INTELLIGENT REFLEXIVE INTERFACES AND THEIR APPLICATIONS

by

Meir H. Levi

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

August 1985

A B S T R A C T

INTELLIGENT REFLEXIVE INTERFACES AND THEIR APPLICATIONS

Control tasks in an industrial process can be partitioned into routine, reflexive tasks, and smart computational tasks. An Intelligent Reflexive Interface (IRI) unit based upon Binary Decision processing of digital data was designed and interfaced to a MC6809 based microcomputer. Advantages of the Binary Decision method as compared to the conventional Boolean technique are analyzed. Both the hardware concepts and BD control algorithms are presented.

Two applications of IRI controllers are presented:

1. The control of a batch weighing system.
2. A pre-processor to scan an array of variable reference sensors.

IRI applications in industrial process control demonstrate how control may be enhanced by the reflexive-vs-smart task separation. A relatively simple microprocessor takes on many of the advantages of a multiprocessor system when augmented by Binary Decision based IRI modules. Thus it can be easily made to control a fairly complicated industrial process.

R E S U M E

LES INTERFACES REFLEXIFS INTELLIGENTS ET LEURS APPLICATIONS

Les tâches de contrôle de processus industriels peuvent être séparées en routine, tâches réflexives et tâches d'arithmétique intelligentes. Une unité d'Interface Réflexif Intelligent (IRI) basée sur le traitement par décision binaire de données digitales fut conçue et interfacée à un micro ordinateur à base MC6809. Les avantages de la méthode par décision binaire par rapport à la technique Booléenne conventionelle sont analysés. Les concepts d'implémentation matérielle et d'algorithmes de contrôle BD sont tous deux présentés.

Deux applications des contrôleurs IRI sont présentées:

1. Le contrôle d'un système de pesage de fournée.
2. Un pré-processeur qui balaie une matrice de capteurs à référence variable.

Les applications de l'IRI aux processus de contrôle industr els démontrent comment le contrôle peut être amélioré par la séparation des tâches réflexives contre intelligentes. Un microprocesseur relativement simple acquiert plusieurs des avantages d'un système à processeurs multiples lorsqu'on lui adjoint un module d'IRI basé sur les décisions binaires. Donc on peut facilement l'utiliser pour contrôler des processus industriels relativement compliqués.

ACKNOWLEDGEMENTS

The research presented in this thesis was carried out under the supervision of Dr. Paul J. Zsombor-Murray and Mr. Louis Vroomen. The author wishes to thank both for their excellent advice, persistent guidance and encouragement.

Special thanks are due to Mr. Robert D. Hudson and Mr. Artun Kucuk who, in addition to being good friends, carried out complementary investigations which contributed materially to the progress of this project. The valuable assistance of all DATAC Laboratory personnel is gratefully acknowledged.

The research was supported by grant A-4219 of the Natural Sciences and Engineering Research Council of Canada.

TABLE OF CONTENTS

PAGE

CHAP. 4: BD BASED MACHINES

LIST OF FIGURES AND TABLES PAGE

LIST OF ABBREVIATIONS

A/D	Analog to Digital
ADC	Analog to Digital Convertor
ACIA	Asynchronous Communication Interface Adaptor
ALE	Auto Load Enable
A/M	Auto/Manual
AML	Auto Manual Logic
BD	Binary Decision
BGRNT	Bus Grant
BNF	Backus Normal Form
BRA	Branch
BREQ	Bus Request
BSL	Board Select Logic
BWL	Byte to Word Logic
CIL	Clock and Interrupt Logic
CLK	Clock
COP	Cut-off Point
CPOS	Canonical Product of Sums
CPU	Central Processing Unit
CR	Control Register
CS	Chip Select
CSOP	Canonical Sum of Products
D/A	Digital to Analog
DAC	Digital to Analog Convertor
DBL	Data Buffer Logic

DCL	Data Channel Low
DCH	Data Channel High
DDC	Direct Digital Control
DDL	Data Direction Logic
DE	Data Enable
DEL	DMA Enable Latch
DLU	Decision Logic Unit
DM	Dedicated Memory
DMA	Direct Memory Access
DMAC	DMA Controller
EOP	End of Program
EOC	End of Conversion
EPI	Essential Prime Implicant
EQU	Equate
FF	Flip Flop
IB	Input Bank
IBS	IB Select
INT	Interrupt
I/O	Input/Output
IRI	Intelligent Reflexive Interface
IRIP	IRI Processor
IRQ	Interrupt Request
ISP	Instruction Set Processor
LED	Light Emitting Diode
LIR	Line Image Register
LS	Least Significant
LSB	LS Byte

N-MOS	N-channel Metal-Oxide Semiconductor
mP	Microprocessor
MS	Most Significant
MSB	MS Byte
OB	Output Bank
OBS	OB Select
OE	Output Enable
OPL	Output Long (Instruction)
OPS	Output Short (Instruction)
OL	Output Long (Control Line)
OS	Output Short (Control Line)
PC	Program Counter
PI	Prime Implicant
PIA	Parallel Interface Adaptor
PLA	Programmable Logic Array
PLC	Programmable Logic Controller
RAM	Random Access Memory
R/W	Read/Write
SDM	Shared/Dedicated Memory
SEL	Select
SM	Shared Memory
SOC	Start of Conversion
SS	Single Step
VLSI	Very Large Scale Integration
VMA	Valid Memory Address
XOR	Exclusive OR

CHAPTER 1: INTRODUCTION

1.1 The Computer Role in Process Control

The application of computers in process control has gradually increased in popularity since the first microprocessors were introduced in the early seventies. In the control loop, process computers replace conventional control elements such as hard-wired switching circuits, pneumatic and hydraulic controllers. Software implemented logic control algorithms can detect process limits and transmit on/off control decisions to process actuators. The development of Digital to Analog and Analog to Digital convertors permitted analog control to be similarly implemented in digital computers. The process control computer is different from other computers because it can control a real-time process interfaced to it via sensors and process actuators. To do this it executes programs containing various control algorithms.

Computer process control can be viewed on four hierarchical levels, Fig. 1.1:

1. The Direct Digital Level: The computer senses and monitors the state of the process via transducers, executes the control algorithms and produces the necessary outputs to drive the actuators. This is the lowest control level.

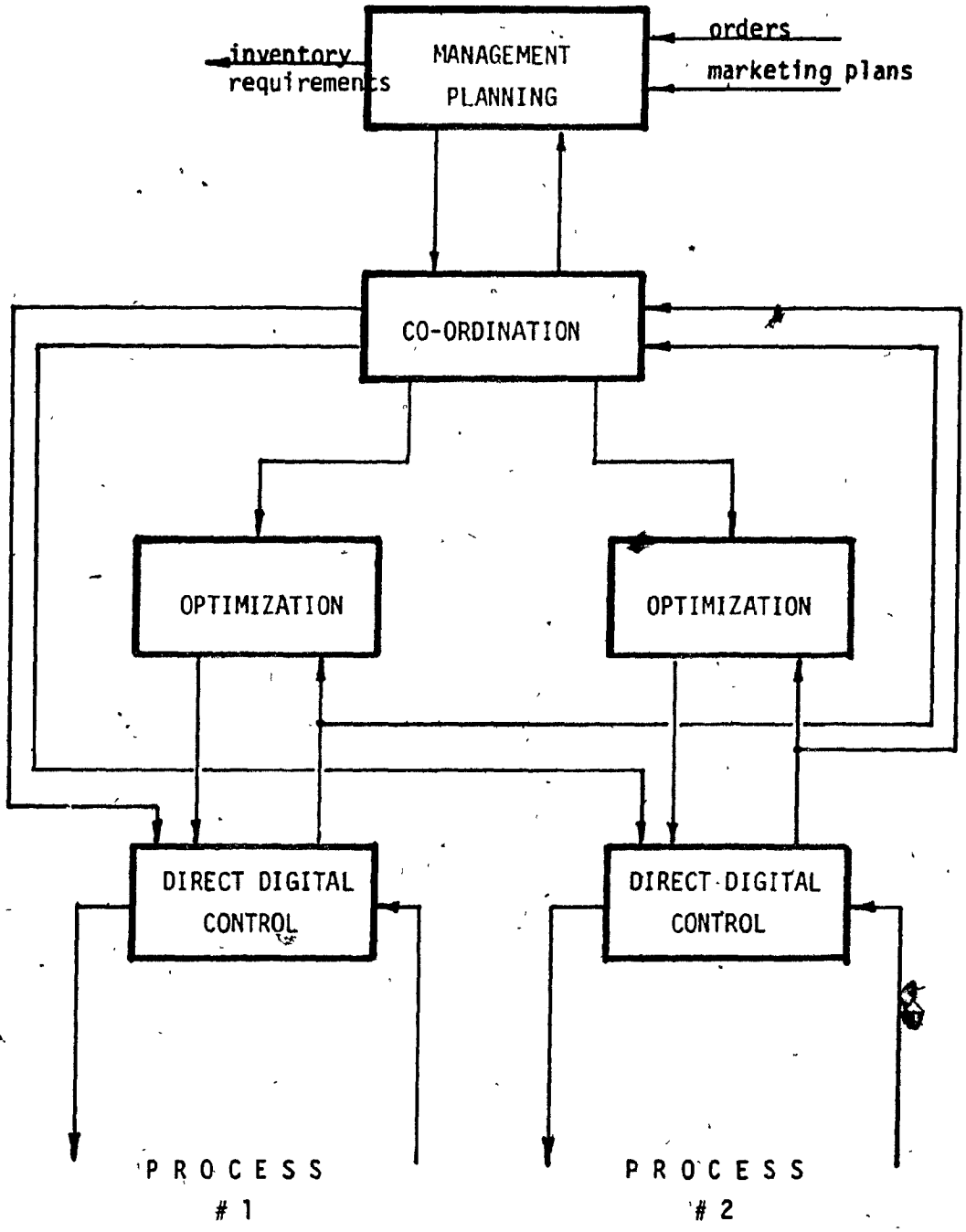


Fig. 1.1: Process Control Hierarchical Levels.

2. The Supervisory Control Level: On this level, control algorithms are created and modified to adapt to process conditions. Supervisory control can be assisted by a simulated model of the controlled process and an optimization program to provide the optimum control algorithms and set-points to the first control level. The adjustments are performed in real-time.

3. The Co-ordination level: This level exists in distributed control systems where several processes are being controlled simultaneously. Communication between the processes and the necessary coordination is performed at this level.

4. The Management Level: At this top level statistical information gathered from all processes, is applied to non-direct-control considerations such as market planning, establishing raw material stock pile levels, analysing cost center profitabilities, etc. All these are executed off-line, i.e., the computer does not have to be physically connected to the process and need not meet real-time constraints.

The desire to achieve a rational partitioning of process control tasks motivated this work. The idea to separate the direct control of the process from other tasks is implemented using the primitive, fast and efficient Binary Decision intelligence.

1.2 Direct Digital Control: Boolean vs. Binary Decision Alternative

Shannon [SH38] showed in 1938 how to apply Boolean algebra in the design of switching circuits. Since then Boolean algebra has been systematically enhanced and adopted as the most popular logic design tool using manipulation and optimization procedures such as Karnaugh maps and DeMorgan's theorems. Control algorithms at the DDC level are constructed using mostly Boolean algebra methods. Hardwired circuitry used to implement Boolean functions has been replaced by programmable automata. Switching control functions are represented in the control computer as Boolean sum-of-products which are evaluated in a stepwise sequential manner. Due to its algebraic nature, the Boolean method gained recognition as the sole logic design and implementation tool but it has some limitations. Speed of execution is limited because the number of program steps increases exponentially as the number of binary input variables or program inputs. A second deficiency of the Boolean method lies in the fact that Boolean functions have single-bit binary results or outputs.

In 1959 Lee [LE59] proposed binary-decision (BD) programs as an alternative to Boolean representation of switching circuits. One of the qualities of the BD technique is the computation speed. Lee demonstrated that a switching function implemented as a BD program is evaluated in a number of steps that is equal to or less than the number of binary input

variables.

Although the BD alternative to Boolean algebra was shown to be more efficient in several ways, it has not been widely accepted by designers of electronic logic. This is because BD programs are non-algebraic.

Extensive theoretical BD research has been done but most of it deals with information theory, image processing, networks and connectivity [MO82]. Several researchers followed Lee's concepts. Akers [AK78] presented BD programs in a graphical form called BD diagrams. He compared the evaluation of digital functions by Boolean algebra with their BD diagram representation. Boute, in 1976 [BO76], formulated the first known design of BD controller hardware.

Potential superiority as regards speed of execution, and lower hardware cost, of BD controllers over their Boolean counterparts provided the incentive for research [LN79,ZM79, HU81,LM82a,ZM83] in the following BD related areas, carried out in the DATAC computer laboratory at McGill University.

1. Development of BD optimization tools,
2. Development of BD Controllers,
3. Applications of BD controllers, and
4. Applied BD theory.

1.3 The Present Work

This thesis is a part of DATAC's applied research program to design a BD based control module which can handle direct digital control tasks efficiently and autonomously. The module can be programmed for a variety of control applications. It is intended to be a component in the process I/O level of a distributed control system. BD based control algorithms enable fast response to process changes. The BD controller module is supervised by a conventional control computer which can dynamically modify BD control programs. This control module is meant to handle routine, repetitive control tasks which are called 'reflexive' tasks. Although reflexive tasks do not require complex control algorithms, their frequent occurrence makes heavy demands on a process computer. The module, which has been designed to alleviate this large overhead is called an Intelligent Reflexive Interface or IRI.

It is obvious that a microprocessor based control system, with one or more IRI modules to autonomously handle direct reflexive digital control tasks, is able to perform the higher process control level tasks better. Thus IRIs enable a relatively low-cost microprocessor to replace more expensive control computers. The main processor supervises the operation of the IRI modules by starting BD control programs and initializing their parameters and by supplying new control algorithms to IRIs, in response to any process changes which require them.

A review of Boolean algebra, related to the process control applications, is given in Chapter 2. This will help us to compare BD vs. Boolean implementation of process controllers.

BD logic theory is presented in the following chapter. Advantages and limitations of the BD methods when applied to the process control environment are described along with examples. BD optimization techniques are briefly described. It will be shown that the BD technique is more efficient than its Boolean counterpart. It is especially suitable to control processes having a small number of input and/or output signals and requiring fast responses.

Chapter 4 is a thorough description of BD automata starting with the first DATAC prototype which was developed from Boute's [B076] design. This was our first practical tool to investigate theoretical and applied aspects of the BD methods. A hybrid system consisting of a BD machine and an 8-bit microprocessor is described next. This system, which was built as part of this work, enabled further research in different BD related fields one of which is the partitioned IRI based system.

The task partitioning concept, the design of the IRI module and the IRI based control system are described in Chapter 5. Several design schemes are presented, highlighting key design aspects of an IRI based controller.

Two applications of an IRI based control systems are given in Chapter 6. The first is an industrial batch weighing system controller. It will be shown that a conventional, low-cost microprocessor system augmented with IRI modules can control a fairly complex process which would otherwise need several microprocessors or a minicomputer system. The second application is an IRI controller serving as a pre-processor to increase the throughput of a variable-reference matrix scanner.

Chapter 7 contains conclusions and several recommendations for further work. Three appendices provide information on the MC6809 based development system (Appendix A), listings of application software (Appendix B), and a description of the BNF and ISP language definitions used to formally describe BD automata and software (Appendix C).

CHAPTER 2: LOGIC THEORY REVIEW

2.1 Introduction.

This chapter describes briefly the theory of combinatorial and sequential logic circuits, as it applies to industrial process control. This provides the necessary background to compare Binary Decision based process controllers with their Boolean counterparts.

A complex process control procedure consists of a digital part, i.e., on/off decisions and an analog part. The digital part performs logical operations upon logic parameters. A logic parameter is a binary variable which may have two distinct values: e.g., on/off, 1/0, true or false. The state of an induction motor is an example of a logic parameter, S. S=1 when the motor is running, and S=0 when it is not. S may depend on the state of other logic parameters such as a pressure limit or a stop flag. Notice that air pressure is an analog parameter, but it may be treated logically. Logical operations describe the relations among circuit parameters.

In 1938 Shannon [SH38] applied the rules of Boolean algebra to the design of logic circuits. Since then, extensive Boolean methods have evolved, enabling logic design problems to be treated systematically.

Logic circuits may be divided into two categories: combinatorial and sequential. Combinatorial circuit outputs are Boolean functions which depend only upon the present inputs. Sequential circuit outputs depend upon the previous

states of the circuit as well as the current inputs.

2.2 Combinatorial Logic

Logic operations relate the input signals of a digital circuit to its output. There are six fundamental logic operators, which describe these operations. These can be implemented with logic gates. Regardless of complexity, any digital circuit may be implemented using these six basic gates. The operation of a logic gate may be represented by a Boolean expression or a truth table. A truth table contains all possible circuit input combinations and their respective outputs. A definition of each logic gate follows. The symbolic representation, Boolean expression, and truth table, are given in Fig. 2.1:

OR gate, Fig. 2.1a: The output of an OR gate will be 1 if and only if at least one of the inputs is 1. The operation is defined as the logic summation of its inputs and denoted by the + sign.

AND gate, Fig. 2.1b: The output of an AND gate will be 1 if and only if all inputs are 1. The operation is defined as the logic multiplication of its inputs and denoted by the · sign.

NOT gate, Fig. 2.1c: The output of a NOT gate is the complement of its input. If the input a=0 then the output f=1 and if a=1 then f=0. The complement of a logic parameter is denoted by placing a bar over the parameter name.

NOR and NAND gates, Fig. 2.1d & 2.1e: The operation of these gates the complement of the OR and AND gates respectively.

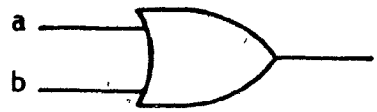
EXCLUSIVE-OR (XOR), Fig. 2.1f: The output of the XOR gate is 0 if both inputs are identical and 1 if they differ. To generalize for any number of inputs: whenever odd number of inputs are 1 the XOR output will be 1. The XOR operation is denoted by the $\oplus$ sign.

An example using some of the above basic gates is given in Fig. 2.2. A digital function F1(a,b,c,d), Equation 2.1, is implemented using a NOT gate, four AND gates, and an OR gate.

$$F1(a,b,c,d) = a \cdot b + a \cdot c + b \cdot d + \bar{a} \cdot c \qquad (2.1)$$

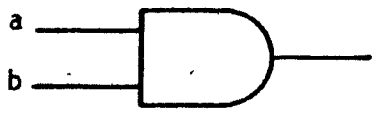
Logic implementations such as those in Fig. 2.2. may be simplified using Boolean Algebra rules. The design guidelines for optimal implementation are minimization of cost and propagation delay. A good criterion for cost estimation is the number of gate leads in a circuit. A 2-input gate will have three leads, two for the inputs and one for the output. Thus the above implementation of F1 requires 19 gate leads. The second design guideline considers the time delay required for the signals to pass through a gate, important in complex circuits to eliminate race problems and to meet critical timing requirements.

a)



a	b	f
0	0	0
0	1	1
1	0	1
1	1	1

b)



a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

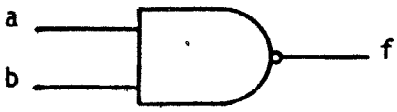
c)

a	f
0	1
1	0

d)

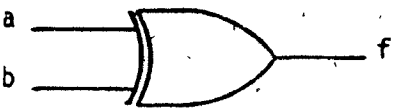
a	b	f
0	0	1
0	1	0
1	0	0
1	1	0

e)



a	b	f
0	0	1
0	1	1
1	0	1
1	1	0

f)



a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 2.1: Fundamental Logic Gates; Symbolic Representation, Boolean Expression and Truth Table.

a) OR b) AND c) NOT d) NOR e) NAND f) XOR.

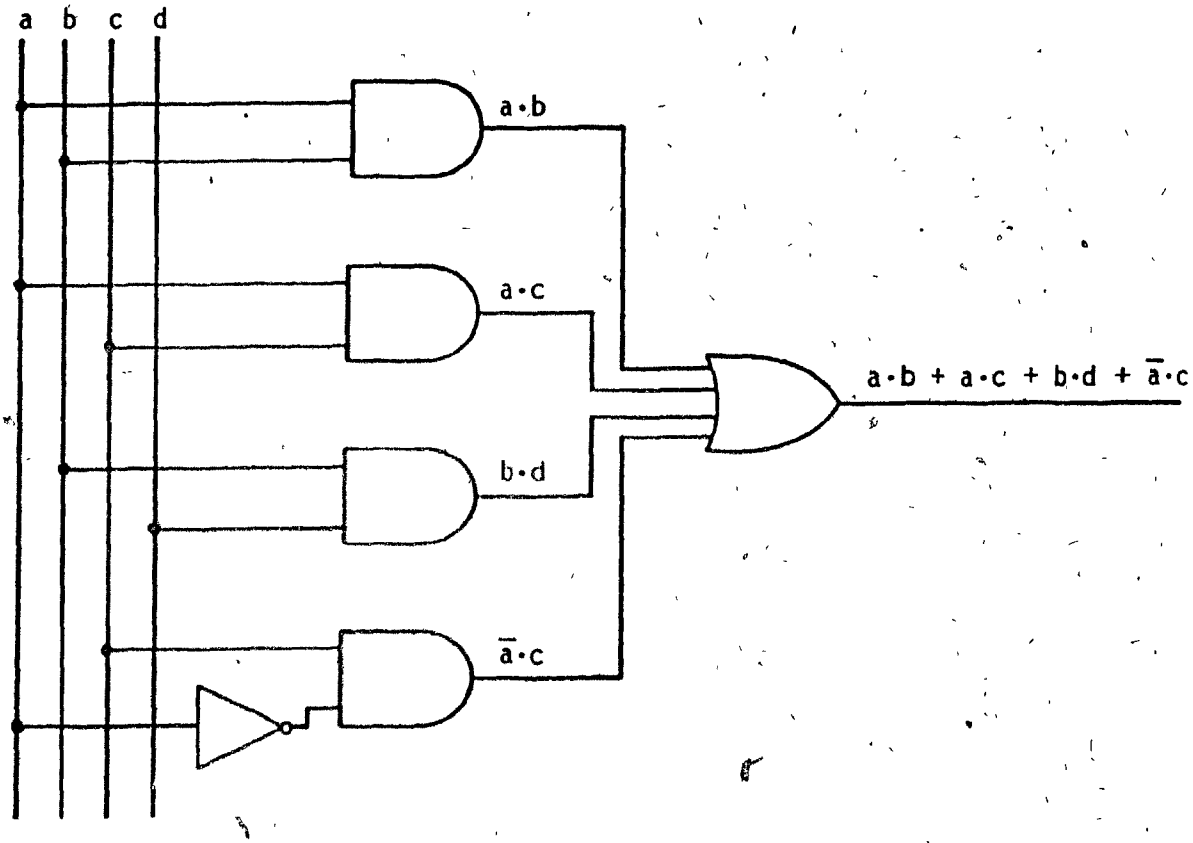


Fig. 2.2: Gate Logic Implementation of F1(a,b,c,d).

2.3 Boolean Algebra Review

Boolean Algebra, also known as the mathematics of logic operations, is the essential tool available to optimize logic circuits. Boolean variables are defined as: $X_i \in \{0,1\}$. Boolean operators are OR, AND and NOT, described in section 2.2. Boolean functions can be reduced by the identities and laws described in this section.

2.3.1 Boolean Algebra Identities

The identities in Equations 2.2a-h reflect the basic postulates on which Boolean algebra is founded.

$$x \cdot 0 = 0 \qquad (2.2a)$$

$$x \cdot 1 = x \qquad (2.2b)$$

$$x + 0 = x \qquad (2.2c)$$

$$x + \bar{1} = 1 \qquad (2.2d)$$

$$x \cdot x = x \qquad (2.2e)$$

$$x + x = x \qquad (2.2f)$$

$$x \cdot \bar{x} = 0 \qquad (2.2g)$$

$$x + \bar{x} = 1 \qquad (2.2h)$$

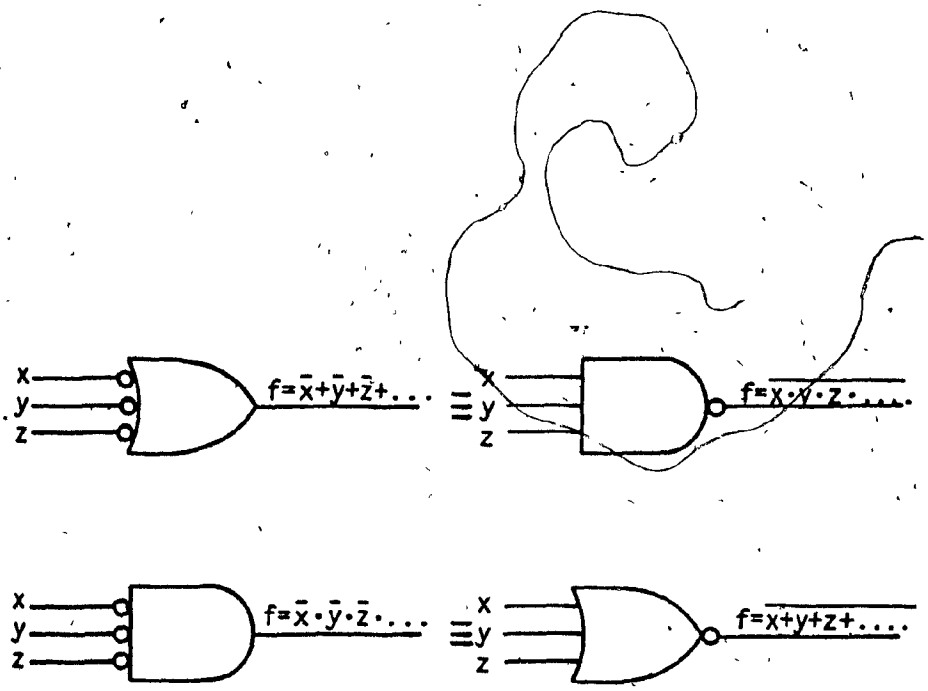
2.3.2 Boolean Algebra Laws

Boolean expressions may be manipulated algebraically using the following laws:

$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	associativity	(2.3a)
$(x + y) + z = x + (y + z)$	associativity	(2.3b)
$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	distributivity	(2.3c)
$x + (y \cdot z) = (x + y) \cdot (x + z)$	distributivity	(2.3d)
$x \cdot y = y \cdot x$	commutativity	(2.3e)
$x + y = y + x$	commutativity	(2.3f)
$x + x \cdot y = x$	absorption	(2.3g)
$x \cdot (x + y) = x$	absorption	(2.3h)
$x + \bar{x} \cdot y = x + y$	absorption	(2.3i)
$x \cdot (\bar{x} + y) = x \cdot y$	absorption	(2.3j)

2.3.3 DeMorgan Theorems

The DeMorgan theorems relate the OR and AND operations using pre- and post-inversions, Fig. 2.3.



$\bar{x}+\bar{y}+\bar{z}+\ldots = \overline{x \cdot y \cdot z \cdot \ldots}$

$\bar{x} \cdot \bar{y} \cdot \bar{z} \cdot \ldots = \overline{x+y+z+\ldots}$

Fig. 2.3: Pre- and Post-inversions;
DeMorgan Theorem Implementation.

2.4 Minimization of Boolean Functions

2.4.1 Boolean Functions: Canonical Forms

Logic functions may take different, equivalent forms. A minimized form is desirable. In order to define minimization procedures one must begin with the most generalized forms, the canonical forms. Two types are considered: canonical sum of products (CSOP) and canonical product of sums (CPOS).

Consider a 3-variable logic function, F(a,b,c). There exist eight combinations of the input variables. The outputs are given by a truth table Fig. 2.4a. Each combination with an output of 1 may be expressed as the product of the appropriate input variables or their complements. Any such product is called a minterm, Fig. 2.4b. ORing all minterms results in the CSOP form of F(a,b,c), Equation 2.3.

Conversely, each combination whose output is 0 may be expressed as the complement of the sum of the appropriate input variables or their complements. Any such sum is called a maxterm, Fig. 2.4c. ANDing all maxterms results in the second canonical form of F(a,b,c), the CPOS form, Equation 2.4.

$$F(a,b,c) = \bar{a}\cdot\bar{b}\cdot\bar{c} + \bar{a}\cdot\bar{b}\cdot c + \bar{a}\cdot b\cdot c + a\cdot b\cdot c \qquad (2.3)$$

$$F(a,b,c) = (a+\bar{b}+c)\cdot(\bar{a}+b+c)\cdot(\bar{a}+b+\bar{c})\cdot(\bar{a}+\bar{b}+c) \qquad (2.4)$$

a	b	c	f	minterms	maxterms
0	0	0	1	$\bar{a}\cdot\bar{b}\cdot\bar{c}$	
0	0	1	1	$\bar{a}\cdot\bar{b}\cdot c$	
0	1	0	0		$a+\bar{b}+c$
0	1	1	1	$\bar{a}\cdot b\cdot c$	
1	0	0	0		$\bar{a}+b+c$
1	0	1	0		$a+b+\bar{c}$
1	1	0	0		$\bar{a}+\bar{b}+c$
1	1	1	1	$a\cdot b\cdot c$	

a) b) c)

Fig. 2.4: Representation of a Logic Function F(a,b,c);
a) Truth Table. b) Minterms. c) Maxterms.

2.4.2 Karnaugh Maps

Karnaugh maps are semi-graphical representations of truth tables used to minimize logic functions. Each square represents a minterm or a maxterm and is assigned a 1 or a 0 respectively, Fig. 2.5. E.g., a 4 input function, F2(a,b,c,d), Equation 2.5, is mapped into a 4x4 array of squares.

$$F2(a,b,c,d)=abc\bar{d}+ab\bar{c}d+ab\bar{c}\bar{d}+a\bar{b}cd+a\bar{b}c\bar{d}+\bar{a}bcd+\bar{a}bc\bar{d}+\bar{a}b\bar{c}d+\bar{a}b\bar{c}\bar{d}+\bar{a}\bar{b}cd \quad (2.5)$$

Square addresses may be assigned arbitrarily but adjacent squares must be numbered using a Gray code in both the rows and columns. Karnaugh maps wrap around two ways: top to bottom and right to left, resulting in four adjacent squares for each square.

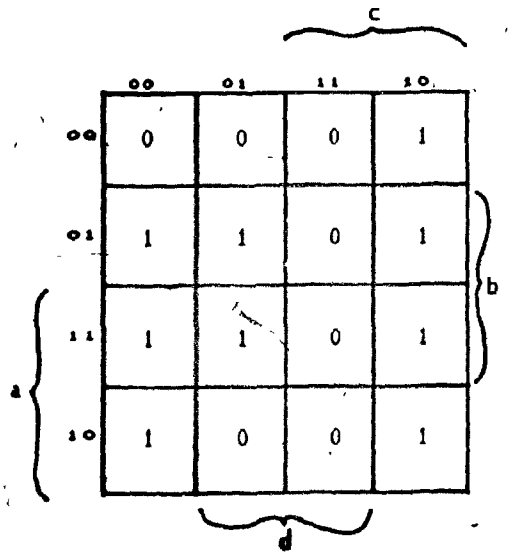


Fig. 2.5: F2(a,b,c,d); Karnaugh Map Representation.

The minimization procedure is performed by grouping together adjacent squares with identical output into cells (or implicants). Because adjacent squares differ only in one variable, asserted in one and negated in the other, grouping them together eliminates the non-identical variable. Similarily, adjacent cells may be grouped into larger cells eliminating additional variables. The grouping is performed on

either all the '0' squares or all the '1' squares. A 'don't care' square may be included in any grouping since the value of that minterm is irrelevant to the value of the function. Every square containing the output chosen for the grouping, must appear at least in one of the cells but may appear in more than one cell.

A prime implicant (PI) is a maximal cell that cannot be a part of a larger cell. An essential prime implicant (EPI) is a prime implicant that contains a square which does not belong to another prime implicant. To obtain an optimal simplification of the logic function, one first has to search for essential prime implicants, and to group the remaining squares in prime implicants. A prime implicant is preferable even if it covers an already covered square because its address is shorter. Take as an example the Karnaugh map of Fig. 2.6 which describes F3(a,b,c,d), Equation 2.6.

$$F3(a,b,c,d) = ac\bar{d}+abcd+ab\bar{c}+\bar{a}\bar{c}d+a\bar{b}d+\bar{a}\bar{b}c \qquad (2.6)$$

Each one of the designated groups is an essential prime implicant due to the starred squares. Therefore the optimal solution is:

$$F3(a,b,c,d) = ab + \bar{c}d + \bar{b}c \qquad (2.7)$$

Karnaugh maps become impractical when the number of variables exceeds six.

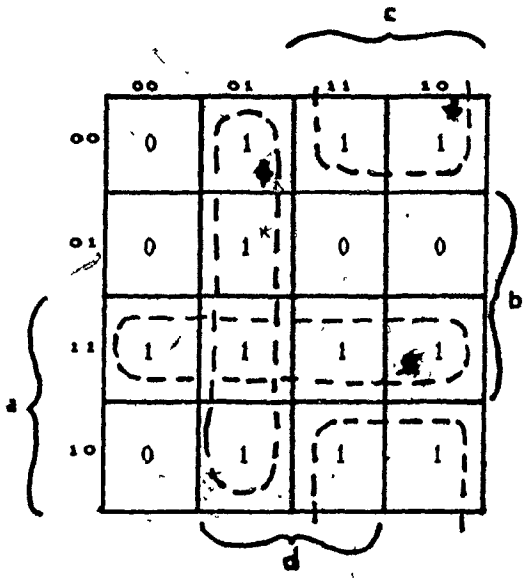


Fig. 2.6: Optimal Solution with Karnaugh Maps.

2.4.3 Quine McCluskey (QM) Method

The QM method determines EPIs and PIs algorithmically. This is useful when the number of parameters is large. The method can be implemented as a computer algorithm.

The tabular procedure will be demonstrated by an example taken from [CH82]. Consider the canonical sum of minterms form of $F(w,x,y,z)$:

$$F(w,x,y,z)=\bar{w}\bar{x}\bar{y}\bar{z}+\bar{w}\bar{x}y\bar{z}+\bar{w}x\bar{y}\bar{z}+w\bar{x}\bar{y}\bar{z}+\bar{w}xyz+wx\bar{y}z+wxy\bar{z}+wxyz \qquad (2.8)$$

Step 1: The minterms in their binary form (i.e., 0 represents a negated variable and 1 an asserted one) are grouped according to the number of 1's they contain, Table 2-1.

minterm	binary form	number of 1's
$\bar{w}\ \bar{x}\ \bar{y}\ \bar{z}$	0000	0
$\bar{w}\ \bar{x}\ y\ \bar{z}$	0010	1
$\bar{w}\ x\ y\ \bar{z}$	0110	2
$w\ \bar{x}\ y\ \bar{z}$	1010	2
$\bar{w}\ x\ y\ z$	0111	3
$w\ x\ \bar{y}\ z$	1101	3
$w\ x\ y\ \bar{z}$	1110	3
$w\ x\ y\ z$	1111	4

Table 2-1: Minterms Classification by Number of 1's.

Step 2: Each member from each group is compared with each member from the next group. Whenever adjacent minterms are located, i.e., containing only one non-identical bit, these minterms are marked (*), meaning they are not PIs, and a new implicant containing these minterms is entered in the first reduction table. The non-identical bit is replaced by a - (dash), meaning that both 0 and 1 are possible.

Step 3: The same procedure is performed on the first reduction table, thus creating a second reduction table. Notice that the dashes should be identical, for instance 0-10 + 1-10 = --10 for a successful merge to take place. The process is repeated until no adjacent minterms can be found.

original form		1st reduction		2nd reduction
wxyz		wxyz		wxyz
0000	*	00-0		
0010	*	0-10	*	--10
0110	*	-010	*	-11-
1010	*	011-	*	
0111	*	-110	*	
1101	*	1-10	*	
1110	*	-111	*	
1111	*	11-1		
		111-	*	

Table 2-2. Prime Implicants for F(w,x,y,z)

The prime implicants are the unchecked terms (from all tables), thus the minimized function is:

$$F(w,x,y,z) = \bar{w}\bar{x}\bar{z} + wxz + y\bar{z} + xy \qquad (2.9)$$

Step 4: Further minimization of the function can be done by performing dominant Row/Column operations on the PIs found in steps 1-3. These operations cause the elimination of identical subsets of the PIs [CH82].

2.5 Sequential Logic and Finite State Automata

A sequential circuit consists of a combinatorial part and memory elements, Fig. 2.7. It is often described by an abstract model called a finite state machine. The operation of this machine may be described as a sequence of events that occur at discrete instances t_i, i=1.....n. At each moment t_i the machine responds to some input signals I(t) as well as to some historical input and output signals J(t), producing output signals F(t). The number of possible past histories may grow to be infinite, yet, they usually affect the present output F(t) in a finite number of ways only. Thus, past inputs are classified by examining the way in which they influence the machine response. These classes are called the internal states of the machine or just states.

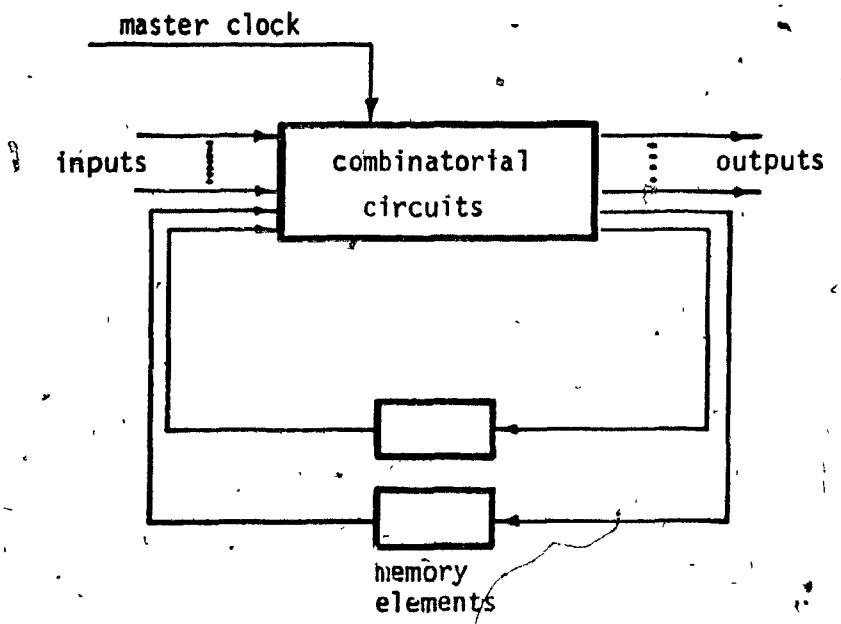


Fig. 2.7: Sequential Logic Machine; Typical Block Diagram.

The states and transitions that a finite state machine goes through during its course of operation are well described in the state diagram. The diagram consists of nodes equivalent to the machine states, and arcs connecting the nodes, equivalent to the transitions between the states. Each arc contains information with regard to the current inputs and the produced outputs pertaining to that transition.

Similar to a truth table description of a combinatorial circuit operation, a transition table describes the operation of a sequential machine by listing the present inputs and the present machine state versus the present resulting output and the next machine state.

As a simple sequential circuit example consider the differential encoder in Fig. 2.8. The encoder examines a serial bit stream and produces a "1" output whenever two consecutive bits are different and "0" otherwise. The history of past input values, which establishes the encoder's present states, corresponds to the two possible values of the previous input bit. Thus the encoder is a 2-state machine where the two states are defined as S_1 and S_0 corresponding to the previous bit values 1 and 0 respectively. If the encoder is in state 0, i.e., previous input was 0, and the present input is 1 then the output will be 1 and the encoder will go to state 1. All possible state transitions and the associated inputs and outputs are described in the transition table, Fig. 2.9a and in the state diagram, Fig. 2.9b.

Sequential circuits may be synchronous or asynchronous. Synchronization is enabled by a master clock. The sequential operations occur only when allowed by the proper pulse edges.

In an asynchronous circuit operations occur at their own rate responding to state transitions and input signals. Because they are not bound to a clock speed, these circuits may operate faster than synchronous circuits but they are more prone to race hazards.

Digital circuits can either be distributed in space or distributed in time. Comparator type A/D convertors generate a binary equivalent of an analog voltage by comparing the input to a reference voltage. The SAR (successive approximation register) type uses one comparator and a controlled D/A. It is distributed in time. The parallel type uses 2^{n-1} comparators

to generate an n-bit result in one cycle. It is distributed in space.

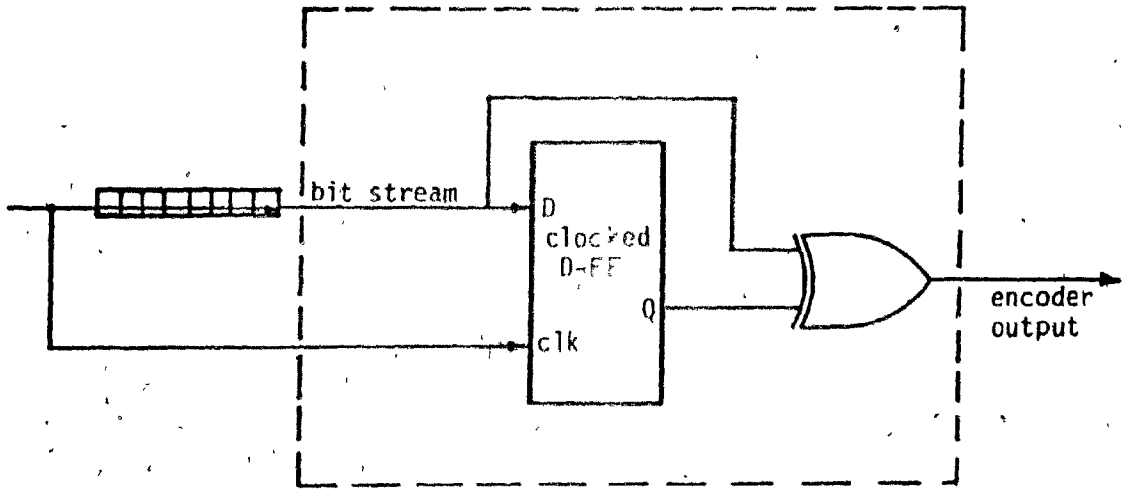


Fig. 2.8: Differential Encoder, Logic Scheme.

a)

present state	next state	output
0	0	0
0	1	1
1	0	1
1	1	0

b)

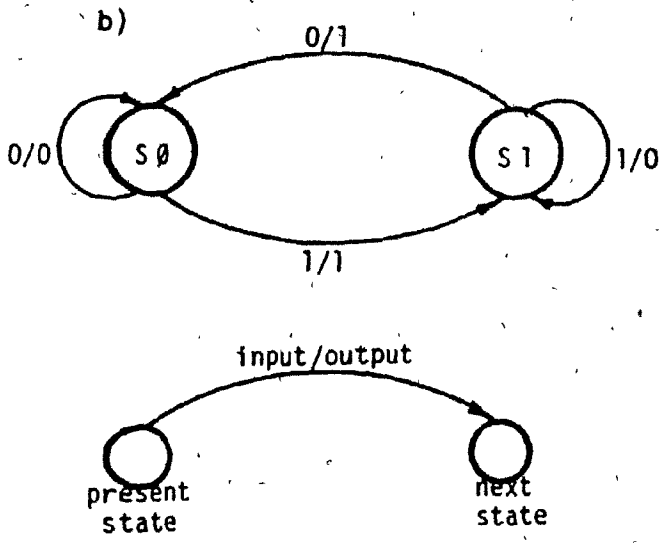


Fig. 2.9: Differential Encoder, a) Transition Table.
b) State Diagram.

Memory elements. The fundamental digital memory element, the Flip Flop (FF), features two basic properties:

1) It exists in one out of two distinct states, i.e., the FF is a 2-state automaton, and
2) it has one or more input connections to force a state transition.

Digital FFs may be built from the basic six gates. The simplest FF is the latch FF which is constructed with two NAND or two NOR gates, the output of each is connected to one of the inputs of the other in the configuration shown in Fig. 2.10. Binary Decision logic can implement a FF simply by connecting output lines to input lines, as shown in Chapter 3.

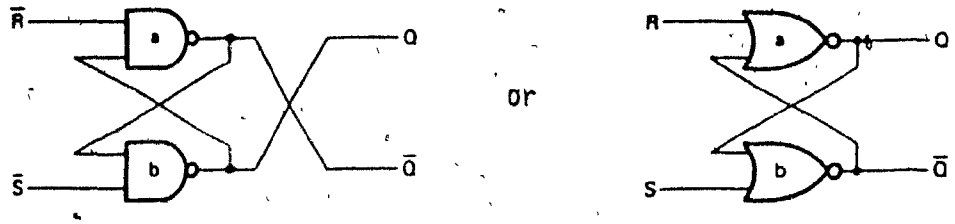


Fig. 2.10: The Latch Flip Flop.

Single Shots. Single shots, Fig. 2.11, also called monostable multivibrators, have only one stable state. A one shot logic element is triggered by a falling or rising edge of an input, causing an output pulse whose width may be varied.

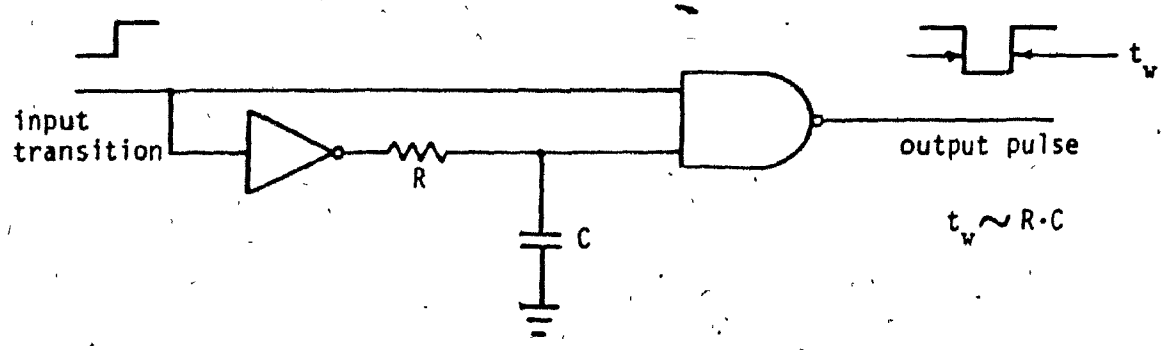


Fig. 2.11: Single Shot.

2.6 Logic Design Methods, Examples

2.6.1 Industrial Control Example

The following simple industrial control system will illustrate some of the concepts presented above.

Consider the water reservoir in Fig. 2.12. The two level indicators A and B provide a 'TRUE' signal when the water level goes below 2m and 3m respectively. Two outlet valves, one big and one small are connected to two micro-switches C and D, respectively. The switches are activated (i.e., 'TRUE') when their respective valve is open.

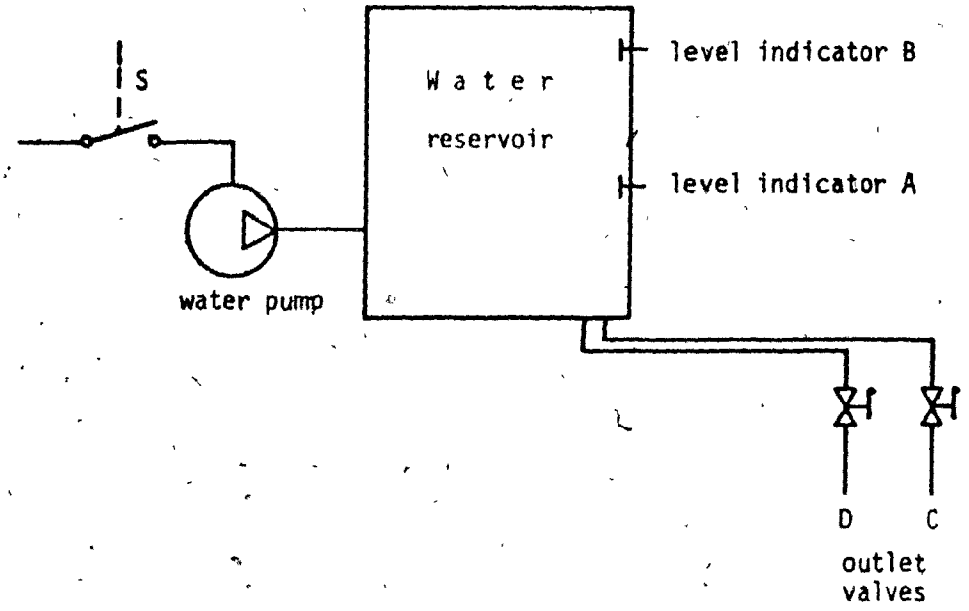


Fig. 2.12: Water Reservoir System.

The control system activates the water pump when the following conditions prevail:

1. The water level is under 2m, regardless of the valves' position.
2. The water level is between 2m and 3m and the big valve is open.
3. The water level is over 3m and both valves are open.

The Boolean parameters A, B, C, and D represent the level indicators and the valve micro-switches respectively. The water pump is represented by the Boolean function S(A,B,C,D) where S=1 means the pump is ON and S=0 the pump is OFF. The state of A,B,C and D affect the water pump in the following manner:

	A	B	C	D	S
water level < 2m	1	1	x	x	1
3m > water level > 2m	0	1	1	x	1
water level > 3m	0	0	1	1	1

In all other permutations the pump should be turned off.

The function which starts the water pump is:

$$S = AB + \bar{A}BC + \bar{A}\bar{B}CD \qquad (2.10)$$

Because B can never be 0 (i.e., water level > 3m) when A = 1 (i.e., water level < 2m), a 'don't care' condition can be assigned to B in the first minterm. The function S(A,B,C,D) can be rewritten as:

$$S = A + \bar{A}BC + \bar{A}\bar{B}C0 \qquad (2.11)$$

Using the Karnaugh map in Fig. 2.13, Equation 2.11 may be simplified to :

$$S = A + B \cdot C + C \cdot D = A + C \cdot (B + D) \qquad (2.12)$$

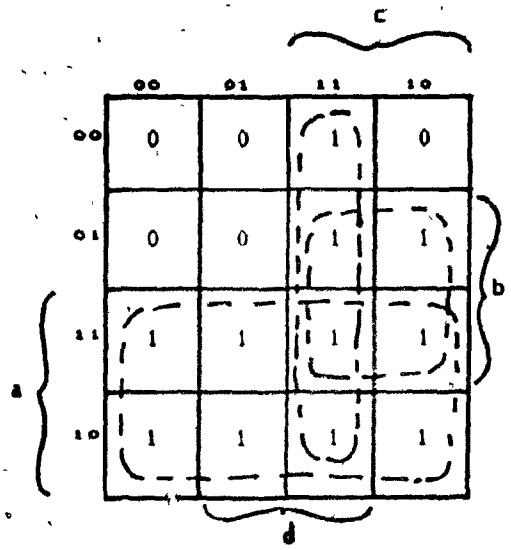


Fig. 2.13: Karnaugh Map for Equation 2.11.

This function may be implemented by two 2-input OR gates and a 2-input AND gate as shown in Fig. 2.14a. If a requirement for a NOR gates implementation exists due to the universality of that gate, a maxterm implementation is preferred, thus getting:

$$\bar{S} = \overline{AC} + \overline{ABD} \qquad (2.13)$$

And using DeMorgan theorems:

$$S = \overline{(\overline{(A+C)} + \overline{(A+B+D)})} \qquad (2.14)$$

The implementation with three NOR gates is shown in Fig. 2.14b.

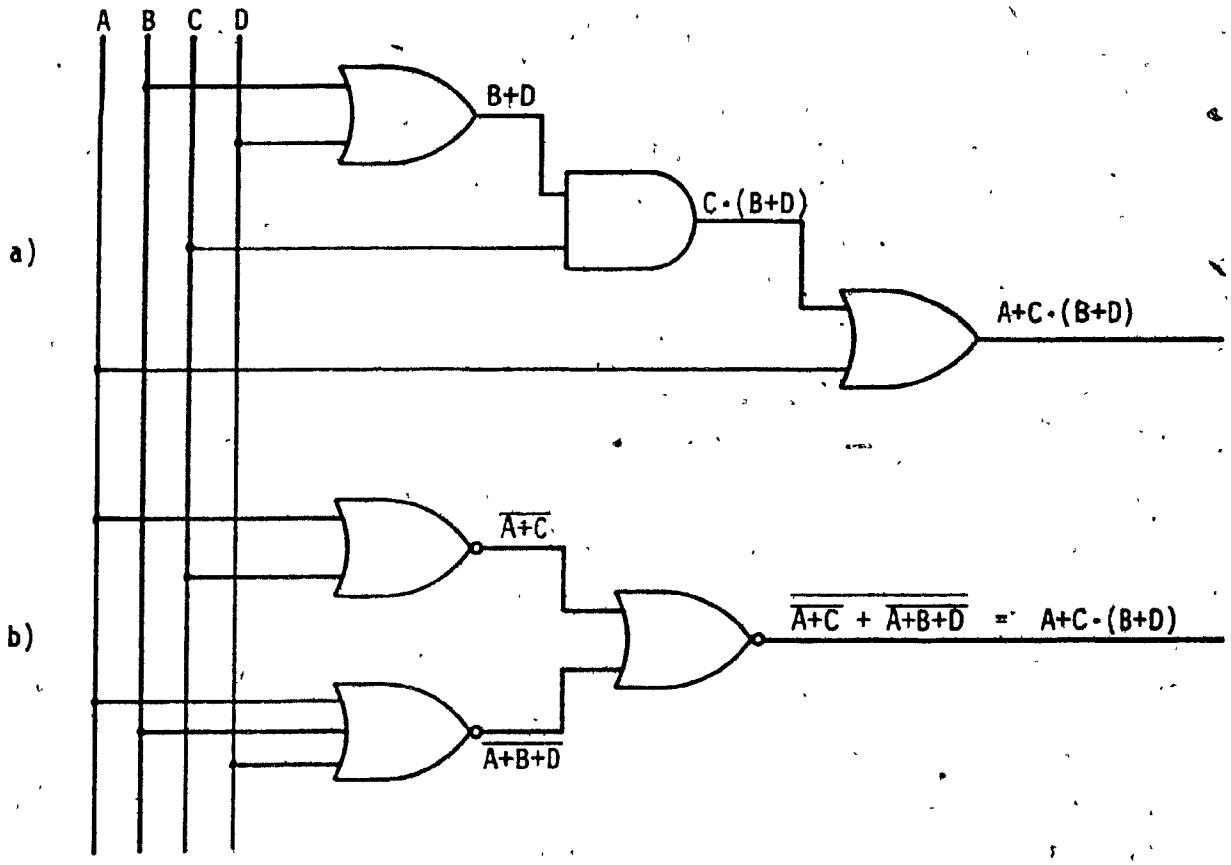


Fig. 2.14: Control System Implementation: a) OR and AND gates.
b) NOR gates.

