THE DESIGN OF A HYBRID MICROPROCESSOR/ BINARY DECISION PROGRAMMABLE CONTROLLER

by

Robert Douglas Hudson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Master of Engineering

Department of Mechanical Engineering

McGill University

Montreal, Canada

October, 1984

ABSTRACT

THE DESIGN OF A HYBRID MICROPROCESSOR/ BINARY DECISION PROGRAMMABLE CONTROLLER

The design and implementation of a hybrid microprocessor/binary decision (mP/BD) programmable controller is presented in terms of both hardware and software. This hybrid configuration enables ON/OFF control tasks to be realized, in linear time, by the BD processor, while modulating control tasks are performed conventionally in the mP. In addition, mP-assisted BD program and hardware management, and operator-to-BD communication are made possible. A prototype programmable controller, consisting of two stand-alone processors, was constructed. An operating system, BD09, was developed which supports all process tasks and internal system functions. As well, a mP-based optimizing compiler was written to translate high level logic descriptions into executable BD machine code.

The theory of binary decision processing is presented and BD computational characteristics are analyzed. A new algorithm for the reduction of BD logic is also described. The performance of the hybrid controller is evaluated and compared to that of a boolean, single-bit, Industrial Control Unit through two application examples. A substantial improvement in time and space complexity is demonstrated.

RESUME

CONCEPTION D'UN CONTROLEUR PROGRAMMABLE HYBRIDE COMPOSE D'UN MICROPROCESSEUR ET D'UNE MACHINE DE DECISION BINAIRE

La conception et la réalisation d'un contrôleur programmable hybride, composé d'un microprocesseur et d'une machine de décision binaire (mP/BD) sont décrits en termes de matériel ainsi que de logiciel. Cette configuration hybride permet au processeur BD d'effectuer en temps linéaire les tâches de contrôle logique et au même moment, le mP exécute les tâches de contrôle proportionnel. En plus, ce système permet: 1) une gestion du programme et du matériel BD par le mP et aussi, 2) une communication entre l'opérateur et la machine BD. Un prototype de contrôleur programmable, comportant deux processeurs indépendants, a été construit. Un système d'exploitation BD09, qui effectue toutes les tâches de processus industriel ainsi que les fonctions inhérentes au système, a été développé. Finalement, un compilateur optimisant qui traduit des systèmes logiques combinatoires sous forme compréhensible par la machine BD, a été également développé.

La théorie et les caractéristiques du processus de décision binaire sont présentées et analysées. Un nouvel algorithme pour la réduction de la logique BD est aussi décrit. La performance du contrôleur hybride est évaluée et comparée à celle d'une Unité de Contrôle Industrielle (MC14500) pour deux applications typiques. Une amélioration substantielle de la complexité dans le temps et dans l'espace est démontrée lorsque la technique développée est utilisée.

ACKNOWLEDGEMENTS

The work presented in this thesis was carried out under the supervision of Dr. Paul J. Zsombor-Murray and Mr. Louis J. Vroomen. The author wishes to express his deepest gratitude to them for the guidance and encouragement received during the course of this study. The work was supported by grant A 4219 of the Natural Sciences and Engineering Research Council of Canada.

Special thanks are also due to Mr. Meir Levi and Mr. Artun C. Kucuk who, in addition to being good friends, contributed materially to the success of this project. Finally, the author wishes to thank his wife, Anne Sage-Hudson, for her love and support, and for her cheerful and careful typing of the manuscript, which was prepared on the word-processing facilities of the McGill Cancer Centre.

LIST OF ABBREVIATIONS

ACIA	Asynchronous Communications Interface Adaptor
BD	Binary Decision
CLK	Clock
CLR	Clear
DAG	Directed Acyclic Graph
DEMUX	Demultiplexor
EOP	End of Program
EPROM	Erasable Programmable Read Only Memory
EW	East/West
IC	Integrated Circuit
ICU	Industrial Control Unit
IN	Instruction
I/O	Input/Output
IRI	Intelligent Reflexive Interface
ISP	Instruction Set Processor
IV	Input Variable Register
LCU	Logic Control Unit
LED	Light Emitting Diode
LMU	Library Management Unit
MMU	Memory Management Unit
mP	Microprocessor
Mp	Program Memory
MUX	Multiplexor
NS	North/South
OB	Output Bank Register
OP	Operation Code
OR	Output Register
O/S	Operating System
OV	Output Variables
PC	Program Counter
PCU	Programmable Control Unit
PIA	Peripheral Interface Adaptor
PID	Proportional Integral and Derivative
PLA	Programmable Logic Array
PLC	Programmable Logic Controller
PMA	Pattern Matching Algorithm
PROM	Programmable Read Only Memory
PSU	Program Scheduler Unit
ROM	Read Only Memory
R/S	Run/Stop

LIST OF ABBREVIATIONS (cont'd)

RTX	Real-Time Executive
R/W	Read/Write
SPST	Single-Pole/Single-Throw Switch
SS	Single Step
TTL	Transistor-Transistor-Logic
VLSI	Very Large Scale Integration
XOR	Exclusive-OR

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

LIST OF FIGURES AND TABLES

CHAPTER 1 - INTRODUCTION

CHAPTER 2 - BINARY DECISION THEORY

LIST OF FIGURES AND TABLES (cont'd)

CHAPTER 3 - THE DESIGN OF THE HYBRID mP/BD SYSTEM

Page

Figures

Tables

CHAPTER 4 - mP/BD HYBRID PLC OPERATING SYSTEM DESIGN

Figures

Tables

LIST OF FIGURES AND TABLES (cont'd)

CHAPTER 5 - BINARY DECISION PROGRAM OPTIMIZATION **Page**

Figures

Tables

CHAPTER 6 - MP/BD PLC APPLICATIONS

Figures

Tables

LIST OF FIGURES AND TABLES (cont'd)

CHAPTER 6 (cont'd) **Page**

CHAPTER 1

INTRODUCTION

1.0 General Introduction

Industrial process control can be regarded as a means to produce a set of responses to a set of stimuli, i.e., to generate control actions in response to measured process variables. The procedures by which the control actions are produced are often divided into two categories; logical and analog. Logical procedures are satisfactorily described by a few binary digits, or bits of information, which are processed by simple digital logic circuits. Analog control procedures are best described by real numbers or, at least, by many bits. They are processed by analog circuits or by digital processors capable of fairly precise arithmetic.

A "hardwired" controller is composed of the minimum circuitry necessary to serve a specified process. This traditional approach tends to produce the fastest, physically smallest and most energy efficient implementation of a control procedure. However, except in circumstances in which these attributes are exigent, modern industrial controllers are now designed around programmable automata which offer the advantage of standardized hardware components with the flexibility to accommodate virtually any control procedure.

As a result distributed microprocessor (mP) networks are gradually replacing conventional hardwired analog equipment, i.e., of the second category, for continuous control applications, while programmable logic

controllers (PLC) are replacing discrete component gate and relay logic, i.e., of the first category, for ON/OFF control. This thesis presents the design of such a PLC which is based in part on a "binary decision" programmable automaton.

Most contemporary PLC designs use table-lookup-based PLA/ROM architectures and finite-state machine or microprocessor architectures which operate on encoded Boolean sum-of-products expressions. Unfortunately, many program steps are required to evaluate all but the most trivial Boolean switching functions. In fact, the number of steps increases exponentially with the number of binary input variables needed in the computation. As well this approach is limited to the serial generation of single, binary-valued control functions.

For these reasons, switching function design methods, more efficient than Boolean equations, were sought and in 1959 Lee devised an improved technique which he called binary decision (BD) programming [Lee59]. He demonstrated that this approach always evaluates switching functions in a number of steps equal to or less than the number of input variables. The binary decision approach can also generate multiple, parallel-output switching functions. However, the practical implementation of binary decision programming to the design of programmable sequence controllers awaited the design of suitable hardware, a formal logic representation and the development ot minimization methods.

Unlike Boolean logic, the mathematical basis of binary decision logic was not immediately evident. Since it did not seem amenable to algebraic manipulation, Lee thought that formal logic reduction

procedures such as Karnaugh mapping did not exist. BD program simplification was a cut-and-try effort which relied on the intuition and experience of the programmer.

The research presented in this thesis describes the design of a programmable controller system which includes a prototype binary decision-based sequential automaton for the execution of ON/OFF process control functions. This controller is integrated into a conventional microprocessor system so that complex control functions involving both ON/OFF and proportional tasks may be performed concurrently and efficiently. To exploit program optimization or to adapt the BD control logic to changing process conditions, the microprocessor also acts as a supervisor to the BD machine. Updated control routines are transferred by the mP to the BD program memory in the same manner as it stores data into its own memory.

A mathematical basis for binary decision logic, developed from Boolean algebra, is also presented. It is used to analyze the computational characteristics of the binary decision process. Formal logic minimization procedures are developed. A functional BD program compiler is described which accepts switching functions presented in truth table form. This compiler reduces them to a binary decision structure and translates this logic into BD machine language.

Finally, a real-time process control operating system is described which performs job scheduling, inter-processor and operator communication functions and program library/system memory management tasks. The program supports multi-programmed BD processes and maintains a BD program library in disk storage.

1.1 Introduction to Programmable Logic Controllers

A programmable logic controller is an industrial process control device used to implement ON/OFF control functions, e.g., in chemical batch processing, electromechanical machinery and motor control. ON/OFF control is characterized by signal type, i.e., a binary digital signal, and by control function type, i.e., combinatorial or sequential switching functions which do not require arithmetic processing.

Programmable logic controllers have replaced traditional hardwired relay and gate logic circuits in many applications because of the high cost of design, fabrication, testing, installation, startup and maintenance of discrete component circuits. In addition program modification is a fairly simple task with stored program automata whereas it often requires major redesign of a hardwired system. In other words, hardwired logic is inflexible and expensive while programmable logic is very flexible and economical. Although many types of programmable logic controllers exist, they all consist of a basic control circuit and a program memory. The control unit is usually one of the following:

1) Programmable Logic Array (PLA). A PLA is an integrated circuit containing an array of semiconductor logic gates in which inputs and outputs are connected by a cross-bar matrix of a semiconductor Read Only Memory (ROM). Any gate logic circuit representation of a Boolean sum-of-products expression can be implemented simply by programming the required erasure pattern of cross-point connections. This is done during manufacture (ROM) or in the field (Programmable Read Only Memory

- PROM) leaving only those connections which define the logic. The connections may be conductive links, diodes, transistors or invertors [Korn77].

Two gate matrices are contained in the PLA logic gate array. One logically ANDs inputs together to form minterms or implicants of the function while the second ORs these minterms together to define complete switching functions. Up to 2^n parallel functions of the n input variables can be programmed in this way. The method is quite economical since gating for unused input combinations need not be provided. A PLA is illustrated in figure 1.1.

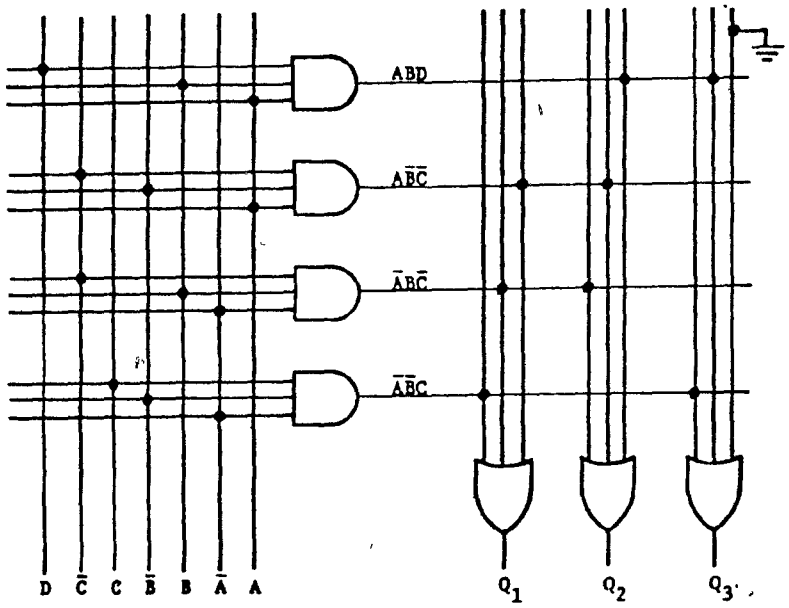
A PLA-based controller is indicated when the control functions are strictly combinatorial functions of a relatively small number of Boolean variables. As ROM access times vary between 35 and 1500 nano-seconds, outputs are available in periods comparable to those of high density circuit gate logic delays. Although multiple functions are computed in parallel, these functions are limited to single-bit output. As well, this approach cannot accommodate sequential logic or outputs of timed duration.

2) Finite-State Sequential Automata. A finite-state machine is an abstract model of a sequential switching function. A sequential circuit differs from a combinatorial circuit in that the outputs are contingent upon both the current inputs and the particular sequence of previous inputs to the circuit, represented by internal circuit states. Examples of finite-state sequence controller applications include vending machines, arcade games and traffic control lights [Koha70].

Finite-state controllers are modelled as shown in figure 1.2. The next internal state of the machine is computed by a combinatorial function of current inputs and current state. The output of this function is introduced to a memory device, often implemented as a latch or a D-type flip-flop, which retains the current state until the new state is "latched in" or stored, either by a synchronizing pulse from an external clock or by a change of inputs. Switching function outputs are generated by a second combinatorial function of external inputs and by the present state. These combinatorial circuits are frequently implemented as PLAs or may be stored in ROM as a set of program instructions sequentially executed under the control of a program counter. Figure 1.3 illustrates sequence controllers based on a) PLA logic and b) stored program logic.

State machine based programmable controllers are used in low-level control applications because their mass-produced hardware is fairly economical and the control logic is not usually altered during the lifetime of the product. Standard off-the-shelf components are used to design hardware. This limits the development cost to that of designing the control sequence itself.

3) Single or Multi-bit Microprocessors. Combinatorial, sequential and even proportional control algorithms can be executed by single-or multi-bit microprocessors using their arithmetic and logic processing power. The incentive to use microprocessor-based PLCs was the simplification of peripheral interfacing to equipment such as printers and recorders for the generation of program documentation and status reports, the improvement of fault diagnosis capacity resident in the PLC and the



$$Q_1 = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}$$
$$Q_2 = \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABD$$
$$Q_3 = \bar{A}\bar{B}C + ABD$$

figure 1.1 Programmable Logic Array.

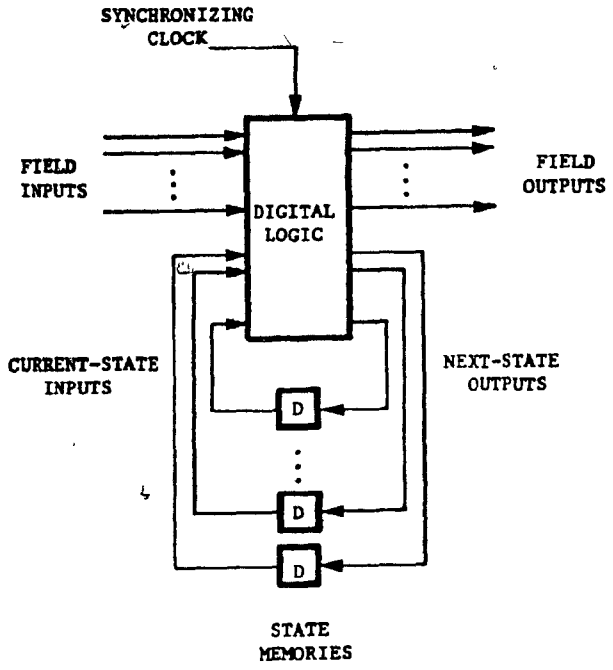
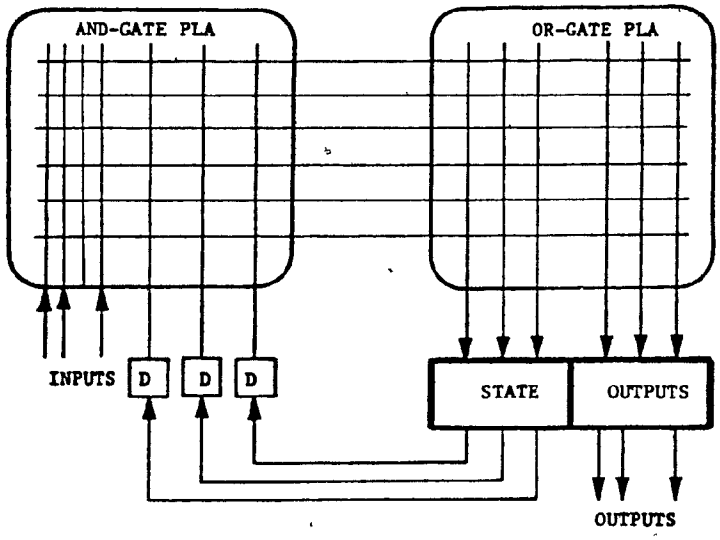
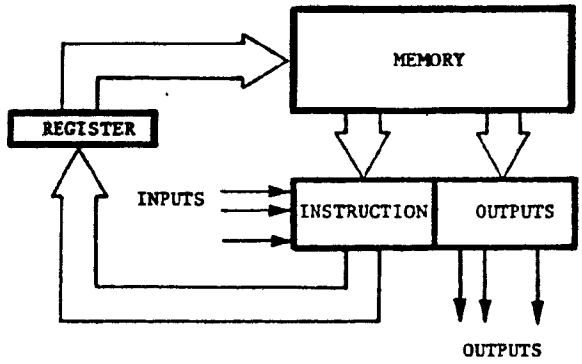


figure 1.2 Finite-state machine model.



(a)



(b)

figure 1.3 Finite-state sequence controller models.
(a) PLA-based logic and, (b) stored program logic.

exploitation of conventional programming languages to simplify the production of control programs from relay ladder diagram, gate logic diagram or control flowchart process descriptions. As a result many traditional computer programming languages such as FORTRAN and BASIC have also been adapted to include process control statements. Microprocessor hardware is well developed and inexpensive, and programming techniques are well understood.

However, in switching logic applications, the parallel architecture of common 8-bit or 16-bit microprocessors is often a liability. Programs that consist of encoded Boolean sum-of-products expressions describing the switching functions to be realized, examine single-bit variables serially and produce single-bit results. Most conventional microprocessor architectures cannot access single accumulator bits in a straightforward manner. Some indirect means of isolating individual bits such as bit-masking or shifting the accumulator contents into the carry-bit must be employed to emulate serial input or output. In addition to introducing this considerable program instruction overhead, these techniques make it difficult to permute the order in which variables are tested.

The Motorola MC14500 single-bit microprocessor, called an Industrial Control Unit (ICU), has received considerable attention for use in a programmable controller design [Taba81,Greg77]. The ICU system requires a multi-bit microprocessor-based development facility to compile machine code programs from gate logic or relay ladder representations. Furthermore the ICU evaluates functions in a purely serial manner. This lengthens the cycle time of the control algorithm

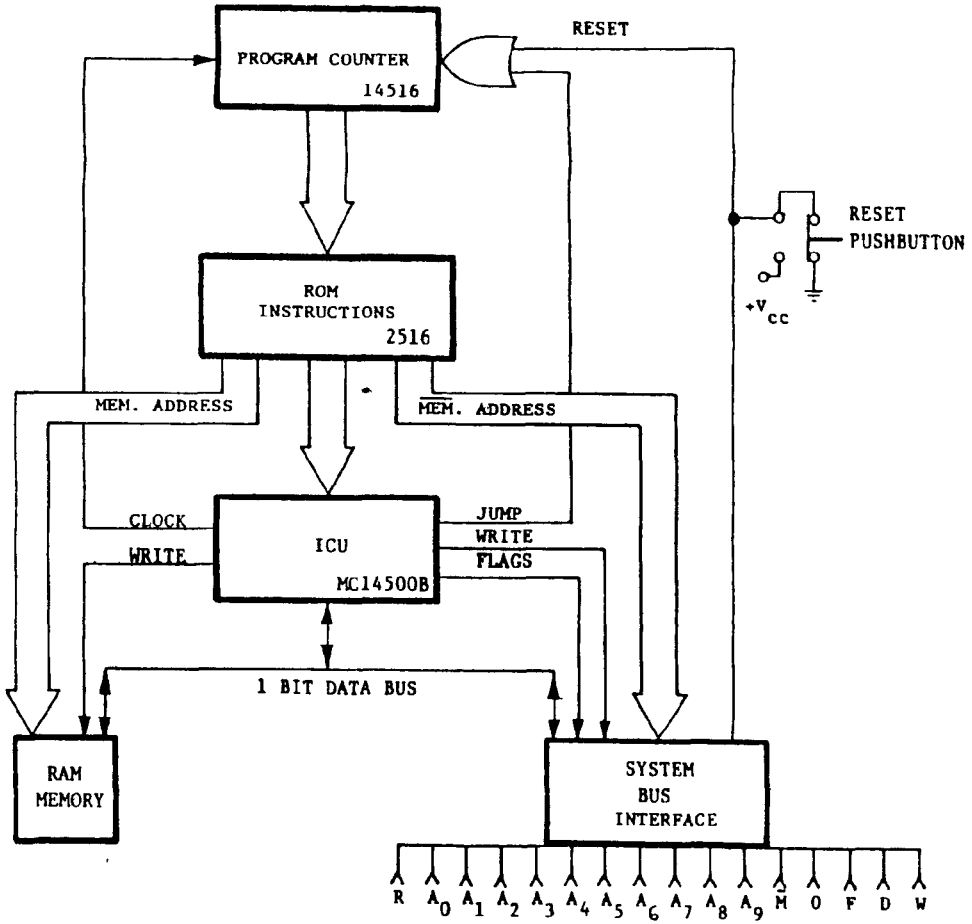
reducing the controller's effectiveness in fast time-constant processes. The serial evaluation of output variables may also cause incorrect control actions to take place if a group of output variables is required to appear simultaneously to define a certain process state, e.g., as input to a digital-to-analog converter. An ICU system is illustrated in figure 1.4 together with a sample program.

It will be shown in chapter 2 that a large number of program steps are required to evaluate all but the most trivial of Boolean expressions. This number increases exponentially with the number of input variables applied to the controller. In addition, microprocessor instructions frequently consist of many bytes of machine code and consume several clock cycles to execute. These factors combine to increase the overall cycle time of the control algorithm, an important parameter in the selection of programmable controller equipment in many industrial applications.

The characteristics of three main programmable controller architectures can be defined in terms of mode of operation, speed, cost, complexity and flexibility. The comparison summarized in table 1.1 was previously described by Tho et al [Tho79].

1.2 Introduction to Binary Decision-Based Programmable Controllers

1.2.1 Binary Decision Processors. A binary decision processor is a finite-state automaton that evaluates combinatorial or sequential logic represented as a binary decision program instead of a Boolean equation. Figure 1.5 illustrates a binary tree flowchart representing a typical BD



LD	A
OR	B
STO	MEMORY
LD	C
OR	D
AND	MEMORY
STO	LOAD

figure 1.4 MC14500-based Industrial Control Unit.

	RULES OF OPERATION	PROCESSING FORMAT	INPUT FORMAT	OUTPUT FORMAT	SPEED LIMITING FACTOR	HARDWARE COMPLEXITY	APPLICATION FLEXIBILITY
BINARY DECISION BASED PROGRAMMABLE CONTROLLER	BINARY DECISION PROGRAMS	SEQUENTIAL	SEQUENTIAL	PARALLEL	MEMORY ACCESS TIME	SIMPLE	SOFTWARE PROGRAM-MABLE
PROGRAMMABLE LOGIC ARRAY BASED CONTROLLER	BOOLEAN ALGEBRA	PARALLEL	PARALLEL	PARALLEL	INPUT/ OUTPUT LONGEST PATH PROPAGATION DELAY	INCREASES WITH CIRCUIT COMPLEXITY	HARDWARE PROGRAM-MABLE
MICROPROCESSOR BASED CONTROLLER	BOOLEAN ALGEBRA AND SEQUENTIAL CIRCUITS	SEQUENTIAL	SEQUENTIAL	SEQUENTIAL	PROCESSOR SPEED	SIMPLE	SOFTWARE PROGRAM-MABLE

table 1.1 Sequence controller comparisons.

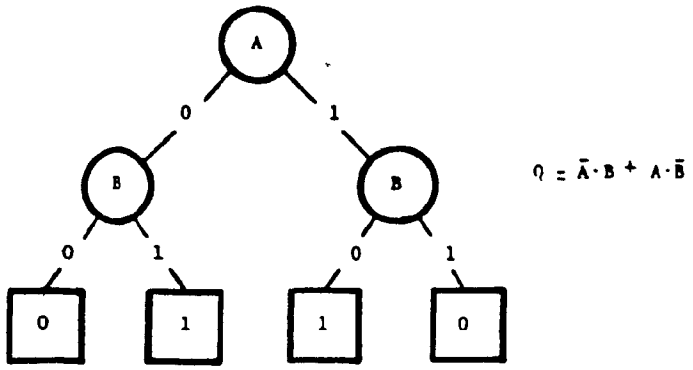


figure 1.5 BD representation of switching logic.

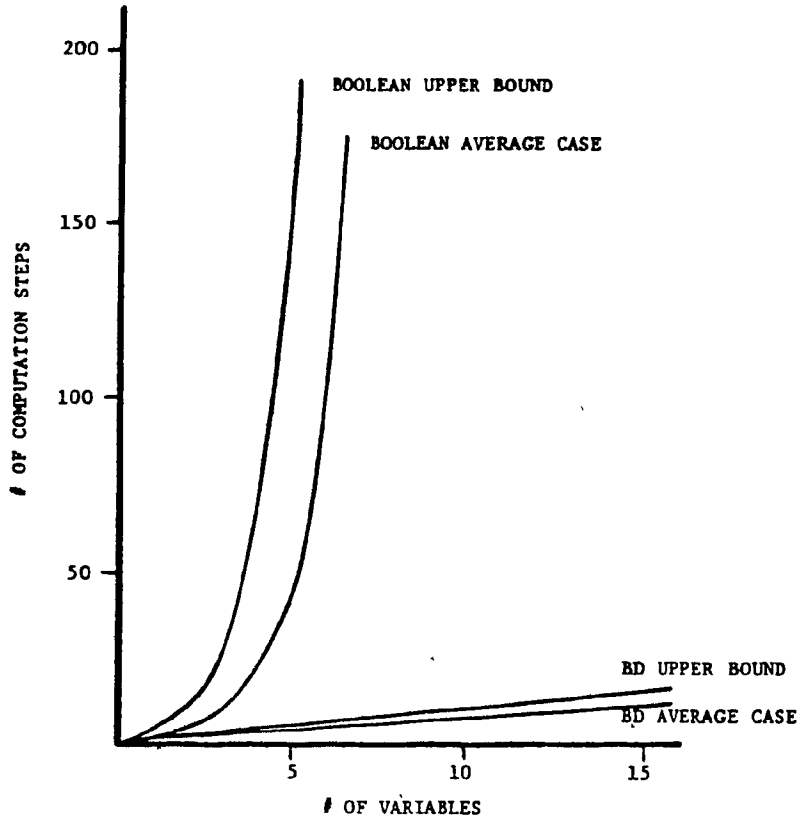


figure 1.6 Computational workload: BD vs. Boolean logic.

program along with the corresponding Boolean function. The BD machine executes the program by examining the variables associated with each instruction (shown as a Boolean literal within a node). The sampled logic levels provide branching criteria upon which the next instruction is selected. In the figure, the branching possibilities are shown as directed edges connecting pairs of nodes. Flow is from top to bottom. As can be seen each instruction connects with at most two other instructions and thus the processor must choose one out of two, a binary decision. This testing sequence leads the processor to a terminal or output instruction (shown as boxes) in which the appropriate control action has been tabulated.

Two factors make BD processors faster than other stored-program controllers. The first is that fewer variables are tested when evaluating switching functions. Boolean-based PLCs use equations encoded as sum-of-products expressions in which literals (variables) may appear several times. Every variable in an expression is tested in sequence to evaluate the complete function. The BD processor needs only to compute one path, from root to leaf, of the program, where each path is mathematically equivalent to a product-of-variables expression. Thus the binary decision processor tests fewer variables in order to evaluate a logic function.

The second factor is shorter variable testing time. A microprocessor-based controller requires from one to three machine instructions to read and test an input variable. Each instruction typically consumes from three to five clock cycles. Hence, an average of ten cycles may be required to test each variable. The BD processor

uses just one clock cycle to input a variable, test it, and branch to the next instruction. In fact several complete functions can be evaluated in only n cycles, where n is the number of independent variables of the function.

The difference between Boolean and BD program instruction sequence length is illustrated in figure 1.6. The Boolean upper bound is estimated from the worst-case situation in which all 2^n minterms are significant and each minterm subsumes all n literals (a degenerate case, since then the function would always be implied). In this case the Boolean-based controller would execute $n2^n$ instructions during each loop through the program. The average-case workload is estimated by assuming that the logic can be reduced by Karnaugh mapping or some similar method, and that on the average the number of terms of the function is reduced by two-thirds. Also the implicants subsume on the average only three-quarters of the set of independent literals. Thus the Boolean-method average workload is estimated to be $(n/4)2^n$. Clearly this is still an exponential order growth.

The BD upper bound is known to be n variables per program cycle. Since BD logic is also reducible (this topic is discussed in chapters 2 and 5), it might be assumed that the average program running time can be reduced by one-quarter. Figure 1.6 shows that the growth in BD processor workload to evaluate combinatorial switching functions is strictly linear in the number of process variables.

The speed of the binary decision algorithm makes it applicable to fast time-constant processes, permits greater process model complexity, and/or enables time-sharing of the controller among process loops.

1.2.2 The Microprocessor/Binary Decision Hybrid Controller. In industrial control systems the overall control procedure can often be divided into repeated combinatorial or sequential ON/OFF functions and proportional functions involving the acquisition and generation of analog process signals. Note that BD programmable controllers are well suited to performing ON/OFF functions while microprocessors are better suited to perform the arithmetic computation required to approximate the continuous functions found in control modulation tasks. The hybrid combination of the two processor types enables both control functions to be provided by a single controller. In addition the BD processor gains access through the mP interface to traditional computer peripherals such as terminals, printers and bulk memory.

A microprocessor/binary decision (mP/BD) hybrid differs from conventional programmable controllers in several ways. Having two (or more) co-processors permits the simultaneous execution of classical PID (Proportional, Integral and Derivative) and ON/OFF control algorithms. Secondly, the increased speed of the BD processor makes real-time concurrent multi-programming and program scheduling practical. Since the BD is provided access to a program library via the mP interface and disk, several programs can be scheduled to start and stop at predetermined times. Thirdly, the BD controller can respond to unscheduled events such as process excursions by bringing the appropriate recovery programs in from the library.

A block diagram of the general layout of the mP/BD hybrid programmable controller is shown in figure 1.7. The design of the BD hardware and mP interface is treated in chapter 3.

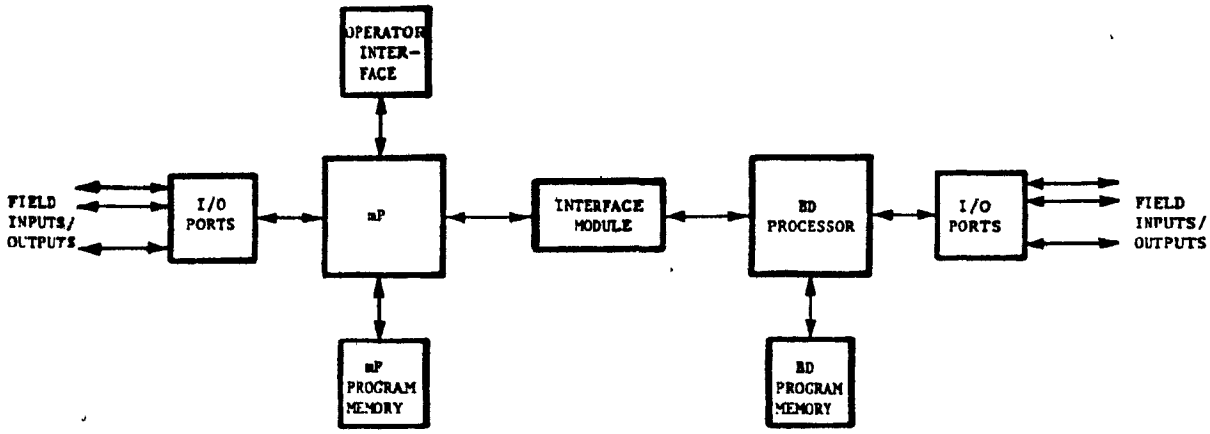
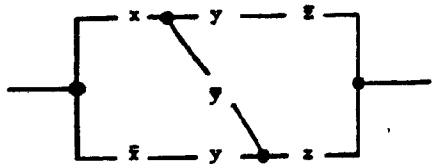


figure 1.7 mP/BD sequence controller block diagram.




```
1.  T   x ;  2,4
2.  T   y ;  0,3
3.  T   z ;  0,I
4.  T   y ;  3,5
5.  T   z ;  I,0
```

figure 1.8 Early BD program by C.Y. Lee.

1.3 Binary Decision Literature Survey

Although BD methods were proposed as an alternative to conventional Boolean techniques by C.Y. Lee in 1959, the idea failed to win much support among logic designers. Clearly Boolean algebra has produced many systematic and effective tools which the logic designer is unlikely to give up in favour of some less developed approach. However a small number of researchers have contributed to the development of BD methodology and hardware. This research is cited below in chronological order.

Shannon - 1938. Prior to 1938, switching circuit engineering consisted of mapping intricate interconnections of relay contacts and switches for the electrical control of telephone exchanges, motor control equipment, etc. Although the theory of general impedance networks had been well defined, the design of switching logic could not be comprehensively described mathematically. A systematic theory for the analysis of switching networks based on the Propositional Calculus subset of Boolean Algebra was presented by Claude Shannon of MIT [Shan38]. This work provided a basis for the representation of switching circuits by sets of equations. These could be manipulated, according to algebraic rules, so as to simplify the circuit.

Shannon presented many of the postulates and theorems now used to manipulate Boolean logic including laws of commutation, association and distribution. A series expansion for switching functions analogous to the Taylor series expansion of continuous functions was also defined. This expansion, known as Shannon's Expansion Theorem, is used to derive the mathematical basis of binary decision logic in chapter 2.

Together with the De Morgan's theorems, this work was used to develop many practical design techniques such as Karnaugh mapping which aids in the recognition of AND, OR, XOR and MAJORITY implicants. A short review of Boolean Algebra is presented in Appendix I.

Lee - 1959. An alternative to the Boolean representation of switching circuits was proposed by C.Y. Lee [Lee59]. Lee's research attempted to overcome the inherent inflexibility of Boolean algebra in manipulating other than series-parallel circuits. Moreover, he thought the Boolean representation to be extremely inefficient for switching function evaluation. Lee proposed a structure called binary decision programming to symbolically represent switching circuits. The method was based on a single instruction

T x; A, B

which says, if the variable x is 0, take the next instruction from program address A, but if x is 1, the next instruction is taken from address B. Switching circuits were described by sequences of these instructions. An example from Lee's work of a typical relay circuit and its binary decision program are shown in figure 1.8. The program is evaluated beginning with instruction 1. If x is 0 then the instruction at address 2 is computed, if x is 1 then address 4. This procedure is continued until the symbol 0 or I is encountered, representing the circuit outputs 0 and 1 respectively. These may also indicate exit addresses.

Binary decision programs were recognized as an efficient representation for the computation of switching functions. Functions

were always evaluated in a number of program instructions that never exceeded the number of switching variables. Lee also showed that binary decision programming could be used as a logic design technique. He proved that switching functions could be more compactly represented in binary decision form than by Boolean sum-of-product expressions. However Lee's logic reduction procedures were not systematic and relied heavily on the skill of the programmer.

Boute - 1976. A binary decision-based programmable controller was described by R.T. Boute of Bell Telephone, Belgium [Bout76]. Contemporary Boolean-based controllers evaluated every Boolean literal in the series of terms comprising the control function. The possibility that the logical result of a term might be determined before all of the variables were tested was ignored. Such controllers were judged too slow for high speed control applications. Boute's research was directed at detecting when the logical proposition implied by a Boolean expression had been conclusively satisfied or violated, so that the controller could branch away from the computation at the earliest opportunity. Instead of accumulating Boolean combinations of input variables to arrive at control outputs, the binary decision controller used the inputs to direct branching within the program so that only enough literals to uniquely define the function output were tested. Only two machine instruction types were included, branch-on-test and output.

Boute proposed to implement the controller using only seven IC chips representing six functional blocks. These blocks included program memory (two ROMs), a presettable program counter, preset logic, an input

variable selector, an output variable selector and latch, and a clock. A block diagram of Boute's binary decision controller is shown in figure 1.9a, together with the instruction word format, figure 1.9b.

Boute concluded that the use of binary decision controllers presented advantages in terms of speed, simplicity and ease of programming. Both combinatorial and sequential switching functions were readily programmed. The problem of binary decision logic reduction was not addressed in this research.

Akers - 1978. A method to define digital functions in terms of a "binary decision diagram" was explored by S.B. Akers of General Electric [Aker78]. The research was aimed at finding concise, implementation-free logic descriptions to bridge the gap between analytically-weak functional design languages and conventional descriptions such as truth tables, Boolean equations, Karnaugh maps, etc., all of which tend to grow exponentially with the number of variables involved. Akers showed that these diagrams could be used to determine the output value of a digital function for analysis and test generation, and to obtain actual implementations. Methods were also described for defining larger digital functions by interconnecting the diagrams.

A binary decision diagram to describe the Boolean function

$$F = A + \bar{B}C$$

is shown in figure 1.10. The function is evaluated by entering the diagram at the node labelled A. If A = 1, then F = 1 and the procedure ends. If A = 0, the branch labelled 0 is followed to node B. If B = 1, then F = 0 and again the procedure ends. Otherwise, the 0-branch is

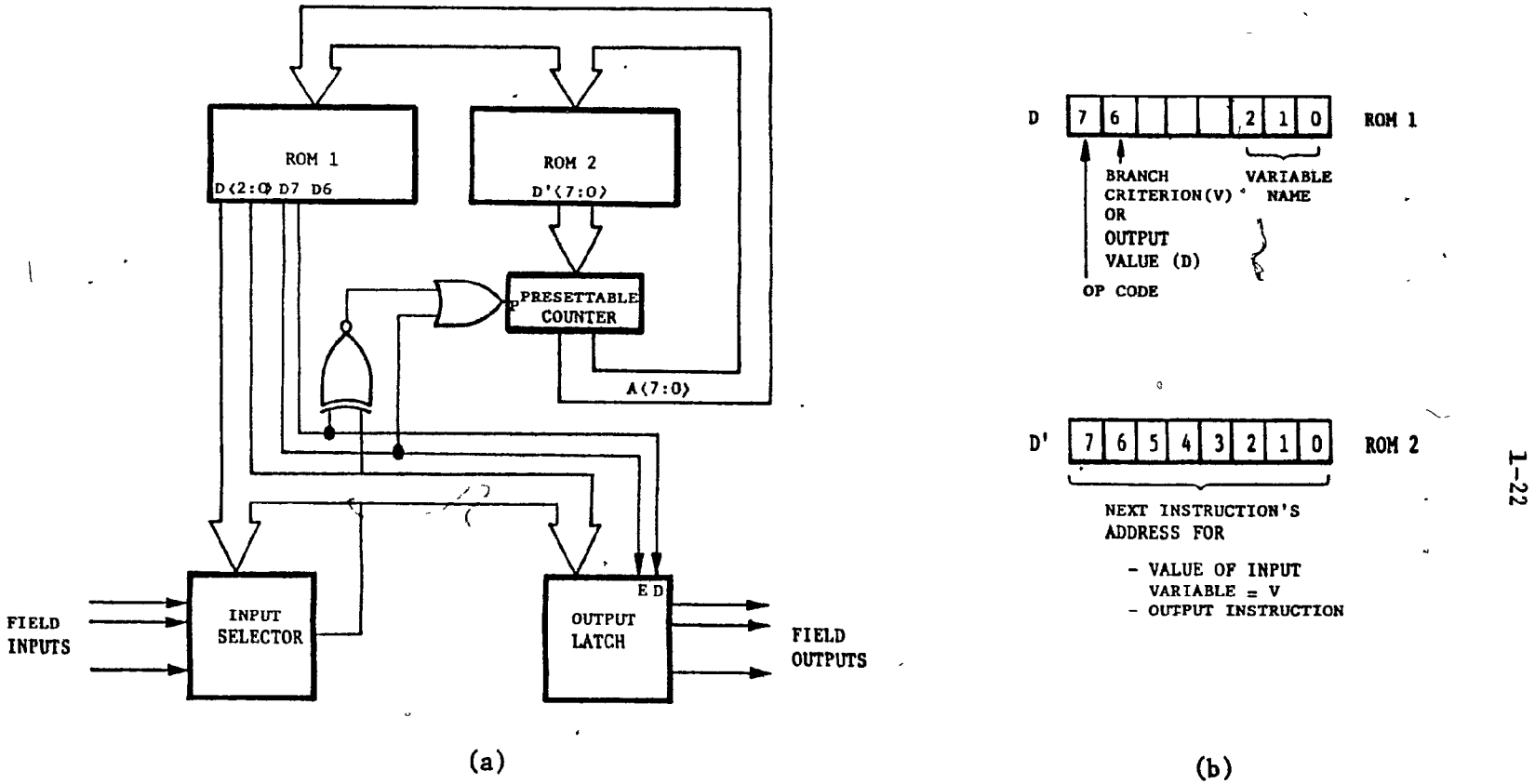


figure 1.9 Boute's BD machine. (a) Block diagram and, (b) instruction word format.

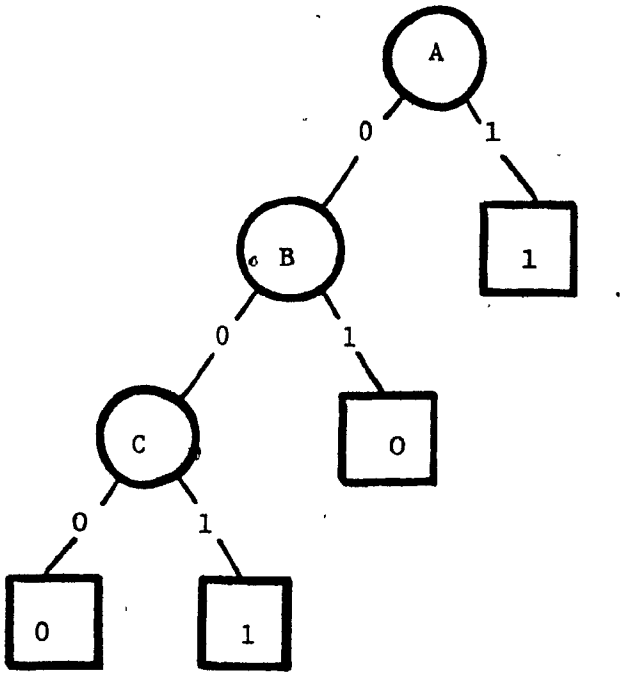


figure 1.10 BD diagram by S.B. Akers.

followed to node C, which determines the value of the function.

Diagrams for several common combinatorial and sequential switching functions were obtained by Akers along with rules for deriving binary decision diagrams from truth tables and Boolean equations. A method to define a BD diagram in a computer as a list of ordered triples (variable name, 0-branch, 1-branch), was also devised. It is easily seen that the binary decision diagram of Akers is functionally equivalent to the binary decision program of Lee. They are in fact mutually complementary as the BD program is easily translated to computer code but is difficult to analyze, whereas the BD diagram, being a graphical description, is easily analyzed but is difficult to implement directly on a computer. In this way they have the same relationship as a conventional computer program and its logic flowchart. Akers also noted the need for logic reduction and demonstrated diagram structures that permitted simplification. He recognized that different and often simpler diagrams could be obtained by changing the order of examination of variables. This concept is discussed in chapter 5.

Mange, Cerny, Thayse, et al. - 1978. The problems involved in the representation, minimization and hardware implementation of binary decision programs/diagrams were studied cooperatively and separately by Mange, Cerny, Thayse, et al. in 1978.

A general method to implement any binary decision diagram in hardware was found by Mange which involved replacing every node in the diagram by a one-to-two demultiplexor (DEMUX) [Mang78]. Unlike binary trees, BD diagrams can have more than one path into a node. These were handled by ORing together all multiple node inputs to obtain a single

bit input to the DEMUX. Such a circuit provides a spatial realization of a binary decision diagram. Mange also described a general sequential realization which represented the BD diagram as a series of instructions executed sequentially by a "Binary Decision Machine."

The synthesis of minimal binary decision diagrams for multiple output, incompletely specified switching functions was also studied by Cerny, Mange and Sanchez [Cern79]. Their method was to first reduce the logic to its minimal (prime implicant) form. Two algorithms were devised to do this, a deterministic but lengthy procedure and a fast heuristic. The resulting binary tree was then converted to a binary decision diagram. The methods were programmed in FORTRAN on a PDP-11/20 in 1979.

The binary decision program optimization methods considered by Cerny and others were not applicable to programs in which parallel logic branches reconverged at a node. A class of functions called P-functions was devised by Thayse [Thay81] to provide an analytical means of finding a reduced form for reconvergent functions. The method transforms unreduced Boolean equations to binary diagrams by iterative application of a set of composition laws acting upon a pair of functions (g,h) called the domain function and the codomain function, respectively. The initial value of the domain function is the Boolean equation F and the initial value of the codomain function is the Boolean constant 1. The procedure terminates when the domain function has been transformed into 1 and the codomain function has become the function F. Many ways exist to apply the composition laws each producing a different binary program. All of the rules are exhaustively applied to the functions during each

iteration. Only a small number of these functions are useful for generating optimal programs. It is not known whether this method has been implemented in software.

Zsombor-Murray, Vroomen et al. - 1978. An extension of Boute's Binary Decision machine to incorporate some measure of parallel architecture was proposed and implemented by Zsombor-Murray, Vroomen, Tho and Holck at McGill [Tho79,Zsom79]. This improvement enabled multiple switching functions to be evaluated simultaneously. The instruction set was expanded to include conditional branching on either input polarity and parallel output of, optionally, four or fourteen bits. The number of input and output points was expanded to 64 and 14, respectively, as opposed to eight of each in the Boute machine.

The binary decision controller was implemented as a demonstration unit consisting of a BD processor, field I/O ports and a control and programming console. The system was TTL compatible, needed only a single 5-volt power supply and operated in three modes; loading, verification and execution. Hand-compiled programs were loaded into the instruction memory by means of 16 data entry switches after presetting the program counter to the address of the first instruction. The program could be verified by single-stepping through it from the control console or could be executed at either of two clock rates, 0.5Hz or 70kHz. Examples were devised and tested to demonstrate the feasibility of implementing combinatorial and sequential switching logic in binary decision logic. A block diagram of the Holck machine is illustrated in figure 1.11.

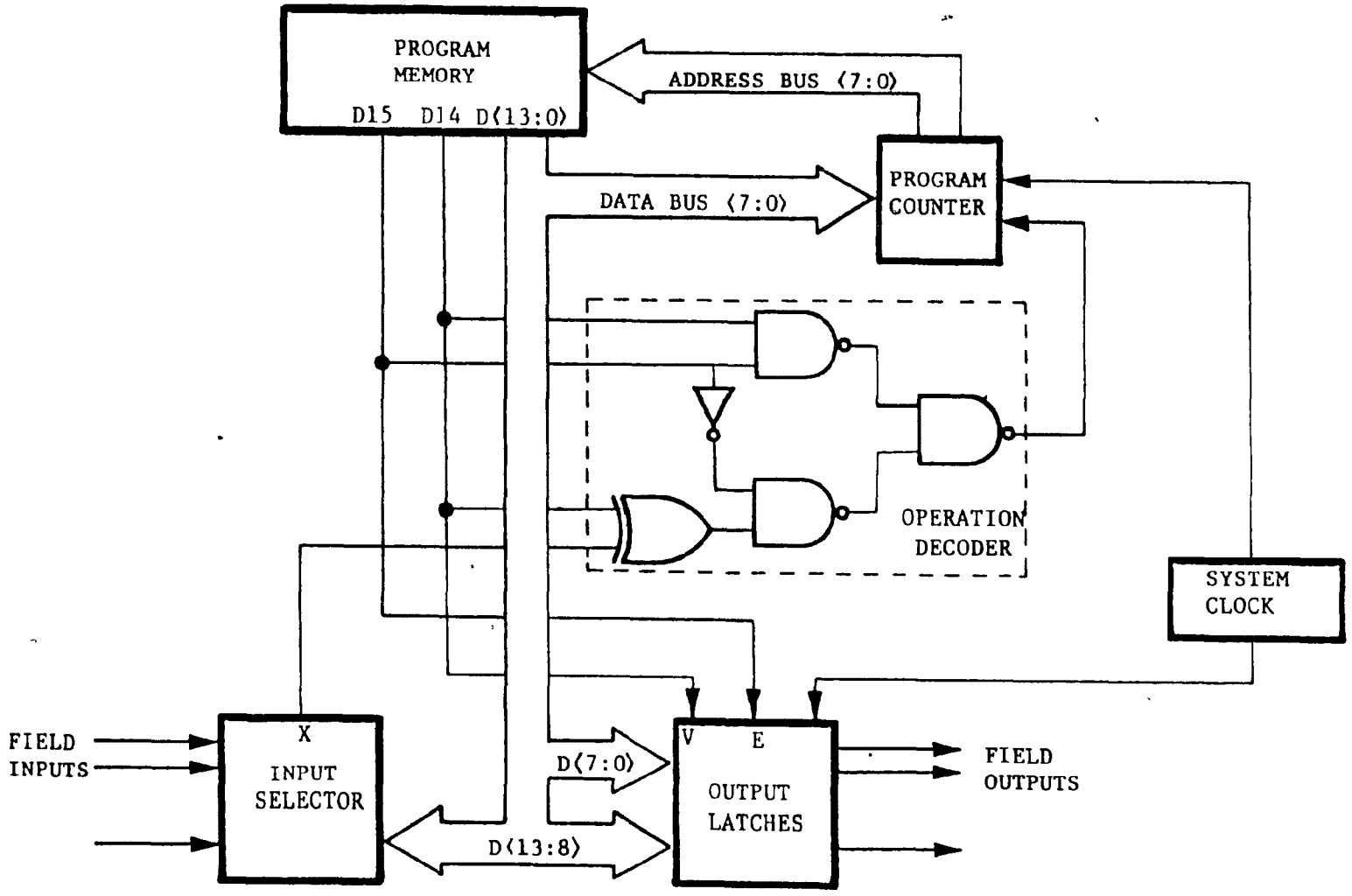


figure 1.11 Holck BD machine block diagram.

1.4 Research Motivation and Objectives

The concept, design and implementation of binary decision-based programmable controllers has been a topic of interest to several independent researchers as shown above. They have established that purely ON/OFF control functions are best performed by BD-based automata as opposed to conventional microprocessors. However, current BD architectures are mainly suited to applications in which a controller must simultaneously attend to a number of small independent process loops, e.g., each requiring only a few bits of input and output. The objective of this research was to develop a hybrid mP/BD PLC prototype that would be better suited for process control applications of a larger scale.

Consider that the size of a complete BD program grows exponentially as $2^{n+1}-1$, where n is the number of input variables. Five 4-input procedures would require no more than 155 instructions, easily accommodated within a 256 location program memory. It is doubtful whether any programmable controller, PLA or microprocessor-based, or even hardwired logic could economically compete with a BD implementation which would inevitably produce a computed output after executing only four instructions.

On the other hand, a single 64 input program could require as many as $2^{65}-1$ or $3.7\text{x}10^{19}$ instructions, an intractable size! Using standard 64k memory, the limiting program size is still only 15 inputs. Since many common processes have thirty to forty inputs, one (or more) of the following solutions might be necessary to accommodate these in a BD controller:

1) Limit the number of inputs to the BD processor.

2) Partition the global control algorithm into several manageable independent sections, each executed by a separate BD controller.

3) As 2) above, but the program sections required at any particular time are paged in and out of the BD program memory. This is equivalent to "virtual memory".

4) Reduce the program logic to its minimum size so that the reduced program can be accommodated in memory.

The hybrid controller proposed in this thesis consisting of a BD machine and a microprocessor "supervisor" would be able to implement any of these solutions using the integration scheme as shown in figure 1.12.

The standard microprocessor system which consists of basic blocks such as the microprocessing unit, RAM memory and analog I/O convertors, acts as the central processor. One (or more) BD controllers are connected to the mP via standard data, address and control busses. Binary decision program code, compiled by the mP, is transferred to the BD memory in the same manner as data is stored in a standard RAM memory. The operation of the BD is under the control of the supervisory mP. In the loading mode, the BD program counter is preset to the initial location of the program and is incremented by the mP as instructions are loaded. The BD is activated by again presetting its program counter and setting it in the run mode. From this moment, the BD controller performs its repeated ON/OFF control routines and the mP is free for other tasks such as running proportional control algorithms.

This research encompasses the design of hardware required to interface the BD with the mP, and the establishment of necessary

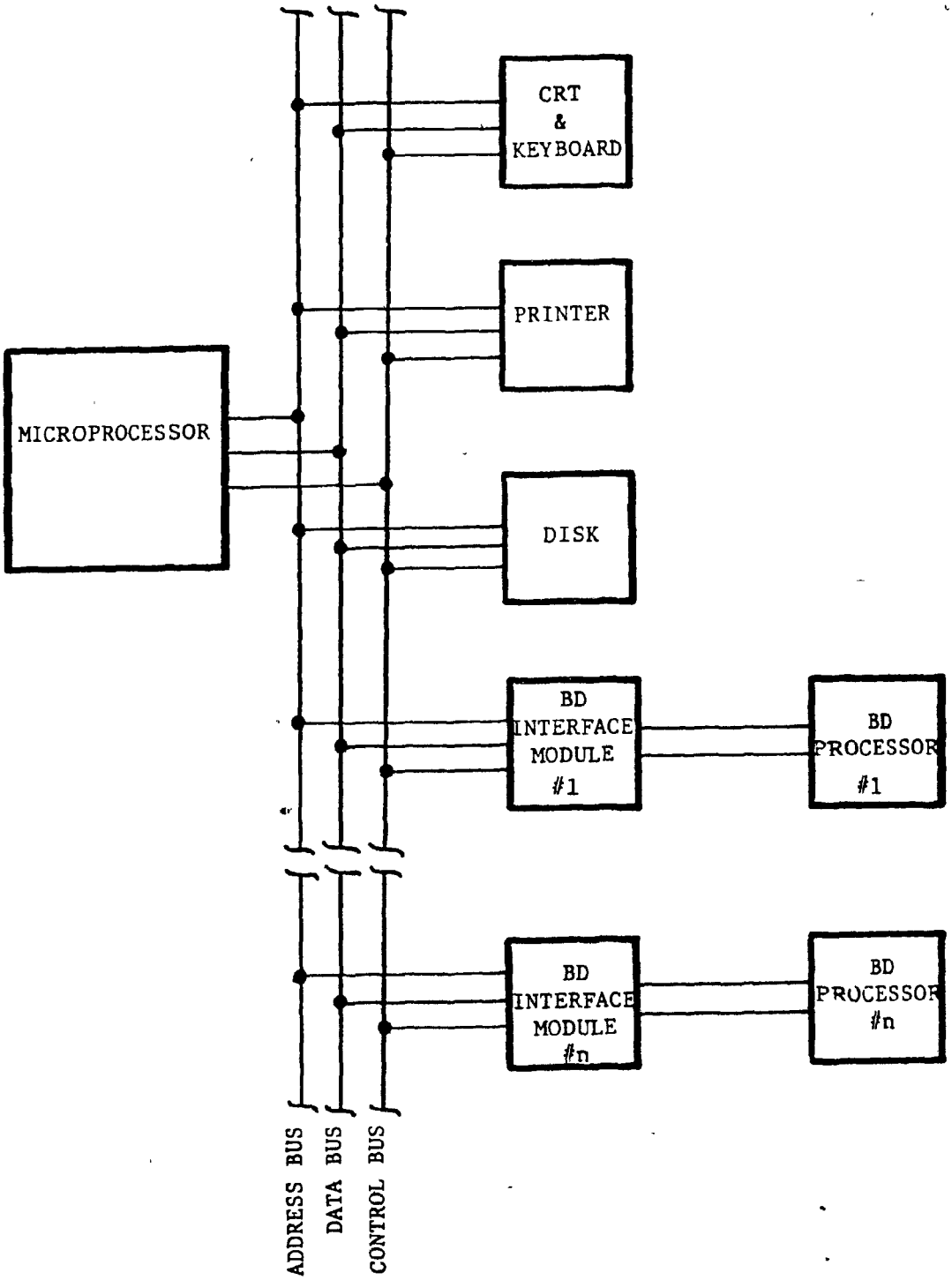


figure 1.12 mP/BD integration scheme.

structures and protocols to enable interprocessor communication. This was achieved by the author in cooperation with Levi [Huds82,Levi82a]. The design of the mP/BD system's hardware and software is discussed in chapters 3 and 4, respectively. In addition, the scope includes the development of a practical optimizing compiler for BD programs that would fit into the hybrid controller and produce control programs without presenting too much overhead to the main tasks of the system. Since it has been shown that decision program optimization is an NP-complete problem under polynomial sized inputs [Hyaf76,More80], we were restricted to finding a method that produces an near-optimal program in reasonable time. The compiler design is presented in chapter 5.

