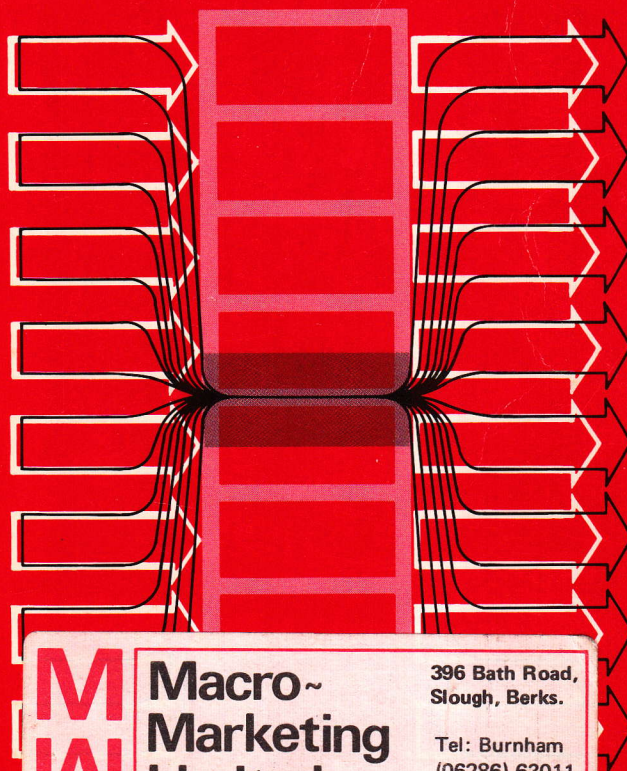




MOTOROLA

MCI4500B

INDUSTRIAL CONTROL UNIT HANDBOOK



**M
W**

**Macro~
Marketing
Limited**

396 Bath Road,
Slough, Berks.

Tel: Burnham
(06286) 63011
Telex: 847083

**THEORY AND OPERATION OF A CMOS ONE-BIT PROCESSOR
COMPATIBLE WITH B SERIES CMOS DEVICES**



MOTOROLA Semiconductor Products Inc.

MC14500B INDUSTRIAL CONTROL UNIT HANDBOOK

Authors

Vern Gregory
Brian Dellande

Principal Contributors

Ray DiSilvestro
Terry Malarkey
Phil Smith
Mike Hadley

© Copyright 1977 by Motorola Inc.
All Rights Reserved



MOTOROLA Semiconductor Products Inc.

PREFACE

A large number of the problems found in controlling electronic and electromechanical devices involve decision oriented tasks. In addition, these decisions usually result in commands as simple as turning something on or off. Some examples are: Is the limit switch closed? Has the timer interval ended? Turn on pump P17 when relays A, B, and C are closed. Send 20 pulses to the triac. Turn on the TILT light. Count 60 pulses and start motor M1, and an infinity of like jobs.

There are, of course, many ways to solve these types of problems. Originally, conceptually simple and easily maintained relays were used extensively. However, relays are bulky, expensive, consume a great deal of power, suffer in terms of long range reliability and also from the fact that they do not lend themselves easily to system changes.

Next came solid state logic. These devices are quite small, have become extremely inexpensive, consume a fraction of the power of a relay and have tremendous long term reliability while remaining conceptually simple and easily maintainable. However, they still suffer from the fact that, once in the system, they are not easily programmable and system changes cannot be made quickly and inexpensively.

Computers and microcomputers may also be used, but they tend to overcomplicate the task and often require highly trained personnel to develop and maintain the system.

A simpler device, designed to operate on inputs and outputs one-at-a-time and configured to resemble a relay system, was introduced. These devices became known to the controls industry as Programmable Logic Controllers (PLC).