2.6.2 Hard Wired Implementation, Ladder Diagrams

Ladder diagrams have been used for many years to describe logic functions. This is common in an industrial environment where most logic used to be implemented with electromechanical relays, Fig. 2.15. Notice that the vertical lines are equivalent to the ground (low) and voltage (high) levels of the circuit. Each "rung" represents a logic circuit.

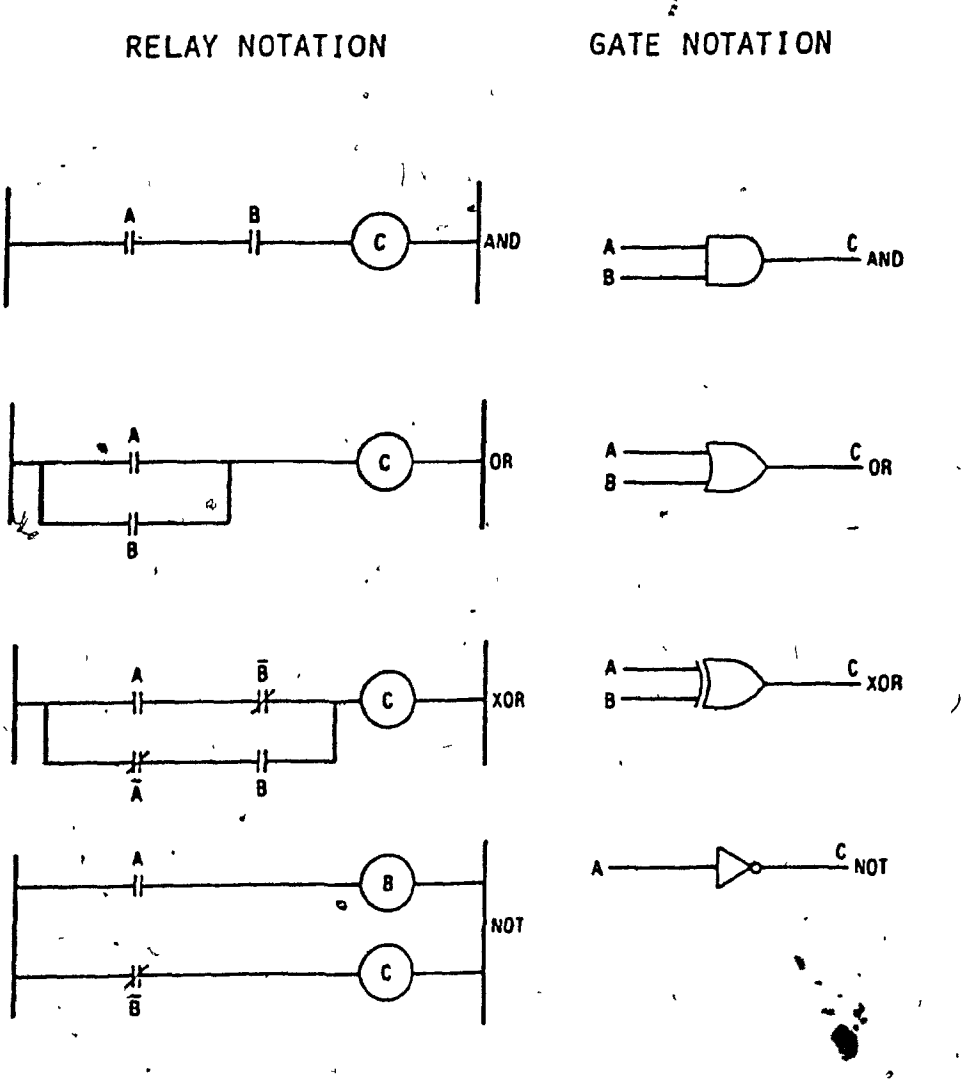


Fig. 2.15: Relay Logic/Ladder Diagram Notation.

A simple motor starter circuit is illustrated in Fig. 2.16. When the circuit output 1M (circled) is TRUE the motor coil is energized. This output signal also serves as an input to the start/seal segment operating as a FF element to memorize or latch the momentary closing of the start button.

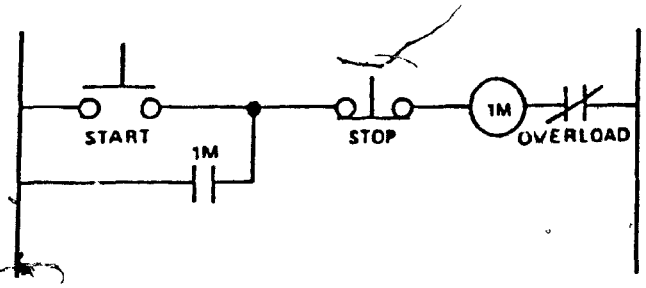


Fig. 2.16: Motor Starter; Ladder Diagram.

A batch process of a material weighing system is shown in Fig. 2.17. Batch ingredients are fed from their respective storage bins into the weigh scale(s) according to a prescribed formula. The weighed batch is then routed to the mixer to be mixed for a specified period of time.

The ladder diagram describing some of the control functions involved in the weighing and mixing process is illustrated in Fig. 2.18. The first "rung" is the implementation of the bin 024 feeder control. Comparator and timer modules are used in order to monitor material weight and batch mixing time. The process is described in detail in chapter 6.

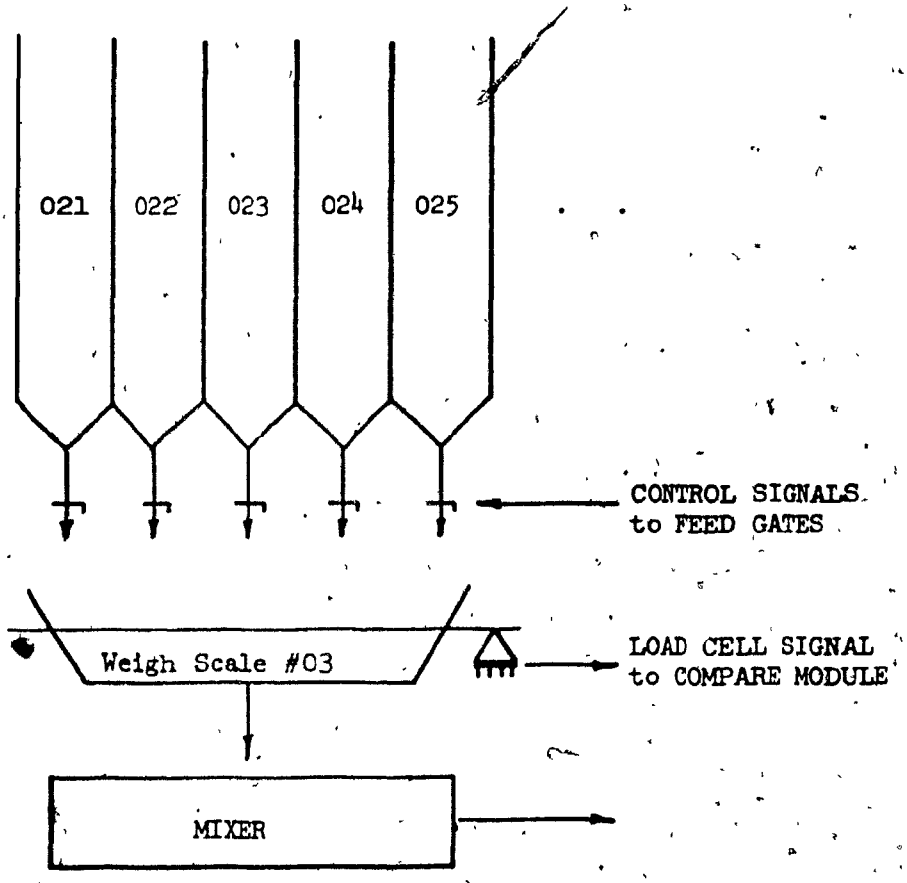


Fig. 2.17: Batch Process; Material Weighing.

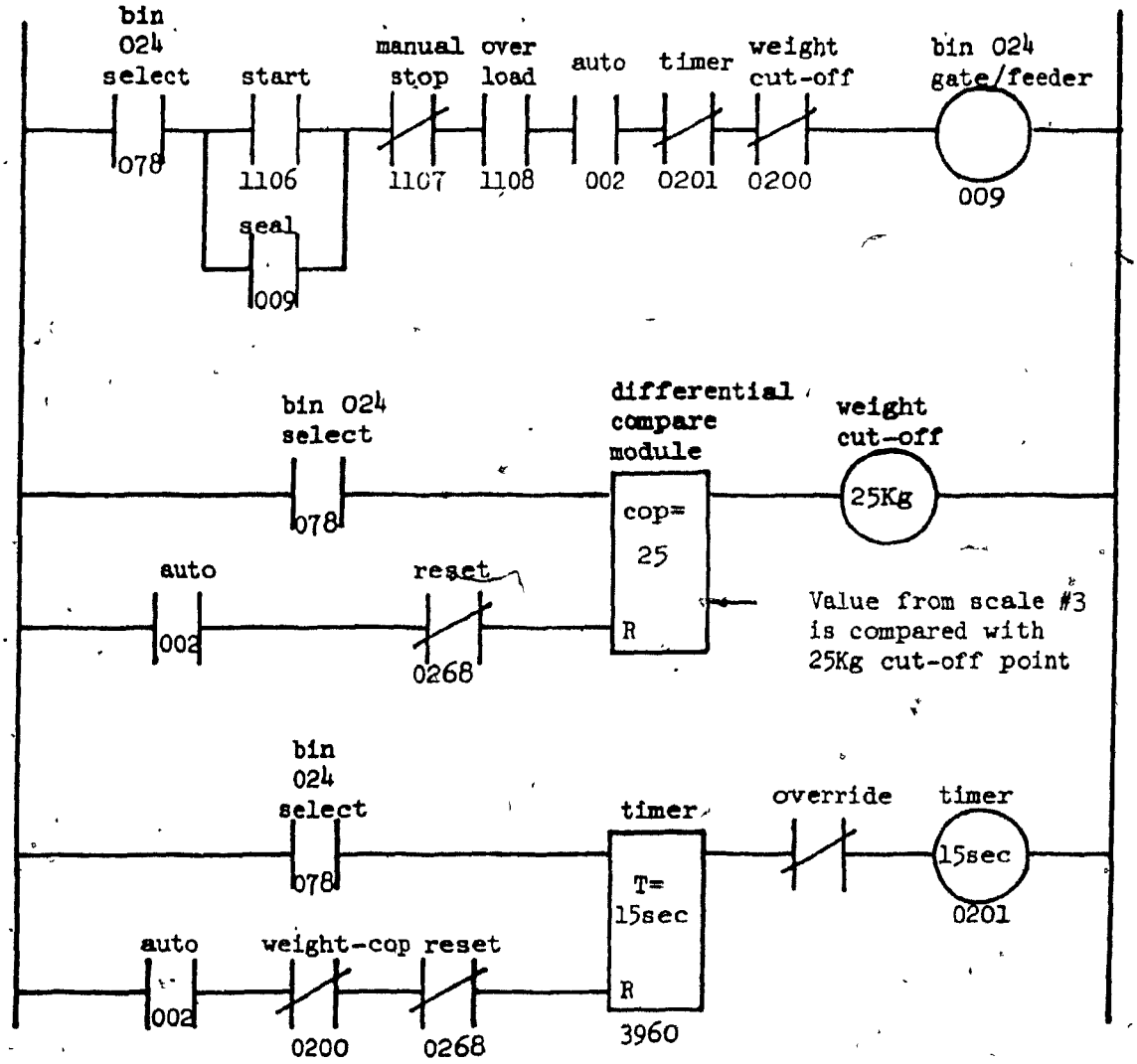


Fig. 2.18: Batch Process; Ladder Diagram.

2.6.3 Function Generation - PLAs

Canonical forms of logical functions can be generated using 'two-level networks'. A sum of products form is shown in Fig. 2.19a. In the first level each minterm is generated by an AND gate. The function value is generated in the second level by ORing all the AND gates outputs. This AND-OR network can be used to implement any digital function. Similarly, a canonical product of sums form can be obtained using an OR-AND network, Fig. 2.19b.

Multiple-functions can be implemented by the 'two level' networks known as Programmable Logic Arrays (PLA). Fig. 2.20 demonstrates the realization of three digital functions F0(x,y,z), F1(x,y,z), and F2(x,y), Equations 2.15a-c, using a single PLA.

$$F0 = xyz + x\bar{y}\bar{z} + \bar{x}y\bar{z} + \bar{x}\bar{y}z \qquad (2.15a)$$

$$F1 = xyz + xy\bar{z} + x\bar{y}z + \bar{x}yz \qquad (2.15b)$$

$$F2 = x\bar{y} + \bar{x}y \qquad (2.15c)$$

The PLA consists of an AND matrix and an OR matrix. The AND matrix accepts the functions' variables as inputs and generates the minterms. The variables are represented by the horizontal lines. Each generated minterm, represented by a vertical line, only consists of those variables connected to it. The interconnections are indicated by a '. Similarly, the OR matrix accepts the AND matrix minterm outputs to generate the function outputs.

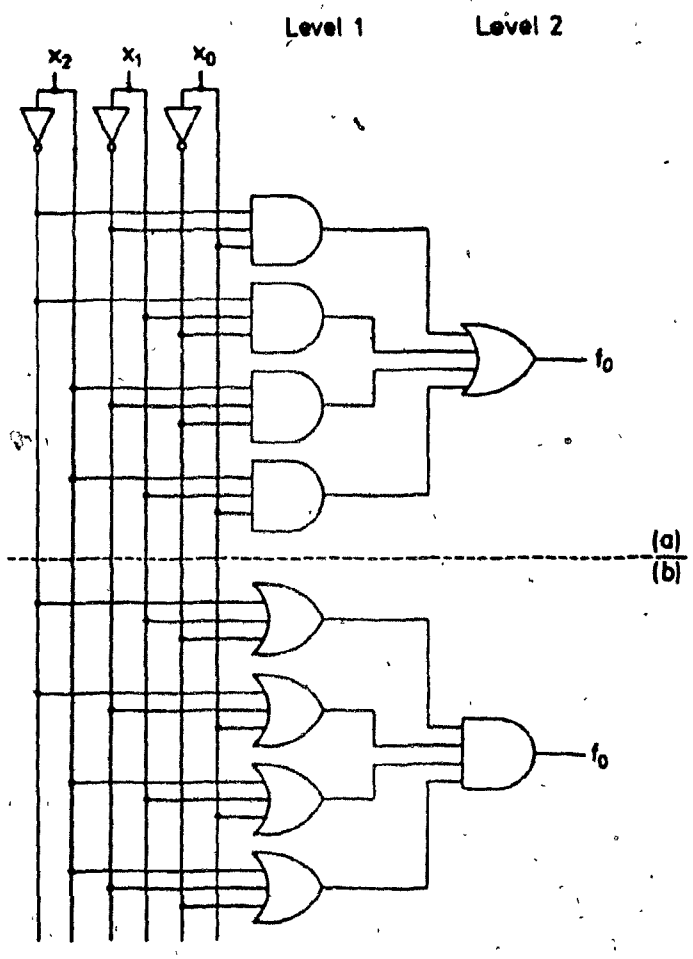


Fig. 2.19: Canonical Functions Implementations
a) AND-OR Network. b) OR-AND Network.

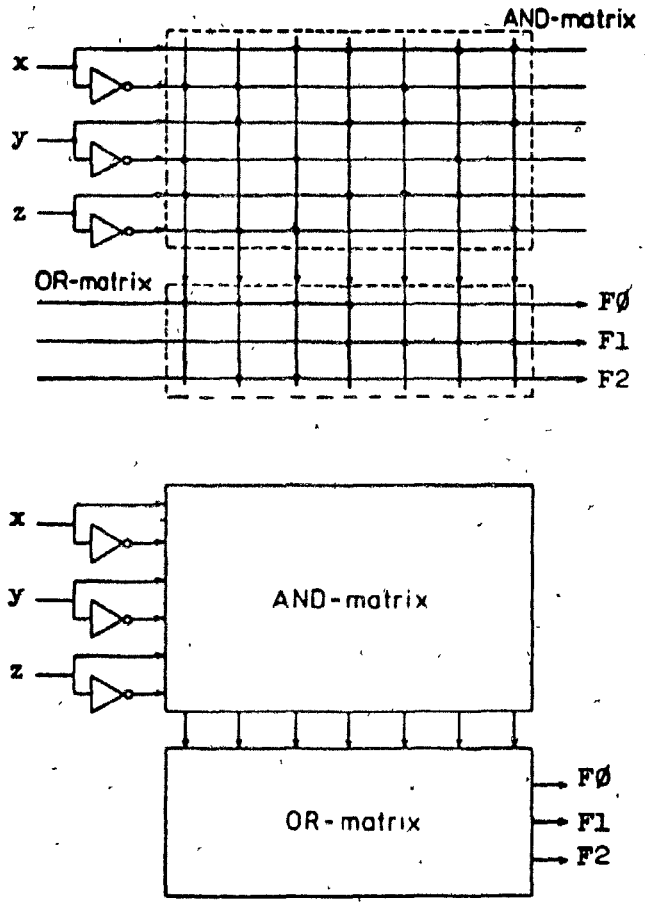


Fig. 2.20: Multiple Functions Implementation with PLAs.

CHAPTER 3: BINARY DECISION THEORY

3.1 Introduction

BD machines are simple automata that execute Binary Decision instructions to evaluate logic functions. Binary Decision can be applied to many diverse fields. Some examples are: decision table programming, databases, identification, pattern recognition, artificial intelligence, biology, switching theory.

Lee's BD program representation of Boolean functions in 1959 [LE59] was the first known application of BD theory to logic design and switching circuits implementation. Lee showed that the representation of logic functions by BD programs has some advantage over conventional Boolean representation. In 1978 Akers [AK78] introduced a tree like graphical representation of binary decision programs. These trees are simple to visualize and understand. The design of Binary Decision process control computing devices was begun by Boute [BO76] in 1976. A BD controller prototype based on the Boute machine was developed in the DATAC computer laboratory in 1979.

This chapter covers the fundamentals of BD logic theory as applicable to logic design and process control. BD theorems, their properties and equivalence to Boolean operations are presented in section 3.2. Techniques to optimize BD trees and programs are discussed in section 3.3. Presentation and analysis of single-bit and multi-bit BD comparator programs follows in section 3.4.

3.2 Binary Decision Logic

3.2.1 Binary Decision Programs

BD programs consist of TEST & BRANCH instructions which can be written as:

$$X \; ; \; A \; , \; B \qquad (3.1)$$

This reads as:

IF X THEN A ELSE B

A and B are the locations of two additional instructions one of which is chosen to be executed next. The branching 'decision' is based on the value of the digital parameter X. One or more TEST & BRANCH instructions are executed sequentially before reaching a program output assignment.

As a simple example the 2-input Boolean XOR expression, Equation 3.2, is described by a BD program:

$$F(x,y) = \bar{x}y + x\bar{y} \qquad (3.2)$$

instruction address	test variable	TRUE branch	FALSE branch
0	x	1	2
1	y	F=0	F=1
2	y	F=1	F=0

Two execution steps are required to evaluate the function's value. First, x is tested (by instruction #0) and based on its value, either instruction #1 or #2 is executed. Function output is assigned based on the result of the test of y.

BD programs may be described graphically by BD diagrams which show the BRANCH & TEST decision instructions as nodes interconnected in a diagram or tree form. Each decision node, Fig. 3.1, may have one or more nodes leading to it but can have no more than two emergent branches. The test of a digital parameter x can yield two possible results: A branch (for x='true') and B branch (for x='false'). The branches of the decision node can lead either to additional decision nodes or to a final process output.

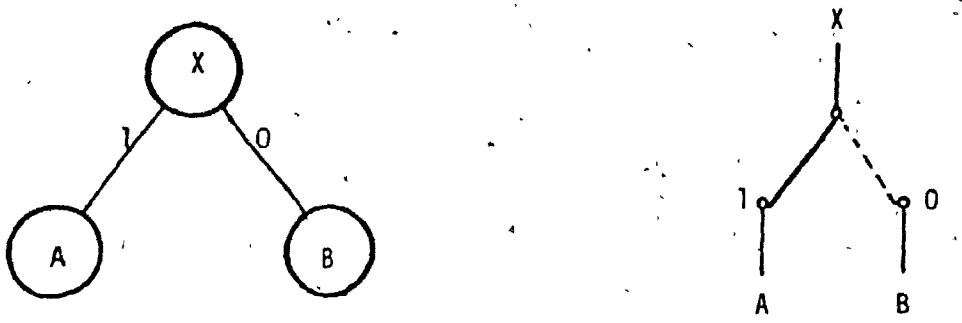


Fig. 3.1: The Elementary Binary Decision Test and Branch Operation.

Nodes of BD trees are qualified by the single parent property, i.e., a single branch is directed to each node. BD diagrams as defined by Akers may have nodes with more than one branch directed to them. BD trees are therefore a subset of BD diagrams.

3.2.2 BD Theorems

Theorem #1: Any Combinatorial Function can be evaluated by a BD Program.

Proof: Expanding the Boolean expression of any combinatorial function using the classical Shannon Expansion theorem leads to an equivalent BD program of that function. A general combinatorial function F(w,x,y,z,...) can be expanded in a stepwise fashion starting with a first (arbitrarily chosen) variable 'w':

$$F(w,x,y,z, \ldots) = \bar{w}F0(x,y,z,\ldots) + wF1(x,y,z, \ldots) \quad (3.3)$$

F0 and F1 represent two subfunctions derived from F when substituting w=0 and w=1 respectively. The two cases may be viewed as the two branches of the first TEST & BRANCH instruction of the BD program for the function F or the top decison node in the BD diagram for F. In the same manner F0 and F1 may be further sub-divided into two subfunctions each, corresponding to x=1 and x=0. The complete expansion results in the function's BD diagram and program and is independent of

the function size or form.

Example: Consider the four variable function F(w,x,y,z) of Equation 3.4.

$$F(w,x,y,z) = \bar{w}\bar{x}\bar{z} + wxz + y\bar{z} + xy \qquad (3.4)$$

The first expansion step (for w) yields:

$$F = \bar{w}(\bar{x}\bar{z} + y\bar{z} + xy) + w(xz + y\bar{z} + xy)$$
$$= \bar{w}\cdot F0 + w\cdot F1$$

The second expansion step (for x) is performed on F0 and F1 separately:

$$F0 = \bar{x}\bar{z} + y\bar{z} + xy$$
$$= \bar{x}(\bar{z} + y\bar{z}) + x(y\bar{z} + y)$$
$$= \bar{x}\cdot F00 + x\cdot F01$$

$$F1 = xz + y\bar{z} + xy$$
$$= \bar{x}(y\bar{z}) + x(z + y\bar{z} + y)$$
$$= \bar{x}\cdot F10 + x\cdot F11$$

The third expansion step (for y) is performed on F00, F01, F10 and F11 separately:

$$F00 = \bar{y}(\bar{z}) + y(\bar{z}) = \bar{y}\cdot F000 + y\cdot F001$$

$$F01 = \bar{y}(0) + y(\bar{z} + 1) = \bar{y}\cdot F010 + y\cdot F011$$

$$F10 = \bar{y}(0) + y(\bar{z}) = \bar{y}\cdot F100 + y\cdot F101$$

$$F11 = \bar{y}(z) + y(z + \bar{z} + 1) = \bar{y}\cdot F110 + y\cdot F111$$

The fourth expansion step (for z) is performed on F000, F001, F010, F011, F100, F101, F110 and F111 separately:

$$F000 = \bar{z}(1) + z(0)$$

$$F001 = \bar{z}(1) + z(0)$$

$$F010 = \bar{z}(0) + z(0)$$

$$F011 = \bar{z}(1) + z(1)$$

$$F100 = \bar{z}(0) + z(0)$$

$$F101 = \bar{z}(1) + z(0)$$

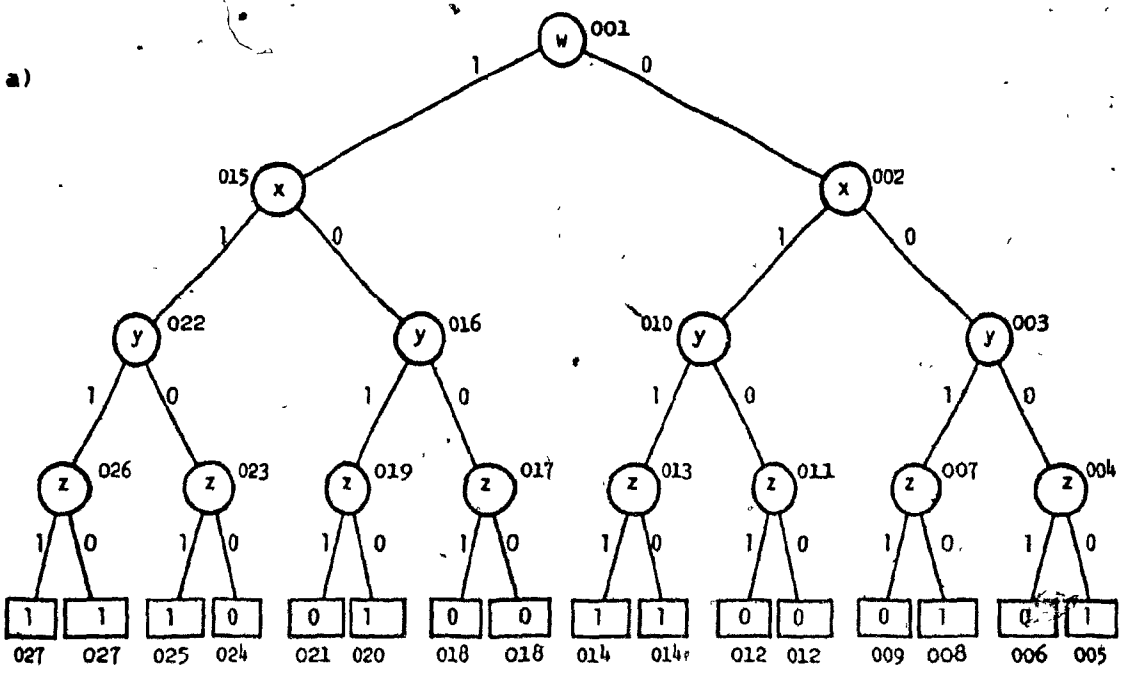
$$F110 = \bar{z}(0) + z(1)$$

$$F111 = \bar{z}(1) + z(1)$$

The BD diagram and its associated BD program are given in Fig. 3.2.

Any combinatorial Boolean expression has an equivalent BD program. Notice however that a full BD diagram and program was obtained as a result of the Shannon Expansion. The above BD program may be reduced using minimization techniques (see section 3.3).

a)



b)

instruction address	instruction	input variable	TRUE branch	FALSE branch	output value(s)
001	TEST INPUT	w	015	002	
002	TEST INPUT	x	010	003	
003	TEST INPUT	y	007	004	
004	TEST INPUT	z	006	005	
005	OUTPUT				-- 1 --
006	OUTPUT				-- 0 --
007	TEST INPUT	z	009	008	
008	OUTPUT				-- 1 --
009	OUTPUT				-- 0 --
010	TEST INPUT	y	013	011	
011	TEST INPUT	z	012	012	
012	OUTPUT				-- 0 --
013	TEST INPUT	z	014	014	
014	OUTPUT				-- 1 --
015	TEST INPUT	x	022	016	
016	TEST INPUT	y	019	017	
017	TEST INPUT	z	018	018	
018	OUTPUT				-- 0 --
019	TEST INPUT	z	021	020	
020	OUTPUT				-- 1 --
021	OUTPUT				-- 0 --
022	TEST INPUT	y	026	023	
023	TEST INPUT	z	025	024	
024	OUTPUT				-- 0 --
025	OUTPUT				-- 1 --
026	TEST INPUT	z	027	027	
027	OUTPUT				-- 1 --

Fig. 3.2: The Expanded F(w,x,y,z): a) BD Diagram. b) Program.

Theorem #2: The number of steps required to evaluate an n-input digital combinatorial function in a BD machine is equal to n or less.

Proof: The proof of this theorem can be shown graphically as follows:

Take a general n-input digital function described in a truth table with 2^n rows, each row representing a minterm. A full BD tree representation of such function, consisting of 2^n-1 decision nodes and 2^n output nodes provides 2^n possible paths leading to the 2^n possible output nodes. Each path corresponds to a truth table row. The function is evaluated by traversing through a single path. I.e., only one minterm needs to be evaluated. The traversal procedure is initiated by entering the top node and stepping down according to the test result at each decision node until an output is produced. Each level corresponds to one input variable. During execution there is no repetitious examination of input variables. Hence, the maximum number of steps is equal to the number of variables.

Example:

A general 2-input digital function may be described in a four-row truth table, Fig. 3.3a. The BD tree representation of such function, Fig. 3.3b, consisting of three decision nodes and four output leaves (in rectangular shape), provides four possible paths producing four possible outputs corresponding to the four truth table rows. Function evaluation is done by entering the top node and stepping down according to the test

result at each decision node until an output is produced. The path corresponding to a<b is darkened. No path can include an input variable twice. Path execution implies the testing of each input variable once and only once. Most digital functions which represent actual cases do not require a full BD tree description. Some paths therefore do not include all input variables, resulting in function evaluation in less then n steps.

a)

a	b	F(a,b)
0	0	f00
0	1	f01
1	0	f10
1	1	f11

b)

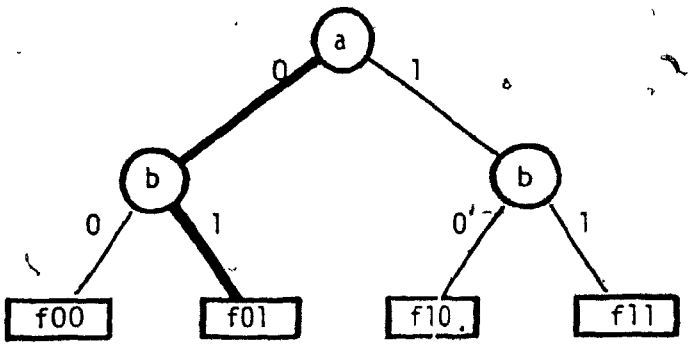


Fig. 3.3: Full BD Tree Representation of F(a,b),
a) Truth Table. b) BD Tree.

A comparison between speeds of function evaluation for both the BD and Boolean implementations is shown graphically in Fig. 3.4. The BD method has an upper bound for the number of steps required to arrive at the function output which is equal to the number of variables. The average number of steps required for function evaluation using Boolean Logic grows exponentially with the number of inputs as all function literals must be evaluated and accumulated before reaching an output.

3.2.3 BD Boolean Equivalences

Fig. 3.5 depicts the elementary Boolean combinatorial logic operations OR, AND, XOR and NOT with their equivalent BD diagrams. Any combinatorial logic function can be described by a BD diagram derived from logic primitives.

Notice the dashed branches in the OR and the AND diagrams. In the OR diagram for example, the dashed branch can replace the non-dashed 1 branch emerging from node 'a'. Such a diagram can be reduced by pruning because no matter what 'b value is, the function value is 1. Similarly, the 0 branch leaving node 'a' in the AND diagram can be pruned and replaced by the 0 dashed branch. BD diagram minimization techniques are discussed in section 3.3.

Any sequential function can be implemented by combining combinatorial logic elements with memory elements composed of Flip Flops. If a Flip Flop can be implemented in a BD program then any sequential function can be implemented.

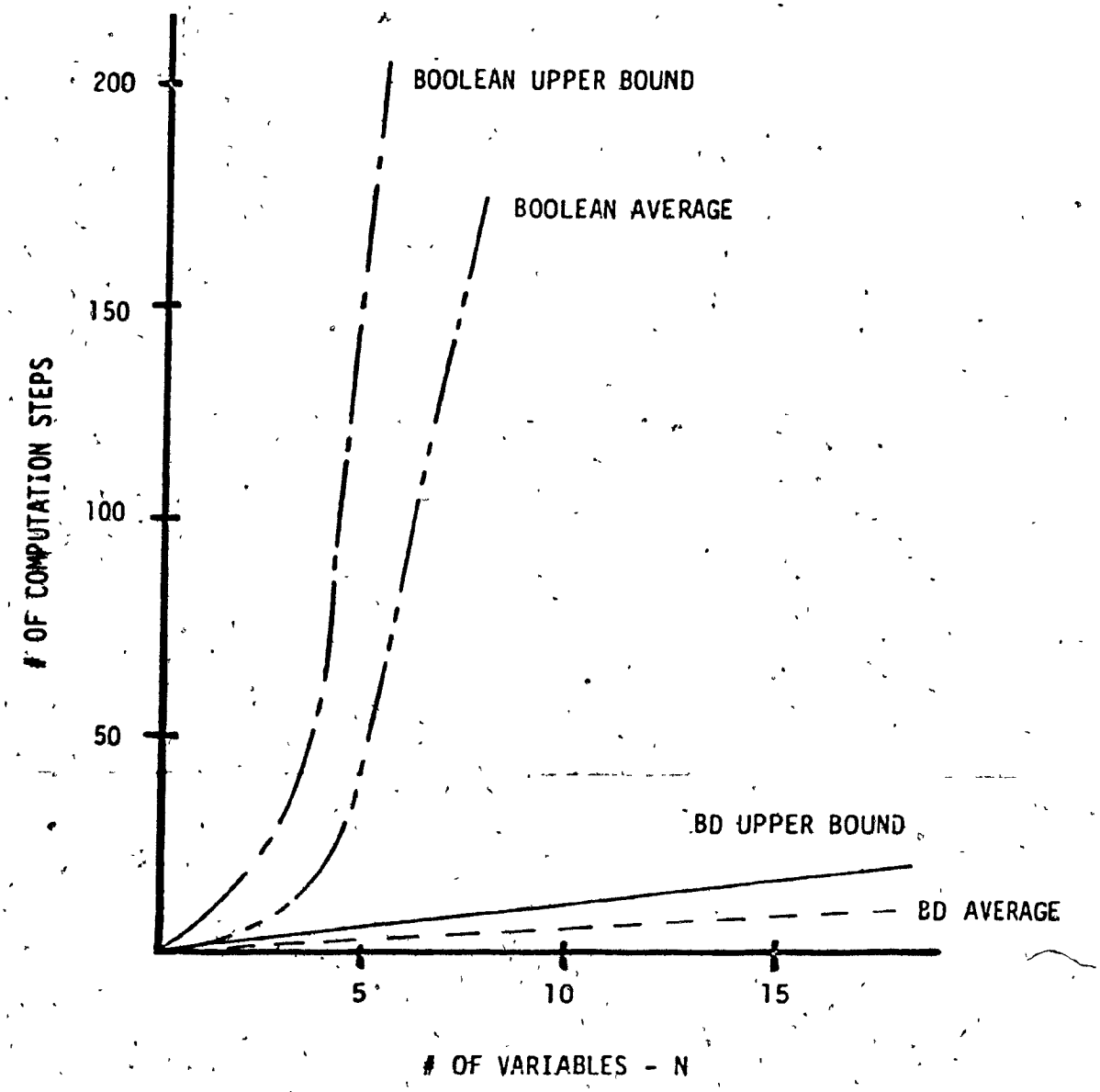
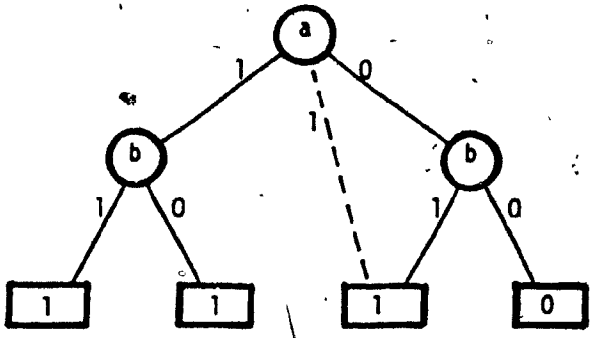


Fig. 3.4: Function Execution Speed; BD vs. Boolean.

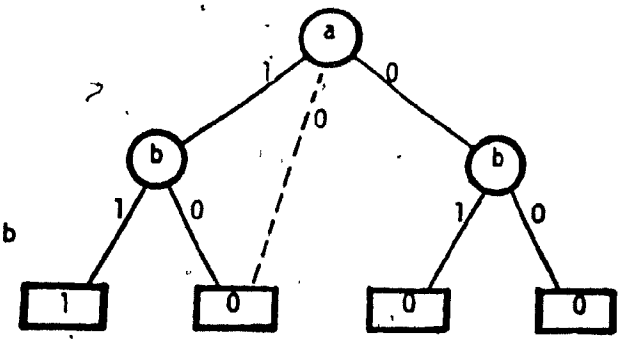
OR gate:

F(a,b) = a+b



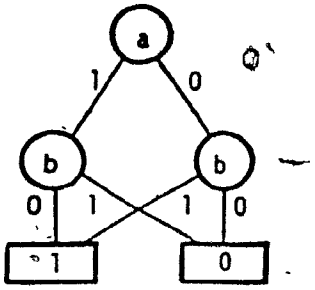
AND gate:

F(a,b) = a·b



XOR gate:

F(a,b) = a + b



NOT gate:

F(a) = $\bar{a}$

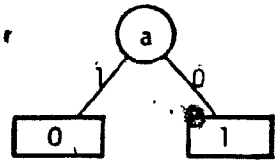


Fig. 3.5: BD Diagrams of OR, AND, XOR and NOT Gates.

Sequential logic implementation is demonstrated in Fig. 3.6, by a Toggle Flip Flop. The FF output Q is effected by the edge-trigered toggle input T. The left side of the diagram represents Q=0 and the right side Q=1. The previous state of the FF is memorized in the diagram itself by the two feed-back loops which don't pass through an output assignment block. A switch from one side to the other will occur only when input T changes. The inherent description of previous states within the diagram represents an additional advantage of BD over

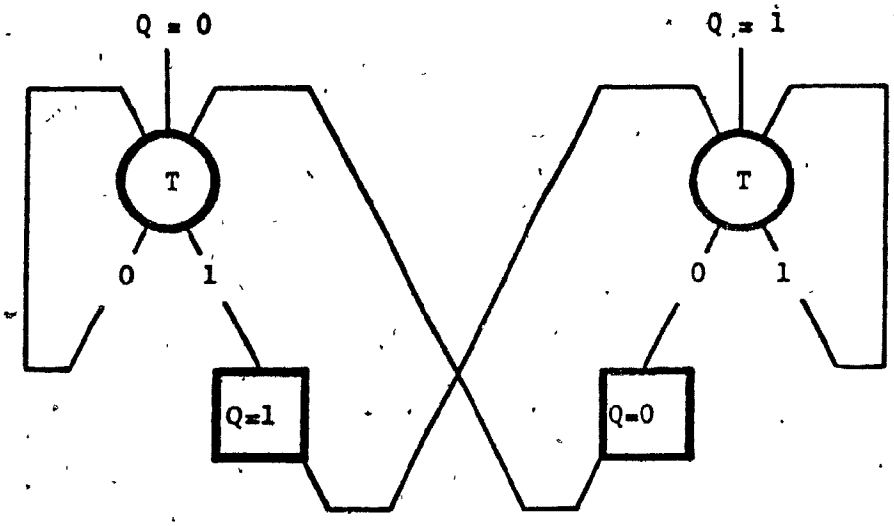


Fig. 3.6: BD Diagram of a Toggle Flip Flop.

Boolean sequential logic, which requires separate external memory to store the previous output states. BD diagrams of other combinatorial and sequential functions may be found in Akers [AK78].

3.2.4 BD Program Properties

BD programs have two important properties that give them computational advantage over the Boolean logic; The ability to compute,

1. a number of different functions in the same program, and
2. functions with n-ary variable outputs.

Multi-functions Program

A number of digital functions, which share the same group of variables or any subset of the group, can be evaluated in a single BD program. Consider the three functions S1(a,b), S2(a,b,c), and S3(b,c) in Equations 3.5a,b,c which share the variables a, b, and c.

$$S1(a,b) = \bar{a}b + a\bar{b} \qquad (3.5a)$$

$$S2(a,b,c) = ab\bar{c} + \bar{a}\bar{b} + bc \qquad (3.5b)$$

$$S3(b,c) = b + c \qquad (3.5c)$$

The truth table for all three functions is shown below.

a	b	c	S1	S2	S3
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	1	0	1	1

Using Boolean technique, the three functions are evaluated by separately computing the sum-of-products for each of the functions. A single BD program, however, can evaluate all three functions simultaneously in a number of steps that is equal to or less than the number of input variables. Input variables in the BD tree serve as the branching criteria to reach the final output. All three functions must share the same set of inputs. The output field can be extended to produce multi-bit output, where each bit corresponds to one function.

In the above example, a 3-bit output field, corresponding to the 3-bit output field in the truth table can generate the 3 functions simultaneously, in parallel.

Multiple-valued n-ary Functions

The advantage of BD over the Boolean method in simultaneous evaluation of several logic functions can be extended to the parallel generation of multiple bits as the output value of a single function. This capacity to evaluate Multiple-valued functions is possible in BD automata with parallel output architecture such as the DATAC prototype built by Holck (see Chapter 4).

A logic function evaluated by a Boolean automata can have one of two discrete values: [0,1]. The BD automata having parallel output architecture, on the other hand, can assign several bits in its parallel output field to represent the value of the function. Logic functions, therefore, are no longer restricted to the [0,1] representation but can take [0,1,01,11,100,101,...] values, or as many discrete values as the size of the output field permits.

An implementation of a single-bit comparator is used as an example. Two binary bits A and B are compared. Three unary output values x, y, and z may be used to describe three possible comparison outcomes: A>B, A=B, and A<B. The comparator truth table is:

inputs		outputs		
A	B	x	y	z
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

Generating x,y, and z by Boolean automata or by serial architecture BD automata (E.g., the Boute machine, Chapter 4, Section 4.2.1) requires three functions:

$$x = A \cdot \bar{B} \qquad (3.6)$$
$$y = A \oplus B$$
$$z = \bar{A} \cdot B$$

Parallel architecture BD automata using parallel outputs, can generate all three values in a single output instruction in two execution steps. The comparator BD diagram is shown in Fig. 3.7. Its program may be written as:

		TRUE	FALSE
0	TEST A	1	2
1	TEST B	"010"	"100"
2	TEST B	"001"	"010"

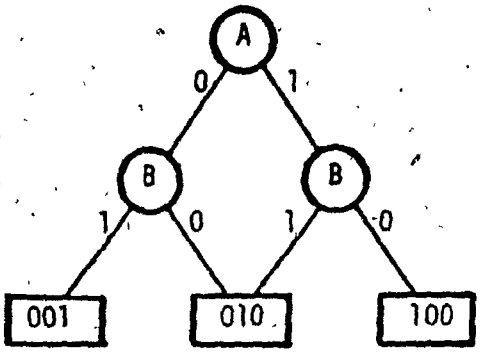


Fig. 3.7: BD Diagram for a Single-Bit Comparator.

3.3 Minimization of BD Programs

3.3.1 BD Program Size

A full BD program (i.e., representing a fully expanded BD tree) requires $2^{(n+1)} - 1$ instructions corresponding to $2^n - 1$ decision nodes and 2^n output leaves, where n is the number of input variables. This kind of exponential growth is highly undesirable. A 20-variable function requires a 2097151 instruction program.

A process represented by 20 variables is quite common, but fortunately most if not all practical processes require only a limited number of unique outputs which is much less than the 2^n possible ones. An impractical large program as described above can be avoided by:

1. Pruning the BD tree to eliminate redundant outputs and impossible outputs. Tree and program minimization and optimization will be discussed in the next section.
2. Assembling the inputs into effectively independent sub-groups each of which has its own program. As an example, the 20-variable program can be broken-down into four 5-variable independent sub programs of 63 instructions each. This tremendous size reduction to 252 instructions incurs no penalty as regard speed of execution.
3. Executing logically independant sub programs in the active working BD memory and dynamically down-loading other sub-programs from storage memory only when these are required. This is possible in a BD machine which is interfaced to a

microcomputer that allows dynamic program downloading (see chapter 4).

3.3.2 BD Program Pruning

A full BD tree representation such as that shown in Fig. 3.1 is only required when all paths produce unique outputs. In most practical applications the number of unique outputs is much less then the 2^n possible ones. It was shown that as the number of inputs grows, BD programs, implemented as complete trees, increase in size exponentially. Similar to minimization of Boolean expressions BD trees can be 'pruned' resulting in an optimized tree which reduces the amount of program instructions and execution steps.

Consider a simple pruning example of the full tree representation of a 4-input AND gate, Fig. 3.8. The full-tree implementation requires 31 instructions. In addition, execution of the tree will always require four test & branch steps to determine the output. The pruned-tree implementation requires only six instructions as only two possible outputs exist, '1' if all four inputs are 1 and '0' for all other conditions. For a uniform random distribution of function variables an average of 1.875 steps is required for its execution.

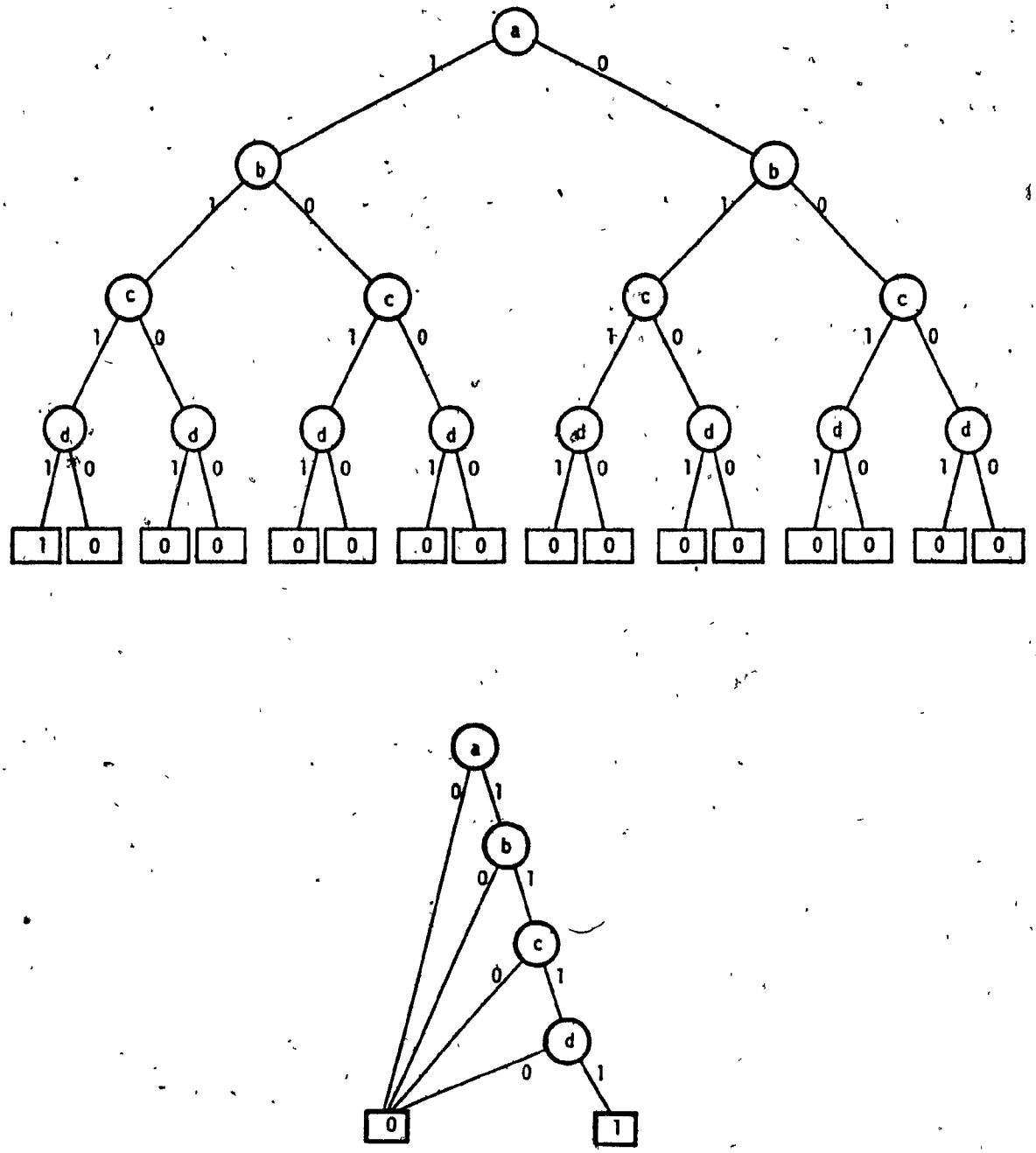


Fig. 3.8: 4-input AND Function:

Full vs Pruned Binary Tree Implementations.

A brief review of BD tree optimization methods developed by Hudson, Karasick and Van-Driel [HU82,VA82,KA84,HU84] is given here. The objectives of the tree optimization are twofold:

1. to obtain the most efficient BD program with regard to program size, i.e., minimization of the required memory storage.
2. to obtain the program that provides the shortest average path length from program (or tree) root to output leaves, i.e., minimization of execution time.

As the tree grows larger optimization procedures tend to become computational intensive and time consuming. A minute of processing time might be considered acceptable for some off-line applications but it is certainly beyond real-time constraints of the process control environment. The time required to execute an optimization procedure should be weighed against process constraints and faster procedures may be selected with possible penalty of incomplete optimization. As the number of inputs increases and the number of unique outputs decreases the optimization process requires larger working memory and more processing time. Only the heuristic Pattern Matching methods presented here are acceptable in those applications where real-time optimization is necessary.

3.3.3 Pattern Matching Algorithms

Pattern Matching algorithms reduce full binary trees to near minimum size by searching for redundant output conditions. Because these algorithms do not reorder the input variables (as, for example, the Quine McCluskey based algorithms do) they are not capable of generating optimal trees.

The Full Tree Reduction method described by Hudson in [HU82] starts from a full tree representation of the function. An ordered truth table containing all possible permutations of input variables and corresponding output states is converted into a BD table, whose entries correspond to the tree nodes and contain information on left branch, right branch and from pointers. All output elements are grouped into 2^n-1 subsets. Starting from the top node of the BD tree, all outputs associated with the left branch are one subset of n/2 elements. Similarly a same size subset is associated with the right branch. The next level is associated with four output subsets each of which has n/4 elements, etc. Subsets of the same size are examined for identity and similarity of their elements. When a redundant subset is detected the BD table is pruned by removing the nodes belonging to the duplicate subset and rerouting its pointer to the original one.

The Virtual Tree Reduction method developed by Karasick [KA84] does not require the full tree BD table representation. The generation of the minimal BD table is accomplished through a recursive post order traversal scheme which creates only those nodes that branch to existing nodes or to output leaves. The method uses an open hash table structure to accumulate all internal nodes employing concatenation of the left and right branch pointers as the entry key. For same input order, the tree generated is the same as in the Full Tree Reduction method. Workspace required for this method is limited to the reduced tree storage requirement.

Differently pruned trees can be obtained from the same function depending upon the ordering sequence of the input set. Both Pattern Matching algorithms do not attempt to reorder the input sequence implied by the truth table. Improvement of the Pattern Matching methods, however, may be achieved if information is available regarding a preferred order of input data and by preprocessing of the truth table.

3.3.4 Quine McCluskey Based Algorithms

Two BD minimization algorithms which generate optimized trees from the Quine McCluskey 'reduced state table' (see chapter 2), were developed by Van Driel [VA82]:

1. The minimum size algorithm: This algorithm generates a tree with a minimum number of decision nodes.
2. The minimum average running time algorithm: This algorithm generates a tree with the minimal average path length.

The reduced state table lists the states of input variables for each of the prime implicants. Each of the input variables may have one of three states in determining the value of each of the prime implicants:

1. the variable is 1
2. the variable is 0
3. the variable state does not play a role, i.e., 'don't care'.

The two algorithms apply different sets of rules according to which the state table is examined and the variables are selected. The optimized tree is constructed from top node to output leaves in the same sequence of the selected variables. Unlike the Pattern Matching algorithms no specific variable order is necessary as the set of rules is designed to determine the preferable variable order from the information given within the state table.

In the minimum size algorithm the first variable is the one which has the least 'don't care' conditions in the PIs as well as the highest count of zeros or ones. In the minimum average running time algorithm the first variable is selected based on the maximum weighted relative information contribution. This value is obtained from the probability that the variable can be a 1 or a 0. Subsequent variables are selected using the same rules which are now applied to subtables generated after discriminating on the selected variable. The number of subtables increases with each step as the process continues.

In addition to the time required for computing the variables from an exponentially increasing number of subtables, the QM methods require a large amount of working memory to store all the subtables. The Pattern Matching algorithms on the other hand do not use tables except the ones which represent the actual tree.

3.4 BD Based Comparison Algorithms

Comparison algorithms are used extensively. A single bit BD based comparator was presented earlier to show the implementation of multi-valued functions. BD comparison algorithms are presented in more detail in this section. The design of a 2-bit BD based comparator is shown in comparison to a Boolean hardwired gate comparator. Extension of the single-bit comparator follows to show a modular n-bit comparator implementation. A comparison with Boolean based comparators is presented to demonstrate the superiority of the BD approach.