The ulterior motivation of this research, however, is to promote the concept of using binary decision automata for the computation of combinatorial and sequential ON/OFF control functions. A recent survey of programmable controllers did not reveal a single BD-based machine [Flyn84]. In spite of this the BD machine could be a valuable industrial tool. To gain acceptance, the theory, design and implementation of BD machines must be formalized. It is believed that the machine is not generally understood because the theory of operation is not yet rigorously defined and because conventional relay ladder and gate diagrams remain ditticult to represent in terms of BD logic. As well systematic procedures for writing and minimizing BD programs need to be developed. It is hoped that this work will provide some of this formalism and encourage the commercialization of BD hardware.

CHAPTER 2

BINARY DECISION THEORY

2.0 Introduction

This chapter presents a theoretical basis for binary decision programming. The relationships between the binary decision method and conventional combinatorial and sequential switching function representations are developed. Proofs are offered to show the equivalence of the different methods, and certain characteristics of binary decision programs that suggest an approach to logic minimization are discussed.

2.1 Binary Decision Programs

It may be shown that any combinatorial or sequential switching function can be represented as a sequence of binary decision instructions i.e., as a binary decision program. This method of description was proposed by Lee as an alternative to the use of Boolean equations, introduced into switching theory by Shannon in 1938.

A binary decision program instruction is a two-address conditional transfer statement of the form:

Label x; A, B

This instruction is read: if the variable x is 0, then evaluate the program statement labelled A, else if x is 1, then evaluate statement B. This instruction is represented diagrammatically in figure 2.1.

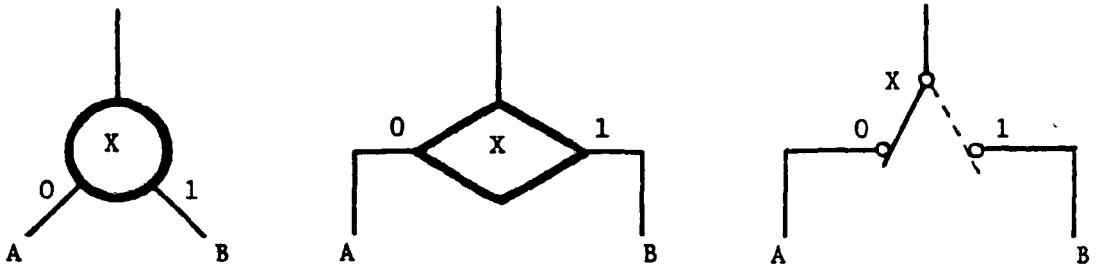


figure 2.1 IF-THEN-ELSE schematic representations.

The deficiency of the Boolean method lies in the large amount of computational work that must be expended to evaluate a typical expression. In particular, the Boolean method exhaustively evaluates every literal and implicant comprising the switching function, an amount of work of exponential order. The fact that an implicant can be shown to be FALSE if at least one of its literals is FALSE or that an entire expression is TRUE if at least one of its implicants is TRUE is completely disregarded. In contrast, the binary decision method evaluates any combinatorial switching function in a number of steps which is always equal to or less than the number of input variables.

Consider the following control function of four implicants in three independent variables:

$$Q(A,B,C) = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \qquad (2.1)$$

A PLC program to implement equation 2.1 in an MC14500 ICU-based controller is presented in figure 2.2a.

Figure 2.2b illustrates the mathematical operations performed by the PLC program to evaluate Q(0,1,0). Although nineteen instructions are executed in computing the function in the conventional way, the value of the first implicant was known after just two program steps since the second literal in this term has the value 0. Likewise the value of Q was conclusively determined after the seventh program step since the value of the second implicant is 1. A binary decision program for equation 2.1 is shown in figure 2.2c. Figure 2.2d illustrates the sequence of BD instructions that are executed to compute Q(0,1,0). In the BD case, only three instructions are executed, equal to the number of input variables.

```
LDC     A
ANDC    B
AND     C
STO     TEMP

LDC     A
AND     B
ANDC    C
OR      TEMP
STO     TEMP

LD      A
ANDC    B
ANDC    C
OR      TEMP
STO     TEMP

LD      A
AND     B
AND     C
OR      TEMP
STO     Q
```

(a)

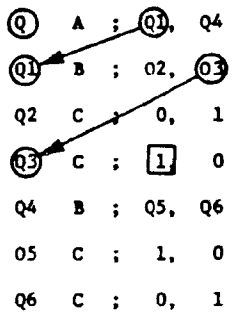
$$
\begin{aligned}
Q &= 1\cdot\bar{B}\cdot C + \bar{A}\cdot B\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 1\cdot 0\cdot C + \bar{A}\cdot B\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 0\cdot 0 + \bar{A}\cdot B\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 0 + \bar{A}\cdot B\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 0 + 1\cdot B\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 0 + 1\cdot 1\cdot\bar{C} + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 0 + 1\cdot 1 + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 0 + 1 + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 1 + A\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 1 + 0\cdot\bar{B}\cdot\bar{C} + A\cdot B\cdot C \\
&= 1 + 0\cdot 0\cdot\bar{C} + A\cdot B\cdot C \\
&= 1 + 0\cdot 1 + A\cdot B\cdot C \\
&= 1 + 0 + A\cdot B\cdot C \\
&= 1 + A\cdot B\cdot C \\
&= 1 + 0\cdot B\cdot C \\
&= 1 + 0\cdot 1\cdot C \\
&= 1 + 0\cdot 0 \\
&= 1 + 0 \\
&= 1
\end{aligned}
$$

19 STEPS

(b)

```
Q    A ; Q1, Q4
Q1   B ; Q2, Q3
Q2   C ; 0,  1
Q3   C ; 1,  0
Q4   B ; Q5, Q6
Q5   C ; 1,  0
Q6   C ; 0,  1
```

(c)



(d)

figure 2.2 Sequence controller evaluation of combinatorial switching functions. (a) ICU program for equation 2.1. (b) Boolean evaluation method. (c) BD program for equation 2.1. (d) BD evaluation method.

Another deficiency of the Boolean method is the description of sequential control functions. The temporal character of such functions cannot be modelled by simple Boolean Algebra, and external memory is required to save previously computed values of the function. Binary decision programs can embody circuit history information within the transfer paths of the program, and so are better suited to the representation of sequential functions. Figure 2.3 shows a BD program flowchart for a common sequential circuit, the toggle flip-flop. Program flow is in the left side of the diagram when Q=0 and in the right side when Q=1. A toggle input controls the switching of the program flow from side to side. Thus the BD program is able to remember the previous state of the T flip-flop in the transfer structure of the program and does not have to reread Q from an external memory. BD flowcharts are discussed in detail in section 2.2.2.

2.2 The Binary Decision Representation of Combinatorial Switching Functions

The mathematical basis of binary decision logic did not seem evident to Lee. It has since been demonstrated by various authors [Aker78,Cern79,More82], however, that binary decision programs are related to the series expansion of a Boolean equation.

2.2.1 The Binary Decision-Combinatorial Circuit Analogy. Consider the switching function:

$$Q = f(X_1, X_2, \ldots, X_n) \qquad (2.2)$$

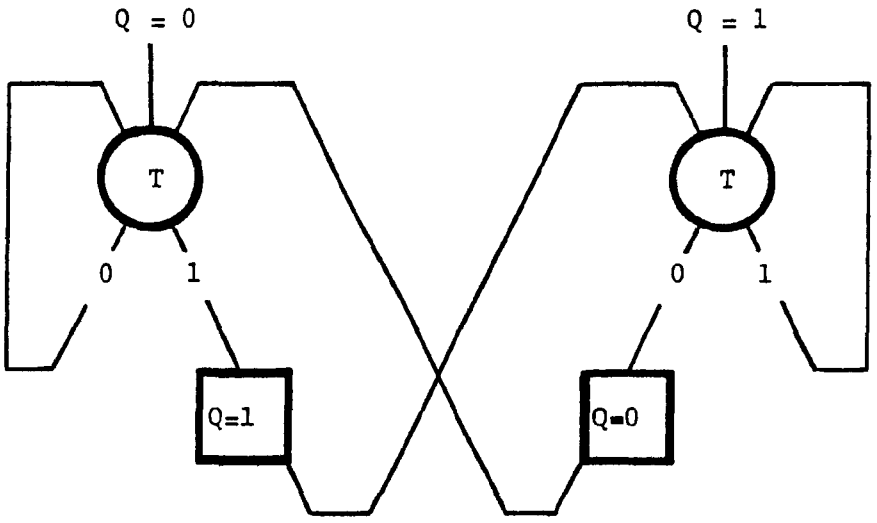


figure 2.3 BD representation of the toggle flip-flop.

The series expansion of this function is obtained by recursively expanding equation 2.2 about each of its variables according to the formula:

$$f(X_1,X_2,\ldots,X_n) = \bar{X}_1 \cdot f(0,X_2,\ldots,X_n) + X_1 \cdot f(1,X_2,\ldots,X_n) \qquad (2.3)$$

Continuing this process n times yields:

$$f(X_1,\ldots,X_n) = f(0,\ldots,0)\bar{X}_1\bar{X}_2\ldots\bar{X}_n + f(0,\ldots,1)\bar{X}_1\bar{X}_2\ldots X_n + \ldots + f(1,\ldots,1)X_1X_2\ldots X_n \qquad (2.4)$$

Equation 2.3 is known as Shannon's Expansion Theorem. The proof of equation 2.3 is demonstrated in Theorem 2.1.

Theorem 2.1: Shannon's Expansion Theorem.

Proof: Let X_1 equal 0. Thus:

$$f(0,X_2,\ldots,X_n) = (\bar{0}) \cdot f(0,X_2,\ldots,X_n) + (0) \cdot f(1,X_2,\ldots,X_n)$$
$$= f(0,X_2,\ldots,X_n)$$

Now let X_1 equal 1. Thus:

$$f(1,X_2,\ldots,X_n) = (\bar{1}) \cdot f(0,X_2,\ldots,X_n) + (1) \cdot f(1,X_2,\ldots,X_n)$$
$$= f(1,X_2,\ldots,X_n)$$

This completes the proof by perfect induction. □

If equation 2.2 is expanded about an arbitrary variable, X_i, the following equation is obtained:

$$Q = \bar{X}_i \cdot f(X_1,\ldots,X_i=0,\ldots,X_n) + X_1 \cdot f(X_1,\ldots,X_i=1,\ldots,X_n) \qquad (2.5)$$

Now let:

$$f(X_1,\ldots,X_i=0,\ldots,X_n) = G$$

and,

$$f(X_1,\ldots,X_i=1,\ldots,X_n) = H$$

then,

$$Q = \bar{X}_i \cdot G + X_i \cdot H \tag{2.6}$$

for any arbitrary function Q and variable X_i. The binary decision instruction:

$$Q \quad X_i;\ G,\ H \tag{2.7}$$

is shown to be equivalent to equation 2.6 in Lemma 2.1.

Lemma 2.1: $Q = \bar{X}_i \cdot G + X_i \cdot H \equiv Q\ X_i;\ G,\ H$

Proof: By definition, the binary decision instruction can be represented by the following truth table:

X_i	Q
0	G
1	H

Evaluating equation 2.6 with $X_i = 0$, i.e.,

$$Q = (\bar{0}) \cdot G + (0) \cdot H$$
$$= G$$

and with $X_i = 1$, i.e.,

$$Q = (\bar{1}) \cdot G + (1) \cdot H$$
$$= H$$

yields the truth table:

X_i	Q
0	G
1	H

which is identical to the truth table of the binary decision instruction. Thus equations 2.6 and 2.7 are shown to be equivalent by perfect induction. □

Since the product of each expansion can be represented as a binary decision instruction, it follows directly that a binary decision program is equivalent to the complete series expansion of a Boolean switching function. To illustrate this point consider the series expansion of equation 2.1, i.e.,

$$Q(A,B,C) = \bar{A}\bar{B}\bar{C}\cdot f(0,0,0) + \bar{A}\bar{B}C\cdot f(0,0,1) + \bar{A}B\bar{C}\cdot f(0,1,0) + \bar{A}BC\cdot f(0,1,1) + A\bar{B}\bar{C}\cdot f(1,0,0) + A\bar{B}C\cdot f(1,0,1) + AB\bar{C}\cdot f(1,1,0) + ABC\cdot f(1,1,1) \quad (2.8a)$$

where: $f(0,0,0) = 0$ $\quad f(1,0,0) = 1$

$f(0,0,1) = 1$ $\quad f(1,0,1) = 0$

$f(0,1,0) = 1$ $\quad f(1,1,0) = 0$

$f(0,1,1) = 0$ $\quad f(1,1,1) = 1$

Substituting the values of f(0,0,0), etc., into equation 2.8a and factoring terms using the Distributive Property yields equation 2.8b:

$$Q(A,B,C) = \bar{A}[\bar{B}(\bar{C}\cdot 0+C\cdot 1) + B(\bar{C}\cdot 1+C\cdot 0)] + A[\bar{B}(\bar{C}\cdot 1+C\cdot 0) + B(\bar{C}\cdot 0+C\cdot 1)] \quad (2.8b)$$

Now let:

$$Q1 = [\bar{B}(\bar{C}\cdot 0+C\cdot 1) + B(\bar{C}\cdot 1+C\cdot 0)] \quad (2.9)$$

and,

$$Q4 = [\bar{B}(\bar{C}\cdot 1+C\cdot 0) + B(\bar{C}\cdot 0+C\cdot 1)] \quad (2.10)$$

then,

$$Q(A,B,C) = \bar{A}\cdot Q1 + A\ \ Q4$$

Similarly, let:

$$Q2 = \bar{C}\cdot 0 + C\cdot 1$$

and,

$$Q3 = \bar{C}\cdot 1 + C\cdot 0$$

in equation 2.9, and let:

$$Q5 = \bar{C} \cdot 1 + C \cdot 0$$

and,

$$Q6 = \bar{C} \cdot 0 + C \cdot 1$$

in equation 2.10. Equation 2.8b is rewritten in terms of the expansion equations for each variable as follows:

$$Q = \bar{A} \cdot Q1 + A \cdot Q4$$

$$Q1 = \bar{B} \cdot Q2 + B \cdot Q3$$

$$Q2 = \bar{C} \cdot 0 + C \cdot 1$$

$$Q3 = \bar{C} \cdot 1 + C \cdot 0$$

$$Q4 = \bar{B} \cdot Q5 + B \cdot Q6$$

$$Q5 = \bar{C} \cdot 1 + C \cdot 0$$

$$Q6 = \bar{C} \cdot 0 + C \cdot 1$$

However since it is known by Lemma 2.1 that:

$$Q = \bar{X}_i \cdot G + X_i \cdot H \equiv Q \quad X_i; \; G, H$$

then this sequence of equations is logically equivalent to the binary decision program:

Q A; Q1, Q4

Q1 B; Q2, Q3

Q2 C; 0, 1

Q3 C; 1, 0

Q4 B; Q5, Q6

Q5 C; 1, 0

Q6 C; 0, 1

This equivalence is formally stated in Theorem 2.2.

Theorem 2.2: For every combinatorial switching function there exists an equivalent binary decision program.

Proof: Any combinatorial switching function can be expressed as an equivalent series expansion by Shannon's Expansion Theorem. A series of single variable equations of the form:

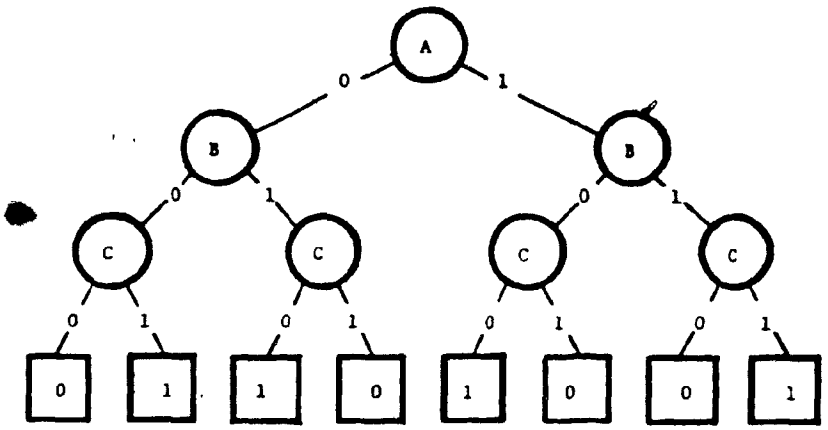
$$F = \bar{X}_i \cdot G + X_i \quad H$$

where $i \in \{1,2,\ldots,n\}$ and G, H define subexpressions of F in the series expansion, may be obtained by algebraic manipulation. Taken as a whole, these equations are equivalent to the original switching function. By Lemma 2.1, each of the single variable equations is equivalent to a binary decision instruction. Hence, the sequence of binary decision instructions, i.e., the binary decision program, must also be equivalent to the original switching function. □

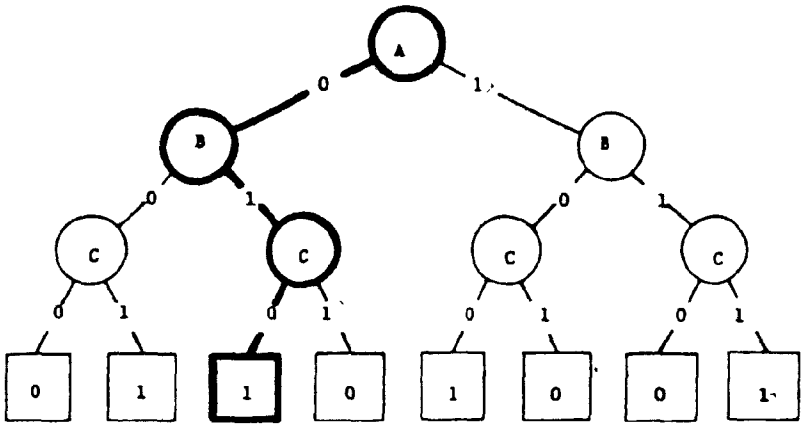
2.2.2 Binary Decision Diagrams. A method of representing digital functions in terms of a binary decision diagram was proposed by Akers [Aker78].

BD program and BD diagram representations of switching functions are equivalent and complementary. A BD program is easily translated to executable computer code but is difficult to analyze, whereas a BD diagram, being a graphical description, is easily analyzed but is difficult to implement on a computer. It is convenient to regard the BD diagram as the logic flowchart for BD programs.

Consider the BD diagram for equation 2.1, figure 2.4a. The function is evaluated by traversing the diagram, i.e., visiting a sequence of nodes indicated by the directed edges, or branches, of the



(a)



(b)

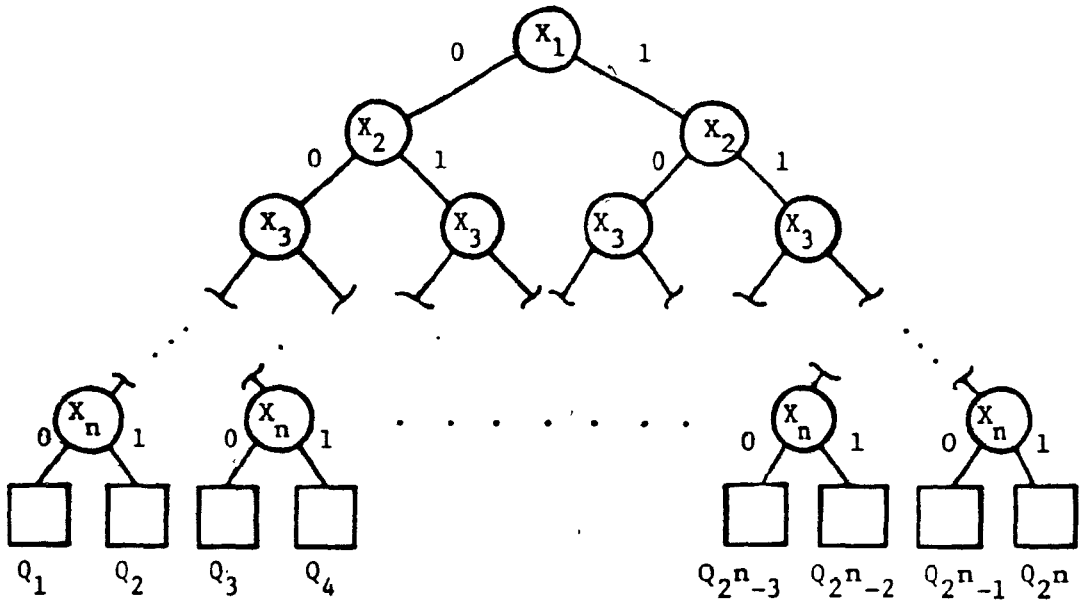
figure 2.4 BD diagram for a combinatorial switching function (a). (b) BD evaluation method.

diagram, beginning from the topmost node. (In this example each node has just one entry and two exit branches: a binary tree [Aho74]. Nodes may be permitted to have more than one entry branch, but not more than two exits.) During the traversal, the exit branch from each node is selected on the basis of the value of the input variable listed in the node. This process continues until an exit branch is encountered which leads to one of the rectangular nodes at the lowest level of the diagram, in which the values of the function have been pretabulated. Figure 2.4b illustrates the evaluation of Q(0,1,0). Comparing figure 2.4 with figure 2.2c) and d) reinforces the analogy between the two representations.

Depending on the form in which the function is defined a number of procedures for deriving BD diagrams from switching functions have been proposed. A simple procedure to construct a BD diagram from an ordered truth table is described below. A n-level binary tree is constructed such that every node in the tree has one unique entry branch (except for the root) and two unique exit branches. All of the exit branches of the nodes at the lowest level are terminated by rectangular output nodes. The non-output nodes are labelled with the switching variables appearing at the top of the truth table, in the following manner. The root node is labelled with the left-most variable of the truth table. The second variable from the left is associated with all the nodes at the second level of the tree, immediately below the root. The third variable is associated with the third level of the tree, etc., figure 2.5a and b. The left exit branch of every node is labelled with a 0 and the right exit branch with a 1. Thus each path in the tree is uniquely identified

X_1	X_2	. . .	X_n	Q
0	0	. . .	0	Q_1
0	0	. . .	1	Q_2
.	.		.	
.	.		.	
.	.		.	
1	1	. . .	1	Q_{2^n}

(a)



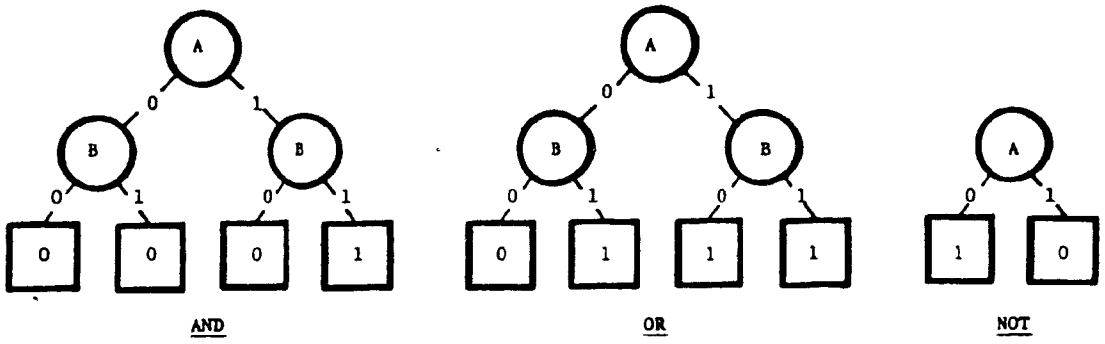
(b)

figure 2.5 Conversion of truth table logic to BD logic.
(a) Truth table and, (b) corresponding BD diagram.

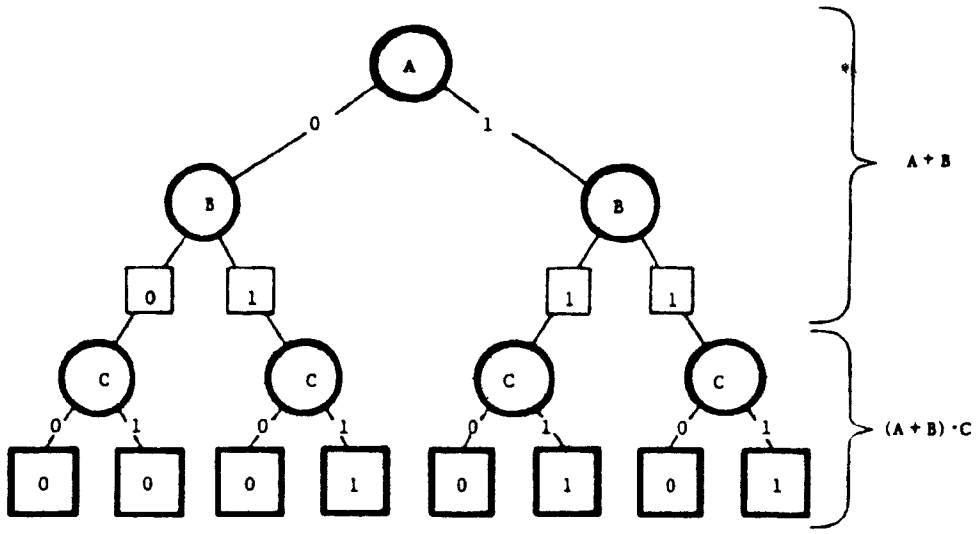
by the sequence of branch labels, 0 or 1, formed by the concatenation of the input values selecting that path, e.g., the leftmost path is selected by the set (A,B,C) = (0,0,0). The output nodes are then labelled from left to right with the corresponding values of the switching function tabulated in the truth table from top to bottom. A BD diagram consisting of all 2^n combinations is called a complete diagram and can be shown to have 2^n-1 decision nodes and 2^n outputs. Since a complete BD diagram contains no additional information compared to that contained in a truth table, they may be considered equivalent representations differing only in information topology.

This knowledge leads to another simple way of proving that any combinatorial switching function can be represented as a binary decision program. It is known that any switching circuit can be written in the canonical form, involving only three operators, AND, OR, and NOT. Since BD diagrams can be constructed from truth tables for these three functions, figure 2.6a, it follows that any function can be "built" from combinations of these simple BD diagrams in the same manner as a logic gate circuit is assembled using the conventional Boolean approach, figure 2.6b. (The BD diagrams in figure 2.6 are shown unreduced. BD logic minimization is discussed in section 2.2.4 and in chapter 5.)

2.2.3 Analysis of Binary Decision Method. As previously discussed in chapter 1, the growth in computational complexity of the Boolean representation of switching circuits is an exponential function of the number of independent circuit variables, whereas the binary decision method exhibits strictly linear growth. These relationships were



(a)



(b)

figure 2.6 BD diagrams for the fundamental logic connectives. (a) AND, OR and NOT functions. (b) Superposition of BD diagrams to form a compound function.

illustrated in figure 1.6. The essential difference between the two computational methods was suggested in figure 2.2.

A physical understanding of the BD computational process can be gained from the topology of a binary decision diagram. The series expansion of a n-variable function, equation 2.4, contains 2^n terms, representing all of the combinations of the n variables beginning with $(\bar{X}_1\bar{X}_2\ldots\bar{X}_n)$ and ending with $(X_1X_2\ldots X_n)$. The coefficient of each term in the series is the value of the function corresponding to that combination, such that for a binary-valued function the coefficient is 1 for all of the minterms that are subsumed by the prime implicants of the function and 0 for all others. A complete binary decision diagram, figure 2.5, is a one-to-one mapping of the terms of the series expansion to the 2^n paths of the tree. Coefficients are represented by the output boxes at the bottom of the tree. Hence if one writes out the literal expression implied by the tree, one obtains:

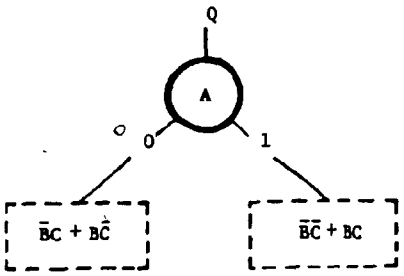
$$f(X_1,X_2,\ldots,X_n) = \bar{X}_1[\bar{X}_2\ldots[\bar{X}_n\cdot f(0,0,\ldots,0) + X_n\cdot f(0,0,\ldots,1)] + \ldots] + X_1[\bar{X}_2\ldots[\bar{X}_n\cdot f(1,0,\ldots,0) + X_n\cdot f(1,0,\ldots,1)] + \ldots]$$

i.e., the series expansion of the Boolean function in fully factored form.

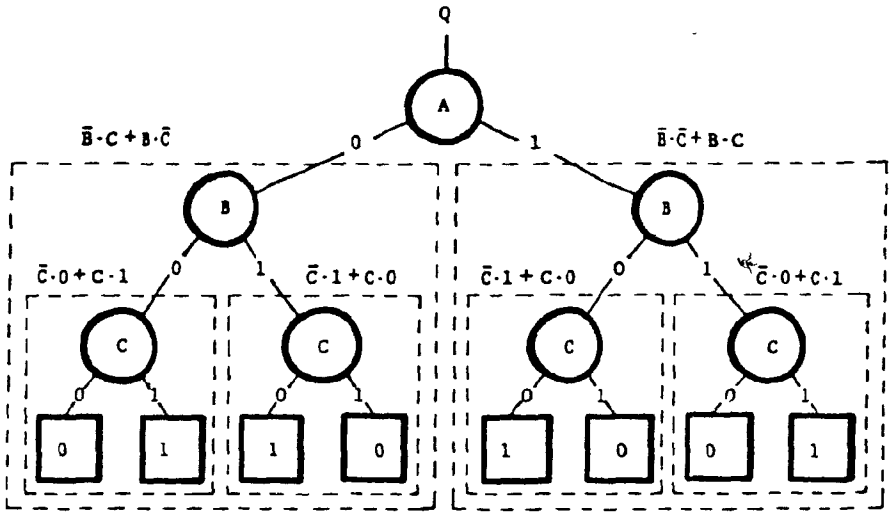
In this way, the BD diagram factors the terms of the function such that all of the minterms subsuming the literal $\bar{X}_1$ are grouped into one subtree, and the minterms subsuming the literal X_1 appear in a separate subtree. Thus the minterms represented by these two subtrees are mutually disjoint. If $\bar{X}_1$ is TRUE then X_1 must be FALSE and all of the minterms subsuming X_1 are FALSE as well. Else, if X_1 is TRUE then $\bar{X}_1$ must be FALSE and the set of minterms subsuming $\bar{X}_1$ must all be FALSE.

In either case, exactly one-half of the minterms are rejected immediately based on the evaluation of only one variable, which is never reevaluated. (This is strictly true only in the case of so-called simple BD programs, i.e., those in which a variable never appears more than once in any particular path. The more general case of non-simple programs is discussed by Thayse [Thay81].) This division process is repeated at each level of the diagram. As only one variable is examined per level, and as there exists only n levels, hence not more than n variables are ever examined in this computational process. The binary decision method branches away from implicants whose logical proposition is known to be FALSE and does so at the earliest possible time. This is the crucial difference between the conventional Boolean approach and the binary decision program method.

Consider figure 2.7, the BD diagram of equation 2.1. Figure 2.7a shows a diagram of the function after the first application of Shannon's Expansion Theorem. The function is divided into two bracketed terms, one ANDed with $\bar{A}$ and the other with A. The expressions within the brackets no longer involve the literal A as a result of the factoring process. Figure 2.7b shows the complete BD diagram of the fully expanded function in which the subtrees have been labelled with the corresponding terms from the series expansion. At each level one additional literal has been factored from the remaining expressions. The evaluation of Q(A,B,C) = (0,1,0) was shown in figure 2.4. In figure 2.7c the corresponding series expansion is evaluated in the binary decision manner.



(a)



(b)

$$
\begin{aligned}
Q(0,1,0) &= \bar{0}\cdot[\bar{B}(\bar{C}\cdot 0 + C\cdot 1) + B(\bar{C}\cdot 1 + C\cdot 0)] + 0\cdot[\quad] \\
&= \bar{1}(\bar{C}\cdot 0 + C\cdot 1) + 1(\bar{C}\cdot 1 + C\cdot 0) \\
&= \bar{0}\cdot 1 + 0\cdot 0
\end{aligned}
$$

3 STEPS

(c)

figure 2.7 Conversion of Boolean equations to BD diagrams using the Shannon series expansion.
(a) Expansion of equation 2.1 about variable A.
(b) Complete series expansion.
(c) Evaluation of equivalent Boolean expression.

A further computational advantage of binary decision over conventional Boolean algebra is the ability of the BD method to evaluate multiple switching functions simultaneously. For example, consider the following set of process control functions.

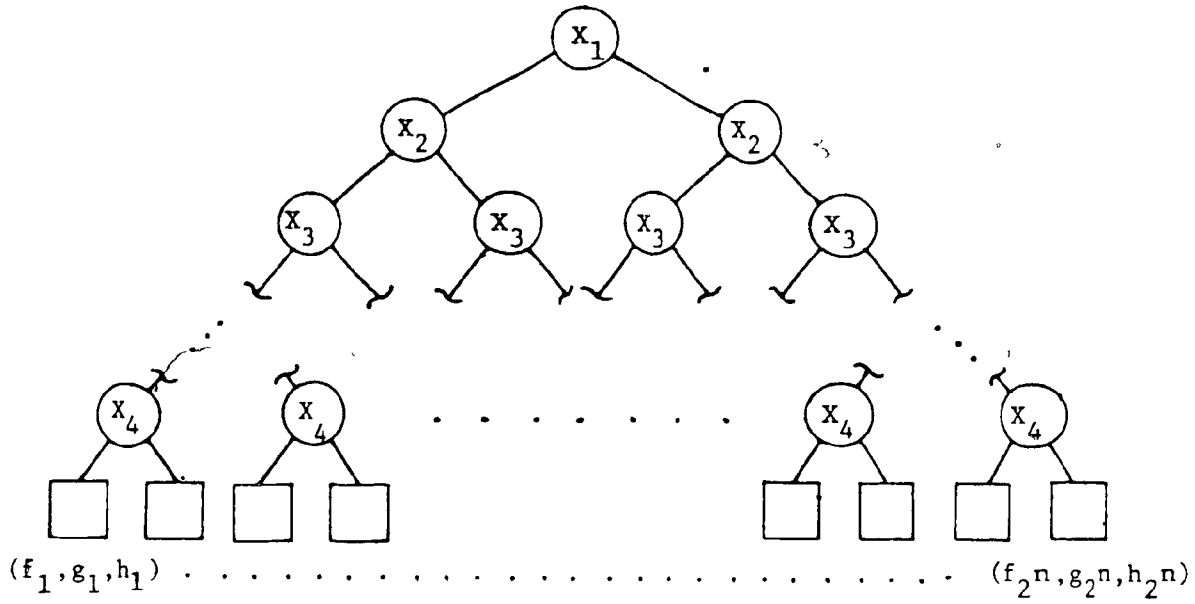
$$Q_1 = f_1(X_1, X_2, \ldots, X_n)$$
$$Q_2 = f_2(X_1, X_2, \ldots, X_n)$$
$$Q_3 = f_3(X_1, X_2, \ldots, X_n)$$

In the traditional Boolean approach these three functions would be computed serially by the evaluation of independent sum-of-product expressions describing each function. However, Q_1, Q_2 and Q_3 can be evaluated, in parallel, using the BD method, requiring only n steps or less to compute all three functions. Unlike Boolean methods which evaluate functions by arithmetic computation of algebraic equations, binary decision programming uses the input values as branching criteria, to direct the computation towards an output instruction in which the correct results have been previously stored. Conceptually, the data field of an output instruction can be made arbitrarily large to generate multiple functions in parallel. The procedure for programming multiple switching functions is defined by the truth table describing three control functions, F, G, and H, shown in figure 2.8a. Parallel output data for each combination of the input variables is formed simply by concatenating the three functions. Figure 2.8b illustrates the BD diagram for these functions.

Likewise, binary decision programming can evaluate switching functions that have more than two possible discrete modulo-two values, i.e., $Q_1 \in \{0,1,10,11,\ldots\}$. The application of the BD method to the

X_1	X_2	. . .	X_n	F	G	H
0	0	. . .	0	f_1	g_1	h_1
0	0	. . .	1	f_2	g_2	h_2
.	.		.			
.	.		.			
.	.		.			
1	1	. . .	1	f_2n	g_2n	h_2n

(a)



(b)

figure 2.8 BD representation of multi-function logic.
(a) Multi-function logic truth table.
(b) Equivalent BD diagram.

computation of multi-valued logic is another area which demonstrates the advantage of this method over its Boolean counterpart. The theoretical basis for doing so is also suggested by the series expansion of a Boolean switching function. Recall that the complete expansion of a general function takes the form of equation 2.4.

$$r(X_1,\ldots,X_n) = \bar{X}_1\bar{X}_2\ldots\bar{X}_n \cdot f(0,0,\ldots,0) + \bar{X}_1\bar{X}_2\ldots X_n \cdot f(0,0,\ldots,1) + \ldots + X_1X_2\ldots X_n \cdot f(1,1,\ldots,1) \quad (2.4)$$

In conventional binary decision logic, f(0,0,...,0) to f(1,1,...,1) have only two values, 0 or 1. However, in multi-valued logic these can take on any multi-bit value within a defined range. The values of f(0,0,...,0) to f(1,1,...,1) are stored in the data fields of BD output instructions. Since, as previously described, the data field is arbitrarily large, multi-bit data is easily accomodated within this field. (The support of multi-valued logic represents a major difference between Boute's BD machine and the Holck model.)

2.2.4 Minimization of Binary Decision Programs. In his article on binary decision diagrams, Akers demonstrated that combinatorial logic expressed in binary decision form could be minimized in a manner analogous to Boolean logic reduction [Aker78,McC156]. An example from his work is described below.

Consider the BD diagram resulting from the function $f = \bar{A}B\bar{C} + AC$, figure 2.9a. Note that the output of the leftmost node at the C-level is 0 both when C is FALSE and when C is TRUE. Since the value of the output is independent of C, the node is safely pruned from the diagram and replaced by a single output, figure 2.9b. Likewise, the pair of

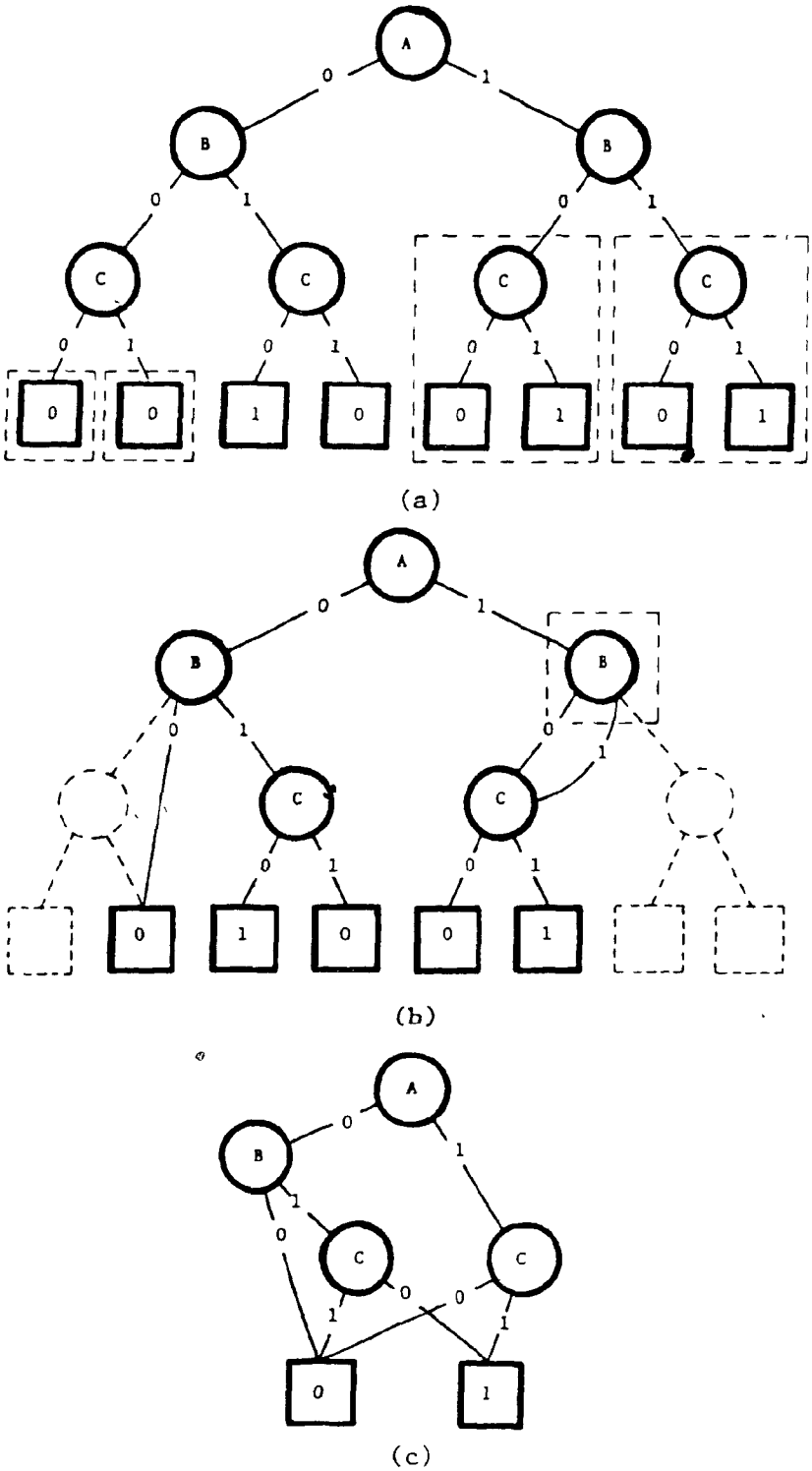


figure 2.9 BD logic minimization techniques.
(a) Detection of isomorphic program constructs.
(b) Elimination of isomorphic logic by "pruning".
(c) Trellising of resultant logic.

rightmost nodes at the C-level generate identical output combinations and so must imply the same logical proposition. These are combined into a single node, figure 2.9b. Note, however, that the two branches of the rightmost node at the B-level now lead to a single node, meaning that this path is independent of the value of B. In figure 2.9c, the superfluous node is removed and the diagram is further simplified by grouping all of the 0-outputs and 1-outputs together to form a so-called trellis structure.

The mathematical basis for the minimization of BD logic is derived by the consideration of equivalent Boolean equations obtained from Lemma 2.1. By this Lemma, the Boolean equation describing the leftmost node at the C-level of figure 2.9a is:

$$Q_c = \bar{C} \cdot 0 + C \cdot 0$$

But Q_c is also equal to 0 by:

$$Q_c = \bar{C} \cdot 0 + C \cdot 0 \qquad (2.11a)$$

$$= (\bar{C} + C) \cdot 0 \quad \text{(Distributive Property)} \qquad (2.11b)$$

$$= 1 \cdot 0 \quad \text{(Complementation Property)} \qquad (2.11c)$$

$$= 0$$

Likewise, the pair of rightmost nodes at the C-level of figure 2.9a are described by the equations:

$$Q_{c(left)} = \bar{C} \cdot 0 + C \cdot 1 \qquad (2.12)$$

and,

$$Q_{c(right)} = \bar{C} \cdot 0 + C \cdot 1 \qquad (2.13)$$

$$= Q_{c(left)} \qquad (2.13a)$$

Thus $Q_{c(left)} = Q_{c(right)}$. The B node is eliminated according to the relations:

$$Q_b = \bar{B} \cdot Q_{c(left)} + B \cdot Q_{c(right)} \quad (2.14a)$$

$$= \bar{B} \cdot Q_{c(left)} + B \cdot Q_{c(left)} \quad \text{(by 2.13a)} \quad (2.14b)$$

$$= (\bar{B} + B) \cdot Q_{c(left)} \quad \text{(Distributive Property)} \quad (2.14c)$$

$$= 1 \cdot Q_{c(left)} \quad \text{(Complementation Property)} \quad (2.14d)$$

$$= Q_{c(left)}$$

As can be seen from the above, the mathematical basis for the minimization of BD logic is a result of the Distributive and Complementation Properties of combinatorial logic.

The trellising exhibited in figure 2.9c is also a direct outcome of the Distributive Property. Consider the BD diagram of figure 2.10a. Note that the second and third outputs of the diagram both imply the value b. According to the method suggested above, these should be combined to form a single output node. By Lemma 2.1 this diagram is equivalently described by equations 2.15a, b, and c.

$$Q = \bar{X}_i \cdot Q_1 + X_i \cdot Q_2 \quad (2.15a)$$

$$Q_1 = \bar{X}_j \cdot a + X_j \cdot b \quad (2.15b)$$

$$Q_2 = \bar{X}_j \cdot b + X_j \cdot c \quad (2.15c)$$

By substitution of the values of Q_1 and Q_2 into 2.15a, we obtain:

$$Q = \bar{X}_i(\bar{X}_j a + X_j b) + X_i(\bar{X}_j b + X_j c) \quad (2.16a)$$

$$= \bar{X}_i\bar{X}_j a + \bar{X}_i X_j b + X_i\bar{X}_j b + X_i X_j c \quad \text{(Distributive Property)} \quad (2.16b)$$

$$= \bar{X}_i\bar{X}_j a + (\bar{X}_i X_j + X_i\bar{X}_j)b + X_i X_j c \quad \text{(Distributive Property)} \quad (2.16c)$$

The middle term of equation 2.16c represents the trellising of the two identical outputs nodes, figure 2.10b. Trellising results in more than one transfer path entering the b output node in the BD diagram. At this point, the diagram can no longer be called a binary tree and must instead be called a directed acyclic graph (DAG) [Stan80].

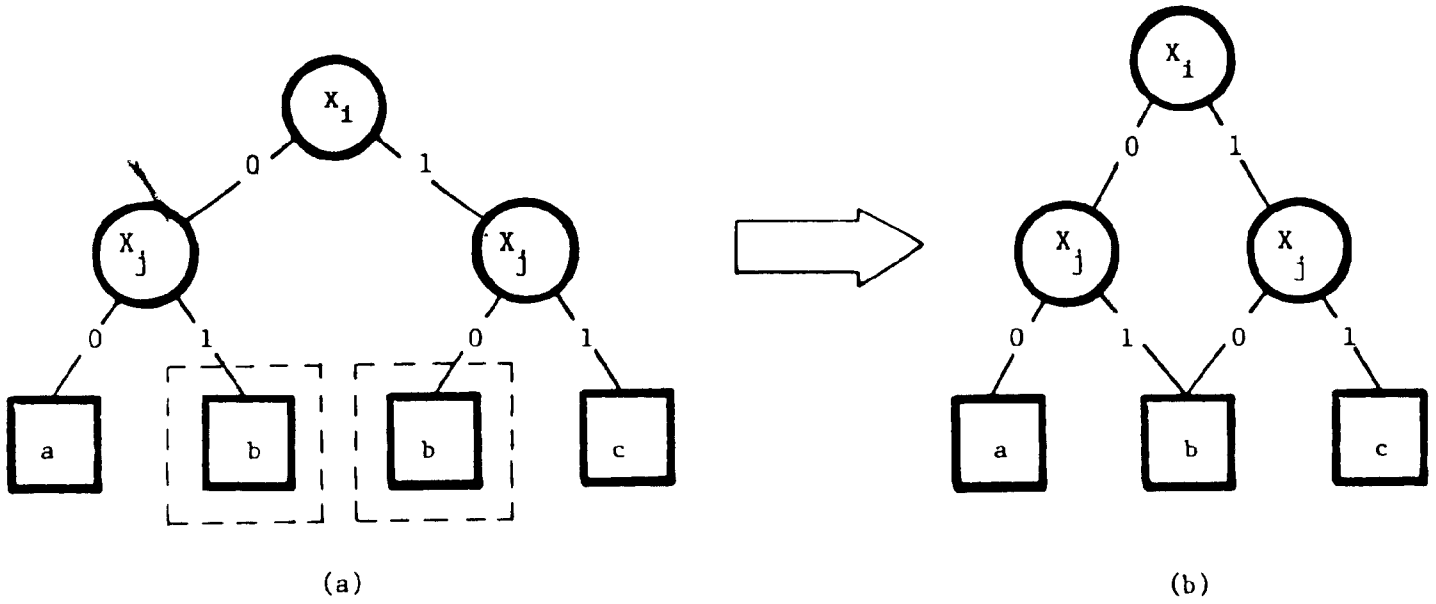


figure 2.10 Analysis of the trellis structure.
(a) Detection of isomorphic nodes.
(b) Trellising to combine isomorphic nodes.

The minimization principles discussed above can be applied to BD programs as well as BD diagrams. Consider the BD program in figure 2.2c. Note that the instructions labelled Q2 and Q6 are identical, as are instructions Q3 and Q5. Thus program size is reduced by combining these equivalent instructions, resulting in the program of figure 2.11a. Figure 2.11b shows the trellised BD diagram of the program.

The topic of BD program optimization is discussed in further detail in chapter 5, which considers the design of a BD program optimizing compiler for use in the mP/BD hybrid programmable controller.

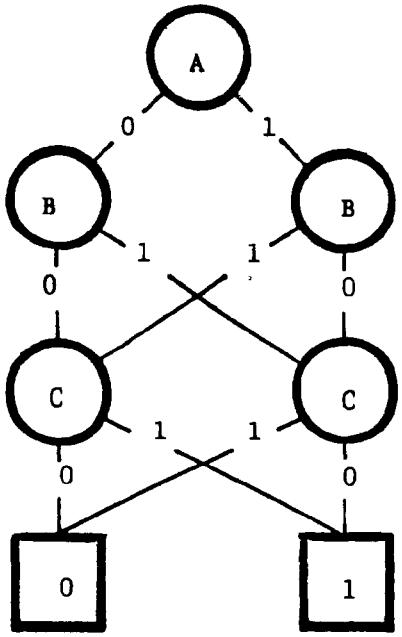
2.3 The Binary Decision Representation of Sequential Switching Functions

As described in section 2.1, a BD program is a sequence of transfer instructions in which the values of the input variables are used to control the program flow. Each instruction sequence terminates with an output instruction which contains the appropriate value of the control function. Since the discrete transfer paths are capable of storing input histories and since the program is capable of performing logical operations on the input data, then BD programs are, by definition, capable of describing sequential switching circuits. An example which illustrated the use of BD programming techniques for the representation of a sequential switching control function was presented in figure 2.3.

2.3.1 The Binary Decision-Sequential Circuit Analogy. Consider the sequential switching function:

```
Q    A  ;  Q1  ,  Q4
Q1   B  ;  Q2  ,  Q3
Q2   C  ;  0   ,  1
Q3   C  ;  1   ,  0
Q4   B  ;  Q3  ,  Q2
```

(a)



(b)

figure 2.11 Example of a minimized BD program.
(a) Minimized BD program for equation 2.1.
(b) BD diagram showing trellis structure.

$$Z = f(X_1, X_2, \ldots, X_n, S) \qquad (2.17)$$

The behaviour of this function is dependent upon both the current input and the particular input history which preceded the current input. Although the number of such histories may be large, they can often be grouped into a finite number of distinct classes called the internal states S, of the function. In general, a sufficient number of states must be defined to ensure that the appropriate output is produced for all valid input combinations.

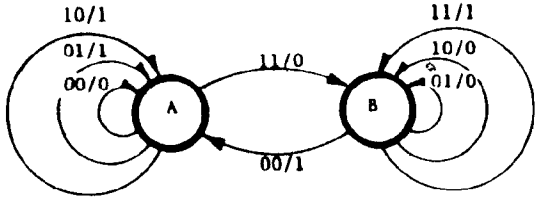
The logical operation of a sequential function is specified by mapping, for all states, the output and state transitions generated by the function for each input within a given state. This mapping is presented in various ways, e.g., in graphical form (State Diagram), in tabular form (State Table), or in equation form (State Output and State Transition functions). Figure 2.12 illustrates the form of each of these representations for a common sequential function, the binary serial adder.

This function adds two binary-coded numbers by the serial addition of digit pairs starting with the lowest order digits. Two distinct input states are required to distinguish between the different sum and carry outputs when carry = 0 and carry = 1, which result from the addition of identical input bits. Figure 2.12a tabulates the binary arithmetic results which are used to define the input-output mapping of the function.

The state diagram, figure 2.12b, represents the function as a directed graph in which the vertices are labelled with the state names and arcs are labelled with the input/output mappings corresponding to

S_I	C_{IN}	X_1	X_0	z	C_{OUT}	S_O
A	0	0	0	0	0	A
A	0	0	1	1	0	A
A	0	1	0	1	0	A
A	0	1	1	0	1	B
B	1	0	0	1	0	A
B	1	0	1	0	1	B
B	1	1	0	0	1	B
B	1	1	1	1	1	B

(a)



(b)

PRESENT STATE	NEXT STATE (X_1X_0)				CIRCUIT OUTPUT (X_1X_0)			
	00	01	10	11	00	01	10	11
A	A	A	A	B	0	1	1	0
B	A	B	B	B	1	0	0	1

(c)

$$z = S_I \oplus X_1 \oplus X_0$$

$$S_0 = S_I(X_1 + X_0) + \bar{S}_I X_1 X_0$$

(d)

figure 2.12 State descriptions of the binary serial adder. (a) The truth table, (b) state diagram, (c) state table and, (d) state output and transition functions.

each state. Let A designate the state of the adder when carry = 0 and B designate the state when carry = 1. The connectivity network indicates the appropriate state transition for each combination of current state and input. The state table, figure 2.12c, lists the next state and output of the function for each combination of current state and input. Finally, the combinatorial equations of the state output and state transition functions are shown in figure 2.12d.

All of these function descriptions are clearly equivalent since they all convey exactly the same information. It follows that any structure that correctly encodes this data is also a valid description of a sequential function. Consider for example the following generalization of Lee's binary decision instruction:

$$A \quad X_1X_0;\ (A,0),\ (A,1),\ (A,1),\ (B,0) \tag{2.18}$$

This instruction is read: in state A, if the variables X_1X_0 have the values 00 then transfer to state A and output 0, else if the variables have the values 01 then transfer to state A and output 1, else if the variables have the values 10 then transfer to state A and output 1, finally if the variables have the values 11 then transfer to state B and output 0.

Equation 2.18 is a sequential quaternary decision instruction that encodes the information in the first row of figure 2.12b. The trio of binary decision instructions:

$$A \quad X_1;\ A1,\ A2$$

$$A1 \quad X_0;\ (A,0),\ (A,1)$$

$$A2 \quad X_0;\ (A,1),\ (B,0)$$

is shown to be equivalent to equation 2.18 in Lemma 2.2.

Lemma 2.2: A quaternary decision instruction can be decomposed into an equivalent series of binary decision instructions.

Proof: Consider that the four pairs of (S_1,Z_1) terms in equation 2.18 are related to the switching variables X_1,X_0 by the combinatorial function:

$$A(X_1,X_0) = [\bar{X}_1\bar{X}_0 \cdot f(0,0) + \bar{X}_1X_0 \cdot f(0,1) + X_1\bar{X}_0 \cdot f(1,0) + X_1X_0 \cdot f(1,1)] \quad (2.19a)$$

where: $f(0,0) = (A,0)$ $\quad$ $f(1,0) = (A,1)$

$f(0,1) = (A,1)$ $\quad$ $f(1,1) = (B,0)$

Substituting the values of f(0,0), etc., into equation 2.19a and factoring terms using the Distributive Property yields equation 2.19b.

$$A(X_1,X_0) = \bar{X}_1[\bar{X}_0 \cdot (A,0) + X_0 \cdot (A,1)] + X_1[\bar{X}_0 \cdot (A,1) + X_0 \cdot (B,0)] \quad (2.19b)$$

Now let:

$$A1 = [\bar{X}_0 \cdot (A,0) + X_0 \cdot (A,1)]$$

and,

$$A2 = [\bar{X}_0 \cdot (A,1) + X_0 \cdot (B,0)]$$

Equation 2.19b is rewritten in terms of the single variable equations as follows:

$$A = \bar{X}_1 \cdot A1 + X_1 \cdot A2$$

$$A1 = \bar{X}_0 \cdot (A,0) + X_0 \cdot (A,1)$$

$$A2 = \bar{X}_0 \cdot (A,1) + X_0 \cdot (B,0)$$

However, by Lemma 2.1 this sequence of equations is logically equivalent to the binary decision program:

A X_1; A1, A2

A1 X_0; (A,0), (A,1)

A2 X_0; (A,1), (B,0)

The extension of this Lemma to higher order decision instructions is straightforward. □

A two-instruction quaternary decision program is clearly sufficient to represent the complete state of the function. Repeating the process for state B of the adder example produces the binary decision program:

B X_1; B1, B2

B1 X_0; (A,1), (B,0)

B2 X_0; (B,0), (B,1)

Together these two program segments completely define the sequential operation of the binary serial adder function.