The Motorola MC14500B Industrial Control Unit (ICU) is the monolithic embodiment of the PLC's central architecture. Some of the features of the Motorola MC14500B ICU are:

- *16 instructions.*
- *Easily programmed, uncomplicated, no fear of the unfamiliar.*
- *Easily learned, can be maintained by existing personnel.*
- *Uses external memory for versatile system design.*
- *Can be uniquely tailored to a user's particular requirements.*
- *Readily expandable to any size and complexity.*
- *Offers the advantages of programmability.*
- *B series CMOS JEDEC specification*
- *High noise immunity.*
- *Low quiescent current.*

- *3-18 volt operation.*
- *Static operation.*
- *Wide range of clock frequencies, typical 1MHz operation @ VDD = 5V with 1 instruction/clock period.*
- *Instruction inputs-TTL compatible.*
- *Outperforms microprocessors for decision oriented tasks.*
- *Wide range of applications, from relay ladder logic processing, to moderate speed serial data manipulations, to the unloading of overtaxed microprocessor based systems.*

This handbook serves as a design and application manual for the part.

Table of Contents

Preface	i
CHAPTER 1 — Introduction	1
CHAPTER 2 — Basic Concepts	9
CHAPTER 3 — Basic Programming and Instruction Set	15
CHAPTER 4 — Hardware Systems	25
CHAPTER 5 — Demonstration System	31
CHAPTER 6 — Timing, Signal Conditioning and I/O Circuits	41
CHAPTER 7 — OEN and the IF-THEN Structure	55
CHAPTER 8 — IF-THEN-ELSE Structure	59
CHAPTER 9 — While Structure	63
CHAPTER 10 — Complete Enabling Structures	67
CHAPTER 11 — Traffic Intersection Controller	75
CHAPTER 12 — Adding Jumps, Conditional Branching and Subroutines	87
CHAPTER 13 — Modularizing Hardware Systems	91
CHAPTER 14 — Arithmetic Routines	97
CHAPTER 15 — Translating to ICU Code	101
APPENDIX A — MC14599B Addressable Latch	105

CHAPTER 1 INTRODUCTION

The Motorola MC14500B is a single chip, one-bit static CMOS processor optimized for decision-oriented tasks. The processor is housed in a 16-pin package and features 16-four-bit instructions. The instructions perform logical operations on data appearing on a one-bit bidirectional data line and data in a one-bit accumulating Result Register within the ICU. All operations are performed at the bit level.

The ICU is timed by a single phase clock signal, generated by an internal oscillator that uses one external resistor. Alternatively, the clock signal may be controlled by an external oscillator. In either case, the clock signal is available for synchronization with other systems. Each of the ICU's instructions execute in a single clock period. The clock frequency may be varied over a wide range. At a clock frequency of 1 MHz, some 8300, plus, instructions, may be executed in a 60 Hz power line half cycle.

In a system, the ICU may be used in conjunction with the complete line of over 100 standard B-series CMOS logic devices. This allows tailoring a system to an application, and allows a judicious mix of customized hardware and software to be achieved.

As an initial example, Figure 1.1 shows a block diagram of a minimal ICU system with four component blocks in addition to the ICU. The blocks are:

- The ICU, or central controller of the system.
- The memory, either permanent Read Only Memory (ROM) or temporary Random Access Memory (RAM). Here, the steps of the program are stored, both individual instructions and addresses of inputs and outputs.

