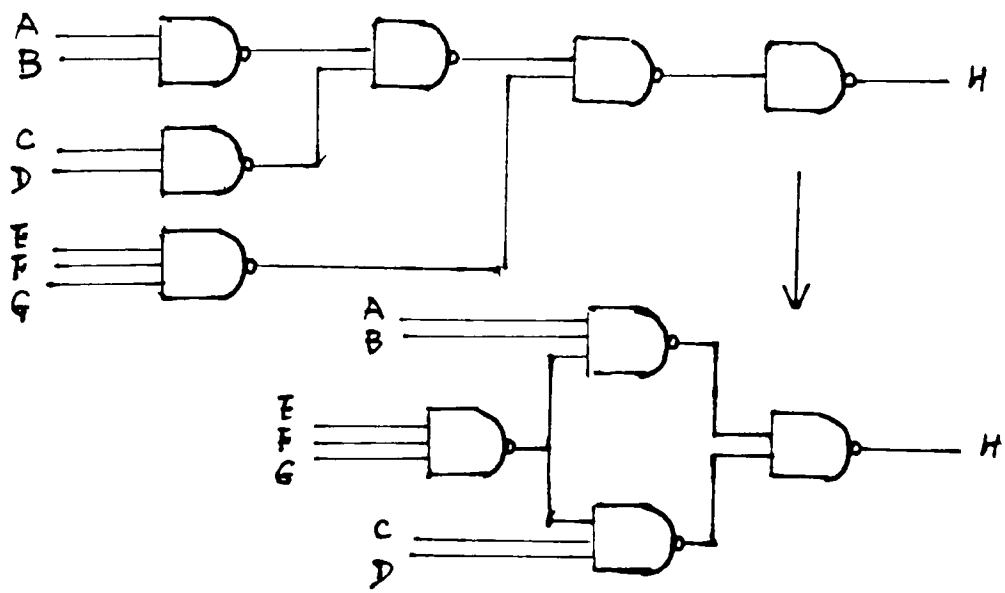


Figure C.10 Nand_Rule 2

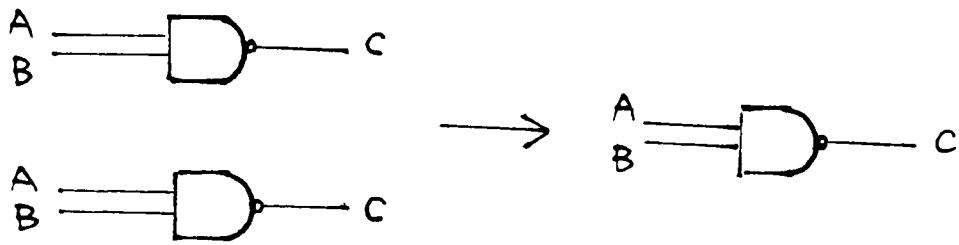


Figure C.11 Nand_Rule 3

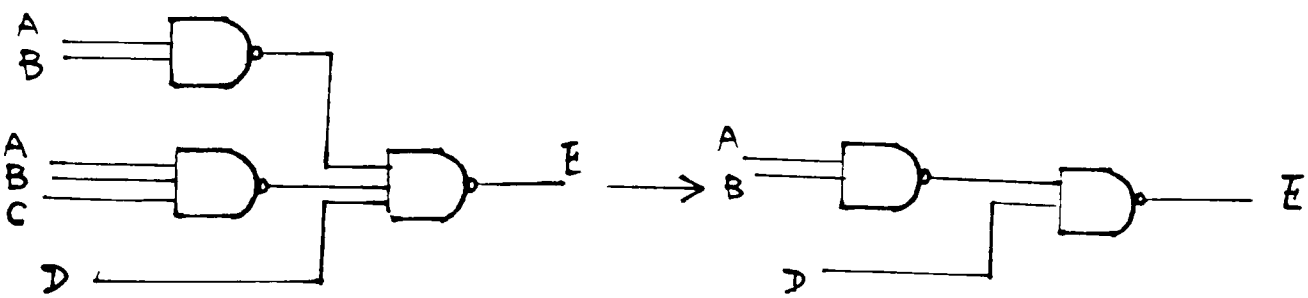


Figure C.12 Nand_Rule 4

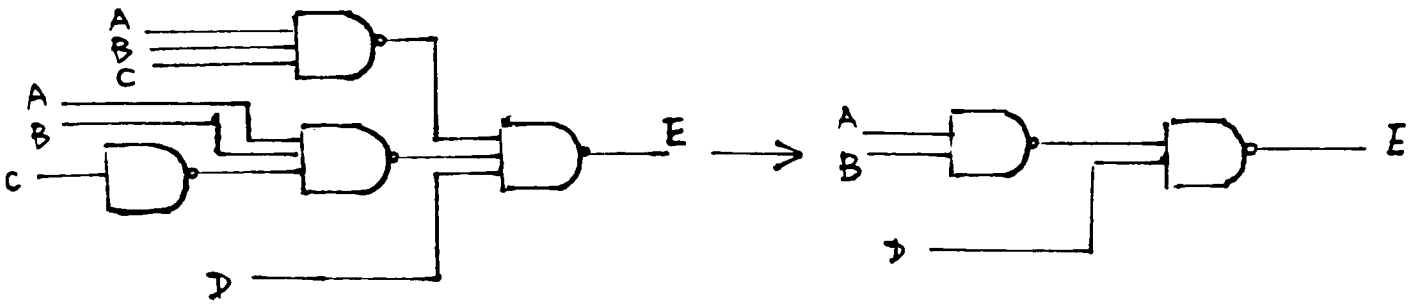


Figure C.13 Nand_Rule 5

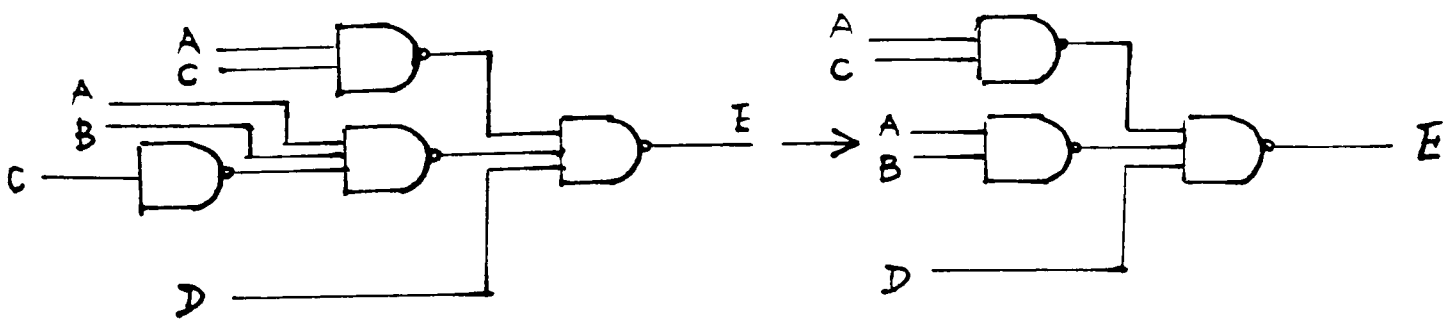