This procedure may be applied, in general, to any sequential function which is adequately described by one of the standard logic representations. It effectively establishes the equivalence of sequential switching functions and binary decision programming. This equivalence is formally stated in Theorem 2.3.

Theorem 2.3: For every finitely-computable sequential switching function there exists an equivalent binary decision program.

Proof: Any finitely-computable, discrete sequential switching function can be expressed by means of a state table, state diagram or similar technique. A series of 2^n-ary decision instructions of the form:

S $X_1X_2...X_n$; (S_1,Z_1), (S_2,Z_2), ... , (S_2n,Z_2n)

where n is the number of external switching variables of the function,

may be obtained by inspection from the rows of the function's state table. These instructions map the combinatorial state transition and state output functions of the switching function for each of it's internal states. Taken as a whole, the instructions form a 2^n-ary decision program equivalent to the original switching function. By a generalization of Lemma 2.2, each of the 2^n-ary decision instructions may be decomposed into a set of 2^n-1 binary decision statements. Hence, the complete sequence of binary decision instructions, i.e., the binary decision program, must also be equivalent to the original sequential switching function. □

Historically, binary decision research has been predominantly confined to the use of BD for purely combinatorial applications. Consequently, very few results which detail the sequential properties of BD have been reported in the literature. Since, as shown above, binary decision programs are equivalent to conventional switching functions, it may be expected that the sequential properties of BD, e.g., logic minimization procedures, etc., will be closely analogous to those of conventional logic. The investigation of the sequential nature of BD is an important area of future research.

CHAPTER 3

THE DESIGN OF THE HYBRID mP/BD SYSTEM

3.0 System Overview

The design of the general purpose mP/BD hybrid programmable controller is shown in figure 1.7. This configuration consists of a microprocessor control subsystem integrated with a BD processor control subsystem by a communications interface. Each unit has its own program memory and field I/O sections. As a result both sequential automation (ON/OFF) and proportional control functions may be executed simultaneously by the hybrid controller.

The microprocessor's other function in the system is to service the operator's interface console (CRT, keyboard, printer, annunciator) and bulk memory units. The BD processor requires communication with the system peripherals to load new program segments from bulk storage, to send data to the CRT or to input from the keyboard. Since BD architecture is unsuited to the generation of communication protocols, the microprocessor handles I/O to the BD through the communications interface.

The hybrid controller, figure 1.12, makes use of microprocessor address, data and control busses to interconnect the system elements. BD interface modules are connected to the peripheral interface bus so that logically and electrically they appear to the mP as ordinary peripherals. This interface design enables several BD processors to be connected with a common mP supervisor. Vroomen [Zsom79] and Levi

[Levi84] have proposed PLC configurations consisting of a number of distributed, semi-independent BD controllers termed Programmable Control Units (PCU) or Intelligent Reflexive Interfaces (IRI) which are connected to and controlled by a shared supervisor. However, the hybrid mP/BD programmable controller presented in this thesis has only one BD unit.

System hardware design can be broken down into three sections, mP and bus architecture, BD processor hardware and the interface module. These elements are described in the following sections.

3.1 mP System Architecture

3.1.1 mP System Organization. The mP system consists of a central microcomputer plus operator interface devices and the process I/O interface. The microcomputer communicates with the system peripherals via a combined address, data and control bus.

The principal task of the mP in the PLC system is the execution of PID control algorithms. According to the task division scheme, ON/OFF control functions are reserved for the BD machine because of its superior logic processing characteristics. The mP is also responsible for ON/OFF and PID program compilation, program library maintenance, task-scheduling, and system communication functions.

Process I/O is achieved with conventional rack mounted I/O terminal and signal conditioning cards which are interfaced to the mP via the system bus. To enhance system reliability these are physically and logically independent from the BD processor's I/O cards.

3.1.2 mP Prototype Hardware Design. Hardware development was confined to the BD processor and a commercially available SWTPc (Southwest Technical Products Corp.) 6809 computer system furnished the mP half of the mP/BD hybrid programmable controller. The SWTPc system uses a Motorola MC6809 8/16 bit NMOS microprocessor on a standard SS-50 bus motherboard [Artw80]. The operator interface equipment is simulated with a Lear-Siegler ADM-5a CRT terminal with keyboard, attached to the SS-30 peripheral interface bus via a MP-S2 RS-232 communication interface adaptor card. The system also has a MF-68 dual 5-inch floppy disk drive for secondary storage and an Epson MX-80 printer for hardcopy output. These are attached to the SS-30 bus with a DC-3 disk controller card and a MP-LA parallel communication interface adaptor card, respectively. The SWTPc system normally runs the TSC (Technical Systems Consultants) Flex 9.0 disk operating system. However for PLC emulation, the system is controlled by the BD09 or BDBUG operating system as described in chapter 4.

The BD processor is interfaced to the SS-30 bus via a custom-built MC6821 PIA (Peripheral Interface Adaptor)-based card. This is described in section 3.3 of this chapter.

3.2 Binary Decision Processor Design

The BD processor's hardware design is based on Holck's stand-alone BD machine which was implemented as a demonstration unit in the DATAC laboratory in 1979 [Zsom79]. Holck's design has been enhanced to include a larger instruction set, more I/O points and a control and data

bus to the mP. (The Holck machine was itself an extension of Boute's design, figure 1.9, to incorporate parallel, multi-bit output.)

3.2.1 Physical Description. The BD processor prototype, figure 3.1, is 35 cm. long by 19 cm. wide. It comprises the expanded Holck BD processor, a local programming and control console consisting of 24 SPST switches and 24 LED status lights, 12 input points, 2 output banks, a 50-conductor cable to the mP/BD interface, an auto/manual mode switch and an auto-mode reset button.

The circuit is constructed with wire-wrapped, discrete TTL logic components and requires an external 5-volt regulated power supply. A 1.00000 MHz crystal is provided to drive the system clock.

Programming and control of the BD processor is performed automatically by the interfaced mP or manually, by an operator, via the switch console. The AUTO/MANUAL switch, located at the left-rear of the circuit board, selects the control mode. In manual mode, the BD processor is isolated from the interface and is controlled directly with the switch console located at the right-front of the board. The console consists of 8 control switches labelled with function descriptions and 16 programming or data entry switches labelled D0 to D15. In manual programming mode, these 16 switches are connected to the data bus and permit the 256x16 RAM memory to be loaded with user programs. In the program counter preset mode, D0 to D7 are connected to the address bus for initialization of the PC to the starting address of the control program. These switches are disabled when the BD processor is operated remotely via the mP interface.

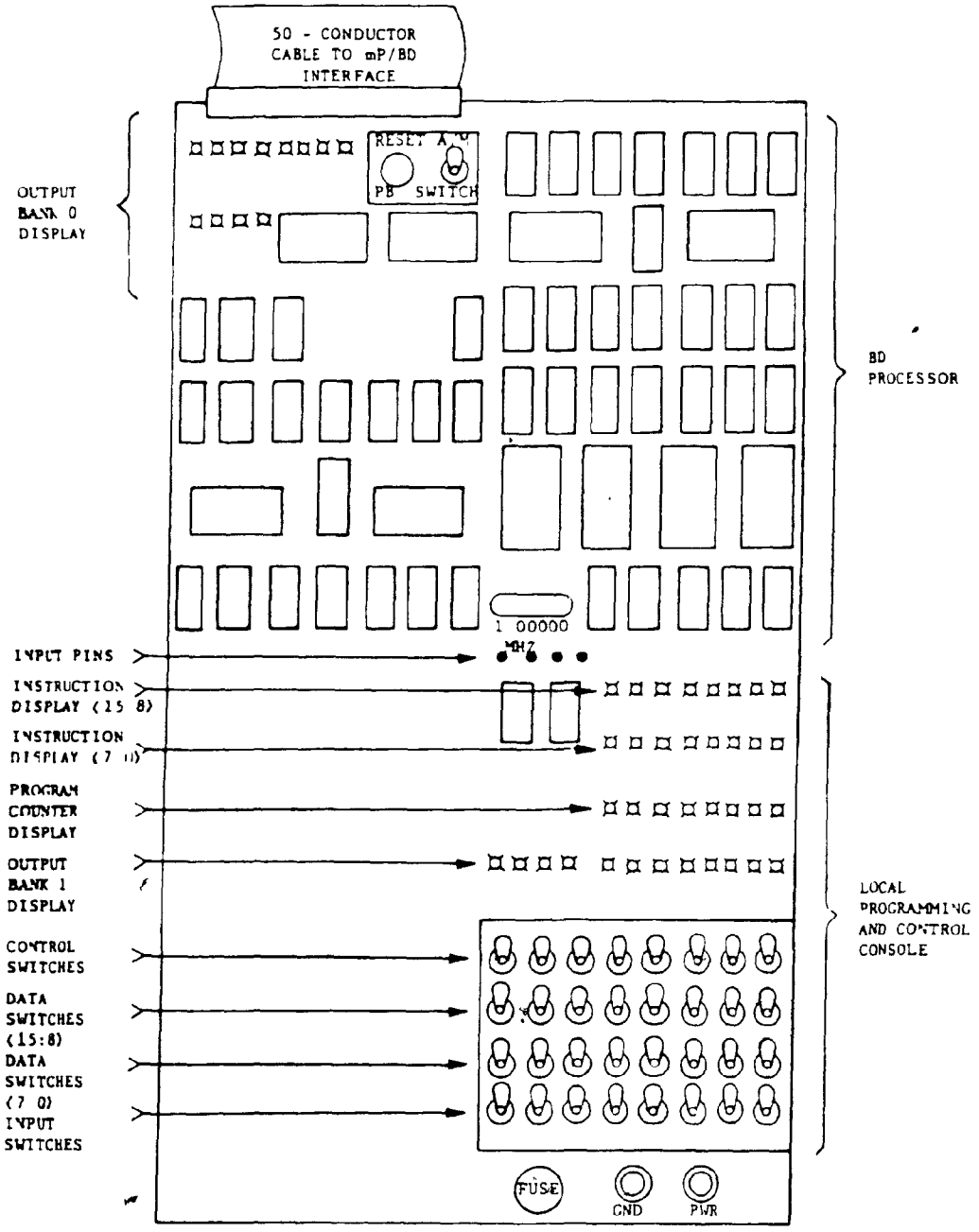


figure 3.1 BD machine prototype layout.

The contents of the data and address busses are continuously displayed on the console by 3 rows of LED status lights, located behind the switch panel. This display operates in both auto and manual control modes.

In the auto mode, programming and control signals are sent to the BD from the mP interface via the cable located at the left-rear of the board. Immediately in front of the cable is the AUTO-MODE RESET pushbutton. This provides the operator with a means of halting the BD hardware when the machine is in the automatic mode and it is inconvenient or impossible to reset the BD from the operator's keyboard.

Only twelve field inputs are implemented in the BD prototype. These are divided into four wired inputs and eight simulated inputs driven by SPST switches labelled I0 to I3 and J0 to J3, located on the switch console at the front-right. The inputs are active during both control modes.

Similarly, only two banks of field outputs are currently implemented. Of these, four bits are available for external control and the remaining twenty bits are simulated on LED lights. The LED indicator lights for Bank 0 are located at the left rear of the circuit board while those for Bank 1 are located immediately behind the console switch panel.

A terminal block is located on the right-front edge of the circuit board to bring in 5-volts from an external voltage-regulated power supply.

3.2.2 Programmer's Model. Instruction Set Processor (ISP) notation is used below to describe the elements of the programmer's model. ISP is a uniform symbolic language used to conveniently describe the organization and operation of computers. The notation is fully explained in Appendix II.

The programmer's model of the BD processor consists of:

1) Program Counter. The 8-bit program counter\PC<7.0> contains the address of the current executable instruction in the 256 location, 16-bit wide program memory\Mp[255:0]<15:0>. During operation the contents of the PC are either incremented or preset to a branch address specified in the current instruction\IN<15:0> as an unconditional function of the operation code and a conditional function of the selected input variable.

2) Input Variable Register. The 64-bit input variable register \IV<63:0> is the main process input channel to the BD processor. All 64 bits are directly addressable from either of two input instructions.

3) Output Registers. The output section consists of sixteen 24-bit output registers\OR[15:0]<11:0> each with twelve independent signals and their logical complements. The programmer outputs data to the process, optionally in groups of four bits or twelve bits, by using a "short" output or a "long" output instruction, respectively. The output data is switched to the appropriate output register by the output bank register.

4) Output Bank Register. The 4-bit output bank register\OB<3:0> controls the switching of the output data to one of the sixteen output

registers. Output to independent process loops under the control of the same BD processor can be kept safely separate in this way. An instruction to select an output bank is usually placed at the beginning of each BD program.

3.2.3 Instruction Set. In principle, BD programs contain only two operating instructions, input-test-branch and output. For the mP/BD hybrid programmable controller the instruction set is enhanced to support multi-programming, i.e., one or more BD programs controlling one or more process loops, and BD-mP communication. Eight instructions have been defined and a further eight operation codes are available for future expansion. The presently defined instructions are summarized in table 3.1. Bit numbers refer to fields within the IN. The instruction format is shown in figure 3.2.

1) INPUT-TEST-BRANCH. The INPUT instruction implements the IF-THEN-ELSE branching logic fundamental to the BD concept. The instruction causes the Logic Control Unit to input the selected variable (X) from the input variable register, test it and to update the program counter to the memory address of the next sequential program instruction. The instruction format comprises the input variable address and the IF condition branch address. On the ELSE condition the processor automatically branches to the default instruction located in the next consecutive memory location. No loss of generality is incurred, however, since two INPUT instructions have been implemented to allow branching on either polarity of the input variable.

OPERATION CODE	INSTRUCTION	FUNCTION
00XX 01XX	INPUT AND TEST	INPUT LINE IS SELECTED BY IN<13.8>. V = X ⊕ IN<14>. IF V=0, GO TO PC+1. IF V=1, JUMP TO IN<7 0>.
1000	OUTPUT LONG	LOAD APPROPRIATE OUTPUT BANK WITH IN<11 0>. GO TO PC+1
1100	OUTPUT SHORT	LOAD APPROPRIATE OUTPUT BANK WITH IN<11 8>. JUMP TO IN<7 0>
1110	OUTPUT BANK SELECT	LOAD OUTPUT BANK REGISTER WITH IN<11 8>. JUMP TO IN<7 0>
1010	END OF PROGRAM	BD CLOCK IS DISABLED IF STOP FLAG IS HIGH. ELSE GO TO PC+1
1101	INTERRUPT	BD INTERRUPT SIGNAL TO mP IS GENERATED. JUMP TO IN<7 0>
1111	JUMP (UNCONDITIONALLY)	JUMP TO IN<7.0>

table 3.1 BD instruction set.

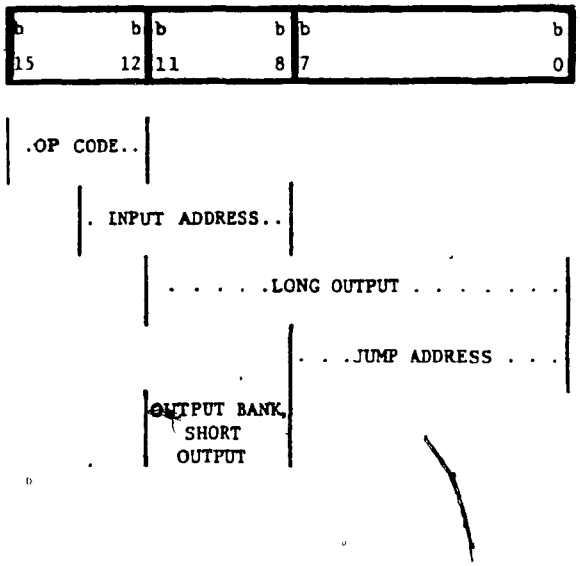


figure 3.2 Instruction word format.

IN<15:14> comprise the operation code (op-code) field, i.e., OP<1:0>:=IN<15:14>. Codes OP<1:0>=00 and 01 define the two types of INPUT. The input variable address contained in IN<13:8> selects one of the 64 input variables. Thus the selected variable is X:=IV<IN<13:8>>. X is EXCLUSIVE-ORed with IN<14> to obtain the next branch address, as follows. If (IN<14> $\oplus$ X) = 1 then the next instruction's address is contained in IN<7:0>. Otherwise if (IN<14> $\oplus$ X) = 0 then the next instruction's address is PC + 1. The op-code field is restricted to two bits by the six-bit input variable address field required to address 64 independent inputs. A proposed modification to the present design is to configure the input section as sixteen banks of sixteen directly addressable inputs. This would reduce the width of the input address field to four bits, but would require a four-bit input bank register similar to the output bank register configuration.

2) OUTPUT. The OUTPUT instruction transfers the computed value of the switching function describing the control procedure to the appropriate output bank via the output register. Two types of instructions are defined, one that outputs a full set of 12 parallel bits to the output register, and a second that outputs only 4 bits. If the process is controlled with four bits or less the so-called SHORT OUTPUT instruction may be used. This saves program memory, since the next sequential instruction's address is found in the lower eight bits of the SHORT OUTPUT instruction. If the LONG OUTPUT is used, the BD processor selects the instruction at the next address.

The instruction is invoked by OP<3:0> = 1000 or 1100. If OP<2>=0 then it is a LONG OUTPUT instruction and IN<11:0> contains literal

output variables\OV<11:0>, which are transferred to the 12 bits of the output register addressed by the output bank register, i.e., OV<11:0> -> OR[OB<3:0>]<11:0>. OP<2>=1 is a SHORT OUTPUT instruction. IN<11:8> contains four literal output variables, OV<11:8>, and IN<7:0> contains the next instruction's address, i.e., OV<11:8> -> OR[OB<3:0>]<11:8>. The lower eight bits of the output register are unchanged by a SHORT OUTPUT instruction.

3) JUMP (UNCONDITIONALLY). This instruction permits the programmer to prevent a programming impasse when an instruction defaults to a next address from which the program flow does not logically continue. Such situations might occur when a LONG OUTPUT instruction requires a supplementary jump or when both results of an INPUT instruction require a jump to different sections of the program. The JUMP instruction is invoked with the op-code OP<3:0>=1111. The JUMP address is contained in IN<7:0>. IN<11:8> is undefined.

4) INTERRUPT. The INTERRUPT instruction is used to initiate BD to mP communication. A BD interrupt may be programmed if a process variable has exceeded its specified limit, a major disturbance has caused an equipment trip, or a "benchmark" event has occurred which must be annunciated. The op-code of an INTERRUPT instruction is OP<3:0>=1101 and IN<7:0> contains the address of the next instruction, normally an END-OF-PROGRAM instruction.

5) END-OF-PROGRAM (EOP). The EOP instruction marks the logical end of each BD program. The BD machine can be halted only after execution of

an EOP to prevent possible loss of process control due to incorrect or incomplete output. The EOP instruction is invoked with OP<3:0> = 1010 and IN<11:0> contains program and possible interrupt information. Program execution continues at the next consecutive instruction if the BD machine has not been halted.

6) OUTPUT BANK SELECT. The OUTPUT BANK SELECT instruction latches the address of the output bank serving the next program segment into the output bank register. This instruction is normally used when several process control programs are running concurrently to separate outputs from individual programs. The instruction is invoked by OP<3:0>=1110. IN<11:8> contains the address of the desired output bank and IN<7:0> contains the address of the next sequential program instruction.

3.2.4 Hardware Design. A block diagram of the binary decision processor is shown in figure 3.3. The machine consists of five major sections; the control section, program memory, system clock, field input and field output sections. The design of each of these sections is presented below.

3.2.4.1 Control Section. The control section consists of the Program Counter, the Logic Control Unit, the Clock-Interrupt Circuit and the Auto/Manual Control Interface.

1) Program Counter. The 8-bit program counter (PC), figure 3.4, is implemented with a pair of 74161 4-bit synchronous presettable counters arranged to form a ripple-counting circuit. The preset inputs are connected to IN<7:0> of the instruction register. Preset data is

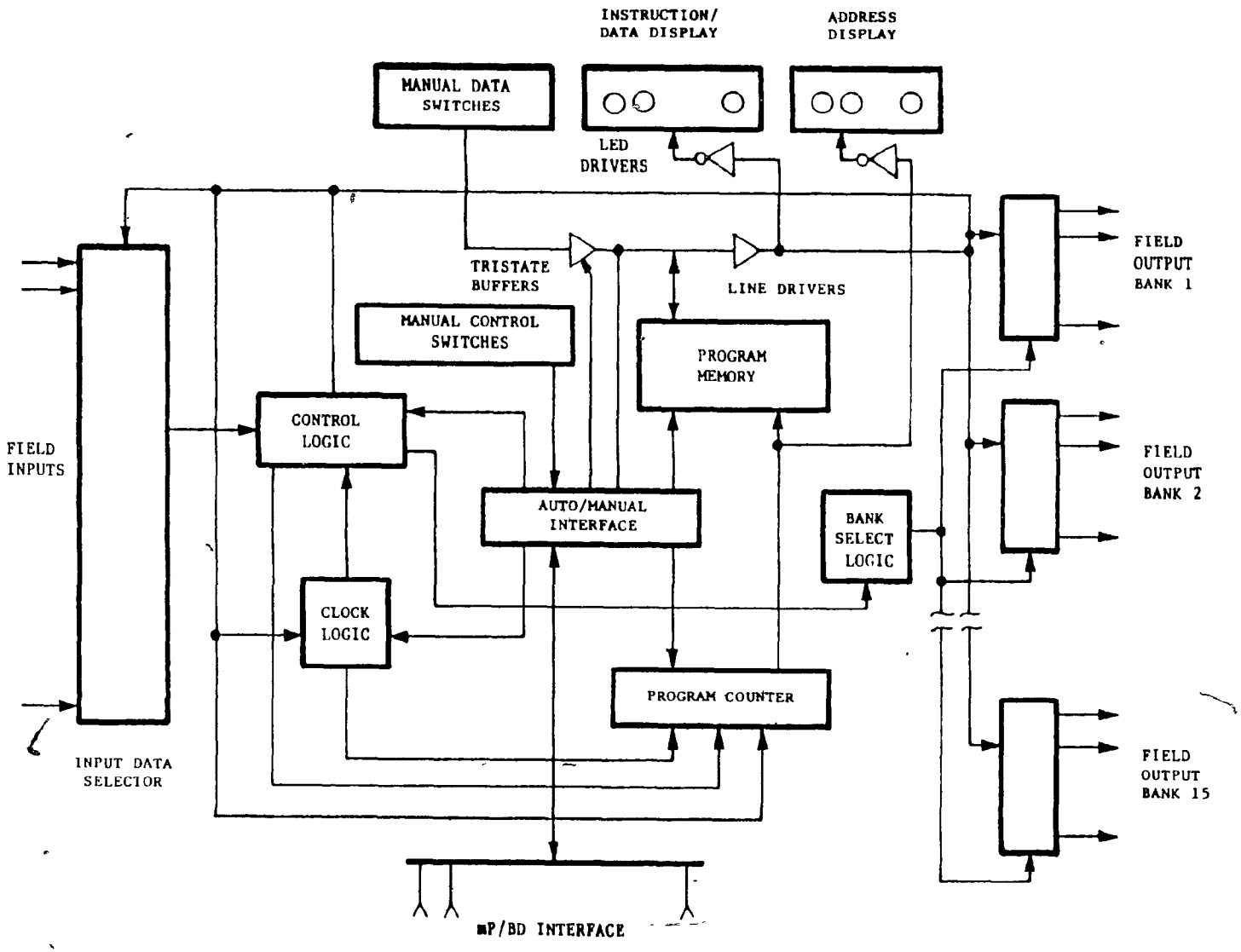


Figure 3.3 BD processor block diagram.

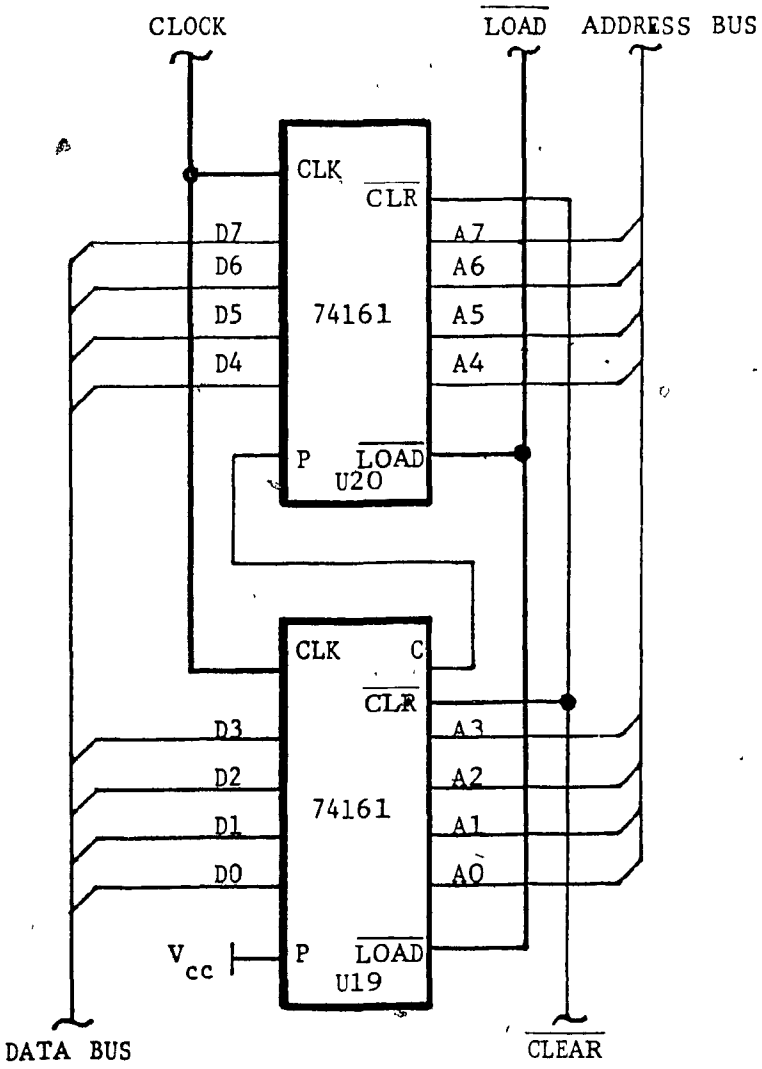


figure 3.4 Program Counter circuit diagram.

latched into the 74161's by a low signal on the $\overline{\text{LOAD}}$ line, which is connected to the branch decision logic of the Logic Control Unit, coupled with a low-to-high transition of the system clock. The PC increments if the $\overline{\text{LOAD}}$ line is high during a low-to-high clock transition. The PC is unconditionally cleared by a high-low-high strobe of the $\overline{\text{CLEAR}}$ line. Outputs of the PC are connected to the system address bus.

2) Logic Control Unit. The Logic Control Unit (LCU), figure 3.5, consists of an op-code decoder and a branch decision circuit which is connected to the $\overline{\text{LOAD}}$ line of the PC. Input to the LCU includes IN<15:12>, the system clock and the selected input variable X. The LCU also has a strobe generating circuit which synchronizes the action of some of the instructions with the low phase of the clock cycle. The operation of the LCU is described as follows.

The OP code, IN<15:12>, is presented to the LCU at the start of each machine cycle. Simultaneously one input variable, selected from the field input register by IN<13:8>, appears on line X. These two actions always occur regardless of which type of instruction is executed. The $\overline{\text{LOAD}}$ output of the branch decision circuit is enabled by either (D15·D14) or conditionally by [(D14 $\oplus$ X)·$\overline{\text{D15}}$]. Thus all instructions with OP<3:2>=11 such as SHORT OUTPUT (1100), BANK SWITCH (1110), INTERRUPT (1101) and JUMP (1111) invoke unconditional branching to IN<7:0> in the next machine cycle. (Recall that all BD program instructions take just one machine cycle.) The two INPUT instructions invoke branching only if the function (D14 $\oplus$ X) is TRUE. Since the input

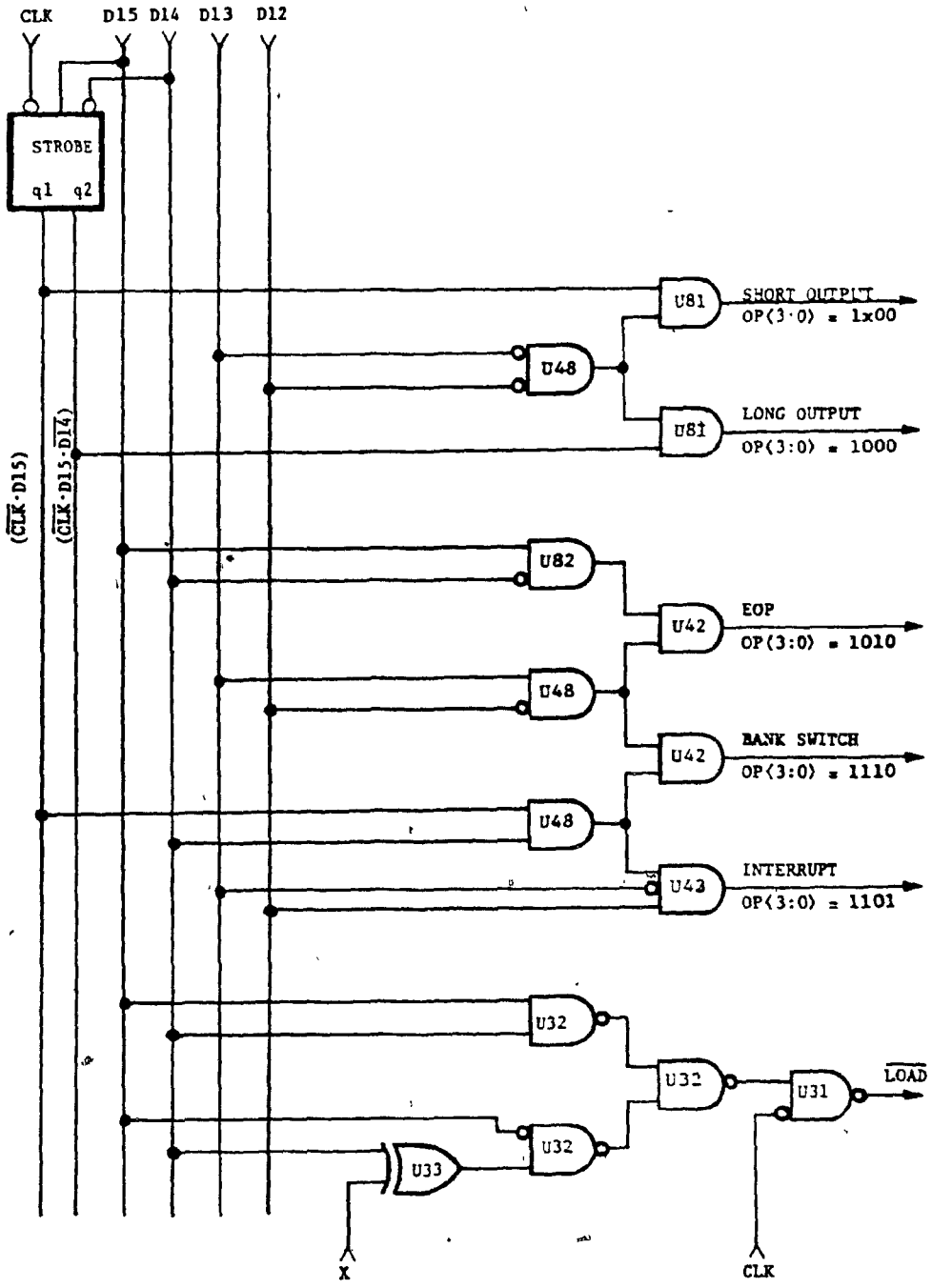


figure 3.5 Logic Control Unit circuit diagram.

section always interprets IN<13:8> as an input address even with non-input instruction types, the (D14 ⊕ X) circuit is interlocked with $\overline{D15}$ to inhibit unwanted branches in these circumstances. At the present time only two OP<3:0>=0XXX instructions have been implemented, the INPUT-TEST-BRANCH pair. The branch decision circuit would have to be augmented to decode all four bits if the remaining 0XXX instructions were to be allocated in the future. The $\overline{LOAD}$ output is interlocked with the system clock to inhibit this signal during the first half of the machine cycle. This ensures that branching data is stable for at least one-half cycle before the PC can be preset and circumvents any timing problems caused by unsynchronized operation of the $\overline{LOAD}$ line.

The strobe generating circuit consists of a 74123 dual monostable multivibrator which produces two short duration strobe signals, q_1 and q_2, used to latch data into the output register. The SHORT OUTPUT instruction, decoded by $(q_1 \cdot \overline{D13} \cdot \overline{D12})$, latches four upper bits of output data from IN<11:8> into OR[OB<3:0>]<11:8>. The LONG OUTPUT instruction, decoded by $(q_2 \cdot \overline{D13} \cdot \overline{D12})$, latches eight bits of output data from IN<7:0> into OR[OB<3:0>]<7:0>. But since q_1 is generated by (CLK·D15) and q_2 by $(CLK \cdot D15 \cdot \overline{D14})$ then q_1 is produced simultaneously with q_2 in all instances when q_2 is generated. Likewise $(q_1 \cdot \overline{D13} \cdot \overline{D12})$ is additionally decoded in all cases when $(q_2 \cdot \overline{D13} \cdot \overline{D12})$ is generated. Thus a LONG OUTPUT operation code also enables the SHORT OUTPUT signal resulting in all twelve output bits being latched in parallel. The actual data output operation is synchronized with the high-to-low transition of the system clock by interlocking q_1 and q_2 with $\overline{CLK}$ to ensure that IN<11:0> is stable before latching it into the output register.

The EOP signal is decoded, without strobe, as ($D15 \cdot \overline{D14} \cdot D13 \cdot \overline{D12}$). The EOP signal is used by the Clock-Interrupt Circuit to permit the BD machine to be halted only at the end of a control program. The operation of this interlock circuit is discussed later.

The BANK SWITCH signal, decoded by ($q_1 \cdot D14 \cdot D13 \cdot \overline{D12}$), causes the output bank address in IN<11:8> to be latched into the output bank register. The data transfer is synchronized with the high-to-low transition of the system clock by q_1 to ensure data stability.

The INTERRUPT signal, decoded by ($q_1 \cdot D14 \cdot \overline{D13} \cdot D12$), is also used by the Clock-Interrupt Circuit.

The JUMP instruction is implicitly decoded by the branch decision circuit since OP<3:2>=11. A JUMP signal does not have to be decoded by the Logic Control Unit since no other action is required.

The operation of the Logic Control Unit is described by the timing diagram shown in figure 3.6.

3) Clock-Interrupt Circuit. The Clock-Interrupt Circuit, figure 3.7, provides an interlock signal to the system clock. BD processor operation is halted by removing the clock signal from the Logic Control Unit. The system clock can be inhibited in four ways; by failure of the 1 MHz clock, by manual reset, by BD generated interrupt, and by mP generated interrupt.

The microprocessor interrupts the BD clock by raising the STOP flag causing U81-3 to fall. This signal propagates through U82 and becomes a low input to U85, a 7474 D-type flip-flop. The next EOP instruction clocks U85 causing the output to fall, which inhibits the system clock through U82 and causes U82-11, the BD-CLOCK-ON flag, to fall as well.

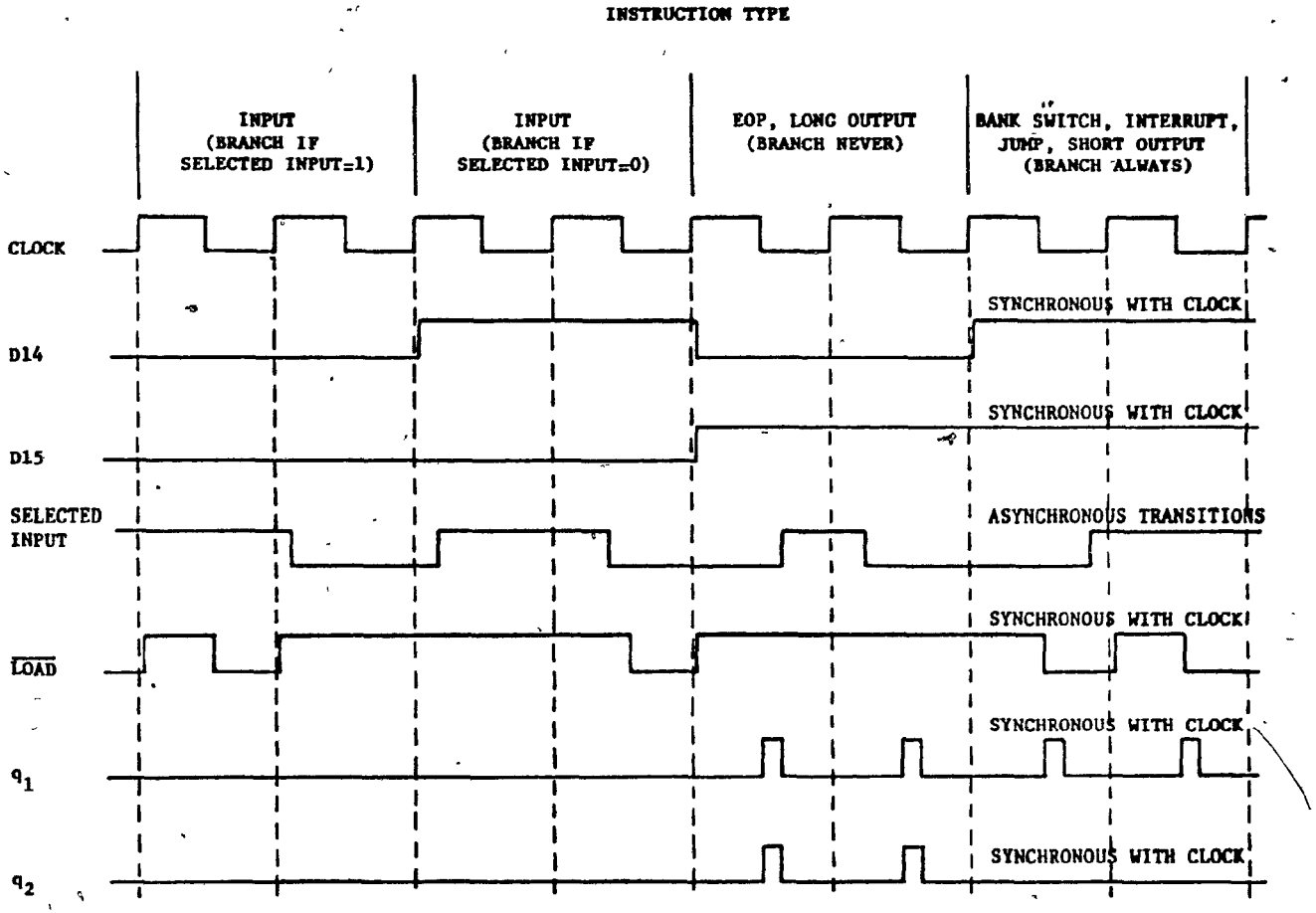


figure 3.6 Logic Control Unit timing diagram.

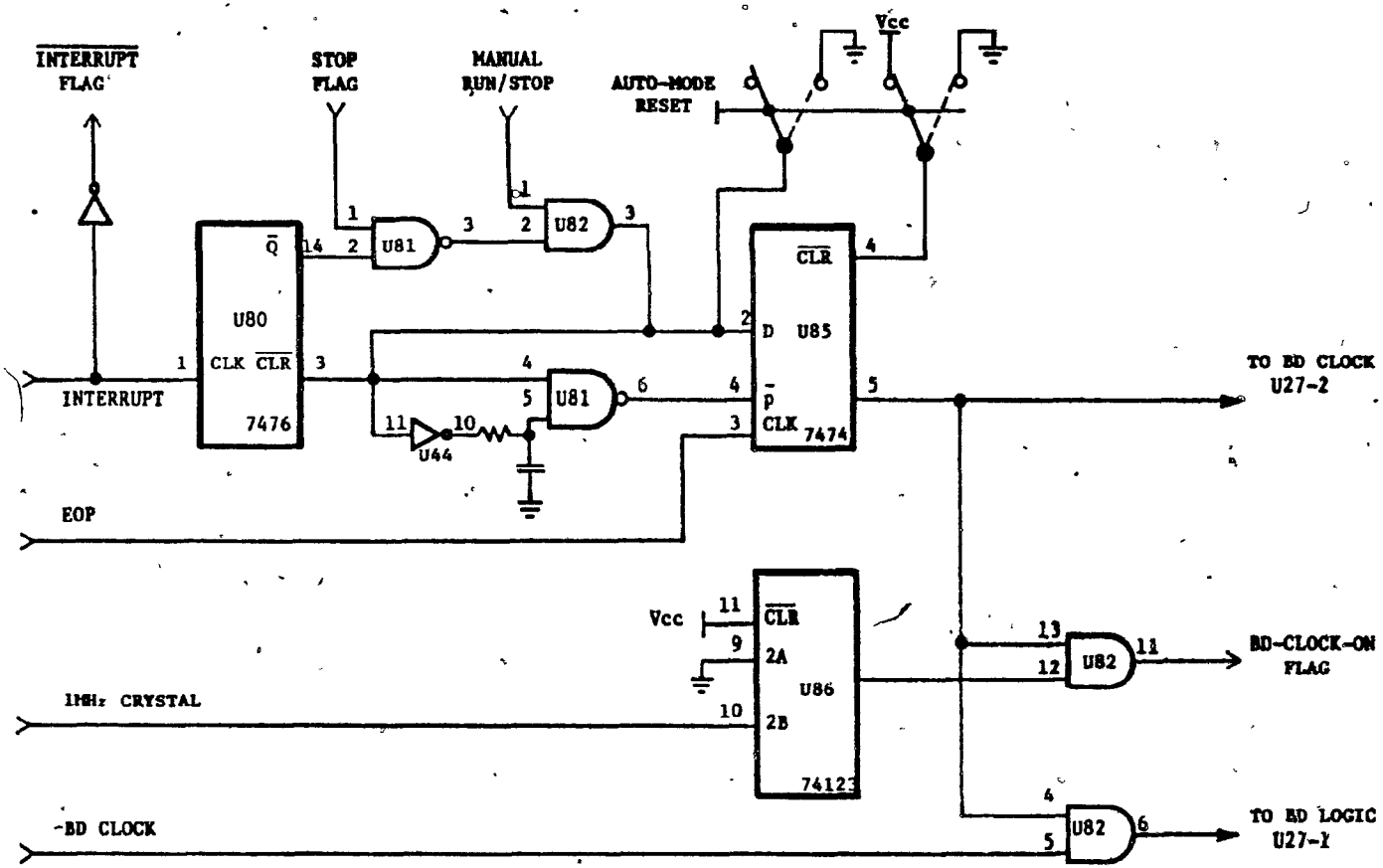


figure 3.7 Clock-Interrupt Circuit diagram.

This latter signal is fed back to the mP to confirm that the BD processor has halted.

At the end of the mP to BD communication the mP restarts the BD processor by lowering the STOP flag causing U81-3 to rise. The D flip-flop, U85, is set by a short pulse to the $\overline{P}$ input produced by a pulse forming circuit consisting of U44 and U81. This circuit operates as follows. During stable operation U82-3 is fed directly and in inverted form to U81. Thus U81-6 is normally high. When the STOP flag is lowered, a low-to-high transition occurs at U82-3. This transition propagates immediately to the first input of the two-input NAND gate, U81-4, but the RC network discharge constant causes a short delay in the transition reaching the second input. Momentarily both inputs are high which results in a brief low pulse at U81-6. The operation of this circuit is illustrated by the timing diagram of figure 3.8.

The BD interrupts itself indirectly through the mP. When an INTERRUPT instruction is executed by the BD, the decoded signal strobes the mP $\overline{\text{INTERRUPT}}$ flag and clocks a 7476 J-K flip-flop set to toggle mode. The microprocessor responds by setting the STOP flag. As before we want the BD system to halt on an EOP instruction. In the event that several programs are being concurrently executed by the BD processor, we want the system to halt on the EOP of the program which generated the interrupt. Since it is unlikely that the mP can respond to the BD interrupt before the interrupting program's EOP is executed, the STOP flag is masked by the J-K flip-flop acting upon U81-2 until the interrupt occurs a second time. This toggles the flip-flop once again and enables the STOP flag. The mP reads the BD data bus to determine

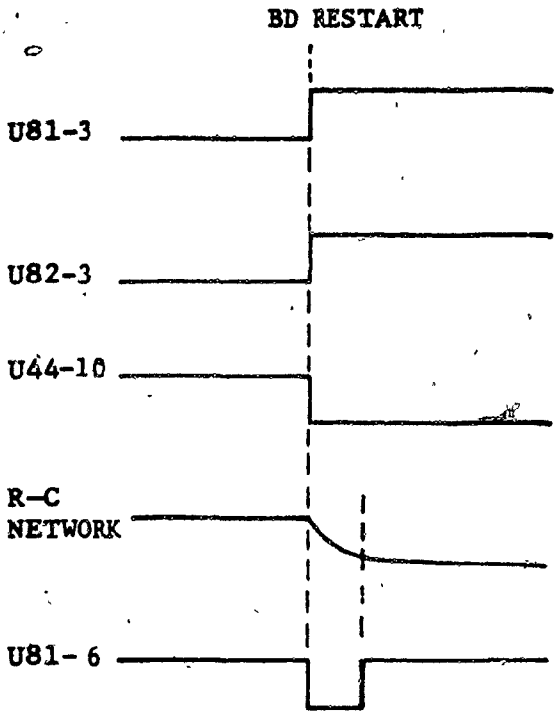


figure 3.8 Clock restart circuit timing diagram.

the reason for the interrupt after it has confirmed that the BD clock is halted. At the end of the BD to mP communication the micro restarts the BD processor as before.

The AUTO-MODE RESET contact is a momentary contact pushbutton that conditionally resets the output of U85 by strobing the $\overline{CLR}$ input. In auto-mode, the mP recognizes that the BD-CLOCK-ON signal has reset and responds by raising the STOP flag, which inhibits the circuit after the pushbutton is released. This reset contact is provided as a safety feature as it allows the operator to halt the BD processor locally while the system is in automatic mode. The operation of this circuit is independent of an EOP instruction and so provides a way to halt the BD machine if a program fails.

In manual mode the clock signal is inhibited by a manual RUN/STOP switch located on the BD switch console.

Failure of the 1 MHz clock is also detected by the Clock-Interrupt Circuit. The clock output is fed to U86, a retriggerable monostable multivibrator, which generates a high output as long as the clock signal is present. Upon clock failure, the multivibrator times out and lowers the BD-CLOCK-ON flag to the mP via U82-12. The mP recognizes the failure and takes appropriate action.

4) Auto/Manual Control Interface. The Auto/Manual Control Interface, figure 3.9, interfaces the two parallel sets of control signals provided by the mP supervisor and manual control console switches. The circuit is controlled by the AUTO/MANUAL (A/M) selector switch. When the switch is in the AUTO position, all manual control switches are locked out and

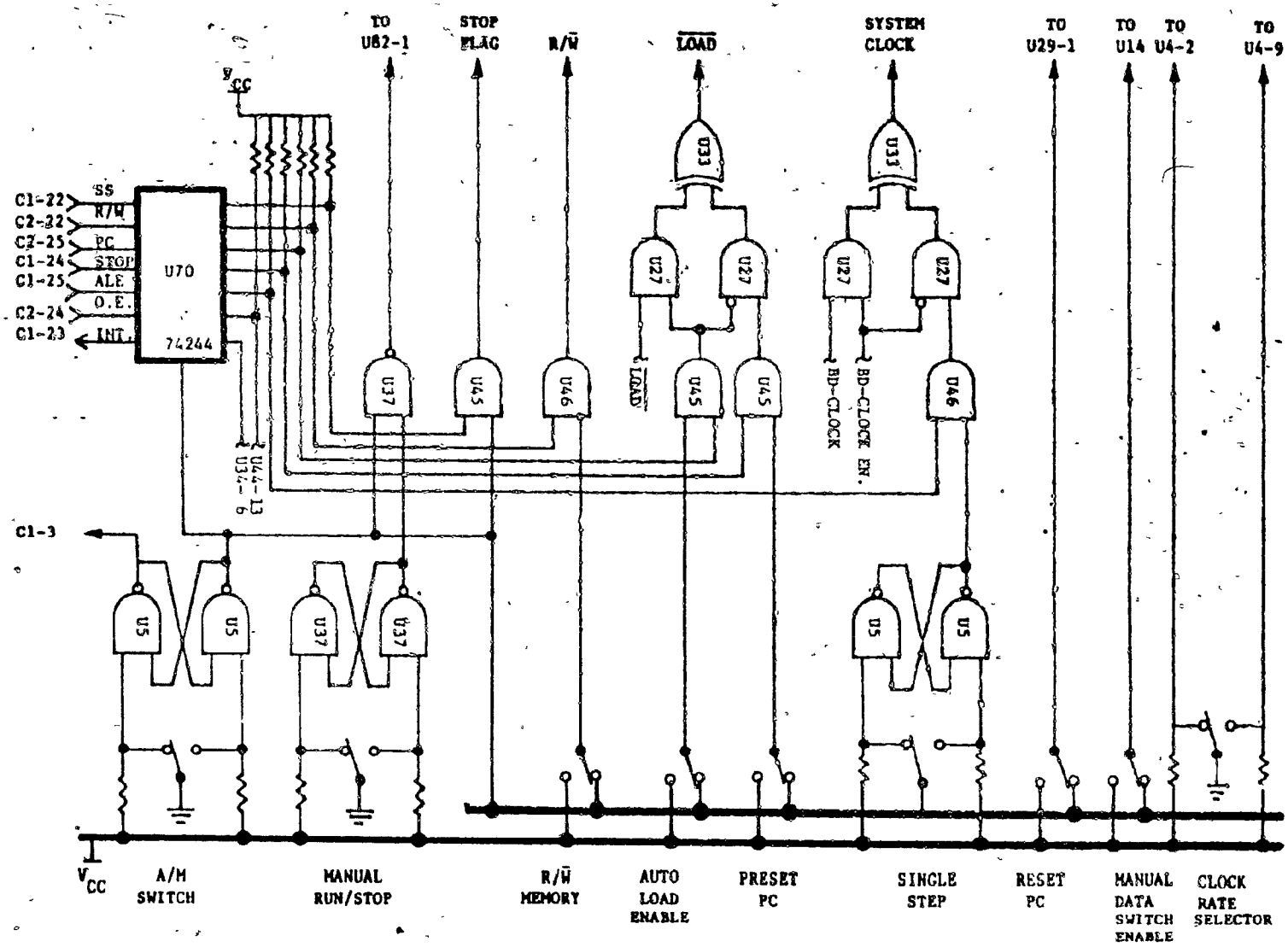


figure 3.9 Auto/Manual Control Interface circuit diagram.

cannot affect the BD processor. Likewise when the AUTO/MANUAL switch is in the MANUAL position, the mP interface circuits are locked-out.

U70 buffers the auto-mode control signals coming from the mP/BD interface module. The outputs of U70 are ANDed with the manual control switches to determine which set of control signals operate the BD processor. The BD-side outputs of the tri-state buffer, U70, are wired with pull-up resistors that pull the lines high when the buffer is deactivated by the AUTO/MANUAL switch. This prevents the floating outputs of U70 from possibly locking-out the manual control switches while in manual mode. Likewise the manual switches default to the high output state when the interface is in the auto-mode. This is achieved by wiring the ground bus of the switches to the AUTO/MANUAL switch. In manual-mode the bus is pulled low, thus enabling normal switch operation. In auto-mode, the bus is pulled high and the manual switches all output the high state regardless of the switch position since both inputs to the switches are high.

Control switch functions are described as follows.

i) MANUAL RUN/STOP. The MANUAL R/S switch is connected to U82-1 in the Clock-Interrupt Circuit, previously described. Through this latter circuit, the MANUAL R/S switch disables the BD Clock to halt program execution after the next EOP instruction. The counterpart to this switch in the mP control bus is the STOP signal which is also fed to the Clock-Interrupt Circuit.

ii) SINGLE STEP PC. The SINGLE STEP switch and flip-flop circuit provide a manual clock input to the BD processor when the system clock is inhibited by the Clock-Interrupt Circuit. The switch is

automatically enabled when the system clock is inhibited. The SINGLE STEP switch is then able to drive the system clock through U33, an XOR gate. This switch is used to advance the PC during program loading and verification. It is duplicated in the mP control bus for remote manipulation of the PC.

iii) RESET PC. The RESET PC switch is directly connected to the CLR inputs of the PC. It enables the operator to reset the program counter. This switch is not duplicated in the mP control bus since the PC is cleared by loading it with 0's in the automatic-mode protocol.

iv) PRESET PC. The PRESET PC switch enables manual presetting of the program counter by D0 to D7 of the manual switch console or from the mP control interface. The output of this switch is connected to the $\overline{\text{LOAD}}$ line of the program counter via U27-3 and U33-3 and is enabled by the AUTO LOAD ENABLE (ALE) switch. The operation of these switches is as follows. When ALE is high the system's $\overline{\text{LOAD}}$ signal, produced by the branch decision circuit of the Logic Control Unit, is allowed to control program counter operation. However when ALE is low the system input is locked-out to prevent the Logic Control Unit from accidentally presetting the program counter. This is necessary during program loading since the LCU is not otherwise prevented from trying to execute the data on the data bus as it is loaded into program memory. The manual PRESET PC switch is active when ALE is low. Both of these controls are duplicated in the mP control bus.

v) R/$\overline{\text{W}}$ MEMORY. The R/$\overline{\text{W}}$ switch controls the operation mode of the RAM program memory. During normal operation the signal is High to enable the reading of memory data. The signal is lowered to enable writing of

new programs into memory. This signal is directly connected to the RAM packages and is duplicated in the mP control bus.

vi) MANUAL DATA SWITCH ENABLE. This switch enables the manual data switches on the local console for program loading and program counter presetting. It activates a set of tri-state buffers which isolate the switches from the data bus.

vii) OUTPUT ENABLE. This signal exists only on the mP control bus. It is used to inhibit the output data from being latched into the output ports by unintentional operations of the Logic Control Unit during mP/BD communications.

3.2.4.2 Program Memory. The BD processor program memory is implemented with a set of four 128x8 MC6810 RAM packages providing a total of 256 words of 16-bit wide memory, figure 3.10. The chips are connected to the processor's address and data busses.

The read/$\overline{\text{write}}$ (R/$\overline{\text{W}}$) mode line is controlled by the Auto/Manual Control Interface circuit described in the previous section. When this line is in the read state, instructions are accessible to the system but cannot be altered. Individual memory locations are accessed by an instruction address provided by the program counter via the address bus.