3.4.1 2-bit Comparator Implementation

The hardwired circuit for a two 2-bit number comparator is shown in Fig. 3.9. A1, A0, B1 and B0 represent the input bits. X, Y, and Z are the comparator outputs corresponding to A>B, A=B, and A<B results, respectively.

The comparator truth table is shown below. The output values X, Y, and Z may be evaluated using Boolean expressions in Eq. 3.11a-c.

A1	A0	B1	B0	X	Y	Z
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	1	0	1	0	0
1	1	1	1	0	1	0
0	X	1	X	0	0	1
1	X	0	X	1	0	0

X = don't care

$$X = A1 \cdot \overline{B1} + \overline{(A1 \oplus B1)} \cdot A0 \cdot \overline{B0} \qquad (3.11a)$$

$$Y = \overline{(A1 \oplus B1)} \cdot \overline{(A0 \oplus B0)} \qquad (3.11b)$$

$$Z = \overline{(A1 \oplus B1)} \cdot \overline{A0} \cdot B0 + \overline{A1} \cdot B1 \qquad (3.11c)$$

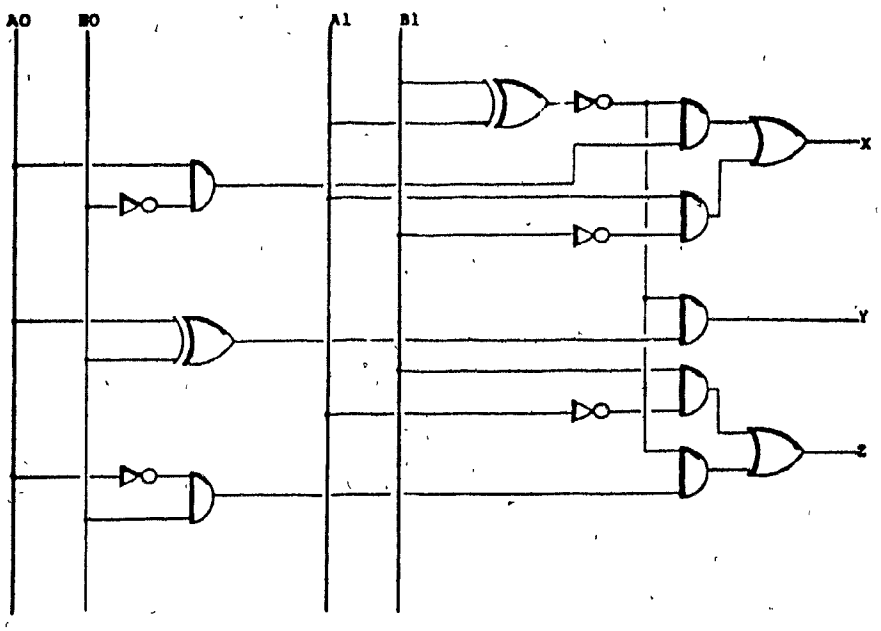


Fig. 3.9: Hardwired Implementation of a 2-bit Comparator.

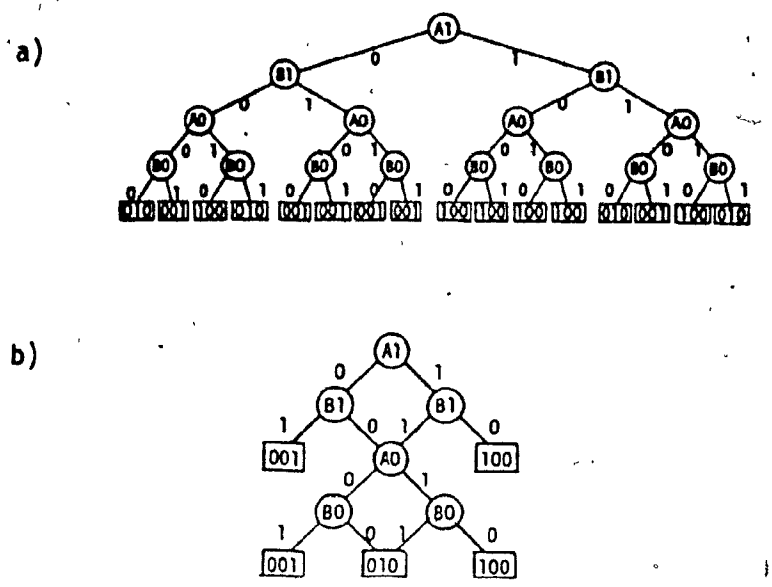


Fig. 3.10: BD Diagram for a 2-bit Comparator:
a) Full Diagram. b) Minimized Diagram.

The full BD diagram for the 2-bit comparator, shown in Fig. 3.10a, may be directly derived from the truth table. This full diagram can be simplified resulting in the diagram shown in Fig. 3.10b. This minimized diagram consists essentially out of two single-bit diagrams interconnected via the equality branch.

The equivalent BD program is shown below.

		TRUE	FALSE
0	TEST A1	1	2
1	TEST B1	3	"100"
2	TEST B1	"001"	3
3	TEST A0	1	2
4	TEST B0	"010"	"100"
5	TEST B0	"001"	"010"

Notice that the number of execution steps can be as low as two but never more than four. Boolean implementation on the other hand requires separate evaluation of x, y, and z.

3.4.2 Multi-bit Comparators

Multi-bit comparator algorithms are an extension of the single bit and the 2-bit comparators. The comparison between two digital numbers is a common function required in control applications. Take a steam boiler control unit as an example. The on/off control of the gas burner is a function of the steam pressure. The burner function is defined by constantly

measuring the steam pressure and comparing it to a predefined set point level. A controller which can perform the numerous comparisons in software, quickly enough to meet real-time constraints, can be employed in many applications using costly hardware-based comparators.

A BD based controller can perform fast comparisons of multi-bit digital numbers. The BD diagram of a general n-bit comparison algorithm is shown in Fig. 3.11. The most significant (MS) pair is compared first. If equality is detected the next MS pair is compared. When inequality is detected an output is established and execution is terminated.

The BD program consists of n segments of the single bit program each of which branches to the next segment in case of equality. The program shown below consists of 3 segments.

		TRUE	FALSE
0	TEST A(n)	1	2
1	TEST B(n)	3	"A>B"
2	TEST B(n)	"A<B"	3
3	TEST A(n-1)	4	5
4	TEST B(n-1)	6	"A>B"
5	TEST B(n-1)	"A<B"	6
6	TEST A(n-2)	7	8
7	TEST B(n-2)	"A=B"	"A>B"
8	TEST B(n-2)	"A<B"	"A=B"

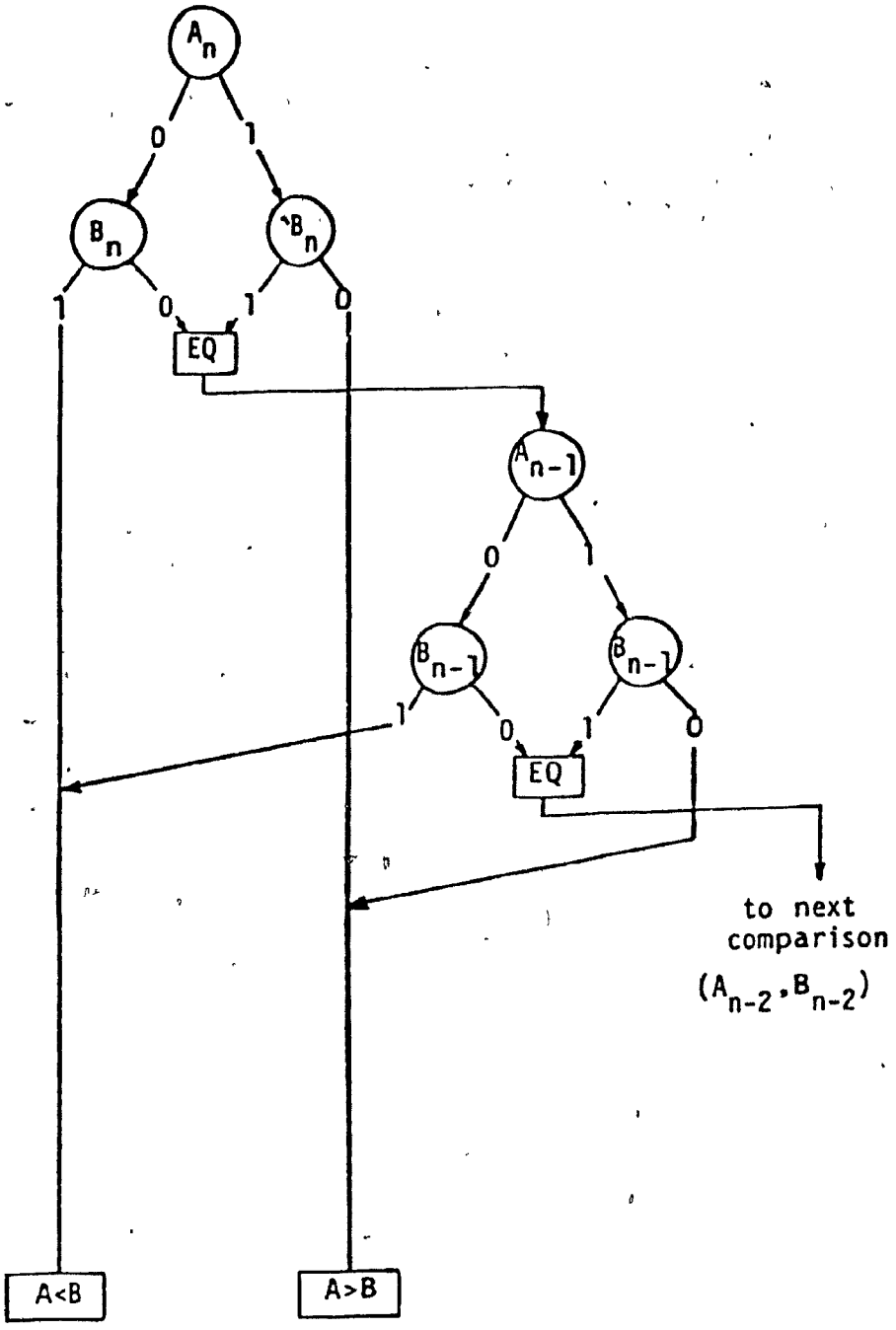


Fig. 3.11: A Multi-bit Comparison Algorithm - BD Diagram.

If a fourth segment is required for comparing 4-bit numbers, the third segment (instructions 6,7,8) should be modified such that the equality outputs (A=B) are replaced with the entry address of segment #4. Different bit-lengths may be accommodated within a single program by simply altering the program entry point.

3.4.3 Time Considerations

Two program steps are required for the execution of each comparison segment. If all n bits are to be compared it will take 2n steps to establish the comparison output. A BD machine operating with 1MHz clock and executing one instruction in one clock cycle can complete a comparison of two 12-bit digital numbers in 24 microseconds.

Normally, inequality will be established in less then 2n steps. The MS bit represents 50% of the full comparison range, two MS bits represent 75%, three MS bits represent 87.5% etc. As the two compared numbers approach equality, more bits must be examined. Thus, a comparison of two n-bit numbers may require as little as two microseconds with a maximum of 2n microseconds.

Most real-time applications require scan times of input variables in the order of few milliseconds. The above comparison example shows that a BD based controller is much faster. It can be used in simultanous execution of a number of control tasks and due to its simplicity it can be employed as a low cost solution in many control applications.

CHAP. 4: BD BASED MACHINES

This chapter describes the BD automaton from which the IRI module was developed. A stand-alone prototype BD processor, suggested by Le-Ngoc [LN79] and built by Holck [ZM79], is described first. The hybrid mP-BD concept is presented next, followed by a detailed description of the expanded prototype BD processor. The fourth section is devoted to the interface between this processor and a microprocessor. A review of the hybrid system's operating modes follows. Finally, the software tools developed for the hybrid system are laid out in the sixth section.

4.1 The Prototype BD Processor

4.1.1 Introduction

The prototype BD processor built in the DATAC computer laboratory is an extension of the Boute machine [B076]. It has been used for BD programming practice, and to demonstrate some simple controller applications and other BD research and development work. It is shown in Fig. 4.1.

The Boute machine is the first known programmable controller that evaluates logic functions using BD rather than Boolean methods. Comprising only 7 chips, the Boute machine architecture, Fig. 4.2, is very simple. It includes a 256 by 16-bit RAM for BD program storage, a presettable program counter with its preset logic, input data selector addressed

by the BD instruction, output latch and clock. BD programs executed in the Boute machine consist of two instructions only: BRANCH and OUTPUT. BD instructions are executed serially. The Branch instruction tests a single input variable, the state of which determines the next executable instruction. Eventually, an OUTPUT instruction places a single bit into the output latch.

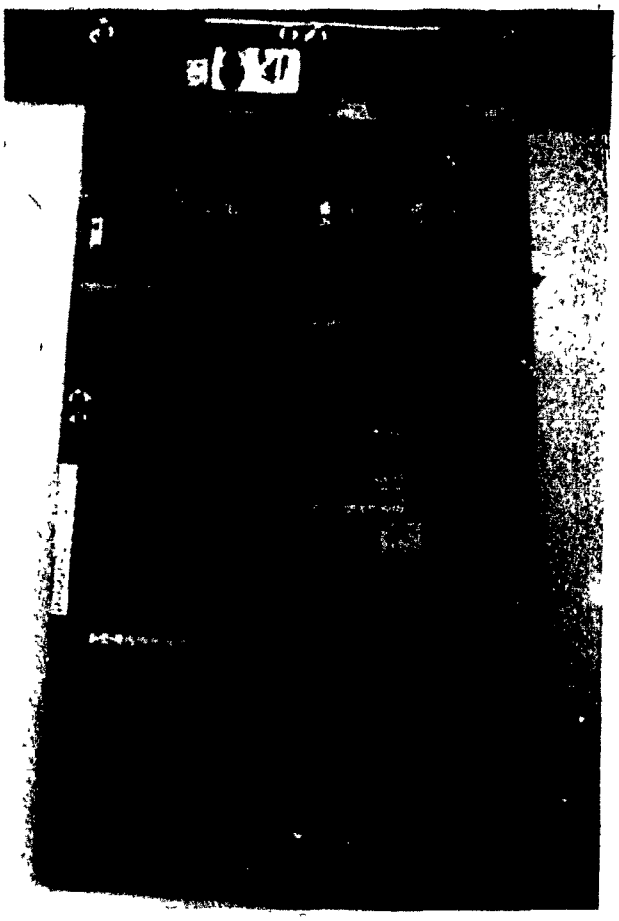


Fig. 4.1: The Prototype BD Processor - Demonstration Unit.

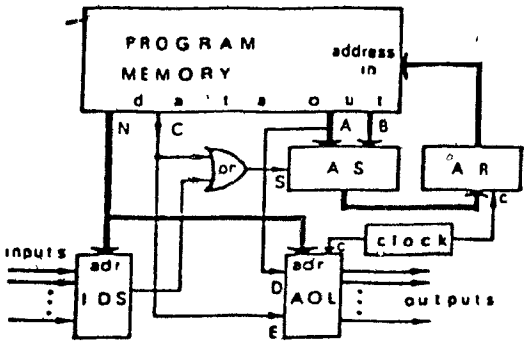


Fig. 4.2: The Boute Machine.

The DATAC prototype extends the Boute concept to incorporate parallel evaluation of multi-bit outputs. A single instruction is executed in one clock cycle, e.g., the selection and the test of the input variable and the branch to the next instruction all take place in one clock cycle. Unlike a conventional Boolean processor which accumulates the Boolean combinations of the input variables in order to arrive at the output, the BD processor examines only one single path through the BD tree. The execution of an n-variable BD program requires a maximum of n steps. The BD processor is therefore superior to conventional Boolean based controllers in those control applications where speed of execution is an important factor. The speed advantage becomes more significant as the number of variables increases.

4.1.2 The Architecture of The BD Processor

The block diagram, Fig. 4.3 describing the BD processor architecture, may be divided into six functional sections:

1. Input Section
2. Decision Logic Unit
3. Clock and Program Control
4. BD Program Storage
5. Output Section
6. Manual Control and Display.

A formal definition of the BD processor is given in Table 4-1 using Instruction-set processor (ISP) notation (see Appendix B for ISP background).

Instruction	\	IR<15:0>
Program Memory	\	Mp[255:0]<15:0>
Program Counter	\	PC<7:0>
Instruction Address	\	IA<7:0> := PC<7:0>
Operation Code	\	OP<1:0> := IR<15:14>
Input Variables	\	IV<63:0>
Output Variables	\	OV<13:0>
Selected Input Varia.	\	X := IV<IR<13:8>>

Table 4-1: The Prototype BD Procesor - ISP Definition.

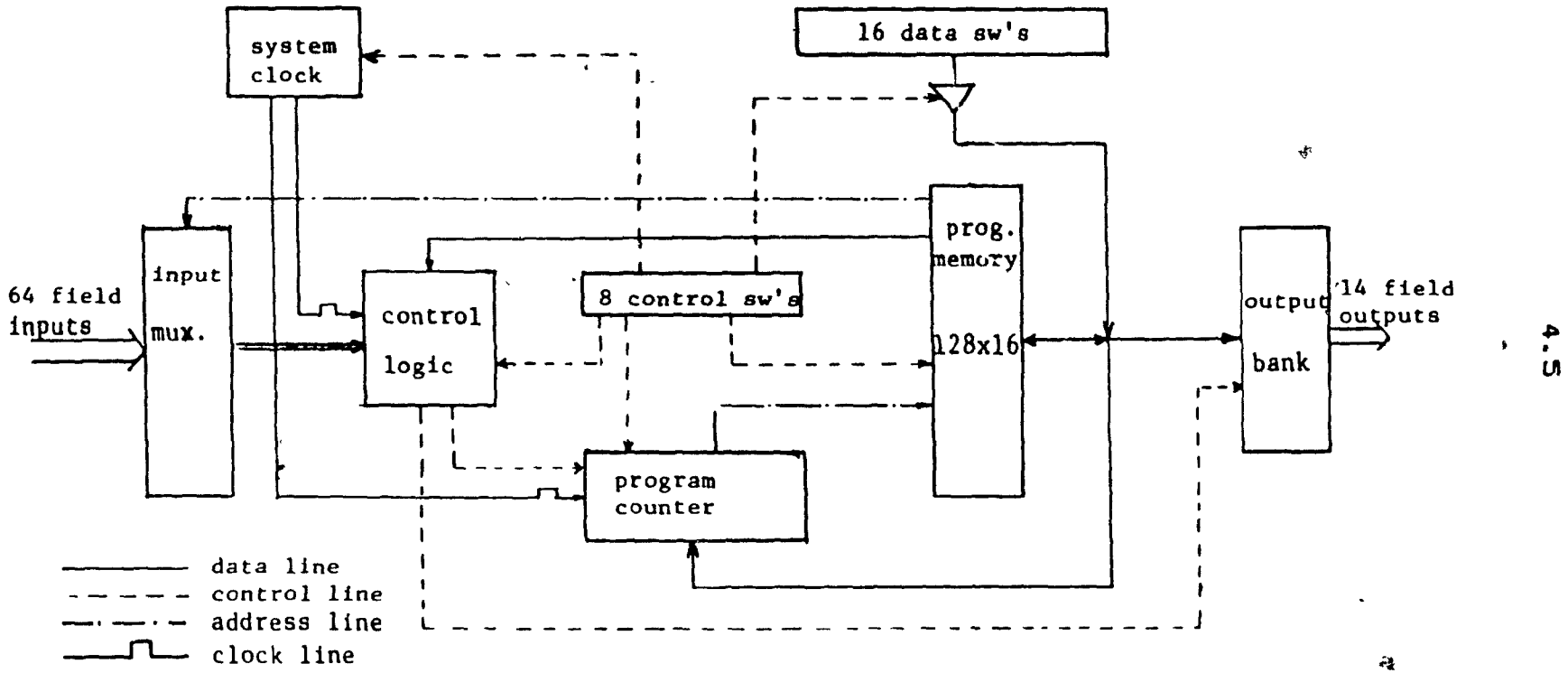


Fig. 4.3: The BD Processor, Block Diagram.

Input Section

The input section shown in Fig. 4.4 consists of input buffers and selection logic. As many as 64 external inputs may be connected. Each input represents a process variable which can be addressed by an input instruction during program execution. The 64 inputs are connected to the data inputs of four 1 of 16 74LS150 data selectors addressed by IR<11:8>. An additional 1 of 4 data selector addressed by IR<13:12> accepts the four outputs of the 74LS150's to route one selected input to the Decision Logic Unit.

Decision Logic Unit

The Decision Logic Unit (DLU), Fig. 4.5, consists of 5 gates. The DLU determines the state of the program counter load line (L) according to the state of the currently selected input line and IR<14> of the current instruction. Two possible branches are provided as follows:

1. If L=1, the PC is preset to IR<7:0> of the current instruction and a jump to another instruction in the memory occurs.
2. If L=0, the PC is incremented by 1 and the next executable instruction is the one immediately after the current instruction.

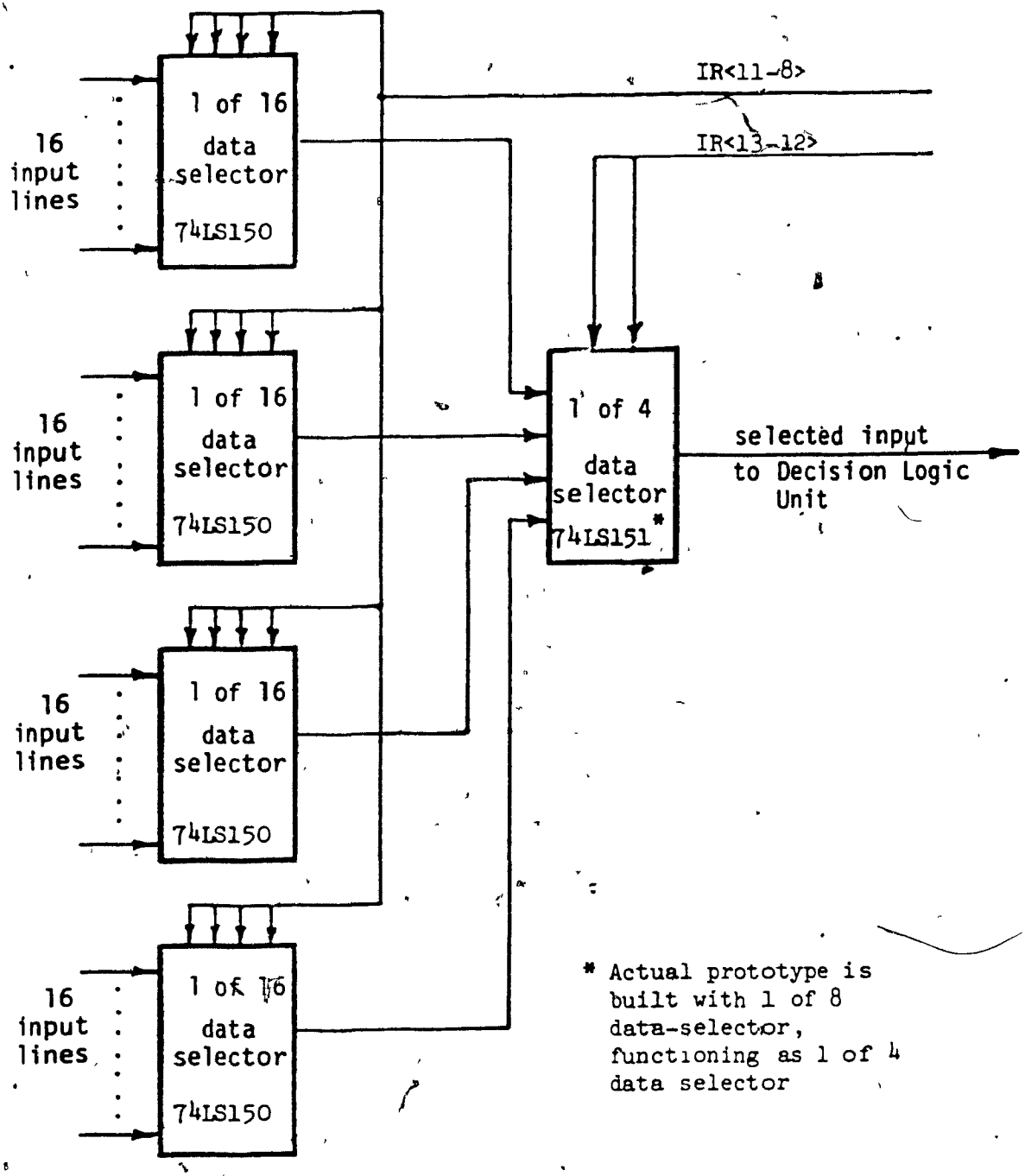


Fig. 4.4: The Input Section.

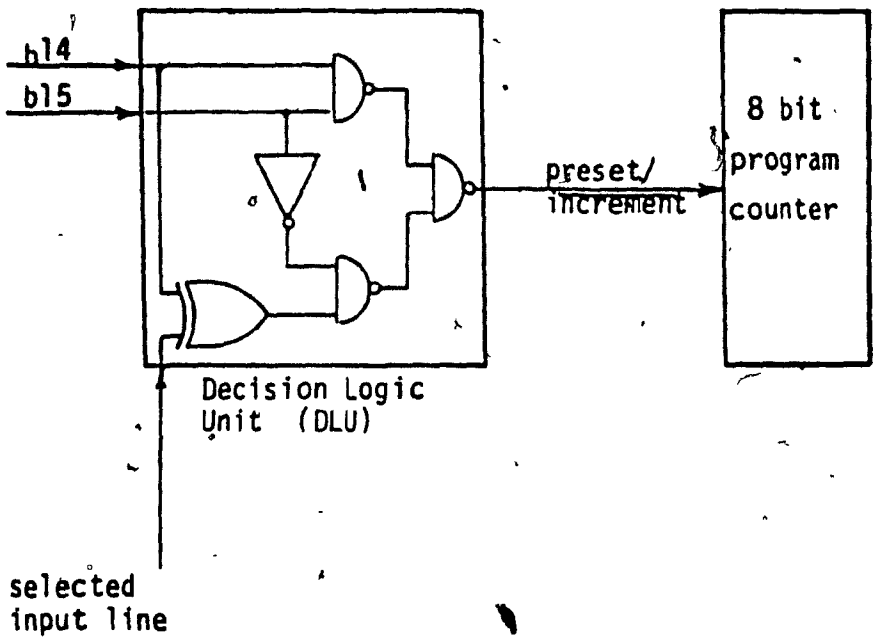


Fig. 4.5: The Decision Logic Unit.

Clock and Program Control

A 1MHz clock is employed to drive the BD prototype circuitry. The clock synchronizes the execution of the BD instructions which take only one clock cycle. The clock is divided to supply two user selectable operating speeds: 70KHz and 0.5Hz. The 0.5Hz is used to observe instruction execution one at a time for debugging and demonstration purposes. In addition programs can be executed in Single-Step (SS) mode. A SS toggle switch, operated manually, provides the clock signal to the system.

BD program Storage

A 256x16-bit RAM is employed to store the executable BD programs. The RAM is addressed by an 8-bit program counter controlled by the DLU. Programs can be downloaded into the memory via a set of 16 data switches and two control switches: R/$\overline{W}$ and Single-Step. (see operational modes).

The Output Section

BD processor outputs, assigned by the output instructions, are latched into a set of 14 D-type Flip-Flop latches. The Output Enable (OE) control switch is used to ensure that outputs can be latched only during program execution phase. Bit 14 and bit 15 of the instruction word are synchronized with the falling edge of the clock via two monostable multi-vibrators (one-shot) - 74LS123. Two strobes are available to implement two types of output instruction. A short output instruction enables only the upper six latches. All 14 latches

are enabled by a long output instruction.

Manual Control and Display

The prototype BD processor is controlled by three sets of manual switches and monitored by two sets of LED displays.

Data-Switches: 16 toggle switches are provided to manually load the BD memory. The switches are buffered by tristate buffers and cannot be activated during execution mode. The eight least significant switches are also used to preset the program counter using the PC Load control switch.

Control Switches: eight control switches are provided.

1. Clear PC - reset PC to 0.
2. R/$\overline{W}$ - directly controls the Read/$\overline{Write}$ input of the program memory.
3. PC Load - loads the program counter to a preset address.
4. Auto Load Enable - enables automatic PC loading by the DLU.
5. 70KHz/0.5Hz - selects one of the two operational clock frequencies.
6. DE - Data Enable - enables data entry from data switches.
7. SS - Single Step Mode - enables the execution of a single instruction.
8. Clock-On - enables clock control of the BD processor.

Input Simulation Switches: eight input lines are connected to these switches to simulate process inputs. They can be set to either 0 or 1 and are addressed as inputs by the program cycling in the BD memory.

Instruction Display: 16 LEDs are used to display the current instruction at the location pointed by the program counter.

Program-Counter Display: eight LEDs display the contents of the current PC.

Output Bank Display: 14 LEDs show the output values assigned by the cycling BD program(s).

4.1.3 The BD Instruction Set

The BD processor executes BD programs such as those described in Chapter 3. The 16-bit long instruction contains the necessary information pertaining to the address of the input variables, node branch addresses and final function outputs. The BD instruction set consists of two INPUT instructions and two OUTPUT instructions. The instruction format is shown in Fig. 4.6.

The operation code stored in IR<15:14> is interpreted as follows:

1. INPUT instructions: IR<15> = 0, IR<14> = 0 or 1. An input variable - X addressed by IR<13:8> is tested. It value is XORed with IR<14> to define the next executable instruction (= branch) as follows:

IR<15> = 0 => X = IV(IR<13:8>> (4.1)

(IR<14> ⊕ X) = 0 => PC <- Mp[IR<7:0>]

(IR<14> ⊕ X) = 1 => PC <- PC + 1

2. OUTPUT instructions: IR<15> = 1. Processor produces outputs as follows:

OP<1:0> = 10 => OV<13:8> <- IR<13:8> (4.2)

PC <- IR<7:0>

OP<1:0> = 11 => OV<13:0> <- IR<13:0>

PC <- PC + 1

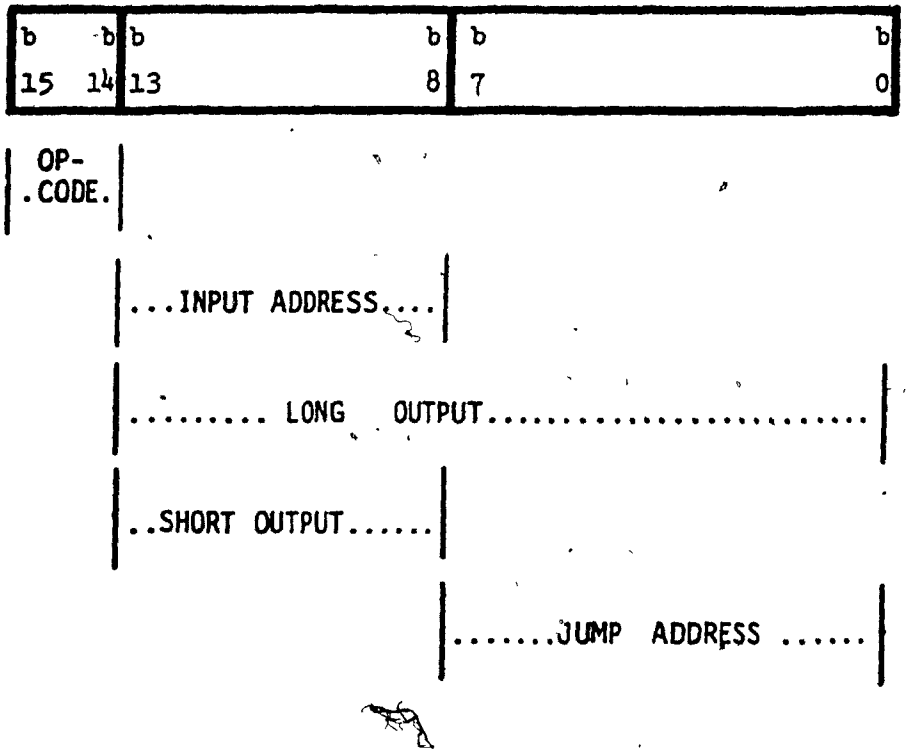


Fig. 4.6: The BD Instruction Format.

4.1.4 The Operational Modes

The BD processor operates in several modes to enable program loading, verification, execution and debugging.

Program Loading: A BD program describing control logic may be loaded into the processor memory via the 16 manual data switches. The Auto Load Enable (ALE) Switch must be in 'off' position to disable unintentional program counter loading. The operator can preset the instruction address to either 0 by clearing the program counter or to some other desired value by setting data switches 0-7 to the desired address and allowing PC load via ALE - ON.

Program Verification: Once a BD program is entered into the memory, it can be read and verified before execution. This is done by disabling the data switches, and stepping through the memory using the Single-Step toggle switch with the R/W switch in Read position. The instruction LEDs will display the stored instruction.

Program Execution: To execute BD programs stored in the memory the data switches must be disabled (DE - OFF),and the DLU must be activated via ALE - ON. Execution can be controlled by the BD clock with two optional frequencies: 70KHz or 0.5Hz. The latter is used to monitor program execution, instruction by instruction, at a rate conveniently perceptible to the user. Manual execution is possible by disabling the clock and activating the Single-Step toggle.

4.2 The Hybrid Concept

The Hybrid scheme, Fig. 4.7, consists of one or more BD processors connected in a network with a microprocessor (mP). The prototype BD processor was originally designed as a manually controlled unit. Programs were loaded and executed via a set of switches. In the Hybrid scheme, the BD processor is interfaced to the mP via the mP-BD interface module. BD programs, data and control signals are transmitted via a BD-mP bus connecting the BD processor to the mP-BD interface module. The mP system consists of a conventional processor, memory, bulk storage and peripheral I/O devices. The processor and all other elements of the mP system communicate via the mP bus.

Process control tasks are often classified as:

1. repetitive on/off (binary) tasks,
2. computational, optimization, data manipulation tasks, and
3. external communication tasks.

The BD processor is best suited for the first class of operations. It can effectively monitor process parameters and react to changes quickly, in a reflexive manner. Complementing the BD processor, a high level mP handles the other tasks better. In the hybrid scheme, one or more BD processors are monitored and supervised by the mP. Under normal conditions they operate autonomously, like the stand alone version. The mP-BD combination offers advantages such as a more user friendly interface to the BD, distributed processing and a BD library in mP memory from which programs can be dynamically downloaded into BD memory. This results in a powerful,

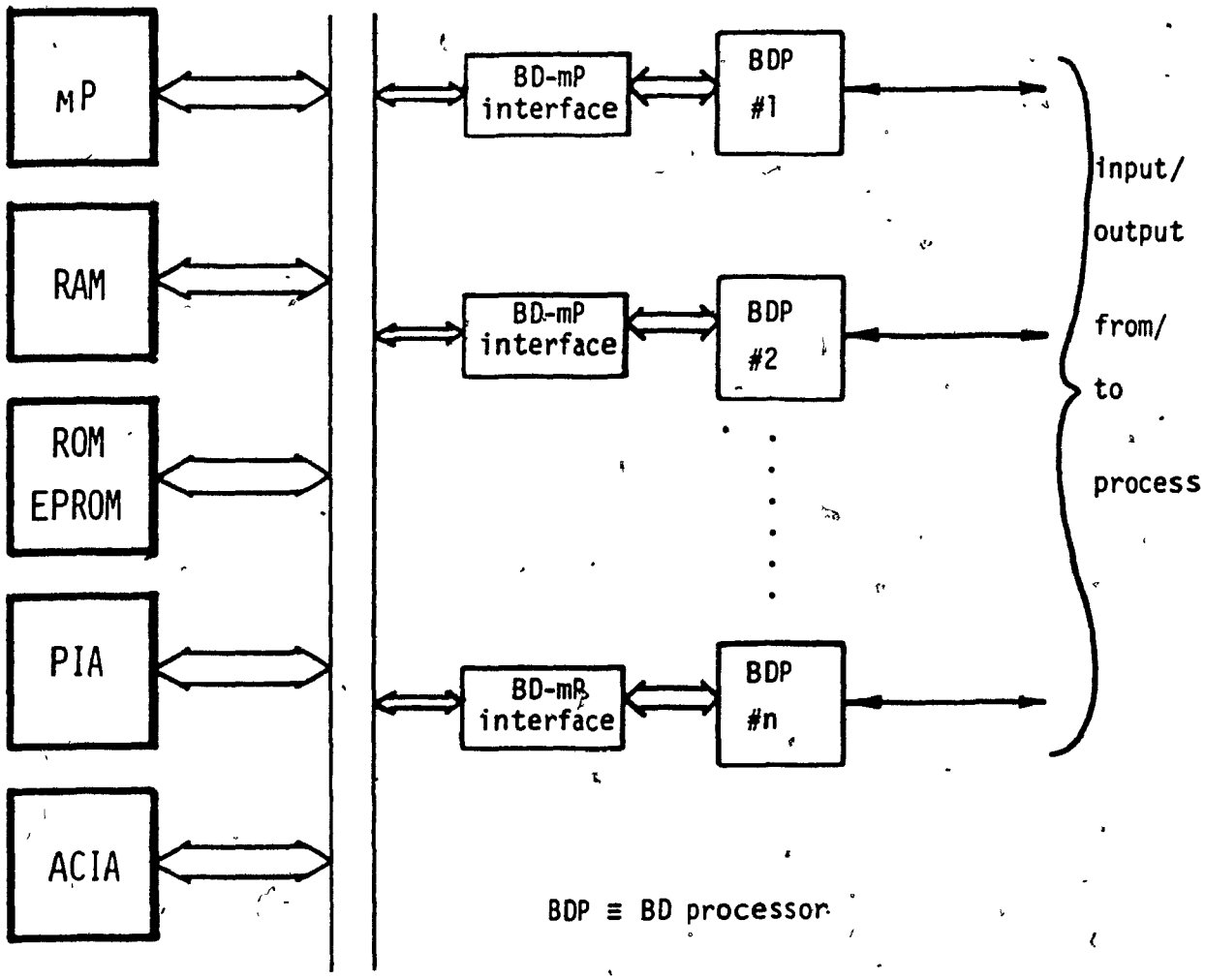


Fig. 4.7: The Hybrid Scheme - Block Diagram.

flexible and inexpensive system which is superior to conventional microprocessor control.

The Hybrid scheme needed:

1. hardware enhancements to the BD processor and implementation of a larger instruction set,
2. development of the interface module, and
3. development of the necessary software tools.

Hardware enhancements to the BD processor included a larger instruction set, handshaking logic to handle the communication protocols between the mP and the BD and more input and output lines arranged in banks with bank selection logic. Special logic ensures compatible BD operation in both the original (Manual) mode and in the mP-controlled (Auto) mode as well as smooth switch-over between the modes.

The mP-BD interface module was developed to handle the bi-directional communication of data and control signals between the mP and the BD processor. It is based on the Parallel interface method which employs a MC6821 Parallel Interface Adaptor (PIA).

Software was developed to operate the BD Hybrid, i.e., an operating system to enable the mP to control the BD and to transfer data. This includes the management of the BD program library and utilities for operator interaction. In addition a compiler which generates minimized BD programs, and an assembler for coding BD programs, were developed.

4.3 The Expanded BD Machine

The prototype BD processor was modified to allow it to operate under mP control. The design retained all existing manual controls so that the original BD machine remained a subset of the Hybrid system.

4.3.1 Hardware Expansion

4.3.1.1 The Expanded BD Processor - Overview

The enhanced BD processor, shown schematically in Fig. 4.8, is described using Instruction-set processor (ISP) notation in Table 4-2. The program counter, the instruction word size, and the program memory are unaltered. The number of input variables was increased from 64 to 256 and organized in input banks. Similarly, the number of output variables was increased from 14 to 192 and organized in output banks. Bank select logic was implemented for both input and output sections. The expanded Clock and Interrupt Logic (CIL) allows BD or mP initiated interrupts, controls program execution and provides bidirectional control and status signals. A new operation-code decoder was implemented to handle the larger instruction set. The original manual BD processor control switches were retained, interfaced in parallel with the corresponding data and control lines of the mP-BD bus. Auto/-Manual select logic was added to enable operation in either mP control mode or Manual control mode. The Auto/Manual select logic avoids control ambiguities during mode transition.

Instruction	\	IR<15:0>
Program Memory	\	Mp[255:0]<15:0>
Program Counter	\	PC<7:0>
Instruction Address	\	IA<7:0> := PC<7:0>
Input Variables	\	IV<15:0>
Input Bank	\	IB[15:0]<15:0>
Output Variables	\	OV<11:0>
Output Bank	\	OB[15:0]<11:0>
Selected Input Varia.	\	X := IV<IR<11:8>>
Input Bank Register	\	IBr<3:0> := IR<11:8>
Output Bank Register	\	OBr<3:0> := IR<11:8>
Clock & Interrupt Logic	\	CIL<3:0> := IR<15:12>

Table 4-2: The Expanded BD Procesor - ISP Definition.

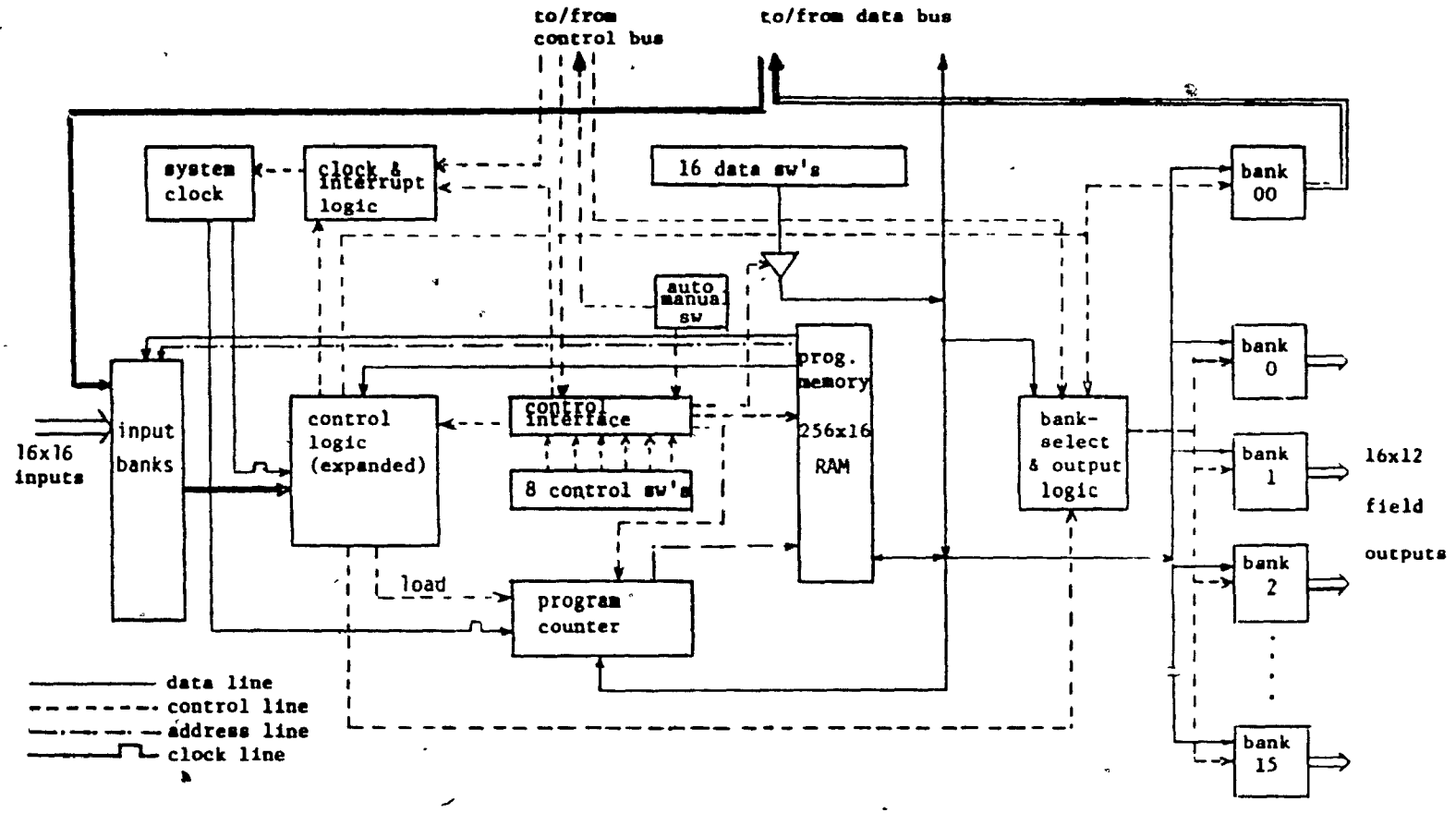


Fig. 4.8: The Expanded BD Processor - Block Diagram.

4.3.1.2 Expanded Input and Output banks, Selection Logic

To separate inputs from various processes, or different parts of one process, the 256 possible input lines are grouped into 16 input banks, IB[15:0]<15:0>. This increases input capacity and simplifies the instruction structure as only 4-bit input addressing is required. It does require the program to select the appropriate input bank, incurring a penalty of one clock cycle.

Field input lines are connected to 1 of 16 74LS150 data-selectors via opto-isolators, Fig. 4.9. The input line address, IR<11:8> of the input instruction, is connected to the four data-select inputs of all IB data-selectors. The 16 outputs generated by the 16 data-selectors are connected to a secondary 1 of 16 74LS150 data-selector.

The address indicated by IR<11:8> of the input bank instruction, is latched into the Input Bank Register IBr<3:0> when this instruction is executed. The latched address is connected to the four data-select inputs of the secondary 74LS150 thereby selecting the line to be tested by the DLU.

The 16 output banks OB[15:0]<11:0> accommodate 12 parallel outputs OV<11:0> each. The output variables are latched into 74LS74 FFs by the output instructions. An output bank with the select logic is shown in Fig. 4.10. The selected bank is indicated by the output bank register OBr<3:0>, during the execution of the output bank select instruction (IR<11:8> => OBr<3:0>). The latched address is connected to the four binary-coded inputs of two 4-to-16 demultiplexers. The activated output of each demultiplexor

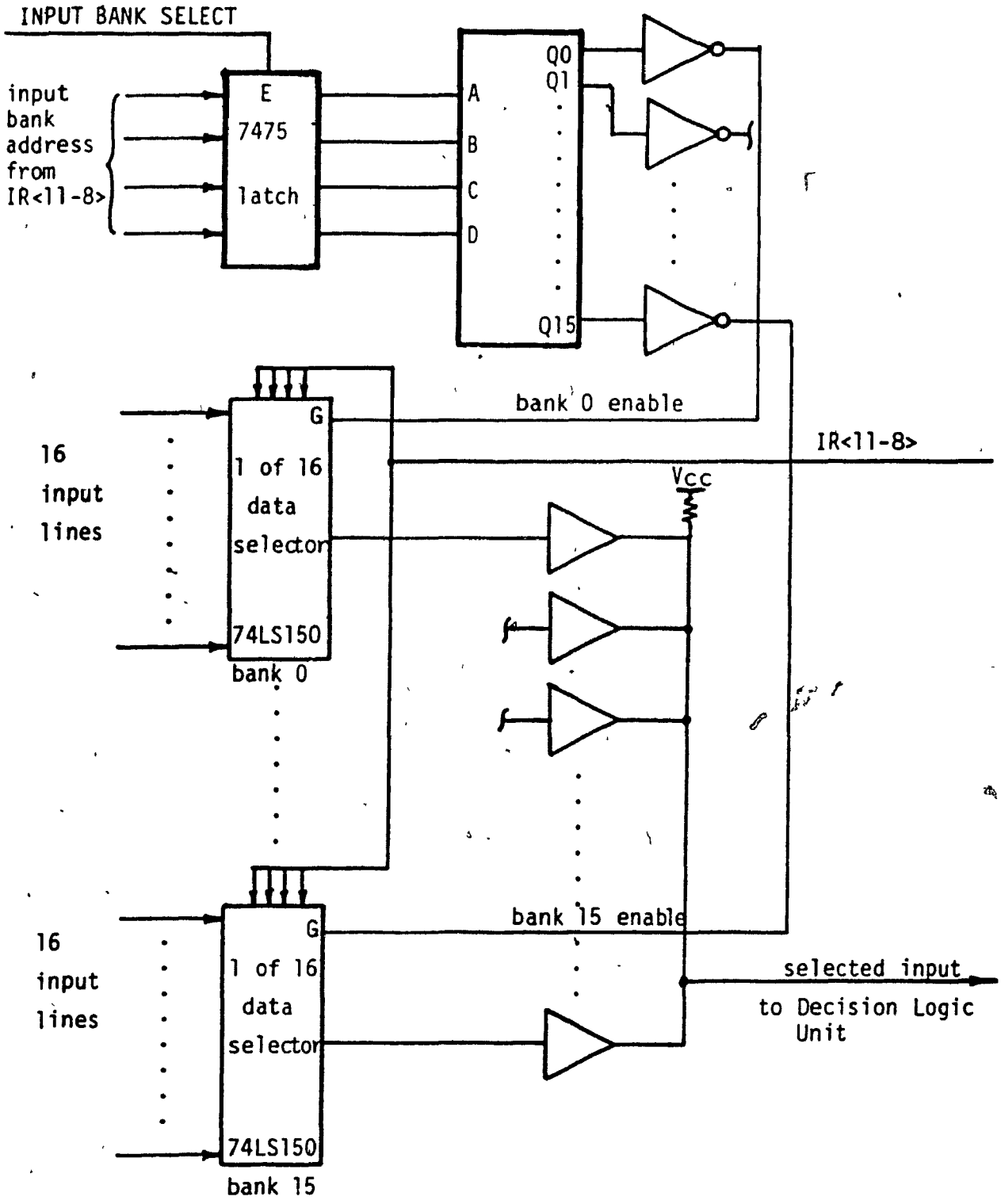


Fig. 4.9: The Expanded Input Section.

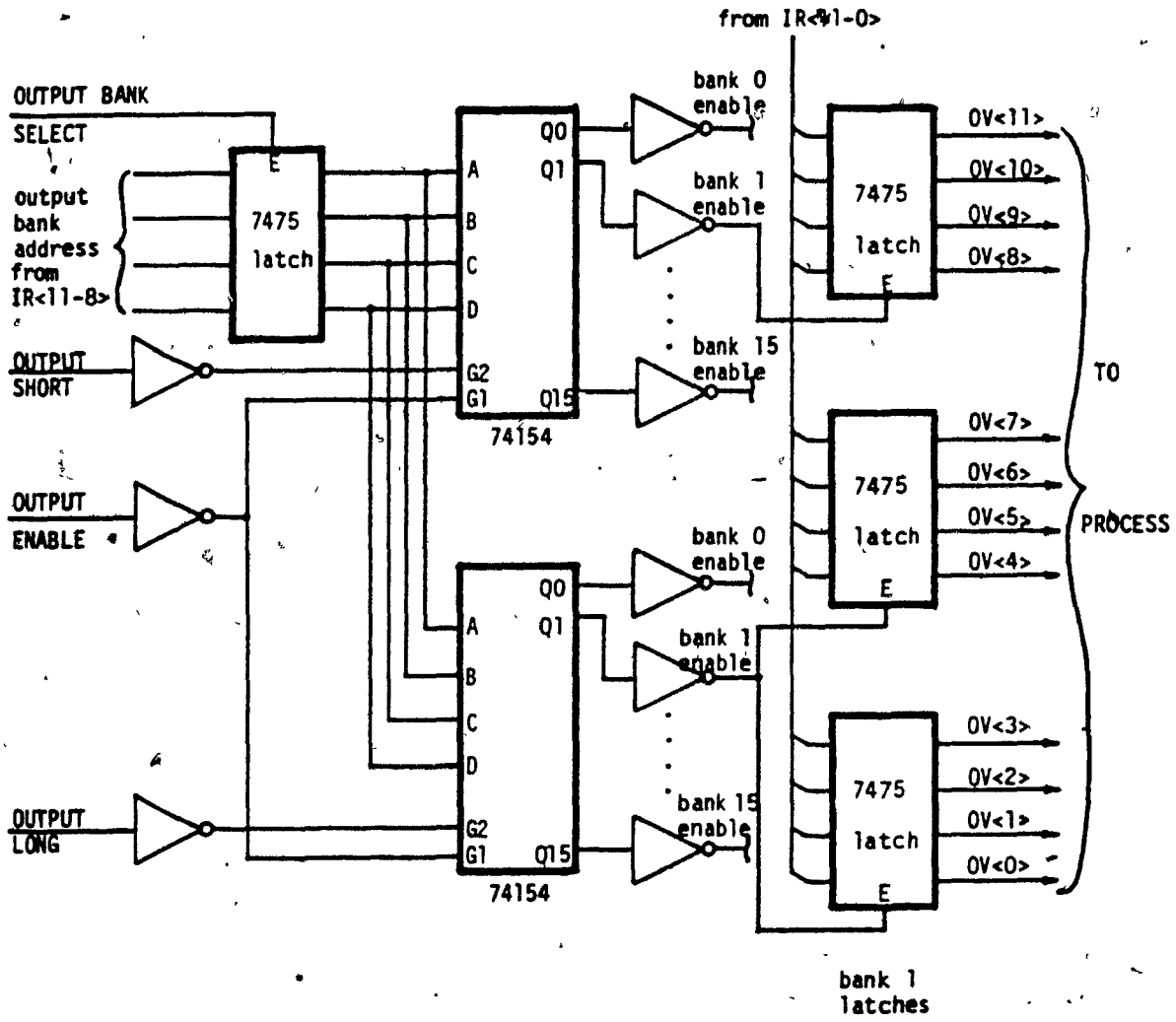


Fig. 4.10: The Expanded Output Section.

serves as the enable signal to the upper (OV<11:8>) and lower (OV'7:0>) sections of the output bank. The upper demultiplexer is enabled during a short output instruction while both are enabled during the long output instruction. The OE (Output Enable) signal serves as a second enable signal to prevent output alterations unless the BD machine is executing a program.