Figure C.14 Nand_Rule 6

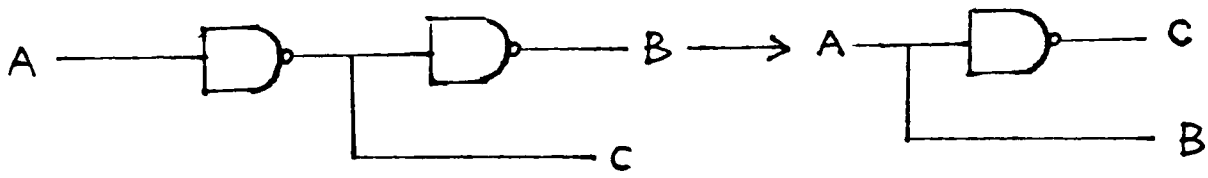


Figure C.15 Nand_Rule 7

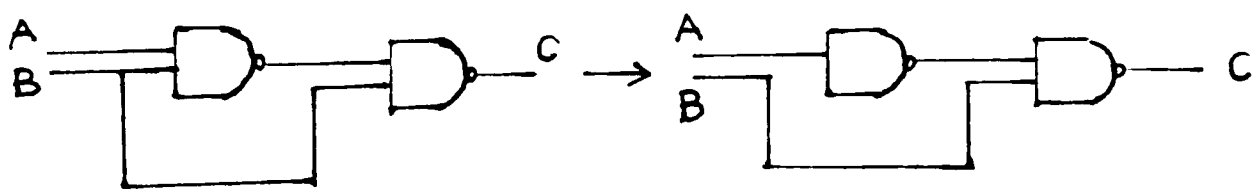


Figure C.16 Nand_Rule 8

BIBLIOGRAPHY

1. R.K. Brayton, et al., ESPRESSO-IIC: Logic Minimization Algorithms for VLSI synthesis, Kluwer Academic Publishers, Netherlands, 1984.