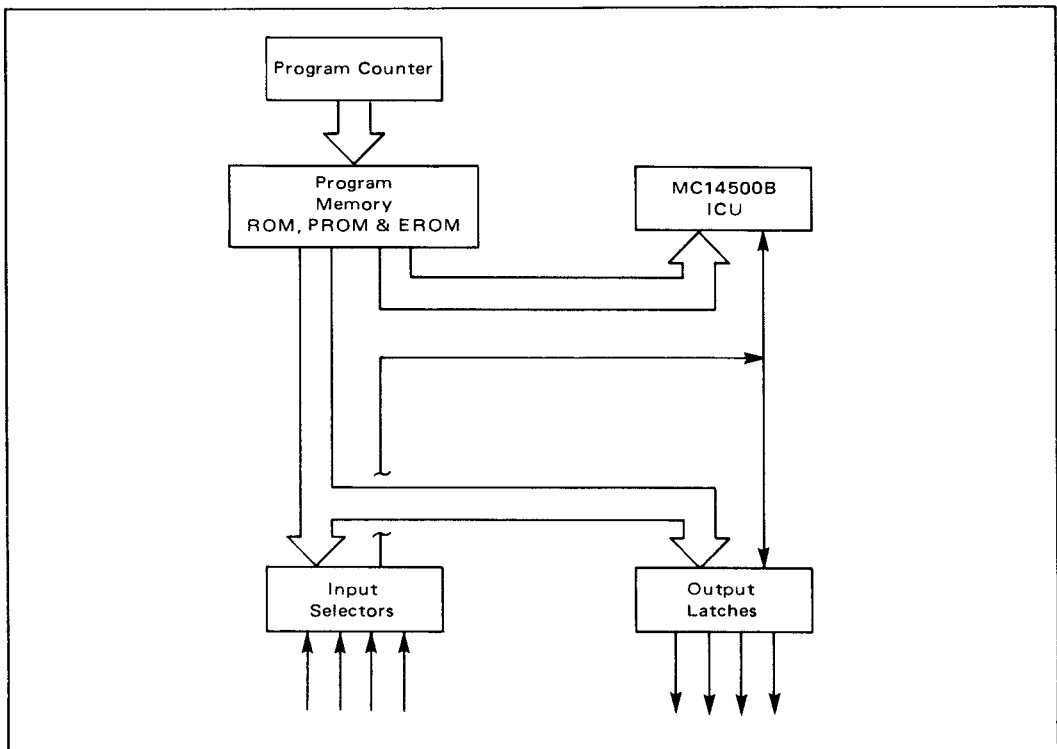


Figure 1.1 Basic ICU System

The instructions, listed in Figure 1.3, are decoded in the Control Logic (CTL), sending the appropriate logic commands to the LU. Further decoding is also performed in the CTL to send a number of output flags (JMP, RTN, FLG0, FLGF) to pins 9 through 12. These are used as external control signals and remain active for a full clock period after the negative-going edge of X1.

Instruction Code		Mnemonic	Action
#0	0000	NOPO	No change in registers. $R \rightarrow R$, $FLG0 \leftarrow \square$
#1	0001	LD	Load Result Reg. $Data \rightarrow RR$
#2	0010	LDC	Load Complement Data $\rightarrow RR$
#3	0011	AND	Logical AND. $RR \cdot D \rightarrow RR$
#4	0100	ANDC	Logical AND Compl. $RR \cdot \overline{D} \rightarrow RR$
#5	0101	OR	Logical OR. $RR + D \rightarrow RR$
#6	0110	ORC	Logical OR Compl. $RR + \overline{D} \rightarrow RR$
#7	0111	XNOR	Exclusive NOR. If $RR = D$, $RR \leftarrow 1$
#8	1000	STO	Store. $RR \rightarrow Data\ Pin$, $Write \leftarrow 1$
#9	1001	STOC	Store Compl. $\overline{RR} \rightarrow Data\ Pin$, $Write \leftarrow 1$
#A	1010	IEN	Input Enable. $D \rightarrow IEN\ Reg.$
#B	1011	OEN	Output Enable. $D \rightarrow OEN\ Reg.$
#C	1100	JMP	Jump. $JMP\ Flag \leftarrow \square$
#D	1101	RTN	Return. $RTN\ Flag \leftarrow \square$, Skip next inst.
#E	1110	SKZ	Skip next instruction if $RR = 0$
#F	1111	NOFP	No change in Registers $RR \rightarrow RR$, $FLGF \leftarrow \square$

Figure 1.3 MC14500B Instruction Set

The timing signals are generated from an on-chip oscillator, (OSC), with the operating frequency set via an external resistor connected between pins 13 and 14. Figure 1.4 shows the relationship between frequency and resistor values. The resultant square wave output appearing at pin 14 is used both within the ICU and as a general system clock. Alternatively, the system may be externally clocked at pin 13.