4.3.1.3 The Clock and Interrupt Logic

The Clock and Interrupt Logic (CIL) is shown in Fig. 4.11. The CIL handles BD and mP interrupt functions, operator interrupts, the wake-up and BD monitoring functions. By means of this logic both the mP and the BD are able to gain each other's attention. As a slave processor, the BD is always monitored by the mP via the CIL in order to detect BD failure. Using the wake-up circuit of the CIL the mP is able to restart the BD processor.

When an interrupt instruction is executed the BD generates an interrupt that is transmitted to the mP via the bus interrupt line CA1 (see interfacing section), while toggling a J-K Flip Flop ($\overline{Q}$ -> L). Upon receiving an interrupt, the mP completes its current execution cycle(s) and jumps to an interrupt handling routine which identifies the type of interrupt and saves the current mP registers on the system's stack. It then executes a polling routine to identify which device issued the interrupt and upon detecting a BD interrupt it responds by raising the STOP flag. Control is then passed to BD09 to service the interrupt (see section

4.6.1). The MC6809 processor may require about 40-50 cycles before responding to the BD interrupt. BD interrupt instructions normally branch to the EOP instruction of the program. As the mP is unable to detect and identify the interrupt and raise the STOP flag within this time frame, the toggle FF action prevents any subsequent EOP instruction from stopping the BD machine until the interrupting program is re-executed, the interrupt is repeated and the J-K Flip Flop is toggled again ($\overline{Q}$ -> H). This results in a low signal at the D input of a D Flip Flop. The next EOP instruction, clocks this Flip Flop which disables the BD clock. The logic fulfills three requirements:

1. The mP acknowledges the interrupt signal.
2. The interrupting BD program and all other programs complete their normal execution.
3. The handshaking cycle is completed by disabling the $\overline{\text{BD-CLK-ON}}$ signal, informing the mP that the BD clock is stopped.

The mP interrupts the BD processor by simply raising the STOP flag. The D input of the D Flip Flop is then triggered and similar to the second part of the BD interrupt cycle, the next EOP signal clocks the D Flip Flop disabling the BD clock. This logic ensures that the interrupted BD program completes its normal course of execution before the BD processor is halted. The $\overline{\text{BD-CLK-ON}}$ signal acknowledges the stopping of the clock.

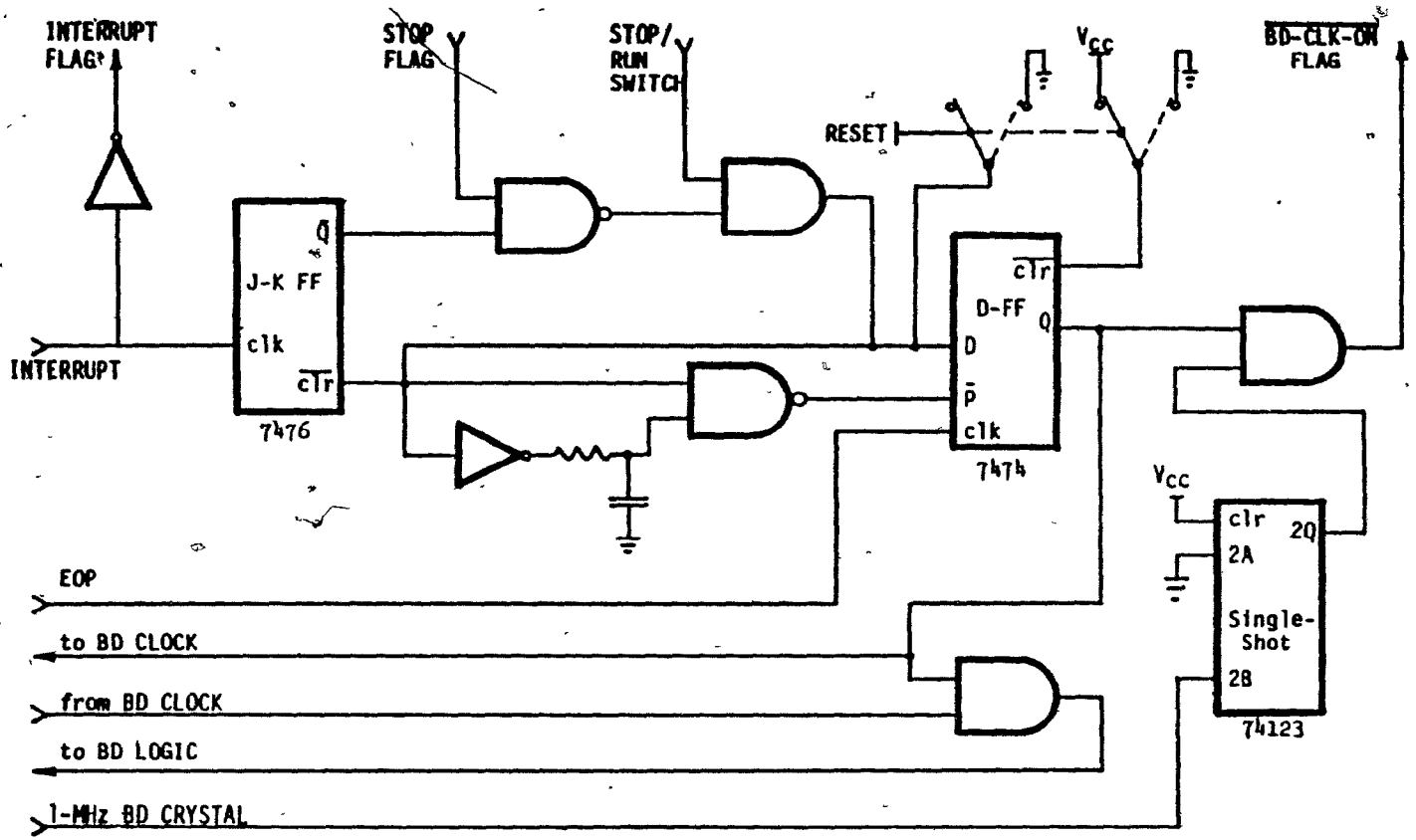


Fig. 4.11: The Clock and Interrupt Logic.

The timing diagram for the BD "wake-up" circuit is shown in Fig. 4.12. The edge triggered circuit connected to the $\overline{P}$ input of the D Flip Flop is used by the mP to start the BD processor. This "wake-up" circuit senses when the STOP flag is lowered by the mP and provides a short (150 ns) HLH pulse presetting the D Flip Flop and thus enabling the BD clock.

The $\overline{\text{BD-CLK-ON}}$ signal is generated by a retriggerable monostable multivibrator. It informs the mP that the BD processor is in running mode, i.e., whether or not the clock is enabled. This signal is active if:

1. the BD clock circuit is functional,
2. the Manual stop line is deactivated, and
3. no interrupts have occurred.

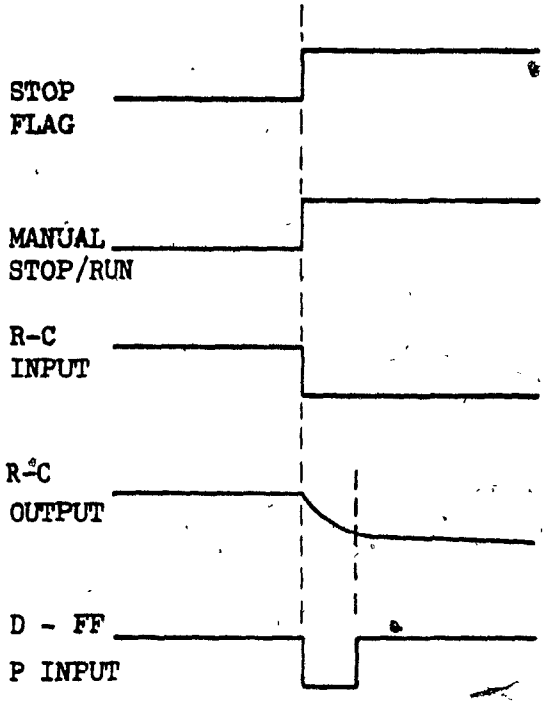


Fig. 4.12: BD "Wake-up" Circuit - Timing Diagram.

By switching to the Manual mode (see Auto/Manual logic) and activating the Stop/Run switch, the operator can manually halt or start the BD processor. The manual Reset button clears the D Flip-Flop.

4.3.1.4 Expanded Operation-code Decoder

The operation-code decoder examines bits IR<15:12> synchronized with the BD clock to generate the required signals for each of the implemented BD instructions.

Input Testing instructions are interlocked with the Auto Load Enable (ALE) to prevent execution when downloading programs (see operating modes section). AND and NAND gates are employed to decode the additional instructions as shown in Fig. 4.13.

Instruction execution requires one clock cycle. Fig. 4.14 depicts the execution cycle of a BD instruction in relation to the rising and falling edges of the BD clock pulse. An instruction cycle is initiated with the rising edge of the BD clock. A 74123 monostable multivibrator generates an unconditional clock strobe interlocked with IR<15> and IR<14> of the instruction word and synchronized with the falling edge of the BD clock. All generated outputs are activated by the strobe.

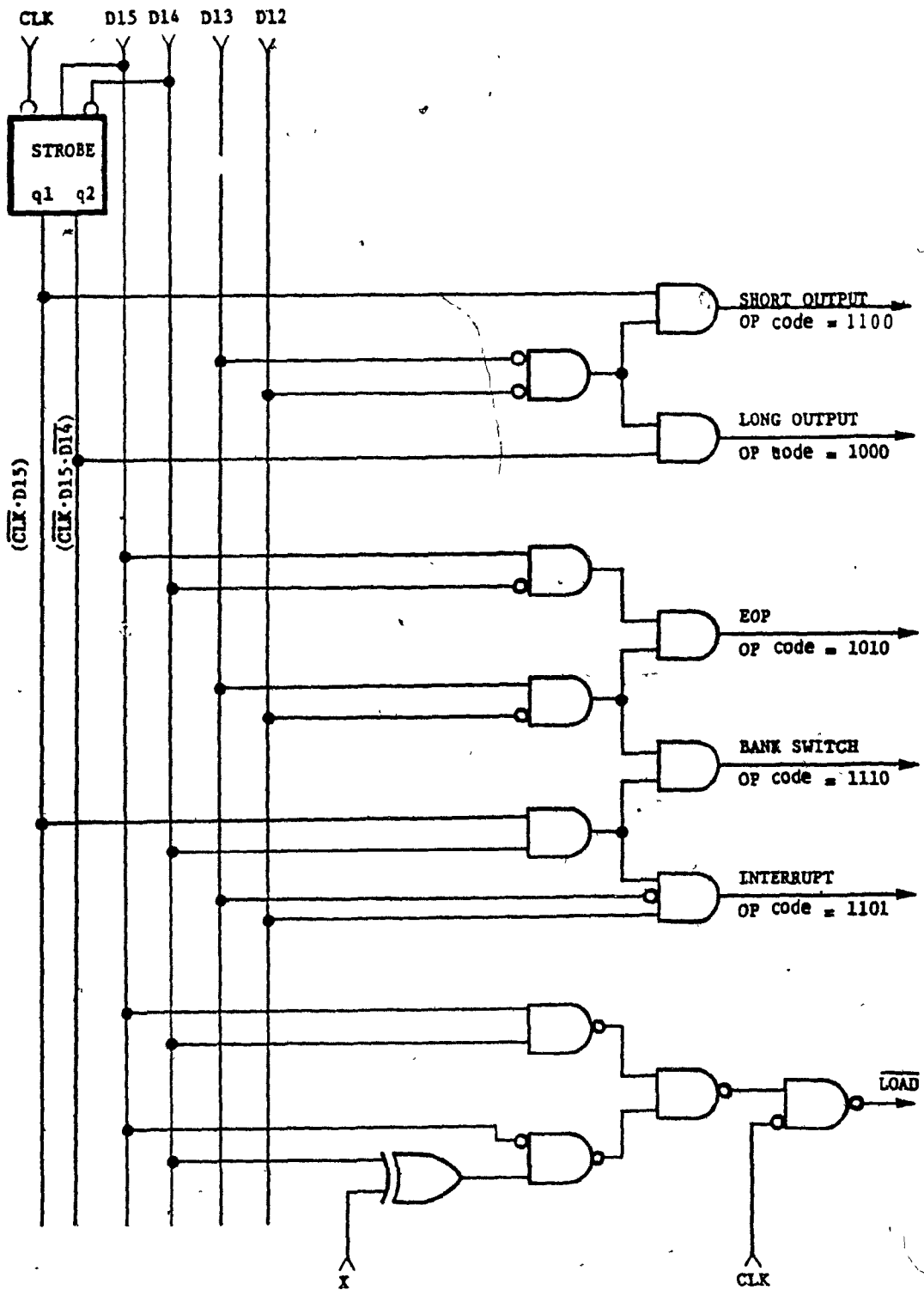


Fig. 4.13: Operation-code Decoder Block Diagram.

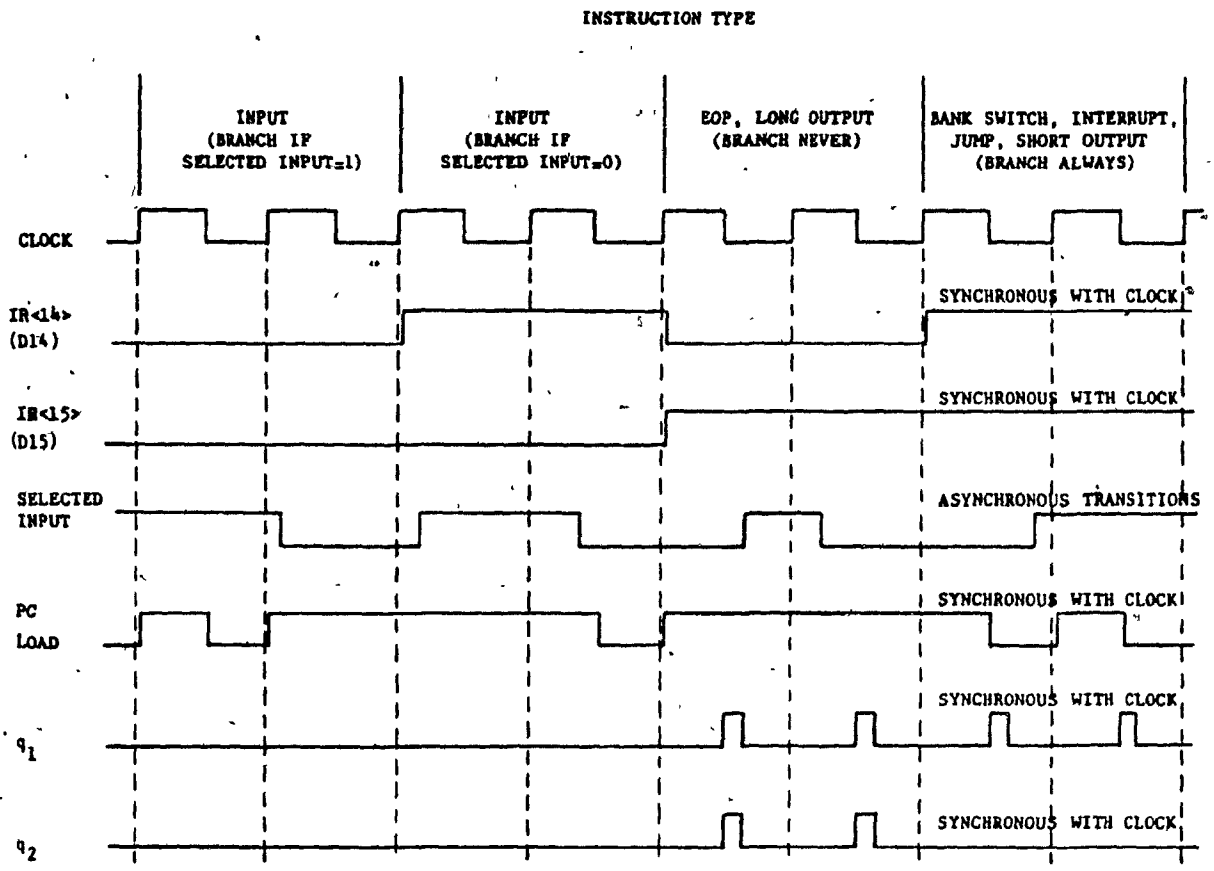


Fig. 4.14: Decoder Timing Diagram.

4.3.1.5 Auto/Manual Select Logic

The Auto/Manual Select Logic (AML) is schematically shown in Fig. 4.15. The BD system can be operated fully in Manual mode. The AML allows smooth switching-over from one mode to the other by toggling a single Auto/Manual selector (A/M) switch causing no interruption to the BD operation.

When Manual mode is selected, the AML disables the mP control and data signals and enables the manual control and data switches and vice-versa when Auto mode is selected. The mP control signals are connected to the BD logic via 74LS244 tristate drivers whose enable lines are directly connected to the A/M switch. The tristate outputs are connected to a set of 2-input AND gates. Each buffered mP control signal is connected to one AND gate input while the equivalent Manual signal goes to the other.

In Auto mode all switches are inoperative. The normally-low input to each of the switches is driven high by the A/M switch and thus, regardless of switch position their outputs are high and the mP signals appear at the AND gate outputs. In Manual mode the mP signal tristates are disabled and their outputs are driven high and the signals from the manual switches appear at the AND gate outputs.

The A/M switch output is also used in the mP-BD interface module to select the appropriate data tranceivers I/O state. In Manual mode no data transfer can occur between the mP and the BD because the Enable signals to the data channels drivers in the interface module are kept high. In the Auto mode manual data switches are disabled because the Data Enable Switch output is similarily forced high.

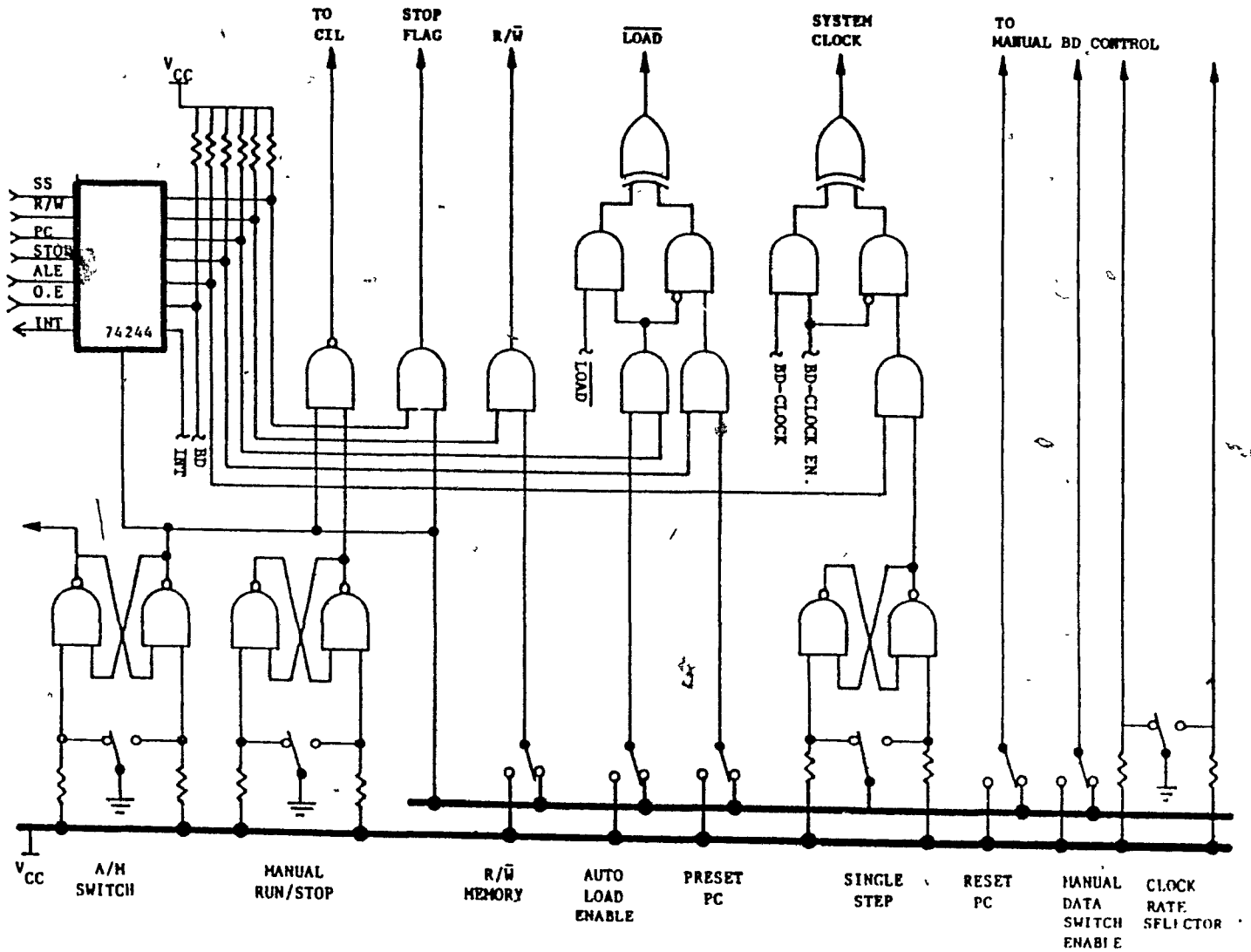


Fig. 4.15: Auto/Manual Select Logic.

4.3.2 Enhanced Instruction Set

4.3.2.1 Overview

Five new instructions were added to the original BD instructions and the four existing ones were slightly modified. The new set is shown in Fig. 4.16 in a Karnaugh map form. The operation-code was extended from two bits to four bits: IR<15:12>. The input variable address is now four bits only: IR<11:8>, instead of the original six bits. Output instructions were also modified. A short output consists of four data bits: IR<11:8> the long output consists of 12 bits IR<11:0>. Input bank and output bank addresses are coded in the 4-bit segment: IR<11:8>. The jump address of the next instruction remains in IR<7:0>.

All BD instructions are summarized in Table 4-3. They may be divided into three groups:

1. input instructions,
2. output instructions, and
3. control instructions

Input instructions include an input bank select instruction - IBS, for selecting the active input bank, and two instructions - IN0 and IN1 for testing input lines. Output instructions include an output bank select instruction - OBS, which selects the active output bank, and two instructions - OPS and OPL, which assign parallel output values to process parameters. Three control instructions were implemented: the interrupt instruction - INT, by which the BD gains mP atten-

tion, the end-of-program instruction - EOP, used as the last instruction of each program to ensure complete execution of critical instruction sequences after an interrupt has occurred, and the branch instruction - BRA, an unconditional jump.

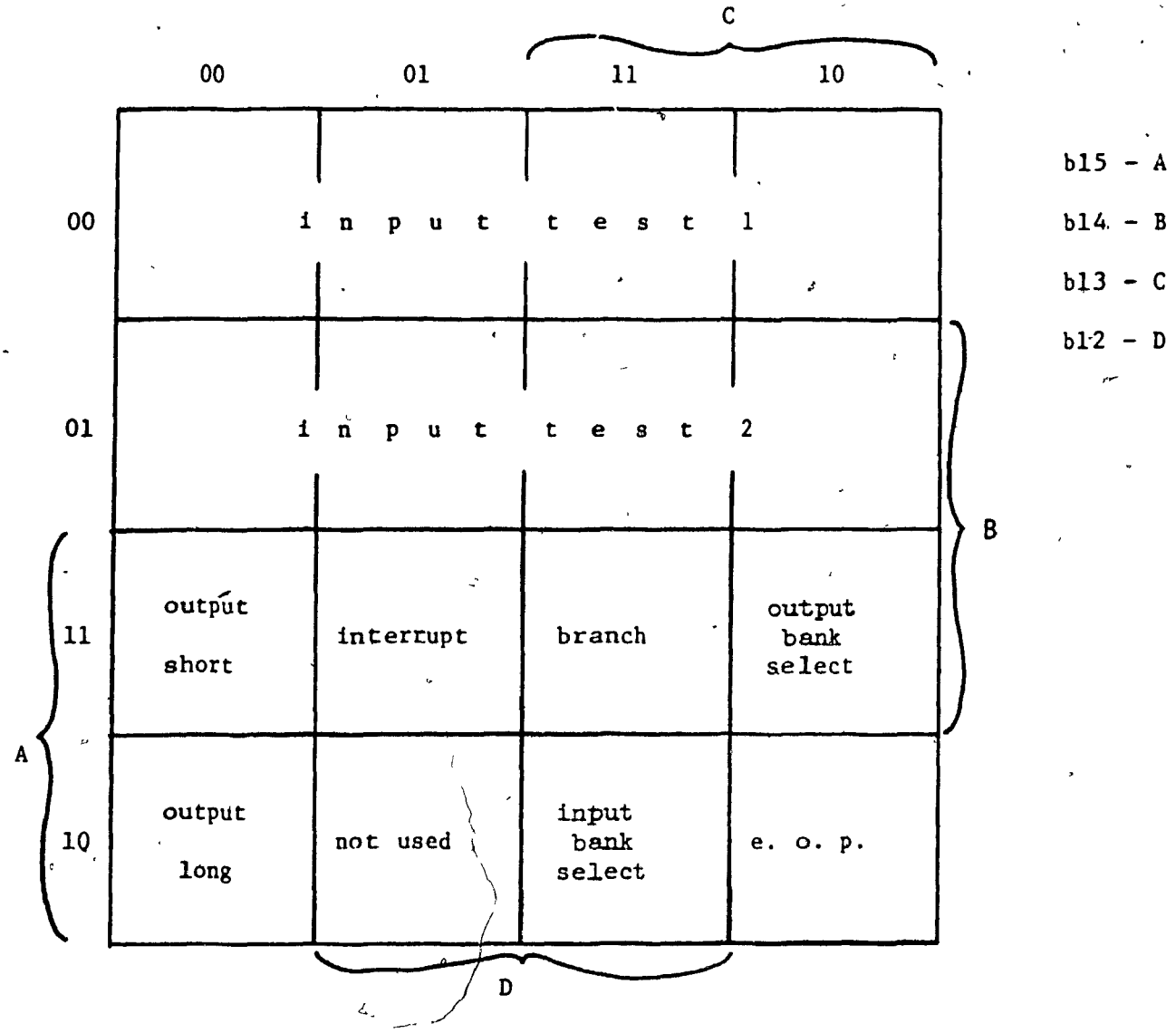


Fig. 4.16: The Extended BD Instruction Set.

op-code	name	description	operand	jump address
00xx	IN0	test input line IN jump if IN=0	IN address	yes
01xx	IN1	test input line IN jump if IN=1	IN address	yes
1011	IBS	input bank select	IB address	yes
1100	OPS	output 4 parallel bits	4 OV	yes
1000	OPL	output 12 parallel bits	12 OV	no
1110	OBS	output bank select	OB address	yes
1101	INT	interrupt to mP	none	yes
1010	EOP	end of program	program info.	no
1111	BRA	unconditional branch	none	yes

Table 4-3: The Enhanced BD Instruction Set.

4.3.2.2 Input Instructions

IBS - Input Bank Select: A BD processor accommodates up to 16 input banks. The IBS instruction, Equation 4.3, supplies four high order address bits IR<11:8>, i.e., the bank address. There must be at least one IBS instruction in each BD program. IR<7:0> points to the next instruction. All subsequent IN1 and IN0 instructions address one of 16 input lines in the bank selected by the most recently executed IBS. Input lines in another bank may be tested after executing another IBS.

$$OP\langle 3:0\rangle = 1011 \Rightarrow \quad IBr\langle 3:0\rangle \leftarrow IR\langle 11:8\rangle \qquad (4.3)$$
$$PC \leftarrow Mp[IR\langle 7:0\rangle]$$

IN0, IN1 - Input Line Test: IN0 and IN1 represent the BD node operation where a binary input value determines which one of two possible branches will be followed. The tested input line is addressed by IR<11:8>. The next executable instruction is defined by the selected input value X and may be either the next sequential instruction or the one pointed to by IR<7:0>. If X is 0 then the next executable instruction will be the one in PC+1 for IN0 and the one in IR<7:0> for IN1. If X is 1 the reverse is true. The above logic is achieved via the DLU controlled program counter and is described by Equation 4.4. When the PC load line - Lpc is high the PC output is incremented by one and when the Lpc is low it is preset to the va of its inputs which are connected to IR<7:0>. Equation 4.4 describes the relations between b14, b15 of the op-code,

the IV value, the ALE signal, the BD clock, and the Lpc.

$$IR\langle 15\rangle=0 \Rightarrow Lpc = (IR\langle 14\rangle \oplus X)\cdot ALE\cdot CLK \qquad (4.4)$$

$$Lpc = 0 \Rightarrow PC \leftarrow Mp[IR\langle 7:0\rangle]$$

$$Lpc = 1 \Rightarrow PC \leftarrow PC + 1$$

$$\text{where } X = IV\langle IR\langle 11:8\rangle\rangle$$

4.3.2.3 Output instructions

OBS - Output Bank Select: The BD processor accomodates up to 16 output banks, each with 12 output lines. Similar to input bank selection, the OBS instruction, Equation 4.5, activates an output bank. An OBS must occur at least once in any BD program with outputs. IR<11:8> indicates which of the 16 banks becomes active. IR<7:0> points to the next instruction to be executed. All subsequent OPS and OPL instructions output their parallel output values to the active bank only.

$$OP\langle 3:0\rangle = 1110 \Rightarrow OBr\langle 3:0\rangle \leftarrow IR\langle 11:8\rangle \qquad (4.5)$$

$$PC \leftarrow Mp[IR\langle 7:0\rangle]$$

OPS, OPL - Output Short and Output Long: Each of the output banks can issue 12 OVs to the process. An output instruction assigns the OVs in parallel. OPS, Equation 4.6, assigns values to the upper four bits of the output bank - IR<11:8>, the lower eight OVs are not affected. The next executable instruction is pointed to by IR<7:0>. OPL, Equation 4.7, assigns all

12 bits in the bank - IR<11:0>. The next executable instruction is the next sequential one in memory.

$$OP\langle 3:0\rangle = 1100 \Rightarrow \quad OV\langle 11:8\rangle \leftarrow IR\langle 11:8\rangle \qquad (4.6)$$
$$PC \leftarrow Mp[IR\langle 7:0\rangle]$$

$$OP\langle 3:0\rangle = 1000 \Rightarrow \quad OV\langle 11:0\rangle \leftarrow IR\langle 11:0\rangle \qquad (4.7)$$
$$PC \leftarrow PC + 1$$

4.3.2.4 Control Instructions

INT - BD Interrupt: The interrupt instruction, Equation 4.8, is used by the BD processor to gain the attention of the microprocessor. Such an interrupt is designed to occur in one of the following cases:

1. Process conditions require another BD program to be down-loaded.
2. BD tree is large and the boundaries of a memory resident segment were reached thus a request to download the next segment is issued.
3. Process conditions require mP intervention, alarm signals to operator or other tasks which the BD processor cannot perform.

IR<7:0> points to the next executable instruction which normally is EOP.

$$OP\langle 3:0\rangle = 1101 \Rightarrow \quad CIL\langle 3:0\rangle \leftarrow IR\langle 15:12\rangle \qquad (4.8)$$
$$PC \leftarrow Mp[IR\langle 7:0\rangle]$$

EOP - End of Program: The EOP instruction, Equation 4.9, is a logical termination of a BD program. It is used to prevent incomplete program execution in cases when the program issues an interrupt to the mP or is interrupted either by the mP or manually by the operator. IR<11:0> are used to indicate to the mP necessary information pertaining to the BD program and possible interrupts. These can be transferred to the mP via an output bank (conventionally, bank 0) directly to a PIA input. The EOP instruction provides the control signal required by the CIL and does not affect any other BD logic or output signals. The next executable instruction is the next sequential one. In a sectored memory structure, if EOP is not the last instruction in the sector, the next sequential instruction will normally be a jump to the first instruction in the next memory sector.

$$\begin{aligned} OP\langle 3:0\rangle = 1010 \Rightarrow \quad & CIL\langle 3:0\rangle \leftarrow IR\langle 15:12\rangle \\ & OB[0]\langle 11:0\rangle \leftarrow Mp[IR\langle 11:0\rangle] \\ & PC \leftarrow PC + 1 \end{aligned} \qquad (4.9)$$

BRA - Unconditional Branch: The branch instruction, Equation 4.10, is unconditional. The next executable instruction is pointed to by IR<7:0>. The execution of this instruction does not alter any outputs nor active banks. It is used to transfer program control when the previous instruction does not have this capability e.g., OPL, or after an input instruction where both branches require jumps.

$$OP\langle 3:0\rangle = 1111 \Rightarrow \quad PC \leftarrow Mp[IR\langle 7:0\rangle] \qquad (4.10)$$

4.4 The mP-BD Interface

4.4.1 Interfacing Methods Overview

Interfacing is defined as connecting one device in a system with another so they become one operational unit. A microprocessor system consists of the CPU which communicates with all other system components via interface units. System components are memory modules, bulk storage devices, terminals, printers, additional processors, BD machines, etc. The CPU is the master device and all controlled components are slave devices. Three common interfacing schemes are the 'parallel', the 'serial', and the 'direct' interfaces. The three methods differ from each other in their hardware complexity, software requirements and transfer characteristics. Interfacing the BD processor to a mP can be done in any one of the three ways.

Parallel Interface

The 'parallel' interface employs a number of data lines, one line per bit of some multibit unit, e.g., a byte. The mP receives or transmits all bits of a data unit, on the communication bus data lines, simultaneously. The Motorola MC6800 family of 8-bit microprocessor product includes the MC6821 which is a dedicated N-MOS parallel interface device known as a PIA (Parallel Interface Adaptor). The PIA contains

two bi-directional 8-bit registers which are connected to the outside world and can be accessed by the mP. It can transfer interrupt signals and has extensive programmable capabilities to facilitate interfacing tasks [AW80]. Speed of transfer is very high but bus lines must be kept short, usually no longer than 20-30 feet.

Serial Interface

A 'serial' interface transfers data with one transmit line and one receive line and a number of control lines. All bits are transferred serially and therefore both the sender and the receiver must have serial to parallel and parallel to serial conversion logic. Serial interface chips are available for almost any mP family. A good example is the MC6850 ACIA (Asynchronous Communication Interface Adaptor). A serial interface can transmit data over longer distances and with less cabling than a parallel one. Physical lines may extend up to one hundred feet, telephone lines, including microwave links, may be used to connect devices which are miles apart. A serial interface is not suitable for very high speed applications. Serial interfaces are commonly used to interface operator terminal devices and some have gained world standardization. The most common serial standard is the RS232C which is specified to operate at speeds from 110 to 19200 serial bits per second asynchronously. Synchronous serial interfaces are faster. The MC6852 SSDA (Synchronous Serial Data Adapter) supports transfer rates of 1.5 Megabits per second. This interface is desirable in high performance data

links and incorporates buffering to allow some synchronization error margin.

Direct Interface

A device connected to the mP bus, without interface logic, is said to be interfaced 'directly'. The mP bus is a set of address, data, control and power lines arranged in a more or less systematic way that operates under strict communication protocols. A direct bus communication is preferrable whenever possible due to standardization of communication, accessibility, speed, and modularity. The slave device must be able to decode bus signals in the same way as interface modules do. A device which is directly connected to the bus may be accessed by the mP via a single instruction. Direct memory transfers may be easily implemented and total system flexibility is increased. Implementation of DMA methods implies that the slave device can exercise control over the bus at certain times. This provides flexibility by sharing memory resources.

4.4.2 The mP-BD Parallel Interface

The parallel interface scheme was chosen for the Hybrid mP-BD scheme. The interface module shown in Fig. 4.17 employs a MC6821 PIA and an 8-bit write-only control register (CR) for transferring all necessary data and control signals to the BD processor. Both the PIA and the CR are under program control. They appear in the memory mapped I/O structure of the microprocessor system as five addresses.

Fig. 4.17a: The mP-BD Interface Module - Photo.

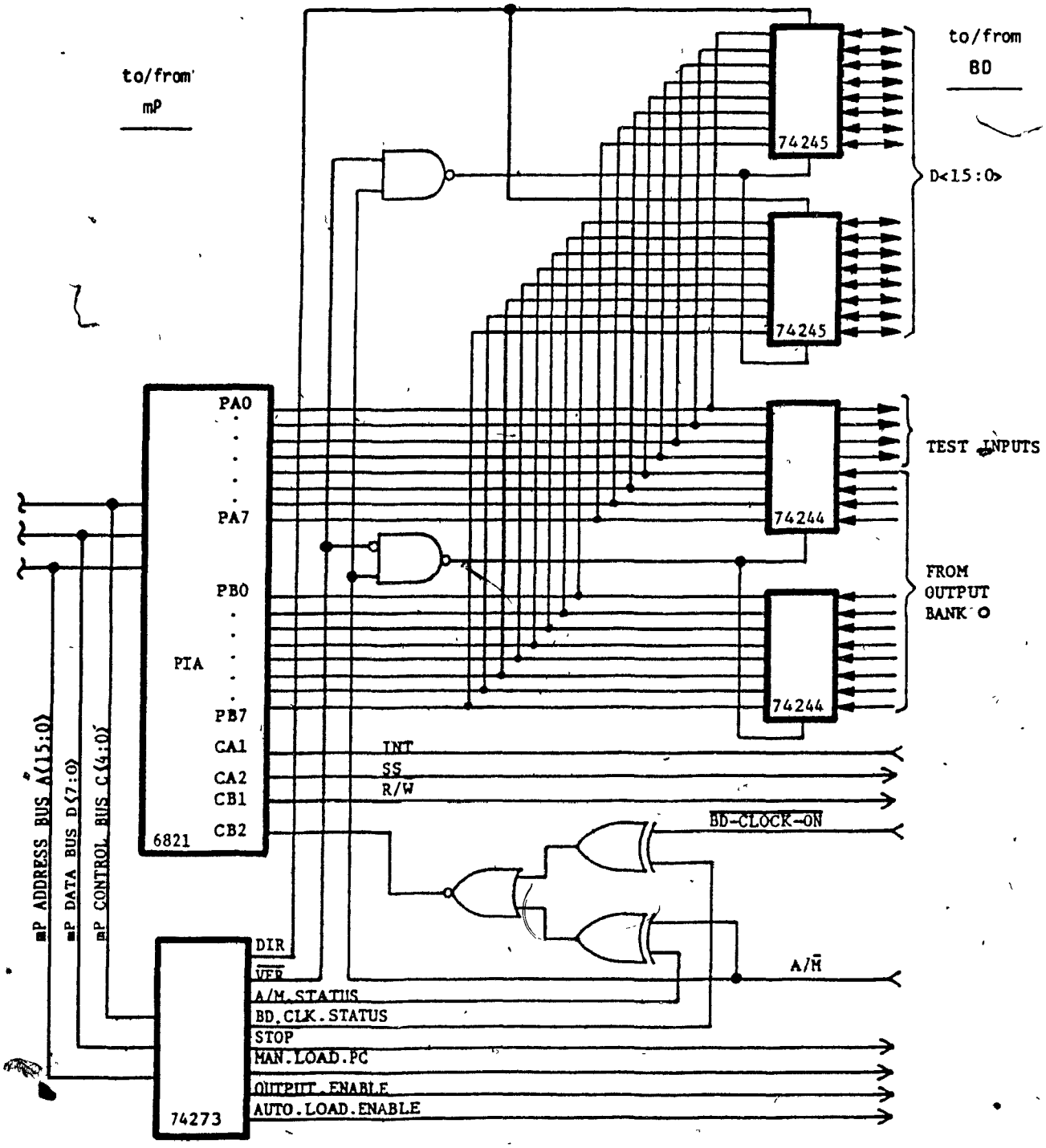


Fig. 4.17b: The mP-BD Interface Module - Schematic Diagram.

The formal ISP definition of the parallel interface is given in Table 4-4.

Bi-Directional Data Register	\	DR<11:0>
Uni-Directional Data Register	\	DR<15:12>
Write-Only Control Register	\	CR<7:0>
Interrupt Flag	\	INT
End of Program	\	EOP
Single Step Strobe (PIA)	\	SS
BD Memory R/$\overline{W}$ Strobe (PIA)	\	R/$\overline{W}$
BD Clock ON	\	$\overline{\text{BD-CLK-ON}}$
Buffer Enable Flag	\	EN
Data Direction Flag	\	DIR
BD Stop Flag	\	STOP

Table 4-4: The mP-BD Interface - ISP Definition.

Data Channels

There are two parallel 16-bit data channels between the mP and the BD processor. Data channel #1 is connected to two 74LS245, 16-bit bidirectional tranceivers between port A and port B of the PIA and the data inputs of the BD program memory. Data channel #2 employs two 74LS244 unidirectional tristate drivers, connected to port A and port B in parallel to the two 74LS245 of channel #1. This second channel transfers four test bits from the mP to the BD machine and the 12-bit output of bank 0 from the BD machine to the mP.

Control Channel

Four PIA control lines (CA1, CA2, CB1, CB2) and eight control register lines are used by both mP and BD processor to handle all communication tasks. Not all control signals generated by the mP are connected to the BD processor. Several are used to control the interface module. The mP controls the data direction of data channel #1 and the activation of either one of the data channels via CR<0> and CR<1>. The mP STOP flag, used to interrupt the BD processor and to respond to BD interrupt, is issued on CR<4>. The BD interrupt line connects to CA1 and $\overline{\text{BD CLK ON}}$ to CB1. The SS (Single-Step), R/$\overline{\text{W}}$ (BD memory read/$\overline{\text{write}}$), PC (preset program counter), OE (output enable), and ALE (auto load enable) are additional control signals which are generated by the mP and sent to the BD through CA2, CB2, CR<5>, CR<6>, and CR<7>, respectively. CR<2> and CR<3> are used internally by the interface module to indicate the current status of the Auto/Manual selector and

the BD Clock. The Auto/Manual Select signal indicating whether the BD processor is currently operating in Manual mode or under mP control, is transferred by the BD processor to the mP via CB1. A transition in either the A/$\overline{M}$ or the $\overline{\text{BD-CLK-ON}}$ generates a CB1 interrupt.

mP bus Connection

The Interface module is connected to the SS-30* bus, a sub-set of the SS-50* bus. The 8-bit data bus is connected to the data input of the PIA. mP bus control signals connected to the PIA are the R/$\overline{W}$, $\overline{\text{RESET}}$, $\overline{\text{IRQ}}$ and ENABLE. The upper 12 bits of the 16-bit address bus are decoded externally to the interface board into an $\overline{\text{I/O SEL}}$ line. The lower four bits are decoded on board to generate the four PIA addresses and the CR address.

4.5 Operating modes

The hybrid system can operate in one of the following modes:

1. Running
2. Program Downloading/Reading
3. Diagnostics/Verify

* This is a mP bus standard used by many MC680x systems. Detailed description is given in Appendix A.

Normally the system is operating in running mode and is monitoring/controlling an external process. The other modes are required to load and start programs and to execute diagnostic routines. All modes are initiated and controlled by the operating system via the communication PIA and the control register.

4.5.1 Running Mode

Initiation of the running mode is done via the start-up procedure. Lowering the STOP flag triggers the wake-up circuit incorporated in the Clock and Interrupt logic of the BD processor. The program-counter must be cleared or preset to a specific address by the microprocessor prior to start-up.

During running mode the STOP flag is kept low and BD programs are continuously executed on a cyclic basis. The Auto Load Enable (ALE) is kept high permitting BD instructions to affect the DLU which controls the PC Auto Load line. The Output Enable is kept high enabling output instructions to access the selected output bank. Data buffers on the interface board are in high impedance state preventing unintentional data transfers. The BD interrupt, $\overline{\text{BD-CLK-ON}}$, A/$\overline{\text{M}}$ selector, and the STOP line are the only active control lines between the mP and the BD. Interruption of the running mode can occur in the following cases:

1. To handle specific process conditions, the BD processor needs a new program or program segment.
2. The mP suspects a malfunction in the BD operation.

In both cases an interrupt is issued, the BD-mP bus becomes active as a result and data transfers can occur.

4.5.2 Program Downloading/Reading Mode

In these two modes data is transfered between the mP and the BD processor. The mP downloads a BD program to the BD memory by presetting the Program Counter to the first address of the program. The Data-dir. and Enable bi-dir. signals are set via the control register and data is transmitted by placing both MS and LS bytes in the PIA ports and strobing the R/W line. The SS line then increments the PC to the next sequential address. Output Enable is kept low during downloading to prevent any output alteration and ALE is kept low to prevent BD instruction execution.

In the reading mode the mP reads the BD memory contents. This may happen in one of the following cases:

1. A malfunction is suspected and the BD memory contents must be compared with a copy kept in the mP's BD program library.
2. An interrupt has occurred and the mP wishes to read the status of the interrupted program available in output bank 0.

All control signals are identical to those used in the downloading mode except that R/$\overline{W}$ and Data-dir are reversed.

4.5.3 Diagnostic/Verify Mode

A hardware test program, permanently residing in the top sector of the BD memory, is executed every cycle to detect primary hardware malfunctions. Upon such a detection the mP is interrupted and a complete diagnostic program is downloaded. The BD hardware can be checked by altering the mP controlled four test inputs and observing the resulting outputs in bank 0.

4.6 Software Tools

Three dedicated software packages were developed to operate the BD processor in the hybrid environment and to facilitate BD software development:

1. BD operating system (BDOS),
2. BD compiler, and
3. BD assembler.

The BD operating system is required to handle all hardware control functions that operate the two processors as one operational unit as well as BD software management functions. The BD compiler and the BD assembler are software development tools which eliminate the tedious task of coding BD programs directly. The compiler converts a truth table representation of the process into an optimized BD tree program. The assembler facilitates programing tasks by enabling the user to program with a symbolic instruction set rather than object code.

4.6.1 Operating System

The BD09 operating system, Fig. 4.18, is written in MC6809 assembler and works in conjunction with the mP Disk Operating System - FLEX09*. It resides in the mP memory and under normal running conditions is dormant. By means of interrupts the BD09 can be activated, in which case the interrupt is decoded and appropriate action follows. The BD09 contains the following modules:

1. Real Time Executive
2. Interrupt Decoder
3. BD Library Manager
4. BD Memory Manager
5. Hardware Drivers
6. Utilities.

Real-time Executive: The Real-time Executive manages the operation of the BD09. The mP recognizes a BD interrupt, sets the STOP flag and passes the control to BD09. The Real-time Executive executes the interrupt decoder to determine the source of interrupt. All interrupts are channeled via a single interrupt line, $\overline{IRQ}$, and treated on a 'first-in first-out' basis.

Interrupt Decoder: There are four types of interrupts: operator, BD program, hardware malfunction, and mP interrupt. The decoder decodes which type of interrupt occurred and polls any additional information associated with the interrupt. The

*TradeMark of Technical Systems Consultants Inc.

control is then passed to the appropriate interrupt handler.

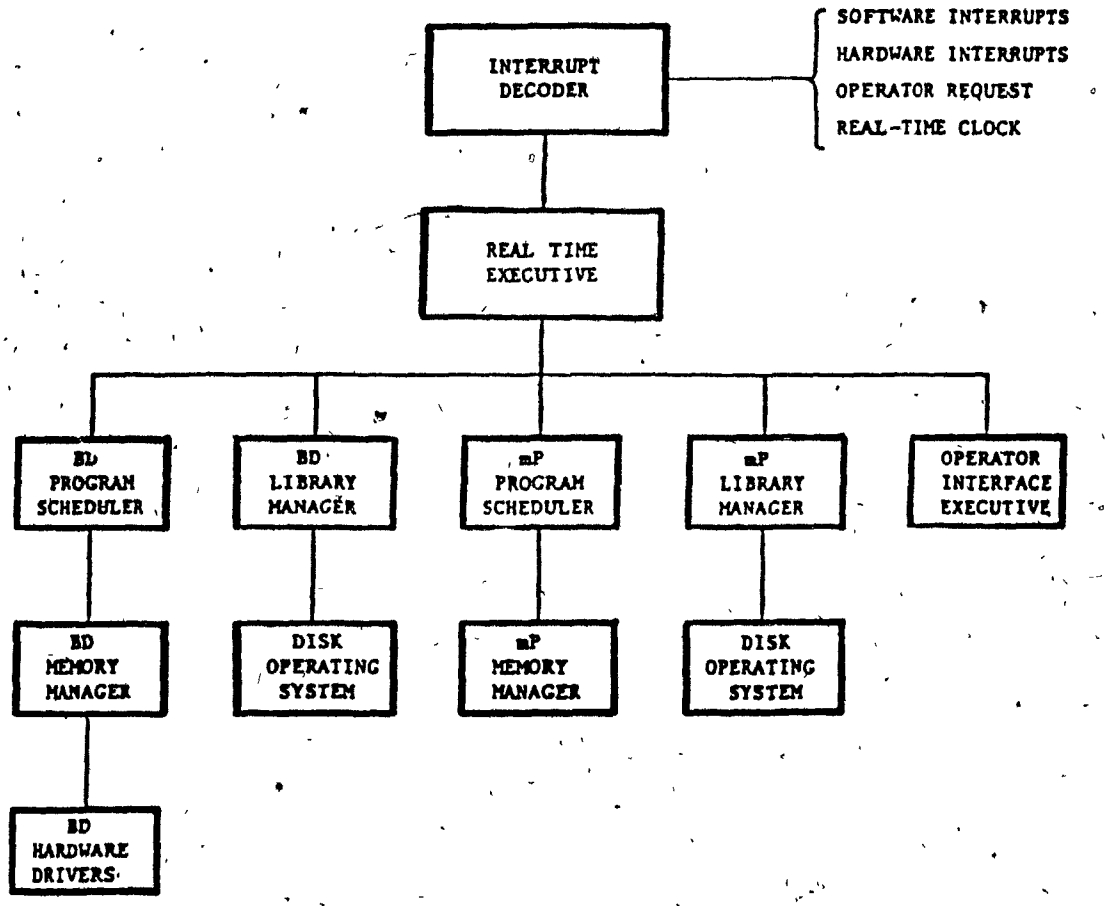


Fig. 4.18: The BD09 Operating System - Block Diagram.

BD Library Manager: The BD Library Manager employs the FLEX09 disk drivers to access the disk based storage memory device in which the library containing all BD programs is stored. The

primary drivers are the READ and the WRITE which allow the transfer of stored information to the mP memory and vice versa. A directory of all programs, containing information pertaining to program location and length, is maintained to facilitate library 'housekeeping' and provide quick retrieval in response to real-time requests.

BD Memory Manager: A directory to keep track of the current occupancy of BD memory is maintained by the BD memory manager. Anytime a new program is added to the BD memory the manager ensures that it will occupy only previously vacant memory regions. Load address dependent code is also generated at this time.

Hardware Drivers: These are low-level routines which control the modes of the BD processor in response to Real-time Executive commands. Each routine handles one operating mode. Control settings for the different BD operating modes are summarized in Table 4-5 for the PIA and Table 4-6 for the Control Register.

Utilities: A set of command utilities is available to the operator upon invoking the BD09 operator interrupt. Via this set of commands, Table 4-7, the operator is able to monitor the status of the BD processor, read or alter the BD program memory, create new library programs by accessing the compiler or assembler and activating any of the hardware drivers.

BD operation mode	Communication from -> to	PIA DATA REGISTERS A	B	SINGLE STEP (CA2)	R/W (CB2)	PIACRA code (HEX)	PIACRB code (HEX)
RUN	BD -> mP	INPUT	INPUT	H	H	3D	3D
STOP	BD -> mP	INPUT	INPUT	H	H	3D	3D
RESET PC	mP -> BD	OUTPUT	OUTPUT	H-L-H	H	35	3D
LOAD MEM.	mP - BD	OUTPUT	OUTPUT	H	H-L-H	3D	35
READ MEM.	BD -> mP	INPUT	INPUT	H	H	3D	3D
SINGLE STEP	N/A	N/A	N/A	H-L-H	H	35	3D
VERIFY	BD -> mP	INPUT	INPUT	H	H	3D	3D

H = digital level high L = digital level low

Table 4-5: PIA MC6821 - Control Codes.

BD operation mode	ALE CR7	OE CR6	PRESET CR5	STOP CR4	BD-CLK CR3	A/M CR2	VER CR1	DIR CR0	CODE (HEX)
RUN	H	H	H	L	H	H	H	L	EE
STOP	H	H	H	H	L	H	H	L	F6
RESET PC	L	L	H-L-H	H	L	H	H	H	17
LOAD MEM.	L	H	H	H	L	H	H	H	37
READ MEM.	L	H	H	H	L	H	H	L	36
SINGLE STEP	H	H	H	H	L	H	H	L	F6
VERIFY	H	H	H	L	H	H	L	L	AC

H = digital level high L = digital level low

Table 4-6: Control Register - Control Codes.

COMMAND UTILITY	DESCRIPTION
CREATE MODULE	converts a generic BD program to an address-dependent run module
DOWNLOAD MODULE	stops BD processor (if running), transfers module to BD memory, and restarts BD processor
GET MODULE	retrieve a module from the library
PRINT MODULE	prints hardcopy of program/module
SAVE MODULE	saves a module in library
COMPILE NEW PROGRAM	generates new generic BD control algorithm
SAVE NEW PROGRAM	save generic program in library
STOP BD	halts BD processor
LOAD MEMORY	writes into BD program memory
RESET BD PC	changes BD program counter to a desired address
READ BACK MEMORY	examines BD program memory
RUN BD	restarts BD processor
VERIFY HARDWARE	loads and executes diagnostics program

Table 4-7: BD Utilities.

4.6.2 The BD Compiler

The BD compiler has two functions:

1. The conversion of a truth table representation of the control problem into an executable BD program, and
2. The generation an optimized version of the BD program.

The creation of BD object code, ready to be downloaded into the BD memory or stored in the BD library, consists of 4 steps:

Step #1: BD Table Generation: This is a computer model of the binary tree generated from the truth table describing the control problem.