In the write state, the contents of the RAM devices can be altered to allow loading of new programs into memory. The literal contents of the data bus are written into the memory location selected by the address bus data during this mode. Successive locations are filled by strobing the clock input of the program counter while its $\overline{\text{LOAD}}$ line is held high to select the incrementing mode. After each new instruction's

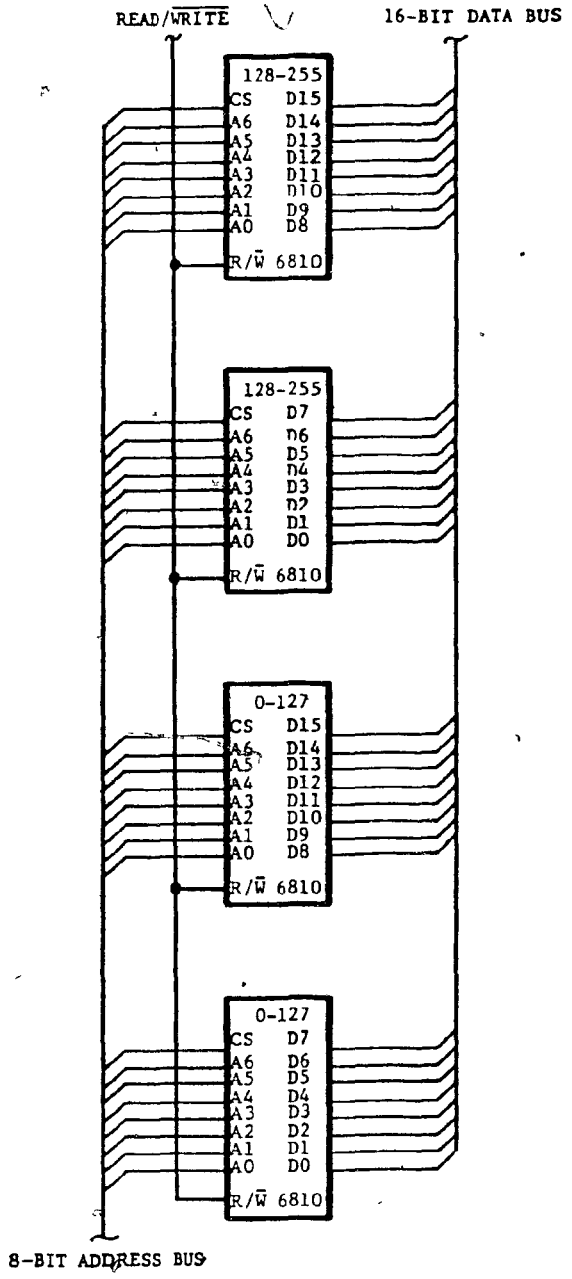


figure 3.10 Program memory circuit diagram.

data has stabilized on the data bus, it is stored in the program memory by a high-low-high strobe of the R/$\overline{W}$ line.

The RAM/address bus interface is driven by 8T95 drivers (not shown) as the current sourcing capacity of the 6810 ICs is insufficient to drive the entire address bus load.

3.2.4.3 System Clock. The system clock consists of the 1.00000 MHz crystal and, on the current BD prototype, a pair of 4040 counters forming a frequency dividing circuit, figure 3.11. The latter circuit furnishes two reduced-frequency clocks, 62.5 kHz (1 MHz/2^4) and 0.5 Hz (1 MHz/2^{21}) which are used as functional system clocks for normal BD processor operation and "slow motion" checkout operation, respectively. The clock speed is selected by the CLOCK RATE SELECTOR switch on the manual control console through the NAND gate circuit of U4. The output of U4-6 is sent to the Clock-Interrupt Circuit as previously described.

The CLOCK RATE SELECTOR switch is not duplicated on the mP control bus and is not affected by the AUTO/MANUAL switch.

The frequency divider circuit was implemented in the Holck prototype to relax the design requirement for high speed IC components and to make circuit debugging easier. However, now that the circuit characteristics are well established the frequency divider is no longer required. It is proposed to eliminate this circuit in future prototypes.

3.2.4.4 Field Input Section. The input section consists of four 74150 16-1 Multiplexors connected with a 74151 8-1 Multiplexor (MUX),

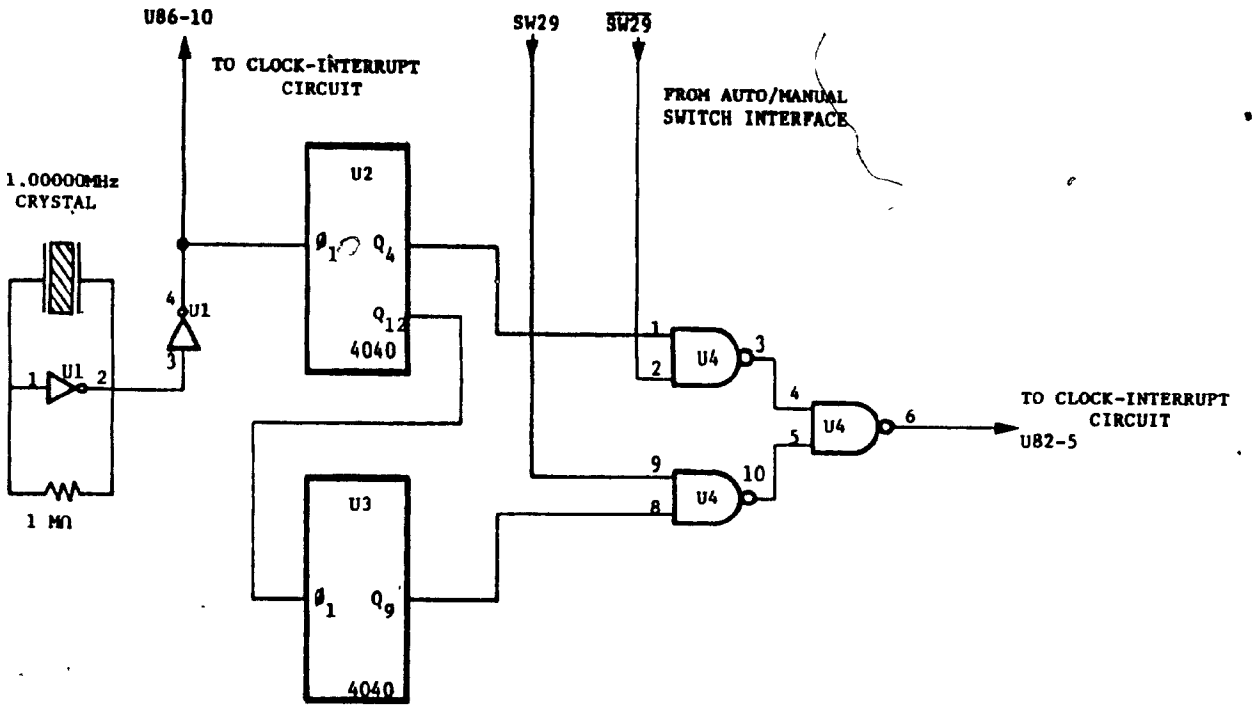


figure 3.11 System clock circuit diagram.

figure 3.12. Together these comprise the 64-bit field input register. The output of U25, the 8-1 MUX, constitutes the selected input variable, X, sent as input to the LCU.

IN<13:10> are connected in parallel to the data selector address inputs of the four 16-1 MUX chips to generate four selected field inputs. U25 selects one of these according to IN<9:8>.

The operation of the input section is not interlocked with the op-code decoder circuit of the LCU (see section 3.2.4.2). Thus it always interprets IN<13:8> as an input address regardless of the actual instruction being executed.

An improved configuration would see U25 replaced by a 74150 16-1 MUX driven by a 4-bit input bank register instead of IN<9:8>. In this case sixteen banks of 16 inputs could be handled, but this would require the implementation of an INPUT BANK SELECT instruction.

Figure 3.12 also shows the distribution of the eight inputs simulated by manual switches and of the four field inputs.

3.2.4.5 Field Output Section. The output section, figure 3.13, consists of the output bank register, the output bank address demultiplexor and sixteen sets of output register latches. The operation of this circuit is as follows.

Upon execution of a BANK SWITCH instruction, the bank select signal generated by the LCU clocks the contents of IN<11:8> into the output bank register, U41, a 7475 quad D-type flip-flop. The outputs of U41 are fed in parallel to two 74154 4-16 demultiplexors, U39 and U40, to decode the latched output bank address. U40 enables the upper four

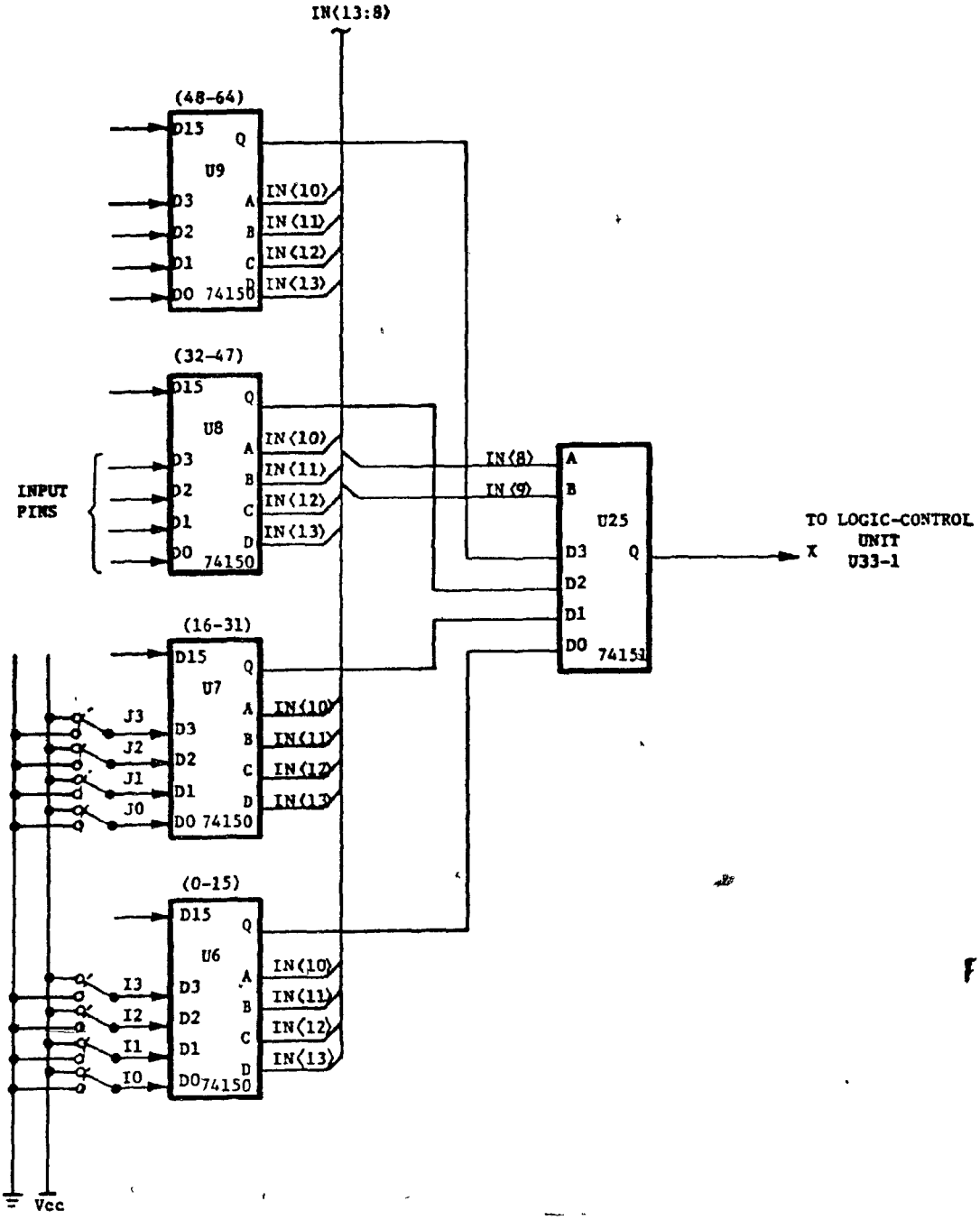


figure 3.12 Field input section circuit diagram.

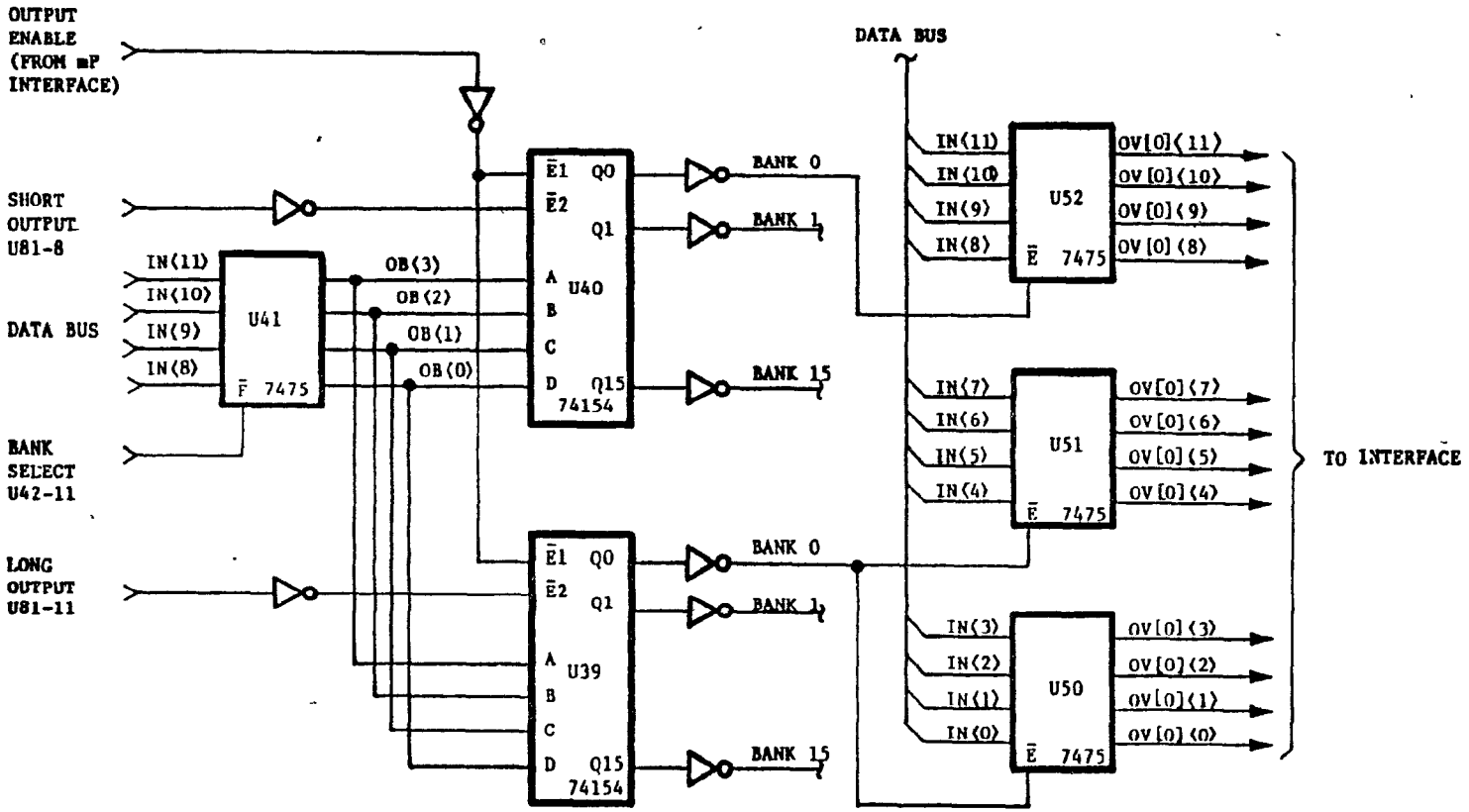


figure 3.13 Field output section circuit diagram.

latches in each output register and U39 enables the lower eight latches. During execution of a SHORT OUTPUT instruction, the LCU generates a pulse on the short output line. This briefly enables U40, causing a pulse to occur on the selected output line which clocks IN<11:8> into a 7475 flip-flop latch (U52 in the figure). Likewise, the execution of a LONG OUTPUT instruction generates a pulse on the selected output line of U39 which latches IN<7:0> into the lower bits of the output register (U50 and U51 in the figure).

The DEMUX chips are interlocked with the OUTPUT ENABLE signal provided by the mP control interface. This inhibits all of the output latches during mP to BD communications to prevent output data from being accidentally overwritten.

3.2.4.6 Timers and Flip-Flops. External 555-based timers are used to generate timing functions, figure 3.14. Timers are started by the outputting of a timing request R, and run for a predetermined, hand set period. The timer's condition is established by reading an input line as shown in table 3.2. To minimize the number of input points required, one additional output common to all timers is used to activate a flip-flop to select between T or $R \cdot \overline{T}$.

Flip-flops are simply implemented by tying one output line back into the input register. The output bank latch becomes the memory element.

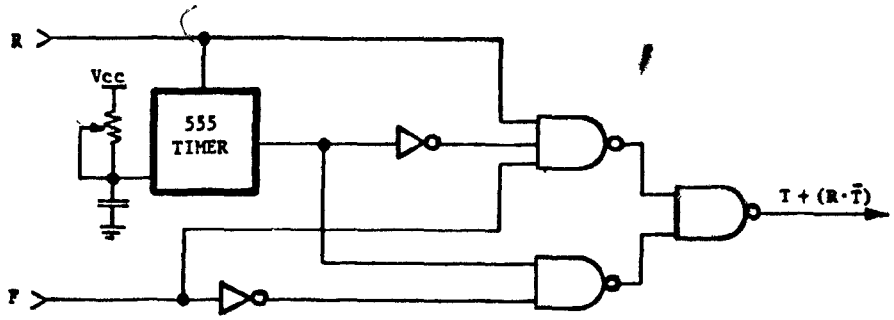


figure 3.14 Timer hardware circuit diagram.

TIMER	R	T	F	R·T
INACTIVE	0	0	0	0
			1	0
TIMING	1	1	0	1
			1	0
EXPIRED	1	0	0	0
			1	1
IMPOSSIBLE CONDITION	0	1	X	X

table 3.2 Timer hardware circuit truth table.

3.3 mP/BD Interface Design

The mP/BD Interface Module is a MC6821 Peripheral Interface Adaptor (PIA)-based card which drives the BD processor's 16-bit bidirectional data bus and 8-bit control bus. The data bus is connected to the BD program memory via U5 and U6, 74245 tri-state transceivers, figure 3.15. BD programs are downloaded from the mP to the BD memory or read back into the mP via this bus.

Additional 74244 tri-state drivers, U7 and U8, connect the lower four bits of the data bus to the field input register of the BD and bring out twelve field outputs from output bank 0. These are required to implement a BD hardware verification mode in which a test program is loaded into the BD memory and executed. The mP transmits test data over the four inputs and compares the program output with a tabulated set of results.

The data bus drivers are controlled by U3, the 74273 8-bit control register. The ENABLE BI/$\overline{\text{VER}}$ line selects which set of drivers, U5-U6 or U7-U8, are connected to the PIA. The transmission direction of U5 and U6 is controlled by the DIR output of the control register.

Auto-mode control signals such as AUTO LOAD ENABLE, STOP, PRESET PC and OUTPUT ENABLE, are also generated by this control register. The R/$\overline{\text{W}}$ and SINGLE STEP signals are provided by the CB2 and CA2 outputs of the PIA. INTERRUPT, BD-CLOCK-ON and AUTO/MANUAL switch position signals are fed back to the mP via the CA1 and CB1 control inputs of the PIA. A mP interrupt is generated by the PIA when a transition occurs on either CA1 or CB1 to indicate to the mP a change in the status of the BD processor.

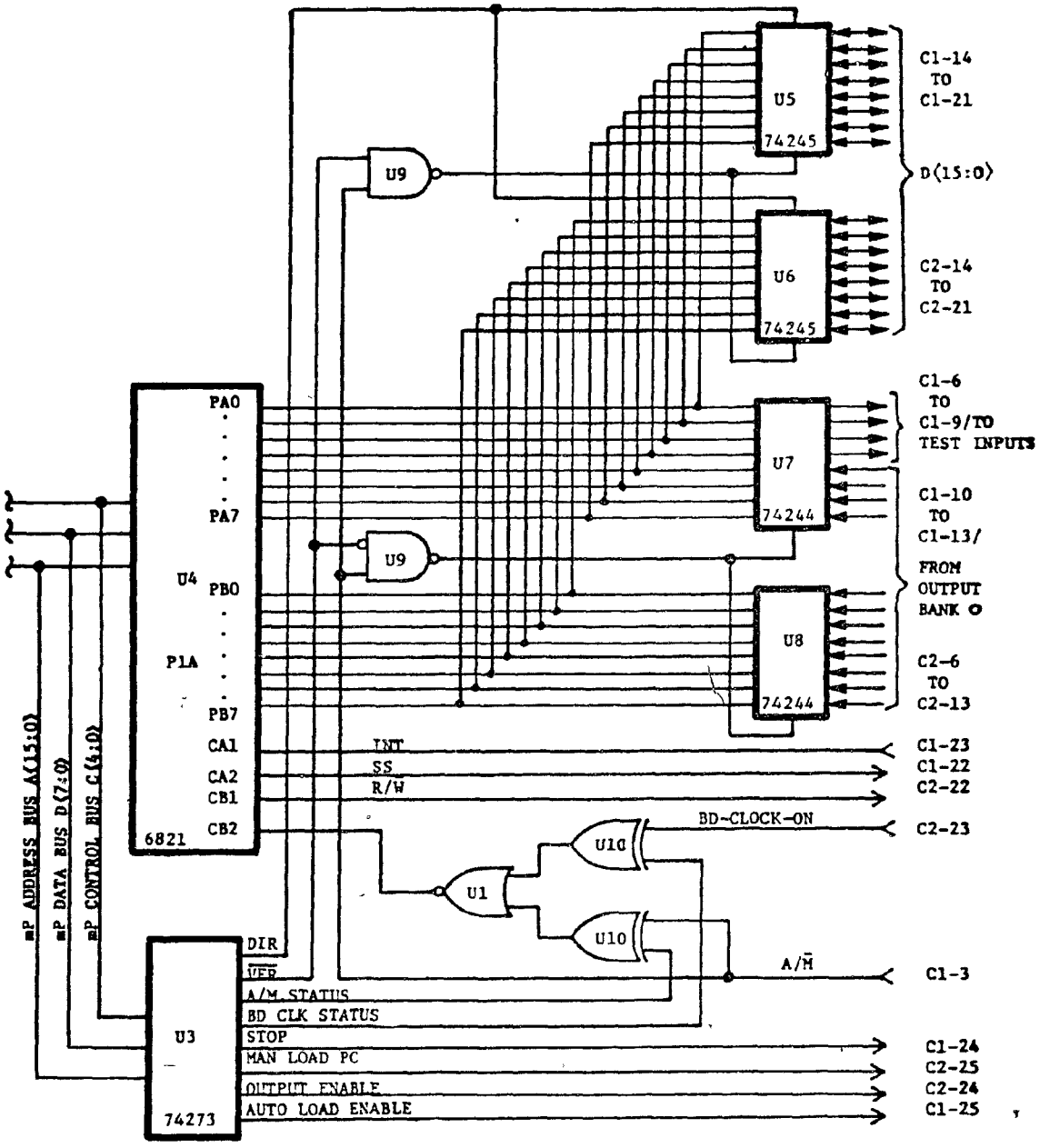


figure 3.15 mP/BD Interface Module circuit diagram.

The EXCLUSIVE-OR circuit consisting of U1 and U10-3 and U10-4 multiplexes the BD-CLOCK-ON and AUTO/MANUAL switch position signals onto the CB1 input. The circuit is driven by the A/M STATUS and BDCLK STATUS signals generated by the 8-bit control register to enable the mP to discriminate between a change in the AUTO/MANUAL switch position and a change in the status of the BD-CLOCK-ON flag.

The mP/BD interface module is connected to the SS-30 bus, I/O expansion area of the SWTPc 6809 computer and occupies five memory locations.

CHAPTER 4

mP/BD HYBRID PLC OPERATING SYSTEM DESIGN

4.0 Introduction

An operating system (O/S) is a program which manages the hardware and software resources of a computing system and coordinates the interaction of the resources to achieve the system's purpose. The hardware resources of the mP/BD PLC consist of the mP-based controller, the BD-based controller, and the operator interface. The software resources include mP and BD process control programs, program compilers, and other support software. This chapter describes two operating systems created for the mP/BD PLC.

The first section discusses the design of a real-time, concurrent multi-programming process control operating system: BD09. This program supports task scheduling in each of the processors, manages program memory allocation for the efficient execution of concurrent tasks, maintains libraries of pre-compiled control programs and executes inter-processor and operator communication functions. The purpose of its discussion is to outline the philosophy of operation of the PLC design. It has been implemented thus far only in rudimentary form.

The second section of this chapter deals with the design of the system software currently used to operate the SWTPc 6809 computer in conjunction with the BD processor prototype. This program, called BDBUG, implements operator-controlled functions:

1) mP-BD communications;

2) BD program retrieval from a disk-based library; and

3) BD processor initialization, loading and operating commands.

4.1 The BD09 Operating System

A commercial version of the mP/BD hybrid programmable controller would differ from conventional PLCs in several ways:

1) The parallel processors (mP and BD) enable simultaneous execution of PID and Sequential Automation control tasks;

2) High processor speed (particularly in the BD unit) makes concurrent multi-programming feasible in real-time; and

3) The hardware interrupt structure of the hybrid PLC enables the controller to respond to scheduled and unscheduled process events with any appropriate strategy including changing the active control algorithms without operator intervention.

The mP-resident BD09 operating system contains the system software required to perform task scheduling for the dual processors and to execute internal and external system communications functions. Consider the program structure illustrated in figure 4.1. The functional elements are described in the following sections.

4.1.1 Interrupt Decoder. The interrupt decoder is the highest level of the BD09 O/S. Process control algorithms are executed on an "round-robin" basis as the primary task of both the mP and BD processors. The O/S program resides in the mP as a background task until an event such

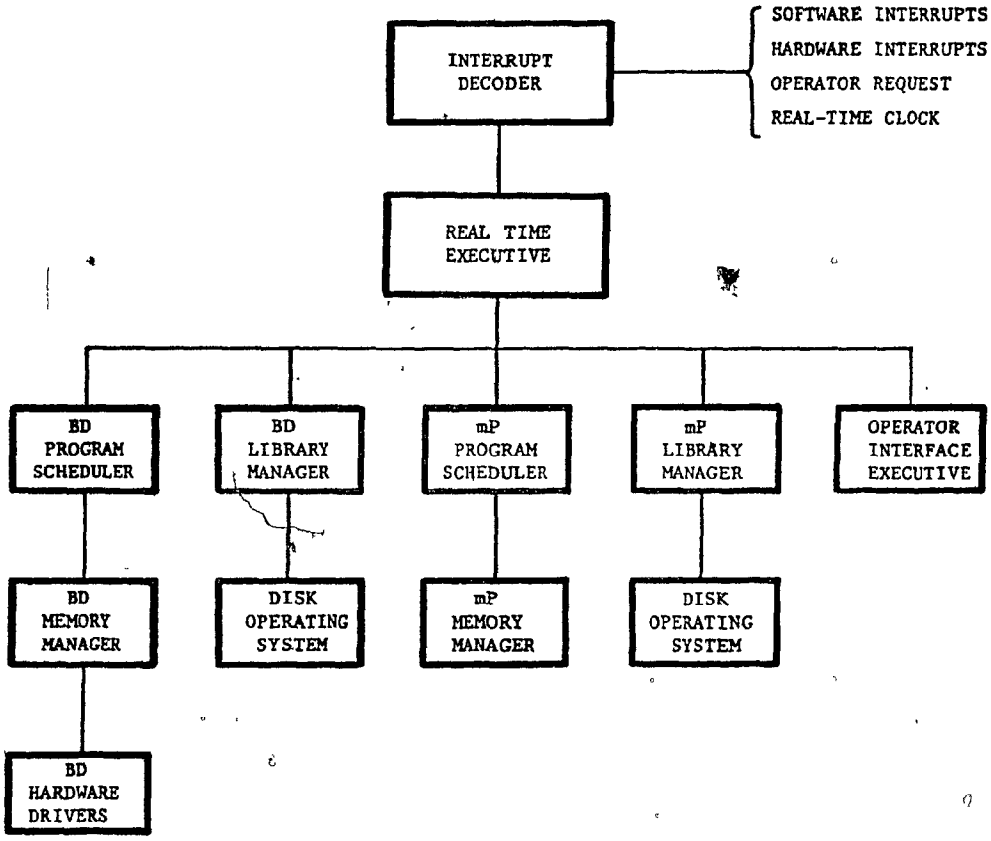


figure 4.1 BD09 O/S block diagram.

as a programmed interrupt, process equipment trip, controller hardware malfunction or operator intervention interrupts one or both of the processors. Control of the mP is switched to the BDO9 O/S by the hardware interrupt vector circuitry of the microprocessor, where the interrupt is decoded in order to start the appropriate service routine.

Multiple interrupts are served on a first-in, first-out basis. The hardware interrupt mask is set by the service routines so that they cannot be halted by subsequent interrupts. A polling routine in the Interrupt Decoder scans the interrupt flag bits of the system peripherals to determine the source and control is passed to the Real Time Executive for action. All interrupts, channelled through the $\overline{IRQ}$ line of the MC6809, have equal priority although the polling routine does impose an order in which multiple interrupts are recognized. The controller resumes execution of the old job schedule or a newly assigned one after all the interrupts have been cleared.

4.1.2 Real Time Executive. The Real Time Executive (RTX) is composed of the interrupt servicing routines which activate program managers and hardware drivers for four categories of interrupts. These are described below.

1) Software traps. These occur it the process variables have exceeded the precompiled range of the active control algorithms. In a scheme analogous to "virtual memory", programs of excessive length are divided into a series of semi-independent sections, each valid in a specific control range, which are moved into and out of program memory as range boundaries are encountered. This is of particular importance in the BD

processor where the memory requirement of binary-complete programs grows with input variable count as 2^{n+1}. Here, software traps are implemented by replacing the control output instructions at the upper and lower limits of the variable range by software interrupt instructions containing an appropriate error code, figure 4.2. Other situations favouring the use of software traps for the detection of abnormal or noteworthy process conditions include equipment trips which require specific shutdown programs or "benchmark" events which must be annunciated to the operator.

The Executive reads the interrupt code from the last executed EOP instruction in either the mP or BD, as appropriate, to determine the type of condition present and the action to be taken. In the case of an mP program interrupt, the Executive can read this data directly as the control of the mP is automatically transferred to the O/S and the process control algorithm is suspended at the interrupt location. However, in the case of a BD interrupt, the protocol is somewhat different. When an interrupt is generated by the BD machine, the Executive responds by raising the STOP flag to halt the operation of the BD processor. The BD unit is not permitted to stop itself since other programs concurrently executing in the BD program queue must be allowed to continue normally. The response time of the O/S is insufficient to ensure that the STOP flag is raised before the interrupting program reaches its EOP instruction, the only legal place a program can be halted. To prevent any of the other EOP instructions in the other programs from stopping the BD processor, a toggle flip-flop sets a mask to lock out the EOP signals from affecting the Clock-Interrupt Circuit

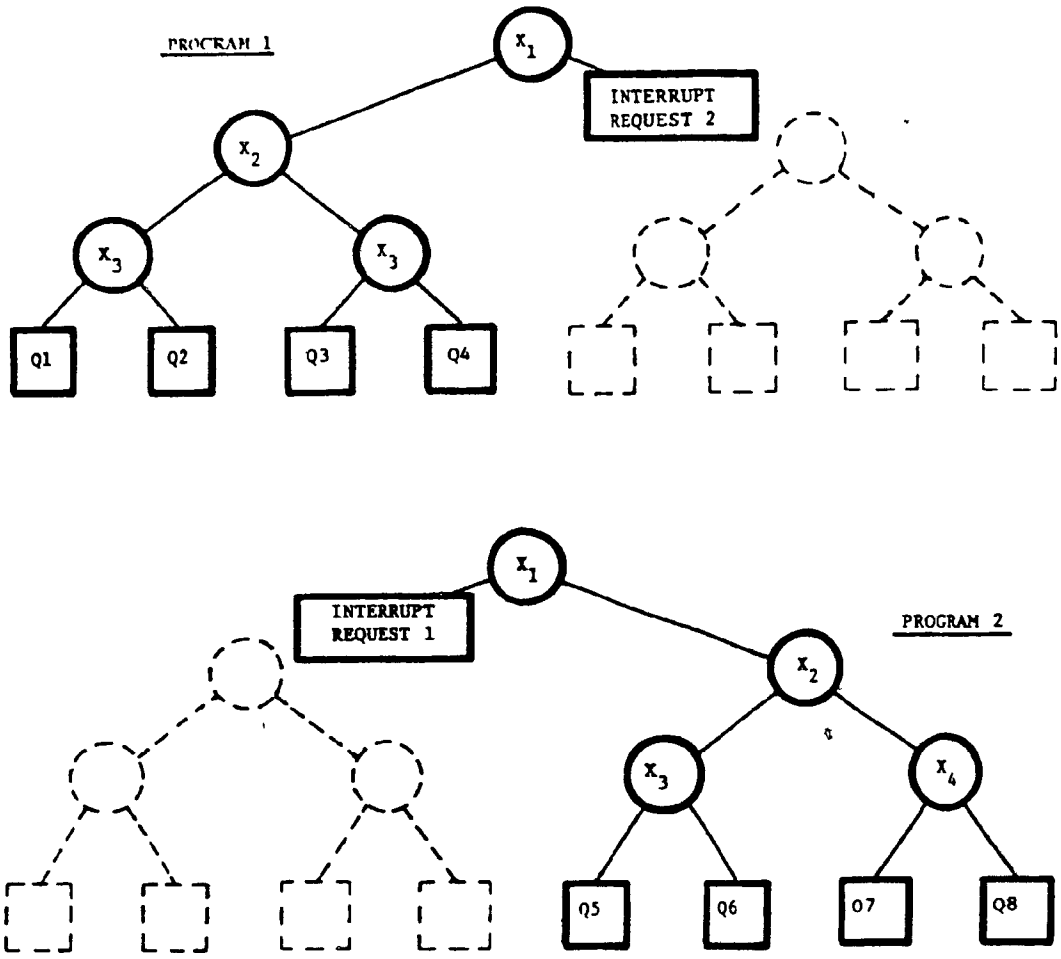


figure 4.2 Program size reduction by partition.

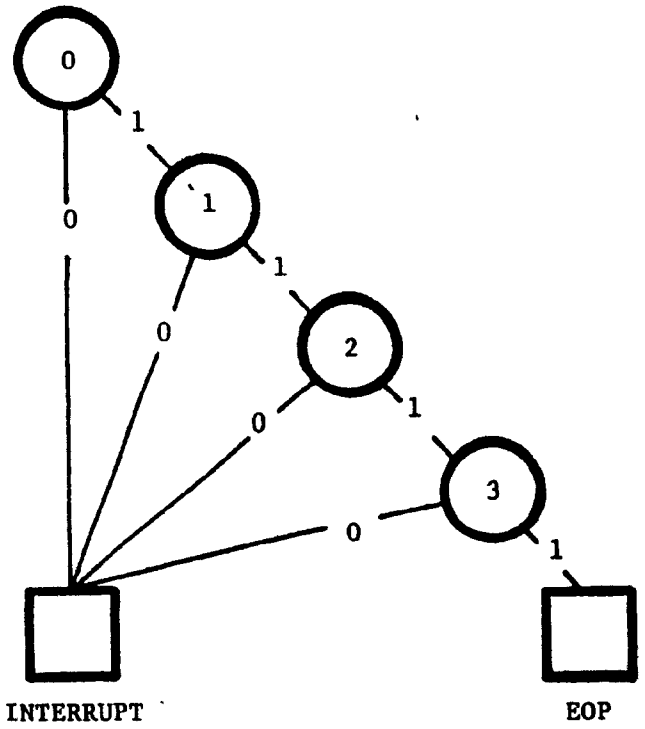
(see chapter 3). The mask is cleared by the execution of a second interrupt instruction. Normally this will come from the same program. The BD machine then stops at the next EOP instruction. (If the subsequent interrupt occurs due to another program detecting an abnormal process condition, then the two events are likely to be connected and the entire program queue probably needs to be changed to adapt the controller to the new operating conditions. If the two events are unrelated the initial software interrupt would recur once the BD processor was released by the O/S, resulting in a second interrupt sequence.) If additional control programs are necessary, in the case of either the mP or BD unit, the Real Time Executive requests the files from the appropriate library management routine and passes it to the appropriate memory manager for installation into the program queue.

2) Hardware Traps. BD processor hardware traps occur if the unit unexpectedly stalls, signalled by the fall of the BD-CLOCK-ON flag in the control interface, or if the machine is decoupled from the mP by operator manipulation of the AUTO/MANUAL switch. Other hardware traps can be generated by self-check programs included in the mP and BD program queues to detect processor or peripheral failures. If the BD unit has failed but can be restarted, a diagnostic program is loaded and run to determine the source of the error. Control program execution is resumed if the fault is non-fatal. Otherwise, the operator is alerted and the process is put under manual control. Methods of detecting hardware failure include time-out circuits which in normal operation are reset by the periodic execution of a particular instruction sequence.

If the sequence is not executed, either due to hardware problems or the problem of a program caught in an infinite loop, the circuitry times-out and generates a warning flag. Another method of detecting hardware problems is to include a short program in the queue which reads a test pattern sequence of input data bits, controlled by the O/S, figure 4.3. Any detected discrepancy indicates errors in the input circuitry and initiates an interrupt.

3) Operator Requests. The operator interrogates the PLC by sending an interrupt through the keyboard device. The Executive can provide information such as the operational status of each processor, the listing of the current program queues, and the contents of program libraries. The operator can also update the program queues via the Program Schedulers for either the mP or BD unit, compile new programs and change the contents of the program libraries.

4) Real-Time Clock Interrupt. This independent time-base is used to synchronize the O/S with scheduled process events such as equipment startup and shutdown. On a periodic basis, the real-time clock generates an interrupt which causes the Executive to increment a register containing time counts representing the actual time-of-day. The Executive interrogates the Program Scheduler routines for both the mP and BD unit to determine if a scheduled event is pending. If so, the memory and library management routines are activated to implement the required changes. If not, the O/S resumes the execution of the previous program queues.



BD PROGRAM

PC	CODE$_{16}$	FUNCTION
F8	40FE	INPUT VAR "0"
F9	48FE	INPUT VAR "1"
FA	50FE	INPUT VAR "2"
FB	58FE	INPUT VAR "3"
FC	A001	EOP
FD	F000	JUMP TO 0
FE	DF00	INTERRUPT
FF	AF01	EOP (INTERRUPT)

figure 4.3 Hardware self-test program.

4.1.3 mP and BD Program Scheduler and Memory Management Units. The Program Scheduler and Memory Management Units for the mP and BD processors are functionally identical. They are described as follows.

1) Program Scheduler Unit (PSU). The PSU is a routine which allows the operator to schedule the startup and shutdown of individual control tasks. A table is maintained by the PSU which lists the programs that have been scheduled together with the time-of-day of the event and an on/off flag to indicate whether the control task is to be started or stopped. The operator makes insertions or deletions to the table from the keyboard device. Periodically, the real-time clock interrupt causes the PSU to scan the program table. The actual time-of-day is compared to the table entries to determine if a scheduled event should be processed during this interrupt period. If so, the PSU requests the program from the library and passes it to the Memory Management Unit (MMU) for loading into program memory prior to execution. The table entry is then deleted. If a program is to be turned-off, the PSU requests the MMU to remove it from the program memory. Controller task processing is resumed after the PSU is finished its examination of the table.

2) Memory Management Unit. Since one or more control algorithms may be in the program memory concurrently, the MMU is responsible for space allocation to ensure that active programs are not damaged by accidental overwriting. A memory map is used to record the disposition of each program in program memory as well as the available free areas. The MMU is made up of three parts:

i) ALLOCATOR. This routine searches the memory map for free areas to assign to incoming programs. The new program is rejected if insufficient space is available. Successive program additions and deletions may leave inter-program gaps which tend to fragment the memory and waste useable space. The Allocator is able to rewrite the program memory to concentrate the free sectors together so that allocation of this space may continue.

ii) RELOCATOR. Control programs which are compiled in absolute addressing mode, such as BD programs, cannot be loaded into memory areas arbitrarily selected by the memory Allocator because the conditional and unconditional jump instructions contain location-dependent data. This data must be translated to coincide with the actual area allocated to the program prior to loading. The Relocator compares the compiled location of new programs as they are processed by the MMU with the load addresses provided by the Allocator routine, and adds or subtracts the difference from each transfer instruction in the programs.

iii) LOADER. Allocated, relocated programs are transferred from the working area of the MMU to the program memory area. In the case of mP-based programs, this is simply accomplished by rewriting the programs in a different section of the same Random Access Memory. Operations in the BD processor's memory are more involved. The Loader signals the Real Time Executive to interrupt the BD processor if it is still executing control programs. This is achieved by raising the STOP flag in the interface control bus. After the execution of the next EOP instruction, the BD clock is interrupted and the BD-CLOCK-ON signal falls to confirm

that the machine has halted. The Loader then actives the control bus to preset the BD processor's program counter to the initial location of the allocated area, after which it transfers the new program to the BD unit. When complete, the Loader signals the Real Time Executive to lower the STOP flag, thus restarting program execution. The procedure for unloading programs is to update the memory map to reflect the additional free area.

4.1.4 mP and BD Library Management Units. Inactive mP and BD control programs are stored in disk-resident program libraries. The Library Management Units (LMU) maintain directories apart from the global disk directory to facilitate program retrieval in response to requests from the Real Time Executive. Each directory record consists of an identification number, a program classification number, the compiled memory location and the length of the program. The LMU's support three types of library requests which originate from either the operator or the RTX:

1) File Storage. Programs are automatically saved in the libraries after compilation. The directory is checked to see if a program of the same number is already present. If so, the operator can choose to replace the existing program with the new one or to abandon the new file.

2) File Retrieval. Programs are transferred from the libraries at the request of the RTX. Disk transfers are handled by the embedded disk operating system routines.

3) File Deletion. Programs are removed from the libraries by operator request when they are no longer required or when they have been superseded by newer versions.

4.2 The BDBUG Operating System

The BDBUG program is currently being used to operate the SWTPc 6809/BD processor development system. It implements many of the high and low level features of the BD09 O/S such as the Interrupt Decoder, three of the four classes of interrupt servicing routines of the Real Time Executive, the Relocator and Loader routines of the BD Memory Management Unit and the operator interface. This program served as the test bed for the communication interface hardware described in chapter 3.

BDBUG is used in conjunction with the FLEX 9.0 DOS and the V-BUG monitor, a local modification to the commercially distributed S-BUG monitor [Vroo81]. These other programs supply disk transfer and terminal communication subroutines for use by BDBUG. A fully integrated, EPROM-based version of BDBUG and the required subsets of FLEX and V-BUG is under development.

Operator interface commands are patterned after the V-BUG command set. Programs, once loaded into the mP working memory by FLEX, can be relocated and loaded into the BD processor's program memory by keyboard commands. The BD unit can be started, stopped and single-stepped through programs, among other functions, via the keyboard. The structure of the BDBUG program is described in the following sections. A complete listing of BDBUG is presented in Appendix III.

4.2.1 Program Initialization. When control of the SWTPc 6809/BD hybrid system is passed to the BDBUG program from either FLEX or V-BUG, certain initialization procedures are required to configure the computer for the hybrid role. These actions are:

1) Set Interrupt Mask. This prevents random interrupts from interfering with the initialization procedure, in particular, while the interrupt vector is being changed.

2) Store IRQ Vector. The starting address of the Real Time Executive kernel is stored in the IRQ vector of the 6809 mP. All subsequent $\overline{\text{IRQ}}$ interrupts (as opposed to $\overline{\text{FIRQ}}$ or $\overline{\text{NMI}}$ interrupts, for example) cause the 6809 to jump to this address for processing.

3) Initialize ACIA. Communication with the operator interface terminal is achieved by a serial Asynchronous Communication Interface Adaptor (ACIA) card which resides on the SWTPc computer SS-30 Input/Output (I/O) bus. This card automatically generates the RS-232 protocols for the serial transmission of keyboard and CRT data. It must be initialized to set the transceiver parameters to the appropriate values.

4) Initialize BD Interface. BD interface communication is controlled by a parallel Peripheral Interface Adaptor (PIA) card in slot 3 of the SS-30 I/O bus. The PIA is initialized to enable the BD to mP communication pathway.

5) Initialize "Events" Printer. An "events" printer is used in industrial control systems to log messages, alarms and actions of the control system during day to day operations. This information is used

in performance analysis of process equipment and provides a record of events surrounding an emergency situation. The mP/BD development system uses a standard EPSON dot matrix printer. Once initialized, it can be turned on or off by the operator via the BDBUG command set.

6) Get Operational Status of the BD Processor. BDBUG remembers the auto/manual and running/stopped status of the BD processor as a means of determining the validity of operator requests. E.g., an operator command to restart the BD processor is invalid if the processor is already running. The actual status is ascertained from the BD interface card.

7) Clear System Interrupts. Interrupts, from either the BD processor hardware, software, or the operator terminal are cleared by performing a microprocessor LOAD operation of the data registers of the SS-30 bus interface adaptor devices.

8) Print Prompt Characters. BDBUG writes the characters ">>" to the CRT screen to indicate that it is operating and ready to receive operator commands.

9) Enable Interrupts. As a final step, the interrupt mask is removed to activate the mP/BD prototype PLC.

4.2.2 Real Time Executive. The RTX comprises the Interrupt Decoder and interrupt handlers for three categories of signals, BD software interrupts (BDSWI), BD hardware interrupts (BDHWI), and keyboard device interrupts. These routines are described below.

4.2.2.1 Interrupt Decoder. This routine is activated by a low pulse on the $\overline{IRQ}$ line of the MC6809. The internal interrupt circuitry stops the execution of the current program at the end of the current instruction and saves the entire machine state (PC, Accumulators, Index Registers, Stack Pointers, Condition Code Register and Direct Page Register) on the system stack. The hardware interrupt mask is set and the PC is loaded with the contents of memory address $FFF8\text{-}9_{16}$, the interrupt vector location. The computer starts execution at this address which contains a vectored jump instruction to the beginning of the Interrupt Decoder routine. Figure 4.4 shows an algorithmic description of the Interrupt Decoder. This routine polls the BD and the keyboard input interface cards to determine which device has initiated the interrupt. The routine examines PIACRA<7> and PIACRB<7>, the control registers of the A side and B side, respectively, of the MC6821 PIA which controls mP-BD communication. A high signal indicates a BDSWI and BDHWI, respectively. The routine then reads ACIACR<7> and ACIACR<0>, the control register of the MC6850 ACIA which controls the operator terminal communications. A high signal indicates a keyboard interrupt.

The polling order prioritizes the interrupts in the order BDSWI, BDHWI, KEYBRD. The appropriate interrupt handling routine is called as a subroutine of the Interrupt Decoder as soon as an interrupt flag is recognized. Thus, if two interrupts are present, the first one polled is serviced immediately. The handling routines return to the Interrupt Decoder at the completion of the subroutine and polling is resumed. If no further interrupts are found, the Interrupt Decoder executes a

```
PROCEDURE  INTERRUPT.DECODER
     IF  INT:=BDSWI THEN
     -  ACTIVATE PROGRAM INTERRUPT HANDLER
     IF  INT:=BDHWI THEN
     -  ACTIVATE HARDWARE  INTERRUPT HANDLER
     IF  INT:=KEYBRD THEN
     -  ACTIVATE OPERATOR  INTERFACE HANDLER
     CLEAR INTERRUPT FLAGS AND RESUME CONTROL TASKS
RETURN
```

figure 4.4 BDBUG Interrupt Decoder algorithm.

```
PROCEDURE  HARDWARE.INTERRUPT.HANDLER
     REPEAT UP TO 256 TIMES
          BEGIN
               IF  BD:=MANUAL/STOPPED THEN
               -  STORE BD STATUS
               ELSE
                    IF BD:=MANUAL/RUNNING THEN
                    -  STORE BD STATUS
                    ELSE
                         IF  BD:=AUTO/STOPPED THEN
                         -  STORE BD STATUS
                         ELSE
                              IF  BD:=AUTO/RUNNING THEN
                               -  STORE BD STATUS
          END
          -  REPORT BD STATUS OR POLLING FAILURE
RETURN
```

figure 4.5 BDBUG hardware interrupt service routine algorithm.

return-from-interrupt (RTI) instruction, to resume whatever task was executing before the interrupt occurred. If no other task was executing the prompt characters are re-issued and the system awaits another event.

4.2.2.2 BDSWI Service Routine. BD program interrupts are generated by INTERRUPT instructions embedded in active BD programs. In the mP/BD PLC these are used to indicate noteworthy process conditions which may require different control sequences. This feature has not been fully implemented in the laboratory prototype. When a BDSWI interrupt is recognized, the service routine simply sends the message "BD PGM INTERRUPT" to the CRT screen and clears the interrupt flag of PIACRA by executing a LOAD instruction of PIAORA, the data register of the PIA. No other action is taken at present. The routine then executes a return-from-subroutine (RTS) instruction to return to the Interrupt Decoder.

4.2.2.3 BDHWI Service Routine. BD hardware interrupts are generated by the BD-CLOCK-ON line of the Clock-Interface Circuit (see section 3.2.4.1) and by transitions of the AUTO/MANUAL switch (see section 3.3). These two lines are multiplexed onto the CB1 input of the PIA. A mP controlled EXCLUSIVE-OR (XOR) circuit is used to discriminate between changes in these two signals. The BDHWI routine activates the XOR circuit to determine the correct operational status of the BD processor and stores the state of the BD-CLOCK-ON and AUTO/MANUAL switch in memory locations CLKREG and AMREG, respectively. This scheme is used because these lines cannot be read directly, due to the "write-only" characteristics of the BD control interface register.

Referring to figure 3.15, the high state on the CB1 input is achieved only when the BD-CLOCK and A/$\overline{M}$ outputs of the BD control register are identical to the actual states of the BD-CLOCK and A/$\overline{M}$ signals from the BD processor. Any other combination produces the low state on CB1. BDHWI polls the XOR circuit by writing the four combinations of BD-CLOCK and A/$\overline{M}$ to the control register in sequence. When the correct combination is polled, CB1 goes high and an interrupt is generated by the PIA. The interrupt does not affect the mP as it is already in the interrupt state and has set the hardware mask, however, the PIA sets a flag in PIACRB<7> to indicate the event. BDHWI scans this flag after each combination is sent, to determine if the correct combination was polled. If so, the routine saves the current state of the BD processor in status registers and reports the new machine state to the CRT. If the polling routine fails to identify the correct combination after trying all four, it retries as many as 255 times and then reports the failure to the CRT. An algorithmic description of the BDHWI interrupt service routine is shown in figure 4.5.

4.2.2.4 Keyboard Service Routine. Operator interrupts are generated by the activation of any key on the terminal keyboard device. BDBUG commands consist of a forward slash character "/", followed by a single character alphabetic code. (This arrangement permits the O/S to discriminate between BDBUG and V-BUG commands. A continuing goal of this research is to load both of these monitors into a single EPROM and to operate them simultaneously.) The input code is matched to the set of legal commands stored in a jump table in memory. A data match causes the routine to call the command as a subroutine. An error message is

issued to the CRT if the input characters are not found in the command table and the O/S returns from the interrupt state. A summary of the valid commands is presented in table 4.1.

4.2.3 BD Memory Management Unit. In addition to transmitting program data to and from the BD program memory, the MMU routines generate control codes for the control of the BD Program Counter and Clock-Interrupt Circuit via the Auto/Manual Control Interface as described in section 3.2.4.

Operator commands implement the kernel of the BD Memory Management Unit described in section 4.1.3. Some of the commands are patterned after conventional microprocessor monitor commands. E.g., individual memory locations can be examined and changed using the "/M" command, whole sections of BD program memory can be examined using the "/E" command, and BD programs can be activated using the "/J" command. As well, several of the commands are unique to the BDBUG O/S, e.g., the program relocation command, "/T", that translates the absolute addresses of executable BD program instructions so that the program can be loaded into any contiguous section of BD memory, the load command, "/L", that loads programs from the mP memory to BD memory, or the single-step command, "/S", that allows the operator to debug program logic by executing instructions one at a time. In addition, other commands are implemented which control the peripheral events printer, report the current status of the BD processor to the CRT and allow an orderly exit from BDBUG to either FLEX or the V-BUG monitor.

COMMAND	FUNCTION
/M XX	MEMORY CHANGE
/E SS-EE XXXX	EXAMINE A BLOCK OF BD MEMORY
/T SSSS-EEEE XX-YY	TRANSLATE BD PROGRAM ORIGIN
/L SS-EE XXXX	LOAD BD MEMORY FROM mP
/J SS	JUMP TO BD PROGRAM
/H	HALT BD PROCESSOR
/S SS	STEP THROUGH BD PROGRAM
/R	REPORT BD PROCESSOR STATUS
/P	PRINTER ON/OFF TOGGLE
/>	RETURN TO V-BUG
/+	RETURN TO FLEX

table 4.1 BDBUG command set.

BD STATE	COMMUNICATION DIRECTION	DATA REGISTER DIRECTION A	DATA REGISTER DIRECTION B	SINGLE STEP CA2	R/W CB2	PIACRA $CODE_{16}$	PIACRB $CODE_{16}$
RUN	BD TO mP	INPUT	INPUT	H[1]	H	3D	3D
STOP	BD TO mP	INPUT	INPUT	H	H	3D	3D
RESET BD PC	mP TO bd	OUTPUT	OUTPUT	H-L[2]-H	H	35	3D
LOAD MEM.	mP TO BD	OUTPUT	OUTPUT	H	H-L-H	3D	35
READ MEM.	BD TO mP	INPUT	INPUT	H	H	3D	3D
SINGLE STEP PC	N/A	N/A	N/A	H-L-H	H	35	3D
VERIFY	BD TO mP	INPUT	INPUT	H	H	3D	3D

1 - H = Digital Signal Level High.

2 - L = Digital Signal Level Low.

table 4.2 MC6821 PIA control codes.

The design of each of these commands is described below.

1) Change BD Memory. This routine is invoked with the "/M XX" command where XX_{16} is the address of the memory location to be examined and changed. The command is functionally identical to the V-BUG memory change command. The operator examines individual BD memory locations beginning with XX_{16} and has the options of changing the contents of the memory location and progressing to the next location, leaving the contents unchanged and progressing to the next location, or leaving the contents unchanged and backstepping to the preceding location. The routine is aborted by a carriage return or invalid (e.g., not hex) memory data. An algorithmic description of this command is shown in figure 4.6.

The BD processor must be in the AUTO mode and be stopped for the routine to be effective. These conditions are verified by checking the BD clock status and BD Auto/Manual status registers maintained by the BDHWI routine.

The direction of the bidirectional BD interface databus is set with the interface driver subroutines TOBD and TOMP. Likewise the presetting of the BD PC to the address XX_{16} and the actions of reading the BD memory data and changing the BD memory data are handled by the subroutines PRSTPC, RDWRD and LDWRD, respectively. These subroutines configure the two control registers of the MC6821 PIA on the interface module as well as the 8-bit BD control register so that the data transfers can take place. A summary of the control codes and resulting BD operational states is presented in tables 4.2 and 4.3.

```
PROCEDURE  BD.MEMORY.CHANGE
     IF BD STOPPED AND IN AUTO MODE THEN
          BEGIN
               INPUT BD MEMORY ADDRESS FROM KEYBOARD
               REPEAT UNTIL A CARRIAGE RETURN OR INVALID DATA IS
               RECEIVED
                  BEGIN
                    - SET BD TO DESIRED ADDRESS
                    - READ MEMORY DATA AND REPORT TO CRT
                    - INPUT USER RESPONSE FROM KEYBOARD
                    IF DATA IS VALID MEMORY DATA THEN
                         - STORE IN BD MEMORY AND READ BACK TO VERIFY
                           CONTENTS, OUTPUT A "?" IF DIFFERENT
                         - INCREMENT BD PC
                    IF DATA IS A CONTROL CHARACTER THEN
                         - DECREMENT PC IF A " " WITHOUT CHANGING
                           MEMORY DATA
                         - INCREMENT PC WITHOUT CHANGING MEMORY DATA
                  END
          END
RETURN
```

figure 4.6 BD memory change algorithm.

```
PROCEDURE  BD.BLOCK.MEMORY.EXAMINE
     IF BD STOPPED AND IN AUTO MODE THEN
        BEGIN
          - READ STARTING AND ENDING ADDRESSES DELIMITING THE BLOCK OF
            BD MEMORY TO BE TRANSFERRED TO THE mP
          - READ mP BUFFER ADDRESS
          REPEAT UNTIL ENTIRE BLOCK IS TRANSFERRED
            BEGIN
              - READ BD MEMORY LOCATION, STORE IN mP BUFFER AND ECHO
                TO CRT
              - INCREMENT BD PC
            END
        END
RETURN
```

figure 4.7 BD block memory examine algorithm.