2. C.L. Forgy, OPS5 User's Manual, technical report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

3. W.F. Clocksin and C.S. Mellish, Programming in Prolog, Springer-Verlag, New York, NY, 1981.

4. J.A. Darringer, et al., Experiments in Logic Synthesis, Proc. IEEE Int'l Conf. Circuits and Computers, 1980.

5. J.A. Darringer, et al., LSS: A System for Production Logic Synthesis, IBM J. Research and Development, Vol. 28, NO. 5, Sep. 1984.

6. R.K. Brayton and C. McMullen, The Decomposition and Factorization of Boolean Expressions, Proc. Int'l Symp. Circuits and Systems, 1982.

7. J. McDermott, R1: A Rule-Based Configurer of Computer Systems, technical report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

8. F. Hayes-Roth, Knowledge-Based Expert Systems, IEEE Computer, October, 1984.

9. W.P. Birmignham and J.H. Kim, DAS/Logic: A Rule-Based Logic Design Assistant, IEEE The Second Conf. on Artificial Intelligence Application, 1985.

10. L.I. Steinberg and T.M. Mitchell, A Knowledge Based Approach to VLSI CAD The Redesign System, The 21st Design Automation Conference, 1984.

11. W.W. Cohen, K.A. Bartlett, and A.J. DeGeus, Impact of Meta-Rules in a Rule-Based Expert System for Gate Level Optimization, Proc. Int'l IEEE Conf. on Computer Design, Oct. 1984.

12. A.J. Degeus and W.W. Cohen, A Rule Based System for Optimizing Combinational Logic, IEEE Design and Test of Computers, August, 1985.

13. R.K. Brayton and C.T. McMullen, Synthesis and Optimization of Multistage Logic, Proc. Int'l Conf. on Computer Design, Oct. 1984.

14. M.A. Breuer, Design Automation of Digital Systems, Prentice-Hall, 1972.

15. D.L. Dietmeyer, Logic Design of Digital Systems, Allyn and Bacon, 1971, 1978.

16. T.D. Friedman, S.C. Yang, Quality of Design From an Automatic Logic Generator (ALERT), Proc, IEEE Design Automation Conference, 1970.

17. G. Hachtel, R.K. Bartlett, W.W. Cohen and A.J. DeGeus, Synthesis and Optimization of Multi-level Logic Under Timing Constraints, Proc. Int'l IEEE Computer Aided Design, 1985.

18. D. Gregory, R.K. Bartlett, and A.J. deGeus, Automatic Generation of Combinatorial Logic from a Functional Specification, Proc. Int'l Symp. on Circuits aand Systems, May 1984.

19. J.A. Darringer, et al., Logic Synthesis Through Local Transformation, IBM J. of Research and Development, Vol. 25, July 1981.

20. K. Garrison, et al., Automatic Area and Performance Optimization of Combinatorial Logic, Proc. Int'l IEEE Computer Aided Design, 1984.

21. M.M. Mano, Digital Logic and Computer Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

22. D.A. Waterman, A Guide to Expert Systems, Addison-Wesley Publishing Co., 1986.

23. F. Hayes-Roth, D.A. Waterman and D.B. Lenat, Building Expert System, Addison-Wesley Publishing Co., 1983.

24. A. Barr and E.A. Feigenbaum, The Handbook of Artificial Intelligence, Vol. 1, William Kaufmann, Inc., 1981.

25. W. Myers, Introduction to Experts Sytems, IEEE Experts, Spring, 1986.

26. T. Uehara, A Knowledge-Based Logic Design System, IEEE Design and Test, Oct 1985.

27 R. K. Brayton, et al., Fast Recursive Boolean Function Manipulation, Proceedings ISCAS, Rome, May 1982.