Step #2: BD Table Pruning: A pattern matching algorithm is employed to locate repetitious patterns in the original truth table output column. Subsets of identical complementary or non-complementary output states are identified in the BD table and redundant nodes are removed.

Step #3: Pseudo-code Generation: The pruned table is restored to an intermediate state before the actual BD code generation.

Step #4: Machine Code Generation: In this final compilation step the actual object code is generated to be stored in the BD program library.

4.6.3 The BD Assembler

The BD assembler language was developed to generate BD object code from symbolic source code and to eliminate the need to write BD programs in hexadecimal or binary object code. Although the BD compiler generates object code from a truth table representation of the control problem, direct

source coding, using the BD assembler, is preferrable if control logic is given in the form of equations, state-assignments, or a flow-chart.

The BD assembler is designed to handle free format statements. There are three types of statements:

1. BD instruction statements,
2. directive statements, and
3. comment statements.

Delimiters are used to separate the fields. CR and LF are the statement delimiters.

The BD assembler language is defined using the Backus Normal Form (BNF) notation in Table 4-8. BNF is a metalanguage first introduced to describe the Algol language in a formal manner [BN71]. A metalanguage uses characters, not used by the defined language, to describe its correct grammar.

The BNF primary symbols and conventions are as follows [GE69]:

1. Three special primary symbols are used to represent the following:
   a. ::= means 'is defined as'
   b. < > delimits the name of a string, i.e., set of entities, and
   c. | means 'or else', i.e., it separates alternatives.
2. All other symbols denote themselves.
3. Symbols, strings, or names following in succession in any combination, implicitly specify concatenation, i.e., and.
4. The order of applications of operations is: concatenation, then alternatives, then the definition symbol.

5. Unless otherwise stated the syntax equation is to be applied in left-to-right order.

A more detailed description of the BNF notation is given in Appendic C.

```
<letter>          ::=   A|B|C|.....Y|Z|a|b|c|.....y|z

<digit>           ::=   1|2|3|4|5|6|7|8|9|0

<alphnum>         ::=   <letter>|<digit>

<name>            ::=   <letter><alphnum><alphnum><alphnum><alphnum><alphum>|
                        <letter><alphnum><alphnum><alphnum><alphnum>|
                        <letter><alphnum><alphnum><alphnum>|
                        <letter><alphnum><alphnum>|
                        <letter><alphnum>|
                        <letter>

<number>          ::=   <digit>|<digit><digit>|<digit><digit><digit>

<field delim>     ::=   <space >
<delim>           ::=   <field delim>
<statm. delim>    ::=   <CR,LF>
<pointer delim>   ::=   :
<value qual>      ::=   $|%
<directive>       ::=   <EQU>
<operand>         ::=   <name>|<number>|<value qual.><number>
<label>           ::=   <name>
<mnemonic>        ::=   <IN0>|<IN1>|<OPS>|<OPL>|<IBS>|<OBS>|<INT>|<EOP>|<BRA>
<address>         ::=   <number>|<value qual.><number>
<pointer>         ::=   <name>|<address>

<instruction>     ::=   <mnemonic><delim><operand><pointer delim><pointer>|
                  ::=   <mnemonic><delim><operand>|
                  ::=   <mnemonic><delim><pointer delim><pointer>

<instruc. stat>   ::=   <label><field delim><instruction>|
                        <field delim><instruction>

<directive stat>  ::=   <name><directive><qualifier><value>

<comment qual>    ::=   < ' >
<comment stat>    ::=   <comment qual><string>
```

Table 4-8: BNF Definition of The BD Assembler.

Instruction Statements

The BD instruction statement may include up to four separate fields, Fig. 4.19:

1. The label field,
2. The instruction name or code – mnemonic,
3. The operand, and
4. The pointer to next instruction (jump address).

LABEL	MNEMONIC	OPERAND	POINTER

Fig. 4.19: The BD Instruction Statement Fields.

Instruction fields are separated by a field delimiter which is a space. The pointer field is identified by an additional pointer delimiter.

Field #1 – Labels are optional. A label should consist of maximum 6 characters and should be unique. It may be referred to by more than one instruction, or not at all. A field delimiter should be entered in the field if no label is used and the instruction address will be the next sequential location.

Field #2 – IR<15-11> must always exist. Table 4-9 contains instruction mnemonics, i.e., the op-codes.

Field #3 - IR<11-8> or IR<11-0> describes operands whenever they are required. The required operands are: input address, I/O bank address, output value, interrupt code.

Field #4 - IR<7-0>, in those instructions which contain a jump address, contains the next address either as an explicit numerical value or implicitly as a symbolic label name. If a pointer is required but is not specified, the assembler will generate a pointer to the next sequential instruction.

Operand names as well as address labels may be defined using the EQU directive.

MNEMONIC	Obj-code	OPERAND	JUMP POINTER
IN0	$0-I-jp	input addr.	yes
IN1	$4-I-jp	input addr.	yes
OPS	$C-S-jp	output value	yes
OPL	$8-LLL	output value	no
IBS	$E-B-jp	input bank addr.	yes
OBS	$B-B-jp	output bank addr.	yes
BRA	$F-x-jp	none	yes
INT	$D-c-jp	interrupt code	yes
EOP	$A-PROG	program information	no

I=input addr. S=short output LLL=long output
B=bank addr. x=don't care c=interrupt code
jp=jump pointer PROG=program information

Table 4-9: Instruction Set.

Similar to MC6809 assembler, the value type may be indicated by one of the following qualifiers:

1. Binary number - %,
2. Decimal number - no qualifier, and
3. Hexadecimal number - $.

The EQU directive can also assign a value to a unique 6 ASCII character name.

Delimiters

Except for the pointer field, a blank is used as the delimiter between the fields. The ':' colon character should precede the pointer.

Directive Statements

EQU is used to assign names to addresses of instructions, input and output banks, and input lines, and to values of output assignments, interrupt and program codes. There are three fields in an EQU statement:

1. name field - must start in first column
2. mnemonic - EQU
3. value field - contains the value and its qualifier

The fields are separated by the field delimiters.

Comment Statements

Comment statements may be inserted anywhere as long as the first character is ' (exclamation mark). The assembler will ignore any string following a comment qualifier.

Fig. 4.20 depicts a simple program written in BD assembler.

```
'                    Input Variables Assignment
A15   EQU IN<15>
A14   EQU IN<14>
A13   EQU IN<13>
A12   EQU IN<12>
B15   EQU IN<7>
B14   EQU IN<6>
B13   EQU IN<5>
B12   EQU IN<4>
'                    I/O Bank Assignment
BEGIN     IBS 000                       ' Input Bank Selection
          OBS 002                       ' Output Bank Selection
'
'         Compare four pairs of binary numbers: if the An number is 0
'         continue to next pair. Otherwise, check the corresponding Bn.
'         If Bn is 0 issue a unique output pattern identifying the pair and
'         continue to next pair. Jump to NEXT when cycle completed.
'
          IN0 A15    :AB14
          IN1 B15    :AB14
          OPL $040
AB14      IN0 A14    :AB13
          IN1 B14    :AB13
          OPL $020
AB13      IN0 A13    :AB12
          IN1 B13    :AB12
          OPL $010
AB12      IN0 A12    :AB11
          IN1 B12    :AB11
          OPL $010
AB11      BRA        :NEXT
```

Fig. 4.20: Assembler Programming Example.

The assembler generates object code using a two-pass method. The first pass generates the label table assigning addresses to all labels found either in the EQU lines or in the label fields. All addresses are assigned using an initial P counter of 0. Partial mnemonic decoding is done as well. The second pass completes the decoding by assigning the right address to the object code. The generated program is a non-conditioned one, i.e., BD09 has to offset all addresses in order to install the program in a memory segment other than 0.

CHAPTER 5: THE INTELLIGENT REFLEXIVE INTERFACE

5.1 The Intelligent Reflexive Interface Concept

The distribution of processing power either by employing several processors or by integrating processing capabilities into peripheral I/O hardware is preferred over a centralized architecture for the following reasons:

1. *Speed.* Tasks are executed concurrently.
2. *Reliability.* A higher degree of system security is achieved. The failure of one processor does not affect the operation of the remaining processors. Redundant processors can be employed in critical applications.
3. *Versatility.* A well designed distributed system is adaptable to a large variety of applications.

Computer interface modules can perform complex I/O functions. Dedicated hardware now performs many tasks which previously were handled by software. An example is a disk interface module. The interface contains a dedicated mP to execute disk data transfer operations. The master mP merely issues an initiation command and all sub-tasks are executed autonomously by the interface.

In a process control environment a wide variety of I/O tasks exist. Implementing distributed intelligence using a dedicated interface similar to the smart disk controller is inferior to a general purpose interface which is adaptable to different processes.

There are two types of process control activities:

1. routine scanning and control, and
2. smart tasks, involving arithmetic and intensive computation.

Routine control tasks detect process changes and take the necessary action(s). These tasks are defined as 'reflexive'. I.e., they are repetitious, often urgent, stimulus-response sequences governed by logical tests that tend not to change with varying process parameters.

BD based hardware handles Direct Digital Control (DDC) level reflexive tasks efficiently. This was the motive for developing an intelligent I/O interface module, simple enough to handle reflexive tasks very quickly. Efficient distributed processing is achieved by employing a number of these I/O modules, all controlled and supervised by a master microprocessor. Resident BD programs adapt the module to a variety of process control tasks. This I/O interface is called an Intelligent Reflexive Interface (IRI).

The IRI should have the following capabilities:

1. Reflexive processing - Monitoring of process signals and generation of output signals, in response to process changes, must occur in a real-time.
2. Programmability - The reflexive 'intelligence', i.e., decision making logic, must be easily adaptable to different processes and to different control ranges within the same process.
3. Independence - It must be able to handle most tasks autonomously, i.e., without main processor intervention.

Control decisions, depending upon their frequency and complexity, are categorized and processed in one of the following ways:

1) Reflexive decisions - frequent control tasks with mainly binary, on-off signals, handled independently by an IRI module.

2) Reflexive-aided decisions - less frequent control tasks, handled by an IRI module with assistance of the mP.

3) Smart decisions - arithmetic calculations, control algorithms and data manipulations, modifications of IRI programs, etc. handled by the mP.

Use of BD intelligence is attractive for a number of reasons:

1. Speed - Binary Decision logic has natural bit processing capability and very few gating levels and thus achieves very high processing speed.
2. Programmability - different control applications can be accommodated with ease.
3. Low cost - BD hardware is very simple.

A system employing a high level microprocessor with one or more BD based IRI modules was developed. It is similar in some ways to the hybrid system. The IRI is connected directly to the main system bus; there is no intervening parallel interface module. Modularity and memory organization promotes fast program transfers between the mP and the IRI's BD processors. The IRI intelligence embedded in BD programs can

be dynamically altered to adapt to new situations. System modularity is an important factor in the process environment as processes differ in size and processing power requirements.

5.2 The IRI Based Controller Architecture

The parallel architecture of an IRI based controller is shown in Fig. 5.1. Several IRI modules are connected directly to the main controller bus together with the master processor, internal memory, interface to bulk storage, serial interfaces to operator consoles and parallel interfaces to cpu controlled I/O. The main bus consists of data, address and control lines accessible to all bus residents.

Each IRI module has a Binary Decision IRI processor (IRIP) on board. This is the BD processor presented in Chapter 4, with slight modifications to adapt it to the IRI environment. The IRI serves as a slave processor, i.e., its operation is controlled and supervised by the controller master processor. Each IRI card is connected to a set of I/O cards via a secondary IRI I/O bus. The IRI I/O bus consists of data and control lines generated by the IRIP. A large number of IRI modules can be incorporated with a single master processor. Each IRI module can accommodate as many as 16 I/O cards.

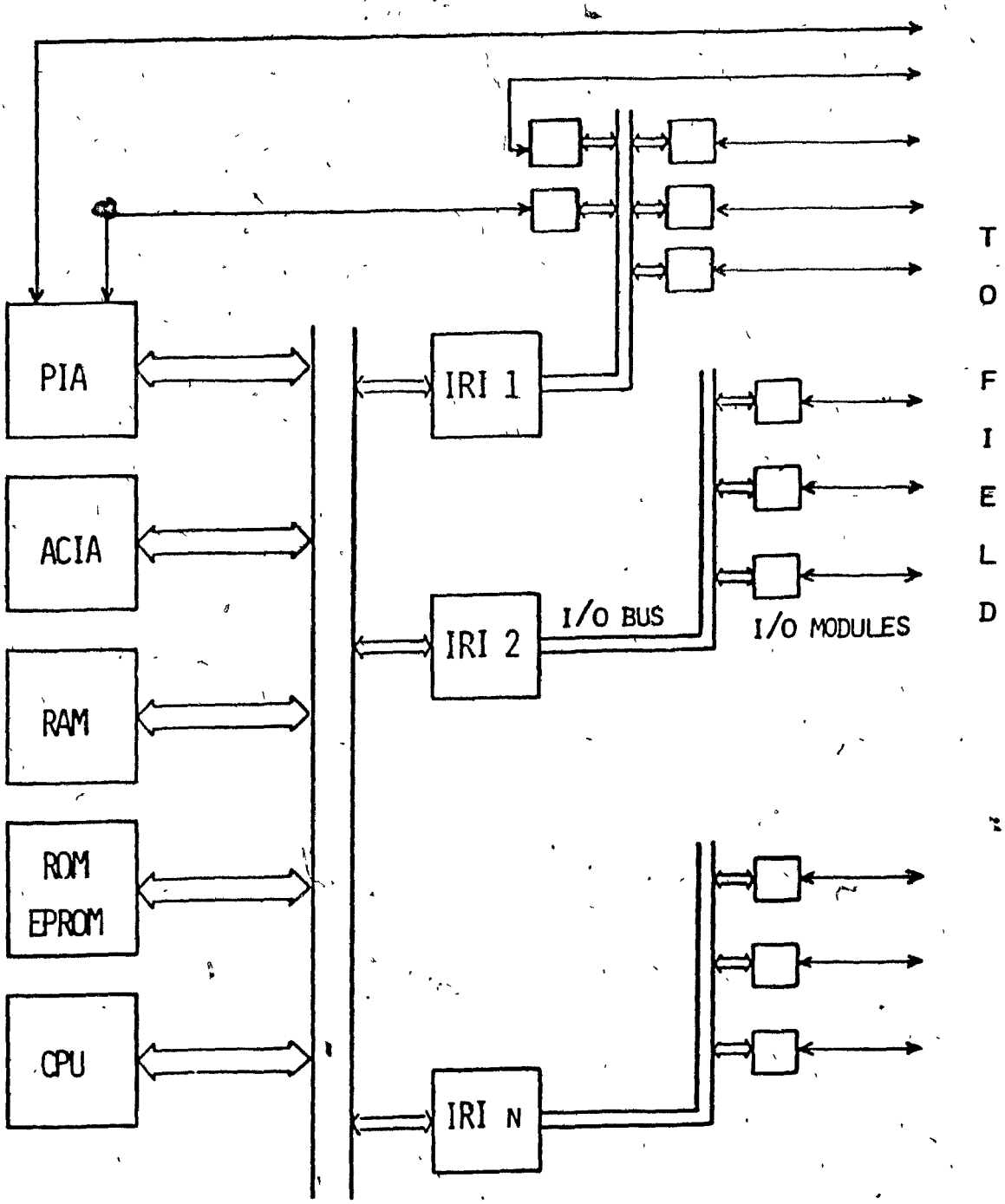


Fig. 5.1: An IRI Based Controller - Block Diagram.

The prototype IRI controller was designed using the MC6809 8-bit microprocessor. Other microprocessors can serve as the controller master processor as long as they support memory mapped I/O.

Four IRI controller versions, with different memory organization and master-slave communication structure, were designed. The IRIP and the IRI I/O interfaces are identical in all versions. They are:

1. The Dedicated Memory (DM) IRI,
2. The DMA IRI,
3. The Shared/Dedicated Memory (SDM) IRI,
4. The Shared Memory (SM) IRI.

The DM version was designed with the IRIP memory on board, similar to the Hybrid architecture, used for storing the BD programs. Only the IRI may address this memory. The mP can access it indirectly. The DMA version is an enhancement of the DM IRI which incorporates the necessary logic to enable DMA transfers. The SDM version allows direct addressing of the IRI memory by the mP. A dynamic memory scheme is offered by the SM version where the IRIP can dynamically access any predefined mP memory segment. The mP can instantly alter the active IRI memory segment within the controller memory map as the process requires.

To accommodate standard process requirements three types of IRI I/O interface cards were designed:

1. Binary I/O,
2. Analog/Binary I/O,
3. Timers.

Other special I/O cards can be accommodated.

I/O communication is handled in four ways:

1. IRI I/O <=========> Process
2. mP I/O <=========> Process
3. mP I/O <=========> IRI I/O
4. IRI I/O <=========> IRI I/O

This versatile communication scheme covers a wide variety of control configurations:

1. It allows an IRI module to directly communicate with the process via its own I/O cards when handling control tasks autonomously.
2. It allows the mP to directly communicate with the process via its own non-IRI I/O interface (e.g., a PIA) when handling mP dedicated tasks.
3. It allows the mP to communicate with the IRI via their respective I/O cards when shared tasks are executed.
4. Similarily, it allows two IRIs to communicate via their own I/O cards.

5.3 The Dedicated Memory IRI Module

The Dedicated Memory (DM) IRI module has its own program memory. This memory is dedicated to the IRI because it:
1. serves the IRIP for BD programs storage and execution, and
2. it does not exist in the mP memory map.

The memory can be directly addressed by the IRIP via the DLU controlled program counter. The mP addresses the IRI memory indirectly by operating on the IRI program counter. This is the simplest implementation which is closely related to the hybrid module described in Chapter 4.

Fig. 5.2 shows the module layout. The IRIP is the central component on the module. Connected to the the IRIP are:

1. Program Memory and its Program Counter,
2. Byte/Word Logic,
3. Communication Logic,
4. Board Select Logic, and
5. IRI I/O Interface.

Two versions of the Dedicated memory IRI were designed for the SS-30 and for the SS-50 bus (see Appendix A), respectively. The module requires less then 16 locations in the system's I/O map and therefore it can reside on the SS-30 bus which provides four bits for internal addressing along with an already decoded Board Select signal. The SS-50 version is not limited to the addresses available by the SS-30 bus but it requires Board Selection Logic to be incorporated on board.

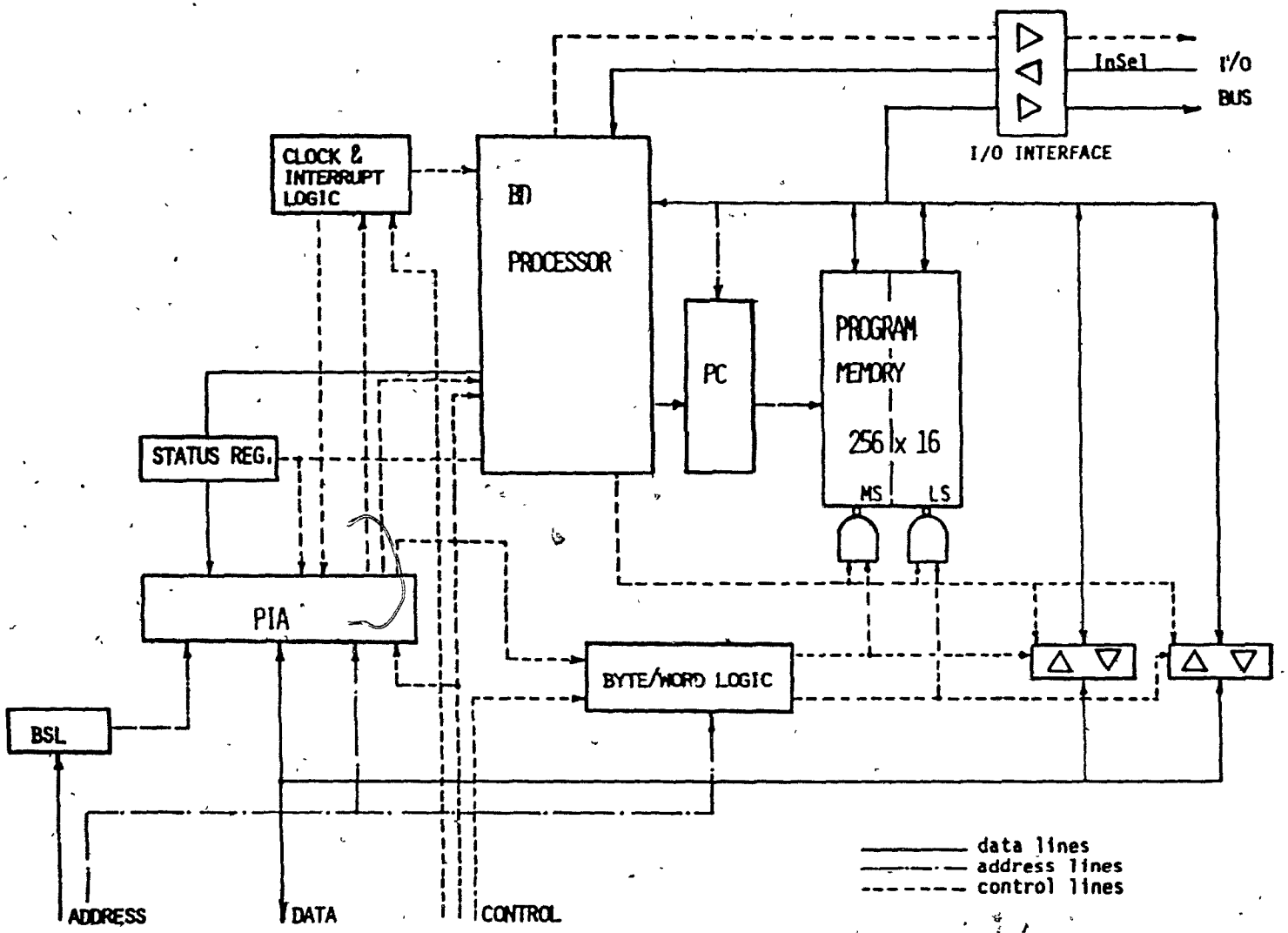


Fig. 5.2: The Dedicated Memory IRI Layout.

5.3.1 The IRI Processor

The IRIP is the Expanded Binary Decision processor described in detail in Chapter 4. It consists of three blocks, the DLU, the op-code decoder and the CIL (shown externally to the IRIP in Fig. 5.2). It cycles through the BD programs stored in the IRI memory, scanning process inputs and issuing outputs determined by the BD algorithms.

An interrupt code capability was added to provide the mP with interrupt information, previously provided by the EOP instruction. The 4-bit code stored in bits <11-8> of the interrupt instruction is transmitted to the mP via the status register. The remainder of the instruction set was unaltered (see section 4.3.2).

5.3.2 Program Memory

BD instructions are 16 bits while the mP data bus is eight bits wide. In the hybrid scheme 8-bit to 16-bit compatibility was handled by the PIA which accepts 8-bit data input and is capable of transmitting 16 parallel bits. In the IRI structure, where direct memory to memory data transfer is required, segmentation of the IRI program memory and Byte/Word logic are employed.

The 256x16-bit static RAM shown in Fig. 5.3 is divided into two sections; the most significant (MS) 256 bytes and the least significant (LS) 256 bytes. Dedicated Byte/Word logic (see section 5.3.3) was developed to enable writing/reading one byte at a time when the mP accesses the memory, and reading two bytes simultaneously, i.e., a whole BD instruction

word, when the IRIP accesses the memory. Memory words are addressed by eight primary address lines (IA<7:0>) generated by the program counter. Single bytes can be accessed by asserting the Chip Select (CS) input of the desired memory segment. The Byte/Word logic enables one CS input at a time when the mP accesses the memory and both CS inputs when the IRIP accesses the memory.

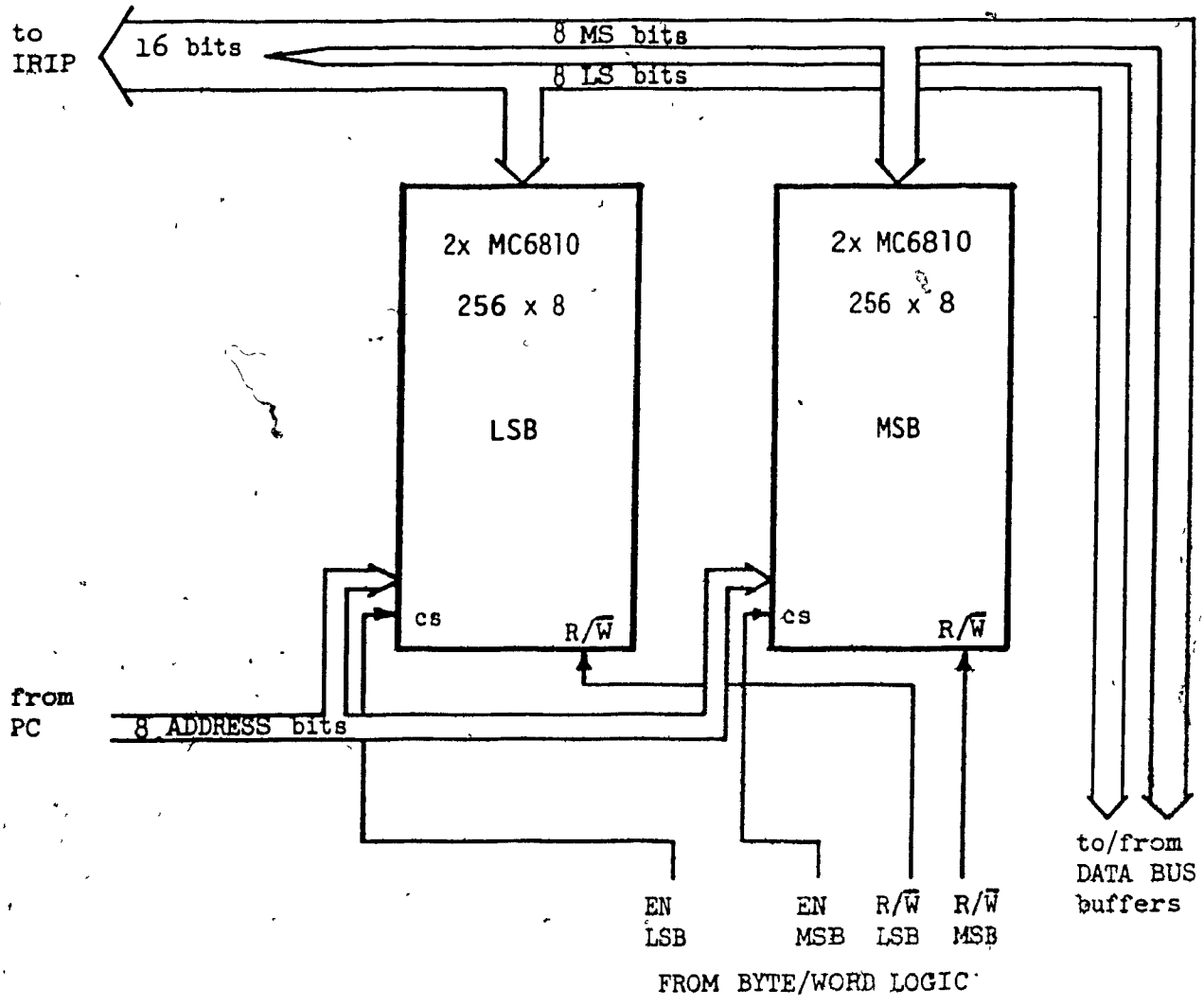


Fig. 5.3: The IRI Dedicated Program Memory.

5.3.3 Byte/Word Logic (BWL)

The BWL circuit controls data transfers between the mP memory and the IRI memory. The MC6809 sends data over an 8-bit data bus. The IRI internal data bus is 16-bit wide. The BWL therefore channels the mP 8-bit bytes into the appropriate BD 16-bit word and vice-versa.

The block diagram, Fig. 5.4, shows the main components of the BWL. It consists of Data Direction Logic (DDL) and Data Buffer Logic (DBL). The BWL controls:

a) the direction and enable inputs of the two 8-bit bi-directional tri-state data buffers connecting the IRI 16-bit data bus to the controller 8-bit data bus, and

b) the enable (Chip Select) and R/$\overline{W}$ inputs of the IRI memory.

The BWL makes use of the following signals in executing its functions:

1. R/$\overline{W}$ – The mP Read/$\overline{\text{Write}}$ bus line defines the data transfer direction.
2. DCL – Data Channel Low. The address of the LS data channel to the IRI memory.
3. DCH – Data Channel High. The address of the MS data channel to the IRI memory.
4. $\overline{\text{BD-CLK-ON}}$ – This signal produced by the Clock and Interrupt Logic (see Chapter 4) enables mP access to the IRI memory when the internal IRI clock is not on.

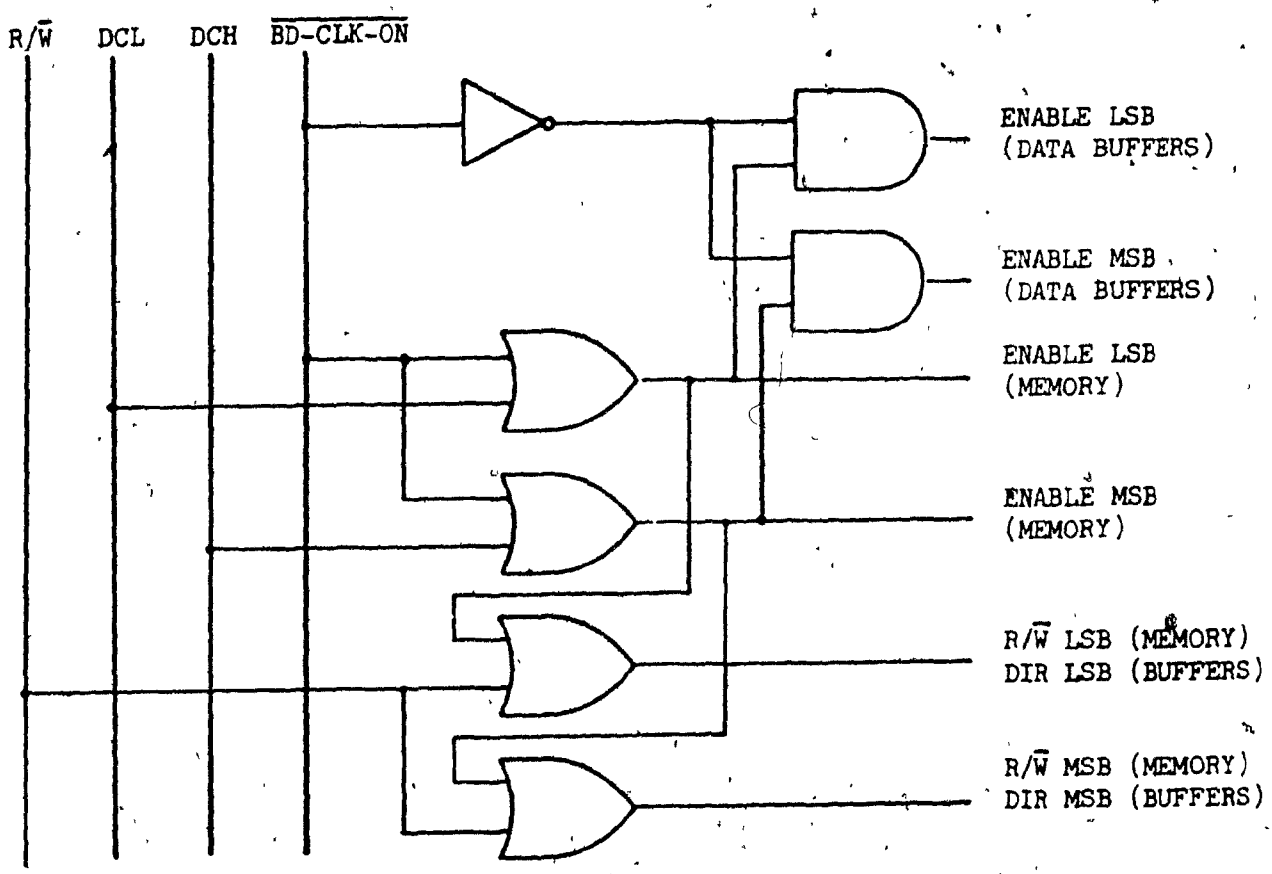


Fig. 5.4: Byte-Word Logic - Block Diagram.

mP Data Transfer: The mP activates the PC Preset line via the communication PIA (section 5.3.4) to preset the program counter and then increments the program counter via the SS (Single-Step - CA2) to access the next two bytes. The SS signal along with alternate addressing of the data channels are employed to access single bytes of the IRI memory. The PC provides the eight MS bits of the 9-bit memory address while the ninth LS bit is defined by the data channel being addressed. The data direction is determined by the R/$\overline{W}$ signal from the controller bus. Data buffers are enabled only if:

1. The BD clock is not on, i.e., the IRI is not in running or diagnostic mode and the mP can access its memory, and
2. One of the data channels is addressed by the mP. Only one channel is enabled at a time.

The R/$\overline{W}$ inputs to the LSB and MSB segments of the memory are separately controlled. During memory READ operation both R/$\overline{W}$ inputs are asserted so that both LS and MS bytes can be read. However, only one of the bytes will be channelled to the mP bus as only one set of data buffers can be enabled by either DCL or DCH. During a WRITE operation however, only one R/$\overline{W}$ input (to the addressed segment) will be negated. The other R/$\overline{W}$ input remains asserted to prevent the writing of non-valid data to the second segment (although the data buffers to that segment are disabled, the data lines are floating and a WRITE operation will result in writing non-valid data into the memory).

IRI memory access: The IRIP has Read access during running or diagnostics modes only. The $\overline{\text{BD-CLK-ON}}$ masks the mP enable signals to the memory CS inputs such that both segments are enabled and a 16-bit word can be addressed. The memory R/$\overline{\text{W}}$ inputs are always asserted.

5.3.4 Communication Logic

A MC6821 PIA, Fig. 5.5, handles communication tasks between the mP and the IRI. Port A is designated as an input port reflecting the contents of the status register. Port B is designated as an output port providing control signals to the BD processor. The control signals are:

1) STOP Flag,
2) P.C. Preset,
3) Auto Load Enable,
4) Output Enable, and
5-8) four Test Inputs.

Interrupt signals transmitted via CA1 by the IRIP and via the STOP flag by the mP provide handshaking between the IRIP and the mP. The 4-bit interrupt code is latched into the status register and can be decoded as 16 different interrupts by the mP. The CB1 line is used by the mP to detect the status of the IRI clock. CA2 is used as single-step signal to increment the program counter.

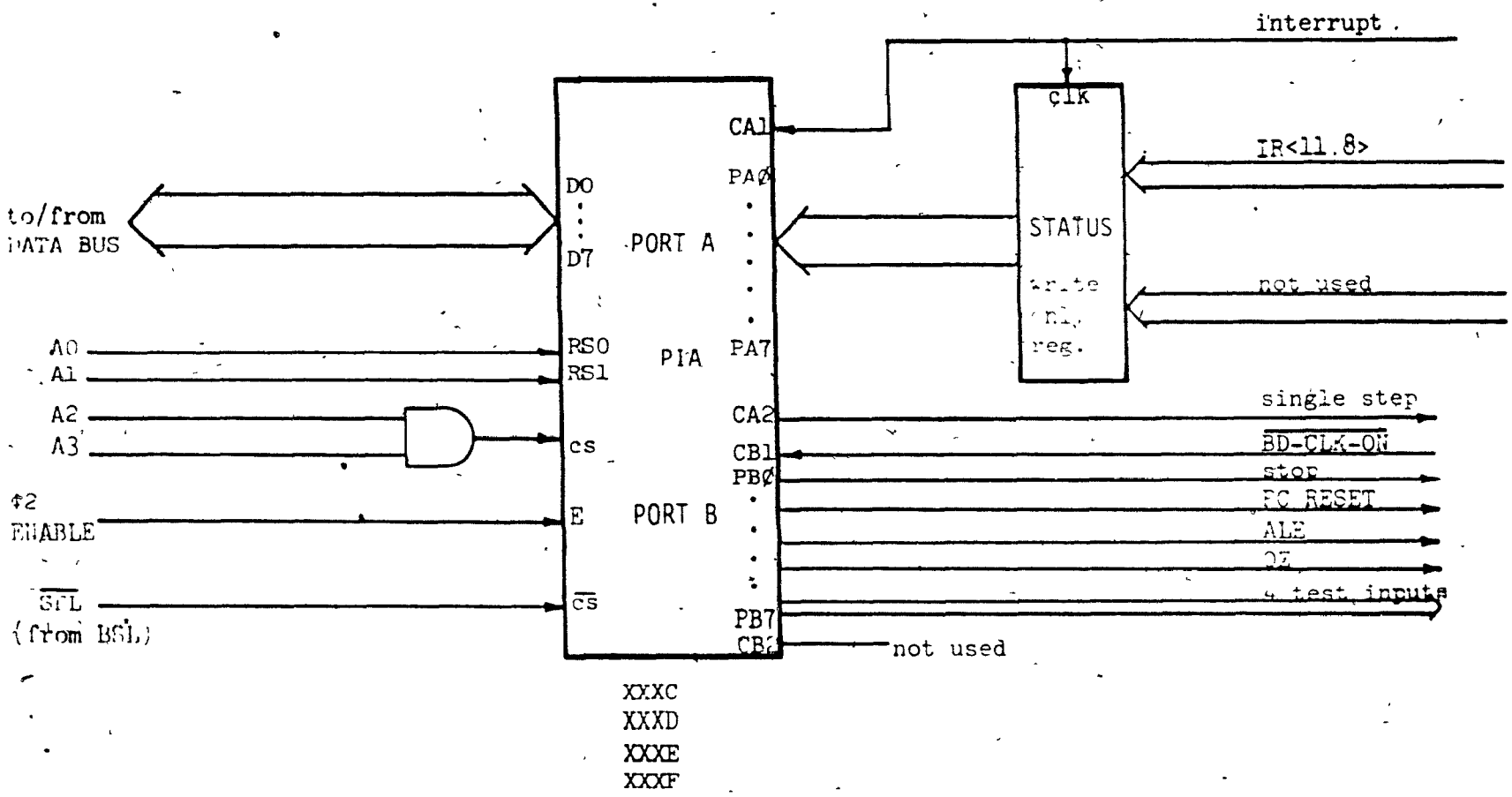


Fig. 5.5: The IRI Communication Logic

5.3.5 Board Select Logic (BSL)

An IRI module requires six internal addresses decoded from the four LS bits of the mP address bus. These are:

1. Control/Status PIA - 4
2. Data Channel Low and High - 2

The 12 MS address bits are used to decode the module global address.

The dedicated memory IRI module is designed in two versions; the SS-30 version which has the module address decoded external to the module, i.e., the bus Board Select signal serves as the module $\overline{SEL}$ signal, and the SS-50 version which has all decoding performed on board.

The SS-50 version is a universal one and does not restrict the module to a specific bus slot. The module address is decoded by connecting the 12 MS bits of the 16-bit address bus to a set of comparators (XOR gates packaged in two 74LS688) in parallel with 12 user selected DIP switches, Fig. 5.6. The comparators output lines are wire-ored to form the $\overline{SEL}$ line. The $\overline{SEL}$ line is activated only if all address lines correspond to the user defined board address.

The decoding logic of the six internal addresses, Fig. 5.7, is identical in both the SS-30 and the SS-50 versions. The $\overline{SEL}$ signal is interlocked with the internal address decoding.

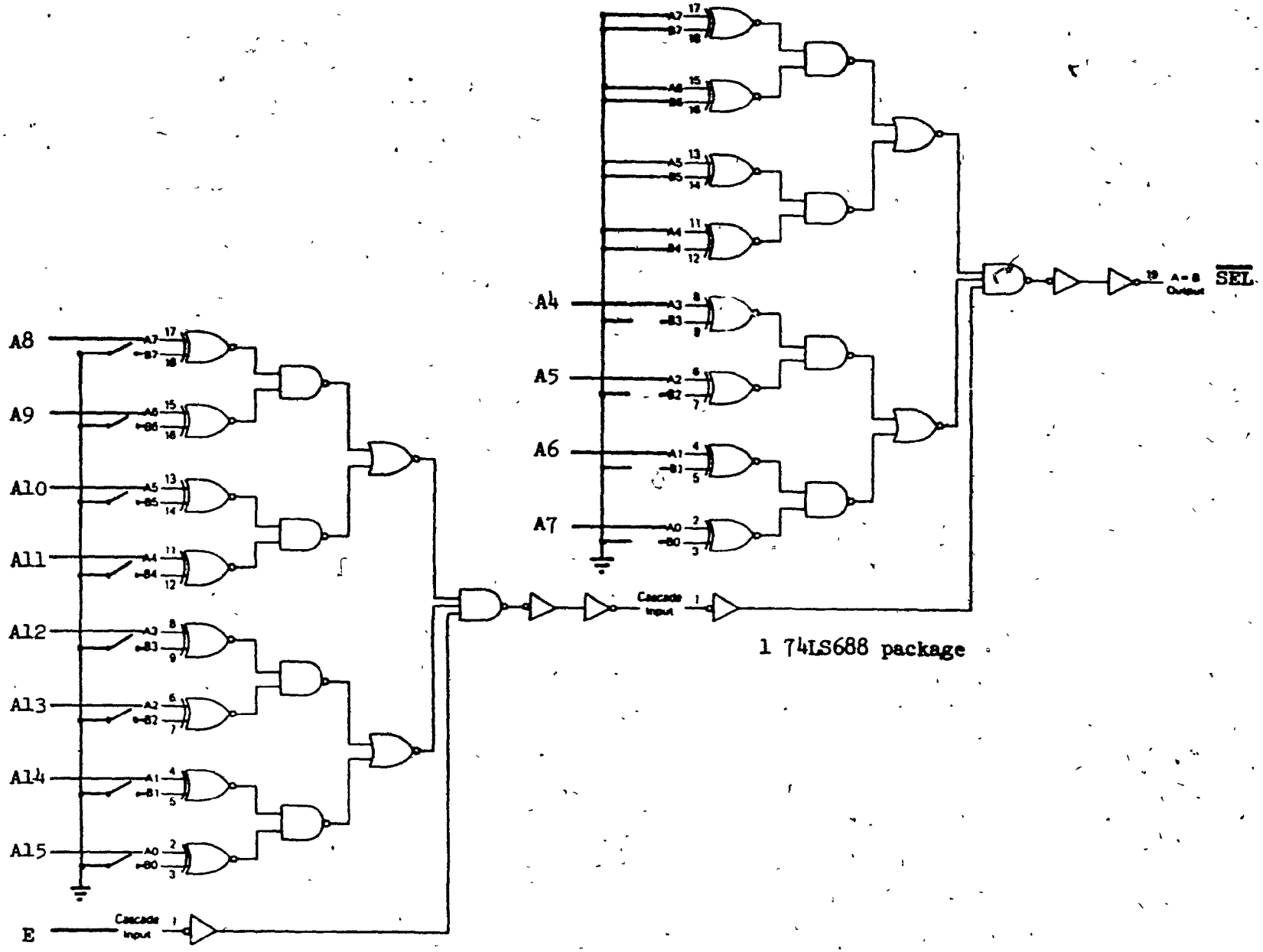
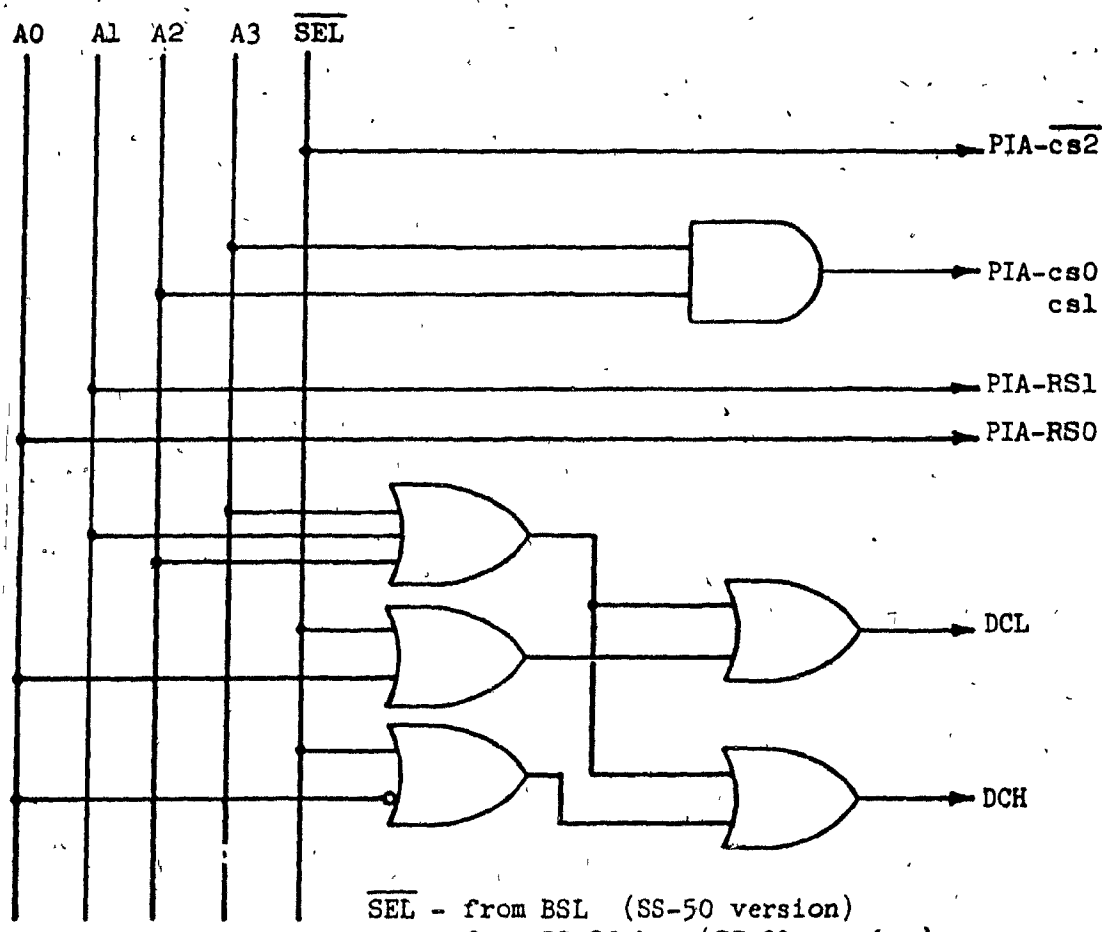


Fig. 5.6: The IRI Select Logic.



SEL - from BSL (SS-50 version)
from SS-30 bus (SS-30 version)

PIA - port A: XXXC
XXXD
port B: XXXE
XXXF

DCL - XXX0

DCH - XXX1

Fig. 5.7: The IRI Internal Address Decoding.

5.3.6 The Controller Bus

The prototype IRI controller is designed around the SS-50 bus or its SS-30 subset [AW80]. The bus is the standardized communication mechanism by which system modularity can be achieved. The bus consists of address, data, control and power lines.

Bus organization requires that devices connected to it will be classified either as master or slave. The mP is normally the bus master. It may gain the attention of an IRI module (which is a slave) by placing its address on the bus. A slave device, therefore, must have a unique set of addresses. A device can transfer data to a second one via the data bus. The connection of each device to the data bus must be buffered so only one device can have access to it at a time. In certain cases the master device can delegate a slave device to temporarily become the bus master, e.g., in DMA mode. The SS-50 and the SS-30 busses are designed for the Motorola 6800 familily of microprocessors and their support interfaces. Appendix A describes the signals and their functions for both busses.

5.3.7 The IRI I/O Bus, I/O Interface

Process communication takes place via the IRI I/O modules. Up to 16 modules may be connected to a 25 line I/O bus originating at the IRI. The bus includes the following signals:

1. D0-D11. These 12 data lines transmit the LS 12 bits of the current instruction word. The I/O module extracts the addresses of the Input Bank, Output Bank and Input Line from <11:8>. The 12 data lines are also connected to the output bank to be latched during an OPS - output short (<11-8>) or OPL - output long (<11-0>) instruction.
2. INSel. The selected input (during an input test instruction execution) is returned on the INSel line.
3. Control lines: Output Short (OS), Output Long (OL), Input Bank Select (IBSel) and Output Bank Select (OBSel).

Fig. 5.8 shows the IRI I/O interface section on the IRI module. The I/O part is common in design to all IRI versions. The interface consists of 18 unidirectional tristate drivers which are permanently enabled. The drivers are interfaced to a 26 pin edge connector to which a flat ribbon is connected. The cable is connected to all I/O cards via similar edge connectors.

The design of the I/O modules themselves is independent of the controller type and is presented in section 5.7.

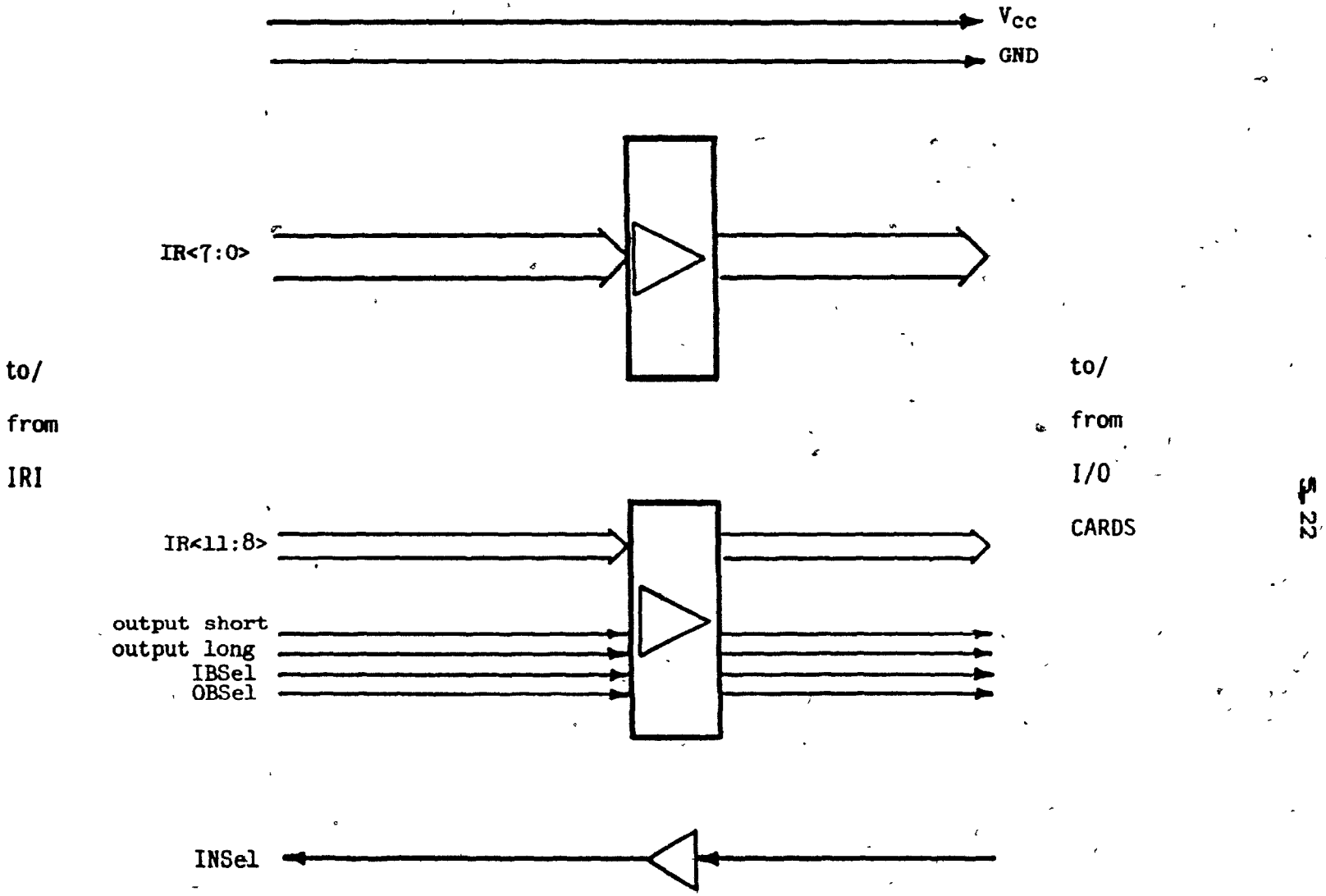


Fig. 5.8: The IRI I/O Interface

5.4 The DMA IRI Module

The DMA IRI module is the Dedicated Memory IRI described in section 5.3, with additional logic allowing it to operate in a DMA data transfer environment. In an IRI controller system with DMA capabilities, Fig. 5.9, data transfers to and from the IRI modules are performed by a DMA controller (DMAC) provided on board the mP module. DMA transfers are fast and require limited mP involvement. The DMAC employed is the MC6844. The IRI requires DMA support logic to conform to the DMA protocol during data transfers.

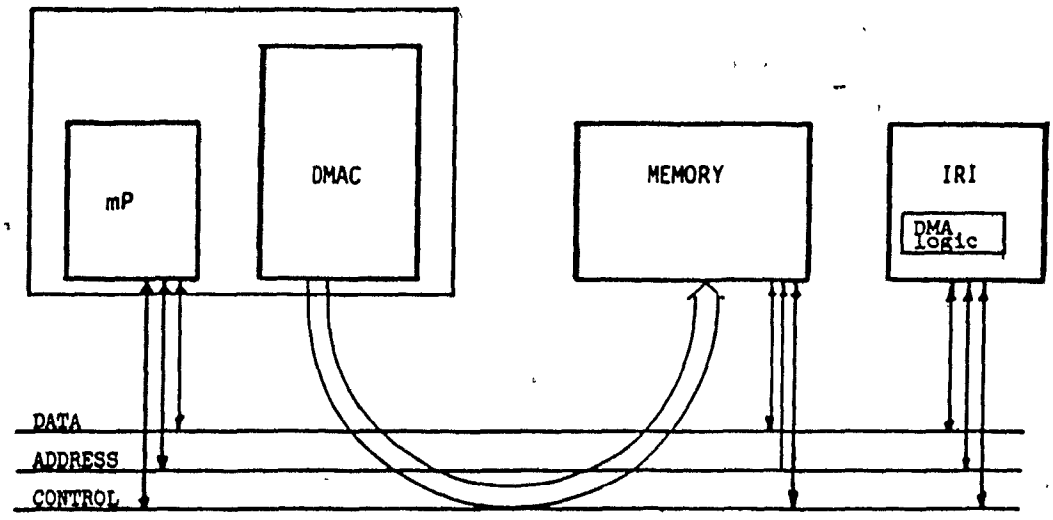


Fig. 5.9: The DMA IRI Controller.