BD STATE	AUTO LOAD ENABLE CR 7	OUTPUT ENABLE CR 6	PRESET PC CR 5	STOP FLAG CR 4	BD-CLOCK POLL CR 3	AUTO/$\overline{\text{MAN}}$ POLL CR 2	$\overline{\text{VERIFY}}$ ENABLE CR 1	COMM DIR'N CR 0	CONTROL CODE$_{16}$
RUN	H	H	H	L	H	H	H	L	EE
STOP	H	H	H	H	L	H	H	L	F6
RESET BD PC	L	L	H-L-H	H	L	H	H	H	17
LOAD MEM.	L	L	H	H	L	H	H	H	37
READ MEM.	L	L	H	H	L	H	H	L	36
SINGLE STEP PC	H	H	H	H	L	H	H	L	F6
VERIFY	H	L	H	L	H	H	L	L	AC

table 4.3 8-bit control register codes.

2) Examine a Block of BD Memory. This routine is invoked with the "/E SS-EE XXXX" command where SS_{16} is the starting address of the BD memory block to be read, EE_{16} is the ending address of the block and $XXXX_{16}$ is the mP address where the data is to be buffered. It transfers a block of BD memory to the mP and outputs it to the CRT. The BD processor must be in the AUTO mode and be in the stopped condition on entry to the subroutine. The interface driver subroutines are once again used to configure the BD processor for this operation. An algorithmic description of the command is presented in figure 4.7.

3) Translate Absolute BD Addresses. This routine is invoked with the "/T SSSS-EEEE XX-YY" command where $SSSS_{16}$ and $EEEE_{16}$ are the starting and ending addresses of the mP file buffer which contains the BD program code to be relocated, and XX_{16} and YY_{16} are the current BD program origin and desired origin, respectively. The purpose of this command is to assist the operator in allocating memory space to previously compiled programs which start from arbitrary memory origins. Once loaded into the mP working memory from disk, a program can be conditioned by translation of the absolute address mode transfer instructions, to load in any contiguous area of BD memory. Figure 4.8 presents an algorithmic description of this command.

4) Load BD Memory. This routine is invoked with the "/L SS-EE XXXX" command where SS_{16} and EE_{16} are the starting and ending addresses of the contiguous block of BD memory to be loaded and $XXXX_{16}$ is the mP address of the file buffer. The operator loads BD programs to the program memory using this command. The BD processor must be in the AUTO mode

```
PROCEDURE  TRANSLATE.BD.PROGRAM.ADDRESSES
     -     READ mP ADDRESSES DELIMITING BD FILE BUFFER AND CALCULATE FILE
           LENGTH.  ABORT PROCEDURE IF < ZERO
     -     READ ORIGINAL AND NEW BD PROGRAM ORIGINS AND CALCULATE THE
           OFFSET BETWEEN THEM
     REPEAT FOR ENTIRE FILE
        BEGIN
          - TEST NEXT INSTRUCTION IN THE BD FILE
          IF NOT AN EOP OR LONG OUTPUT INSTRUCTION THEN
               - ADD OFFSET TO ADDRESS FIELD AND REPLACE IN BUFFER
        END
RETURN
```

figure 4.8 BD address translation algorithm.

```
PROCEDURE  LOAD.BD.MEMORY
     IF BD STOPPED AND IN AUTO MODE THEN
          BEGIN
          - READ STARTING ADDRESS AND PRESET BD PC
          - READ ENDING ADDRESS AND CALCULATE WORD COUNT
          - READ mP BUFFER ADDRESS
          REPEAT FOR ENTIRE FILE
               BEGIN
               - READ NEXT RECORD
               - STORE IN BD
               - INCREMENT BD PC
               END
          END
RETURN
```

figure 4.9 BD load memory algorithm.

```
PROCEDURE  RESTART.BD.CLOCK
     IF BD STOPPED AND IN AUTO MODE THEN
        BEGIN
          - READ STARTING ADDRESS AND PRESET BD PC
          - ENABLE BD CLOCK BY LOWERING STOP FLAG
        END
RETURN
```

figure 4.10 Restart BD clock algorithm.

and be in the stopped state for the subroutine to execute. Interface driver subroutines are called to configure the interface control registers and to execute the data transfer. An algorithmic description is presented in figure 4.9.

5) Restart BD Clock. This routine is invoked with the "/J SS" command where SS_{16} is the address to which the PC is preset before restoring the BD clock to the LCU. This command is functionally equivalent to the V-BUG "Jump" command. The BD processor must be in the AUTO mode and not running, otherwise the subroutine is aborted. The interface drivers PRSTPC and TOBD are called to configure the interface control registers. An algorithmic description of this routine is presented in figure 4.10.

6) Halt BD Clock. This command permits the operator to halt the BD processor execution at the next EOP instruction. It is invoked with the "/H" command. The routine does not verify that the processor has stopped, to avoid a blocking situation if an EOP is not encountered. The BD machine can be halted alternately via the RESET button. The BD processor must be in the AUTO mode and running. Figure 4.11 shows an algorithmic description of the halt command.

7) Single Step BD PC. This command permits the operator to generate single clock pulses to advance the BD PC one instruction at a time for program checkout purposes. The routine is invoked with the "/S SS" command where SS_{16} is the starting address of the BD program to be executed. The subroutine presets the BD PC to SS_{16} and executes the instruction residing in this location. Single stepping is controlled from the keyboard by hitting any key except the "carriage return", which

```
PROCEDURE  HALT.BD.CLOCK
     IF BD RUNNING AND IN AUTO MODE THEN
          -  DISABLE BD CLOCK AT NEXT EOP BY RAISING STOP FLAG
RETURN
```

figure 4.11 Halt BD clock algorithm.

```
PROCEDURE  SINGLE.STEP.BD.PC
     IF BD STOPPED AND IN AUTO MODE THEN
        BEGIN
          -  READ STARTING ADDRESS AND PRESET BD PC
             REPEAT UNTIL A "CR" IS READ FROM KEYBOARD
                BEGIN
                  -  EXECUTE INSTRUCTION AT PC
                  -  INCREMENT PC
                  -  READ USER RESPONSE FROM KEYBOARD
                END
        END
RETURN
```

figure 4.12 Single step BD algorithm.

aborts the subroutine. The operator monitors the execution sequence via the BD console control lights to verify the program operation. The BD processor must be in the AUTO mode and in the stopped state on entry to the subroutine. Figure 4.12 presents a description of the algorithm.

4.2.3.1 Interface Device Drivers. These device drivers implement the basic communication functions of the BD interface module and remotely activate the control circuits of the BD processor.

1) PRSTPC. This subroutine presets the BD program counter remotely to a value passed by the calling routine of PRSTPC in the A register of the MC6809. Prior to sending the PC data, the direction of the 16-bit interface data bus is configured for mP to BD communications by the subroutine TOBD. The $\overline{\text{LOAD}}$ line of the PC is lowered by storing the code 17_{16} into the interface-based BD control register. PC data is put on the data bus by performing a STORE operation of the contents of accumulator A into the PIA data register PIAORA, and is toggled into the program counter by strobing the CA2 line of the PIA. This generates a clock pulse which latches the data into the PC.

The latter half of this subroutine, that part which strobes the BD clock, is also callable as a separate subroutine called STEP which is generally used to strobe the BD clock.

2) TOBD. This subroutine configures the PIA data registers for data output on both the A and B sides. This is accomplished by storing the code 39_{16} in each of the PIA control registers to access the PIA data direction registers. These are configured for data output by storing

the code FF_{16} in each one. The data output registers are restored by storing the code $3D_{16}$ in the PIA control registers. A complete description of PIA programming techniques and codes is presented in the MC6821 Product Information Sheet in the Motorola MC6800 Application Handbook [Moto75].

3) TOMP. This subroutine configures the PIA data registers for data input on both the A and B sides. The procedure is similar to the TOBD subroutine except that the code 00_{16} is stored in the data direction registers to configure the PIA for input.

4) LDWRD. This subroutine transfers one instruction from the mP to the BD program memory into the location currently addressed by the BD PC. The bidirectional interface data bus is configured for mP to BD communication by storing the code 37_{16} into the 8-bit BD control register. The BD instruction word is passed to LDWRD in the D accumulator. It is transferred to the data bus by executing a STORE of the accumulator into PIAORA and PIAORB. The BD R/$\overline{W}$ line is strobed, to write the data into BD memory by storing first the code 35_{16} and then $3D_{16}$ into PIACRB. This lowers the CB2 control output of the PIA which is interfaced to the R/$\overline{W}$ line.

5) RDWRD. This subroutine transfers one instruction from the BD program memory to the mP. The data bus is configured for BD to mP communication and the data is read from the bus by executing LOAD Accumulator A and B instructions from the PIA data registers PIAORA and PIAORB. The subroutine returns the data in the D accumulator.

4.2.4 Miscellaneous Utility Subroutines. In addition to the RTX and MMU, a number of utility operator commands are implemented in BDBUG. These are described below.

1) Report BD Status. This routine is invoked with the "/R" command. It reads the current auto/manual and running/stopped status of the BD processor and reports the information to the CRT.

2) Events Printer Control. This routine controls the on/off status of the system events printer. When on, all data sent to the CRT is echoed to the printer. This is a toggle-type control. It is invoked with the "/P" command.

3) Transfer to V-BUG or FLEX. These commands, invoked with the "/>" and "/+" codes, respectively, execute orderly exits from BDBUG to either V-BUG or FLEX. The MC6809 interrupt circuitry is restored to pre-BDBUG state by executing a return-from-interrupt (RTI) instruction to the warm-start addresses of V-BUG or FLEX as appropriate.

4) Inhibit Output with "Esc" Key. This subroutine allows data output to the CRT or Events Printer to be temporarily inhibited or aborted from the keyboard with the "Esc" key. The first activation of "Esc" stops the output. A subsequent hit resumes the output. Output may be aborted by striking "Carriage Return" while in the inhibited mode.

CHAPTER 5

BINARY DECISION PROGRAM OPTIMIZATION

5.0 Introduction

This chapter describes a BD program optimization method that has been developed based upon a pattern matching algorithm (PMA). The PMA algorithm reduces BD program logic of combinatorial switching functions to near-minimum form by an exhaustive search for redundancies. The method has not yet been generalized to include sequential logic.

5.1 Binary Decision Program Optimization

As shown previously, any combinatorial or sequential switching function can be represented by a binary decision program. Generating a complete BD program from a truth table or state transition table is a trivial but lengthy task, since program size grows exponentially (in the case of combinatorial functions) as 2^n-1 decision instructions and 2^n output instructions for a n-variable program.

In most control situations, however, the number of unique outputs is less than 2^n, implying that the complete program can be optimized by the elimination of redundant decision instructions and outputs. Furthermore, certain infrequently occurring outputs can be deliberately omitted from a BD program if the controlled process has a long time-constant. (Should such a condition occur, the BD processor can generate a system interrupt and have the omitted control outputs downloaded from

the microprocessor to deal with the condition.)

The BDC-4 (Binary Decision Compiler - version 4) program is an MC6809 assembly language program that generates a set of BD instructions from common control logic descriptions such as truth table, Boolean function, or relay- or gate-logic diagram. The BD object code is optimized to remove all redundant information. The output is a BD program load module ready to be downloaded from the microprocessor to the BD program memory. The process by which this is achieved is illustrated in figure 5.1. In the first step, switching function logic is entered in a high level format such as a Boolean equation, etc. This description is then reduced to an equivalent truth table by a set of subroutines which interpret these input formats and generate the required tables.

Subsequently, the truth table data is mapped into a BD program table which is operated on by the optimization algorithm to minimize the table size. The reduced BD table is then converted to BD machine code. The algorithms which perform these other functions are described individually in following sections of this chapter.

5.2 Optimizing Compiler Design

Control logic descriptions are reduced to BD programs in a five step process. The main functions of each step are described below.

5.2.1 Initialization. Compile time parameters for the target BD program are entered into the compiler in the initialization section.

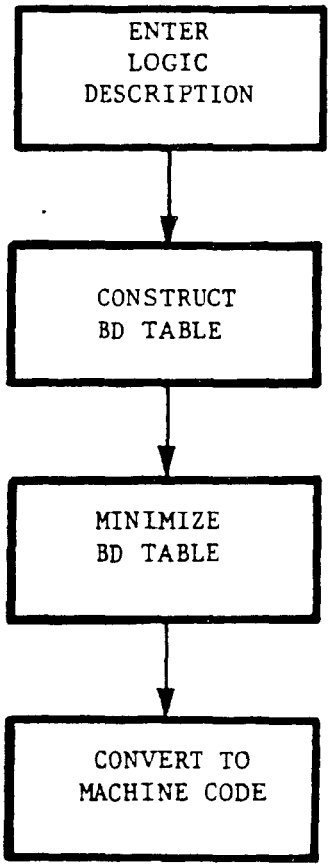


figure 5.1 BD program compiler block diagram.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

← OUTPUT VECTOR

figure 5.2 Truth table output vector.

The compiler begins by writing a title page on the operator's terminal which identifies the program and version number and lists the data entry format options supported by that version. In the current version, only truth table data entry format is supported. BD program parameters are then requested and entered via an interactive question and answer routine. Necessary BD program parameters include:

- BD program number
- BD program class
- the number of independent switching variables (n)
- the number of field outputs
- input terminal assignment for each variable
- output bank assignment

All input data is checked for out-of-range and bad-format errors which cause the program to repeat the data request message and to reread the bad data. Control is passed to the logic function input routine at the end of the initialization process.

5.2.2 Truth Table Generation. A truth table output vector is that part of a truth table in which the outputs corresponding to each combination of the input variables are stored, figure 5.2. The input routine begins by calculating the number of output data elements to input from the keyboard. It then writes a data input request message which specifies the number of elements expected and the required input format. The routine parses the input data stream into 4-digit hexadecimal numbers and writes a carriage return/line-feed code to the terminal after every complete number. After every four numbers, an

additional line feed is sent, figure 5.3.

The input data is stored in the first column of a 2^n row by 2 column array in the working memory area of the program. All input data is checked for out-of-range and bad-format errors. An error message is issued in the event of an input error and the data is reread.

After all of the data elements have been successfully read, the routine calculates the amount of storage area used by the array and allocates the work area for the BD table routine. Control is then passed to that routine.

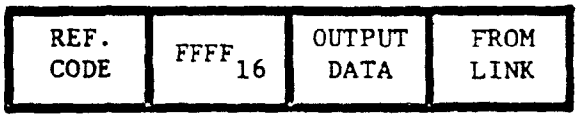
5.2.3 BD Table Generation. This section generates a doubly linked list data structure called a BD table to represent the BD program. Each of the $2^{n+1}-1$ records in the binary-complete BD table comprise four fields: a reference number, a FALSE-condition pointer, a TRUE-condition pointer, and a FROM pointer, figure 5.4a. The reference number is a four-digit hexadecimal (hex) number which names the input variable associated with that BD instruction. The first two digits from the left are an index to the input terminal assignment table constructed in the compiler initilization section previously described. During the machine code translation phase, the symbolic variable name is replaced with the actual address of the input in the 64-bit input register stored in this table. The second two digits enumerate the occurences of a variable in the BD table. Thus every instruction has a unique reference number.

The binary decision transfer logic is represented by the FALSE branch and TRUE branch pointers contained in the second and third fields of the BD table record. Each of these four-digit hex numbers refers to

```
ENTER 16 DATA WORDS IN HEX

0002
0004
0004
0004

0001
0002
0004
0004

0001
0001
0002
0004

0001
0001
0001
0002
```

figure 5.3 Control function input format.

(a)



(b)

figure 5.4 BD table record format. (a) Input instruction format and, (b) output instruction format.

the location in the BD table of the next sequential BD instruction, corresponding to the FALSE and TRUE outcomes, respectively, of the input variable. The pointer fields contain the physical memory addresses of the subsequent instructions to facilitate indexing into the table. The last field links the current BD instruction with its logical predecessor to enable bidirectional program traversal for the logic optimization section of the compiler. The field contains the physical memory address of the preceding instruction in the BD table.

In the case of an output BD instruction, the format of a BD table record is somewhat different, figure 5.4b. The reference number is composed as above except that the first subfield is assigned the value n for all output instructions, where n is the number of input variables comprising the control function. Output instruction occurrences are enumerated in the same manner as transfer instructions by the second subfield of the reference number. The next field is assigned the hex value FFFF. (This serves to further distinguish them from transfer instructions. The actual value has no significance.) The third field in the BD table record contains the particular output data corresponding to the output instruction's location in the truth table output vector. The first output instruction in the BD table is associated with the first entry in the output vector, the second with the second, etc. The last field in the BD table record is once again a link field, containing the physical memory address of the preceding instruction in the table.

Switching function logic is mapped into the BD table by storing the contents of the output vector into appropriate output instructions. Since the output vector is a linear, contiguous data structure, the

output data is more easily processed by the optimizing algorithm within the output vector rather than the BD table. However the algorithm must redirect transfer pointers in the BD table based on the output vector operations to eliminate redundant instructions. Hence, a return index into the BD table is stored in the second column of the output vector array to indicate the location of the output instruction in the BD table containing the output vector data, figure 5.2. The physical memory address of each output instruction is stored by the table generating routine immediately after such an instruction is created.

The BD table is generated with a preorder stack traversal algorithm [Stan80]. Preorder is usually defined, in the context of binary trees, as the traversal of the root node followed by the preorder traversal of the subtrees in left to right order, figure 5.5. In terms of BD programs, preorder refers to the generation of an instruction followed by the preorder generation of the FALSE and TRUE outcome instructions in that order. Since preordering is defined recursively, the TRUE branch is not generated until the entire subprogram rooted by the FALSE branch is complete. The purpose of a stack traversal is to save on a pushdown stack, the set of TRUE branches whose generation is postponed by the preorder generation of the FALSE branches. TRUE branches are removed from the stack on a last-in, first-out basis for processing. The algorithm ends when the stack contains no more postponed TRUE branches. A formal statement of the algorithm is presented below as Algorithm 5.1.

Algorithm 5.1. Preorder stack traversal of BD programs.

1. Let ROOT be a pointer to the first record in an empty BD table. Store ROOT in the FROM field of the first record, i.e.,

FROM(ROOT) <- ROOT.

2. If ROOT has the value $FFFF_{16}$ then go to step 5.

3. Generate a BD table record in the location pointed to by ROOT.

4. Save the table address of ROOT on the stack to postpone the generation of the TRUE branch subtable until the FALSE branch is completely processed, i.e., ROOT -> STACK. Link the FALSE branch to the next consecutive table location and store ROOT in the FROM field of that next location, i.e., FALSE(ROOT) <- ROOT+1 and FROM(ROOT+1) <- ROOT. The FALSE branch of an output record is assigned the value $FFFF_{16}$. If the current record represents an output instruction then store ROOT in the output vector table beside the appropriate output data element. Store the value of the data in the output instruction. Increment ROOT and go to step 2, i.e., ROOT <- ROOT+1.

5. Remove a postponed record from the stack and save the value of ROOT in the TRUE branch field. Store the address of the postponed instruction in the FROM field of ROOT, i.e., TRUE(STACK) <- ROOT and FROM(ROOT) <- STACK. If the stack is empty, then terminate the algorithm.

A BD table generated by algorithm 5.1 is illustrated in figure 5.6a along with the corresponding BD diagram, figure 5.6b. Reference numbers of BD instructions in the table are shown in parentheses beside the matching BD diagram nodes.

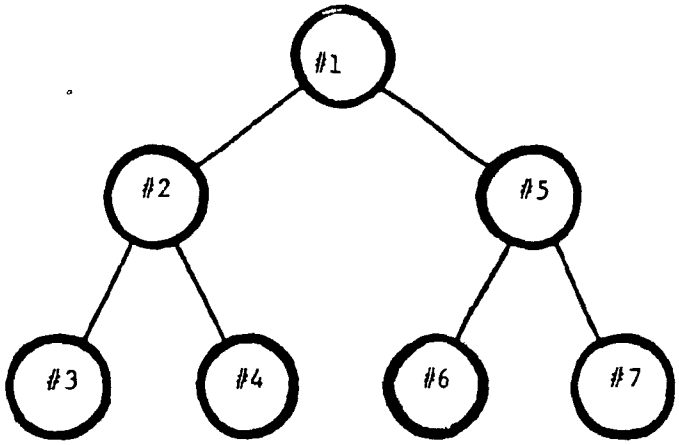
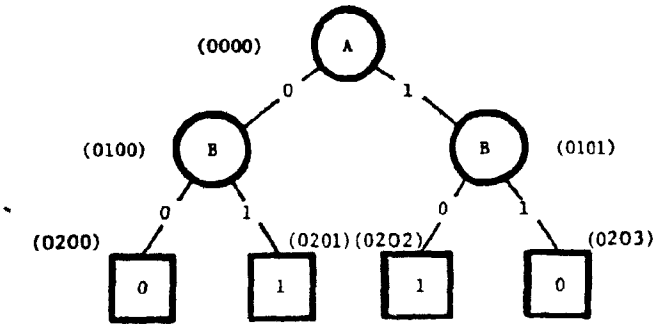


figure 5.5 Preorder BD diagram traversal.

mP ADDRESS	REFERENCE NUMBER	FALSE BRANCH	TRUE BRANCH	FROM BRANCH
1686	0000	168E	16A6	1686
168E	0100	1696	169E	1686
1696	0200	FFFF	0000	168E
169E	0201	FFFF	0001	168E
16A6	0101	16AE	16B6	1686
16AE	0202	FFFF	0001	16A6
16B6	0203	FFFF	0000	16A6

(a)



(b)

Figure 5.6 BD table representation of XOR function.
(a) BD table showing branching logic.
(b) BD diagram produced from table.

Reference numbers are generated by the algorithm such that if the symbolic variable number is 0 to n-1, a transfer instruction is generated. If the number is n, then an output is created. The reference number is incremented every time a new branch is traversed. The result is that all of the instructions which embody a common input variable have the same symbolic variable number. These appear in a BD diagram as the set of nodes comprising one level of the diagram, figure 5.6b.

Following the generation of the complete BD table, control is passed to the optimizing algorithm described next.

5.2.4 BD Table Optimization. BD programs can be reduced in size if more than one of the combinations of the input variables yield identical output states. Optimization is desirable since it significantly reduces program storage requirements, one of the main disadvantages of BD methods where binary-complete program length is of exponential order in the number of input variables.

Consider an eight input AND gate, figure 5.7a. A full BD program would have 511 program steps. Since there are only two possible output states and the FALSE state is generated in all cases except one, the program can be pruned to less than a dozen steps without losing any information. A reduced BD program is illustrated by the BD diagram of figure 5.7b.

The compiler uses a pattern recognition algorithm which searches for repetitious patterns in the truth table output vector data to guide the BD table optimization procedure. The algorithm systematically forms

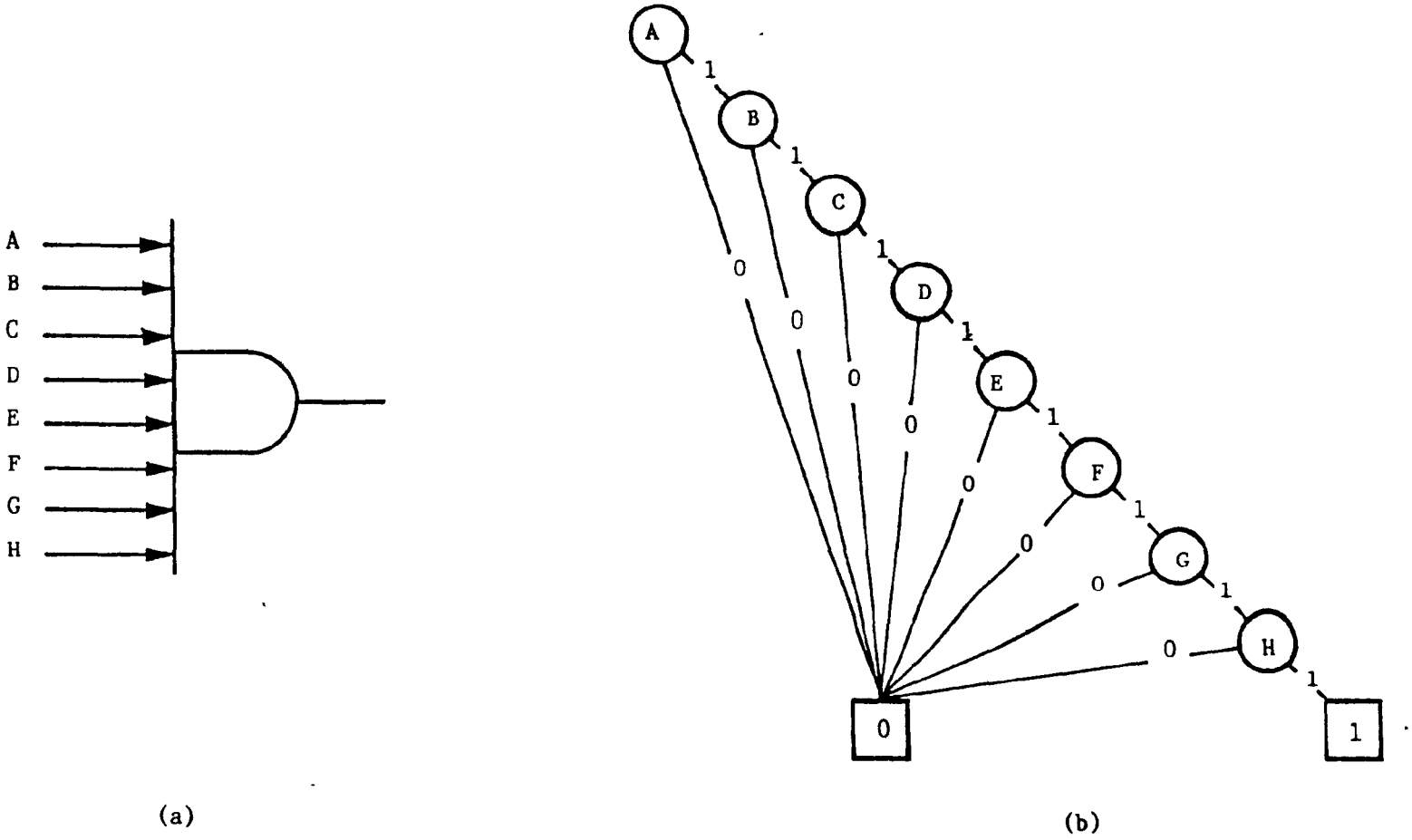


figure 5.7 Minimization of an 8-input AND gate program.
(a) 8-input AND gate and, (b) minimized program.

subsets of the output vector and searches for identical subsets in the remaining elements. Once a corresponding subset is found, the BD table is traversed to the root instructions predecessing the redundant output sets, figures 2.9a and b. The transfer pointers of the instructions are then rerouted as required to eliminate redundant logic from the BD table. The duplicate set of output data elements in the output vector are marked as being eliminated so that they are not subsequently reexamined by the algorithm.

Since the number of instructions that can be removed by a successful matching of duplicate output patterns is proportional to the size of the comparison group, the algorithm begins with the largest possible subset of the output vector, length(vector)/2, and then reduces it in later iterations. In this way all possible combinations of output data elements are exhaustively compared to ensure that an optimally reduced BD program is obtained. An analysis of the Pattern Matching Algorithm efficiency is presented in section 5.3 of this chapter. A formal statement of the PMA is given below as Algorithm 5.2.

Algorithm 5.2. Pattern Matching Algorithm for BD Program optimization.

1. Divide output vector into two subsets. Result is 2 subsets each of 2^{n-1} elements.
2. Compare every combination of these subsets for identical data patterns.
3. Prune BD table where possible.

4. Subdivide subsets by 2. Result is 2^i subsets each of 2^{n-i} elements after the i^{th} iteration. Go to step 2. Continue until i=n. Stop after this iteration.

As discussed, the BD table pruning routine uses the FROM pointers to traverse the program backwards to find the BD instructions at the root of both the subprogram defined by the pattern subset and the duplicate subset of output elements. The root of a subprogram within a complete BD program is found $h=\log_2[\text{size(subset)}]$ levels above the output instructions, figure 5.8a. The subprogram is then eliminated from the BD table by redirecting the branch pointer of the root's predecessor from the redundant subprogram to the root of the pattern subprogram, figure 5.8b.

An example of the operation of the PMA algorithm is shown in figure 5.9. The function, defined by table 5.1, represents a 2-bit magnitude comparator. Two input numbers, A and B, are represented by the binary patterns X_1X_2 and X_3X_4 respectively. The comparator output {100,010,001} represents the conditions {A<B,A=B,A>B}, respectively. The full BD diagram for the function, figure 5.9a, is reduced by algorithm 5.2 to the locally optimum diagram of figure 5.9f in four iterations.

In the first iteration the output vector is divided into two subsets which are compared. No minimization is possible at this stage since the subsets are not identical, figure 5.9b.

In the next iteration, the output vector is further subdivided into four sets of four outputs, figure 5.9c. The first subset is selected as

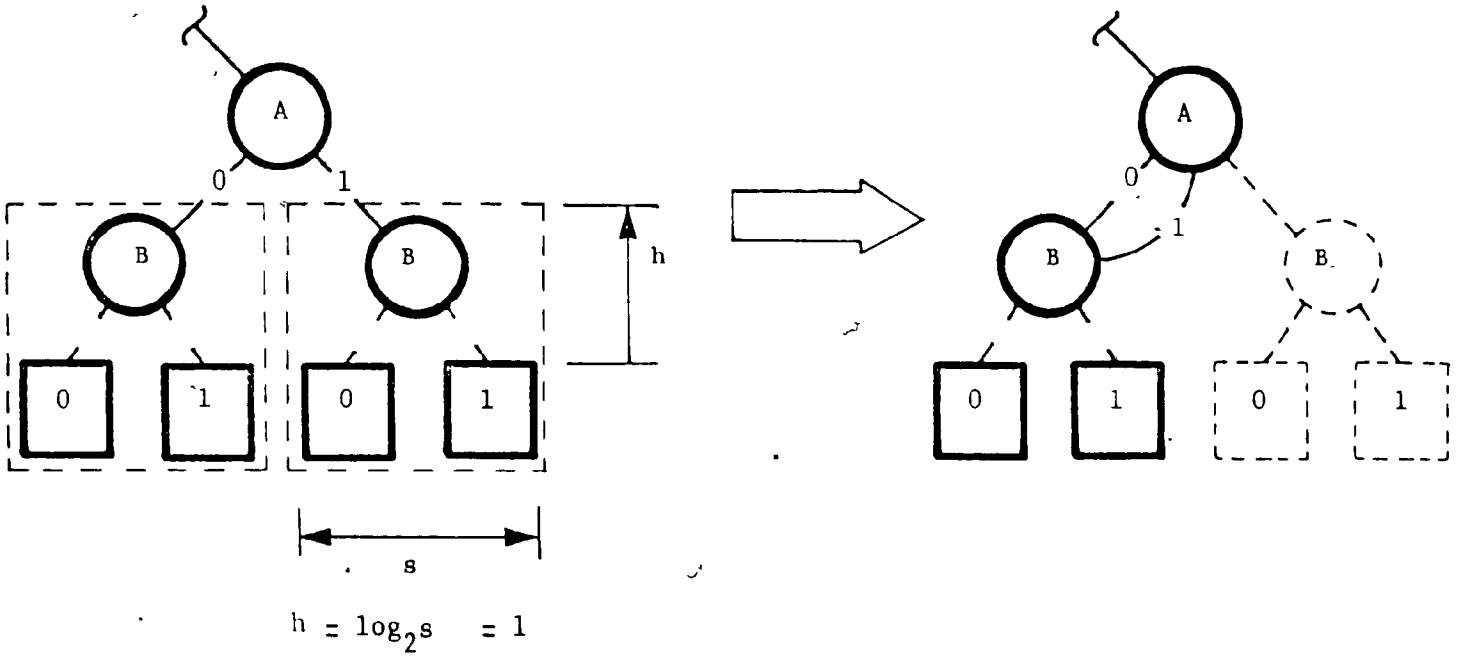


figure 5.8 Reduction of isomorphic program logic. (a) Location of root node. (b) Elimination of isomorphic group (shown in dashed lines).

X_1	X_2	X_3	X_4	Y_1	Y_2	Y_3	
A		B		A<B	A=B	A>B	
0	0	0	0	0	1	0	(2)
0	0	0	1	1	0	0	(4)
0	0	1	0	1	0	0	(4)
0	0	1	1	1	0	0	(4)
0	1	0	0	0	0	1	(1)
0	1	0	1	0	1	0	(2)
0	1	1	0	1	0	0	(4)
0	1	1	1	1	0	0	(4)
1	0	0	0	0	0	1	(1)
1	0	0	1	0	0	1	(1)
1	0	1	0	0	1	0	(2)
1	0	1	1	1	0	0	(4)
1	1	0	0	0	0	1	(1)
1	1	0	1	0	0	1	(1)
1	1	1	0	0	0	1	(1)
1	1	1	1	0	1	0	(2)

table 5.1 2-bit magnitude comparator truth table.

the pattern group and is compared with the remaining three. Again, no match is found. Next the second group is selected as the pattern and is compared with the remaining two for a match. (It is not necessary to compare the second pattern with the first again as no match was found when the two subsets were previously compared.) This test is unsuccessful as is the comparison of the third with the fourth group.

Continuing in algorithm 5.2, the output vector is divided into eight groups of two outputs all of which are exhaustively compared, figure 5.9d. Identical outputs are found between subsets 1-6, 2-4, 3-8 and 5-7. Figure 5.9d shows the duplicate output groups in dashed lines to indicate where they have been eliminated from the BD program. The pointers which led to the duplicate groups have been rerouted to the pattern group in each case.

In the final iteration, the output vector is divided into eight remaining groups, each of just one output, figure 5.9e. Many matches are found since there are only three distinct output values. Once again the nodes which have been eliminated are shown in dashed lines and the pointers have been rerouted. The final, fully reduced diagram is shown in figure 5.9f.

Following the optimization of the BD table, control is passed to the machine code instruction algorithm.

5.2.5 Machine Code Generation. The final part of the compiler involves putting actual BD operation code and memory addresses into the BD table instructions in place of the compiler reference addressing. The set of program parameters, including program number, number of parallel output channels, output bank assignment and interrupt codes, is used by the

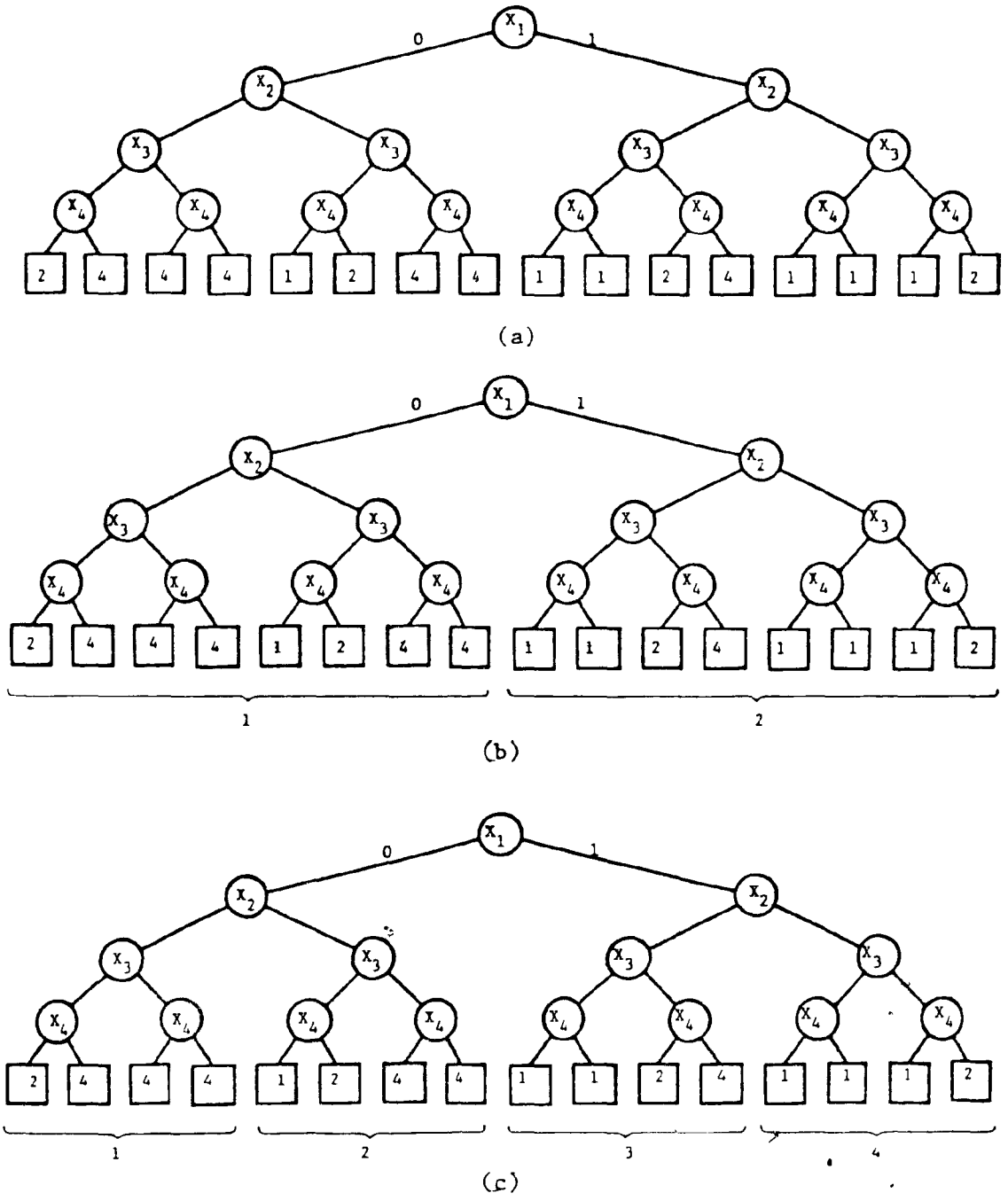
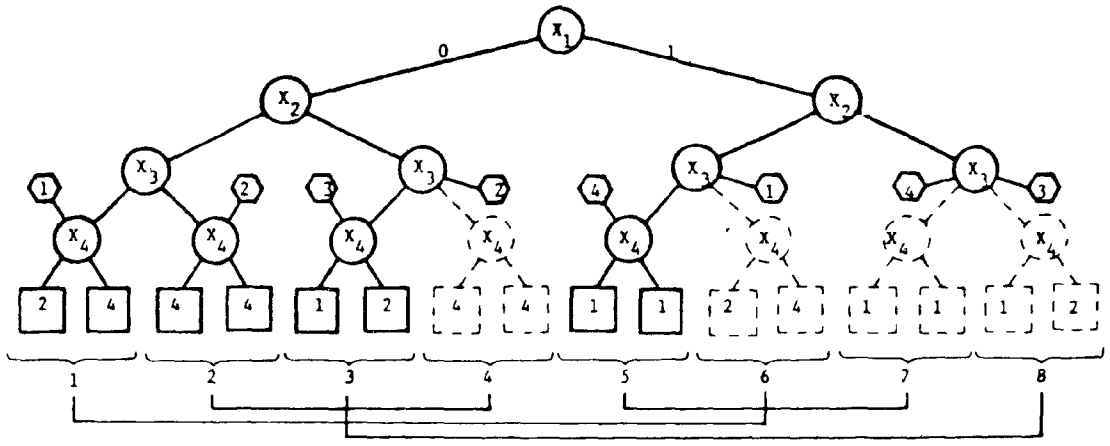
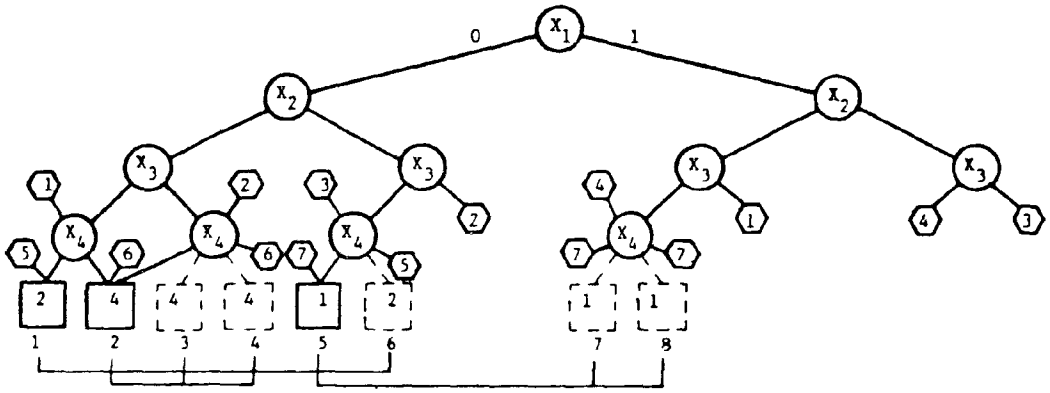


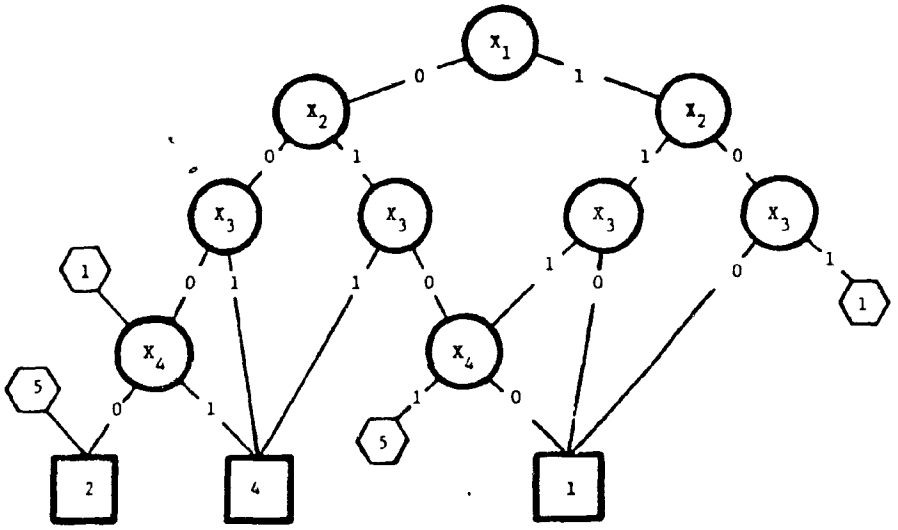
figure 5.9 PMA algorithm operation example. (a) Binary-complete BD diagram. (b) Division of output vector into groups of eight. (c) Division of output vector into groups of four. (d) Division of output vector into groups of two. PMA algorithm succeeds in matching patterns, resulting in elimination of isomorphic logic (shown in dashed lines). (e) Division of output vector into groups of one. Isomorphic logic is eliminated. (f) Final, fully reduced BD diagram.



(d)



(e)



(f)

figure 5.9 (cont'd)

compiler to generate control instructions. Since several forms of input and output instructions are defined in the instruction set (see chapter 3) of the BD processor, the compiler evaluates each situation in which more than one form would fit, and inserts the form which yields the shortest code. In addition an End-of-Program (EOP) instruction is inserted at the end of the BD program.

Machine code is generated in two passes. In the first pass, a preorder stack traversal algorithm is once again used to traverse the reduced BD table. As each record is processed, an equivalent BD machine code instruction is generated in a work area immediately following the BD table. Since actual BD transfer instructions contain just one branch address while the second is assumed, by convention, to be the next consecutive address, the branch-on-TRUE form of INPUT code is normally generated by the preorder traversal algorithm. If, however, the FALSE branch of a transfer instruction has already been generated and the TRUE branch has not, then the branch-on-FALSE form is used. This situation might occur if the FALSE branch pointer has been redirected by the pruning operation to an earlier section of the BD program, generated first according to the preorder traversal algorithm. If both the FALSE and TRUE branches of a transfer instruction are already generated, each requiring branching to previous parts of the program, then a branch-on-FALSE INPUT code is created followed by an unconditional JUMP to the TRUE address.

In many control programs, the OUTPUT instruction is the last code executed before a logical EOP. A branch to the EOP is thus included in every OUTPUT code. If SHORT OUTPUT instructions are specified by the

program parameters, the address of the EOP is inserted into IN<7:0> of the OUTPUT code. Otherwise, unconditional JUMP instructions are inserted in the program immediately following LONG OUTPUT codes. In either case, the actual address of the EOP may be unknown since it is the last instruction to be generated. Hence a temporary code is stored in the appropriate OUTPUT branch address location which is replaced by the actual address on the second pass through the BD machine code program.

A flowchart of the preorder stack traversal algorithm for the generation of BD processor machine codes is given in figure 5.10.

The finished program is saved on the secondary storage device in the BD program library under the program number specified by the programmer. An optional printout of the machine codes is available. A BD program for the 2-bit magnitude comparator function previously described in figure 5.9 is illustrated in figure 5.11. The complete BDC-4 program listing is presented in Appendix IV.

5.3 Optimizing Compiler Analysis

5.3.1 The Mathematical Basis of the PMA Algorithm. As discussed in chapter 2, the mathematical basis for the minimization of BD logic is derived from the Distributive and Complementation Properties of Boolean Algebra, i.e.:

$$A \cdot Q + B \cdot Q = (A + B) \cdot Q \qquad (5.1)$$

and,

$$\overline{A} \cdot Q + A \cdot Q = Q \qquad (5.2)$$

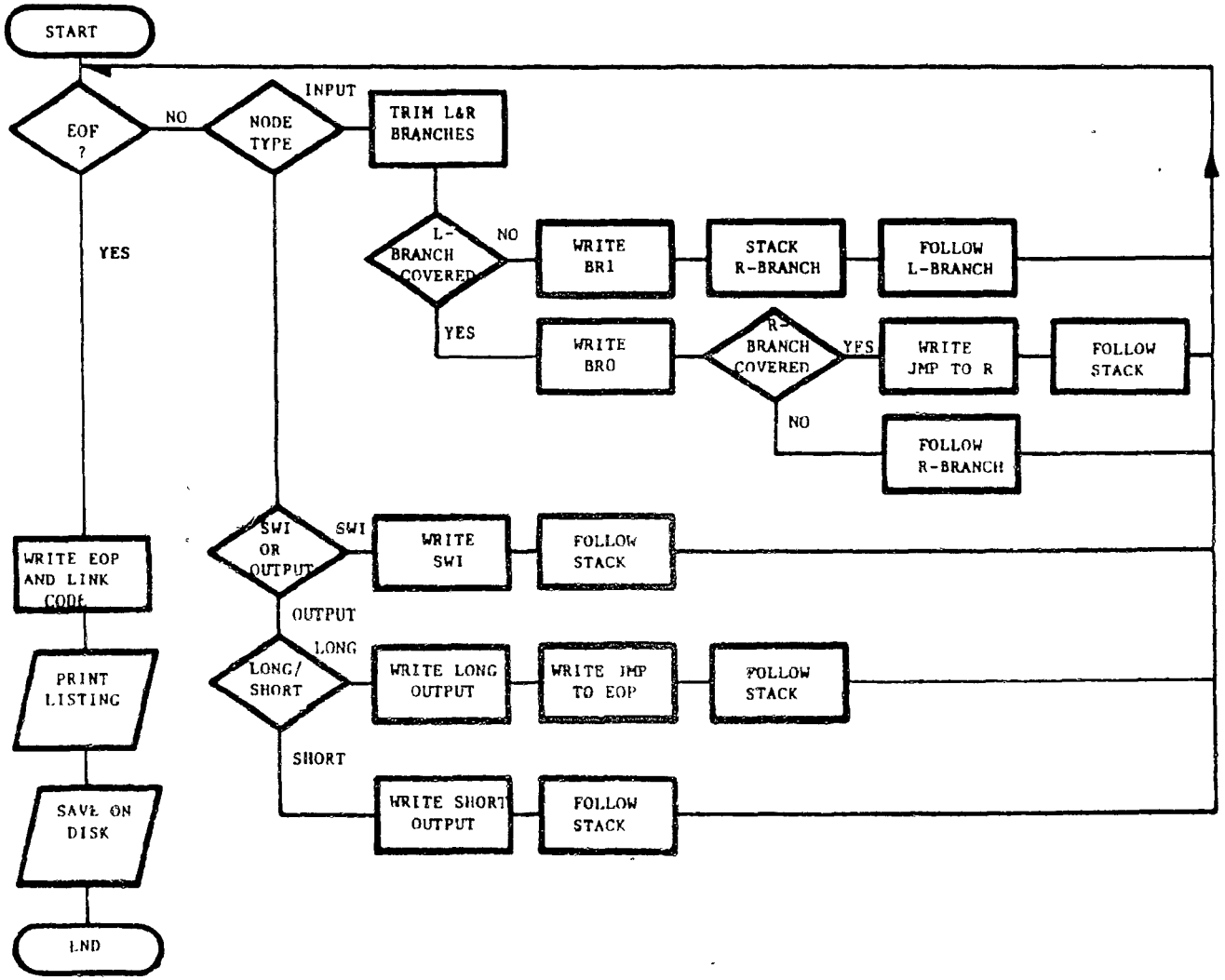


figure 5.10 Machine code generation subroutine flowchart.
(Flow direction is down and right.)

PC_{10}	$CODE_{16}$	COMMENTS
0	E101	SELECT BANK 1, JUMP TO 1
1	4009	INPUT X_1, JUMP TO 9 IF X_1=1
2	4407	INPUT X_2, JUMP TO 7 IF X_2=1
3	4806	INPUT X_3, JUMP TO 6 IF X_3=1
4	4C06	INPUT X_4, JUMP TO 6 IF X_4=1
5	C20F	OUTPUT "2", JUMP TO 15
6	C40F	OUTPUT "4", JUMP TO 15
7	080D	INPUT X_3, JUMP TO 13 IF X_3=0
8	F006	JUMP TO 6
9	440C	INPUT X_2, JUMP TO 12 IF X_2=1
10	4804	INPUT X_3, JUMP TO 4 IF X_3=1
11	C10F	OUTPUT "1", JUMP TO 15
12	080B	INPUT X_3, JUMP TO 11 IF X_3=0
13	0C0B	INPUT X_4, JUMP TO 11 IF X_4=0
14	F005	JUMP TO 5
15	A000	END OF PROGRAM

figure 5.11 BD machine code program for the 2-bit magnitude comparator.

The PMA algorithm searches the output vector to find instances in which either of these two properties apply. Equation 5.1 results in the recombination or trellising of logical transfer paths leading to identical subfunction in a BD program, figure 5.12a, while equation 5.2 results in the elimination of the BD instruction involving B and the unification of the two identical subfunctions, figure 5.12b. The common requirement of both these cases is to identify pairs of identical subfunctions within the BD logic.

The PMA algorithm method relates the logic of the subfunction to the pattern of outputs generated by the output nodes. An assumption is made that if the pattern of outputs of two subfunctions are identical, then the logic represented by the subfunctions must also be identical. This assumption is valid since the BD table is a binary-complete structure, i.e., each level of the BD diagram is homogenous in a variable, figure 5.13a. The subfunction indicated by the first dashed box in figure 5.13a may be described by the BD instructions:

Q_1 B; Q_2, Q_3

Q_2 C; a, b

Q_3 C; c, d

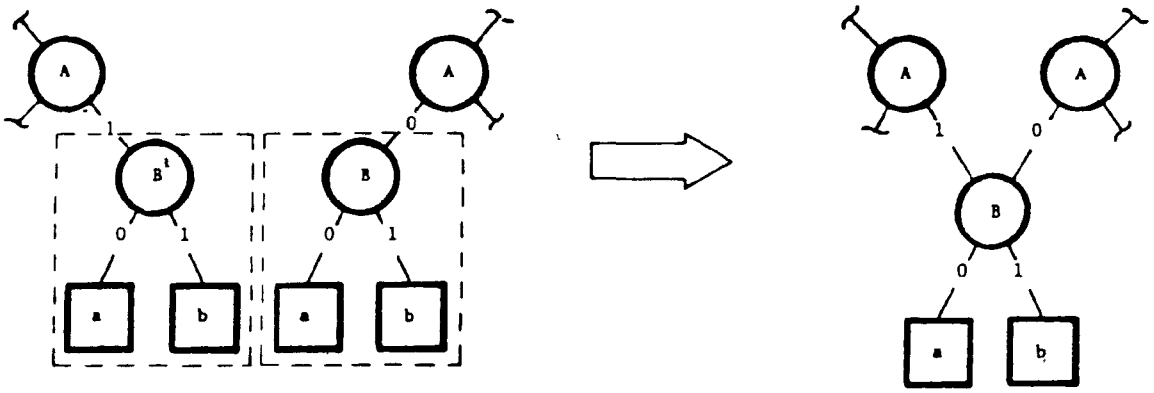
and the second by the instructions:

Q_4 B; Q_5, Q_6

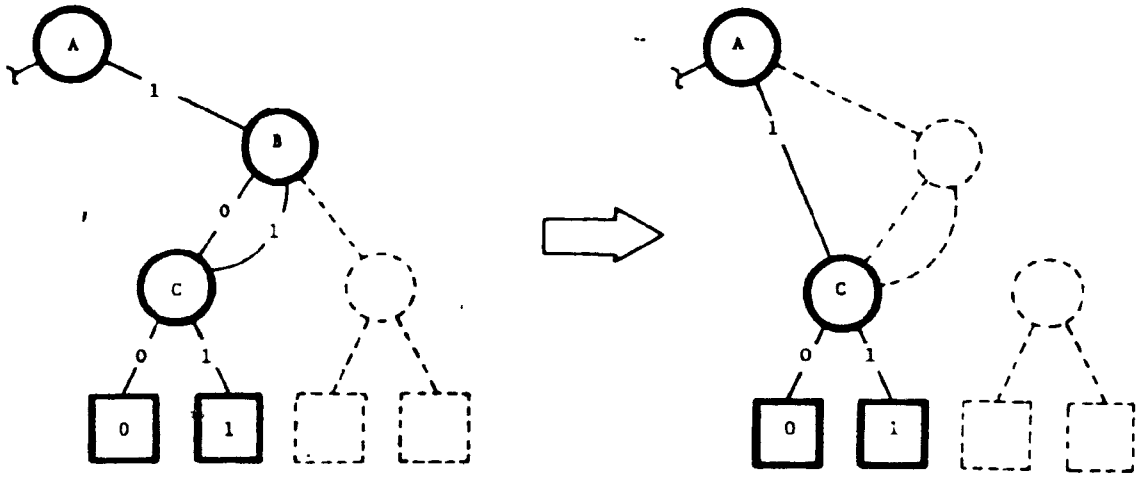
Q_5 C; a, b

Q_6 C; c, d

The logic represented by these two BD programs is identical, only the labels are different. However the subfunctions indicated by the dashed boxes in figure 5.13b, a non-binary-complete tree, are not identical as

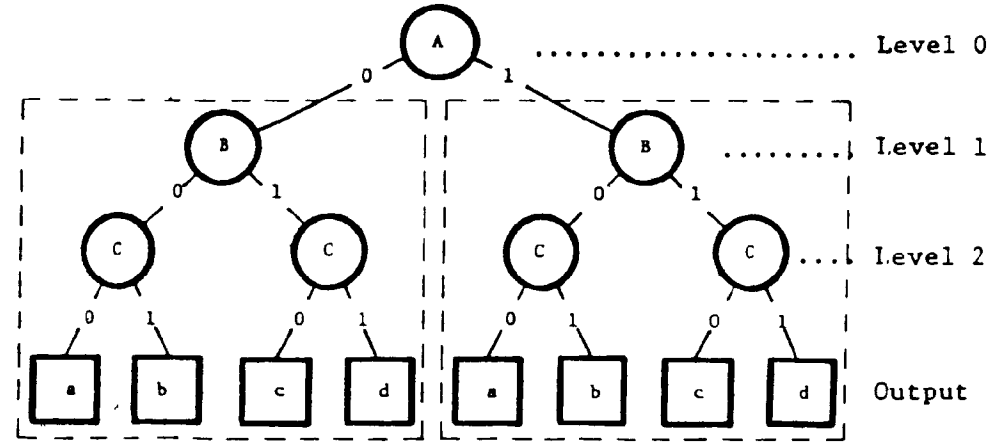


(a)

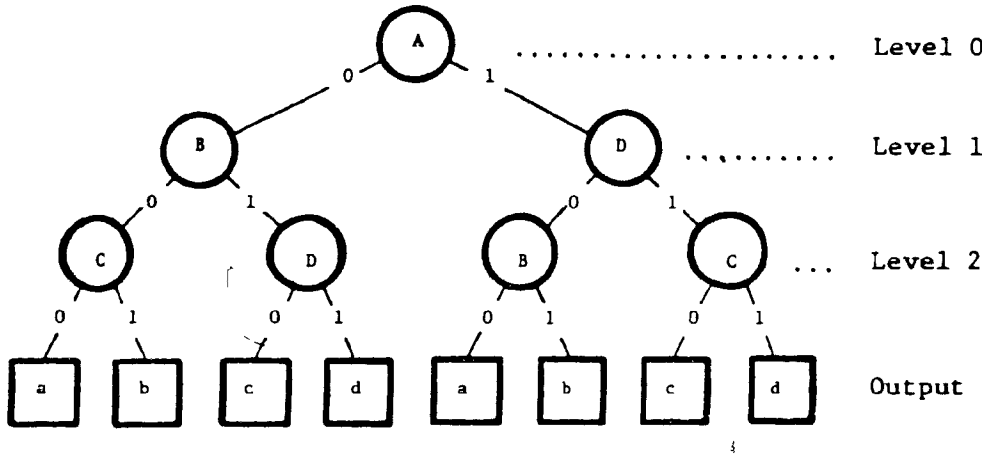


(b)

Figure 5.12 Logic reduction characteristics of the PMA algorithm. (a) Trellising of isomorphic logic and, (b) elimination of redundant logic.



(a)



(b)

figure 5.13 Affect of variable distribution on BD logic reduction. (a) Levels that are homogeneous in a variable are amenable to logic reduction. (b) Levels that are heterogeneous in a variable are not.

the levels are not homogenous in a variable. The subfunctions are described by the two following sets of instructions:

F_1 B; F_2, F_3

F_2 C; a, b

F_3 D; c, d

and,

F_4 D; F_5, F_6

F_5 B; a, b

F_6 C; c, d

Although the output patterns are similar, these subfunctions do not describe the same switching logic.

The usefulness of the PMA algorithm for the minimization of randomly generated BD programs is thus restricted. The approach is assured of success only when binary-complete programs are considered. Such programs are always obtained from the truth table representation of switching functions, however, and so the compiler is designed to reduce any of the other control function descriptions to the truth table form as a necessary first step.

5.3.2 Optimization Efficiency. Through its exhaustive search strategy, the PMA algorithm successfully finds and eliminates all of the logic redundancies in the binary-complete BD table. It can be shown, however, that the minimization efficiency depends on the initial choice of the order in which variables are evaluated by the BD program.

Consider the reduced program obtained by the PMA algorithm for the two-bit magnitude comparator in figure 5.9f. The minimized program examines the variables in the order X_1, X_2, X_3, X_4 and requires twelve

instructions including the output nodes, a 61% reduction from the complete program. Figure 5.14 depicts the minimized program for the same function in which the variables are tested in the sequence X_1, X_3, X_2, X_4. Here only nine instructions remain after minimization, a 71% reduction in size.

At any time, therefore, the PMA algorithm is only certain of finding a local minimum. Other algorithms are reported in the literature which claim to find the global minimum [Poll65,Schw74]. These employ some sort of variable reordering scheme to search for the one sequence of variables which yields the highest degree of optimization. Very little progress has been made, however, in discovering good methods of guiding the search procedure. The size of the search space can be estimated by the number of different BD programs which describe the same function. If the binary-complete program constraint is maintained, then there are n ways of choosing the first variable, n-1 ways of choosing the second, etc., i.e., n! different BD programs which describe the same function.

Consider the more general case in which the binary-complete restriction is relaxed. Each instruction is permitted to evaluate any of the variables as long as a variable is not tested more than once in a transfer path from root to output of the program, figure 5.15. From the diagram, it can be seen that there are n ways of choosing the first variable, n-1 independent ways of choosing each of the two variables at the second level, etc. In general, the number of ways of choosing a variable at the i^{th} level of the BD diagram is $(n-i)^{2^i}$ and so the total number of different BD programs which describe the same function is:

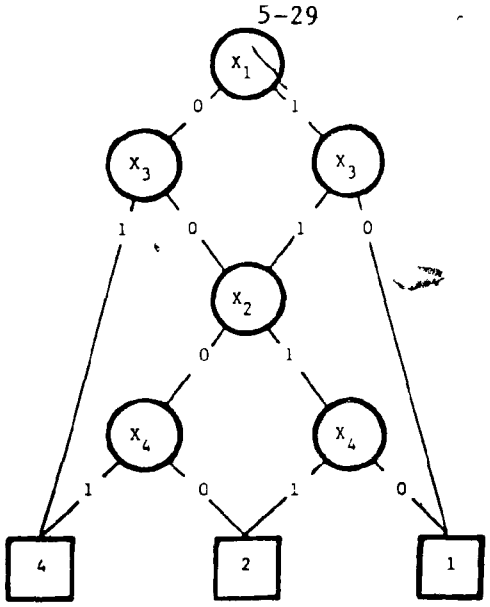


figure 5.14 Affect of variable ordering on BD logic reduction.

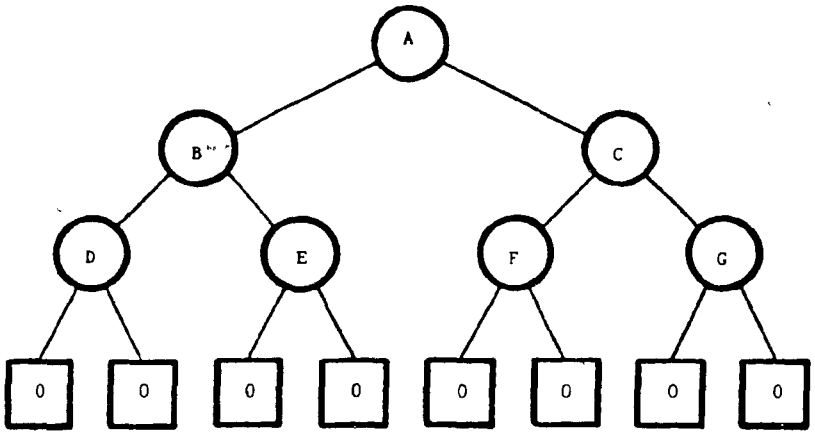


figure 5.15 Example of a non-binary-complete BD program.

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

This represents an enormous search space for the variable-reordering algorithms to search through to find the most favourable sequence. It is submitted that the objective of BD program optimization in the context of industrial process control should be to obtain an acceptable approximation to the global minimum with the least computational work and memory cost. With a sufficiently fast algorithm, real-time adaptive control is feasible. It was shown that BD program optimization without variable reordering is, at best, a NP-complete problem, meaning that the best algorithm still requires an exponential order amount of work. Research continues in the DATAC laboratory to determine the upper and lower bounds for optimization efficiency with and without variable reordering, and preliminary results indicate that the PMA algorithm is expected to find local minima within about 10% of the global.

5.3.3 Computational Time Complexity. The time complexity of an algorithm is a measure of the number of steps required to execute the algorithm as a function of the size of the input. If for a given size the complexity is taken as the maximum complexity over all inputs of that size, then this is called the worst-case complexity. The worst-case complexity of an algorithm is usually much easier to estimate than the average-case complexity since one does not have to make assumptions about the distribution of the inputs.

In the case of BD program minimization, the input size is often

taken to be the number of independent variables of the switching function. Since the PMA algorithm searches for redundant logic by comparison of output vector subsets, the worst-case complexity for a given vector is obtained when every element is exhaustively tested with every other. The number of comparisons made is reduced by successful logic pruning, since duplicate output patterns are eliminated from the output vector at each stage. However, the amount of reduction depends on the distribution of the output data elements, a difficult measure to quantify. The analysis presented below is for the worst-case time complexity of the PMA algorithm.

Definition: 5.1 Let k_i be the number of subsets into which the output vector is divided in the i^{th} iteration of the algorithm.

Definition: 5.2 Let m_i be the number of data elements subsumed by each subset in the i^{th} iteration.

During one iteration of the PMA algorithm, the first subset is compared amongst k_i-1 others, the second is compared amongst k_i-2 others, etc. A total of

$$(k_i-1) + (k_i-2) + \ldots + 1 = \sum_{j=1}^{k_i-1} (k_i-j) \qquad (5.3)$$

subsets are therefore compared for each iteration. The summation of the first k_i-1 terms of equation 5.3 is obtained below.

$$\sum_{j=1}^{k_i-1} (k_i-j) = \sum_{j=1}^{k_i-1} k_i - \sum_{j=1}^{k_i-1} j$$

$$= k_i(k_i-1) - k_i(k_i-1)/2$$

$$= k_i(k_i-1)/2$$

Every comparison between a pair of subsets involves m_i individual comparisons of data elements. Thus, the total number of individual comparisons made during one iteration of the PMA algorithm is:

$$m_i k_i (k_i - 1)/2 \qquad (5.4)$$

As described in section 5.2.4, the algorithm is iterated n times to completion for a n variable BD program. Hence, the total number of comparison steps executed is:

$$\sum_{i=1}^{n} m_i k_i (k_i - 1)/2 \qquad (5.5)$$

Now k_i is equal to 2^i, e.g., 2^1 subsets in the first iteration, $2^2=4$ in the second, etc. And m_i is equal to 2^{n-i}, e.g., each of the two subsets in the first iteration subsume $2^n/2$ outputs, which is reduced by a factor of 2 in each further iteration. Equation 5.5 is thus rewritten as:

$$\sum_{i=1}^{n} 2^{n-i}\, 2^i (2^i - 1)/2 \qquad (5.6)$$

The summation of the first n terms of equation 5.6 is obtained below.

$$\begin{aligned}
\sum_{i=1}^{n} 2^{n-i}\, 2^i (2^i - 1)/2 &= \sum_{i=1}^{n} 2^n/2\, (2^i - 1) \\
&= (2^{n-1}) \sum_{i=1}^{n} (2^i - 1) \\
&= (2^{n-1}) \left[\sum_{i=1}^{n} 2^i - \sum_{i=1}^{n} 1 \right] \\
&= (2^{n-1}) \left[2^{n+1} - 1 - n \right] \\
&= O(2^{2n}) \qquad (5.7)
\end{aligned}$$

(We say that g(n) is O(f(n)) if there exists constants c and n_0 such that g(n) is less than or equal to cf(n) for all n greater than or

equal to n_0.) The last two terms were eliminated to simplify the final step. Since

$$2^{2n} > (2^{n-1})[2^{n+1}-1-n]$$

the order of complexity is unaffected by this omission.

Karasick has designed a variation of the PMA algorithm that has a worst case complexity of $O(n2^n)$ [Kara84,Huds84]. The algorithm simultaneously builds and optimizes the BD program through a postorder traversal procedure, during which all nodes are visited but only those nodes that branch to existing nodes are created.

Since both of these algorithms are of exponential order, computer workload can be expected to increase rapidly with an increasing number of variables. The growth in computational workload for the PMA algorithm is illustrated in figure 5.16. The BDC-4 compiler limits the number of input variables of a problem to eight in order to bound the size of working memory. Significantly, compiler performance was not seen to degenerate when computing eight input variable problems, running on a 1 MHz 6809 computer.

5.3.4 Computational Space Complexity. The space complexity of an algorithm is a measure of the amount of work space required to execute the algorithm as a function of the size of the input.

The space complexity of the BDC-4 program is computed below. The output data entry routine stores the output vector in main memory consuming $O(2^n)$ locations. The BD table, which has length $O(2^{n+1})$, is similarly stored in working memory. It is overwritten by the PMA algorithm such that the reduced BD table does not use any extra memory.

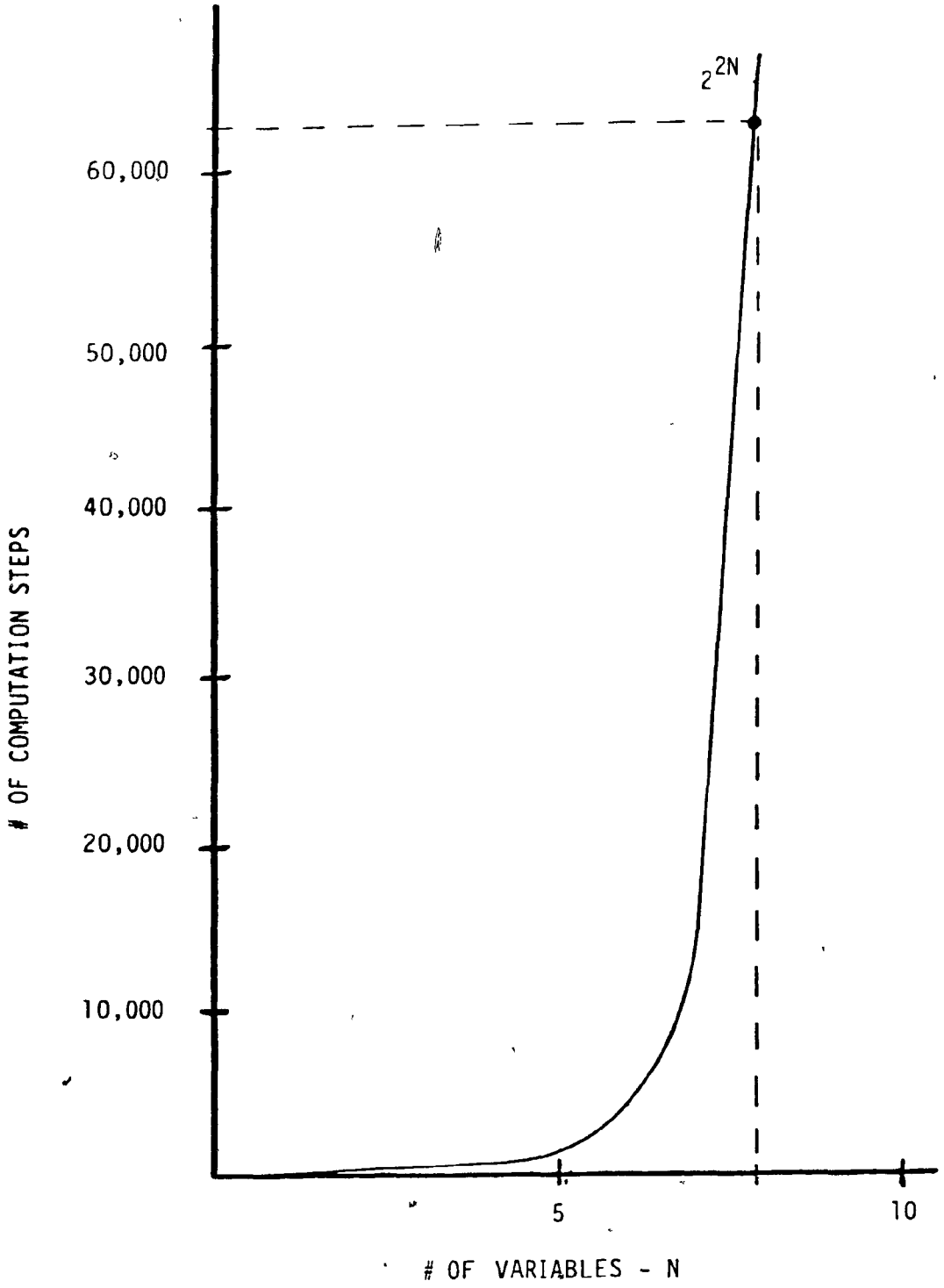


figure 5.16 PMA algorithm computational time complexity.

Finally, the machine code generator creates the executable BD object code which consumes O(size of program) memory. In the worst case, the size of program is once again $O(2^{n+1})$. Therefore the worst case space complexity of the BDC-4 compiler is:

$$O(2^n + 2^{n+1} + 2^{n+1}) = O(5 \cdot 2^n) \tag{5.8}$$

In comparison, Karasick's version of the PMA algorithm has a worst case space complexity of $O(2 \cdot 2^n)$. This improvement is achieved by generating the reduced BD program dynamically without physical reference to the binary-complete BD table. As well the output vector is not required in main memory as it is sequentially scanned only once to build the program. The variable reordering methods described in [Drie82] use extensive working area to describe each subfunction of the main program. In fact, an exponential number of tables are created, each of which uses $O(h^2)$ space in the height, h, of the subfunction represented. These tables are retained throughout the compilation to check for subfunction isomorphisms. This is another indication of the impracticality of the dynamic variable reordering algorithms in a real-time process control environment.

CHAPTER 6

mP/BD PLC APPLICATIONS

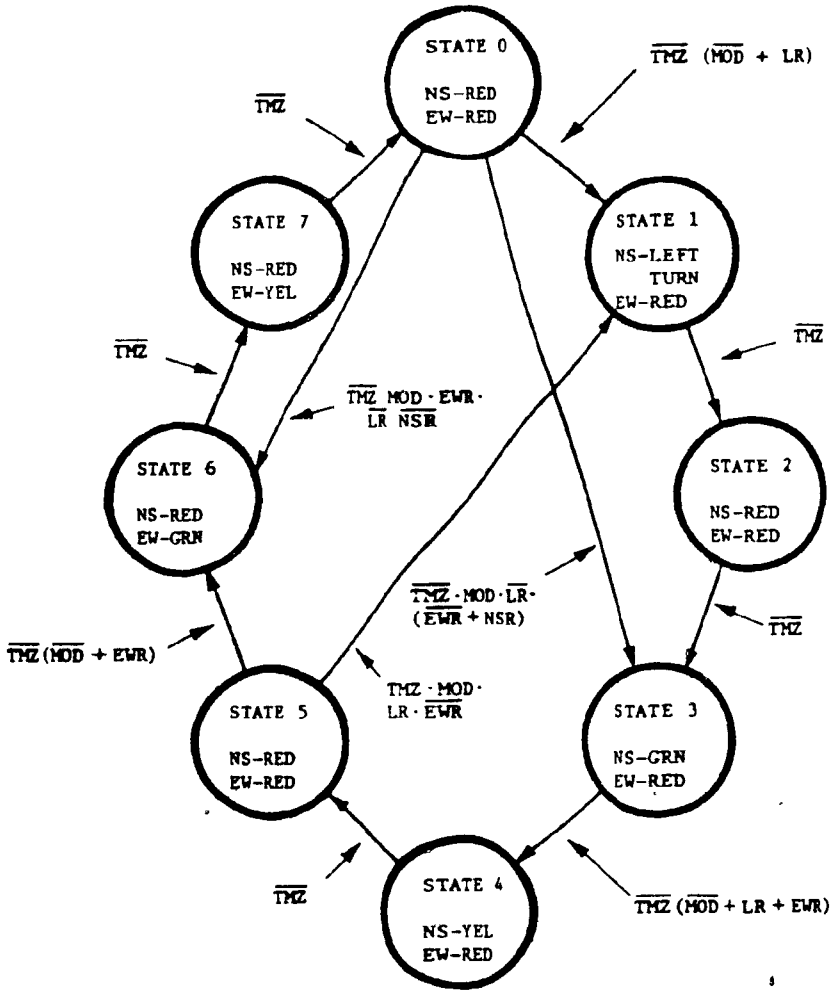
6.0 Introduction

Two examples are presented in this chapter that show how the mP/BD hybrid programmable controller can be applied to the realization of process control tasks. In both examples, the resultant mP/BD PLC programs are compared to programs prepared for a Motorola MC14500 single-bit microprocessor-based controller. The mP/BD-based controller implementations are consistently shorter and faster.

6.1 Example 1 - Traffic Intersection Controller

This example demonstrates the use of the BD processor for sequential automation tasks by the implementation of a traffic intersection controller. The state diagram of the control algorithm, described in the MC14500 handbook [Greg77], is shown in figure 6.1.

The traffic signal switching procedure is as follows. The North-South (NS) and East-West (EW) traffic flow directions are controlled by standard red-yellow-green traffic lights. Traffic may proceed directly through the intersection or turn right on the green signal, but left turns are prohibited. The NS direction is assumed to be a major thoroughfare, conveying large volumes of traffic at rush hours. For the convenience of NS vehicles, a priority left turn signal is provided, i.e., EW and straight-through NS traffic is halted while NS cars may



LEGEND

TMZ - TIMER ELAPSED
MOD - RUSH HOUR MODE
NSR - NORTH/SOUTH REQUEST
EWR - EAST/WEST REQUEST
LR - LEFT TURN REQUEST
NS - NORTH/SOUTH
EW - EAST/WEST
GRN - GREEN LIGHT
YEL - YELLOW LIGHT

figure 6.1 Traffic intersection controller state diagram.

turn left onto the EW street. To improve rush hour flow in the NS direction, the control system can be operated in two modes:

1) a regular mode in which the controller sequences through all of the states in the state diagram of figure 6.1, on a timed basis independent of traffic volume; and

2) a rush-hour mode in which the controller remains in the NS green state unless it receives an alternate sequence request from an attending operator. EW and left-turn requests are serviced for a timed period after which the controller returns to the NS green state.

The mP/BD PLC is capable of implementing this sequential automation control task in the BD processor. As the program is short, only 42 steps, it is not necessary to partition the control algorithm into subsections to be paged into and out of memory by the mP. Three hardware timers are assumed to be connected to the BD I/O banks in the manner of section 3.2.4.6. The timers are started by three field outputs, while three field inputs indicate timing interval expiration, table 6.1. The common field output, used to discriminate between the timer active and expired states, is not required in this example since only one timer is active at a time and the controller remains in the same state until the timed interval has expired. Other field inputs are assigned to a mode control switch and NS, EW or left turn request switches. Seven other field outputs operate the red, yellow and green lights in the two directions plus the left arrow light, table 6.2. The I/O terminal assignments are summarized in table 6.3.

DEVICE	FUNCTION	TIMER TRIGGER SIGNAL	TIMER OUTPUT SIGNAL
TIMER 1	NS GREEN	T_{NSI}	T_{NSO}
TIMER 2	EW GREEN AND LEFT ARROW	T_{EWI}	T_{EWO}
TIMER 3	RED OVERLAP AND YELLOW	T_{YI}	T_{YO}

table 6.1 External timer hardware for the traffic controller.

INPUTS		OUTPUTS	
SIGNAL	DESCRIPTION	SIGNAL	DESCRIPTION
T_{NSO}	TIMER OUTPUT	T_{NSI}	TIMER TRIGGER
T_{EWO}	TIMER OUTPUT	T_{EWI}	TIMER TRIGGER
T_{YO}	TIMER OUTPUT	T_{YI}	TIMER TRIGGER
NSR	NORTH/SOUTH REQUEST	R_{NS}	NS RED LAMP
		Y_{NS}	NS YELLOW LAMP
		G_{NS}	NS GREEN LAMP
EWR	EAST/WEST REQUEST	G_L	NS LEFT ARROW
MOD	REGULAR/RUSH HOUR MODE	R_{EW}	EW RED LAMP
		Y_{EW}	EW YELLOW LAMP
		G_{EW}	EW GREEN LAMP
LR	LEFT REQUEST		

table 6.2 BD processor I/O for the traffic controller.

SIGNAL	OUTPUT BIT	SIGNAL	INPUT CHANNEL
NOT USED	OV<11>	T_{NSO}	34
T_{NSI}	OV<10>	T_{EWO}	33
T_{EWI}	OV<9>	T_{YO}	32
T_{YI}	OV<8>	MOD	0
R_{NS}	OV<7>	NSR	1
Y_{NS}	OV<6>	EWR	2
G_{NS}	OV<5>	LR	3
R_{EW}	OV<4>		
Y_{EW}	OV<3>		
G_{EW}	OV<2>		
G_L	OV<1>		
NOT USED	OV<0>		

table 6.3 BD processor I/O terminal assignments for the traffic controller.

Binary decision diagrams for each control state in figure 6.1 are shown in figure 6.2. The input nodes are labelled with the variable tested by the node. Output nodes are labelled with the control signal data appropriate to each state according to the order, OV<11:0>. Outputs OV<11> and OV<0> are undefined in this example. Output bank 1 is used.

The BD program for this control task, figure 6.3, requires 42 instructions. In contrast, an MC14500-based control program, figure 6.4, requires 152 instructions. While execution speed is not an important parameter in this example, it is clear that the cycle time of the BD processor in each control state is significantly faster than the microprocessor-based program. In addition, the BD program could be partitioned into segments implementing the normal control mode and the rush hour mode separately. The mP could then transfer the appropriate program into BD memory on a pre-scheduled basis.

Example 2 continues on page 6-14.

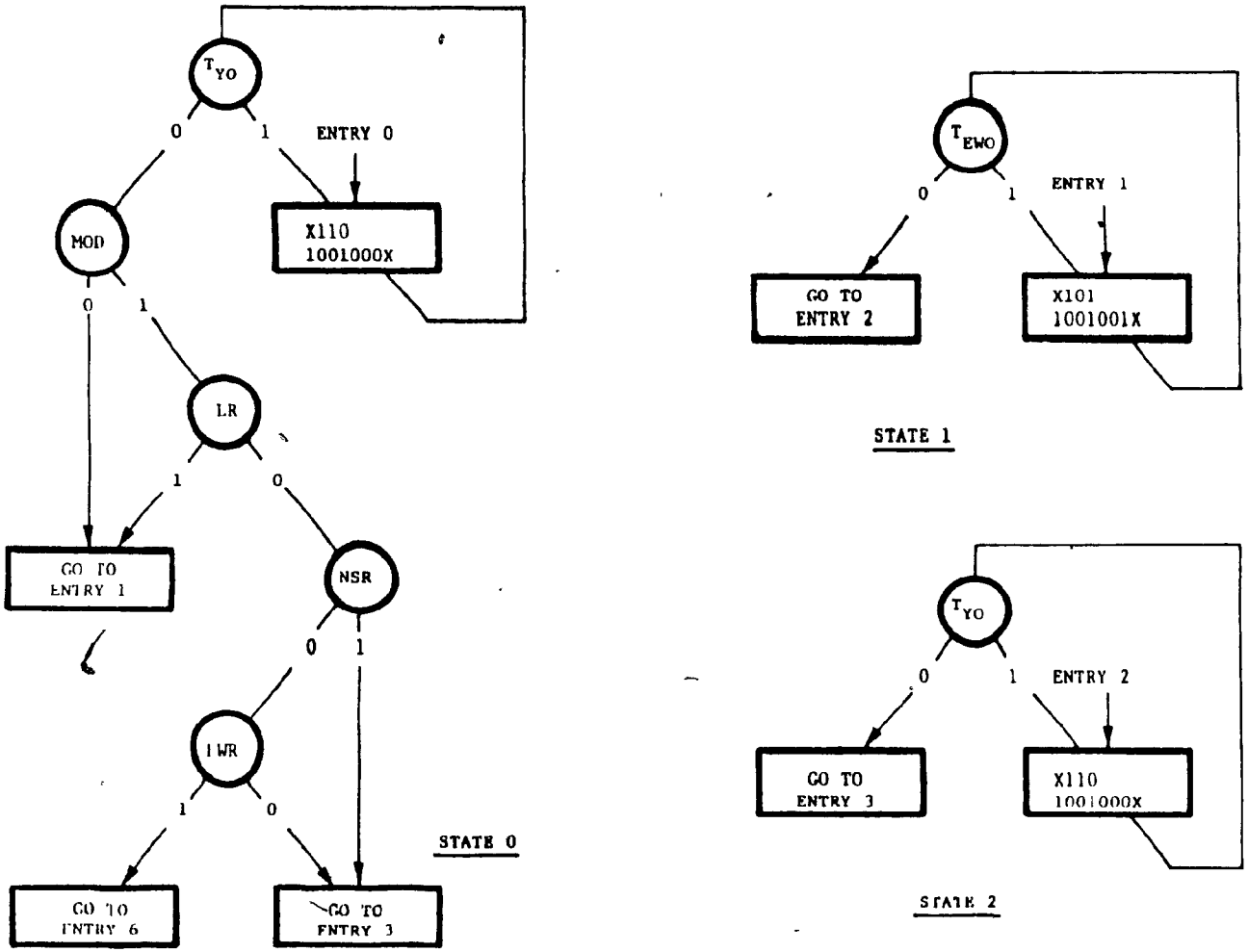
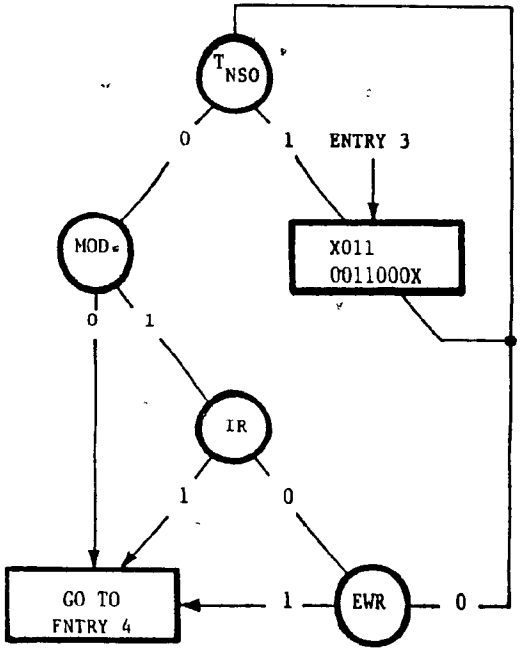
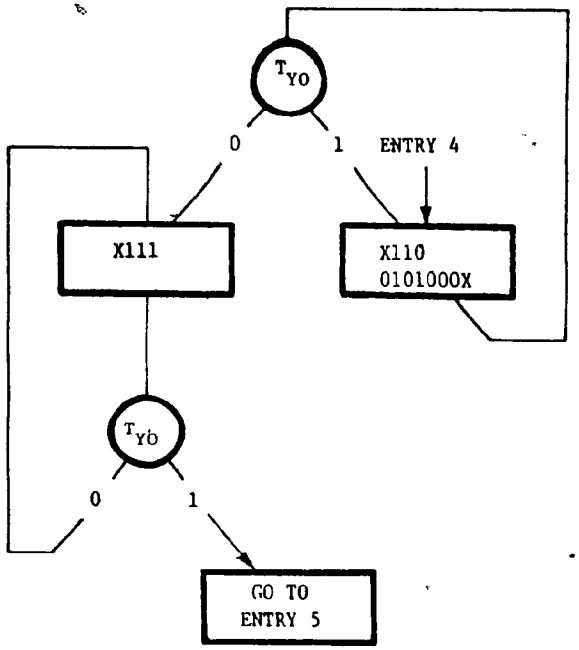


figure 6.2 BD diagrams for each traffic controller state.

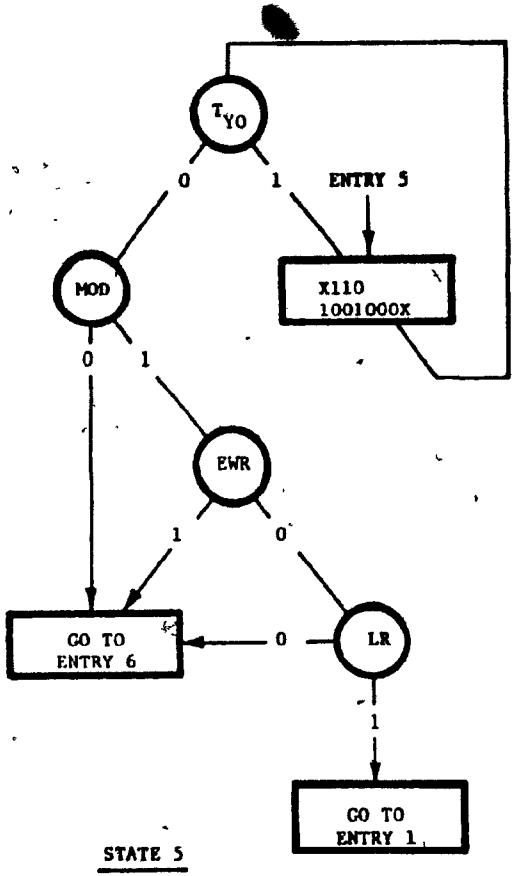


STATE 3

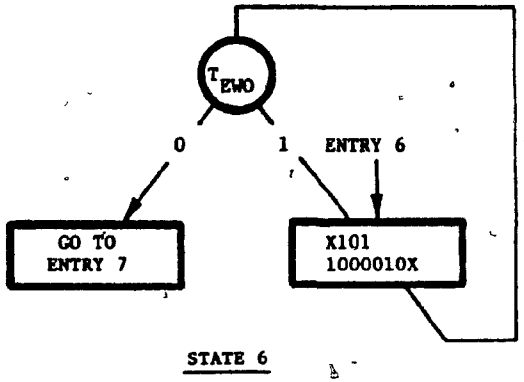


STATE 4

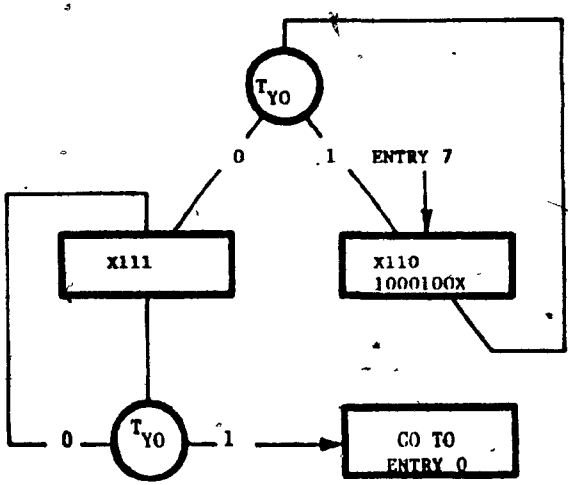
figure 6.2 (cont'd)



STATE 5



STATE 6



STATE 7

figure 6.2 (cont'd)

COMMENT	PC_{10}	$CODE_{16}$	DESCRIPTION
STATE 0	00	E101	SELECT OUTPUT BANK 1, JUMP TO 1
	01	0204	INPUT T_{YO}, JUMP TO 4 IF T_{YO}=0
ENTRY 0	02	8690	OUTPUT 011010010000
	03	F001	JUMP TO 1
	04	000F	INPUT MOD, JUMP TO 15 IF MOD=0
	05	4C0F	INPUT LR, JUMP TO 15 IF LR=1
	06	4408	INPUT NSR, JUMP TO 8 IF NSR=1
	07	4812	INPUT EWR, JUMP TO 34 IF EWR=1
STATE 3, ENTRY 3	08	8330	OUTPUT 001100110000
	09	4A08	INPUT T_{NSO}, JUMP TO 8 IF T_{NSO}=1
	10	0015	INPUT MOD, JUMP TO 21 IF MOD=0
	11	4C15	INPUT LR, JUMP TO 21 IF LR=1
	12	0809	INPUT EWR, JUMP TO 9 IF EWR=0
	13	F015	JUMP TO 21
STATE 1	14	0612	INPUT T_{EWO}, JUMP TO 18 IF T_{EWO}=0
ENTRY 1	15	8592	OUTPUT 010110010010
	16	F00E	JUMP TO 14
STATE 2	17	0208	INPUT T_{YO}, JUMP TO 8 IF T_{YO}=0
ENTRY 2	18	8690	OUTPUT 011010010000
	19	F011	JUMP TO 17
STATE 4	20	0217	INPUT T_{YO}, JUMP TO 23 IF T_{YO}=0
ENTRY 4	21	8650	OUTPUT 011001010000
	22	F014	JUMP TO 20
	23	C718	OUTPUT 0111, JUMP TO 24
	24	0217	INPUT T_{YO}, JUMP TO 23 IF T_{YO}=0
	25	F01B	JUMP TO 27
STATE 5	26	021D	INPUT T_{YO}, JUMP TO 29 IF T_{YO}=0
ENTRY 5	27	8690	OUTPUT 011010010000
	28	F01A	JUMP TO 26
	29	0022	INPUT MOD, JUMP TO 34 IF MOD=0
	30	4822	INPUT EWR, JUMP TO 34 IF EWR=1
	31	4C0F	INPUT LR, JUMP TO 15 IF LR=1
	32	F022	JUMP TO 34
STATE 6	33	0625	INPUT T_{EWO}, JUMP TO 37 IF T_{EWO}=0
ENTRY 6	34	8584	OUTPUT 010110000100
	35	F021	JUMP TO 33
STATE 7	36	0227	INPUT T_{YO}, JUMP TO 39 IF T_{YO}=0
ENTRY 7	37	8688	OUTPUT 011010001000
	38	F024	JUMP TO 36
	39	C728	OUTPUT 0111, JUMP TO 40
	40	0227	INPUT T_{YO}, JUMP TO 39 IF T_{YO}=0
	41	A000	EOP, JUMP TO 0

figure 6.3 BD program for the traffic controller.

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
	00	XNOR	RR	ENABLE INPUTS
	01	IEN	RR	
STATE 0	02	LD	LR	IF [STATE 0·$\overline{TMZ}$·($\overline{MOD}$ + LR)]
	03	ORC	MOD	THEN
	04	ANDC	B0	
	05	ANDC	B1	
	06	ANDC	B2	
	07	ANDC	$\overline{TMZ}$	
	08	OEN	RR	
	09	STO	B0	
	10	STO	T_{EWI}	
	11	STO	PE	FLAG STATE 1
	12	STOC	PE	START LEFT ARROW TIMER
	13	STOC	T_{EWI}	
	14	STO	G_L	TURN ON LEFT ARROW
	15	LD	NSR	IF [STATE 0·$\overline{TMZ}$·MOD·$\overline{LR}$·
	16	ORC	EWR	(EWR + NSR)] THEN
	17	ANDC	LR	
	18	AND	MOD	
	19	ANDC	$\overline{TMZ}$	
	20	ANDC	B0	
	21	ANDC	B1	
	22	ANDC	B2	
	23	OEN	RR	
	24	STO	B0	
	25	STO	B1	
	26	STO	T_{NSI}	FLAG STATE 3
	27	STO	PE	START NS GREEN TIMER
	28	STOC	PE	
	29	STOC	T_{NSI}	
	30	STO	G_{NS}	TURN ON NS GREEN
	31	LDC	B0	IF (STATE 0·$\overline{TMZ}$·EWR·$\overline{LR}$·
	32	ANDC	B1	$\overline{NSR}$·MOD) THEN
	33	ANDC	B2	
	34	AND	MOD	
	35	AND	EWR	
	36	ANDC	LR	
	37	ANDC	NSR	
	38	ANDC	$\overline{TMZ}$	
	39	OEN	RR	
	40	STO	B1	
	41	STO	B2	FLAG STATE 6
	42	STO	T_{EWI}	START EW GREEN TIMER
	43	STO	PE	
	44	STOC	PE	

figure 6.4 MC14500 program for the traffic controller.

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
	45	STOC	T_{EWI}	
	46	STO	G_{EW}	TURN ON EW GREEN
STATE 1	47	LD	B0	IF (STATE 1·$\overline{TMZ}$) THEN
	48	ANDC	B1	
	49	ANDC	B2	
	50	ANDC	$\overline{TMZ}$	
	51	OEN	RR	
	52	STOC	B0	
	53	STO	B1	FLAG STATE 2
	54	STOC	G_L	TURN OFF LEFT ARROW
	55	STO	T_{YI}	START RED OVERLAP TIMER
	56	STO	PE	
	57	STOC	PE	
	58	STOC	T_{YI}	
STATE 2	59	LDC	B0	IF (STATE 2·$\overline{TMZ}$) THEN
	60	AND	B1	
	61	ANDC	B2	
	62	ANDC	$\overline{TMZ}$	
	63	OEN	RR	
	64	STO	B0	FLAG STATE 3
	65	STO	T_{NSI}	START NS GREEN TIMER
	66	STO	PE	
	67	STOC	PE	
	68	STOC	T_{NSI}	
	69	STO	G_{NS}	TURN ON NS GREEN
STATE 3	70	LD	EWR	IF [STATE 3·$\overline{TMZ}$·
	71	OR	LR	($\overline{MOD}$ + LR + EWR)] THEN
	72	ORC	MOD	
	73	ANDC	$\overline{TMZ}$	
	74	AND	B0	
	75	AND	B1	
	76	ANDC	B2	
	77	OEN	RR	
	78	STOC	B0	
	79	STOC	B1	FLAG STATE 4
	80	STO	B2	
	81	STO	Y_{NS}	TURN ON NS YELLOW
	82	STOC	G_{NS}	TURN OFF NS GREEN
	83	STO	T_{YI}	START NS YELLOW TIMER

figure 6.4 (cont'd)

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
	84	STO	PE	
	85	STOC	PE	
	86	STOC	T_{YI}	
STATE 4	87	LDC	B0	IF (STATE 4·$\overline{TMZ}$) THEN
	88	ANDC	B1	
	89	AND	B2	
	90	ANDC	$\overline{TMZ}$	
	91	OEN	RR	
	92	STO	B0	FLAG STATE 5
	93	STOC	Y_{NS}	TURN OFF NS YELLOW
	95	STO	PE	
	96	STOC	PE	
	97	STOC	T_{YI}	
STATE 5	98	LD	EWR	IF [STATE 5·$\overline{TMZ}$·($\overline{MOD}$ + EWR)]
	99	ORC	MOD	THEN
	100	ANDC	$\overline{TMZ}$	
	101	AND	B0	
	102	ANDC	B1	
	103	AND	B2	
	104	OEN	RR	
	105	STOC	B0	
	106	STO	B1	FLAG STATE 6
	107	STO	G_{EW}	TURN ON EW GREEN
	108	STO	T_{EWI}	START EW GREEN TIMER
	109	STO	PE	
	110	STOC	PE	
	111	STOC	T_{EWI}	
	112	LD	B0	IF (STATE 5·$\overline{TMZ}$·MOD·LR·$\overline{EWR}$)
	113	ANDC	B1	THEN
	114	AND	B2	
	115	ANDC	$\overline{TMZ}$	
	116	AND	MOD	
	117	AND	LR	
	118	ANDC	EWR	
	119	OEN	RR	
	120	STOC	B2	FLAG STATE 1
	121	STO	G_L	TURN ON LEFT ARROW
	122	STO	T_{EWI}	START TIMER
	123	STO	PE	
	124	STOC	PE	
	125	STOC	T_{EWI}	

figure 6.4 (cont'd)

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
STATE 6	126	LDC	B0	IF (STATE 6·$\overline{TMZ}$) THEN
	127	AND	B1	
	128	AND	B2	
	129	ANDC	$\overline{TMZ}$	
	130	OEN	RR	
	131	STO	B0	FLAG STATE 7
	132	STO	Y_{EW}	TURN ON EW YELLOW
	133	STOC	G_{EW}	TURN OFF EW GREEN
	134	STO	T_{YI}	START TIMER
	135	STO	$\overline{PE}$	
	136	STOC	PE	
	137	STOC	T_{YI}	
STATE 7	138	LD	B0	IF (STATE 7·$\overline{TMZ}$) THEN
	139	AND	B1	
	140	AND	B2	
	141	ANDC	$\overline{TMZ}$	
	142	OEN	RR	
	143	STOC	B0	
	144	STOC	B1	FLAG STATE 0
	145	STOC	B2	
	146	STOC	Y_{EW}	TURN OFF EW YELLOW
	147	STO	T_{YI}	START RED OVERLAP TIMER
	148	STO	$\overline{PE}$	
	149	STOC	PE	
	150	STOC	T_{YI}	
	151	NOPF		SET FLAG TO RESET PC

figure 6.4 (cont'd)

6.2 Example 2 - Industrial Boiler Control

This example demonstrates the use of the complete mP/BD PLC for both proportional control and sequential automation tasks in parallel.

An industrial facility will usually have one or more single-burner, gas- or oil-fired, industrial package boilers supplying low-pressure steam for process heating, cooling, drying, and space heating [Wood77,B&W75]. Such package boilers are fitted with simple measurement and control devices used to maintain steam pressure and fuel and air flows. In addition, they are provided with some sequential logic to automate startup and shutdown procedures. Typical inputs and outputs for a single boiler system are shown in table 6.4.

Boiler controls are traditionally implemented with pneumatic or electric-analog hardware as well as with relay or digital hardware for the logic tasks. The capital cost of this hardware may be disproportionate to the value of a single-boiler system, since boiler controls are usually designed for complicated multiboiler, multiburner, multifuel installations. Since modern distributed microprocessor-based control systems are also designed for large applications, they present the same economic disadvantage. A hybrid mP/BD PLC system, however, is well-suited to a less complicated installation. Proportional control tasks can be processed digitally in the mP while sequential automation functions can be implemented in the BD processor. Supervisory control is maintained by the mP in conjunction with the operator control interface.

INPUTS TO MICROPROCESSOR	STEAM HEADER PRESSURE STEAM FLOW SENSOR (OR FUEL FLOW) AIR FLOW SENSOR DRUM LEVEL VARIOUS SETPOINTS AND LIMITS
OUTPUTS FROM MICROPROCESSOR	FORCED DRAFT FAN DAMPER POSITION FUEL CONTROL VALVE POSITION FEEDWATER CONTROL VALVE POSITION
INPUTS TO BD PROCESSOR	FURNACE DRAFT LIMITS DRUM LEVEL LIMITS AIR FLOW LIMITS FUEL VALVE LIMITS FLAME DETECTOR SIGNALS STOP/START SIGNALS
OUTPUTS FROM BD PROCESSOR	PURGE SEQUENCE STATUS BURNER LIGHT-OFF STATUS FUEL VALVE INTERLOCK COMMANDS ALARM ANNUNCIATOR COMMANDS
BOILER TYPE:	SINGLE BURNER, GAS-FIRED, PACKAGE BOILER 12 MW (50,000 PPH), 1.7 MPa (250 psi) PRESSURE

table 6.4 Typical package boiler field I/O requirements.

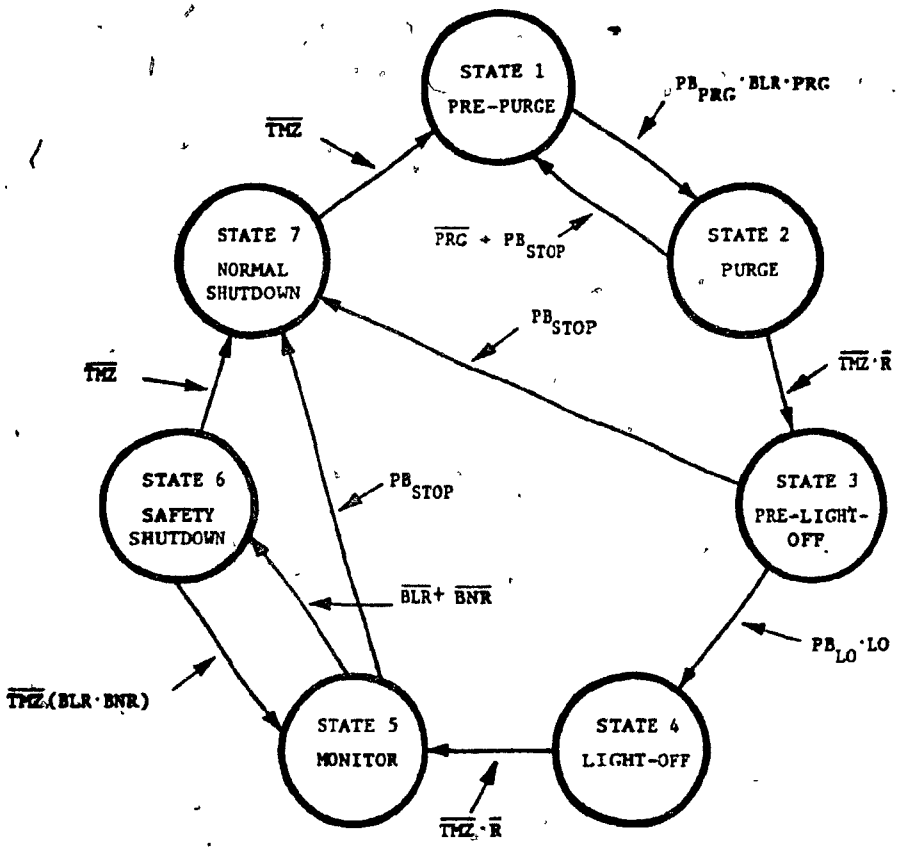
DEVICE	FUNCTION	TIMER TRIGGER SIGNAL	TIMER OUTPUT SIGNAL
TIMER 1	PURGE TIMER	T_{PRGI}	T_{PRG}
TIMER 2	LIGHT-OFF TIMER	T_{LOI}	T_{LO}
TIMER 3	SAFETY DELAY TIMER	T_{SDI}	T_{SD}
TIMER 4	POST-PURGE TIMER	T_{PPI}	T_{PP}

table 6.5 External timer hardware for the burner automation controller.

6.2.1 Burner Automation System Functions. The burner automation system for a common single-burner, gas-fired 12 MW (50,000 PPH) steam capacity, 1.7 MPa (250 psi) steam pressure, industrial package boiler is shown in the state diagram of figure 6.5. The control sequence is characterized by seven states:

1) Pre-purge state. The purge cycle replaces the air inside the boiler furnace with outside air before the boiler is lit to remove any fuel vapors which may cause an explosion during burner ignition. In the pre-purge state, the control system ensures that the fuel valve is closed, that no flame is detected in the furnace, that the furnace airflow is sufficient for purging and that the boiler is ready. These are the purge permissives. When the system proves that these conditions exist, it illuminates the Purge-Ready-Lamp to alert the operator. It then awaits the activation of the Start-Purge-Pushbutton before entering the purge state.

2) Purge state. The operator starts a purge cycle timer which is pre-set for an appropriate timing period based on the furnace volume and the purge air flow-rate. At least eight complete air changes must be accomplished according to the NFPA safety code [NFPA76]. The control system illuminates a Purge-in-Progress Lamp to indicate the control state. The purge cycle may be interrupted if the purge permissives are lost for a period longer than two seconds or if the operator presses the Stop-Pushbutton. In this case the control system returns to the pre-purge state. At the end of the two minute purge period, providing that no interruption occurred, the control system illuminates the Purge-



LEGEND

PB_{PRG}	-	START PURGE P.B.
PB_{LO}	-	START LIGHT-OFF P.B.
PB_{STOP}	-	STOP P.B.
$\overline{TMZ}$	-	TIMER ELAPSED
R	-	TIMER RESET
BLR	-	BOILER PERMISSIVES
BNR	-	BURNER PERMISSIVES
PRG	-	PURGE PERMISSIVES
LO	-	LIGHT-OFF PERMISSIVES

figure 6.5 Burner automation system state diagram.

Complete Lamp and enters the pre-light-off state.

3) Pre-light-off state. The control system ensures that the burner is in the proper state prior to burner ignition by checking that the automatic fuel shut-off valve is closed and that the fuel supply pressure is within limits. These are called the pre-light-off permissives. The Burner-Ready-Lamp is illuminated when these are proven. The operator can interrupt the light-off procedure if the permissives cannot be proven after a reasonable period by pressing the Stop-Pushbutton. Otherwise, the control system awaits the Start-Light-Off-Pushbutton to enter the light-off state.

4) Light-off state. The automatic fuel shut-off valve is opened to admit fuel to the burner and the electric spark igniter is energized. The igniter is operated for a period of ten seconds after which the control system turns off the Burner-Ready-Lamp and enters the burner-monitor state.

5) Burner-monitor state. This is the normal operating-state of the control system while the burner is lit. The monitor continuously checks the condition of the burner permissives and the boiler permissives to ensure that the unit is operating within approved safety limits. If any of the permissives are lost, the control system enters the safety shutdown state. A normal shutdown is invoked by the activation of the Stop-Pushbutton.

6) Safety shutdown state. A two second timer is activated. After this delay, the burner and boiler permissives are rechecked to ensure that

the safety hazard still exists and was not due to a momentary signal loss. If the permissives can be reestablished the control system returns to the monitor state and resumes normal operation. Otherwise, an Alarm Lamp is illuminated and the normal shutdown sequence is initiated.

7) Normal shutdown state. To safely shutdown the boiler, the automatic fuel shut-off valve is closed. However, the normal air flow is maintained to remove unburned fuel and potentially hazardous combustion products from the furnace. This is called the post-purge cycle. The control system activates a post-purge timer and awaits the end of this period after which it shuts down the air fan and returns to the pre-purge state.

The burner automation task is executed in the BD processor. Once again, it is not necessary to partition the program into subsections since it is relatively short, only 94 instructions. Eighteen process inputs and fourteen outputs are used. The outputs are broken down into three categories, timers and flip-flops, indicator lamps, and boiler equipment commands. These are assigned to banks 0, 1, and 2, respectively. The I/O terminal assignments are summarized in tables 6.5, 6.6 and 6.7.

BD diagrams for each control state are shown in figure 6.6. The input nodes are labelled with the process variable tested by each node and output nodes are labelled with the control actions appropriate to each state according to the order OV<11:0>. The BD program for this control task is shown in figure 6.7. Some optimization has been carried out by hand, e.g., the flip-flop used in state 2 saves the re-

examination of the purge and boiler permissives by independent instructions. It causes the program flow to loop back after the timing period expires and to execute the existing set of input instructions a second time. On the second pass, the program flow bypasses the timer and branches to the shutdown state if the purge permissives cannot be reestablished.

In contrast, the MC14500-based control program requires 180 instructions, figure 6.8. The BD program could be further optimized to combine the safety-shutdown state with the normal shutdown and place the safety timer logic within the main monitor loop. In this case, the number of physical state transitions would be reduced and would suggest a program partitioning into startup, monitor, and shutdown sequences. The mP could then transfer the appropriate program into BD memory on an interrupt basis.

6.2.2 Combustion Control System. The combustion control system is responsible for maintaining the air flow and fuel flow into the furnace in the proper ratio to ensure efficient yet safe combustion, figure 6.9. The steam pressure is the primary measured variable for combustion rate. If the heat produced in the furnace decreases, then the rate of steam production is less than the rate of steam consumption and the supply pressure falls. The reverse is true if the heat rate increases. Steam pressure measurement takes into account changes in both the flow rate and BTU content of the fuel.

The primary controlled variable is the Forced Draft (F.D.) fan speed. Steam production rate is increased by increasing the air flow

through the furnace. The combustion control system measures the ratio of the steam flow and air flow and attempts to control fuel flow rate to maintain the proper excess air quantity. When the increase in F.D. fan speed is fed forward to the fuel valve control loop, the control system opens the fuel control valve to admit more fuel. The exact valve setting is trimmed by the steam flow/air flow ratio measurement.

Water level in the steam production drum of the boiler is commonly controlled by a single-element feedwater control loop. The drum level is measured and compared to a fixed set-point, internal to the controller. The difference between the set-point and actual water level is used to modulate the feedwater control valve position. The actual mP control program to implement the combustion control system is not provided as it is a conventional PID control task.

Because of its ability to perform proportional control in addition to, and in parallel with, sequential logic functions, the hybrid mP/BD programmable controller is much more suitable for boiler control tasks than are conventional programmable controllers and requires a lower capital investment than traditional hardware.

INPUTS		OUTPUTS	
SIGNAL	DESCRIPTION	SIGNAL	DESCRIPTION
T_{PRG}	TIMER OUTPUT	T_{PRGI}	TIMER INPUT
T_{LO}	TIMER OUTPUT	T_{LOI}	TIMER INPUT
T_{PP}	TIMER OUTPUT	T_{PPI}	TIMER INPUT
T_{SD}	TIMER OUTPUT	T_{SDI}	TIMER INPUT
FF	FLIP-FLOP	FF_D	FLIP-FLOP
PB_{PRG}	START PURGE PUSHBUTTON	L_{PR}	PURGE READY LAMP
		L_{PIP}	PURGE IN PROGRESS LAMP
PB_{BNR}	START BURNER PUSHBUTTON	L_{PC}	PURGE COMPLETE LAMP
		L_{BR}	BURNER READY LAMP
PB_{STOP}	STOP PUSHBUTTON	L_{TA}	TRIP ALARM LAMP
		L_{STOP}	STOP LAMP
$AFSV_{OPEN}$	SHUTOFF VALVE LIMIT SWITCHES	$AFSV_C$	CLOSE AFSV
		$AFSV_O$	OPEN AFSV
F_{DCTR}	FLAME DETECTOR	FAN_C	CLOSE FORCED DRAFT FAN
PRG_{AF}	MINIMUM PURGE AIRFLOW LIMIT SWITCH	IGN	ENERGIZE IGNITORS
DFT_{MIN}	FURNACE DRAFT		
DFT_{MAX}	LIMIT SWITCHES		
$DLVL_{MIN}$	BOILER DRUM		
$DLVL_{MAX}$	WATER LEVEL LIMIT SWITCHES		
AF_{MIN}	FURNACE AIRFLOW MINIMUM LIMIT SWITCH		
FP_{MIN}	FUEL SUPPLY		
FP_{MAX}	PRESSURE LIMIT SWITCHES		

table 6.6 BD processor I/O for the burner automation controller.