5.4.1 The DMA Module Structure

The DMA IRI module is shown in Fig. 5.10. It has in addition to the Dedicated Memory version hardware, the DMA support logic, Fig. 5.11. The dedicated DMA support logic ensures control of internal memory addresses during DMA transfers. This logit includes:

1. A DMA Enable Latch ($\overline{DEL}$). This signal, latched by the mP during initialization of DMA session, informs the IRI internal logic that it is now selected for the duration of the DMA session. This is required because the address bus is going to be occupied throughout the session pointing to the source or destination of the data in the mP memory. The $\overline{DEL}$ is equivalent to the $\overline{SEL}$ signal of the first version.
2. Deselection Logic. When the DMA session ends $\overline{DEL}$ has to be deselected. This is accomplished by detecting the status of BA (Bus Available) signal on the bus.
3. Program Counter Control. During data transfer initialization the mP presets the IRI program counter. The program counter has to be incremented after every two bytes transfer. This is accomplished by detecting the A0 cycle and toggling the SS (Single-Step) when an A0 falling edge is detected.

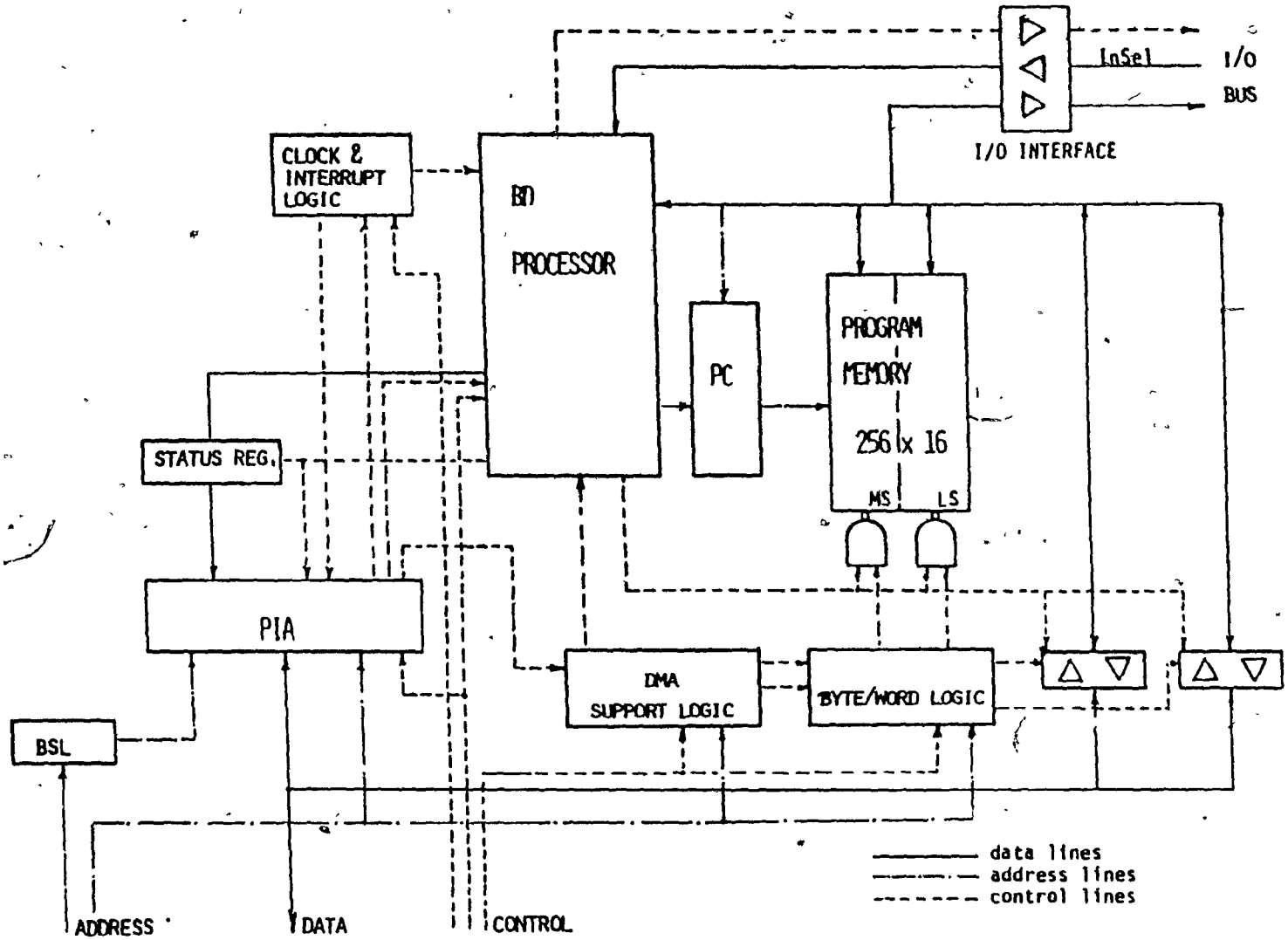


Fig. 5.10: The DMA IRI Module.

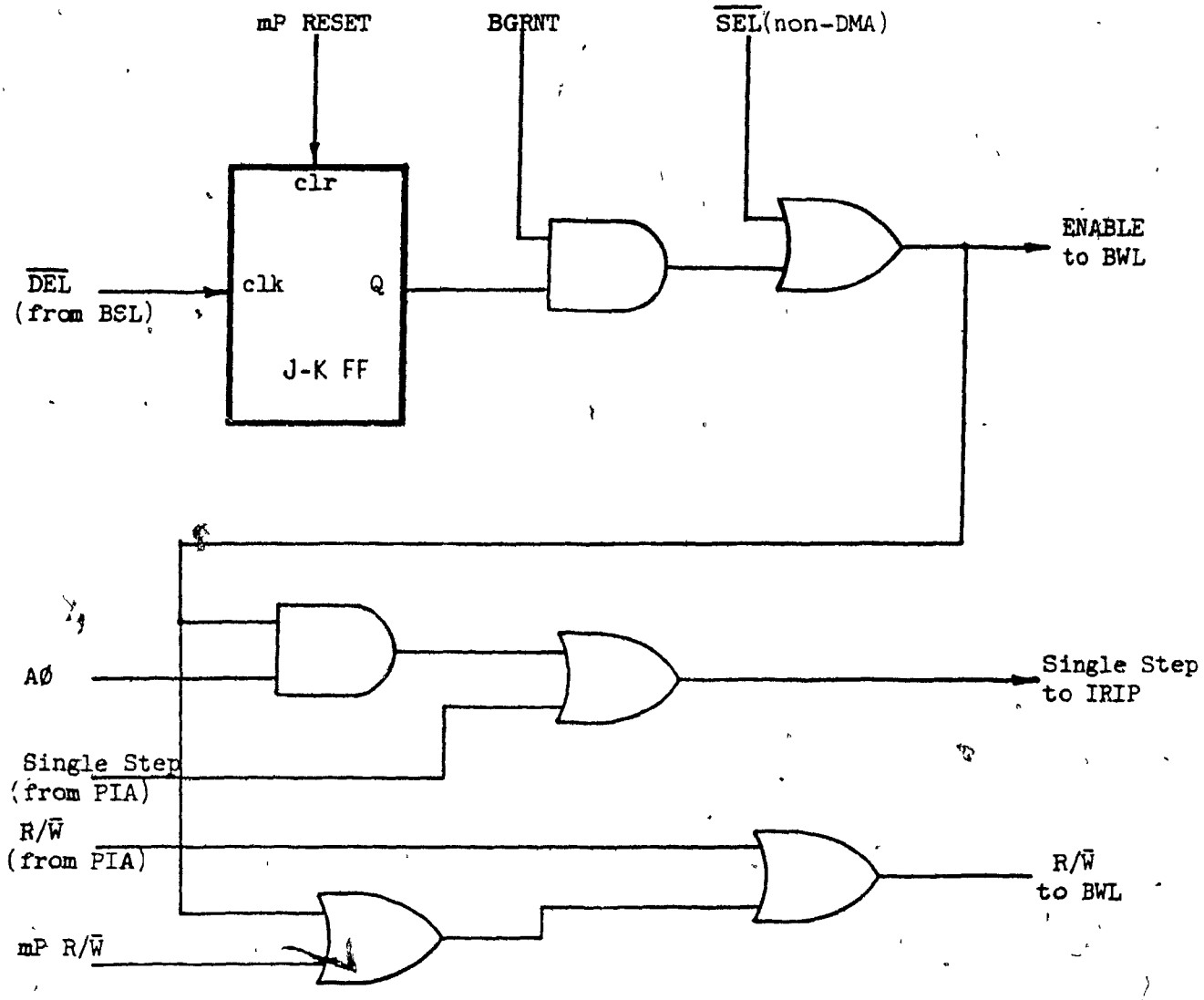


Fig. 5.11: The DMA Support Logic.

5.4.2 DMA Communication

The MC6844 DMAC is capable of taking control of the main bus and generating all bus signals required for transfering data. DMA data transfer does not require the mP to execute several instructions per transferred byte as it normally does during programmed data transfer. DMA rates of 1 Mbyte/sec can be achieved.

The mP initiates the transfer by:

1. presetting the IRI program counter and initializing the DMA support logic,
2. loading the DMA controller with the starting address and the length of the data block, and
3. passing the bus control to the DMA controller.

The DMA cycle is shown in the timing diagram in Fig. 5.12. After the mP initializes the DMAC for a specific data transfer, the latter issues DMA request signal (DMA/REQUEST is negated). The mP gets off the bus by asserting the DMA Grant (DMA/GRANT) signal. The DMAC takes control, places the first memory address on the bus and activates R/$\overline{W}$ to define the transfer direction. The Enable line ($\emptyset$2 of the 6809 clock) synchronizes the transfer which is excuted at 1 Byte/cycle. The IRI detects the bus signals BA, BGRNT and A0 to synchronize the data channeling to/from the program memory. Upon transfer completion, BREQ is deasserted and bus control is returned to the mP. The mP will then reactivate the IRI as required.

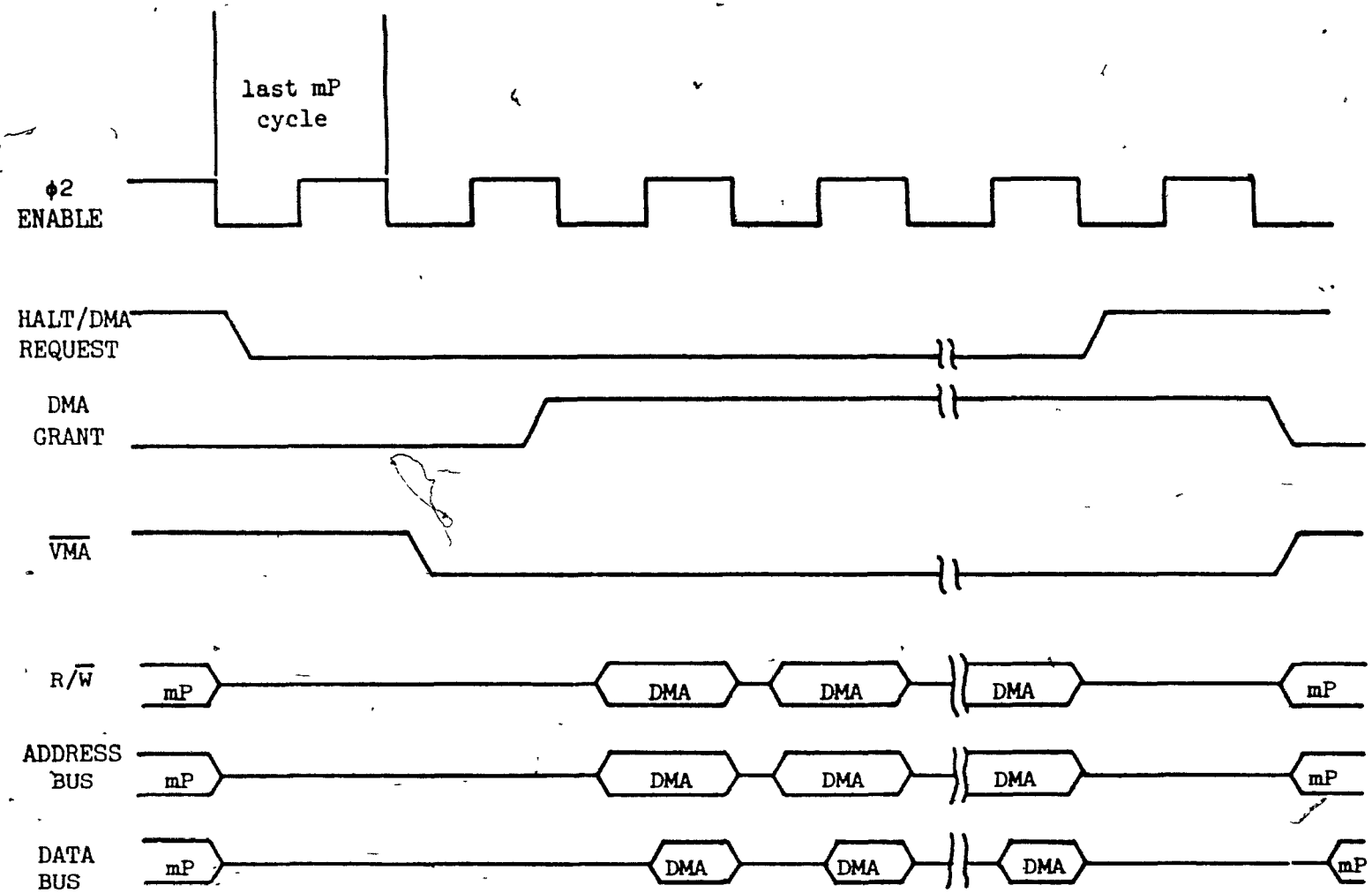


Fig. 5.12: The DMA Cycle - Timing Diagram.

5.5 The Shared/Dedicated Memory

Both Dedicated and DMA IRI versions are designed with an independent BD memory which is not part of the mP memory. In the Shared/Dedicated controller structure, Fig. 5.13, the IRI memory is designed to be a part of the mP memory and thus occupies a segment in the overall system memory map. The memory may be directly shared by both the mP and the IRI by connecting both address busses to the memory via buffers. Under normal IRI running conditions the mP is denied access to the IRI memory segment. The mP may access the memory while the IRIP is halted. The shared/dedicated memory offers flexible mP-IRI operation.

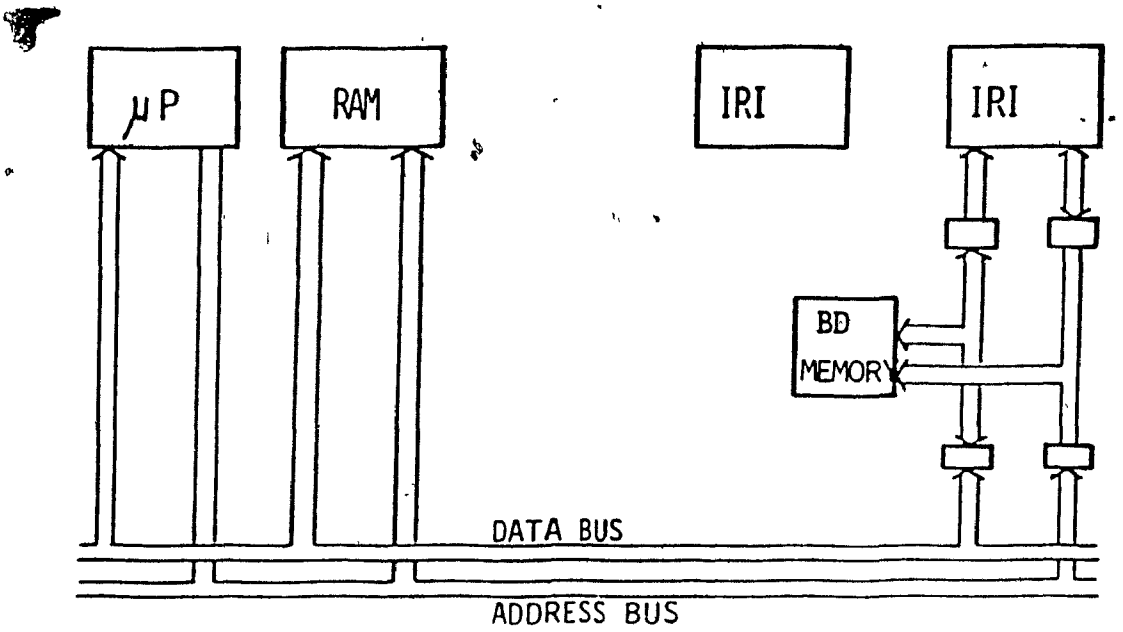


Fig. 5.13: The Shared/Dedicated Controller Architecture.

5.5.1 Controller Structure

This version shown in Fig. 5.14, is designed only for the SS-50 bus because all 16 address lines are required. Modifications to the Board Select Logic of the Dedicated memory version were made to allow direct addressing of all 512 bytes. Address and data buffer control logic enable the mP to access the memory of one IRI processor at a time. Data transfers between the mP and the IRIs are essentially memory to memory transfers as the mP controls both IRI memory and its own memory via direct addressing.

During IRI running mode, its program memory is addressed by the IRIP via the program counter. The output of the program counter is buffered by eight tristate drivers enabled by the $\overline{\text{BD-CLK-ON}}$ signal. Upon an interrupt and the stopping of the IRIP, access is granted to the mP which can now read the program memory or download new programs via direct bus addressing.

Byte/Word Logic ensures that one memory byte at the time is accessed under mP addressing and both LS and MS bytes are being accessed when the IRIP is addressing the memory. During mP access all nine LS address lines are connected to the memory and single bytes are being addressed. When the IRIP accesses the memory, the BWL buffers the LS bit of the address bus, A0. The program counter address lines are equivalent to A1-A8 and one word (two bytes) is addressed at a time.

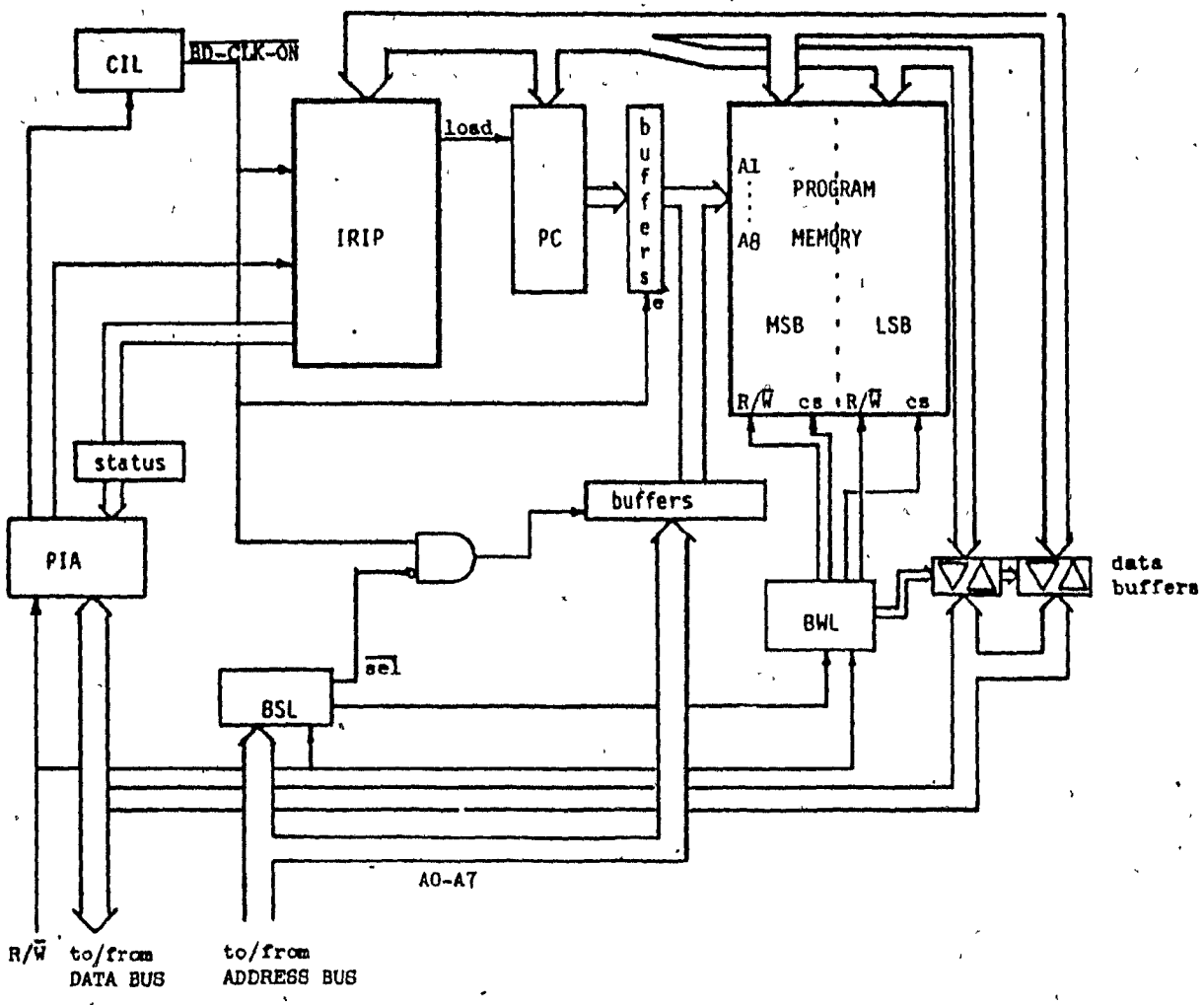


Fig. 5.14: The Shared/Dedicated IRI Module.

5.6 The Shared Memory IRI Module

This version of the IRI controller may be also called a 'dual processor' system since only one IRI (slave processor) and one mP (master processor) are implemented. The two phase clock provides a separate phase to each of the processors so that they are 'invisible' to one another.

5.6.1 Controller Structure

Fig. 5.15 depicts the Shared Controller architecture. The two processors, the mP and the IRI share the same system bus to access common (shared) memory. The mP may access memory during $\phi 2$ of a 2-phase system clock while the IRI has access during $\phi 1$. This makes the mP and the IRI invisible to each other during bus access. Additional IRIs cannot be accommodated because the IRI memory, active during process control session, is accessed via the single system bus.

The controller consists of Clock and Control Logic which provide the timing signals to both processors and to the other bus residents. The IRI can only access the memory for reading data, i.e., accessing its executable BD program.

The IRI and the mP may be mounted on the same board which contains all necessary memory and peripheral interfaces, or may be connected to the system bus.

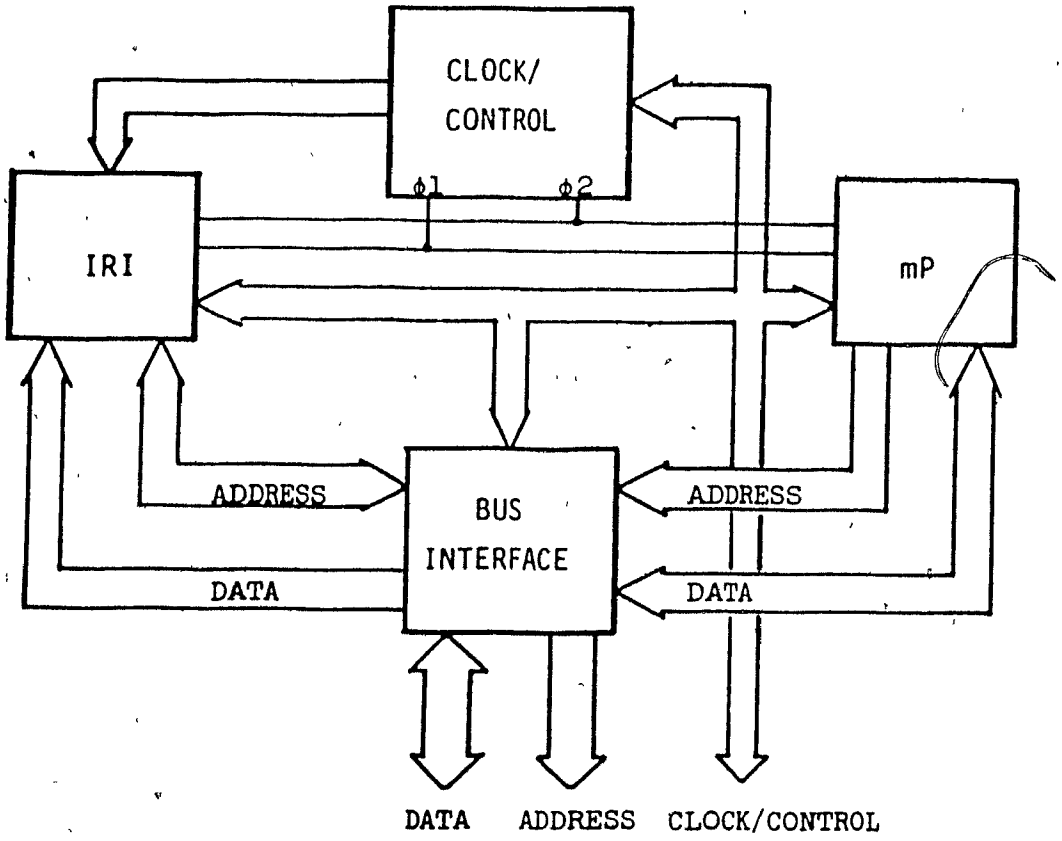


Fig. 5.15: The Shared IRI Controller Architecture.

5.6.2 Clock and Timing

The system has a 2 MHz crystal clock. Its frequency is halved with a toggle Flip Flop. Fig. 5.16 depicts the system timing signals related to data access functions. Non-overlapping $\phi 1$ and $\phi 2$ are generated. To avoid overlapping high clock signals, $\phi 2$ is stretched and kept in its low state beyond the high-to-low transition of $\phi 1$ by inserting 6 gate delays in its clock circuit. This ensures complete separation of bus access between the two processors.

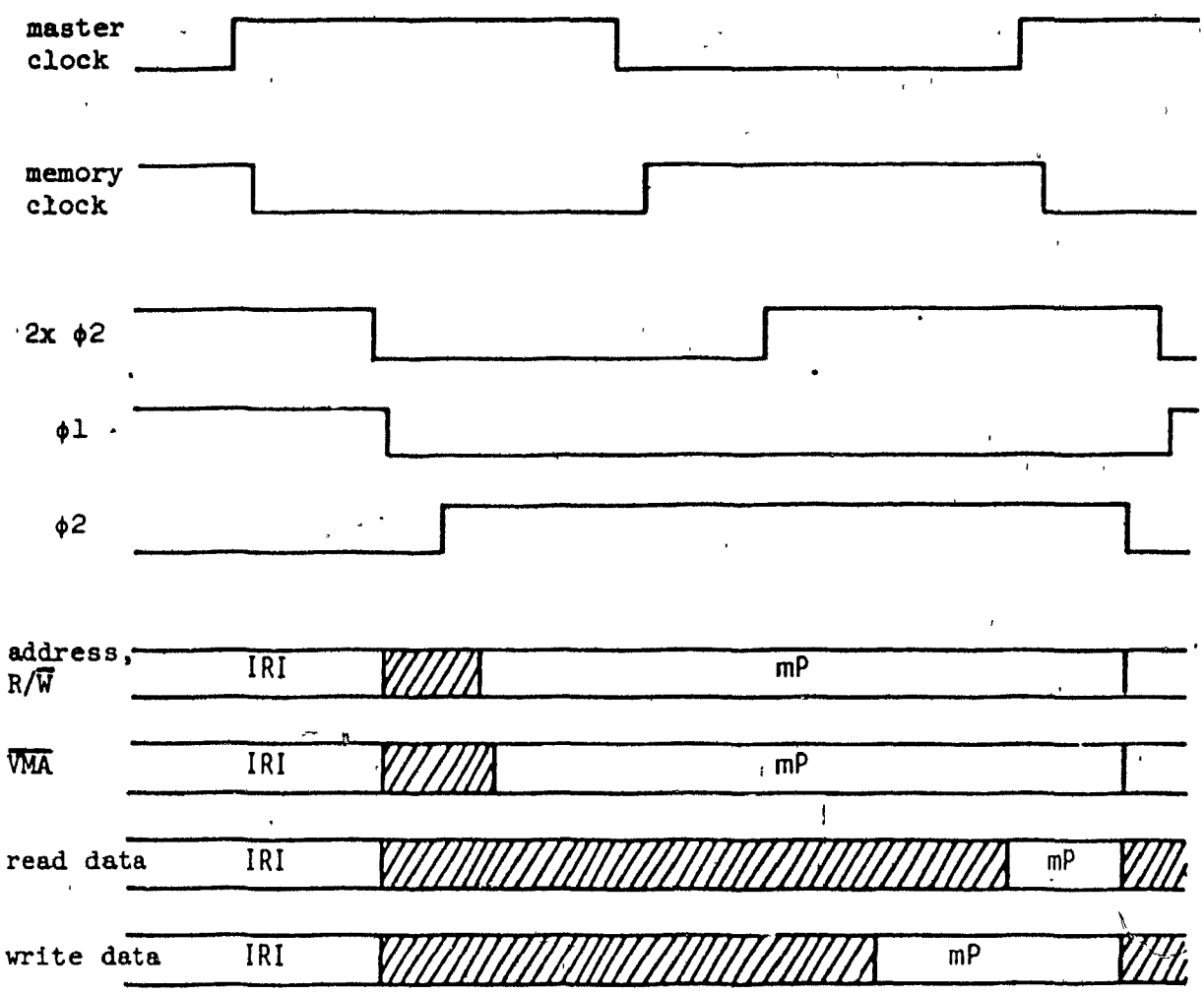


Fig. 5.16: mP and IRI Data Access Timing.

5.6.3 The Bus Interface

The Bus Interface, shown in the block diagram Fig. 5.17, permits the two processors to share memory. Dedicated processor resources (such as memory mapped I/O) are accessed via the same interface. It consists of an address buffering section, data buffering section and R/$\overline{W}$ and $\overline{VMA}$ buffers. The IRI appears as memory to the mP.

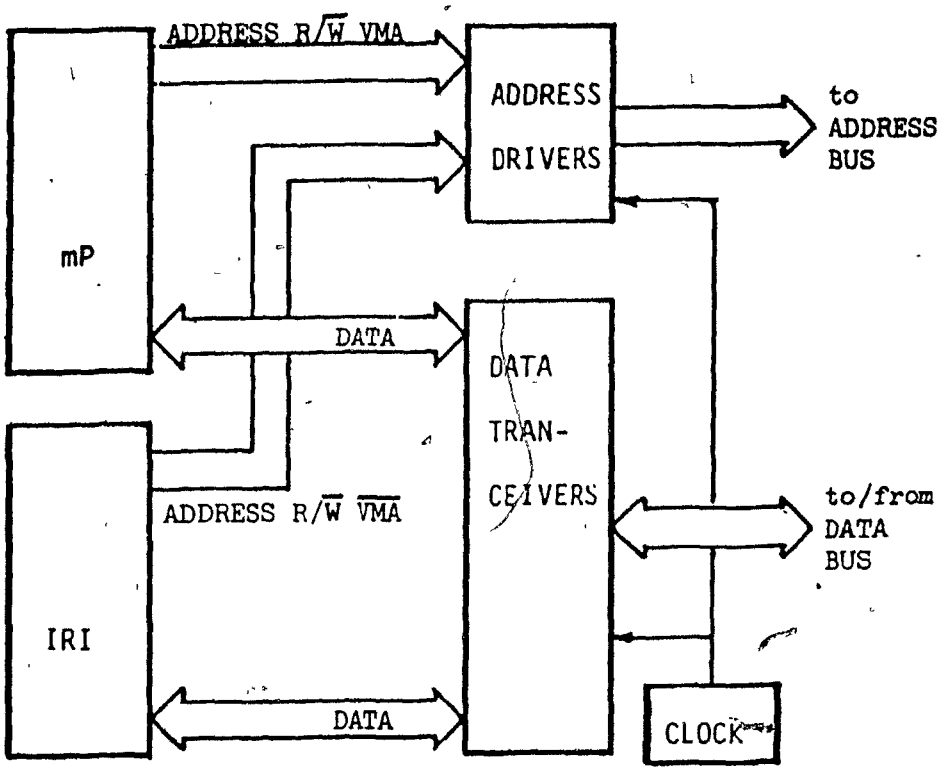


Fig. 5.17: The Bus Interface.

Each processor is interfaced via 16 unidirectional address drivers, plus two unidirectional drivers for R/$\overline{W}$ and $\overline{VMA}$. The IRI R/$\overline{W}$ is tied high, in the READ state. The data buffering section consists of eight data tranceivers for the mP data channel and eight unidirectional data drivers for the IRI data channel. All buffers are enabled by their appropriate processor clock phase. R/$\overline{W}$ determines data direction. $\overline{VMA}$ signal is normally 'low'. It is asserted only to deny bus access when external devices other then the mP or the IRI, (e.g., bulk storage DMAC) request access to the bus.

5.6.4 The Shared IRI Design

The Shared IRI, Fig. 5.18, retains BD and I/O capabilities identical to previous versions. It consists of the IRIP, the 8-bit program counter, the CIL, the BWL, status register, communication PIA, address decoder and page register.

The IRIP is the BD processor employed in all other versions of the IRI. The program counter provides a two byte BD instruction address. I.e., it is connected to bus address lines, A1-A8. A0 is provided by the BWL which also latches the retrieved LS byte of the instruction until the MS byte is retrieved. The page register is an 8-bit write-only register addressable by the mP. Seven bits are employed to assign address lines A9-A15. This is necessary since the IRI can only address 512 bytes. This allows the mP to install a BD memory image in a specific memory page and then set the IRI page to access that image. Communication and status are identical to previous versions.

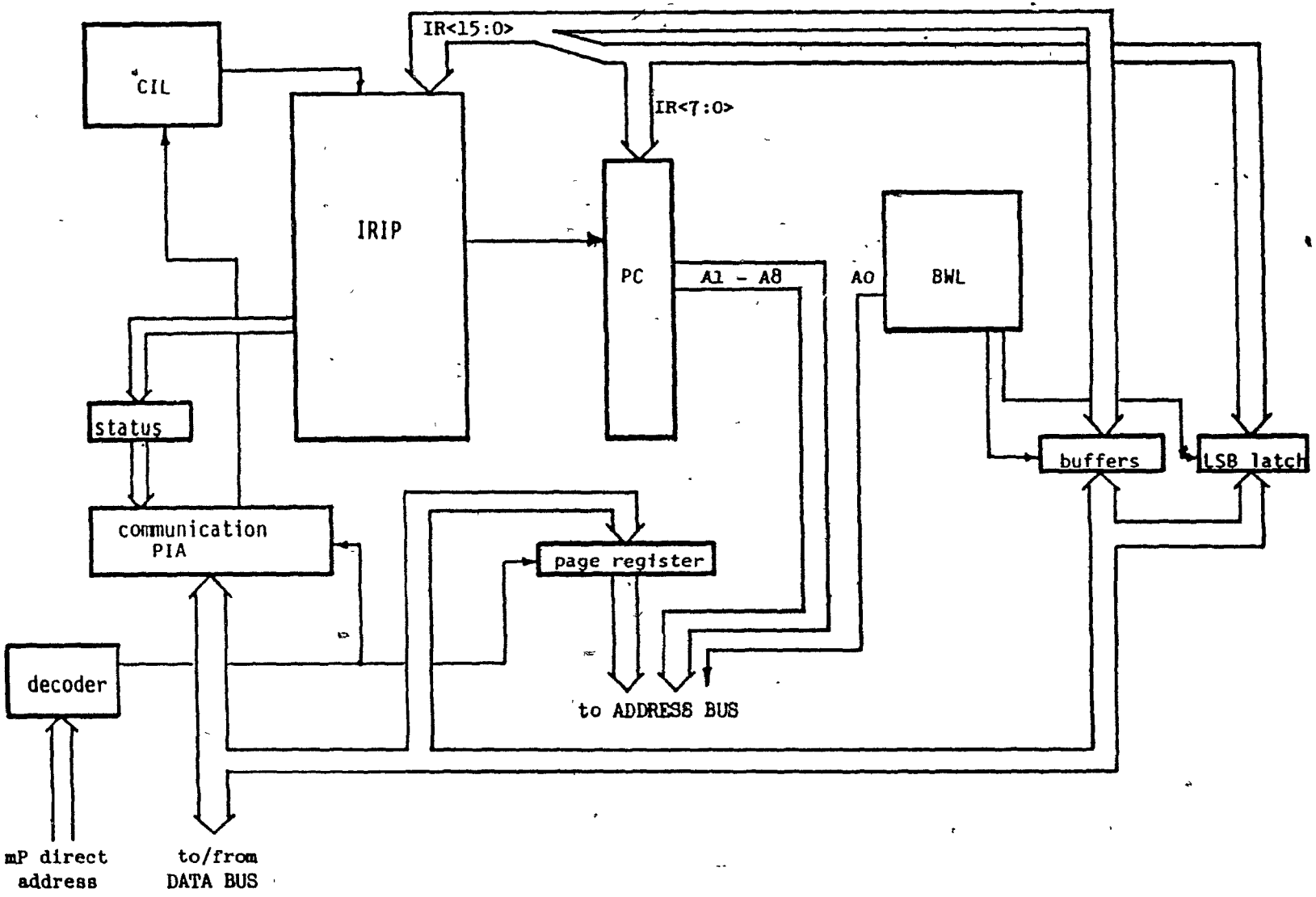


Fig. 5.18: The Shared IRI Layout.

Although both processors are capable of controlling the system's bus, the IRI is a slave processor because all its operations are initiated and monitored by the mP. The IRI can be addressed directly by the mP as follows:

1. Communication PIA - port A - control and test (2 addresses)
   port B - status (2 addresses)
2. Page register (1 address)

5.6.5 The Shared Controller Operation

The mP initializes the IRI by setting the memory page in the page register, presetting the program counter and lowering the STOP flag. The IRI goes into running mode accessing main system memory during Ø1. It needs 2 cycles to fetch an instruction. Normal IRI execution takes place while the mP is free to perform its own tasks. The mP must be prevented from unintentionally altering active IRI memory. To modify an IRI program the mP must follow normal interrupt procedures and halt the IRI first.

The IRI requests a new program by interrupting the mP. If that program resides in system memory, the mP only has to alter the page register and restart the IRI.

Conclusion: Given the two processor limitation, a modular system design cannot be achieved. Nevertheless, BD program alteration can take place the fastest in this minimum hardware configuration. It promises to be a good evaluation system and development facility for control systems designers to explore the capabilities of BD based machines.

5.7 The I/O Modules

IRI I/O modules provide process interface. They reside on the IRI I/O bus (see section 5.3.7). The Binary I/O module is the principal one and forms the basis for the Analog and Timer modules.

5.7.1 The Binary Module

The module shown in Fig. 5.19 contains one output bank (OB), one input bank (IB), selector logic, bank latches and buffers. The module is connected via the I/O bus to the IRI and via signal conditioning units to the process inputs and outputs.

Output bank. The 12 field output latches are connected via opto-isolators to field output lines. When an OPS, short output, instruction is executed by the IRI, the upper four latches of the selected OB output array conform to IR<11:8> of the data lines. In an OPL, long output, instruction all 12 latches conform to IR<11:0>. Outputs can only be affected when the IRI is executing a program in the running mode.

Input bank. This bank contains 16 field input lines connected to a data selector. Four bits, IR<11:8>, serve as address lines to the data selector, which selects one input line for testing by the IRI.

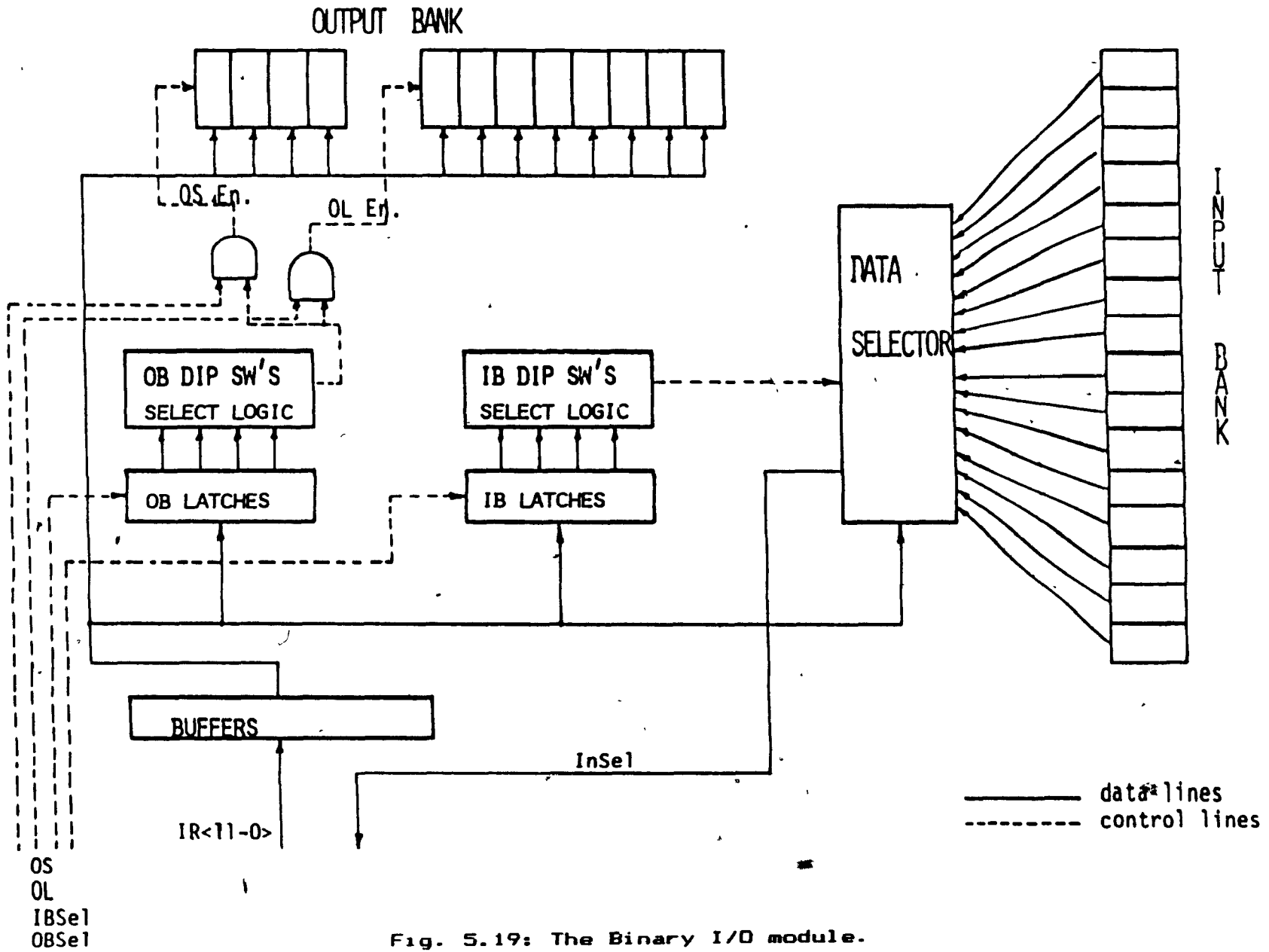


Fig. 5.19: The Binary I/O module.

Selector logic. One IRI may handle up to 16 OBs and 16 IBs. A 4-bit code defines each bank. Two 4-bit latched registers are used to hold the current address codes. The OBSel and IBSel signals, from the respective bank select instruction of the IRI, are used to latch the current bank code. An 8-DIP-switch package is used to assign OB and IB codes. Dedicated selector logic compares the assigned OB and IB codes with the latched values, and outputs the appropriate enable signals. Thus, only a selected bank will be affected by an I/O instruction.

5.7.2 The Analog Module

This module, Fig. 5.20, contains all functions of the Binary I/O module with the addition of Analog to Digital conversion circuitry. Four analog input channels and four analog output channels are implemented. The desired channel is selected externally by four select bits (these may be provided by via another I/O card). Analog input is digitized by a 10-bit A/D convertor whose output is connected to 10 of the 16 inputs of the input bank. The IRI issues SOC (Start of Conversion) to the A/D via the output bank and detects the EOC (End of Conversion) which is connected to one of the remaining six input lines. The digitized value is latched and transmitted to the IRI via 10 LS input lines.

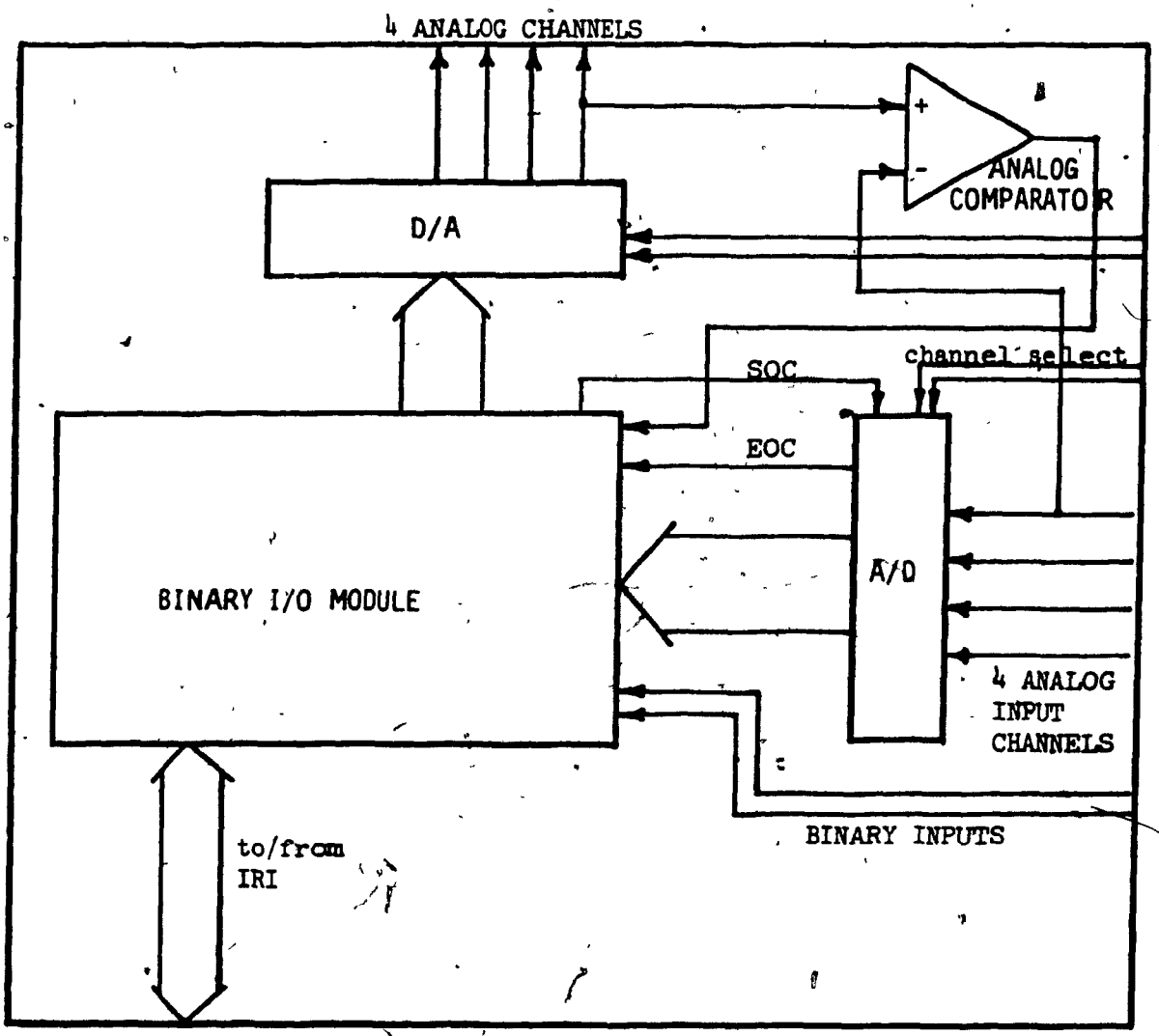


Fig. 5.20:- The Analog I/O Module.

Eight LS output bits may be converted by the D/A convertor to their equivalent analog value which is connected to the process via the selected channel. Three analog channels are available as field outputs. The fourth one can be used internally as a set point for comparison with a selected analog input channel when required. The comparison is executed using an analog comparator on board and the result is available to the IRI via an input line. The remaining input and output lines may be used for monitoring binary signals in addition to the analog ones.

5.7.3 The Timer Module

The Timer module shown in Fig. 5.21 enables timing functions to be executed in hardware under the IRI control. The module consists of eight 555 hardware timers whose timing intervals are hand-set by a potentiometer. Three MS output bits are used by the IRI to address a 3 to 8 demultiplexor which generates the timing request signal R for each of the timers. The fourth bit is the data selector enable.

A timer's state changes from inactive to active when its timing request signal R is asserted. The flag, T->1 remains for the duration of the timing interval. R remains asserted during the timing interval and after its expiry. The timer becomes inactive and ready for a new timing assignment when the IRI resets it.

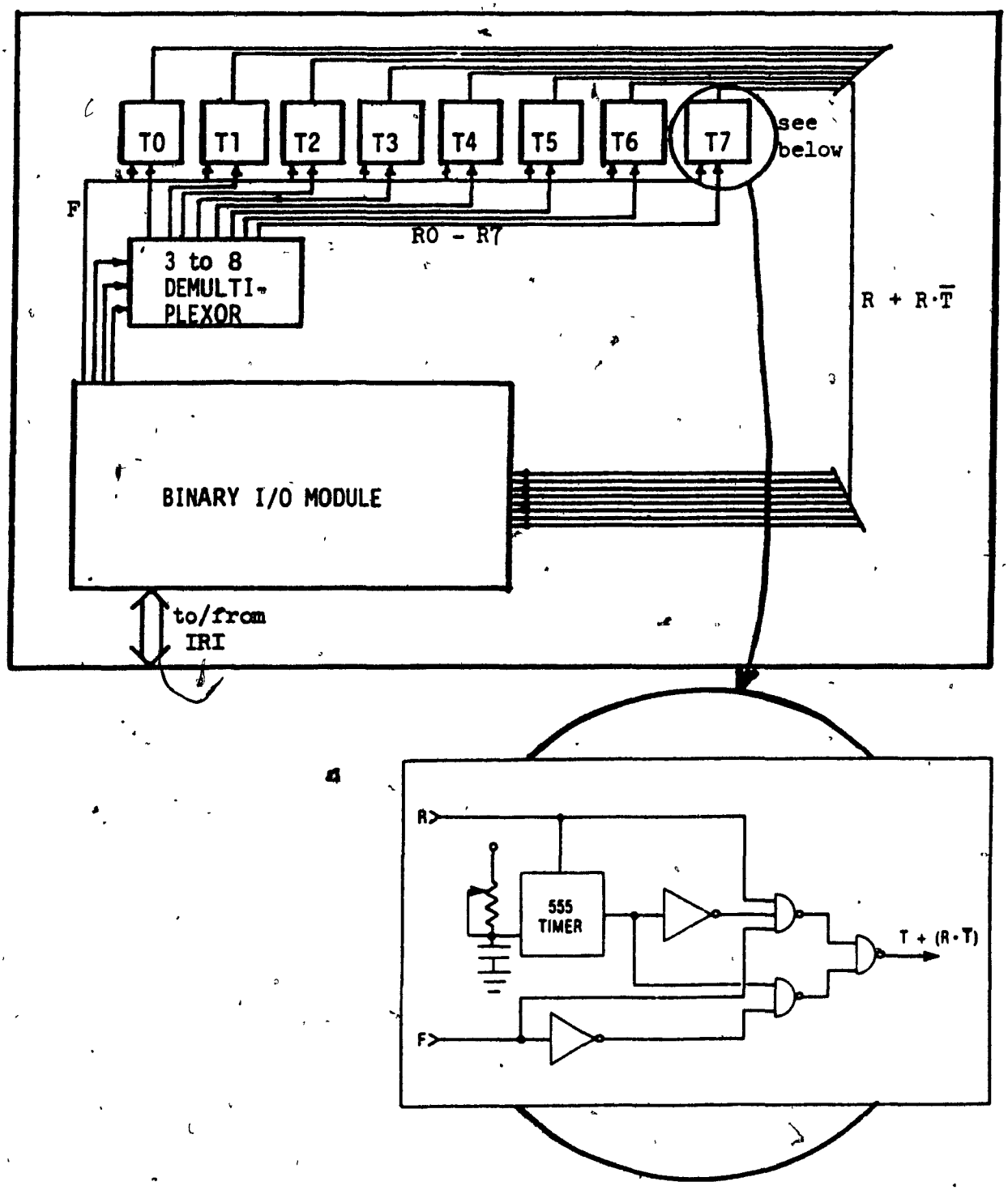


Fig. 5.21: The Timer I/O Module.

The timer can be viewed as a finite-state automaton with three distinguishable states:

1. I - inactive (ready for a timing assignment),
2. M - timing, and
3. X - expired (i.e., timing completed).

The state diagram describing all possible transitions among the three states is shown in Fig. 5.22.