OUTPUTS		INPUTS	
SIGNAL	OUTPUT BIT	SIGNAL	INPUT CHANNEL
(BANK 0)			
NOT USED	OV<11:5>	T_{PRG}	36
T_{PPI}	OV<4>	T_{LO}	35
T_{SDI}	OV<3>	T_{PP}	34
T_{LOI}	OV<2>	T_{SD}	33
T_{PRGI}	OV<1>	FF	32
FF_D	OV<0>		
		PB_{PRG}	18
(BANK 1)		PB_{BNR}	17
NOT USED	OV<11:6>	PB_{STOP}	16
L_{TA}	OV<5>		
L_{BR}	OV<4>	$AFSV_{CLSD}$	10
L_{PC}	OV<3>	$AFSV_{OPEN}$	9
L_{PIP}	OV<2>	F_{DCTR}	8
L_{PR}	OV<1>	PRG_{AF}	7
L_{STOP}	OV<0>	DFT_{MIN}	6
		DFT_{MAX}	5
(BANK 2)		$DLVL_{MIN}$	4
IGN	OV<11>	$DLVL_{MAX}$	3
$AFSV_O$	OV<10>	AF_{MIN}	2
FAN_C	OV<9>	FP_{MIN}	1
$AFSV_C$	OV<8>	FP_{MAX}	0
NOT USED	OV<7:0>		

table 6.7 BD processor I/O terminal assignments for the burner automation controller.

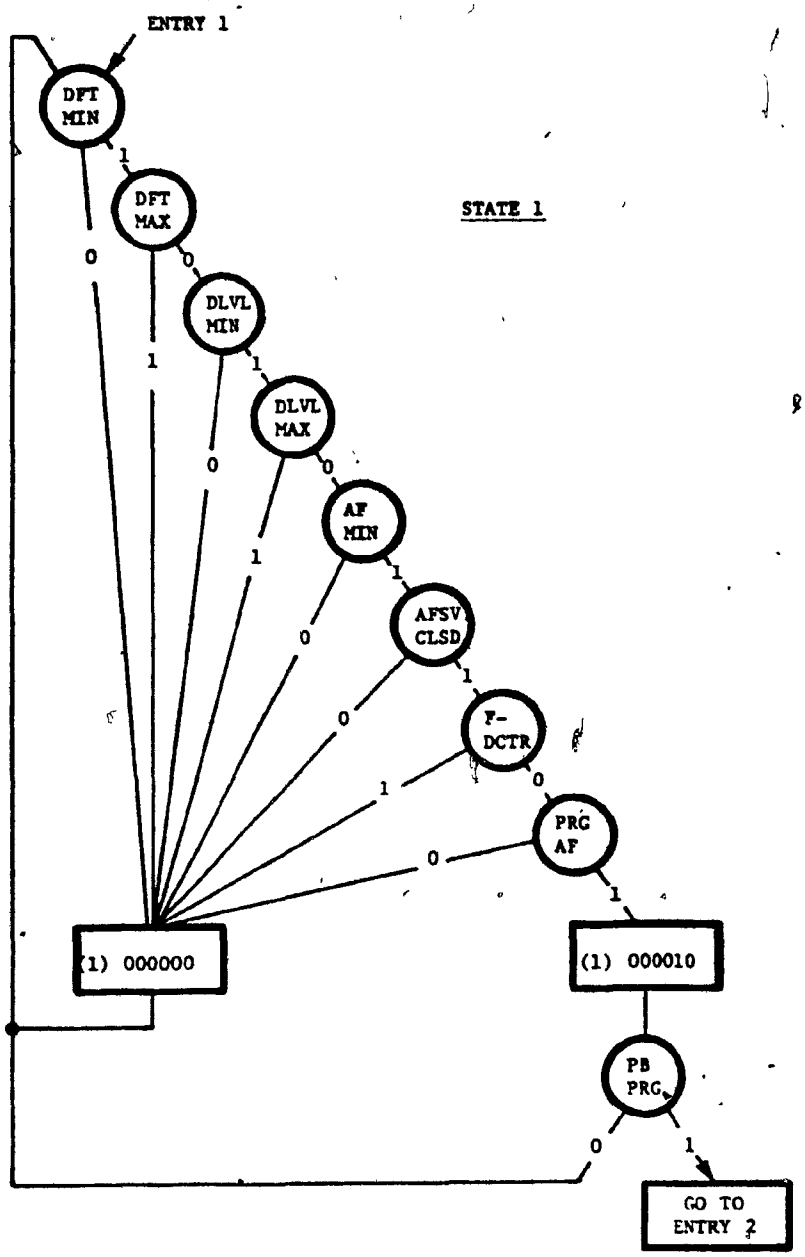


figure 6.6 BD diagrams for each burner controller state.

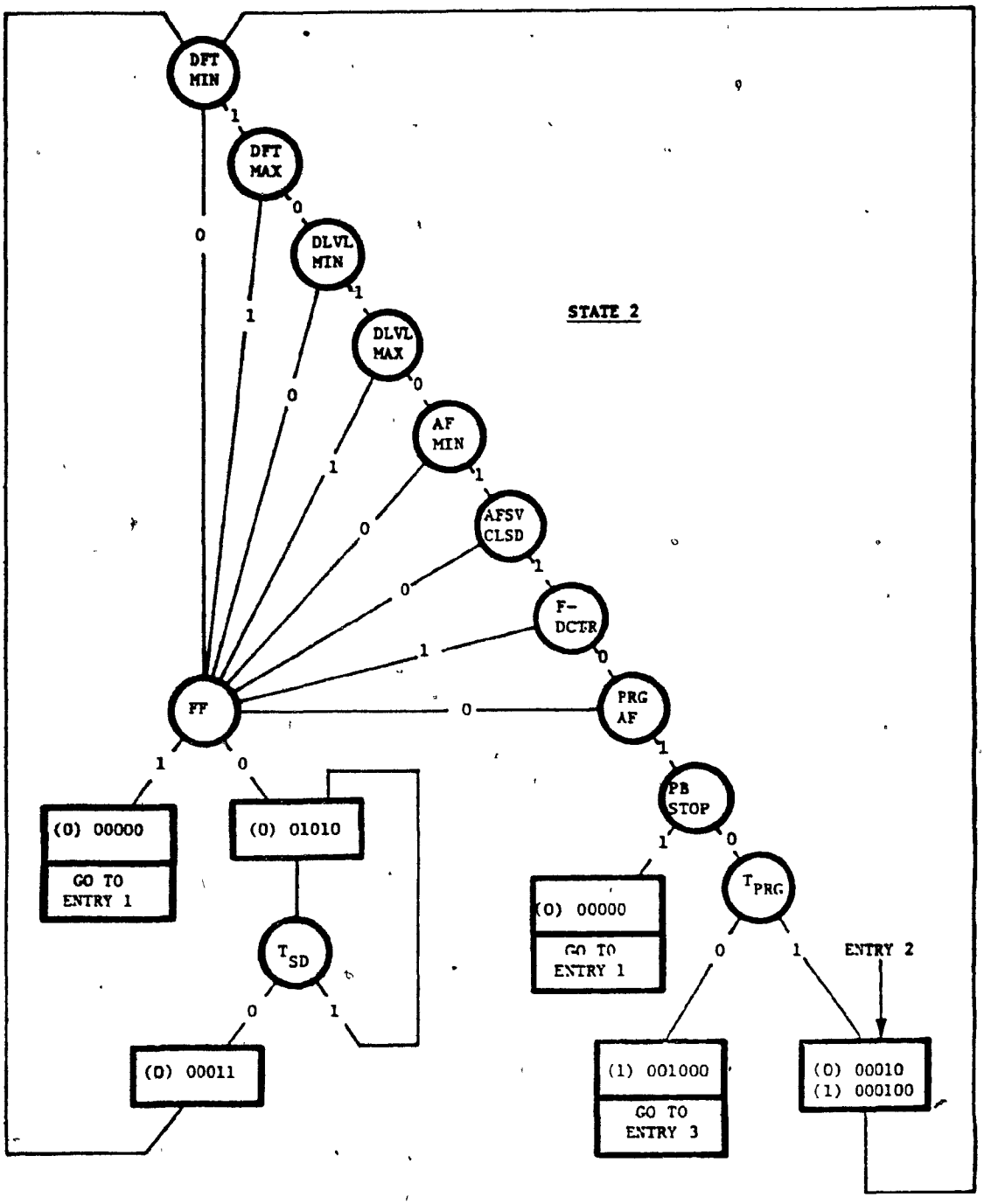


figure 6.6 (cont'd)

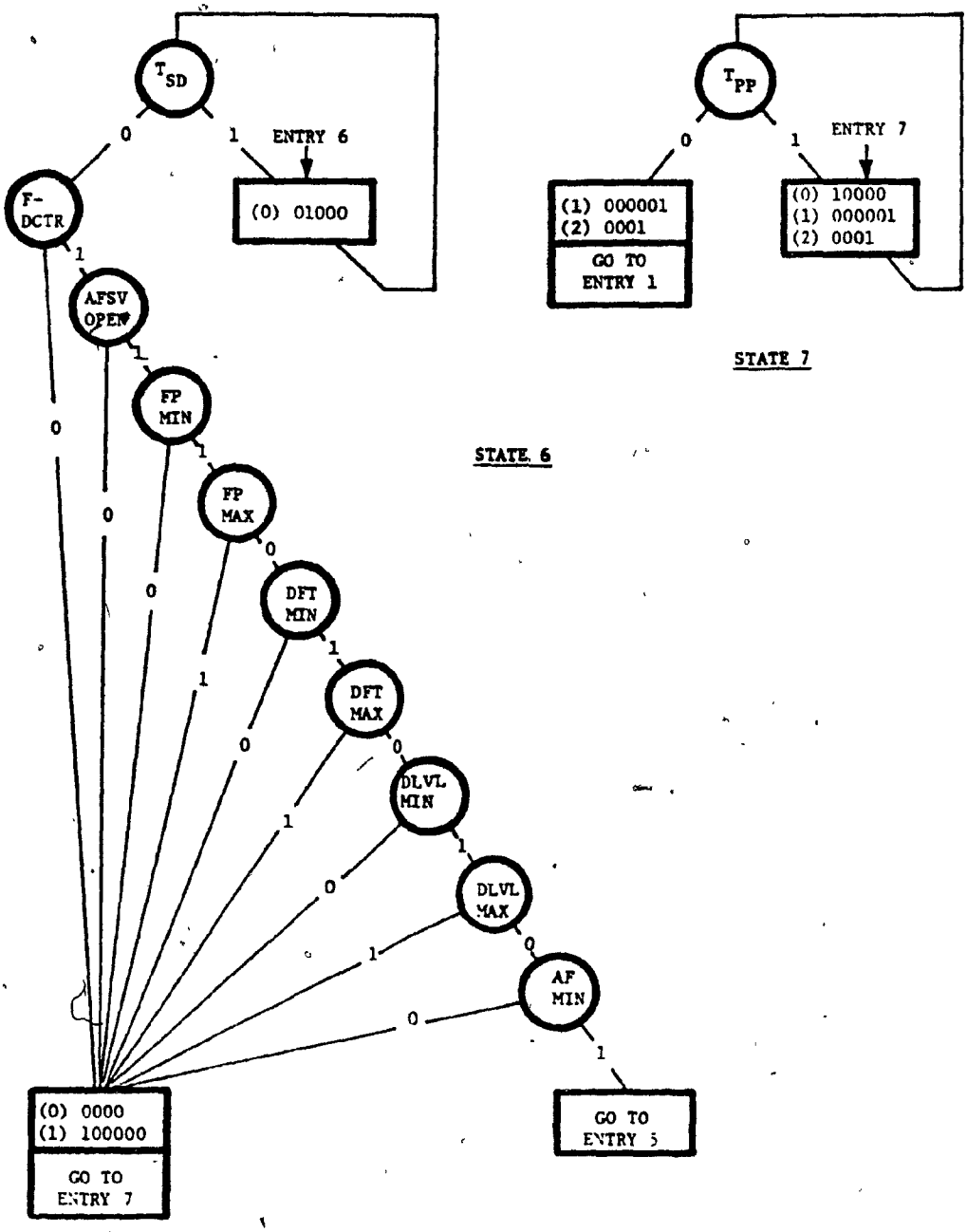


figure 6.6 (cont'd)

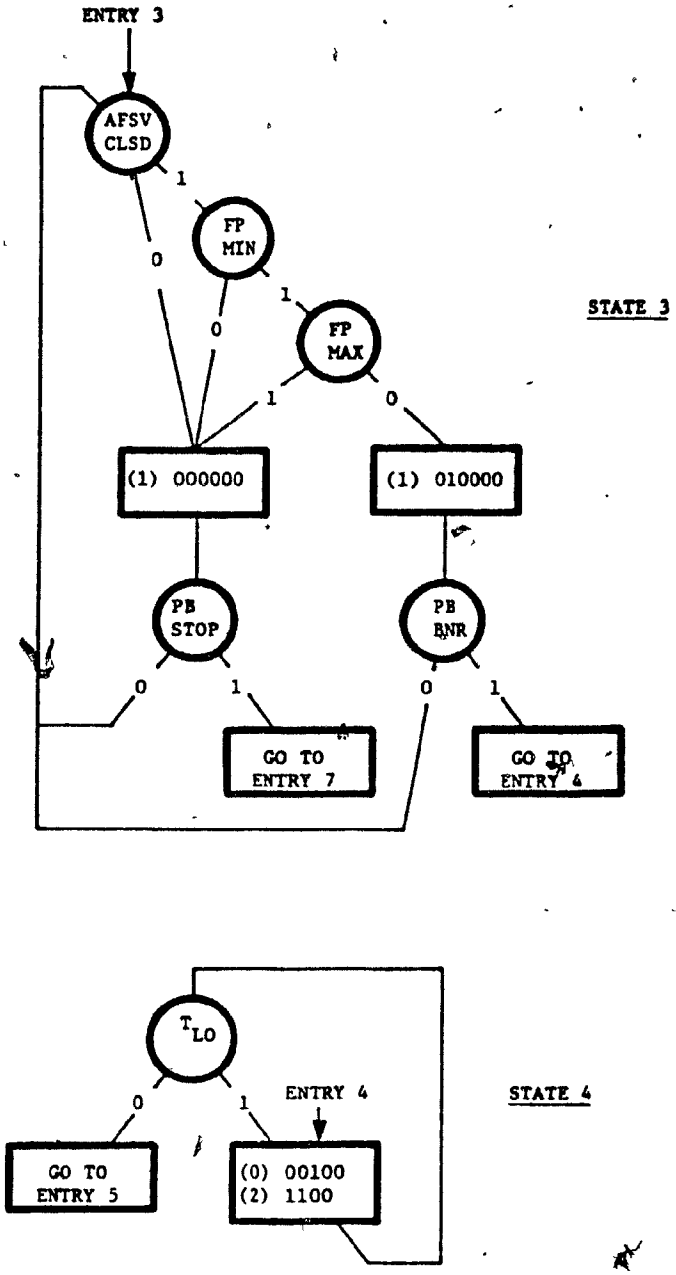


figure 6.6 (cont'd)

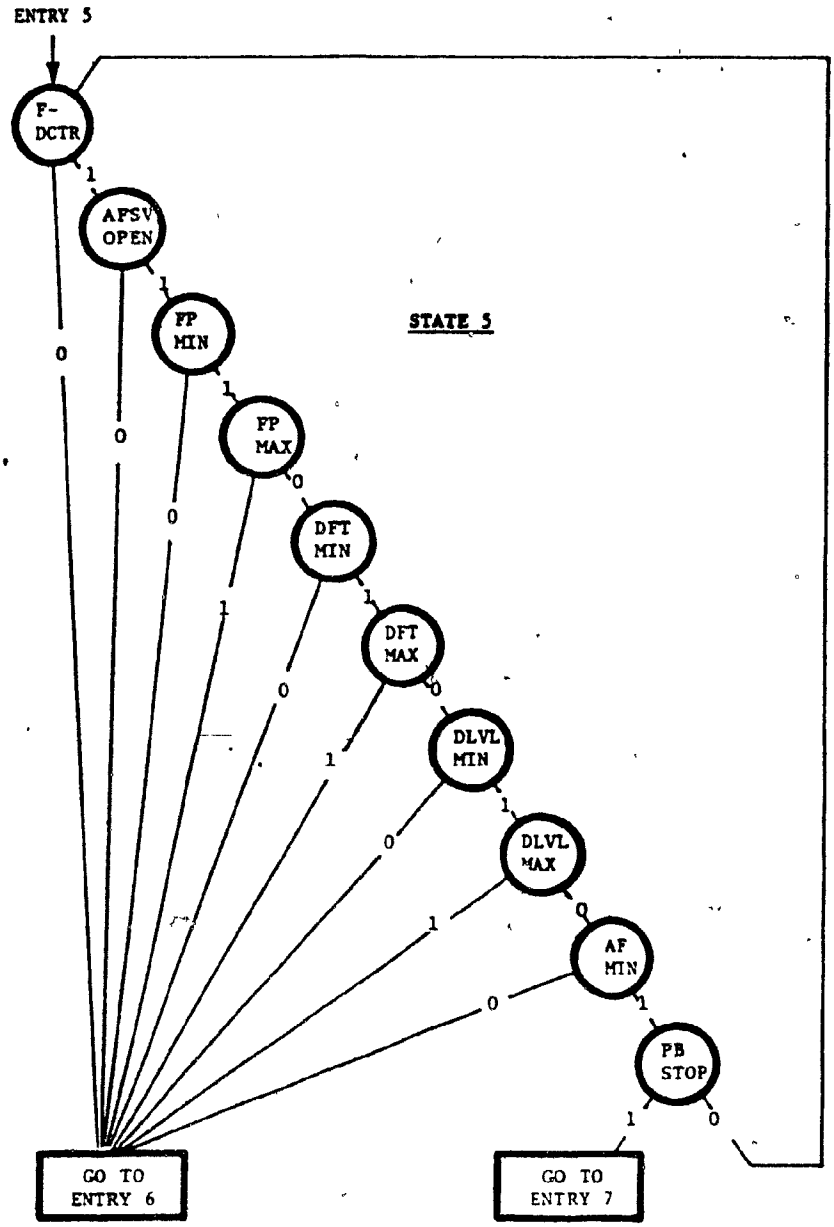


figure 6.6 (cont'd)

COMMENTS	PC_{10}	$CODE_{16}$	DESCRIPTION
STATE 0, ENTRY 1	0	E101	BANK SWITCH TO 1
	1	180C	INPUT DFT_{MIN}, JUMP TO 12 IF 0
	2	540C	INPUT DFT_{MAX}, JUMP TO 12 IF 1
	3	100C	INPUT $DLVL_{MIN}$, JUMP TO 12 IF 0
	4	4C0C	INPUT $DLVL_{MAX}$, JUMP TO 12 IF 1
	5	080C	INPUT AF_{MIN}, JUMP TO 12 IF 0
	6	280C	INPUT $AFSV_{CLSD}$, JUMP TO 12 IF 0
	7	600C	INPUT F_{DCTR}, JUMP TO 12 IF 1
	8	1C0C	INPUT PRG_{AF}, JUMP TO 12 IF 0
	9	8002	OUTPUT 000000000010
	10	491B	INPUT PB_{PRG}, JUMP TO 27 IF 1
	11	F001	JUMP TO 1
	12	8000	OUTPUT 000000000000
	13	F001	JUMP TO 1
STATE 2	14	E00F	BANK SWITCH TO 0
	15	1821	INPUT DFT_{MIN}, JUMP TO 33 IF 0
	16	5421	INPUT DFT_{MAX}, JUMP TO 33 IF 1
	17	1021	INPUT $DLVL_{MIN}$, JUMP TO 33 IF 0
	18	4C21	INPUT $DLVL_{MAX}$, JUMP TO 33 IF 1
	19	0821	INPUT AF_{MIN}, JUMP TO 33 IF 0
	20	2821	INPUT $AFSV_{CLSD}$, JUMP TO 33 IF 0
	21	6021	INPUT F_{DCTR}, JUMP TO 33 IF 1
	22	1C21	INPUT PRG_{AF}, JUMP TO 33 IF 0
	23	011A	INPUT PB_{STOP}, JUMP TO 26 IF 0
	24	8000	OUTPUT 000000000000
	25	F000	JUMP TO 0
	26	121F	INPUT T_{PRG}, JUMP TO 31 IF 0
ENTRY 2	27	8002	OUTPUT 000000000010
	28	E11D	BANK SWITCH TO 1
	29	8004	OUTPUT 000000000100
	30	F00E	JUMP TO 14
	31	8008	OUTPUT 000000001000
	32	F028	JUMP TO 40
	33	0224	INPUT FF, JUMP TO 36 IF 0
	34	8000	OUTPUT 000000000000
	35	F000	JUMP TO 0
	36	800A	OUTPUT 000000001010
	37	460E	INPUT T_{SD}, JUMP TO 14 IF 1
	38	8003	OUTPUT 000000000011
	39	F00E	JUMP TO 14
STATE 3, ENTRY 3	40	E129	BANK SWITCH TO 1
	41	282F	INPUT $AFSV_{CLSD}$, JUMP TO 47 IF 0
	42	042F	INPUT FP_{MIN}, JUMP TO 47 IF 0
	43	402F	INPUT FP_{MAX}, JUMP TO 47 IF 1
	44	8010	OUTPUT 000000010000
	45	0328	INPUT PB_{BNR}, JUMP TO 40 IF 0

figure 6.7 BD program for the burner automation controller.

COMMENTS	PC_{10}	$CODE_{16}$	DESCRIPTION
	46	F033	JUMP TO 51
	47	8000	OUTPUT 000000000000
	48	0128	INPUT PB_{STOP}, JUMP TO 40 IF 0
	49	F054	JUMP TO 84
STATE 4	50	0E37	INPUT T_{LO}, JUMP TO 55 IF 0
ENTRY 4	51	E034	BANK SWITCH 0
	52	8004	OUTPUT 000000000100
	53	E236	BANK SWITCH 2
	54	CC32	OUTPUT 1100, JUMP TO 50
STATE 5, ENTRY 5	55	2043	INPUT F_{DCTR}, JUMP TO 67 IF 0
	56	2443	INPUT $AFSV_{OPEN}$, JUMP TO 67 IF 0
	57	0443	INPUT FP_{MIN}, JUMP TO 67 IF 0
	58	4043	INPUT FP_{MAX}, JUMP TO 67 IF 1
	59	1843	INPUT DFT_{MIN}, JUMP TO 67 IF 0
	60	5443	INPUT DFT_{MAX}, JUMP TO 67 IF 1
	61	1043	INPUT $DLVL_{MIN}$, JUMP TO 67 IF 0
	62	4C43	INPUT $DLVL_{MAX}$, JUMP TO 67 IF 1
	63	0843	INPUT AF_{MIN}, JUMP TO 67 IF 0
	64	0137	INPUT PB_{STOP}, JUMP TO 55 IF 0
	65	F054	JUMP TO 84
STATE 6	66	0646	INPUT T_{SD}, JUMP TO 70 IF 0
ENTRY 6	67	E044	BANK SWITCH TO 0
	68	8008	OUTPUT 000000001000
	69	F042	JUMP TO 66
	70	204F	INPUT F_{DCTR}, JUMP TO 79 IF 0
	71	244F	INPUT $AFSV_{OPEN}$, JUMP TO 79 IF 0
	72	044F	INPUT FP_{MIN}, JUMP TO 79 IF 0
	73	404F	INPUT FP_{MAX}, JUMP TO 79 IF 1
	74	184F	INPUT DFT_{MIN}, JUMP TO 79 IF 0
	75	544F	INPUT DFT_{MAX}, JUMP TO 79 IF 1
	76	104E	INPUT $DLVL_{MIN}$, JUMP TO 79 IF 0
	77	4C4E	INPUT $DLVL_{MAX}$, JUMP TO 79 IF 1
	78	4837	INPUT AF_{MIN}, JUMP TO 55 IF 1
	79	8000	OUTPUT 000000000000
	80	E151	BANK SWITCH TO 1
	81	8020	OUTPUT 000000100000
	82	F054	JUMP TO 84
STATE 7	83	0A5A	INPUT T_{PP}, JUMP TO 90 IF 0
ENTRY 7	84	E055	BANK SWITCH TO 0
	85	8010	OUTPUT 000000010000

figure 6.7 (cont'd)

COMMENTS	PC_{10}	$CODE_{16}$	DESCRIPTION
	86	E157	BANK SWITCH TO 1
	87	8001	OUTPUT 000000000001
	88	E259	BANK SWITCH TO 2
	90	E15B	BANK SWITCH 1
	91	8001	OUTPUT 000000000001
	92	E25D	BANK SWITCH 2
	93	C300	OUTPUT 0011, JUMP TO 0

figure 6.7 (cont'd)

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
START UP	00	XNOR	RR	
	01	IEN	RR	
STATE 0	02	LDC	B0	IF (STATE 0) THEN
	03	ANDC	B1	FLAG STATE 1
	04	ANDC	B2	
	05	OEN	RR	
	06	STO	B0	
STATE 1	07	LD	B0	IF (STATE 1·BLR·BNR) THEN
	08	ANDC	B1	
	09	ANDC	B2	
	10	AND	DFT_{MIN}	
	11	ANDC	DFT_{MAX}	
	12	AND	$DLVL_{MIN}$	
	13	ANDC	$DLVL_{MAX}$	
	14	AND	AF_{MIN}	
	15	AND	$AFSV_{CLSD}$	
	16	ANDC	F_{DCTR}	
	17	AND	PRG_{AF}	
	18	OEN	RR	
	19	STO	L_{PR}	LIGHT PURGE-READY LAMP
	20	STOC	L_{STOP}	RESET STOP LAMP
	21	AND	PB_{PRG}	IF (PB_{PRG}) THEN
	22	OEN	RR	
	23	STOC	L_{PR}	RESET PURGE-READY LAMP
	24	STOC	B0	FLAG STATE 2
	25	STO	B1	
STATE 2	26	LDC	B0	IF (STATE 2·BLR·PRG·PB_{STOP})
	27	AND	B1	THEN
	28	ANDC	B2	
	29	AND	DFT_{MIN}	
	30	ANDC	DFT_{MAX}	
	31	AND	$DLVL_{MIN}$	
	32	ANDC	$DLVL_{MAX}$	
	33	AND	AF_{MIN}	
	34	AND	$AFSV_{CLSD}$	
	35	AND	PRG_{AF}	
	36	STOC	TEMP	
	37	AND	PB_{STOP}	
	38	OEN	RR	
	39	STOC	T_{PRGI}	RESET PURGE CYCLE TIMER
	40	STOC	T_{SDI}	RESET SAFETY DELAY TIMER
	41	STOC	L_{PIP}	RESET PURGE-IN-PROGRESS LAMP
	42	STOC	FF_D	RESET FLIP-FLOP
	43	STO	B0	FLAG STATE 1
	44	STOC	B1	

figure 6.8 MC14500 program for the burner automation controller.

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
	45	LDC	TEMP	IF (STATE 2·BLR·PRG·$\overline{PB_{STOP}}$)
	46	ANDC	PB_{STOP}	THEN
	47	OEN	RR	
	48	STO	T_{PRGI}	START PURGE CYCLE TIMER
	49	STO	L_{PIP}	LIGHT PURGE-IN-PROGRESS LAMP
	50	ANDC	T_{PRG}	IF (TIMER EXPIRED) THEN
	51	OEN	RR	
	52	STOC	T_{PRGI}	RESET PURGE CYCLE TIMER
	53	STOC	T_{SDI}	RESET SAFETY DELAY TIMER
	54	STOC	L_{PIP}	RESET PURGE-IN-PROGRESS LAMP
	55	STOC	FF_D	RESET FLIP-FLOP
	56	STO	L_{PC}	LIGHT PURGE COMPLETE LAMP
	57	STO	B0	FLAG STATE 3
	58	LDC	B0	IF [STATE 2·FF·($\overline{BLR}$ + $\overline{BNR}$)]
	59	AND	B1	THEN
	60	ANDC	B2	
	61	AND	TEMP	
	62	STO	TEMP1	
	63	AND	FF	
	64	OEN	RR	
	65	STOC	T_{PRGI}	RESET PURGE CYCLE TIMER
	66	STOC	T_{SDI}	RESET SAFETY DELAY TIMER
	67	STOC	L_{PIP}	RESET PURGE-IN-PROGRESS LAMP
	68	STOC	FF_D	RESET FLIP-FLOP
	69	STO	B0	FLAG STATE 1
	70	STO	B1	
	71	LD	TEMP1	IF [STATE 2·FF·($\overline{BLR}$ + $\overline{BNR}$)]
	72	ANDC	FF	THEN
	73	OEN	RR	
	74	STO	T_{SDI}	START SAFETY DELAY TIMER
	75	ANDC	T_{SD}	IF (SAFETY DELAY TIMER
	76	OEN	RR	EXPIRED) THEN
	77	STO	FF	SET FLIP-FLOP
	78	STOC	T_{SDI}	RESET SAFETY DELAY TIMER
STATE 3	79	LD	B0	IF (STATE 3·LO) THEN
	80	AND	B1	
	81	ANDC	B2	
	82	AND	$AFSV_{CLSD}$	
	83	AND	FP_{MIN}	
	84	ANDC	FP_{MAX}	
	85	STOC	TEMP	

figure 6.8 (cont'd)

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
	86	OEN	RR	
	87	STO	L_{BR}	LIGHT BURNER READY LAMP
	88	AND	PB_{BNR}	IF (PB_{BNR}) THEN
	90	STOC	L_{BR}	RESET BURNER READY LAMP
	91	STOC	L_{PC}	RESET PURGE COMPLETE LAMP
	92	STOC	B0	FLAG STATE 4
	93	STOC	B1	
	94	STO	B2	
	95	LD	B0	IF (STATE 3·$\overline{LO}$)
	96	AND	B1	
	97	ANDC	B2	
	98	AND	TEMP	
	99	OEN	RR	
	100	STOC	L_{BR}	RESET BURNER READY LAMP
	101	AND	PB_{STOP}	IF (PB_{STOP}) THEN
	102	OEN	RR	
	103	STOC	L_{PC}	RESET PURGE COMPLETE LAMP
	104	STO	B2	FLAG STATE 7
	105	LDC	B0	IF (STATE 4) THEN
	106	ANDC	B1	
	107	AND	B2	
	108	OEN	RR	
	109	STO	T_{LOI}	START LIGHT-OFF TIMER
	110	STO	$AFSV_0$	OPEN FUEL SHUTOFF VALVE
	111	STO	IGN	START IGNITOR
	112	ANDC	T_{LO}	IF (TIMER EXPIRED) THEN
	113	OEN	RR	
	114	STOC	T_{LOI}	RESET LIGHT-OFF TIMER
	115	STOC	IGN	RESET IGNITOR
	116	STO	B0	FLAG STATE 5
STATE 5	117	LD	B0	IF (STATE 5·BLR·BNR)
	118	ANDC	B1	THEN
	119	AND	B2	
	120	AND	F_{DCTR}	
	121	AND	$AFSV_{OPEN}$	
	122	AND	FP_{MIN}	
	123	ANDC	FP_{MAX}	
	124	AND	DFT_{MIN}	
	125	ANDC	DFT_{MAX}	
	126	AND	$DLVL_{MIN}$	
	127	ANDC	$DLVL_{MAX}$	
	128	AND	AF_{MIN}	

figure 6.8 (cont'd)

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
	129	STOC	TEMP	
	130	AND	PB_{STOP}	IF (PB_{STOP}) THEN
	131	OEN	RR	
	132	STO	B1	FLAG STATE 7
	133	LD	B0	IF [STATE 5·($\overline{BLR}$ + $\overline{BNR}$)]
	134	ANDC	B1	THEN
	135	AND	B2	
	136	AND	TEMP	
	137	OEN	RR	
	138	STO	B1	FLAG STATE 6
	139	STOC	B2	
STATE 6	140	LDC	B0	IF (STATE 6) THEN
	141	AND	B1	
	142	AND	B2	
	143	OEN	RR	
	144	STO	T_{SDI}	START SAFETY DELAY TIMER
	145	ANDC	T_{SD}	IF (TIMER EXPIRED) THEN
	146	OEN	RR	
	147	STOC	T_{SDI}	RESET SAFETY DELAY TIMER
	148	STO	TEMP	
	149	AND	F_{DCTR}	IF (STATE 6·$\overline{TMZ}$·BLR·BNR)
	150	AND	AFSV	THEN
	151	AND	FP_{MIN}	
	152	ANDC	FP_{MAX}	
	153	AND	DFT_{MIN}	
	154	ANDC	DFT_{MAX}	
	155	AND	$DLVL_{MIN}$	
	156	ANDC	$DLVL_{MAX}$	
	157	AND	AF_{MIN}	
	158	STOC	TEMP1	
	159	OEN	RR	
	160	STO	B0	FLAG STATE 5
	161	STOC	B1	
	162	LD	TEMP1	
	163	AND	TEMP	IF (STATE 6·$\overline{TMZ}$·($\overline{BLR}$ + $\overline{BNR}$))
	164	OEN	RR	THEN
	165	STO	L_{TA}	LIGHT TRIP ALARM LAMP
	166	STO	B0	FLAG STATE 7
STATE 7	167	LD	B0	IF (STATE 7) THEN
	168	AND	B1	
	169	AND	B2	
	170	OEN	RR	
	171	STO	T_{PPI}	START POST-PURGE TIMER

figure 6.8 (cont'd)

COMMENTS	PC_{10}	INSTRUCTION		DESCRIPTION
	172	STO	L_{STOP}	LIGHT STOP LAMP
	173	STO	$AFSV_C$	CLOSE FUEL VALVE
	174	ANDC	T_{PP}	IF (TIMER EXPIRED)
	175	OEN	RR	THEN
	176	STOC	T_{PPI}	RESET POST-PURGE TIMER
	177	STOC	B1	FLAG STATE 1
	178	STOC	B2	
	179	NOPF		SET FLAG TO RESET PC

figure 6.8 (cont'd)

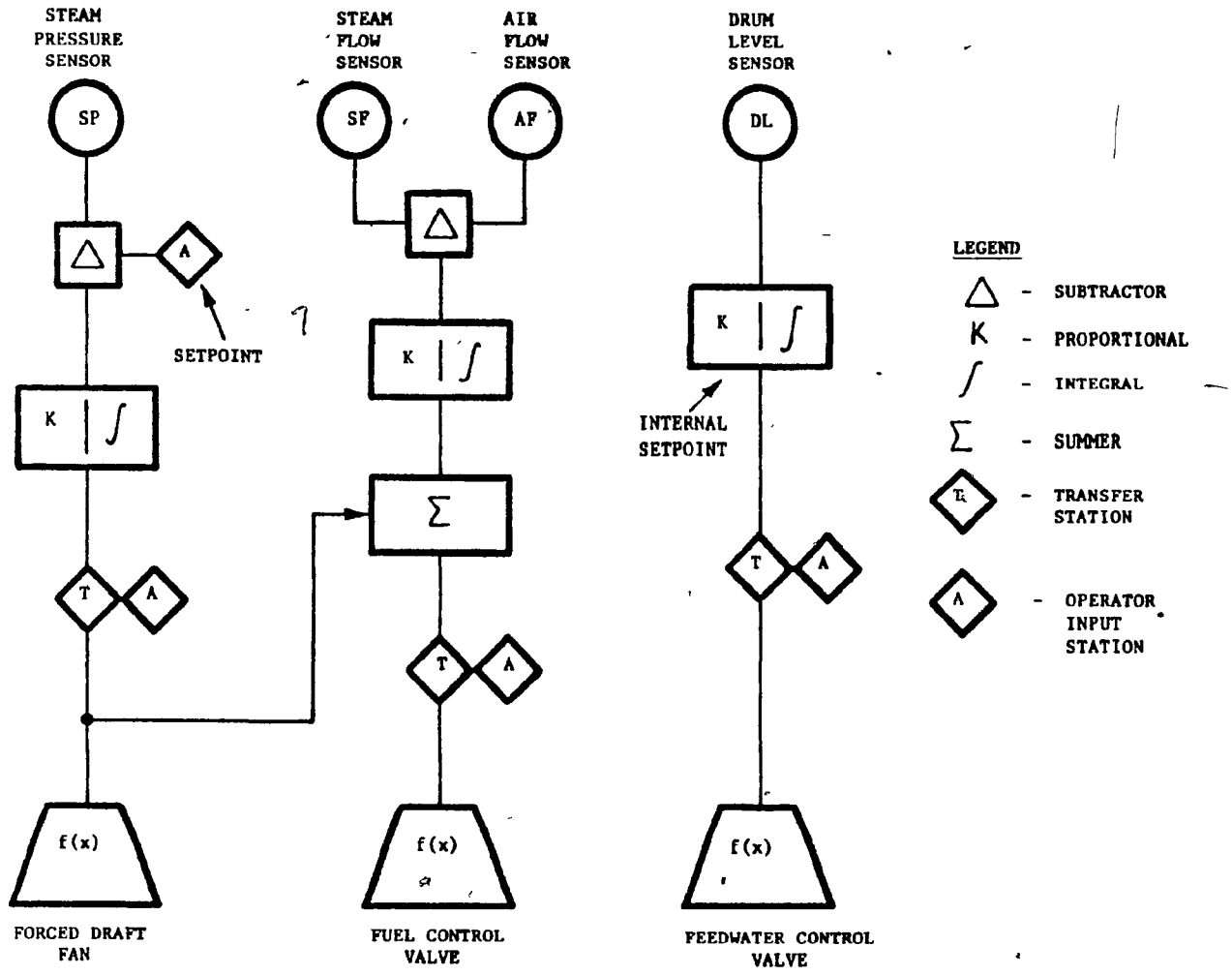


figure 6.9 Combustion control system schematic diagram.

CHAPTER 7

CONCLUSIONS

7.1 Summary of Results

7.1.1 mP/BD Hybrid Controller. It has been shown that a hybrid mP/BD programmable controller architecture offers substantial advantages over conventional programmable automata in terms of flexibility, efficiency and economy. In this regard, it was demonstrated that:

1) Binary decision machines support the evaluation of multi-valued switching logic, the parallel execution of serial switching functions and the realization of time-dependant sequential functions. In contrast, PLA-based PLCs are limited to the evaluation of binary-valued combinatorial logic while multi-bit mP controller architectures are constrained to emulate serial input machines.

2) BD automata always evaluate combinatorial functions in a number of steps equal to or less than the number of input variables, where Boolean-based machines, such as the ICU, consume exponential order processing time. This relative speed advantage of the BD machine is advantageous for fast time-constant process applications, permits the use of more complicated process models and/or allows the control of a number of concurrent tasks.

3) The BD processor's Logic Unit consists of only five logic gates. Hence signal propagation through the LU is extremely fast. This

suggests that planned VLSI implementations of the BD architecture will operate at high clock rates, limited only by instruction memory access and input/output buffer cycle times, and will be inexpensive to design and fabricate due to the relatively low number of logic elements.

4) Previous BD architectures were restricted to simple control tasks by their limited memory resource and the characteristic exponential growth of BD program size with the number of input variables. The instruction set of the existing Holck BD prototype has been enhanced to add more I/O capacity and to introduce an interface bus for communication with a host mP. As a result the controller may employ program optimization and/or partitioning strategies to increase the processing capabilities of the BD machine.

The technical feasibility of the hybrid programmable controller was validated by the construction and testing of a laboratory prototype system which includes a microcomputer, a BD machine and the BD09 process control operating system. Two programmable controller applications are presented which compare the performance of the mP/BD hybrid with a conventional MC14500 ICU. The mP/BD implementations are consistently shorter and faster than the ICU examples. A 72% reduction in program size, from 152 MC14500 instructions to 42 BD instructions, is achieved in the traffic controller task while a 48% reduction, from 180 MC14500 instructions to 94 BD instructions, is obtained in the boiler control application. In the latter example only the sequential automation programs are compared since the ICU's single-bit architecture is not intended for, nor capable of, handling the combustion control modulation tasks. These may be handled in the mP half of the hybrid controller.

7.1.2 BD Program Compiler. The design and operation of a practical optimizing compiler for combinatorial logic functions is described which produces near-optimally reduced BD programs. It is contended that in industrial applications it is preferable to obtain a size reduction within 10%-15% of the optimum in a short time rather than exhaustively searching among up to $\prod_{i=0}^{n-1}(n-i)^{2^i}$ different programs for the global minimum. The worst-case time complexity of the heuristic PMA optimization algorithm is $O(2^{2n})$ and the space complexity is less than $O(2^{n+3})$. A formal expected-case time complexity analysis was not attempted.

7.1.3 Binary Decision Analysis. Another objective of this research was to gain some physical understanding of the binary decision process and its relationship to Boolean logic.

It is known that the 2^n paths of a complete BD program have a one-to-one correspondance with the series expansion terms of a Boolean expression which realizes the same function. From this it is concluded that combinatorial binary decision logic is not morphologically different from Boolean logic. As in the case of Shannon's Expansion Theorem, the BD method factors switching function implicants so that following each evaluation of a literal, half of the remaining terms are determined to be logically FALSE and, hence, need not be further evaluated. This happens at most n times for a n-input function. However, the BD method automatically branches away from FALSE terms at the earliest possible moment whereas the Boolean method continues to

evaluate literals associated with the FALSE implicants. This is the fundamental distinction and the hallmark of the BD method.

The BD branching process is also shown to be completely analogous to the state-transition description of sequential switching logic.

7.2 Recommendations for Future Work

It is suggested that the present research continue in the following directions:

1) System software development. It is recommended that the system software be divided into two separate packages; an EPROM-based monitor composed of BDBUG and elements of V-BUG to provide low level device drivers and the boot-strap loader, and the disk-resident BD09 real-time process control operating system. Further development of BD09 to implement the full scope of program management features is also required.

2) Program compilers. The optimization and run-time characteristics of BDC-4 should be investigated through a series of benchmarking experiments. This would permit an objective evaluation of this compiler's performance in comparison with other optimization algorithms. In addition, the remaining planned data input formats, e.g., relay ladder diagram and logic gate diagram formats, should be implemented in BDC-4.

On a broader horizon, the development of an optimizing compiler for sequential logic BD programs is indicated. This may involve the

adaptation or creation of a high level programming language which could adequately describe both sequential and combinatorial BD logic functions. Finally, a compiler for mP control algorithms is also required.

3) System hardware development. Specific enhancements to current BD machine designs should be studied with the objective of improving the processor's execution speed and efficiency. It is conservatively estimated that a continuous instruction throughput of 10 MIPS (million instructions per second) may be realized with a processor constructed with fast Schottky F-series TTL technology. In practice, the limiting speed parameter is memory access time. Other suggested BD hardware enhancements include an expanded instruction set, an effective increase in the directly addressable program memory space, and a rationalization of the I/O section design.

The design and construction of a VLSI BD processor represents another area for future work. Two versions of this device are envisioned; a single-chip device for use with "smart" BD-based instruments, and a general purpose Binary Decision Unit (BDU) for use in hybrid controller applications.

Finally, the development of a commercial mP/BD programmable controller system is indicated. This would likely require the participation of an industrial PLC manufacturer to provide marketing services and to ensure that the system conforms to accepted industrial standards. A single-board BD evaluation kit has been constructed which may be offered for limited commercial distribution. Its function would be to familiarize engineers and scientists with the principles of BD

processing and could be used to develop new instrumentation and control applications.

4) Applications. It is suggested that BD methods have important applications in the control of low-information bandwidth, high speed systems, e.g., robotics and other dynamically unstable mechanical systems, in which the need for highly precise control actions is secondary to the requirement that they be available in real time. A distributed network of BD controllers, for example, stationed at the joints of a robot manipulator device and each programmed with the control actions for its particular joint, could be used to operate a manipulator in response to commands from a central mP or directly from stimuli in the manipulator's environment. This represents a significant departure from traditional robot control systems which rely on "brute-force" methods to solve large systems of governing equations and which require correspondingly large computers to produce control actions in near real time. The development of BD-based robotic control systems is a very important area of future research.

REFERENCES

Aho74 Aho, A. V., Hopcroft, J.E., and Ullman, J.D., "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Mass., 1974.

Aker78 Akers, S.B., "Binary Decision Diagrams", IEEE Trans. Computers, Vol.C-27, No.6, June 1978, pp. 509-516.

Artw80 Artwick, B.A., "Microcomputer Interfacing", Prentice-Hall, Englewood Cliffs, NJ, 1980.

B&W72 "STEAM/ its generation and use", The Babcock and Wilcox Co., New York, 1972.

Bell71 Bell, C.G., and Newell, A., "Computer Structures - Readings and Examples", McGraw-Hill, New York, 1971.

Bout76 Boute, R.T., "The Binary-Decision Machine as Programmable Controller", Euromicro Newsletter, Vol.1, No.2, 1976, pp. 16-22.

Cern79 Cerny, E., Mange, D., and Sanchez, E., "Synthesis of Minimal Binary Decision Trees", IEEE Trans. Computers, Vol.C-28, No.7, July 1979, pp. 472-482.

Drie82 van-Driel, C.J., "Binary Decision Compiler Algorithms", Memo 82-2, DATAC Computer Laboratory, Montreal, Quebec, September 1982.

Flyn84 Flynn, W.R., "Range of PC Offerings Continues to Grow", Control Engineering, Vol.31, No.1, January 1984, pp. 81-86.

Greg77 Gregory, V., and Dellande, B., "MC14500B Industrial Control Unit Handbook",Motorola Semiconductor Products, Inc., Austin, Tex., 1977.

Huds82 Hudson, R., Zsombor-Murray, P.J., and Vroomen, L.J., "Operating System for the BD/mP Programmable Controller", Int'l Journal of Mini and Microcomputers, Vol. 4, No. 3, 1982, pp. 58-62.

Huds84 Hudson, R., Vroomen, L.J., and Karasick, M., "Binary Decision Program Optimization Algorithms", Proc. of Twenty-Fourth Int'l Symp. on Mini and Microcomputers, Bari, Italy, June 1984, pp. 51-55.

Hyaf76 Hyafil, L. and Rivest, R.L., "Constructing Optimal Binary Decision Trees is NP-Complete", Information Processing Letters, Vol.5, No.1, May 1976, pp. 15-17.

Kara84 Karasick, M., "Binary Decision Tree Reduction", Technical Report SOCS-84.16, McGill University, School of Computer Science, October 1984.

Koha70 Kohavi, Z., "Switching and Finite Automata Theory", McGraw-Hill, New York, 1970.

Korn77 Korn, G.A., "Microprocessors and Small Digital Computer Systems for Engineers and Scientists", McGraw-Hill, New York, 1977.

Lee59 Lee, C.Y., "Representation of Switching Circuits by Binary Decision Programs", Bell Syst. Tech. J., Vol.38, July 1959, pp. 985-999.

Levi82a Levi, M.H., and Hudson, R.D., "Binary Decision Processor/Microprocessor Interface", Internal Memo, DATAC Computer Laboratory, Montreal, Quebec, April 1982.

Levi82b Levi, M.H., Vroomen, L.J., and Zsombor-Murray, P.J., "Modular Programmable Controllers Using an Intelligent Reflexive Interface", Proc. Twentieth ISMM Int'l Symp. on Mini and Microcomputers, July 1982, Cambridge, Mass., pp. 8-10.

Levi84 Levi, M.H., "Intelligent Reflexive Interfaces and Their Applications", M.Eng. Thesis, McGill University, Department of Mechanical Engineering, To be published.

Mang78 Mange, D., "Arbres de Decision Pour Systemes Logiques Cables au Programmes", Bulletin de l'Association Suisse des Electriciens, Vol.69, 1978, pp. 1238-1243.

McCl56 McCluskey, E. J., "Minimization of Boolean Functions", Bell Syst. Tech. J., Vol.35, No.5, Nov. 1956, pp. 1417-1444.

More80 Moret, B.M., Thomason, M.G., and Gonzales, R.C., "The Activity of a Variable and its Relation to Decision Trees", ACM Trans. Program Lang. Syst., Vol.2, No.4, October 1980, pp. 580-595.

More82 Moret, B.M., "Decision Trees and Diagrams", ACM Computing Surveys, Vol.14, No.4, December 1982, pp. 593-623.

Moto75 "M6800 Microprocessor Applications Manual", Motorola Semiconductor Products Inc., New York, 1975.

NFPA76 "Standards for Prevention of Furnace Explosions in Fuel Oil- and Gas-Fired Single Burner Boiler-Furnaces", NFPA National Fire Protection Association Standard 85, Boston, 1976.

Poll65 Pollack, S.L., "Conversion of Limited-Entry Decision Tables to Computer Programs", Comm. ACM, Vol.8, No.11, November 1965, pp. 677-682.

Shan38 Shannon, C.E., "A Symbolic Analysis of Relay and Switching Circuits", Trans. AIEE, Vol.57, 1938, pp. 713-723.

Shwa74 Shwayder, K., "Extending the Information Theory Approach to Converting Limited-Entry Decision Tables to Computer Programs", Comm. ACM, Vol.17, No.9, September 1974, pp. 532-537.

Siew74 Siewiorek, D.P., "Introducing ISP", Computer, Vol.7, No.12, Dec. 1974, pp. 39-41.

Stan80 Standish, T.A., "Data Structure Techniques", Addison-Wesley, Reading, Mass., 1980.

Taba81 Tabachnik, R.L., Zsombor-Murray, P.J., Vroomen, L.J., and Tho Le-Ngoc, "Sequence Controllers with Standard Hardware and Custom Firmware", IEEE Micro, Vol.1, No.2, May 1981, pp. 9-25.

Thay81 Thayse, A., "P-Functions: A New Tool for the Analysis and Synthesis of Binary Programs", IEEE Trans. Computers, Vol.C-30, No.2, February 1981, pp. 126-134.

Tho79 Tho Le-Ngoc, Zsombor-Murray, P.J., and Vroomen, L.J., "A Binary-Decision Approach to Industrial Programmable Controllers", Proc. Int'l Symp. Measurement and Control, Grenoble, June 1979, pp. 57-61.

Vroo81 Vroomen, L.C., and Vroomen, L.J., "V-bug 2.0", Internal Memo, DATAC Computer Laboratory, McGill University, Montreal, July 1981.

Wood77 Woodruff, E.B., and Lammers, H.B., "Steam Plant Operation", McGraw-Hill, New York, 1977.

Zsom79 Zsombor-Murray, P.J., Vroomen, L.J., Tho Le-Ngoc, and Holck, P.H., "A Binary-Decision-Based Programmable Controller", Proc. Second Int'l Symp. on Measurement and Control, Fort Lauderdale, Fla., December 1979, pp. 60-65.

APPENDIX I

A SHORT REVIEW OF BOOLEAN ALGEBRA

A1.1 Switching Functions

Assume the existence of a discrete function

$$F = f(X_1, \ldots, X_n) \tag{A1.1}$$

where the set of independent variables X_1 to X_n are binary valued variables. Let X_i be one member of this set. Thus for all X_i's:

$$X_i \in \{0,1\} \tag{A1.2}$$

A switching function is a discrete function defined by equations A1.1 and A1.2 in which F can take on a set of discrete values determined by the state of the independent variables and the rules of algebra used to combine them. The function is said to switch among its possible discrete values. If the rules of Boolean Algebra are used then a binary switching function is obtained.

A1.2 Operators

Three principal operations for the combination of variables are defined in Boolean Algebra.

1) Disjunction between two variables is denoted by the symbol '+'. It is defined such that the result of the operation is 1 if either or both of the variables have the value 1 and is 0 if both of the variables are 0. This operation is illustrated in table A1.1.

X_1	X_2	X_1+X_2
0	0	0
0	1	1
1	0	1
1	1	1

table A1.1 - Disjunction Operation

2) Conjunction between two variables is denoted by a '·'. It is defined such that the result of the operation is 1 only if both of the variables have the value 1 and is 0 if either or both of the variables are 0. This operation is illustrated in table A1.2.

X_1	X_2	$X_1 \cdot X_2$
0	0	0
0	1	0
1	0	0
1	1	1

table A1.2 - Conjunction Operation

3) Complementation is a uniary operation denoted by the symbol '¯'. It is defined such that the result of the operation is 1 if the variable has the value 0 and is 0 if the variable has the value 1. This operation is illustrated in table A1.3.

X_1	$\bar{X}_1$
0	1
1	0

table A1.3 - Complementation Operation

A1.3 Boolean Equations

A Boolean switching function is a binary function in which the value of the function F is either 0 or 1. It can be described by a mapping of the solution set of F and its variables or by a general equation expressed in terms of the set of independent variables and the principal operations. The Boolean equation can be derived from the mapping by segregating all the points in the domain which map to the value 1. These particular values are called the minterms of the function. The minterms are rewritten in terms of the names of the independent variables implied in the expression such that where a variable has the value 1 it is written in its asserted form and where it is 0 it is written in its complemented (or negated) form.

A Boolean equation is evaluated from left to right according to the order of precedence of the operators. Thus complementations are performed first followed by conjunctions and finally disjunctions. Variables that are grouped together with parentheses must be evaluated before non-bracketed terms.

A1.4 Algebraic Properties

In addition to the logical operations, or connectives described above, there exists a group of algebraic rules that define the manner in which switching functions are manipulated. These are summarized below.

1) The Distributive Law defines the way in which terms of a Boolean equation may be factored or expanded, equations A1.5a and A1.5b:

$$X \cdot Y + X \cdot Z = X \cdot (Y + Z) \qquad \text{(A1.5a)}$$

or,

$$X \cdot Y + Z \cdot Y = (X + Z) \cdot Y \qquad \text{(A1.5b)}$$

2) The Associative Law defines the way in which terms of a Boolean equation may be grouped together, equations A1.6a and A1.6b:

$$X + (Y + Z) = (X + Y) + Z \qquad \text{(A1.6a)}$$

or,

$$X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z \qquad \text{(A1.6b)}$$

3) The Commutative Law defines the order in which variables may appear in a function, equations A1.7a and A1.7b:

$$X + Y = Y + X \qquad \text{(A1.7a)}$$

or,

$$X \cdot Y = Y \cdot X \qquad \text{(A1.7b)}$$

4) The Absorption Law defines equivalent expressions for a set of Boolean expressions, equations A1.8a, A1.8b, A1.8c and A1.8d:

$$X + (X \cdot Y) = X \qquad \text{(A1.8a)}$$

or,

$$X \cdot (\bar{X} + Y) = X \qquad \text{(A1.8b)}$$

or,

$$X + (\bar{X} \cdot Y) = X + Y \qquad \text{(A1.8c)}$$

or,

$$X \cdot (\bar{X} + Y) = X \cdot Y \qquad \text{(A1.8d)}$$

5) The Complementation Law defines the result of operations applied to a single variable in its asserted and negated form, equations A1.9a. and A1.9b:

$$X + \bar{X} = 1 \qquad \text{(A1.9a)}$$

or,

$$X \cdot \bar{X} = 0 \qquad \text{(A1.9b)}$$

6) The rules of Boolean Algebra include a set of definitions for the result of miscellaneous operations as follows:

$$X + X = X \qquad \text{(A1.10a)}$$

$$X + 0 = X \qquad \text{(A1.10b)}$$

$$X + 1 = 1 \qquad \text{(A1.10c)}$$

$$X \cdot X = X \qquad \text{(A1.10d)}$$

$$X \cdot 1 = X \qquad \text{(A1.10e)}$$

$$X \cdot 0 = 0 \qquad \text{(A1.10f)}$$

7) De Morgan's Theorems

$$\overline{(X + Y)} = \bar{X} \cdot \bar{Y} \qquad \text{(A1.11a)}$$

$$\overline{(X \cdot Y)} = \bar{X} + \bar{Y} \qquad \text{(A1.11b)}$$

A1.5 Shannon's Expansion Theorem

Finally, a Boolean function can be expressed in terms of two other functions by means of Shannon's Expansion Theorem.

$$F = f(X_1, \ldots, X_n) = \bar{X}_i \cdot F1 + X_i \cdot F2 \qquad \text{(A1.12)}$$

such that $X_i = 0$ in F1 and $X_i = 1$ in F2.

The expansion theorem may be regarded as a means of factoring out a particular variable from an equation. The result is composed of two parts, an expression which includes the negated form of the factored variable and another with the asserted form. It may also be noted that the factored variable no longer appears in either of the sub-functions.

APPENDIX II

ISP REPRESENTATION OF COMPUTER STRUCTURES

A2.1 ISP Notation

Instruction Set Processor (ISP) notation was developed to provide a uniform symbolic language in which to describe the organization and operation of computers [Bell71,Siew74]. The organization of a computer or controller is described in ISP by declarations of memory, registers, data types, data operations, and instructions.

ISP declarations contain:

- the normally used name,
- an abbreviation to be used in subsequent definitions, separated from the normally used name by the alias operator "\", and
- a complete description including size and a numbering sequence.

Sharp brackets "< >" indicate the size of a register, including some counting scheme. Square brackets "[ ]" specify the number of identical entities that are present. The default, no square brackets, indicates that only one entity is present. The range operator ":" is used to denote an abbreviated list of elements. The order of the range numbers indicates the physical order of the range elements.