The IRI detects both T and $R \cdot \overline{T}$ in order to unambigously establish the state of a timer. Instead of connecting both signals to the IRI input bank (i.e., requiring 16 inputs for the eight timers), F, an output signal commoq to all timers is used to mask either T (when F is 0) or $R \cdot \overline{T}$ (when F is 1), thus enabling the IRI to monitor both T and $R \cdot \overline{T}$ via a single input line.

Operational modes and variable states of a timer are summarized in Table 5.4:

Mode	R	T	F	$R \cdot \overline{T}$
Inactive	0	0	0	0
			1	0
Timing	1	1	0	1
			1	0
Expired	1	0	0	0
			1	1
Illegal	0	1	x	x

Table 5-1: Timer's Operational Modes.

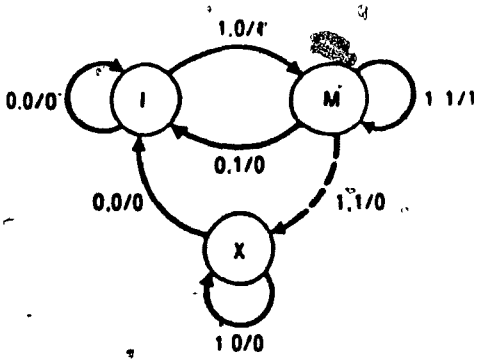


Fig. 5.22: Timer State Diagram.

CHAPTER 6: APPLICATIONS

6.1 Introduction

In this chapter two applications are presented to demonstrate the advantage of an IRI based controller over a conventional mP based control system. The first example is the control of an industrial batch weighing and mixing process. It shows how an IRI based hybrid lowers cost and complexity of control hardware. The second application shows how an IRI pre-processor increases variable reference matrix sensor data throughput so that it can track quickly changing dynamic situations.

6.2 Industrial PLC, Batching Process

6.2.1 Process Description

Batch weighing and mixing are common to many industrial processes: e.g., chemical, food, textile, glass and cement production. Fig. 6.1 shows a typical food or animal feed plant mixing process.

The production line includes:

20-25 ingredient (raw material) bins,
2-3 liquid tanks,
2-3 weigh scales of different ranges,
1-2 flow-meters,
1-2 mixers,
0-1 surge bins,
10-15 finished products storage bins, conveyors and feeders.

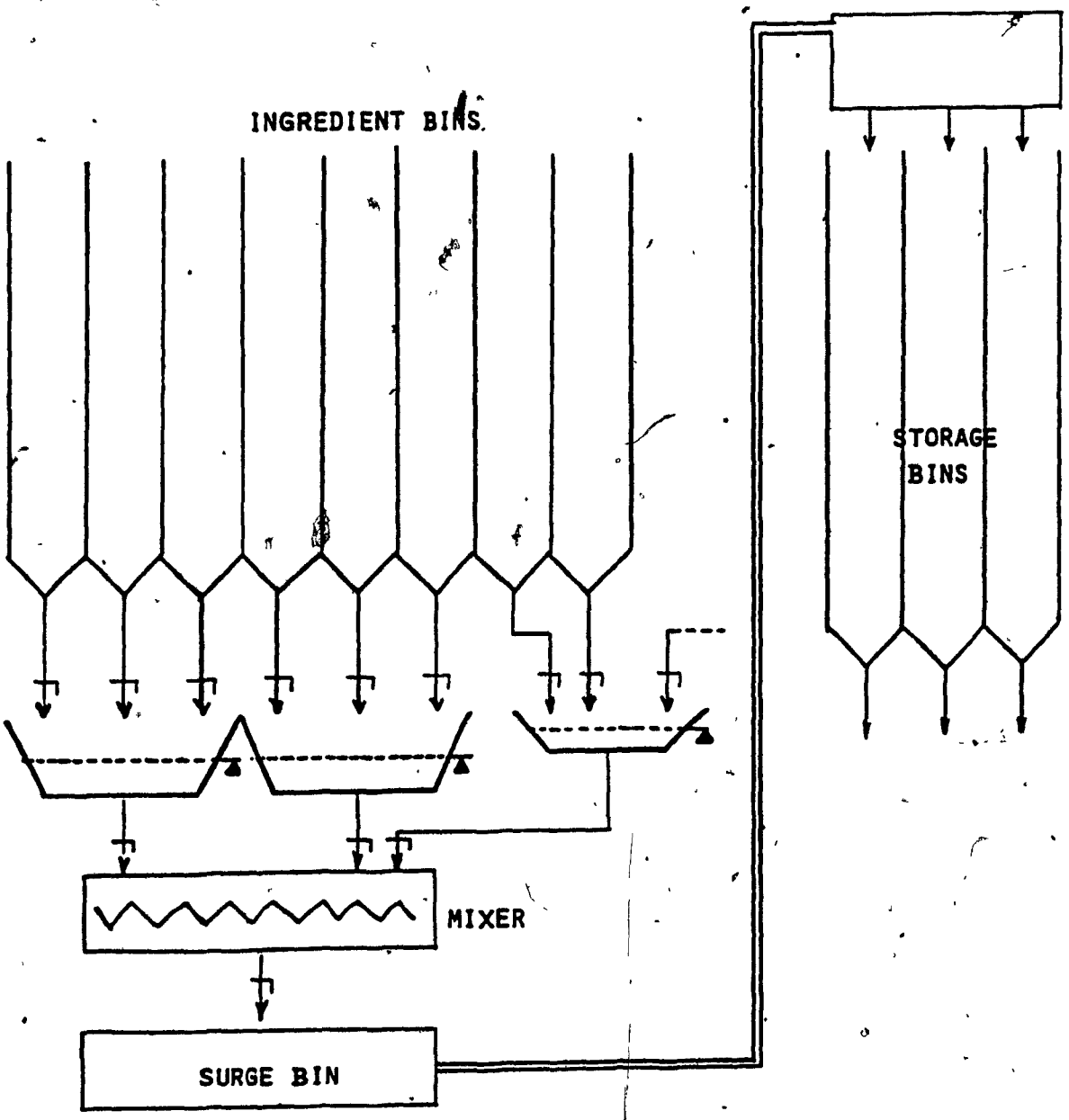


Fig. 6.1: A Typical Batching and Mixing Process.

The process of producing a batch involves three stages:

1. weighing the ingredients according to a prescribed batch formula,
2. mixing the batch for a prescribed period of time, and
3. routing it to a designated storage bin for further processing or delivery.

All scales can weigh simultaneously. Once a batch has cleared the scales it is transfered to the mixing stage and the next one may be started. The process is sequential but "pipelined". A complex system may have many weighing lines and batch processing stations, while a simple one, such as a bag packaging station, may contain only one weighing line.

Fig. 6.2 shows a typical weigh scale with its associated control signals and timing diagram. Material weight is measured in the scales by a straingauge type load-cell. The weighing part of the process involves three phases:

1. Material feed; coarse,
2. Material feed; fine, and
3. Scale discharging.

To achieve minimum deviation from target weight at maximum feed rate, an ingredient is fed to a weigh scale in two stages; coarse and fine. The control system which operates the feeders is provided with coarse and fine cut-off points (c.o.p.). These are determined by the nature of the ingredient and its afterflow characteristics and are constantly updated to correct actual weight deviations.

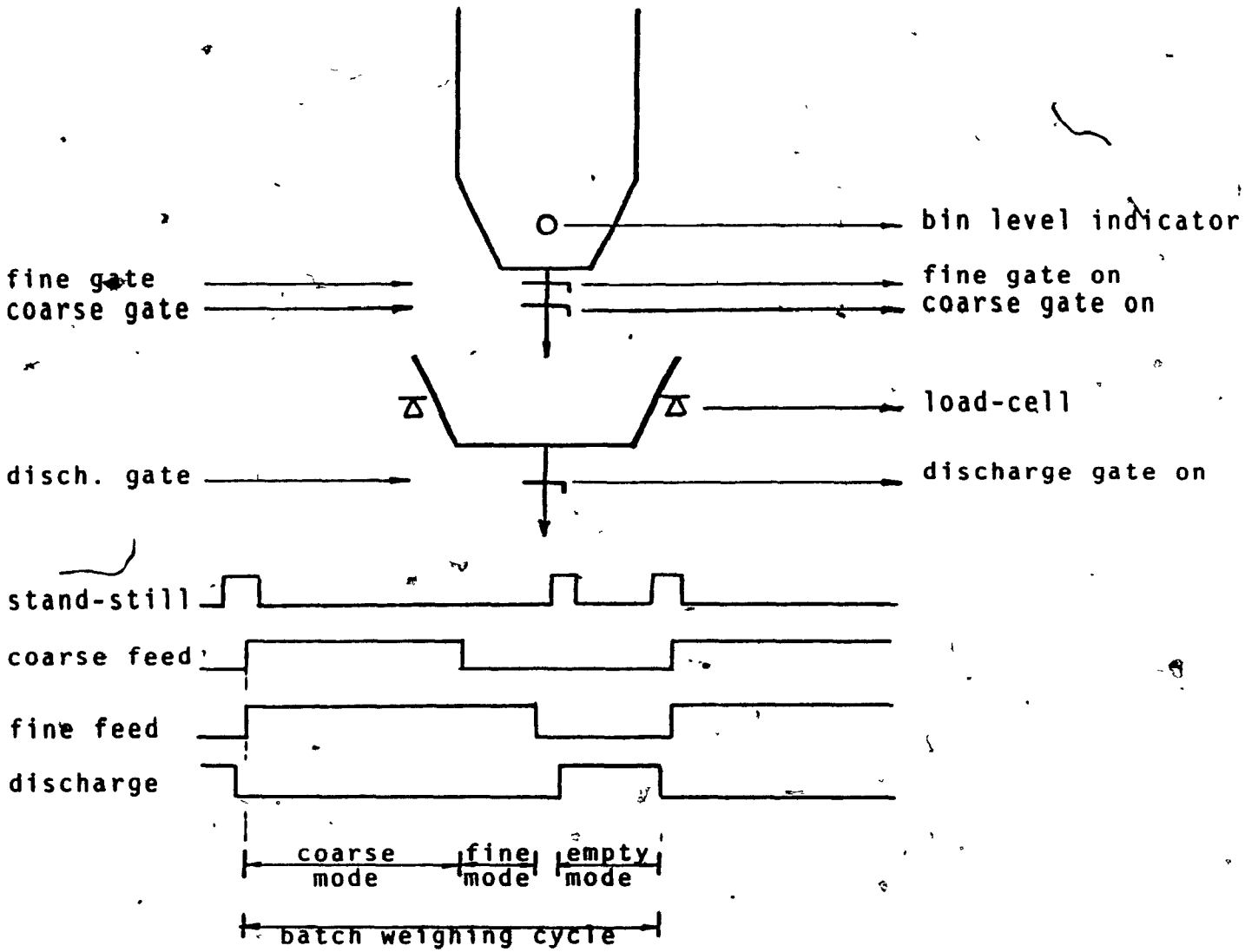


Fig. 6.2: Weigh Scale Control Signals

The control system verifies scale-empty condition prior to feed initialization, and registers tare weight on the scale. It then closes the discharge gate and opens both coarse and fine feed gates. The weight is monitored continuously until it reaches the coarse c.o.p., when the coarse feed gate closes. The system continues in the fine feed mode until the fine c.o.p. is reached. After closing the fine feed gate, the control system waits to ensure that all flowing material which was downstream of the gate has come to rest on the scale. Afterwards the actual weight is measured and compared with the target weight. The difference is used by a c.o.p. adjustment algorithm to improve the accuracy of future batches.

6.2.1.1 Cut-off Points Optimization

Fine and coarse cut-off points are adjusted, after each weighing, according to an optimization algorithm which aims to:

a) minimize the difference between actual weight and target weight, and
b) maximize the fraction of weighing time spend in coarse mode so as to shorten the weighing cycle.

An ingredient target weight is prescribed by the batch formula. Cut-off points are dynamic parameters, assigned as percentage value per ingredient. The fine c.o.p. is set at slightly less then the target weight to compensate for material which flows to the scale after the feeding gate is closed. The coarse c.o.p. is set to a point which can vary

between 80% and 95% of the target weight. To achieve faster weighing the coarse c.o.p. is adjusted to increase the coarse portion of the weighing interval. This can be done as long as the material flow characteristics remain stable and larger weighing errors do not occur as a result of the shorter fine weighing interval. For example, one simple weighing algorithm invokes an earlier fine c.o.p. if an overweight occured. The adjustment might typically shorten the fine feed interval by a percentage, say, 60% of the measured overweight error. In the case of shortfall, the adjustment might extend the fine feed interval by the estimated time needed to supply the entire underweight amount. A more elaborate algorithm might incorporate statistics and a dynamic model of the material flow process. It is desirable that a weighing error converges to zero. Weighing algorithms have been developed for many applications. The design or selection of an algorithm depends on the material stability, weighing restrictions, i.e., negative or positive tolerances and other factors. Real-time c.o.p. optimization takes place while scales are discharging. Optimization tasks are not subject to real-time constraints as those imposed by DDC response requirements.

6.2.1.2 Smart/Reflexive Partition of Control Functions

In order to illustrate the advantage of using an IRI based controller in such a batch weighing process, the control functions will be divided into reflexive, reflexive-aided and smart categories as described in chapter 5.

Reflexive Tasks:

Monitor weight during feeding

Monitor weight during discharge

Monitor system parameters:

gate positions,

air-pressure, etc.

Detect and annunciate malfunctions

Control mixing time

Control routing to storage bins

Reflexive-aided Tasks:

Initiation of new weighing sequences

Treatment of process irregularities and malfunctions

Smart Tasks:

Supervision of Interface Processors

Operator interaction

Maintenance of data bases:

ingredients - c.o.p.s, tolerances

batch formulae

production data

bin-material data, etc.

Registration and updating of actual weights

C.o.p. adjustments and optimization

Optimization of formula

Bin assignment, Scale assignment

Generation of statistical data

6.2.2 IRI Controlled Batch Weighing

A microcomputer-only control system is capable of performing the control tasks described above. The number of scales that can be controlled and the number of other parameters that can be monitored is limited by the complexities of the operations and the speed of the process. In addition, any supervisory task such as synchronization of parallel operations, data management, optimization of batch formula, etc., will be delayed while the cpu is occupied with process tasks. Several microcomputers may be required to control a complex process each of which is dedicated to a separate part of the process. An additional microcomputer may be employed for supervision, statistics and optimization. One commercial system [IA81] employs a large computer system, i.e., a minicomputer or even a mainframe, to handle control tasks and a backup computer which handles optimization and other off-line tasks during normal operating conditions.

A single microcomputer augmented with IRI modules can perform tasks which conventionally employ large computer control systems. By partitioning the control tasks and by letting the IRI modules perform the reflexive ones, the cpu is relieved from routine but time critical monitoring and control tasks. Smart, supervisory tasks are thus handled more efficiently.

Due to the fast scanning capabilities of the BD processor, a single IRI is able to simultaneous handle parallel operations without incurring any time penalty. E.g., the weighing operation may be controlled by an IRI which at the same time monitors material levels in ingredient bins and sufficient air pressure in pneumatic feed lines.

6.2.2.1 The IRI Based Controller Architecture

An IRI based modular programmable controller, Fig. 6.3, used to control the process described in section 6.2.1, employs one mP with four IRI modules. Three IRI modules are assigned to control three weigh-scales simultaneously, as well as to monitor their respective feeders, ingredient bin sensors and other parameters. The fourth IRI controls the mixing operation and the batch routing to the proper storage bin.

Each of the weigh-scale IRIs contains as many binary I/O cards as needed for on/off control. One Analog I/O card is employed for the weigh-scale load-cell signal monitoring, whose signal is also provided, via a dedicated PIA, to the mP. Via another PIA, the mP provides each of the IRIs with the current c.o.p. In addition to monitoring the weight, the IRI monitors and controls the scale feeders via binary I/O.

The mixing IRI employs a timer card to control the mixing time. The mP supervises the mixing operation via the downloaded BD program. The IRI issues the control signals to empty the mixer and route the batch to its next destination. In addition, it communicates with the other IRIs via their binary I/O cards to permit scale discharge to the mixer.

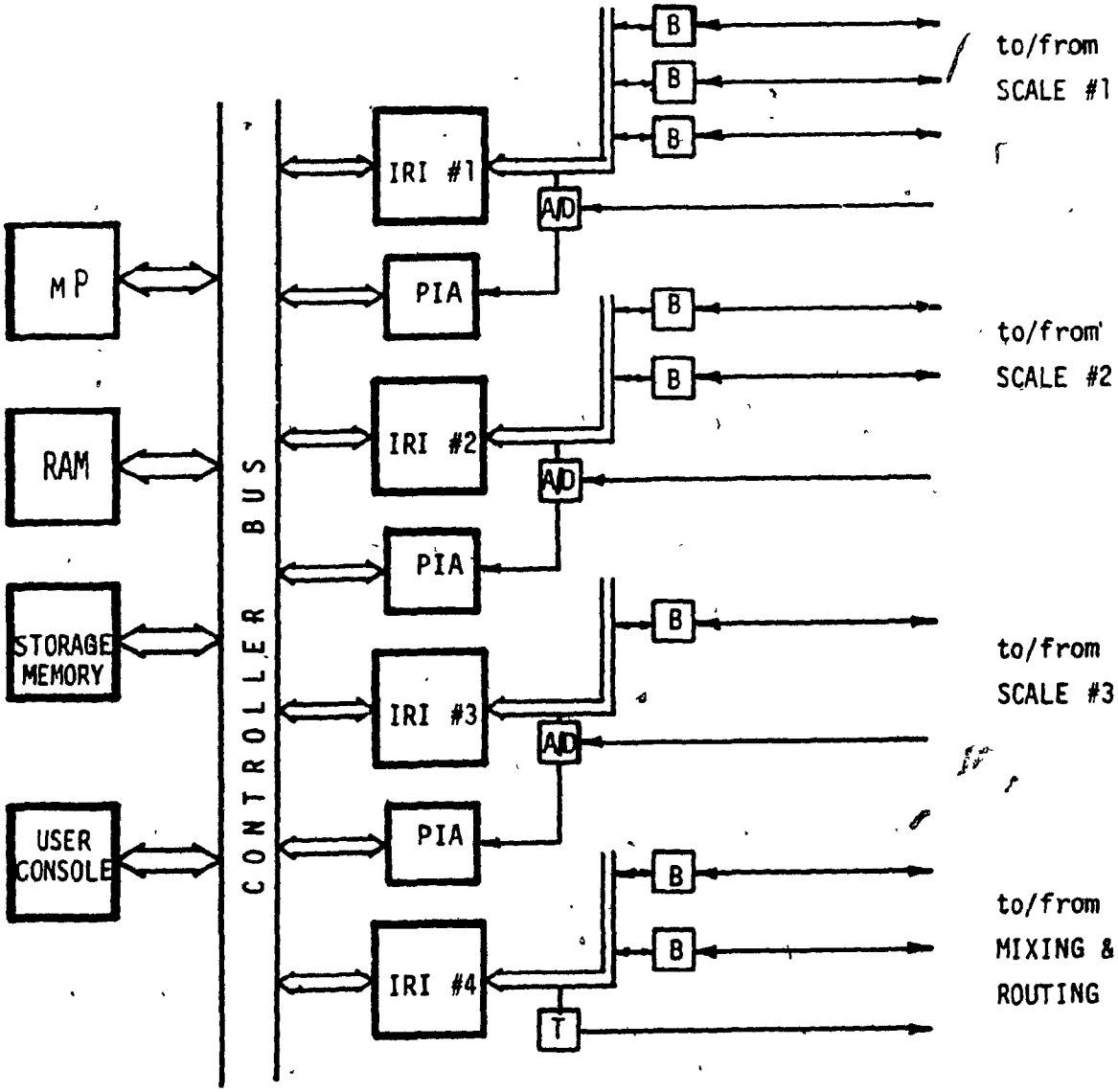


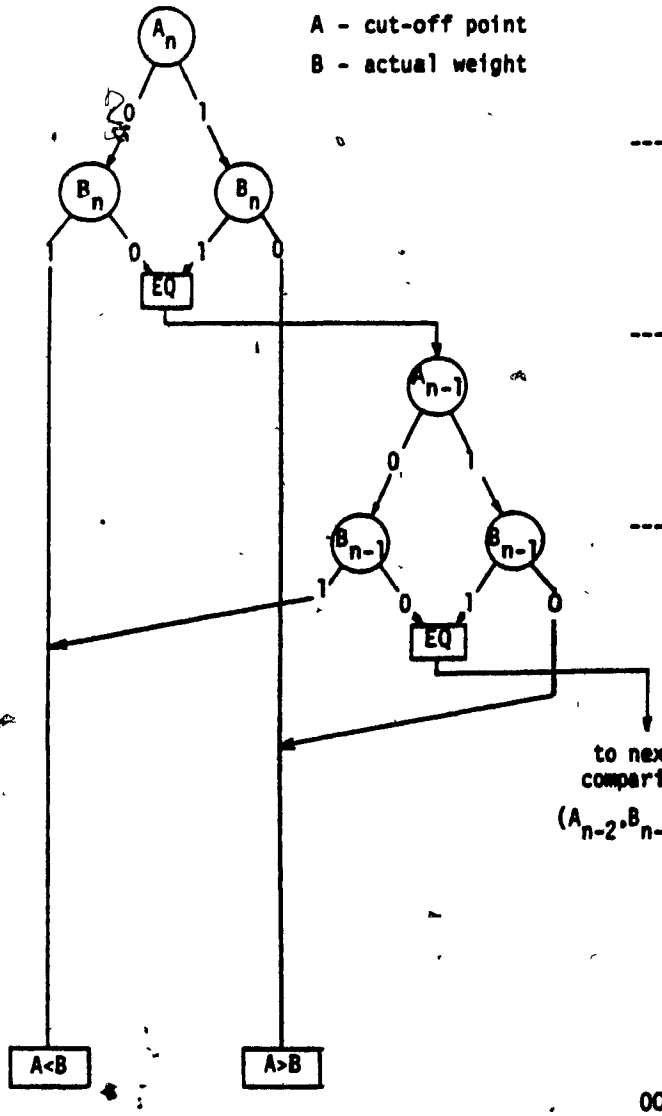
Fig. 6.3: An IRI Based Programmable Controller.

6.2.2.2 Weight Monitoring by IRI

Current weigh-scale charges have to be constantly measured and compared to their c.o.p. Measurement rate is determined by the permitted weight tolerance and by the feeder throughput. A binary sequential comparison technique which replaces the arithmetic computation used by microcomputers, reduces monitoring time considerably.

An IRI module, employing two input banks compares two 12-bit binary numbers, one is the actual weight provided via an A/D from the load-cell, and the other is the c.o.p. furnished by the mP via a PIA. The BD comparison tree is shown in Fig. 6.4 along with its associated BD program. This program is executed on a cyclic basic with additional control programs (see Appendix B) which monitor other process parameters. The mP waits for an 'end of weighing phase' interrupt to take action (change c.o.p. or download new program). The MC6809 weight control assembler routines are listed in Appendix B.

Because the actual weight is less then the c.o.p. throughout the weighing cycle, comparison always starts with the most significant bits. A one bit comparison is sufficient until the actual weight reaches half c.o.p., two bits suffice until 75% of c.o.p., and so on. With a 12-bit weight representation, the full 12-bit comparison tree is executed only when the weight is at or very close to c.o.p. Even the full comparison will not exceed 24 clockcycles. More significantly, only a few clock cycles are needed most of the time for each load-cell comparison, allowing one IRI to concurrently serve many weighscales.



BD Program

```
0000  IBSel
0001  OBSel
-------------------------------------
0010  TEST(1) An  jump to 0101
0011  TEST(0) Bn  jump to 0110
0100  BRA to 00110100 (A<B)
0101  TEST(0) Bn  jump to 00110101 (A>B)
-------------------------------------
0110  TEST(1) An-1 jump to 1001
0111  TEST(0) Bn-1 jump to 1010
1000  BRA to 00110100 (A<B)
1001  TEST(0) Bn-1 jump to 00110101 (A>B)
-------------------------------------
1010
.
.
.
.
.
.
.
.
.
.
.
.
.
00110100  OUTPUT A<B  jump to Interrupt
00110101  OUTPUT A>B  jump to EOP
00110110  OUTPUT A=B  jump to Interrupt
00110111  Interrupt
00111000  EOP
```

Fig. 6.4: A Multi-bit Comparison BD Tree.

The mP can read the current weight at any time because the A/D output is also available via a PIA port. Under normal runnning conditions this occurs only when a "comparison-completed" signal is sent by the IRI. The mP goes into a check-loop to verify stand-still condition. It then registers the final actual weight and issues a signal to the IRI to open the discharge gate. The IRI handshakes with the mP and the latter registers the weight, deducts the residue weight that existed in the scale before feeding and saves the value for data management purposes. Cut-off point adjustment is done at this stage in case of deviation from target weight. The discharge gate is opened by the IRI, which then monitors weight until the scale is empty.

The mP initiates and supervises the operation. The operator selects the formula to be produced and the mP, using the necessary information available in the formula and ingredients file, provides the IRIs with the ingredient weights, cut-off points and operation sequences.

6.3 IRI Based Scanner

The IRI's capacity to process digital signals quickly is exploited in this second application example. The IRI based control system is used to pre-process signals detected by an array or matrix of binary sensors. The sensors compare a measured parameter with a controlled reference parameter which is dynamically varied. Such variable reference sensors can be devised to measure many different physical properties. The IRI preprocessor functions to increase the system's measurement rate and allows transducer arrays to be used in dynamic applications.

A matrix of comparators, with variable reference, can be used to measure the values of an analog parameter at a number of spatially distributed points, e.g., the pressure distribution over a surface. Existing, conventional methods for parameter mapping use:

a) a single analog transducer multiplexed to a number of points in the domain, or

b) individual transducers distributed at various points.

The former method is speed limited by the switching and settling time of the multiplexed analog system. Individual analog transducers are capable of fast measurement but at the high cost of many transducers.

A compromise between the high cost and complexity of an individual transducer matrix and the limited scanning speed of a multiplexed transducer, can be achieved by using a binary comparator matrix with the very fast BD based IRI pre-processor.

One practical application in which this BD based IRI multiplexor approach can be used, is to map, for diagnostic therapeutic purposes, the contact pressure under a seated patient, suffering from neurophysiological disorder.

6.3.1 A Matrix of Comparators with Variable Reference

A comparator matrix consisting of simple inexpensive digital transducers was constructed. It compares the external parameter, P_m, with a common internal reference parameter which monotonically increases -or decreases- over the measurement range, P_i such that if $P_m<P_i$, the output is 0, and if $P_m>=P_i$ the output is 1. By labeling the occurrence of the transition with the internal reference parameter value, a table of external parameter values can be obtained.

The dynamic behaviour of the binary sensor is illustrated by the series of events associated with the occurence of a transition, Fig. 6.5. If coincidence between the reference and the measured parameter occurs at time t_m, the sensor will make a transition later, at t_s, where t_s-t_m is a measure of the sensor threshold. Scanning might not detect the transition until another delay, t_m-t_s, has elapsed. The monitoring system issues the command to measure the reference value at t_m, so the measured value is derived from the reference value

at t_m, which might differ considerably from the true value which existed at t_a. For accurate results, especially in a dynamic environment, it is important to minimize these delays.

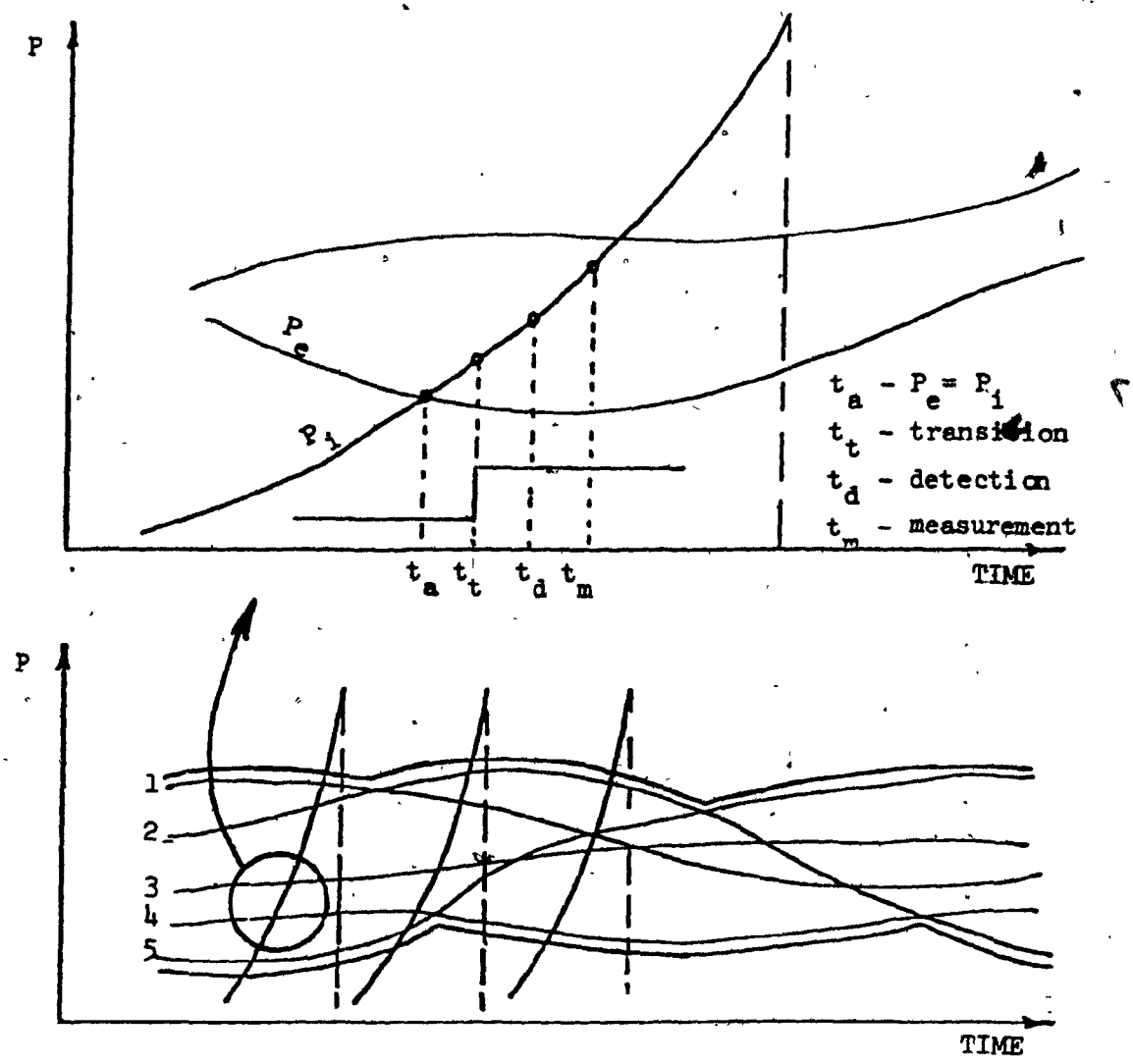


Fig. 6.5: Scanner and Sensor Dynamics.

6.3.2 Comparator Matrix Augmented with an IRI Controller

A comparator matrix system, capable of fast sampling, can be configured by employing an IRI based controller to monitor sensor transitions. The BD based IRIs scan all sensors rapidly, detect sensor transitions and assign an input measurement reading to each sensor transition with minimal delay.

The IRI based controllers presented here are designed around a 16x16 sensor matrix, i.e., 256 sensors. IRI systems are modular and can be easily adapted to different matrix sizes.

The IRI scanner controller block diagram is shown in Fig. 6.6. Each IRI controls a segment of four matrix lines, 16 sensors each. I.e., four IRI modules are required. Each IRI employs a number of binary I/O modules connected to the matrix via a special scanner interface. Three IRI controllers with different scanning algorithms and scanner interface structures were designed:

1. The Single-Sensor Scanner Interface
2. The Line by Line Scanner Interface
3. Scanner Interface with Memory

Elements common to all three scanner interfaces are the Line Image Registers (LIR) which store the status of all sensors connected to the interface by accumulating sensor transitions throughout the measurement cycle. Using information stored in the LIRs, the scanning algorithm ignores sensors whose transition took place before a previous scan.

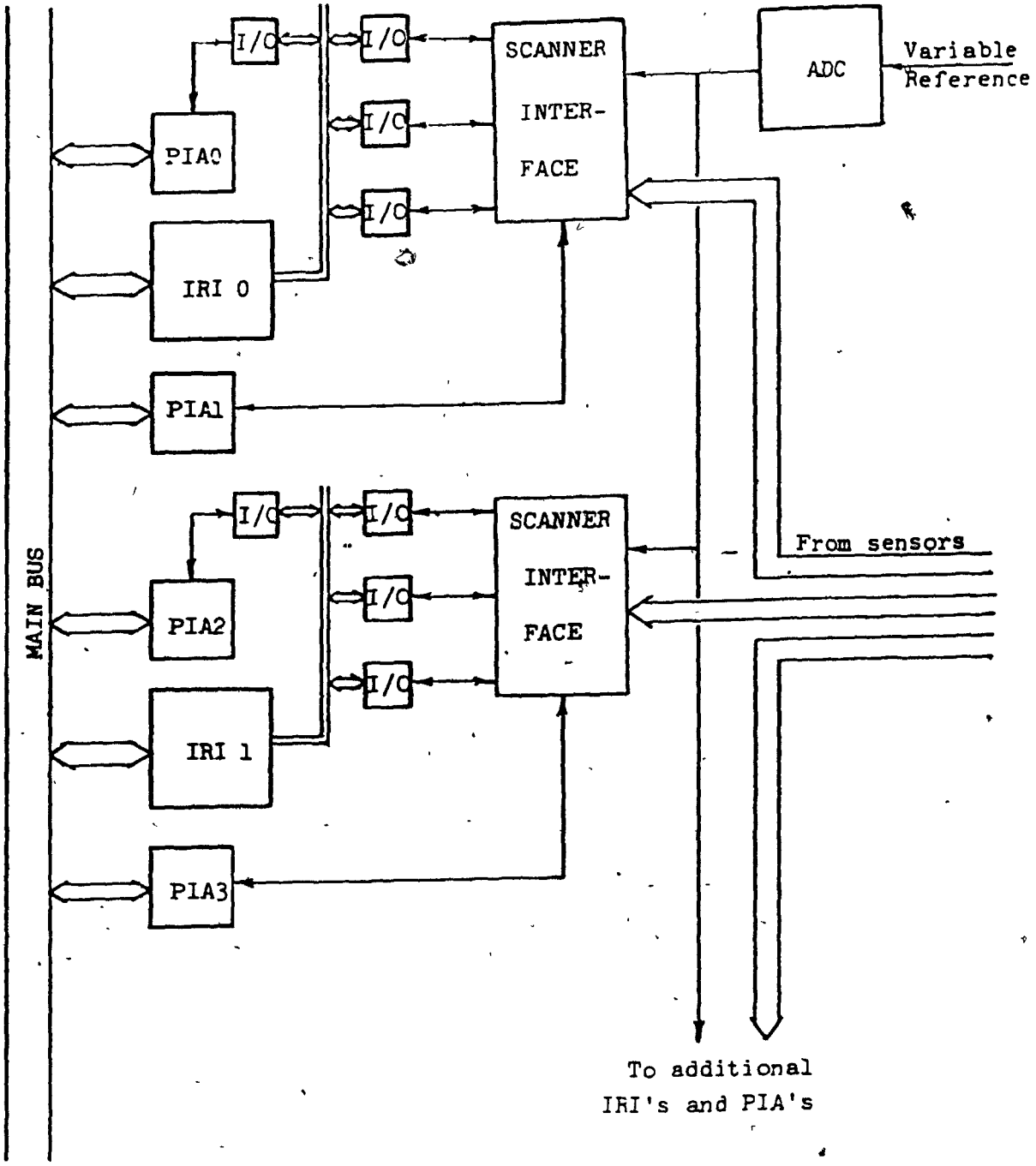


Fig. 6.6: An IRI Based Scanner - Block Diagram.

The three designs can be described by the manner in which the IRI preprocessors report their findings to the mP. In the Single-Sensor version a single transition is reported each time it is detected. In the Line by Line version a complete matrix line is reported if one or more transition occurred in the line. In the Memory version all transitions, i.e., their corresponding reference values, are stored in the interface and are transmitted to the mP for post-processing at the completion of the measurement cycle.

The variable reference signal may be analog or digital depending upon the application. If an external source generates the reference such as air pressure (see example, section 6.3.6) then the analog variable reference signal has to be digitized by an ADC. The ADC output is recorded whenever a transition is detected. In applications using voltage comparators, the reference may be generated by a micro or BD driven DAC, in which case the digital reference value is already available.

6.3.3 The Single-Sensor Scanner Interface

The Single-Sensor Interface is shown in Fig. 6.7. Each interface card is dedicated to one IRI module which monitors a four line region of the matrix, each line has 16 sensors. The scanner interface contains four LIRs composed of S-R latches which store the image of a 16 sensor line. Four 4 to 16 demultiplexors are connected to the data inputs of the four LIRs. An additional 2 to 4 demultiplexor uses the 2-bit line address to generate a signal which enables one of the 4 to 16

demultiplexors. The 8-bit latch accepts the digitized reference value at its inputs and latches it when enabled by the IRI.

What follows is a brief description of how the IRI, via the scanning algorithm elaborated in 6.3.4, below, treats the sensor auxiliary interface. The IRI compares each sensor with its corresponding historical bit stored in the LIR. If a sensor transition had already occurred, as indicated by the corresponding LIR bit, that sensor's input line is ignored, i.e., not checked by the scanning algorithm. When the IRI encounters an up-to-now undetected sensor transition the following activities ensue:

1. The current digitized reference value is stored in the 8-bit latch, connected to the mP via a PIA.
2. The sensor's address is transmitted to the mP via a PIA.
3. The corresponding LIR bit is updated.

The mP stores the reference value against the 6-bit sensor address, composed of a 2-bit line number and a 4-bit position in the line, in its memory to be post-processed at the end of the measurement cycle. In addition, the relative sensor location is transmitted to all four demultiplexors, one of which, addressed by the line number, is enabled via the 2 to 4 demultiplexor.

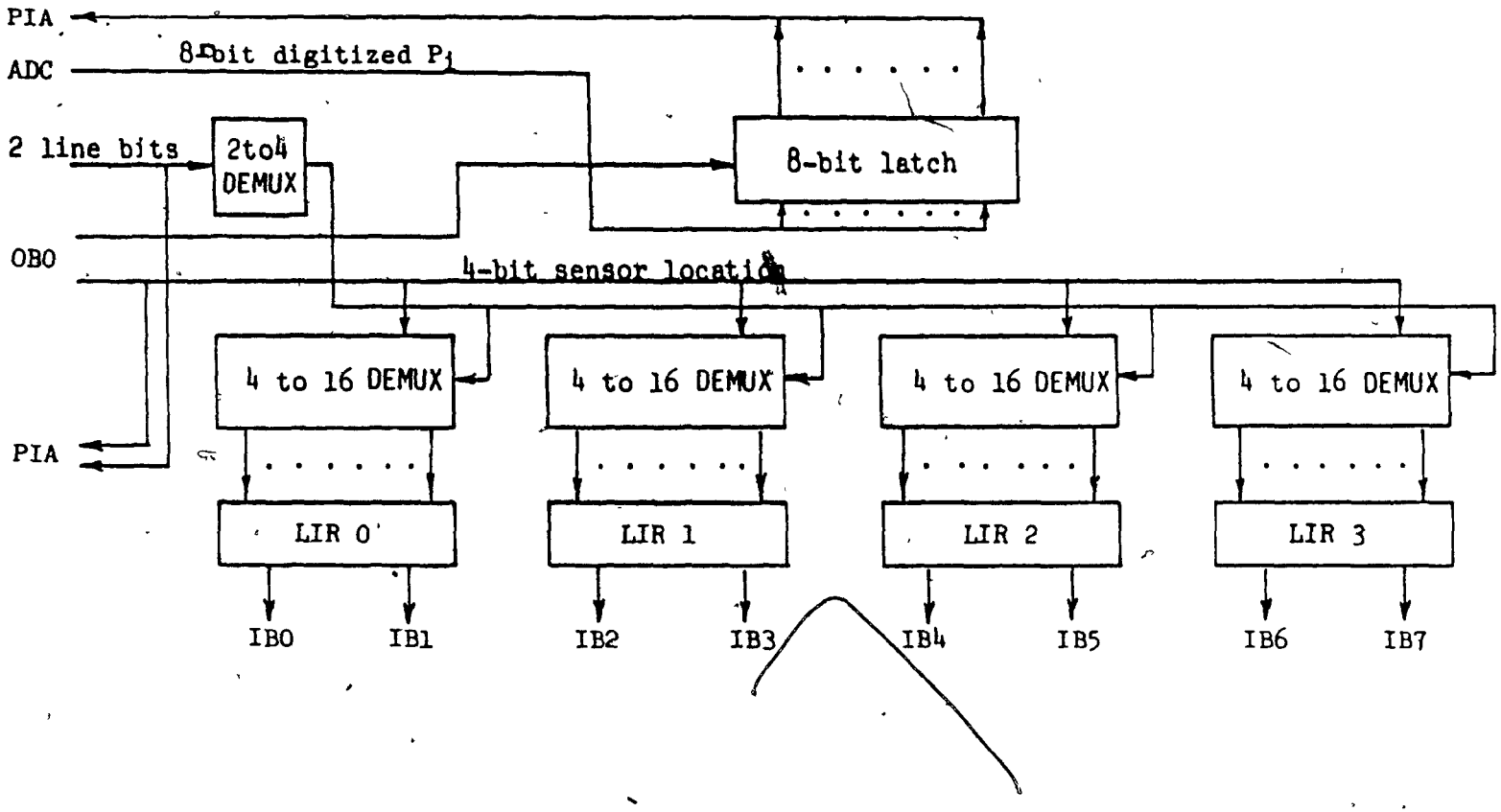


Fig. 6.7: The Single-Sensor Scanner Interface.

Each IRI therefore requires:

1. Eight input banks, two for each sensor line and its corresponding LIR.
2. One common output bank which transmits the 4-bit sensor location and the 2-bit line address to the interface and to the mP.
3. A second output bank and an additional input bank to communicate with the mP and to report the current line status.

The mP uses the six output bits, connected to one side of a PIA, as the reference value storage address. The reference value itself is available at the other side of the PIA. The IRI is temporarily disabled using an interrupt-handshaking routine until the mP has read the PIA.

This scheme incurs the least amount of microcomputer overhead but requires a significant amount of external hardware, i.e., nine input banks, two output banks, four LIRs and one PIA for each IRI. Using a 1 MHz clock for both the IRI and the micro, this scheme can resolve the entire 16x16 matrix in less than 10 msec. A considerable disadvantage is that only one transition is reported at a time. An IRI which is waiting to report a detected transition is idle while the mP responds to other IRIs. In the meantime additional transitions, under the scrutiny of the idle IRI, may occur. An unfavorable sequence of events may result in an unreasonable delay before certain sensor transitions are detected.

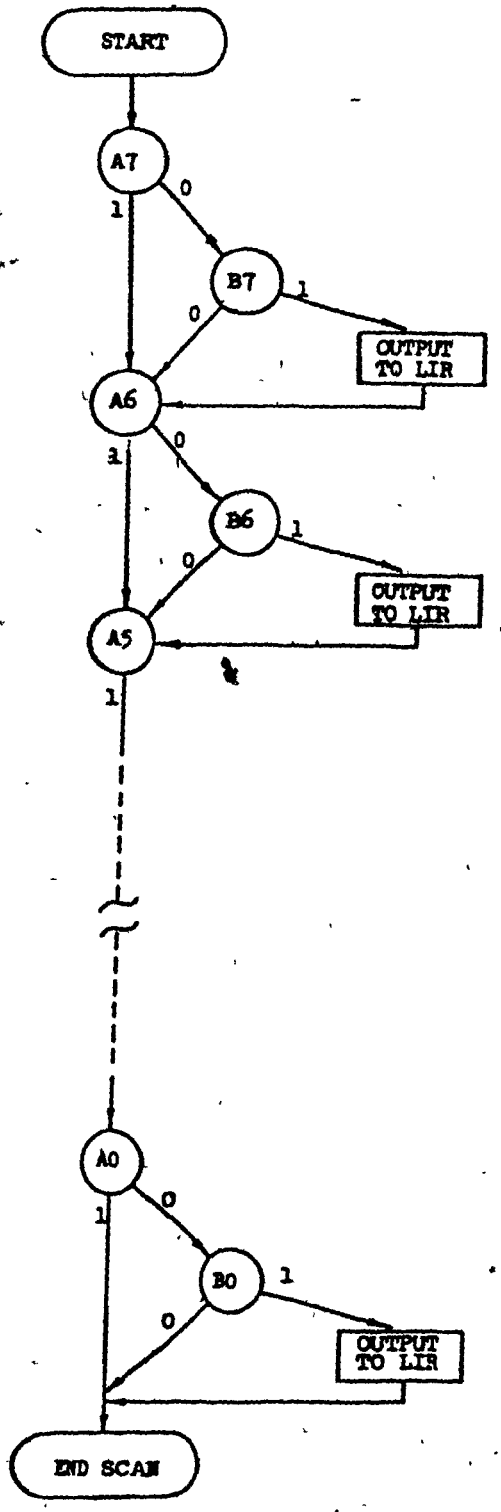
6.3.4 BD Scanning Algorithm

The scanning algorithm was designed to serve two purposes:-

1. To quickly detect a transition of a binary sensor. This transition, which occurs when the variable reference reaches the level of the measured parameter at the location of the sensor, is of paramount importance because it identifies the value of the measured parameter which is latched by the IRI and transmitted to the micro.
2. To increase scanning speed by eliminating all previously activated sensors from the scanning cycle.

Throughout the scanning cycle the BD routine continuously compares an input from a binary sensor (Bxx) with its historical status stored in the LIR (Axx). Fig. 6.8 depicts the tree and its associated BD routine for scanning eight sensors. Except for some minor differences, in the output structure, this routine is similar for all proposed schemes. All schemes use four IRIs, each of which scans four lines of 16 sensors each. Each line is scanned in two stages. It takes only one clock cycle to determine that no transition has yet occurred, and two clock cycles to detect a transition and issue the necessary output. If the LIR indicates a previous transition, no output is generated. As a result all IRI outputs contain only those sensors that have undergone transition since the last output.

In addition to the scanning routine, additional BD programs are required for I/O bank switching, communication with the microprocessor and special hardware control.



Input Bank Assignments

8 LIR Bits (Axx) 8 Sensors (Bxx)

Main Scan Subroutine:

```
          ORG:100
BEGIN     IN1:AB6       A7
          IN0:AB6       B7
          OPL           $080
AB6       IN1:AB5       A6
          IN0:AB5       B6
          OPL           $040
AB5       IN1:AB4       A5
          IN0:AB4       B5
          OPL           $020
AB4       IN1:AB3       A4
          IN0:AB3       B4
          OPL           $010
AB3       IN1:AB2       A3
          IN0:AB2       B3
          OPL           $008
AB2       IN1:AB1       A2
          IN0:AB1       B2
          OPL           $004
AB1       IN1:AB0       A1
          IN0:AB0       B1
          OPL           $002
AB0       IN1:BS        A0
          IN0:BS        B0
          OPL           $001
          BRA:BS
           .
           .
           .
           .
BS        <BANK SWITCH ROUTINE>
```

Fig. 6.8: The Scanning BD Tree.

6.3.5 The Line by Line Scanner Interface

This version, Fig. 6.9, is designed to overcome the delay problem of the single transition algorithm used in the Single-Sensor Interface by reporting up to 16 transitions at one time. The interface structure eliminates all of the 4 to 16 demultiplexors and some of the input banks, necessary in the first version, by using tristate buffers at the LIRs' inputs instead of the demultiplexors.

As in the previous version, each line has a dedicated LIR, but these LIRs are buffered and cannot be accessed during their inactive state. The currently scanned line is the only active line, i.e., the LIR buffers and the input tristate buffers are activated by the line bits generated by the line address using a 2 to 4 demultiplexor. Only two output and two input banks are required for the four sensor lines. An additional input and output bank is required, as before, for the current line status and microcomputer communications.

All 16 sensors are scanned each time. An output is generated and the LIRs are updated if at least one new transition was detected. Up to 16 bits are reported via two sides of a PIA to the micro. The digitized reference value, four bits indicating IRI and line number and the number of transitions is transmitted via a second PIA.

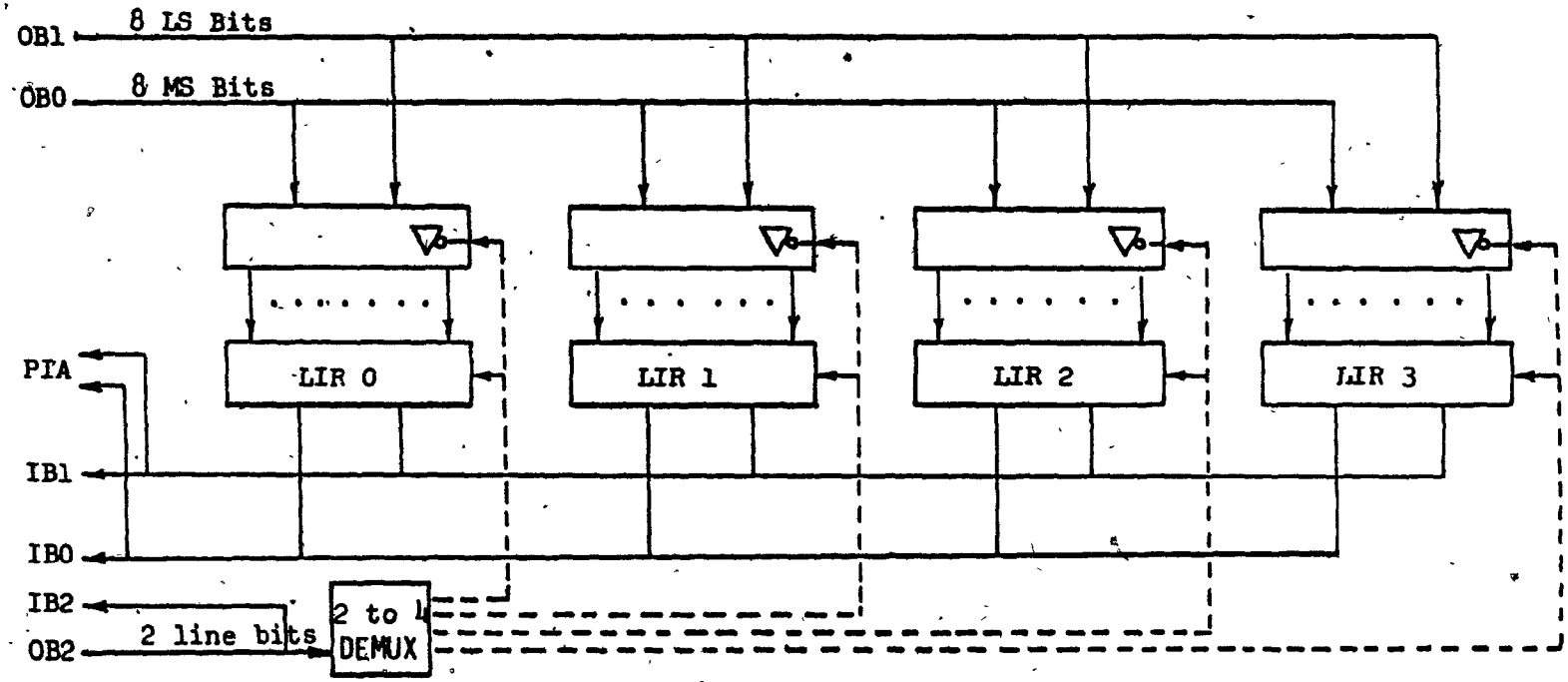


Fig. 6.9: The Line by Line Scanner Interface.

In the single sensor version reference values were stored against sensor addresses at transition time, thus matrix mapping was a straightforward task. In this version a whole line image is stored in the mP memory against the reference value where each stored image only differs from the previous stored image in those bits which have changed. This method therefore requires some postprocessing after the whole matrix has been scanned. Using standard 1 MHz hardware a 16x16 matrix can be resolved in less than 20 msec. Even though this is slower than the single transition method, the probability of a large delay between transition occurrence and detection has been considerably reduced. The hardware required for this scheme is three input banks, three output banks, four LIRs and two PIAs per IRI.

6.3.6 Scanner Interface With Memory

The limiting factor in both aforementioned methods is the microcomputer. The detect-interrupt-load-store sequence, requires a number of clock cycles. Using faster mPs (e.g., 2 MHz) will improve the situation slightly. On the other hand BD based IRI hardware can easily operate at 10 or 20 MHz. In those applications that are time critical, e.g., image conversion, the following scheme, Fig. 6.10, is proposed.