The replacement operator ":=" is used to define equivalence relationships among a set of elements. The causes and consequences of an instruction are represented by the conditional action operator "=>" where a logical condition describes when the consequential action sequence is invoked and the action sequence describes what resultant

transformations take place. The transfer operation "<-" denotes the exchange of bit patterns from one data carrier to another. Concurrent or alternative activities are described by the concurrency operator ";". Sequential activities are denoted by the word "next". Boolean operators (~,/\,\/) denoting complementation, conjunction and disjunction, respectively, and relational operators (=,≠,<,>) are also defined in ISP.

A2.2 ISP Representation of the Prototype BD Processor

A2.2.1 Processor Declarations

```
Processor state
     Program.Counter\PC<7:0>
     Output.Bank.Register\OB<3:0>

Primary memory
     Program.Memory\Mp[255:0]<15:0>

Console state
     Control.Switches\C.SW<7:0>
     Data.Switches\D.SW<15:0>
     Input.Switches\I.SW<7:0>
     Input.Pins\I.PN<3:0>
     Console.Lights\C.LT<23:0>
     Output.Lights\O.LT[1:0]<11:0>
     Auto.Manual.Switch\A.SW
```

```
Reset.Button\PB
Output.Pins\O.PN<3:0>

mP/BD interface state
Control.Register\CR<7:0>
Data.Register\DR<15:0>
I/O.Register\IOR<15:0>
Interrupt.Flag\Interrupt
End.of.Program.Flag\EOP
Single.Step.Strobe\SS.Strobe
Memory.Write.Strobe\W.Strobe
BD.Clock.On\BD.CLK.ON

Input/Output state
Input.Data.Type.Logical\X
Input.Variable\IV<63:0>
Output.Data.Type.Logical.Multibit\OV<11:0>
Output.Registers\OR[15:0]<11:0>
```

A2.2.2 Processor State Definitions

```
System.Clock:=(
    (C.SW<3>=1 => System.Clock=70kHz);
    (C.SW<3>=0 => System.Clock=0.5Hz))
Clock\CLK:=(
    (A.SW=0/\
        (C.SW<7>=1 => CLK:=System.Clock);
```

```
            (C.SW<7>=0 => CLK:=C.SW<0>));
        (A.SW=1/\
            (CR<4>=0 => CLK:=System.Clock);
            (CR<4>=1 => CLK:=SS.Strobe)))

Console.Lights:=(
    (A.SW=0/\
        (C.SW<2>=1 => C.LT<7:0>:=PC;
                      C.LT<23:8>:=Mp[PC]);
        (C.SW<2>=0 => C.LT<7:0>:=PC;
                      C.LT<23:8>:=D.SW<15:0>));
    (A.SW=1 => C.LT<7:0>:=PC;
               C.LT<23:8>:=Mp[PC]))

I/O.Mapping:=(
    (IV<3:0>:=I.SW<3:0>);
    (IV<19:16>:=I.SW<7:4>);
    (IV<35:32>:=I.PN<3:0>);
    (OR[OB]<11:0>:=OV<11:0>);
    (O.LT[1:0]<11:0>:=OR[1:0]<11:0>);
    (O.PN<3:0>:=OR[1]<11:8>))

Clear.PC:=(
    (A.SW=0/\C.SW<6>=0 => PC <- 0))
```

```
Preset.PC:=(
     (A.SW=0/\C.SW<7:0>=0100x011/\CLK => PC <- D.SW<7:0>);
     (A.SW=1/\CR<7:0>=00010111/\CLK => PC <- DR<7:0>))

Load.Memory:=(
     (A.SW=0/\C.SW<7:0>=0101x001 => Mp[PC] <- D.SW<15:0>);
     (A.SW=1/\CR<7:0>=00110111/\W.Strobe => Mp[PC] <- DR<15:0>))

Read.Memory:=(
     (A.SW=0/\C.SW<7:0>=0101x111 => C.LT<23:8> <- Mp[PC]);
     (A.SW=1/\CR<7:0>=00110110 => DR<15:0> <- Mp[PC]))

Step.Through:=(
     (A.SW=0/\C.SW<7>=0/\CLK);
     (A.SW=1/\CR<7:0>=11110110/\CLK))

Reset.BD:=(
     (A.SW=1/\PB=1);
     (A.SW=0/\C.SW<7>=0/\PB=1))

Run:=(
     (A.SW=0/\
          (C.SW<7:0>=1111x111));
     (A.SW=1/\
          (CR<7:0>=11101110))
```

```
~Run:=(
     (A.SW=0/\
          (C.SW<7>=0/\EOP=1/\Interrupt=0));
     (A.SW=1/\
          (CR<7:0>=11110110/\EOP=1/\Interrupt=0));
     (Reset.BD))

BD.Clock.On:=Run

Verify:=(
     (A.SW=1/\CR<7:0>=10101100 =>
               IV<35:32>:=IOR<3:0>;
               OR[0]<11:0>:=IOR<15:4>))
```

A2.2.3 Instruction Format

Instruction\IN<15:0>:=Mp[PC]	All instructions
Transfer.Address:=IN<7:0>	Input, Short Output, Interrupt, Bank Select, and Jump instructions
Interrupt.Data:=IN<11:0>	EOP instruction
Long.Output.Data:=IN<11:0>	Long Output instruction
Short.Output.Data:=IN<11:8>	Short Output instruction
Output.Bank.Register\OB:=IN<11:8>	Bank Select instruction
Abbreviated.Operation.Code\A.OP<1:0>:=IN<15:14>	Input instructions
Full.Operation.Code\OP<3:0>:=IN<15:12>	All other instructions

A2.2.4 Instruction Execution

```
Instruction.Set:=(
(A.OP=00/\X=0 => PC <- IN<7:0>);                          Input Instruction
(A.OP=0/\X=1 => PC <- PC+1);                              Input Instruction
(A.OP=01/\X=1 => PC <- IN<7:0>);                          Input Instruction
(A.OP=01/\X=0 => PC <- PC+1);                             Input Instruction
(OP=1000 => OR[OB]<11:0> <- IN<11:0>;PC <- PC+1);         Long Output In-
                                                          struction
(OP=1100 => OR[OB]<11:8> <- IN<11:8>;PC <- IN<7:0>);      Short Output In-
                                                          struction
(OP=1010 => EOP=1;PC <- PC+1);                            EOP Instruction
(OP=1101 => INTERRUPT=~INTERRUPT;PC <- IN<7:0>);          Interrupt In-
                                                          struction
(OP=1110 => OB<3:0> <- IN<11:8>;PC <- IN<7:0>);           Bank Switch In-
                                                          struction
(OP=1111 => PC <- IN<7:0>))                               Jump Instruction
```

A2.2.5 Interpreter

```
Instruction.Interpreter:=(
(Run=1 => Instruction.Execution;
        next Instruction.Interpreter))
```

APPENDIX III

A3.0 BDBUG PROGRAM LISTING

```
*
*****************************************
*                                       *
*       BDBUG - BD MONITOR              *
*       VERSION 1.1 - MARCH 1984        *
*                                       *
*       R.D. HUDSON                     *
*       DATAC COMPUTER LABORATORY       *
*       MCGILL UNIVERSITY               *
*       MONTREAL CANADA                 *
*                                       *
*       FOR USE WITH A BD/6809          *
*       EVALUATION SYSTEM W/            *
*       V-BUG MONITOR AND FLEX 9.0      *
*       DOS                             *
*                                       *
*****************************************
*
*  -- SYSTEM EQU'S --
*
PIAORA   EQU   $E030          H/W EQU'S
PIACRA   EQU   PIAORA+1
PIAORB   EQU   PIAORA+2
PIACRB   EQU   PIAORA+3
BDMCR    EQU   PIAORA+$E
ACIACR   EQU   $E004
ACIASR   EQU   ACIACR
ACIADR   EQU   ACIASR+1
PINIT    EQU   $FC6F          V-BUG EQU'S
BADDR2   EQU   $FD20
BADDR    EQU   $FD2B
BYTE     EQU   $FD3C
OUT4H    EQU   $FD6A
OUT2H    EQU   $FD72
PSTRNG   EQU   $FD9E
PCRLF    EQU   $FDA2
PDATA    EQU   $FDAE
INCHE    EQU   $FDBA
INCH     EQU   $FDC0
OUTS     EQU   $FDDD
OUTCH    EQU   $FDDF
ACINIZ   EQU   $FE10
*
*  --- BDBUG INITIALIZATION ---
*
         ORG   $A000
*
BDBUG    ORCC  #$50           SET INTERRUPT MASK
         TFR   S,U
         LDD   #RTX
         STD   $DFC8          STORE IRQ VECTOR
         LDA   #$91           INITIALIZE ACIA
         STA   ACIACR
```

```
          LBSR   TOMP        INITIALIZE PIA
          LDX    #$E060
          LDA    #1
          LBSR   PINIT       INITIALIZE EVENT PRINTER
          CLR    $DFE5
          BSR    BDIRQ       GET ACTUAL BDM STATUS
          LDA    PIAORA      CLEAR INTERRUPTS
          LDA    PIAORB
          LDA    ACIADR
*
BUG2      LDX    #MSG1       PRINT PROMPT '  ' CHARACTERS
          LBSR   PSTRNG
          CWAI   #$EF        ENABLE INTERRUPTS AND WAIT
          BRA    BUG2
*
*  --- INTERRUPT DECODER ---
*
RTX       TST    PIACRA      SOFTWARE (CA1) INTERRUPT?
          BPL    RTX1
          BSR    BDSWI
RTX1      TST    PIACRB      HARDWARE (CB1) INTERRUPT?
          BPL    RTX2
          BSR    BDHWI
RTX2      LDA    ACIASR      OPERATOR (ACIA) INTERRUPT?
          BPL    RTX3
          LSRA
          BCC    RTX3
          LBSR   KEYBRD
RTX3      RTI
*
*  -- SOFTWARE INTERRUPT (CA1) SERVICE ROUTINES --
*
BDSWI     LDX    #MSG12      'BD PGM INTERRUPT'
          LBSR   PSTRNG
          LDA    PIAORA
          RTS                RETURN
*
*  -- HARDWARE INTERRUPT (CB1) SERVICE ROUTINES --
*
BDHWI     LDD    #$FF3F      SET LOOP COUNTER
          PSHS   A
          STB    PIACRB      SET CONTROL REGISTERS
          CLR    CLKREG      FOR POLLING
          CLR    AMREG
          LDA    BDMTR
          LDB    PIAORB      CLEAR EXISTING INTERRUPT
*
HWI1      ORA    #$C         TEST RUNNING/AUTO
          STA    BDMCR
          TST    PIACRB
          BMI    HWI2
          ANDA   #$F7        TEST STOPPED/AUTO
          STA    BDMCR
```

```
          TST   PIACRB
          BMI   HWI2
          ANDA  #$F3          TEST STOPPED/MANUAL
          STA   BDMCR
          TST   PIACRB
          BMI   HWI2
          ORA   #8            TEST RUNNING/MANUAL
          STA   BDMCR
          TST   PIACRB
          BMI   HWI2
          DEC   0,S           IF INCONCLUSIVE, LOOP AND
          BNE   IRQ1          POLL AGAIN
          LDX   #MSG2
          LBSR  PSTRNG        REPORT POLLING FAILURE
*
HWI2      LDB   #$3D          RESET PIA CONTROL REG
          STB   PIACRB
          BITA  #8            SET STATUS REGISTERS
          BEQ   HWI3
          COM   CLKREG
HWI3      BITA  #4
          BEQ   HWI4
          COM   AMREG
HWI4      LDD   #$F2EA
          TST   AMREG
          BEQ   HWI5
          LDD   #$F6EE
HWI5      TST   CLKREG        SET BD CONTROL REG
          BEQ   HWI6
          TFR   B,A
HWI6      STA   BDMCR
          STA   BDMTR
          LDA   PIAORB
*
          LEAS  1,S           RESTORE STACK
          LBSR  BDSTAT        REPORT BD STATUS
          RTS                 RETURN
*
*  -- OPERATOR REQUEST SERVICE ROUTINES --
*
KEYBRD    LBSR  INCHE
          CMPA  #$2F          '/'?...IF NOT, ABORT
          BNE   KBD2
          LBSR  INCHE         READ COMMAND CODE
          TFR   A,B
          LBSR  OUTS
          LDX   #CMDTAB
KBD1      CMPB  0,X+          MATCH COMMAND TO TABLE
          BEQ   KBD3
          LEAX  2,X
          CMPX  #TABEND       NOT FOUND?...ALERT OPERATOR
          BNE   KBD1
KBD2      LDX   #MSG2         ' WHAT?'
```

```
          LBSR  PDATA
          RTS                 RETURN
KBD3      JSR   [0,X]         SERVICE OPX REQUEST
          RTS                 RETURN
*
*   COMMAND JUMP TABLE
*
CMDTAB    FCC   'L'
          FDB   BDLOAD
          FCC   'E'
          FDB   BDEXM
          FCC   'M'
          FDB   BDMCHG
          FCC   'J'
          FDB   BDRUN
          FCC   'H'
          FDB   BDHALT
          FCC   'D'
          FDB   BDSTAT
          FCC   'T'
          FDB   BDTRNS
          FCC   'S'
          FDB   BDSTEP
          FCC   '+'
          FDB   FLEX
          FCC   '>'
          FDB   MON
          FCC   'P'
          FDB   PRNTR
TABEND    EQU   *
*
*  /D - REPORT BD STATUS
*
BDSTAT    LDX   #MSG8         'BD STATUS: '
          LBSR  PSTRNG
          LDX   #MSG9A        'MANUAL/'
          TST   AMREG
          BEQ   STAT1
          LDX   #MSG9B        'AUTO/'
STAT1     LBSR  PDATA
          LDX   #MSG10A       'STOPPED'
          TST   CLKREG
          BEQ   STAT2
          LDX   #MSG10B       'RUNNING'
STAT2     LBSR  PDATA
          RTS                 RETURN
*
*  /P - PRINTER TOGGLE
*
PRNTR     TST   $DFE5         TEST PRINTER STATUS
          BNE   PRNT1
          INC   $DFE5         TOGGLE ON IF OFF
          RTS
```

```
PRNT1   CLR   $DFE5       TOGGLE OFF IF ON
        RTS
*
*   'ESC' STOPS LISTING
*
ESCTRP  LDA   ACIASR      KEY HIT?
        BITA  #1
        BEQ   ESC2
        LDA   ACIADR      READ CHARACTER
        ANDA  #$7F
        CMPA  #$1B
        BNE   ESC2        RETURN IF NOT 'ESC'
ESC1    LDA   ACIASR
        BITA  #1
        BEQ   ESC1        WAIT FOR ANOTHER HIT
        LDA   ACIADR
        ANDA  #$7F
        CMPA  #$1B
        BEQ   ESC2        RETURN IF 'ESC'
        CMPA  #$0D
        BNE   ESC1
        ORCC  #2          SET V-FLAG IF 'CR'
ESC2    RTS               RETURN
*
*  /> - TRANSFER TO MONITOR
*
MON     LEAS  4,S         RESTORE STACK TO INTERRUPT
        LDD   #$FF00
        STD   10,S        CHANGE RTI RETURN ADDRESS
        RTI
*
*  /+ - TRANSFER TO FLEX
*
FLEX    LBSR  ACINIZ      RESTORE ACIA COMM.
        LDA   #$D
        STA   $C011       CLEAN FLEX LINE BUFFER
        LEAS  4,S         RESTORE STACK TO INTERRUPT
        LDD   #$CD03
        STD   10,S        CHANGE RTI RETURN ADDRESS
        RTI
*
        LIB   BDBUG2

*  -- BDBUG MESSAGES AND BUFFERS --
*
MSG1    FCC   '  ',$4
MSG2    FCC   ' WHAT?',$4
MSG5    FCC   '   - ',$4
MSG8    FCC   'BD STATUS: ',$4
MSG9A   FCC   'MANUAL/',$4
MSG9B   FCC   'AUTO/',$4
MSG10A  FCC   'STOPPED',$4
MSG10B  FCC   'RUNNING',$4
```

```
MSG12    FCC   'BD PGM INTERRUPT',$4
*
         ORG   *
*
AMREG    RMB   1
CLKREG   RMB   1
BDMTR    RMB   1
*
         END   BDBUG
```

```
*
*  /M - BD MEMORY CHANGE AT $BB
*
BDMCHG   TST    CLKREG      TEST BD IS STOPPED AND
         BNE    CHG2        IN AUTO, ABORT IF NOT
         TST    AMREG
         BEQ    CHG2
         LBSR   BYTE        READ BD ADDRESS
         BVS    CHG2
CHG1     LBSR   PRSTPC      SET BD PC
         TFR    D,Y
         LDX    #MSG5
         LBSR   PSTRNG
         TFR    Y,D
         LBSR   OUT2H       PRINT ADDRESS
         LBSR   OUTS
         LBSR   TOMP
         LBSR   RDWRD       GET DATA FROM BD AND PRINT
         TFR    D,X
         LBSR   OUT4H
         LBSR   OUTS
         LBSR   BYTE        READ USER RESPONSE
         BVC    CHG3
         CMPA   #'^         '^'?...GO BACK ONE STEP
         BEQ    CHG5
         CMPA   #$D         'CR'?...ABORT
         BNE    CHG4
         LDA    #$F6
         STA    BDMCR
         STA    BDMTR
CHG2     RTS
CHG3     PSHS   A
         LBSR   BYTE        DATA?...READ NEXT BYTE
         BVS    CHG2
         TFR    A,B
         PULS   A
         PSHS   A,B
         LBSR   TOBD
         LBSR   LDWRD       STORE IN BD
         LBSR   TOMP
         LBSR   RDWRD       READ BACK TO CONFIRM DATA
         CMPD   0,S++
         BEQ    CHG4
         LDA    #'?         PRINT '?' IF ERROR
         LBSR   OUTCH
CHG4     TFR    Y,D         OTHERWISE GO FORWARD ONE STEP
         INCA
         BRA    CHG1
CHG5     TFR    Y,D
         DECA
         BRA    CHG1
*
*  /J - START BD AT $BB
```

```
*
BDRUN   TST   CLKREG     TEST BD IS STOPPED AND
        BNE   RUN1       IN AUTO, ABORT IF NOT
        TST   AMREG
        BEQ   RUN1
        LBSR  BYTE       READ FIRST ADDRESS AND
        BVS   RUN1       PRESET BD PC
        LBSR  PRSTPC
        LBSR  TOBD
        LDA   #$EE       RESTART BD
        STA   BDMCR
        STA   BDMTR
RUN1    RTS              RETURN
*
*  /E - EXAMINE BD MEMORY $SS-$EE, SAVE AT MP $XXXX
*
BDEXM   TST   CLKREG     TEST BD IS STOPPED AND
        BNE   EXM3       IN AUTO, ABORT IF NOT
        TST   AMREG
        BEQ   EXM3
        LBSR  BYTE       READ STARTING BD ADDRESS
        BVS   EXM3
        LBSR  PRSTPC     PRESET BD PC
        PSHS  A
        LDA   #'-
        LBSR  OUTCH
        LBSR  BYTE       READ LAST BD ADDRESS
        BVS   EXM2
        SUBA  0,S        CALCULATE WORD COUNT TO
        INCA             TRANSFER
        STA   0,S
        LBSR  OUTS
        LBSR  BADDR      READ mP BUFFER ADDRESS
        BVS   EXM2
        TFR   X,Y
        LBSR  PCRLF
        LBSR  PCRLF
        LBSR  TOMP
EXM1    LBSR  RDWRD      READ BD DATA, SAVE IN mP,
        STD   0,Y++      AND OUTPUT TO SCREEN
        TFR   D,X
        LBSR  OUTS
        LBSR  OUT4H
        LBSR  STEP
        DEC   0,S        DECREMENT WORD COUNT
        BNE   EXM1       AND LOOP
EXM2    LEAS  1,S
EXM3    LDA   #$F6
        STA   BDMCR
        STA   BDMTR
        RTS              RETURN
*
*  /L - LOAD BD MEMORY $SS-$EE FROM MP $XXXX
```

```
*
BDLOAD  TST     CLKREG      TEST BD IS STOPPED AND
        BNE     LOAD3       IN AUTO, ABORT IF NOT
        TST     AMREG
        BEQ     LOAD3
        LBSR    BYTE        READ FIRST BD ADDRESS
        BVS     LOAD3
        LBSR    PRSTPC      PRESET BD PC
        PSHS    A
        LDA     #'-
        LBSR    OUTCH
        LBSR    BYTE        READ LAST BD ADDRESS
        BVS     LOAD2
        SUBA    0,S         CALCULATE WORD COUNT
        INCA
        STA     0,S
        LBSR    OUTS
        LBSR    BADDR       READ mP BUFFER ADDRESS
        BVS     LOAD2
LOAD1   LDD     0,X++
        LBSR    LDWRD       STORE DATA IN BD
        LBSR    STEP
        DEC     0,S         DECREMENT WORD COUNT AND
        BNE     LOAD1       LOOP
LOAD2   LEAS    1,S
LOAD3   LDA     #$F6
        STA     BDMCR
        STA     BDMTR
        RTS                 RETURN
*
*  /S - SINGLE STEP BD STARTING AT $BB
*
BDSTEP  TST     CLKREG      TEST BD IS STOPPED AND
        BNE     STP2        IN AUTO, ABORT IF NOT
        TST     AMREG
        BEQ     STP2
        LBSR    BYTE        READ STARTING BD ADDRESS
        BVS     STP2
        LBSR    PRSTPC      PRESET BD PC
        LDB     #$F6
        STB     BDMCR       EXECUTE INSTRUCTION AND
STP1    LBSR    INCH        READ USER RESPONSE
        ANDA    #$7F
        CMPA    #$D         'CR'?...ABORT
        BEQ     STP2
        LBSR    STEP        OTHERWISE SINGLE STEP
        BRA     STP1        AND LOOP
STP2    LDA     #$F6
        STA     BDMCR
        STA     BDMTR
        RTS                 RETURN
*
*  /H - HALT BD AT NEXT EOP
```

```
*
BDHALT   TST    CLKREG     TEST BD IS RUNNING AND
         BEQ    HALT1      IN AUTO, ABORT IF NOT
         TST    AMREG
         BEQ    HALT1
         LBSR   TOMP
         LDA    #$F6       SEND STOP CODE
         STA    BDMCR
         STA    BDMTR
HALT1    RTS               RETURN
*
*  /T - TRANSLATE BD PGM IN mP $SSSS-$EEEE
*       FROM ADDRESS $XX TO $YY
*
BDTRNS   LBSR   BADDR2     READ mP ADDRESSES
         BVS    TRANS4
         PSHS   Y
         TFR    X,D
         SUBD   0,S++      CALCULATE WORD COUNT
         BLT    TRANS4     ABORT IF   ZERO
         TFR    D,X
         LEAX   1,X
         LBSR   OUTS
         LBSR   BYTE       READ ORIGINAL BD ADDRESS
         BVS    TRANS4
         PSHS   A
         LDA    #'-
         LBSR   OUTCH
         LBSR   BYTE       READ NEW BD ADDRESS
         BVS    TRANS3
         SUBA   0,S        CALCULATE OFFSET
         STA    0,S
TRANS1   LDD    0,Y
         ANDA   #$F0
         CMPA   #$A0       'EOP'?...DON'T OFFSET
         BEQ    TRANS2
         CMPA   #$80       'LONG OUTPUT'?...DON'T OFFSET
         BEQ    TRANS2
         ADDB   0,S        OTHERWISE ADD OFFSET AND
TRANS2   LDA    0,Y        LOOP
         STD    0,Y++
         LEAX   -1,X
         BNE    TRANS1
TRANS3   LEAS   1,S
TRANS4   RTS               RETURN
*
*  AVAILABLE SUBROUTINES
*
PRSTPC   LBSR   TOBD       PRESET BD PC
         LDB    #$17
         STB    BDMCR
         STA    PIAORA
STEP     PSHS   A,B
```

```
         LDD    #$353D
         STA    PIACRA
         STB    PIACRA
         PULS   A,B
         RTS
*
TOBD     PSHS   A,B          SET INTERFACE FOR mP TO
         LDD    #$39FF       BD COMMUNICATION
         STA    PIACRA
         STA    PIACRB
         STB    PIAORA
         STB    PIAORB
         LDA    #$3D
         STA    PIACRA
         STA    PIACRB
         PULS   A,B
         RTS
*
TOMP     PSHS   A,B          SET INTERFACE FOR BD TO
         LDD    #$393D       mP COMMUNICATION
         STA    PIACRA
         STA    PIACRB
         CLR    PIAORA
         CLR    PIAORB
         STB    PIACRA
         STB    PIACRB
         PULS   A,B
         RTS
*
LDWRD    PSHS   A,B          LOAD ONE WORD TO
         LDB    #$37         BD MEMORY
         STB    BDMCR
         LDB    1,S
         STB    PIAORA
         STA    PIAORB
         LDD    #$353D
         STA    PIACRB
         STB    PIACRB
         PULS   A,B
         RTS
*
RDWRD    LDA    #$36         READ ONE WORD FROM
         STA    BDMCR        BD MEMORY
         LDA    PIAORB
         LDB    PIAORA
         RTS
*
```

APPENDIX IV

A4.0 BDC-4 OPTIMIZING COMPILER LISTING

```
          NAM   BDC-4
*
*****************************************
*                                       *
*    BDC-4 : BD PROGRAM COMPILER        *
*    VERSION 4.1                        *
*    MARCH 1984                         *
*                                       *
*    R.D. HUDSON & A. KUCUK             *
*    DATAC COMPUTER LABORATORY          *
*    MCGILL UNIVERSITY                  *
*    MONTREAL CANADA                    *
*                                       *
*    REQUIRES 6809 MPU, FLEX            *
*    9.0 DOS AND V-BUG MONITOR          *
*                                       *
*****************************************
*
*  --- SYSTEM EQU'S ---
*
ACIASR    EQU    $E004        HARDWARE EQU'S
ACIADR    EQU    $E005
INCHE     EQU    $F806        V-BUG EQU'S
OUTCH     EQU    $F80A
PDATA     EQU    $F80C
PCRLF     EQU    $F80E
PSTRNG    EQU    $F810
INLINE    EQU    $F818
DECBIN    EQU    $F81A
BINDEC    EQU    $F81C
PINIT     EQU    $FC6F
BADDR     EQU    $FD2B
OUT4H     EQU    $FD6A
OUT2H     EQU    $FD72
OUT2S     EQU    $FDDB
LINBUF    EQU    $C080        FLEX EQU's
DOCMND    EQU    $CD4B
*
*  -- BDC-4 BUFFER ASSIGNMENTS --
*
          ORG    $0000
*
CONSTA    RMB    1
CONSTB    RMB    1
CONSTC    RMB    1
CNTR      RMB    2
DIF       RMB    1
FNCTN     RMB    1
INLIST    RMB    8
INPUTS    RMB    1
LASTX     RMB    2
LASTY     RMB    2
LVLCTR    RMB    1
MCODE     RMB    2
```

```
NODE     RMB   1
OFFSET   RMB   2
OUTBNK   RMB   1
OUTNUM   RMB   2
OUTPTS   RMB   1
PC1      RMB   2
PGMADD   RMB   2
PGMCLS   RMB   2
PGMNUM   RMB   2
PGMSIZ   RMB   2
PNTR1    RMB   2
PNTR2    RMB   2
PRP      RMB   1
RSLT1    RMB   2
SUB      RMB   10
TEMP     RMB   2
TREE     RMB   2
*
*  -- INPUT BDC-4 PARAMETERS AND OPTIONS --
*
*    PGMNUM - BD PROGRAM NUMBER
*    PGMCLS - BD PROGRAM CLASS
*    INPUTS - NUMBER OF SERIAL INPUT VARIABLES
*    OUTPTS - NUMBER OF PARALLEL OUTPUT VARIABLES
*    OUTBNK - BD PROGRAM OUTPUT BANK
*    INLIST - SERIAL INPUT CHANNELS
*    CNTR   - TEMPORARY LOCATION
*
         ORG   $0200
*
BDC      LDU   #$0200      DEFINE USER STACK
         LDX   #MSG0       HARDCOPY REQUEST
         JSR   [PSTRNG]
         JSR   [INCHE]
         CMPA  #'Y
         BNE   BDC1
         LDX   #$E060      YES?...THEN INITIALIZE PRINTER
         LDA   #1
         JSR   PINIT
*
BDC1     LDX   #MSG1
         JSR   [PSTRNG]    PRINT MENU
         LBSR  INDATA      INPUT OPTION CODE
         BVS   BDC1
         CMPD  #3
         BLT   BDC1A
         LDD   #0
BDC1A    CMPD  #0
         BGT   BDC1B       VALID OPTION CODE?
         LDX   #MSG5       IF NOT, REPEAT MENU
         JSR   [PSTRNG]
         BRA   BDC1
BDC1B    PSHU  D
*
BDC2     LDX   #MSG2B      ENTER PROGRAM NUMBER
```

```
        JSR   [PSTRNG]
        LBSR  INDATA
        BVS   BDC2
        LBSR  BINASC
        STD   PGMNUM
*
BDC3    LDX   #MSG2C      ENTER PROGRAM CLASS
        JSR   [PSTRNG]
        LBSR  INDATA
        BVS   BDC3
        LBSR  BINASC
        STD   PGMCLS
*
BDC4    LDD   0,U         ASSUME 7 INPUTS IF TF
        CMPD  #1
        BEQ   BDC4A
        LDA   #$37
        STA   INPUTS
        BRA   BDC5
BDC4A   LDX   #MSG2       OTHERWISE ENTER NUMBER
        JSR   [PSTRNG]    OF INPUT CHANNELS
        LBSR  INDATA
        BVS   BDC4
        LBSR  BINASC
        STB   INPUTS
*
BDC5    LDX   #MSG2A      ENTER NUMBER OF OUTPUT
        JSR   [PSTRNG]    CHANNELS
        LBSR  INDATA
        BVS   BDC5
        STB   OUTPTS
*
        LDA   #$31        ENTER INPUT CHANNEL
        STA   CNTR        ASSIGNMENTS AND STORE
        LDY   #INLIST     IN LIST
BDC6    LDX   #MSG2D
        JSR   [PSTRNG]
        LDA   CNTR
        JSR   [OUTCH]
        JSR   OUT2S
        LBSR  INDATA
        BVS   BDC6
        STB   0,Y+
        INC   CNTR
        LDA   CNTR
        CMPA  INPUTS
        BLE   BDC6
        SUBA  #$31
        STA   INPUTS
        STA   CNTR
*
BDC7    LDX   #MSG2E      ENTER OUTPUT BANK
        JSR   [PSTRNG]
        LBSR  INDATA
        BVS   BDC7
```

```
          STB    OUTBNK
*
BDC8      PULU   D            RESTORE OPTION CODE
          CMPD   #1
          BEQ    ENTER
          BRA    TFENT
*
*  -- INPUT TRUTH TABLE --
*
*   OUTNUM - NUMBER OF BYTES IN OUTPUT VECTOR
*   OFFSET - SUB-TREE DISPLACEMENT IN TRUTH TABLE
*   TRUTAB - ADDRESS OF FIRST BYTE IN TRUTH TABLE
*   TREE   - ADDRESS OF BYTE FOLLOWING END OF
*            TRUTH TABLE
*
ENTER     LDB    #1            CALCULATE TRUTH TABLE LENGTH
ENT1      ASLB                 EQUAL TO 2^N (N=# OF INPUTS)
          ROLA
          DEC    CNTR
          BNE    ENT1
          STD    OUTNUM
*
ENT2      LDY    #TRUTAB
          JSR    [PCRLF]
          LDX    #MSG3         PRINT INPUT REQUEST PART ONE
          JSR    [PSTRNG]
          LDX    #LINBUF
          LDD    OUTNUM
          JSR    [BINDEC]      CONVERT DATA ELEMENT COUNT
          LDA    #4            INTO DECIMAL BEFORE PRINTING
          STA    0,X
          LDX    #LINBUF
ENT3      LDA    0,X+
          CMPA   #$30
          BEQ    ENT3
          LEAX   -1,X
          JSR    [PDATA]
          LDX    #MSG4
          JSR    [PDATA]
*
          LDD    OUTNUM        INITIALIZE DATA COUNTER
          STD    CNTR
          JSR    [PCRLF]
*
ENT4      LDA    #4            INPUT DATA IN GROUPS OF FOUR
          STA    -1,U
ENT4A     JSR    BADDR
          BVC    ENT4B
          LDX    #MSG5         REPORT ERROR IF NOT HEX
          JSR    [PDATA]
          BRA    ENT4A
ENT4B     JSR    [PCRLF]
          STX    0,Y           STORE DATA IN TRUTH TABLE
          LEAY   4,Y
          LDX    CNTR
```

```
          LEAX  -1,X
          STX   CNTR
          BEQ   ENT5
          DEC   -1,U
          BGT   ENT4A
          JSR   [PCRLF]
          BRA   ENT4
*
ENT5      LDD   OUTNUM      CALCULATE NUMBER OF BYTES OF
          ASLB              STORAGE USED
          ROLA
          ASLB
          ROLA
          STD   OFFSET
          ADDD  #TRUTAB     CALCULATE DATA DELIMITING
          STD   TREE        ADDRESSES
*
ENT6      BRA   BDCMPIL
*
*  -- TABULATE THE VALUES OF THE TF=(A/S+B+C*S) --
*
*         FNCTN - THE INPUT FUNCTION (7 BITS)
*         DIF   - DIFFERENTIAL PART OF TF;
*                 3 L.S. BITS OF FNCTN
*         PRP   - PROPORTIONAL PART OF TF;
*                 4 M.S. BITS OF FNCTN
*         RSLT1 - PARTIAL RESULT BUFFER
*
TFENT     LDD   #$80        NO. OF BYTES IN OUTPUT VECTOR
          ASLB
          ROLA
          ASLB
          ROLA
          STD   OFFSET
          ADDD  #TRUTAB
          STD   TREE
*
TF1       LDX   #MSG20      PRINT THE TYPE OF
          JSR   [PSTRNG]    TRANSFER FUNCTION
          LBSR  INDATA
          BVS   TF1
          STB   CONSTA
*
TF2       LDX   #MSG21B     PRINT INPUT REQUEST
          JSR   [PSTRNG]    AND ENTER CONSTANT B
          LBSR  INDATA
          BVS   TF2
          STB   CONSTB
*
TF3       LDX   #MSG21C     PRINT INPUT REQUEST
          JSR   [PSTRNG]    AND ENTER CONSTANT C
          LBSR  INDATA
          BVS   TF3
          STB   CONSTC
*
```

```
            CLR   FNCTN
TF4         LDA   FNCTN       GET INPUT FUNCTION
            ANDA  #$7         DIFFERENTIAL PART OF THE TF
            STA   DIF
            LDB   FNCTN
            LSRB
            LSRB
            LSRB
            STB   PRP         PROPORTIONAL PART OF THE TF
            LDD   #$0
            STD   RSLT1
            LDA   CONSTB      MULTIPLICATION OF CONSTB
            LDB   PRP         WITH PROPORTIONAL PART
            MUL
            ADDD  RSLT1
            STD   RSLT1
            LDA   CONSTC      MULTIPLICATION OF CONSTC
            LDB   DIF         WITH DIFFERENTIAL PART
            MUL
            ADDD  RSLT1
            STD   RSLT1
            ANDB  #$FE        DROP OF LSB
            STD   ,Y++
            LDB   FNCTN       STORE INPUT FUNCTION AND
            LDA   #$0         OUTPUT OF TF
            STD   ,Y++
            INC   FNCTN
            LDA   FNCTN
            CMPA  #$80        IS END OF FUNCTION REACHED?
            BNE   TF4         NO?...CONTINUE CALCULATION
TF5         NOP               CONTINUE
*
            LIB   BDC-4A
            END   BDC
```

```
BDCMP4   LEAX  8,X        INCREMENT CURRENT ADDRESS
         PULU  Y          GET ADDRESS OF LAST INCOMPLETE
         CMPY  #0         NODE FROM STACK AND CHECK
         BEQ   BDCMP5     IF ITS THE DUMMY NUMBER
*                         IF SO THE PROGRAM IS FINISHED
         STX   4,Y        IF NOT COMPLETE 'TO' POINTER
         LDA   0,Y        OF THAT NODE, ADJUST NODE
         STA   NODE       ACCORDINGLY AND STORE NEW
         STY   LASTX      LASTX
         BRA   BDCMP2
*
BDCMP5   STX   MCODE
         NOP   CONTINUE
*
*  --- PRUNE BD TABLE ---
*
*   LVLCTR - PRUNING LEVEL COUNTER
*   PNTR1  - MASTER SUB-TREE POINTER
*   PNTR2  - TRIAL SUB-TREE POINTER
*
PRUNE    LDD   OFFSET     DIVIDE TRUTH TABLE IN TWO
         LSRA
         RORB
         STD   OFFSET
         LDA   #1         INITIALIZE SUB-TREE LEVEL COUNTER
         STA   LVLCTR
*
PRN1     LDD   #TRUTAB    ADDRESS MASTER SUB-TREE
         STD   PNTR1
PRN2     TST   [PNTR1]    IF MASTER SUB-TREE IS CANCELLED
         BMI   PRN5       SELECT ANOTHER
         LDD   PNTR1
         ADDD  OFFSET     ELSE ADDRESS TRIAL SUB-TREE
         STD   PNTR2      STORE IN POINTER
*
PRN3     TST   [PNTR2]    IF TRIAL SUB-TREE IS CANCELLED
         BMI   PRN4       SELECT ANOTHER,
         BSR   CMPARE     ELSE BEGIN COMPARISON
         BVC   PRN4
         BSR   ADJUST     PRUNE SUB-TREE IF REDUNDANT
*
PRN4     LDD   PNTR2      ADDRESS NEXT TRIAL SUB-TREE
         ADDD  OFFSET
         STD   PNTR2
         CMPD  TREE       MORE SUB-TREES?
         BLO   PRN3       IF YES, CONTINUE PRUNING
*
PRN5     LDD   PNTR1      ELSE ADDRESS NEXT MASTER SUB-TREE
         ADDD  OFFSET
         STD   PNTR1
         ADDD  OFFSET
         CMPD  TREE       MORE SUB-TREES?
         BLO   PRN2       IF YES, CONTINUE PRUNING
*
         INC   LVLCTR     ELSE REDUCE OFFSET SIZE
```

```
          LDD   OFFSET
          LSRA
          RORB
          STD   OFFSET      IF SUB-TREE SIZE IS NON-ZERO
          CMPD  #2
          BNE   PRN1        CONTINUE PRUNING
          BRA   MACHINE     ELSE BEGIN MACHINE CODE
*
*  -- COMPARE MASTER AND TRIAL SUB-TREES --
*
CMPARE    LDD   OFFSET      CALCULATE NUMBER OF LEAF NODES
          LSRA              TO COMPARE = OFFSET/4
          RORB
          LSRA
          RORB
          PSHS  A,B
          LDX   PNTR1       ADDRESS MASTER AND TRIAL
          LDY   PNTR2       SUB-TREES
*
CMP1      LDD   0,X         COMPARE LEAF NODES IN SEQUENCE
          CMPD  0,Y
          BNE   CMP2        ABORT ROUTINE IF NOT EQUAL
          LEAX  4,X
          LEAY  4,Y
          LDD   0,S         DECREMENT NODE COUNTER
          SUBD  #1          AND LOOP
          STD   0,S
          BNE   CMP1
          ORCC  #2          SET OVERFLOW FLAG IF SUB-TREES
CMP2      LEAS  2,S         ARE REDUNDANT
          RTS
*
*  -- ADJUST SIBLING POINTERS AND CANCEL REDUNDANT NODES --
*
ADJUST    LDY   PNTR1       ADDRESS MASTER SUB-TREE
          BSR   POINTR      FIND PARENT NODE AND SAVE
          TFR   Y,X
          LDY   PNTR2       ADDRESS REDUNDANT SUB-TREE
          BSR   POINTR      FIND PARENT NODE
          LDD   0,Y
          LEAY  [6,Y]       IS PARENT LEFT OR RIGHT OFFSPRING
          LSRA              OF GRANDPARENT
          RORB
          BCS   ADJ1
          STX   2,Y         AS APPROPRIATE, CHANGE LEFT OR
          BRA   ADJ2        RIGHT POINTER OF GRANDPARENT
ADJ1      STX   4,Y         TO PARENT OF MASTER SUB-TREE
*
ADJ2      LDD   OFFSET      DIVIDE OFFSET BY FOUR TO
          LSRA              CALCULATE # OF LEAF NODES
          RORB              IN SUB-TREE
          LSRA
          RORB
          LDY   PNTR2       ADDRESS REDUNDANT SUB-TREE
*
```

```
*
*  -- CREATE BD TABLE FROM TRUTH TABLE DATA --
*
*    NODE  - INPUT VARIABLE IDENTIFIER
*    SUB   - INPUT SUBNODE IDENTIFIER
*    LASTX - LAST USED TABLE ADDRESS
*    TEMP  - TEMP STORAGE
*    X-REG - CURRENT ADDRESS POINTER
*    Y-REG - USED AS AN INDEX TO SEARCH SUB
*
BDCMPIL  DEC   INPUTS      INITIALIZE BUFFER LOCATIONS
         LDY   #SUB        AND STACK
         LDA   #$FF
         STA   NODE
*
BDCMP1   STA   0,Y+
         CMPY  #SUB+9
         BLE   BDCMP1
         LDY   #TRUTAB
         STY   TEMP
         LDD   #0
         PSHU  A,B
         LDX   TREE        INITIALIZE LASTX
         STX   LASTX
*
BDCMP2   LDY   #SUB        SELECT NEXT NODE AND
         INC   NODE        PUT NODE AND SUB ID
         LDA   NODE        IN TABLE
         INC   A,Y
         LDB   A,Y
         STD   0,X
*
         CMPA  INPUTS      CHECK IF THIS NODE IS AN
         BGT   BDCMP3      OUTPUT IF SO BRANCH TO
*                          OUTPT
         LEAX  8,X         PUT ADDRESS OF NEXT NODE
         STX   -6,X        IN 'LEFT' BRANCH COLUMN
         LEAX  -8,X
         PSHU  X           PUT PRESENT ADDRESS ON STACK
         LDD   LASTX       FOR LATER RETRIEVAL
         STD   6,X         STORE LASTX ADDRESS IN 'FROM'
         STX   LASTX       POINTER, STORE NEW LASTX AND
         LEAX  8,X         INCREMENT X-REG
         BRA   BDCMP2
*
BDCMP3   LDY   TEMP        IF NODE IS AN OUTPUT GET
         LDD   0,Y++       VALUE FROM VECTOR AND
         STD   4,X         PUT IN BD TABLE ALSO
         LDD   #$FFFF      PUT 'FROM' POINTER IN VECTOR
         STD   2,X
         LDD   LASTX
         STD   6,X
         STX   0,Y++
         STY   TEMP
*
```

```
ADJ3     COM   0,Y          COMPLEMENT LEAF NODE DATA
         LEAY  4,Y
         SUBD  #1
         BNE   ADJ3         CANCEL REMAINING LEAF NODES
         RTS
*
*  -- FIND PARENT OF A SUB-TREE --
*
POINTR   LEAY  [2,Y]        ADDRESS PARENT NODE AND
PTR1     LDA   0,Y          READ NODE ID
         CMPA  LVLCTR
         BLE   PTR2         STOP IF PARENT IS FOUND
         LEAY  [6,Y]        OTHERWISE CONTINUE
         BRA   PTR1
PTR2     RTS
*
         LIB   BDC-4B
```

```
*
*  --- GENERATE BD MACHINE CODE ---
*
MACHINE  LDD   #0          INITIALIZE BUFFERS
         PSHU  A,B
         STD   PGMADD
         STD   LASTY
         STD   PC1
         LDX   TREE
         LBSR  TRIM1
         TFR   X,Y
         LDX   MCODE
*
         LDA   OUTBNK      WRITE A BANK-SELECT INST
         ORA   #$E0
         LDB   #1
         STD   0,X
*
MC1      CMPY  #0          TEST END CONDITION
         BEQ   MC8
         LDD   PC1
         ADDD  #1
         STD   PC1
         LEAX  2,X
*
MC2      LDA   0,Y         TEST NODE TYPE
         CMPA  INPUTS
         BGT   MC6
         LBSR  TRIM        TRIM L&R BRANCHES
*
MC3      LDD   2,Y         TEST IF L-BRANCH PRUNED
         CMPD  LASTY
         BLS   MC4
         LBSR  BRANCH      WRITE A BRANCH-ON-1 INST
         LDA   0,X         IF L-BRANCH NOT PRUNED
         ORA   #$40
         STA   0,X
         LDD   4,Y
         PSHU  A,B,X       STACK R-BRANCH AND
         LDY   2,Y         FOLLOW L TO NEXT NODE
         BRA   MC1
*
MC4      BSR   BRANCH      WRITE A BRANCH-ON-0 INST
         LDD   4,Y         IF L-BRANCH PRUNED
         CMPD  LASTY
         BHI   MC5
         LDY   4,Y         WRITE A JUMP-TO-R INST
         LDB   7,Y         IF R-BRANCH ALSO PRUNED
         LBSR  JUMP
         LBSR  UNSTACK     FOLLOW STACK TO NEXT NODE
         BRA   MC1
*
MC5      LDY   4,Y         OTHERWISE FOLLOW R-BRANCH
         BRA   MC1
*
```

```
MC6     LDA   4,Y         TEST OUTPUT OR SWI
        CMPA  #$77
        BEQ   MC7
        LBSR  OUTPUT      WRITE AN OUTPUT INST
        LBSR  UNSTACK     AND FOLLOW STACK TO
        BRA   MC1         NEXT NODE
*
MC7     LBSR  SWI         WRITE AN SWI INST AND
        LBSR  UNSTACK     FOLLOW STACK TO NEXT NODE
        BRA   MC1
*
MC8     LDD   0,X
        CMPD  #$F0EE      OVERWRITE JUMP-TO-EOP
        BEQ   MC9         IF IT'S THE LAST INST
        LEAX  2,X
        LDD   PC1
        ADDD  #1
        STD   PC1
*
MC9     LDD   #$A000      WRITE AN EOP INST
        STD   0,X
        STX   LASTX
        LDD   PC1
        STD   PGMSIZ
        LDX   MCODE
*
MC10    LDD   0,X         REPLACE EE-CODES WITH
        CMPB  #$EE        PC OF EOP INST
        BNE   MC10A
        LDD   PC1
        LDA   0,X
MC10A   STD   0,X++
        CMPX  LASTX
        BLO   MC10
        LBSR  PRINT       OPTIONAL PRINTOUT ROUTINE
*
MC11    LBSR  DISC        SAVE BD PROGRAM ON DISC
        JSR   [PCRLF]     RETURN TO FLEX
        CLR   $DFE5
        LDA   #$0D
        STA   $CC11
        JMP   $CD03
*
*  -- CREATE A BRANCH INSTRUCTION --
*
BRANCH  LDB   0,Y         LOOK UP INPUT CHANNEL
        PSHS  U           NUMBER FROM INLIST
        LDU   #INLIST
        LDA   B,U
        PULS  U
        STA   CNTR        EXTRACT INPUT SELECTOR
        ANDA  #$0F        ADDRESS
        LSLA
        LSLA
        LSR   CNTR
```

```
          LSR    CNTR
          LSR    CNTR
          LSR    CNTR
          ORA    CNTR
          PSHS   Y,A,B          LINK TREE TO CODE AND
          LDD    PC1            WRITE A BRANCH-ON-CONDITION
          STD    6,Y            INST
          PULS   A,B
          LDY    2,Y            ADD BRANCH-ON-CONDITION
          LDB    7,Y            ADDRESS
          STD    0,X
          PULS   Y
          STY    LASTY
          RTS                   RETURN
*
*  -- CREATE AN OUTPUT INST --
*
OUTPUT    LDA    OUTPTS         LONG OR SHORT OUTPUT?
          CMPA   #4
          BLE    OUT1
          LDD    4,Y            WRITE A LONG OUTPUT INST
          ORA    #$80
          STD    0,X
          LDD    PC1            LINK TREE TO CODE
          STD    6,Y
          LDB    #$EE           WRITE A JUMP-TO-EOP INST
          BSR    JUMP
          BRA    OUT2
*
OUT1      LDA    5,Y            WRITE A SHORT OUTPUT INST
          ORA    #$C0
          LDB    #$EE
          STD    0,X
          LDD    PC1            LINK TREE TO CODE
          STD    6,Y
OUT2      STY    LASTY
          RTS                   RETURN
*
*  -- CREATE AN SWI INST --
*
SWI       LDD    PC1            LINK TREE TO CODE
          STD    6,Y
          ADDD   #1             WRITE AN SWI INST
          STD    PC1
          LDA    #$D0
          STD    0,X++
          LDA    #$A0           WRITE AN EOP INST WITH
          LDB    5,Y            SWI CODE
          STD    0,X
          LDB    #$EE           WRITE A JUMP INSTRUCTION
          BSR    JUMP           TO NEXT PGM
          STY    LASTY
          RTS                   RETURN
*
*  -- CREATE A JUMP INST --
```

```
*
JUMP     LEAX  2,X          WRITE A JUMP INSTRUCTION
         LDA   #$F0         TO PC SUPPLIED IN B-REG
         STD   0,X          BY CALLING ROUTINE
         LDD   PC1
         ADDD  #1
         STD   PC1
         RTS                RETURN
*
*  -- FOLLOW A POINTER TAKEN OFF STACK --
*
UNSTACK  LDY   0,U          TEST EOF CONDITION
         BEQ   UNST2
         PSHS  X
         PULU  A,B,X        UNSTACK A BRANCH AND
         CMPY  LASTY        TEST IF PRUNED
         BHI   UNST1
         LDB   7,Y          LINK BRANCH TO CODE
         STB   1,X          IF PRUNED AND FOLLOW
         PULS  X            STACK AGAIN
         BRA   UNSTACK
UNST1    LDD   PC1          OTHERWISE FOLLOW BRANCH
         ADDD  #1
         STB   1,X
         PULS  X
UNST2    RTS
*
*  -- TRIM COLLAPSED L&R BRANCHES --
*
TRIM     PSHS  X            TEST IF L-BRANCH COLLAPSED
         LDX   2,Y
         BSR   TRIM1
         STX   2,Y
         LDX   4,Y          TEST IF R-BRANCH COLLAPSED
         BSR   TRIM1
         STX   4,Y
         PULS  X
         BRA   TRIM2
TRIM1    LDA   0,X          DOES BRANCH LEAD TO INPUT?
         CMPA  INPUTS
         BGT   TRIM2
         LDD   2,X
         CMPD  4,X          IF SO, IS THAT INPUT COLLAPSED?
         BNE   TRIM2
         TFR   D,X          IF SO, TRIM THE NODE AND CHECK
         BRA   TRIM1        IT'S SIBLINGS
TRIM2    RTS
*
*  -- PRINT BD PROGRAM LISTING --
*
PRINT    LDX   #MSG17       PRINT LISTING OF PGM
         JSR   [PSTRNG]     STARTING WITH PGM NUMBER
         LDD   PGMNUM
         JSR   [OUTCH]
         TFR   B,A
```

```
        JSR   [OUTCH]
        LDX   #MSG17A
        JSR   [PSTRNG]
        LDY   MCODE
        LDD   #0
        STD   PC1
PRNT1   LEAX  0,Y
        JSR   OUT4H       PRINT mP ADDRESS OF DATA
        JSR   OUT2S
        LDX   PC1
        JSR   OUT4H       PRINT BD PC NUMBER
        LEAX  1,X
        STX   PC1
        JSR   OUT2S
        LDX   0,Y++
        JSR   OUT4H       PRINT BD OP-CODE
        BSR   ESCTRP      CHECK FOR 'ESC' HIT
        BVS   PRNT2
        JSR   [PCRLF]
        CMPY  LASTX
        BLS   PRNT1
PRNT2   RTS               RETURN
*
*  -- SAVE BD PROGRAM ON DISC --
*
DISC    LDX   #MSG6       PRINT DISC TRANSFER REQUEST
        JSR   [PSTRNG]
        JSR   [INCHE]     ENTER REPLY
        CMPA  #'N         IF NO RETURN TO FLEX
        BEQ   DISC2
        LDX   #LINBUF     OTHERWISE CALL SAVE SUBROUTINE
        LDY   #MSG7       FROM FLEX
DISC1   LDD   0,Y++
        STD   0,X++
        CMPA  #4          LOAD COMMAND STRING
        BNE   DISC1
        LDD   PGMNUM
        STD   $C089       ADD PGM NUMBER
        LDX   #$C090
        LDD   MCODE       ADD MEMORY BOUNDS
        BSR   HEXASC
        TFR   B,A
        BSR   HEXASC
        LEAX  1,X
        LDD   LASTX
        BSR   HEXASC
        TFR   B,A
        BSR   HEXASC
        LDD   #$C080      RESET LINE BUFFER POINTER
        STD   $CC14
        JSR   DOCMND      CALL DOS
DISC2   RTS
*
*  -- INPUT EDITABLE BINARY DATA --
*
```

```
INDATA  LDX   #LINBUF     FLEX LINE BUFFER
        JSR   [INLINE]
        LDX   #LINBUF
        JSR   [DECBIN]    CONVERT DATA TO BINARY
        RTS               RETURN
*
*  -- CONVERT BINARY TO ASCII --
*
BINASC  LDX   #LINBUF     FLEX LINE BUFFER
        JSR   [BINDEC]
        LDD   -2,X        LOAD LAST TWO CHARS.
        RTS               RETURN
*
*  -- CONVERT HEX TO ASCII --
*
HEXASC  PSHS  A           SAVE NUMBER
        LSRA
        LSRA
        LSRA
        LSRA
        BSR   HEX1        CONVERT FIRST PART
        PULS  A
        ANDA  #$F         CONVERT SECOND PART
HEX1    ADDA  #$30
        CMPA  #$39        CONVERT NUMBER TO ASCII
        BLE   HEX2
        ADDA  #7
HEX2    STA   0,X+        SAVE ASCII NUMBER
        RTS
*
*  -- INTERRUPT PRINTING WITH 'ESC' --
*
ESCTRP  LDA   ACIASR      KEY HIT?
        BITA  #1
        BEQ   ESC2
        LDA   ACIADR      READ CHARACTER
        ANDA  #$7F
        CMPA  #$1B
        BNE   ESC2        RETURN IF NOT 'ESC'
ESC1    LDA   ACIASR
        BITA  #1
        BEQ   ESC1        WAIT FOR ANOTHER HIT
        LDA   ACIADR
        ANDA  #$7F
        CMPA  #$1B
        BEQ   ESC2        RETURN IF 'ESC'
        CMPA  #$0D
        BNE   ESC1
        ORCC  #2          SET V-FLAG IF 'CR'
ESC2    RTS               RETURN
*
*  --- BDC-4 BUFFERS AND MESSAGES ---
*
MSG0    FCC   $A,'Output hardcopy required (Y*/N)? ',$4
MSG1    FCC   $A,'          BD PROGRAM COMPILER V4.1'
```

```
                FCC   $D,$A,$A
                FCC   '1  -  Combinatorial or sequential logic '
                FCC   '(TRUTH TABLE)',$D,$A
                FCC   '2  -  PID control (TRANSFER FUNCTION)'
                FCC   $D,$A,$A
                FCC   'Enter option code - ',$4
MSG2            FCC   'Enter number of inputs (up to 8) - ',$4
MSG2A           FCC   'Enter number of output channels '
                FCC   '(up to 12) - ',$4
MSG2B           FCC   'Enter BD program number - ',$04
MSG2C           FCC   'Enter program class - ',$4
MSG2D           FCC   'Enter channel assignment for input #',$4
MSG2E           FCC   'Enter output bank assignment - ',$4
MSG3            FCC   'Enter ',$04
MSG4            FCC   ' output words in HEX ',$04
MSG5            FCC   '    ** ILLEGAL DATA, RE-ENTER **',$D,$4
MSG6            FCC   'Save new program on disc (Y*/N)? ',$4
MSG7            FCC   'SAVE,1.BDxx.BDP,xxxx,xxxx',$D,$4
MSG17           FCC   $A,'BD MACHINE CODE - PROGRAM NUMBER ',$04
MSG17A          FCC   'ADDR   PC   CODE',$0A,$0D
                FCC   '----  ----  ----',$D,$A,$4
MSG20           FCC   'The transfer function is in the form '
                FCC   ' (A/S+B+C.S) ',$D,$A
                FCC   'Enter constant A-',$4
MSG21B          FCC   'Enter constant B-',$4
MSG21C          FCC   'Enter constant C-',$4
*
TRUTAB          EQU   *
*
```