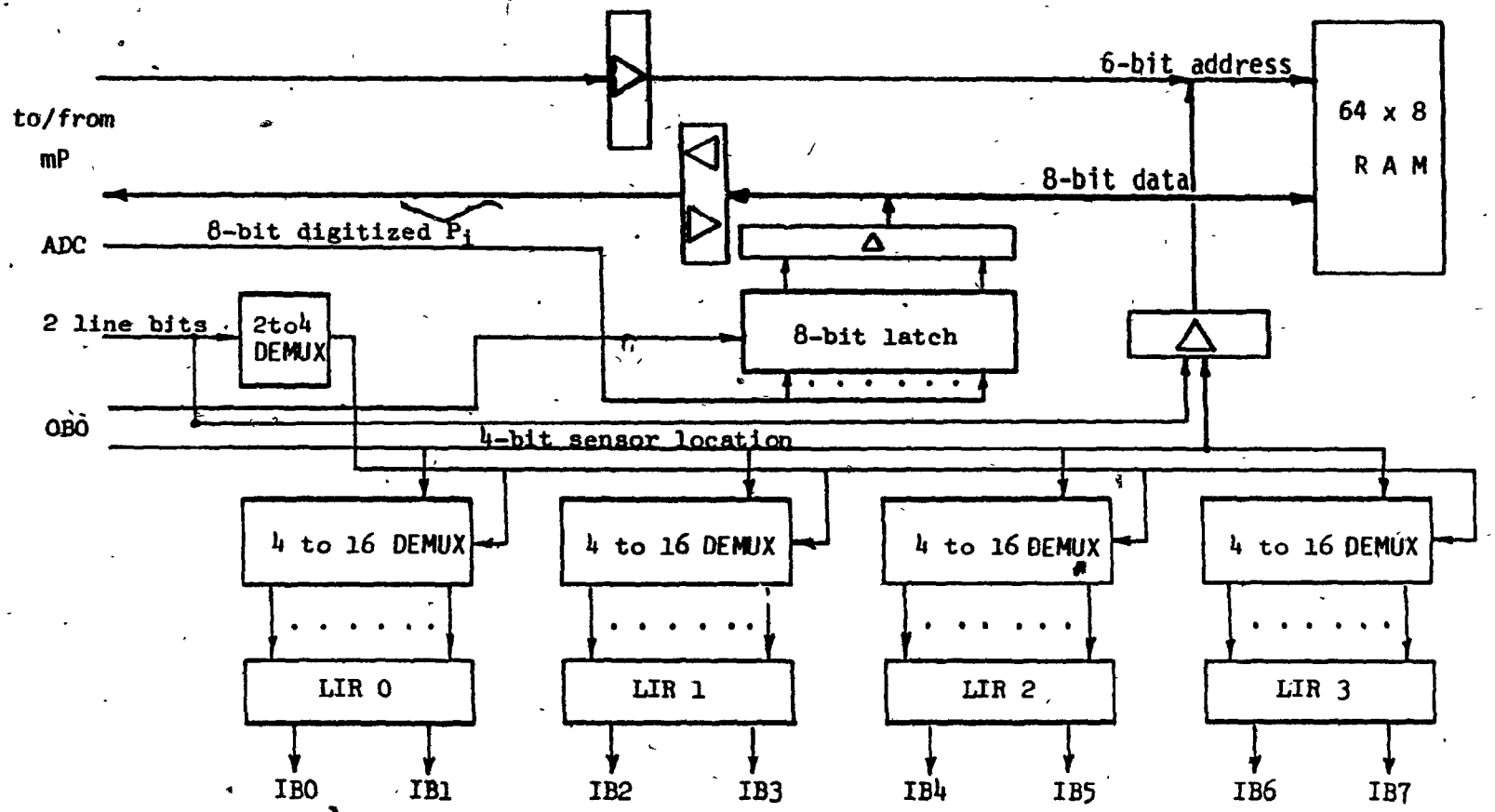


Fig. 6.10: Scanner Interface With Memory.

The interface structure is identical to the Single-Sensor interface except that an additional 64x8 RAM memory is installed in the interface and is connected to the reference value latch. The 6-bit sensor address, 4-bits to locate the sensor in a line, 2-bits to determine which line, generated in a way similar to that used in the first Single-Sensor scheme, is used here to store the current digitized reference in this RAM. The RAM is accessible by the mP through a DMA channel, and it can be downloaded as soon as a counter indicates that all 256 sensors have undergone transition. In addition, this scheme permits an increase in resolution simply by using a wider ADC to measure the reference and by using wider RAM memory to store the values.

As no interaction is required between the IRIs and the micro during the scanning phase, all 256 analog values can be stored, using a 1 MHz clock, in less than 1 msec.

6.3.7 Scanner Application: Contact Pressure Measurement

A practical situation to which an IRI based scanner may be applied is the mapping of contact pressure on the human body while seated or recumbent [VE83]. Consider the known medical problem of contact pressure exerted, for extended period of time by, for example, a wheelchair seat upon an immobilized patient. The mapping of the pressure over the contact surface can help in properly designing the wheel-chair seat and adapting it to specific patients. Existing mapping methods employ a few surface transducers and make measurements at these few points.

The IRI scanner methods described above appear to be suited to this application. Speed is the main benefit of scanning this matrix with BD intelligence. Snapshot pressure maps can be succesfully obtained notwithstanding patient fidgeting. Applying the IRI version with Memory, for example, requires minimal mP intervention during the acquisition of data, small memory requirements for both mP programs and storage and thus a minimal system can be configured. This is of particular importance for a selfcontained clinical version of the pressure scanner.

The IRI based system compares very favourably with Vega's conventional 8-bit microcomputer scheme for similar data acquisition system [VE83]. Vega mapped the 16 x 16 matrix by repeatedly storing binary matrix images, and the related pressure readings. Vega's method needed large data memory, and took considerable time to measure (up to 400 msec.). Due to the signal preprocessing by the BD processor and filtering by the LIRs, an IRI scanner requires minimal memory storage and only several milliseconds to accomplish the matrix mapping task.

Larger sensor matrices can be simply served with additional IRI modules. The proposed scanner system is not limited to pressure mapping. It can be used for example, to process images or to map temperature distribution.

CHAPTER 7: CONCLUSIONS

7.1 Objective of the Thesis

The work presented in this thesis exploits the advantages of BD automata to design a low-cost, fast, programmable I/O interface module; the Intelligent Reflexive Interface. The objective of the research was to design a fast, powerful and inexpensive family of modular programmable controllers that incorporate one or more IRI modules and a higher level microprocessor. The following goals were set:

1. To study the feasibility of using BD automata in conjunction with a microprocessor and identify the advantages, in control applications, offered by this configuration, compared to Boolean based microprocessors.
2. To interface the prototype BD machine built by Holcl in DATAC to an 8-bit MC6809 mP enabling the implementation of Hudson's BD operating system and compiler and to test the operation of the two processors.
3. To design a novel modular intelligent I/O interface module - the IRI.
4. To apply IRI based controllers to a pair of widely different types of application.

7.1.1 BD Automata Feasibility

It was shown that BD methods have inherent advantages over their Boolean counterparts in terms of program execution speed, computation throughput and simplicity of implementation. Several BD advantages were demonstrated:

1. Speed of execution has linear upper bound as compared to the Boolean exponential bound.
2. Multiple functions of the same variables can be evaluated simultanously.
3. Multi-valued functions can be generated.

It was also shown that any logic function, combinatorial or sequential, can be implemented in a BD program.

The core of the BD automaton was shown to have an extremely simple structure; 5 logic gates. This implies minimal signal propagation delays as well as low cost implementation.

Because the size of BD programs tends to grow exponentially, if no minimization techniques are applied, a stand-alone BD automaton was found to be particularly suited to applications involving many independent processes, each of which has a limited number of input bits.

The BD automaton in conjunction with a high level microprocessor can overcome program size limitations by having the microprocessor dynamically download new programs as required by process conditions.

7.1.2 Interfacing BD Automata with an 8-bit mP

The feasibility of a high-level microprocessor working in conjunction with a BD automaton was demonstrated with the hardware interface and the development of operating software. It was shown that the two processors complement each other and can operate independently as well as interrupt each other when required. The design of the multi-processing scheme was succesfully applied. The hardware work was complemented by the implementation of the BD compiler and the BD09 operating system. The hybrid scheme is fully functional and is extensively used in the DATAC Lab. Further development of the optimizing compiler, an application to the control of an inverted pendulum and other control projects are in progress.

7.1.3 IRI Design

BD automata can be best exploited if control tasks are partitioned based upon their nature and complexity. The task partitioning concept is employed in the design of the modular Intelligent Reflexive Interfaces.

In designing the IRI module and the overall IRI based system architecture the main objective was to obtain highest control throughput with a minimized hardware configuration. It was shown that the IRI module acting as a reflexive I/O processor, a programmed BD machine, responds quickly and independently in process control environment.

The BD automaton employed in the hybrid scheme was upgraded and redesigned to enable direct interfacing to the main processor bus. The IRI module does not need a separate parallel interface. Four of IRI module versions were designed: Dedicated, DMA, Shared/Dedicated and Shared. All versions employ the enhanced BD processor as the IRI processor. These different hardware configurations demonstrate design flexibility in adapting to different applications. As an example, the Shared version, with a single IRI module, is suitable to specialized control applications which are limited in parallel processing requirements but do require very short response time.

7.1.4 Applying IRIs

Control task partitioning is the major concept behind the IRI based Programmable Controllers proposed in this thesis. Their objective is twofold:

1. Maximizing controller throughput with minimized hardware configuration.
2. Achieving a very fast real-time response.

The concept of control tasks partitioning was demonstrated in a batch weighing application to show how simple IRI modules can enhance system's performance by autonomously handling routine but time consuming monitoring and control tasks.

IRIs implemented in an industrial PLC can reduce system scan time, therefore enabling more input/output variables to be accomodated. Processes that require multiple, cpu based, PLCs for parallel processing to reduce system's scan time, can enjoy the benefit of a low-cost multiple processing using several IRI modules.

The industrial weighing application demonstrated this distributed control concept by having each IRI control a different interrelated segment of the process. In addition to IRI supervision and co-ordination, higher level control tasks such as control algorithms and set-points optimization, statistics gathering and data management, are performed by the master processor.

The second application demonstrated the benefits of using IRI binary multiplexors in simplifying sensor cabling, reducing data storage requirement and increasing frequency response. The IRI scanner compares very favourably with a conventional microprocessor based scanner, both speedwise and data storage requirments. Larger sensor matrices can be simply served with additional IRI modules. The proposed scanner system is not limited to pressure mapping. It can be used for example, to process images or to map temperature distribution. Future work will extend Vega's analysis of sensor dynamics, evolve improved sampling algorithms to adjust reference signal limits so as to increase accuracy by reducing scanning delays.

7.2 Future Work

Future work will take two directions:

1. Enhancement of BD architecture and the IRI module.
2. Applications development using the IRI concept.

The concepts presented in this thesis can be further developed to meet commercial requirements. Industrial application of BD processor is still in gestation. Further development can overcome remaining limitations. One area that can be attacked is the addressing capability of the BD processor. Shared Memory structure enables the sharing of larger memory with ease. This requires addresses longer than 8-bits which can be implemented by either changing the BD instruction length or with special, multi-word addressing instructions.

Implementation of the BD processor in a single VLSI microchip is another challenging task which was initiated by Vega and Li [VEB2]. Availability of such a functional unit will enable more compact hardware packaging of IRI based controllers.

IRI based controllers may be applied in numerous fields. E.g., distributed control and hierarchical multi-level control. Robotic systems may also be candidates. Each axis of motion provides a natural site for control by an IRI module, under microprocessor supervision. Additional BD processors can be used in hierarchical fashion to analyze and control displacement, velocity and acceleration.

LIST OF REFERENCES

[AK78] S.B. Akers, "Binary Decision Diagrams", IEEE Transactions on Computers, Vol. C-27, no. 6, 1978, pp. 509-516.

[AW80] B.A. Artwick, "Microcomputer Interfacing", Prentice Hall, Englewood Cliffs, NJ, 1980.

[BA59] C. Backus, "The Syntax and Semantics of The Proposed Algebraic Language", Proc. UNESCO Conf. Information Processing, Paris, June 1959, pp. 125-132.

[BN71] C,G. Bell, A. Newell, "Computer Structures: Readings and Examples", McGraw Hill, New-York, NY, 1971.

[BO76] R.T. Boute, "The Binary Decision Machine as Programmable Controller", Euromicro Newsletter, Vol. 1, #2, 1976, pp. 16-22.

[CE79] E. Cerny, D. Mange, E. Sanchez, "Synthesis of Minimal Binary Decision Trees", IEEE Transactions on Computers, Vol C-28, No. 7, July 1979, pp. 472-482.

[CH82] P.M. Chirlian, "Digital Circuits with Microprocessor Applications", Matrix Publishers, Beaverton, OR., 1982

[DA83] M. Davio, J.P. Deschamps, A. Thayse, "Digital Systems with Algorithm Implementation", John Wiley & Sons, N.Y., 1983.

[GE69] C.W. Gear, "Computer Organization and Programming", McGraw Hill, New York, 1969, pp. 362-368.

[HE73] H. Hellerman, "Digital Computer System Principles", McGraw Hill, New York, 1973, pp. 112-115.

[HU81a] R. Hudson, L.J. Vrocmen, P.J. Zsombor-Murray and T. Le-Ngoc, "A Hybrid Binary-Decision/Microprocessor Programmable Controller", Proc. of Fourth IASTED Int'l Symp., Cairo, September 1981, pp. 109-114.

[HU81b] R. Hudson, L.J. Vroomen, P.J. Zsombor-Murray "Industrial Applications of a A Hybrid Binary-Decision/-Microprocessor Programmable Controller", Proc. IEC & C Conf., Toronto, Oct. 1981, pp. 116-117.

[HU82] R. Hudson, P.J. Zsombor-Murray and L.J. Vroomen, "Operating System for Hybrid BD/mP Programmable Controller", Proc. of Twentieth ISMM Int'l Symp. on Mini and Microcomputers, Cambridge, MA., July 1982, pp. 1-4.

[HU84] R. Hudson, L.J. Vroomen, and M. Karasick, "Binary Decision Program Optimization Algorithms", Proc. of Twenty-Forth Int'l Symp. on Mini and Microcomputers, Bari, Italy, June 1984, pp. 51-55.

[IA81] "Computer Controlled Weighing and Batching Installations", Industrie Automation, Wage und Prozesstechnik GMBH, Heidelberg, W. Germany, 1981

[KA84] M. Karasick, "Binary Decision Tree Reduction", Technical Report SOCS-84.16, McGill University, School of Computer Science, October 1984.

[KH70] Z. Kohavi, "Switching and Finite Automata Theory", McGraw-Hill, New-York, 1970

[KU82] A.C. Kucuk, P.J. Zsombor-Murray, and L.J. Vroomen, "Process Control Using a Binary Decision/Microprocessor Hybrid", Proceedings of Twentieth ISMM Int'l Symposium on Mini and Microcomputers, Cambridge, MASS., July 1982.

[KU83] A.C. Kucuk, P.J. Zsombor-Murray and L.J. Vroomen, "A Binary-Decision Controlled Robotic Balancer", Proc. CAN/CAM Conference Applied Mechanics, Saskatoon, SK, May 1983, pp. 855-856.

[LE59] C.Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs", Bell Systems Tech. J., July 1959, pp. 985-999.

[LM82a] M.H. Levi and R. Hudson, "Binary Decision-Microprocessor Interface", Memo 82-3, DATAC Computer Laboratory, Montreal, April 1982.

[LM82b] M.H. Levi, L.J. Vroomen and P.J. Zsombor-Murray, "Modular Programmable Controllers using an Intelligent Reflexive Interface" Proc. of Twentieth ISMM Int'l Symp. on Mini and Microcomputers, Cambridge, MA., July 1982, pp. 8-11.

[LM83] M.H. Levi, L.J. Vroomen and P.J. Zsombor-Murray, "Modular Programmable Controllers using an Intelligent Reflexive Interface", Int'l Journal of Mini and Microcomputers, Vol. 5, #1, Dec. 1983, pp. 1-6.

[LN79] T. Le-Ngoc, P.J. Zsombor-Murray and L.J. Vroomen, "A Binary-Decision Approach to Industrial Programmable Controllers", Proc. of the Int'l Symp. Measurement and Control, Grenoble, June 1979, pp. 57-61.

[MD80] "The 584 User Manual", Gould Inc. Modicon Div., 1982.

[MD82] R.A Whitehouse, "184/384 MODICON PC Manual", Gould Inc. Modicon Div., 1982.

[MO82] B.M.E. Moret, "Decision Trees and Diagrams", ACM Computing Surveys, Vol. 14, No. 4, December 1982, pp. 593-623.

[PA82] T. Pavlidis, "Algorithms for Graphics and Image Processing", Computer Science Press, Rockville, MD, 1982.

[PE72] J.B. Peatman, "The Design of Digital Systems", McGraw-Hill Co., New-York, NY, 1972.

[SH38] C.E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits", Trans. AIEE, Vol. 57, 1938, pp. 713-723.

[TA81] R.L. Tabachnick, P.J. Zsombor-Murray, L.J. Vroomen and T. Le-Ngoc, "Sequence Controllers with Standard Hardware and Custom Firmware", IEEE MICRO, May 1981, vol. 1, #2, pp. 9-25.

[VA82] C.J. Van Driel, "Binary Decision Compiler Algorithms", Mechanical Engineering Research Laboratories Memo 82-2, McGill University, Montreal, August 1982.

[VE82] M. Vega and T.F. Li, "Design of a VLSI Circuit for the Processor of a Binary-Decision Machine", Internal Report, DATAC Computer Laboratory, Montreal, May 1982.

[VE83] M.P. Vega, "Contact Pressure Measurements With Pressurized Force Switches", M. Eng. Thesis, McGill University, Montreal, October 1983.

[ZM79] P.J. Zsombor-Murray, L.J. Vroomen, T. Le-Ngoc and P. Holck, "A Binary Decision Based Programmable Controller", Proc. of the Second Int'l Symp. on Mini and Microcomputers, Fort Lauderdale, December 1979, Vol. 5, #2, pp. 60-65.

[ZM83a] P.J. Zsombor-Murray, L.J. Vroomen, R. Hudson, T. Le-Ngoc and P. Holck, "Binary Decision Based Programmable Controllers - Part 1", IEEE Micro, Vol. 3, #4, August 1983, pp. 67-83.

[ZM83b] P.J. Zsombor-Murray, L.J. Vroomen, R. Hudson, T. Le-Ngoc and P. Holck, "Binary Decision Based Programmable Controllers - Part 2", IEEE Micro, Vol. 3, #5, October 1983, pp. 16-26.

[ZM83c] P.J. Zsombor-Murray, L.J. Vroomen, R. Hudson, T. Le-Ngoc and P. Holck, "Binary Decision Based Programmable Controllers - Part 3", IEEE Micro, Vol. 3, #6, December 1983, pp. 24-39.

APPENDIX A: THE MC6809 BASED DEVELOPMENT SYSTEM

The development system used in this research project consists of a MC6809 based SWTPC* microcomputer and the FLEX09** disk operating system. A brief description of both the hardware and the operating system is given in this appendix.

System's Hardware

The SWTPC microcomputer is shown in Fig. A.1. It consists of:

1. Main system containing a 2 MHz MC6809 CPU, 56K of RAM, 6K of EPROM and parallel and serial interfaces,
2. Dual floppy disk drive,
3. Operator console,
4. Printer.

The interface hardware, designed in this project, was implemented using wire-wrap type prototype cards.

The system's mother board contains the system busses to connect the different modules into one operational unit. The SS-50 main system's bus has 7 slots, the secondary SS-30 I/O bus - 8 slots. The CPU and RAM cards reside in the SS-50 bus. Peripheral interfaces are connected to the SS-30 bus. Developed prototype cards may be connected to either bus. The BD parallel interface card was designed and implemented using

* SWTPC is a Trademark of Southwest Technical Prods. Corp.

** FLEX is a Trademark of Technical Systems Consultants Inc.

Fig. A.1: The SWPTC Microcomputer Development System.

the SS-30 bus. Both busses have gained world recognition and are employed by several other microcomputers [AW80].

A detailed description of the SS-50 and the SS-30 busses is given below. Each of the busses consists of 3 sub-busses:

1. the data bus,
2. the address bus, and
3. the control bus.

In addition power and ground lines are provided.

The pinout of SS-50 bus is given in Table A-1. The D0 - D7 lines form the data bus which carries 8 inverted data bits.

The A0 - A15 lines form the address bus capable of addressing 64K memory locations. The 4 User-Defined lines may be employed to extend the address bus to 20-bit addresses (1 MB of addressed memory).

Power lines to all the system's modules are 7-8V unregulated, used to generate standard TTL voltage level of 5V, +/-16V regulated to +/-12V by serial interfaces and dynamic memories, and common ground lines to be shared by all modules.

BUS PINOUT	SIGNAL	FUNCTION
1 - 8	$\overline{D0}$-$\overline{D7}$	Data Bus (inverted)
9 - 24	A15-A0	Address Bus
25,26,27	GND	Common Ground
28,29,30	7-8V UNR	Power (regulated to 5V on board)
31,32	+/-16V UNR	Power (regulated to +/-12V on board)
33	INDEX	Module Orientation
34	MR	Memory Ready (strech E - slow dev.)
35	$\overline{BU}$	Busy (R-Mod-W cycle)
36	$\overline{IRQ}$	Interrupt
37	$\overline{FIRQ}$	Fast Interrupt
38	Q	Phase 1: H to L = valid addresses
39	E	Enable (phase 2) data valid HL - W LH - R
40	$\overline{VMA}$	Valid Memory Address
41	R/$\overline{W}$	Read/Write
42	$\overline{RST}$	Reset
43	BA	Bus Available
44	BS	Bus status
45	$\overline{HALT}$	Halt processor, external bus control
46	$\overline{BREQ}$	Bus request
47 - 50	USER DEFINED	

Table A-1: The SS-50 Bus.

The control bus consists of several control lines enabling communication protocols and bus arbitration. These are:

1. $\overline{MR}$ - Memory Ready, streches the E phase of the clock up to 10 microsec to allow interfacing to slow devices.
2. $\overline{BU}$ - Busy, asserted by the mP during Read/Modify/Write cycle to deny external access to the bus.

3. $\overline{IRQ}$, $\overline{FIRQ}$ - interrupt and fast interrupt.
4. Q - phase 1 of the clock, leads E by 90 degrees. High to Low transition indicates a valid address on the bus.
5. $\overline{E}$ - phase 2 of the clock, used to validate data - on falling edge when write and on rising edge when read.
6. $\overline{VMA}$ - valid memory address.
7. R/$\overline{W}$ - Read/$\overline{Write}$, defines the direction of data between memory and mP or interface.
8. BA,BS - Bus Available and Bus Status, indicate normal bus operation (0,0), interrupt acknowledge (0,1), sync acknowledge (1,0), and halt acknowledge (1,1).
9. $\overline{HALT}$ - Used by bus devices other then the mP to halt the mP and gain control over the bus.
10. $\overline{BR}$ - Bus Request, Similar to Halt but for short term bus control.

The SS-30 bus is a subset of the SS-50. Its pinout is given in Table A-2. It provides 4 LS address lines only. In addition, each bus slot contains a decoded $\overline{I/O\ SEL}$ signal which is equivalent to the MS 12 address lines. The bus is used for interfacing with system's peripherals that do not require more then 16 dedicated addresses for their operations. Other bus signals are similar to the SS-50 ones.

BUS PINOUT	SIGNAL	FUNCTION
10,11,1,2	RS0-RS3	Register Select lines (A0-A3)
3,4	+/-16V UNR	Power
5,6	GND	Common Ground
7	INDEX	Module Orientation
8	$\overline{IRQ}$	Interrupt
9	$\overline{FIRQ}$	Fast Interrupt
12 - 19	D0-D7	Data Bus (non-inverted)
20	E	Enable (phase 2) data valid HL - W LH - R
21	R/$\overline{W}$	Read/Write
22,23	7-8V UNR	Power (regulated to 5V)
24 - 28	110-9600	Baud Rate
29	$\overline{RST}$	Reset
30	$\overline{I/O\ SEL}$	I/O Select

Table A-2: The SS-30 Bus.

System's Software

A brief descciption of the FLEX09 disk operating system and its accompanying software development tools is given below.

The FLEX09 consists of three main parts:

1. File Management System (FMS),
2. Disk Drivers, and
3. Utilities.

FMS handles file storage and removal from disk as well as disk space allocation. It is the link between the microcomputer and the disk drivers which control the actual disk hardware. FMS communicates via File Control Blocks (FCB). The FCB contains file information, e.g., file name, disk drive, etc. An FCB is assigned to a file whenever it is opened by the operating system for reading or writing. FLEX09 allows for multiple files to be opened simultaneously.

The disk drivers are interface routines between FLEX09 FMS and the disk hardware to control the floppy disks. They are being called by FLEX09 whenever a disk access is required. They can be called by user written programs. E.g., the BD09 operating system calls these drivers when retrieving or saving BD programs on disk. The drivers' calling program uses the MC6809 A and B accumulators and the X index register to pass information such as FCB address, address on the floppy disk (indicated by track and sector numbers). The main disk drivers are:

1. READ - reads a single sector
2. WRITE - writes a single sector
3. VERIFY - verifies that a sector just written has no CRC error
4. SELECT - selects the drive to be accessed
5. CHECK - checks if drive is ready

The FLEX09 utility set allows the most commonly used disk operations to be executed in a single command. When a command is called, FLEX09 checks the disk directory, loads the corresponding program into RAM memory and passes control to the program. Upon termination of program execution, control is returned to FLEX09. Commonly used utilities are:

1. DIR - lists the directory of all files on disk
2. COPY - copys a file from one disk to another
3. SAVE - saves content of memory on disk
4. DELETE - deletes a file from disk
5. EDIT - edits a disk file

Other development tools are the Assembler and the Debugger. The FLEX09 6809 Mnemonic Assembler is directly interfaced to the FLEX09 disk operating system. It accepts Motorola 6809 mnemonics. The source language code is stored as a text file on disk. The Assembler generates object code from the source program in 2 passes. The object code is saved on disk separately and can be loaded into memory and executed.

FLEX09 offers an excellent debugging facility called DEBUG. DEBUG enables the execution of a program in a stepwise manner to detect errors. All CPU registers are displayed at each step allowing the user to monitor and verify the correct register contents.

In addition to the above operating system, the microcomputer employs an EPROM resident monitor called V-BUG. V-BUG contains a boot-strap routine to initiate FLEX09 from disk upon power-up and it enables low level control of the system's hardware such as:

1. View/Modify RAM contents,
2. View/Modify CPU registers,
3. Loading and Executing actual object code, and
4. Some debugging tools.

All the above tools were extensively used in all phases of the BD hardware and software development.

APPENDIX B: APPLICATION SOFTWARE LISTINGS

This appendix includes listings of several IRI control routines for the batch weighing, section B.1, and for a single-sensor version of the variable reference comparator matrix scanner, section B.2.

The following batch weighing routines are included:

1. Monitoring process parameters: Main line air-pressure.
2. Monitoring min/max level indicators of 4 ingredient bins.
3. Weight cycle control: monitoring material weighing in
   Coarse and Fine modes,
   Scale feeding and discharge.

The following IRI scanning routines are included:

1. IRI scanning of 4-line section of a 16x16 sensor matrix,
2. Bank switching routine to be used by the scanning algorithm.

Input and output variables are assigned at the beginning of the routines.

```
'*****************************************************************
'                                                                 *
'  IRI #3: Process Monitoring and Control of a Weighing Cycle     *
'                                                                 *
'*****************************************************************
'
' Input Variables assignments:
' ----------------------------
'
'  Input Bank #0:
'  -------------
'  internal testing inputs (line 0 to line 3)
'
'  general process parameters (line 4 to line 16)
'
AIRP  EQU IN<4>   ' main air pressure line indicator
'
SSV   EQU IN<5>   ' Stand Still Validation from mP via PIA
'
CORS  EQU IN<9>   ' internal switch - set by the IRI output bank 003, used
'                   for determining the return address at the end of the
'                   comparison routine when used in coarse weighing mode.
FIN   EQU IN<10>  ' same as CORS - when used in fine weighing mode
'
CCOPV EQU IN<14>  ' coarse cut-off-point is valid (from mP via PIA)
FCOPV EQU IN<15>  ' fine cut-off-point is valid (from mP via PIA)
'
'
' Input Bank #1
' -------------
'               8 LS bits = 8-bit input of weighscale load cell A/D
LC1 EQU IN<0>
LC2 EQU IN<1>
LC3 EQU IN<2>
LC4 EQU IN<3>
LC5 EQU IN<4>
LC6 EQU IN<5>
LC7 EQU IN<6>
LC8 EQU IN<7>
'
'              8 MS bits = 8-bit cut off point value from PIA
COP1 EQU IN<8>
COP2 EQU IN<9>
COP3 EQU IN<10>
COP4 EQU IN<11>
COP5 EQU IN<12>
COP6 EQU IN<13>
COP7 EQU IN<14>
COP8 EQU IN<15>
'
' Input Bank #2: feeder positions (1-on, 0-off)
' ---------------------------------------------
FDC21 EQU IN<0>  '  bin 21 feeder coarse actual position
FDF21 EQU IN<1>  '  bin 21 feeder fine   actual position
FDC22 EQU IN<2>  '  bin 22 feeder coarse actual position
```

```
FDF22 EQU IN<3>   !  bin 22 feeder fine   actual position
FDC23 EQU IN<4>   !  bin 23 feeder coarse actual position
FDF23 EQU IN<5>   !  bin 23 feeder fine   actual position
FDC24 EQU IN<6>   !  bin 24 feeder coarse actual position
FDF24 EQU IN<7>   !  bin 24 feeder fine   actual position
FIC21 EQU IN<8>   !  bin 21 feeder coarse image position
FIF21 EQU IN<9>   !  bin 21 feeder fine   image position
FIC22 EQU IN<10>  !  bin 22 feeder coarse image position
FIF22 EQU IN<11>  !  bin 22 feeder fine   image position
FIC23 EQU IN<12>  !  bin 23 feeder coarse image position
FIF23 EQU IN<13>  !  bin 23 feeder fine   image position
FIC24 EQU IN<14>  !  bin 24 feeder coarse image position
FIF24 EQU IN<15>  !  bin 24 feeder fine   image position
!
! Input Bank #3: level indicators (1-on, 0-off)
! ---------------------------------------------
MIN21 EQU IN<0>   !  bin 21 minimum level indicator
MAX21 EQU IN<1>   !  bin 21 maximum level indicator
MIN22 EQU IN<2>   !  bin 22 minimum level indicator
MAX22 EQU IN<3>   !  bin 22 maximum level indicator
MIN23 EQU IN<4>   !  bin 23 minimum level indicator
MAX23 EQU IN<5>   !  bin 23 maximum level indicator
MIN24 EQU IN<6>   !  bin 24 minimum level indicator
MAX24 EQU IN<7>   !  bin 24 maximum level indicator
WDCA1 EQU IN<8>   !  Weigh-Scale #1 discharge gate actual position
!                                                  (1-open, 0-closed)
MIXA1 EQU IN<8>   !  Mixer actual state (1-on, 0-off)
MIXDC EQU IN<8>   !  Mixer discharge gate actual position (1-open, 0-closed)
! 12- not used
WDCI1 EQU IN<8>   !  Weigh-Scale #1 discharge gate image position
!                                                  (1-open, 0-closed)
MIXI1 EQU IN<8>   !  Mixer image state (1-on, 0-off)
WDCI1 EQU IN<8>   !  Mixer discharge gate image position (1-open, 0-closed)
! 16- not used
!
! Output Variables assignments:
! -----------------------------
!
! Output Bank #1:
! ---------------
! all feeder actuators are tied to their corresponding image input lines
! OU<0>  bin 21 feeder coarse actuator (1-on, 0-off) - FDC21
! OU<1>  bin 21 feeder fine   actuator (1-on, 0-off) - FDF21
! OU<2>  bin 22 feeder coarse actuator (1-on, 0-off) - FDC22
! OU<3>  bin 22 feeder fine   actuator (1-on, 0-off) - FDF22
! OU<4>  bin 23 feeder coarse actuator (1-on, 0-off) - FDC23
! OU<5>  bin 23 feeder fine   actuator (1-on, 0-off) - FDF23
! OU<6>  bin 24 feeder coarse actuator (1-on, 0-off) - FDC24
! OU<7>  bin 24 feeder fine   actuator (1-on, 0-off) - FDF24
! OU<8>  bin 21 minimum level indicator warning (1-on, 0-off) - MLIZ21
! OU<9>  bin 21 maximum level indicator warning (1-on, 0-off) - MLIZ21
! OU<10> bin 22 minimum level indicator warning (1-on, 0-off) - MLIZ22
! OU<11> bin 22 maximum level indicator warning (1-on, 0-off) - XLIZ22
!
```

```
! Output Bank #2
! --------------
! OU<0>  bin 21 coarse feeder alarm condition (1-on, 0-off) - FDCZ21
! OU<1>  bin 21 fine   feeder alarm condition (1-on, 0-off) - FDFZ21
! OU<2>  bin 22 coarse feeder alarm condition (1-on, 0-off) - FDCZ22
! OU<3>  bin 22 fine   feeder alarm condition (1-on, 0-off) - FDFZ22
! OU<4>  bin 23 coarse feeder alarm condition (1-on, 0-off) - FDCZ23
! OU<5>  bin 23 fine   feeder alarm condition (1-on, 0-off) - FDFZ23
! OU<6>  bin 24 coarse feeder alarm condition (1-on, 0-off) - FDCZ24
! OU<7>  bin 24 fine   feeder alarm condition (1-on, 0-off) - FDFZ24
! OU<8>  bin 23 minimum level indicator warning (1-on, 0-off) - MLIZ23
! OU<9>  bin 23 maximum level indicator warning (1-on, 0-off) - XLIZ23
! OU<10> bin 24 minimum level indicator warning (1-on, 0-off) - MLIZ24
! OU<11> bin 24 maximum level indicator warning (1-on, 0-off) - XLIZ24
!
! Output Bank #3
! --------------
! OU<1>   weigh scale discharge gate actuator (1-on, 0-off) - WDCA1
! OU<2>   mixer discharge gate actuator (1-on, 0-off) - MXDA1
! OU<3>   weigh scale discharge gate alarm condition (1-on, 0-off) - WDCZ1
! OU<4>   mixer discharge gate alarm condition (1-on, 0-off) - MIXDZ
! OU<5> -
! OU<6> -
! OU<7> -
! OU<8> - connected to input bank 0, bit <9>  (see bank 0 assignments)
! OU<9> - connected to input bank 0, bit <10>  (see bank 0 assignments)
! OU<10> - connected to mP PIA to indicate coarse weighing is completed
! OU<11> - connected to mP PIA to indicate fine weighing is completed
!
!************** end of I/O assignments ******************************
!
!  Monitoring Process Parameters:
!  ------------------------------
!
MONIT1  IBS 000
        OBS 000
        INO AIRP    :LEVEL      !check air pressure main line
        OPL 001                 !pressure is low; annunciate operator
        INT 0001    :END        ! issue interrupt level #1 to mP
!
LEVEL   IBS 001
        OBS 001
        IN1 MIN21   :XL21       ! minimum level indicators must be ON
        OPS 001                 ! annunciate operator and continue
XL21    INO MAX21   :ML22       ! maximum level indicators must be OFF
        OPS 002
ML22    IN1 MIN22   :XL22
        OPS 003
XL22    INO MAX22   :ML23
        OPS 004
ML23    OBS 002                 ! switch output bank
        IN1 MIN23   :XL23
        OPS 001
XL21    INO MAX23   :ML24
        OPS 002
```

```
ML22    INI MIN24    :XL24
        OPS 003
XL22    INO MAX24    :FDPOS
        OPS 004
!
FDPOS   IBS 002                  ! monitor ingredient feeder positions
        OBS 002
        INO FDC21    :FF21
        OPL 001
        BRA

END2    EOP          :BEGINW     ! EOP jump address is set by the Operating
!                                  System to the beginning of next routine

!
!  Weighing Cycle Control:
!  -----------------------
!                            (weighing ingredient from bin #21)
!
BEGINW  IBS 000
CORCOP  INO CCOPV    :CORCOP     ! wait for valid coarse cut-off-point
        OBS 001
        OPL 003                  ! actuate both bin #21 coarse & fine feeders
        OBS 003
        OPS 001                  ! set a return switch; bit <8> of output
!                                  bank 3 is tied to bit <9> of input bank 0
!                                  serving as a "latch" to memorize the return
!                                  address to COARSE
        BRA CMPCOP               ! jump to comparison routine
!
COARSE  OBS 001                  ! coarse c.o.p. has been reached
        OPL 002                  ! close coarse feeder, fine feeder stays open
        IBS 000
FINCOP  INO FCOPV    :FINCOP     ! wait for valid fine cut-off-point
        OBS 003
        OPS 002                  ! set a return switch; bit <9> of output
!                                  bank 3 is tied to bit <10> of input bank 0
!                                  serving as a "latch" to memorize the return
!                                  address to FINE
        BRA CMPCOP               ! jump to comparison routine
FINE    OBS 001                  ! fine c.o.p. has been reached
        OPL 000                  ! close fine feeder
        IBS 000
SSTILL  INO SSV      :SSTILL     ! wait for stand-still validation
        OBS 003
        IBS 000
        OPL 001                  ! open discharge gate
SSTIL2  INO SSV      :SSTIL2     ! wait for stand-still validation
!                                  (mP verifies scale empty and records tare
!                                    weight)
END2    EOP                      ! EOP jump address is set by the Operating
!                                  System to the beginning of next routine
```

```
!  IR1 - 8-bit Comparison Routine
!  ------------------------------
!
!  Compare the current weight with the coarse c.o.p.
!
START1  IBS 001                 ! select I/O banks for comparison routine
        OBS
!                                 start with MS bit
BIT8    IN1 COP8    :LCO8
        IN0 LC8     :BIT7       ! both bits are 0 - proceed to next bit
        BRA         :CMPEND     ! weight is greater than c.o.p. - terminate
!                                 comparison
LCO8    IN0 LC8     :BIT8       ! weight is lesser than c.o.p. - restart from
!                                 this bit
!
BIT7    IN1 COP7    :LCO7
        IN0 LC7     :BIT6       ! both bits are 0 - proceed to next bit
        BRA         :CMPEND     ! weight is greater than c.o.p. - terminate
!                                 comparison
LCO7    IN0 LC7     :BIT7       ! weight is lesser than c.o.p. - restart from
!                                 this bit
!
BIT6    IN1 COP6    :LCO6
        IN0 LC6     :BIT5       ! both bits are 0 - proceed to next bit
        BRA         :CMPEND     ! weight is greater than c.o.p. - terminate
!                                 comparison
LCO6    IN0 LC6     :BIT6       ! weight is lesser than c.o.p. - restart from
!                                 this bit
!
BIT5    IN1 COP5    :LCO5
        IN0 LC5     :BIT4       ! both bits are 0 - proceed to next bit
        BRA         :CMPEND     ! weight is greater than c.o.p. - terminate
!                                 comparison
LCO5    IN0 LC5     :BIT5       ! weight is lesser than c.o.p. - restart from
!                                 this bit
BIT4    IN1 COP4    :LCO4
        IN0 LC4     :BIT3       ! both bits are 0 - proceed to next bit
        BRA         :CMPEND     ! weight is greater than c.o.p. - terminate
!                                 comparison
LCO4    IN0 LC4     :BIT4       ! weight is lesser than c.o.p. - restart from
!                                 this bit
!
BIT3    IN1 COP3    :LCO3
        IN0 LC3     :BIT2       ! both bits are 0 - proceed to next bit
        BRA         :CMPEND     ! weight is greater than c.o.p. - terminate
!                                 comparison
LCO3    IN0 LC3     :BIT3       ! weight is lesser than c.o.p. - restart from
!                                 this bit
!
BIT2    IN1 COP2    :LCO2
        IN0 LC2     :BIT1       ! both bits are 0 - proceed to next bit
        BRA         :CMPEND     ! weight is greater than c.o.p. - terminate
!                                 comparison
LCO2    IN0 LC2     :BIT2       ! weight is lesser than c.o.p. - restart from
!                                 this bit
```

```
BIT1    IN1 COP1    :LC01
        IN0 LC1     :BIT0     ! both bits are 0 - proceed to next bit
        BRA         :CMPEND   ! weight is greater than c.o.p. - terminate
                              comparison
LC01    I           :BIT1     ! weight is lesser than c.o.p. - restart from
                              this bit

BIT0    IN1 COP0    :LC00
        IN0 LC0     :CMPEND   ! both bits are 0 - terminate comparison
        BRA         :CMPEND   ! weight is greater than c.o.p. - terminate
                              comparison
LC00    IN0 LC0     :BIT0     ! weight is lesser than c.o.p. - restart from
                              this bit

CMPEND  IBS 000               ! comparison completed check for return
                              address
        OBS 003
        IN0 COR     :FMOD
        OPS 004               ! inform mP Coarse mode is completed
                              mP send fine c.o.p.
        BRA         :COARSE

FMOD    IN0 FIN     :ENDCMP
        OPS 008               ! inform mP Fine mode is completed
                              mP waits for stand-still, record actual
                              weight and validate stand-still (via PIA to
                              SSV input)

        BRA         :FINE

ENDCMP  INT 004     :EOP      ! ERROR; no return address
        EOP
```

```
'*****************************************************************
'                                                                *
'  IRI #1: Pre-processing data detected from 4 rows of 16 sensor *
'          each in a 16x16 variable reference sensor matrix -    *
'          single sensor version                                 *
'                                                                *
'*****************************************************************
'
' Input Variables assignments:
' ---------------------------
'  Input Banks 0 to 7:
'  -------------------
' Each bank: 8 LS bits = 8 matrix sensors
'            8 MS bits = 8 corresponding LIR bits (see: Fig. 6.8, page 6.24)
'                        Sensors  = Bnn
'                        LIR bits = Ann
'       row #1 - input banks 0 and 1
'       row #2 - input banks 2 and 3
'       row #3 - input banks 4 and 5
'       row #4 - input banks 6 and 7
'
B0 EQU IN<0>
B1 EQU IN<1>
B2 EQU IN<2>
B3 EQU IN<3>
B4 EQU IN<4>
B5 EQU IN<5>
B6 EQU IN<6>
B7 EQU IN<7>
B8 EQU IN<0>
B9 EQU IN<1>
B10 EQU IN<2>
B11 EQU IN<3>
B12 EQU IN<4>
B13 EQU IN<5>
B14 EQU IN<6>
B15 EQU IN<7>
'
A0 EQU IN<8>
A1 EQU IN<9>
A2 EQU IN<10>
A3 EQU IN<11>
A4 EQU IN<12>
A5 EQU IN<13>
A6 EQU IN<14>
A7 EQU IN<15>
A8 EQU IN<8>
A9 EQU IN<9>
A10 EQU IN<10>
A11 EQU IN<11>
A12 EQU IN<12>
A13 EQU IN<13>
A14 EQU IN<14>
A15 EQU IN<15>
```

```
'  Input Bank #9:
'  --------------
LINE0  EQU IN<0>
LINE1  EQU IN<1>
LINE2  EQU IN<2>
LINE3  EQU IN<3>
'
'
LSMSSW EQU IN<9>            '  LS or MS switch from output bank 1
'
'
'
'  Input Bank #10:
'  ---------------
'  internal testing inputs (line 0 to line 3)
'
'  general process parameters (line 4 to line 16)
'
'
'
' Output Variables assignments:
' -----------------------------
'
' Output Bank #0
' --------------
' OU<2:1>                  '  line address
' OU<11:8>                 '  Sensor address in line <2:1>
'
' Output Bank #1
' --------------
' OU<8> 'switch to indicate whether the MS 8 sensors were scanned (on)
'        or the LS 8 sensors were scanned (off)
'
'*************** end of I/O assignments ********************************
'
```

```
' IRI scanner main routine
' ------------------------
'
START    OBS 000
         OPL 000                  ' Select line 0
'
'        start scanning algorithm
'
START1   IBS 9
         OBS 0          :LIN0
'
NXTLN    INT 0002       :EOP1     ' interrupt mP and jump to EOP1
'                                   mP will resume operation after
'                                   the registration of sensor address
'                                   and variable reference value.
'
LIN0     IN0 LINE0      :LIN1
         OPL $000                 ' select line 0
         IBS 0
         BRA STACMP
LIN1     IN0 LINE1      :LIN2
         OPL $001                 ' select line 1
         IBS 2
         BRA STACMP
LIN2     IN0 LINE2      :LIN3
         OPL $002                 ' select line 2
         IBS 4
         BRA STACMP
LIN3     IN0 LINE3      :LIN0
         OPL $003                 ' select line 3
         IBS 6
         BRA STACMP
!
         EOP1
'
'
'
'  Line comparison routine:
'  ------------------------
'
'       when this routine is called, line number is assigned
'       in output bank 0 and first line input bank is selected
'
STACMP   IN1 A15        :AB14     ' test LIR bit, goto next bit if transition
'                                   had already occured
         IN0 B15        :AB14
         OPS $F         :NXTLN    ' bit 15 - report transition detected
AB14     IN1 A14        :AB13
         IN0 B14        :AB13
         OPS $E         :NXTLN    ' bit 14 - report transition detected
AB13     IN1 A13        :AB12
         IN0 B13        :AB12
         OPS $D         :NXTLN    ' bit 13 - report transition detected
AB12     IN1 A12        :AB11
         IN0 B12        :AB11
```

```
        OPS $C          :NXTLN  ' bit 12 - report transition detected
AB11    IN1 A11         :AB10
        IN0 B11         :AB10
        OPS $B          :NXTLN  ' bit 11 - report transition detected
AB10    IN1 A10         :AB9
        IN0 B10         :AB9
        OPS $A          :NXTLN  ' bit 10 - report transition detected
AB9     IN1 A9          :AB8
        IN0 B9          :AB8
        OPS $9          :NXTLN  ' bit  9 - report transition detected
AB8     IN1 A8          :BS1
        IN0 B8          :BS1
        OPS $8          :NXTLN  ' bit  8 - report transition detected
'
BS1     OBS 001
        OPS $01                 ' set switch to indicate MS 8 sensors scanned
        BRA BANKSW              ' branch to bank switch routine
'
LSCMP   IN1 A7          :AB6
        IN0 B7          :AB6
        OPS $7          :NXTLN  ' bit  7 - report transition detected
AB6     IN1 A6          :AB5
        IN0 B6          :AB5
        OPS $6          :NXTLN  ' bit  6 - report transition detected
AB5     IN1 A5          :AB4
        IN0 B5          :AB4
        OPS $5          :NXTLN  ' bit  5 - report transition detected
AB4     IN1 A4          :AB3
        IN0 B4          :AB3
        OPS $4          :NXTLN  ' bit  4 - report transition detected
AB3     IN1 A3          :AB2
        IN0 B3          :AB2
        OPS $3          :NXTLN  ' bit  3 - report transition detected
AB2     IN1 A2          :AB1
        IN0 B2          :AB1
        OPS $2          :NXTLN  ' bit  2 - report transition detected
AB1     IN1 A1          :AB0
        IN0 B1          :AB0
        OPS $1          :NXTLN  ' bit  1 - report transition detected
AB0     IN1 A0          :BS2
        IN0 B0          :BS2
        OPS $0          :NXTLN  ' bit  0 - report transition detected
'
        OBS 1
        OPS 00                  ' turn off the MS/LS switch
'
BS2     BRA             :BANKSW
```

```
'
'  Bank switch routine
'  -------------------
'
BANKSW  IBS 9
        OBS 1
'
LLIN0   IN0 LINE0     :LLIN1     ' is it line 0
        IN0 LSMSSW    :LLLIN1    ' is it MS 8 sensors, if not line 1 is next
        OPS 0                    ' turn-off LM/MS switch
        IBS 1         :LSCMP     ' branch to scan LS 8 sensors of line 0
LLLIN0  OPS 1                    ' turn-on  LS/MS switch
        IBS 0         :STACMP    ' branch to scan MS 8 sensors of line 0
'
LLIN1   IN0 LINE0     :LLIN2     ' is it line 1
        IN0 LSMSSW    :LLLIN2    ' is it MS 8 sensors, if not line 2 is next
        OPS 0                    ' turn-off LM/MS switch
        IBS 1         :LSCMP     ' branch to scan LS 8 sensors of line 1
LLLIN1  OPS 1                    ' turn-on  LS/MS switch
        IBS 0         :STACMP    ' branch to scan MS 8 sensors of line 1
'
LLIN2   IN0 LINE0     :LLIN3     ' is it line 2
        IN0 LSMSSW    :LLLIN3    ' is it MS 8 sensors, if not line 3 is next
        OPS 0                    ' turn-off LM/MS switch
        IBS 1         :LSCMP     ' branch to scan LS 8 sensors of line 2
LLLIN2  OPS 1                    ' turn-on  LS/MS switch
        IBS 0         :STACMP    ' branch to scan MS 8 sensors of line 2
'
LLIN3   IN0 LINE0     :LLIN0     ' is it line 3
        IN0 LSMSSW    :LLLIN0    ' is it MS 8 sensors, if not line 0 is next
        OPS 0                    ' turn-off LM/MS switch
        IBS 1         :LSCMP     ' branch to scan LS 8 sensors of line 3
LLLIN3  OPS 1                    ' turn-on  LS/MS switch
        IBS 0         :STACMP    ' branch to scan MS 8 sensors of line 3
'
```

APPENDIX C: BNF, ISP

Both BNF (Backus Normal Form) and ISP (Instruction Set Processor) are languages used to describe computer operations. They are used within the body of this report to describe the operations of the BD automata. The BNF defines source language syntax while the ISP describes computer hardware organization interrelated to its operations.

C.1 The Backus Normal Form Language

The BNF is a metalanguage used to describe the grammar and syntax of another language. It was developed back in 1959 to describe the syntax of ALGOL 60 [BA59]. Within the family of "context free" languages it is the most commonly used [GE69,HE73]. FORTRAN 77 for example has been entirely defined using an extended version of the BNF language. The BNF is used in this work to define the correct syntax of the BD assembler language.

The BNF language consists of a set of special symbols and a number of definition rules.

The following special BNF symbols are used:

1. ::= implies "is defined as"

2. | implies " or else "

3. < > delimits the string or name of the defined parameter

Example: <DIGIT> ::= 1|2|3|4|.....9|0

which implies that the word DIGIT is defined as either one of the 10 digits.

Newly defined words or symbols can be used in subsequent definitions. The word DIGIT, for instance, may be used later in the definition of other words containing several digits.

The BNF definition rules are:

1. New words or symbols may be defined using the BNF special symbols and/or previously defined words or symbols.

2. Words or symbols that follow in succession in any combination imply concatenation, i.e. name1 "and" name2 etc.

3. Concatenation preceeds "or" operation

4. The syntax equation is written in the left to right order.

A source language is defined by first defining all primitive symbols. The syntax definition of an up to 6-character word, commonly used to denote program parameter name in many high level languages, is used as a simple example. A parameter name as such may be constructed from any combination of alphabetic charachters and digits. First, the primitives LETTER and DIGIT are defined. These are used to define the alphanumeric character LETDIG and intermediate subnames. Finaly, the paramter NAME syntax is given based on the previously defined subhames primitives:

```
<DIGIT>     ::= 1|2|3|4|.....9|0
<LETTER>    ::= A|B|C|D|.................X|Y|Z
<LETDIG>    ::= <DIGIT>|<LETTER>
<SUBNAMES>  ::= <LETDIG><LETDIG>
<SUBNAMEL>  ::= <LETDIG><LETDIG><LETDIG>
<NAME>      ::= <LETTER>|<LETTER><LETDIG>|<LETTER><SUBNAMES>|
                <LETTER><SUBNAMEL>|<LETTER><SUBNAMES><SUBNAMES>|
                <LETTER><SUBNAMES><SUBNAMEL>
```

Notice that the defined syntax for the parameter name NAME:

1. requires alphabetic letter as the first character
2. allows uppercase letters only
3. allows any length from 1 to 6 characters.

Qualifiers, special symbols and sentence syntax are defined in similar way. The BNF syntax definitions are part of the high level source program compiler. Program logic correctness cannot be defined as systematically as program syntax unless the language is built to a specific limited application. Therefore general purpose high level language compilers can only detect grammatical syntax errors in source code while detecting program logic errors is a task left to the programmer.

C.2 The Instruction Set Processor

Similar to the BNF which is independent of the language whose syntax and grammar it defines, the ISP is independent of the computer architecture whose organization and operations it describes. The ISP [BN71], is a metalanguage consisting of declarations.

There are two main classes of declarations:

1. Declaration of hardware architecture
2. Declaration of processor actions or operations.

Described by the declarations are: memory organization, processor registers, data types, data operations, processor instructions.

Each ISP declaration, describing a part of the computer hardware, consists of:

1. The full part name,
2. An assigned abbreviated name, and
3. A functional description (e.g. part size, numbering sequence).

ISP special symbols are:

1. " \ " separates the full name from its declared abbreviation.
2. " := " indicates content assignment, i.e., the content of part A is assigned to part B.
3. " = > " indicates action sequence.
4. "< >" and "[ ]" these brackets contain information about

size and numbering schemes of memory arrays and registers.

5. " <- " indicates an assignment which is a result of an action.

Example #1: A declaration describing the BD program memory is:

Instruction memory \ Mp[255:0]<15:0>

Meaning: 1. Mp is the abbreviation assigned to describe the instruction memory.

2. The memory has cells numbered from 0 to 255.

3. Each memory cell is 16-bit long. The bits are numbered from 0 to 15.

Example #2: The assignment of the Program Counter content to the instruction address is declared as follows:

Instruction Address \ IA<7:0> := PC<7:0>

Meaning: 1. IA is the abbreviation assigned to the instruction address.

2. PC is the program counter register (assignment of abbreviation should have been done in previous declaration)

3. Both IA and PC are 8-bit in size, bits are numbered from 0 to 7.

4. The content of PC is addigned to IA.

Example #3: The action implied by a certain operation code in the processor instruction may be declared as follows:

$$IR\langle 15\rangle = 0 \Rightarrow X \leftarrow IV\langle IR\langle 13:8\rangle\rangle$$

$$(IR\langle 14\rangle \oplus X) = 0 \Rightarrow PC \leftarrow Mp[IR\langle 7:0\rangle]$$

$$(IR\langle 14\rangle \oplus X) = 1 \Rightarrow PC \leftarrow PC + 1$$

Meaning:
1. If bit 15 of the instruction word is 0, then the input variable IV whose address is indicated by bits 8 to 13 of the instruction word IR is assigned to X representing the selected input.
2. The second action depends on the result of XORing bit 14 of the IR with the selected input X. A 0 result causes the lower 8 bits of the IR to be assigned to the PC. A 1 result causes the PC to be incremented by 1.

The ISP description of a conventional computer is very complex and may require several hundred declarations. An indication as to the simplicity of the BD processor is the fact that it can be described by less than 20 declarations.