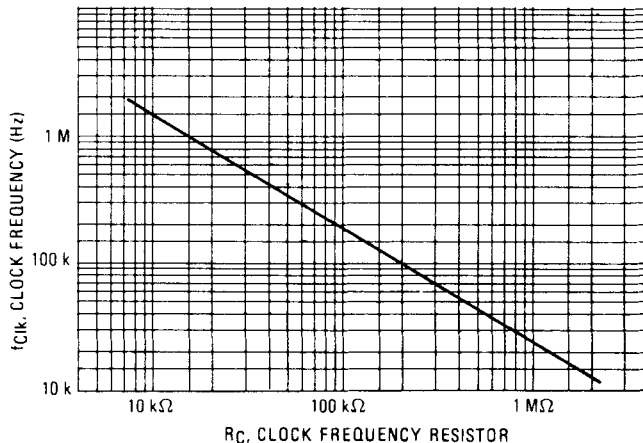


Figure 1.4 Typical Clock Frequency Versus Resistor (R_C)

Two internal latches, Input Enable Register, (IEN), and Output Enable Register, (OEN), control data transfers to and from the ICU. The IEN acts to enable the data path to the LU when in the high state. The OEN, in the high state, enables the Write signal. It should be noted that both of these registers are set via the Data pin.

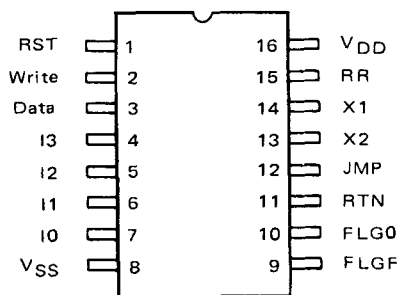


Figure 1.5 Pin Assignment

A Master Reset pin (RST), active high, is provided to clear all registers and hold the FLAG signals within the ICU at zero. The oscillator pin (X1) is held in the high state when RST is high. When RST goes low, the oscillator starts after a delay. In addition, the state of RR is available at the buffered RR pin 15.

The ICU chip is housed in a 16-pin dual-in-line package, available in either plastic or ceramic. The various temperature ranges and package types are as follows:

MC14500BAL: Ceramic package; MIL temperature range

MC14500BCL: Ceramic package, Commercial temperature range

MC14500BCP: Plastic package, Commercial temperature range

Pin assignments are shown in Figure 1.5.

The maximum ratings and the electrical characteristics of the ICU are shown in Figure 1.6. These characteristics conform to JEDEC B-Series specifications governing CMOS B-Series devices which have a recommended supply voltage operating range from 5 to 15 Vdc. In electrically noisy industrial environments, supply voltages of 15 Vdc are recommended to make best use of the excellent noise immunity characteristics of CMOS logic. In addition to being able to work in conjunction with over 100 B-series devices, the ICU also works with non-B-series CMOS parts. Refer to Motorola Semiconductor Data Library, CMOS Volume 5/ Series B, for further information regarding the many devices that are compatible with the ICU.

The switching characteristics and explanatory waveforms are shown in Figures 1.7 and 1.8, respectively. All times are related to the pin 14 clock signal, X1. At this printing, only specifications for typical times, at $V_{DD} = 10$ Vdc, were available. Refer to the MC14500B data sheet for up-to-date specifications.

ELECTRICAL CHARACTERISTICS

Characteristic	Symbol	V _{DD} Vdc	25°	Unit
			Typ	
Output Voltage V _{in} = V _{DD} or 0 V _{in} = 0 or V _{DD}	"0" Level V _{OL}	10	0	Vdc
	"1" Level V _{OH}	10	10	Vdc
Input Voltage* RST, D, X ₂ (V _O = 9.0 or 1.0 Vdc) (V _O = 1.0 or 9.0 Vdc)	"0" Level V _{IL}	10	4.50	Vdc
	"1" Level V _{IH}	10	5.50	Vdc
Input Voltage I0, I1, I2, I3 (V _O = 9.0 or 1.0 Vdc) (V _O = 1.0 or 9.0 Vdc)	"0" Level V _{IL}	10	2.2	Vdc
	"1" Level V _{IH}	10	3.1	Vdc
Output Drive Current D, Write (V _{OH} = 9.5 Vdc) (V _{OL} = 0.5 Vdc)	Source I _{OH}	10	-6.0	mAdc
	Sink I _{OL}	10	6.0	mAdc
Output Drive Current (AL Device) Outputs (V _{OH} = 9.5 Vdc) (V _{OL} = 0.5 Vdc)	Source I _{OH}	10	-2.25	mAdc
	Sink I _{OL}	10	2.25	mAdc
Output Drive Current (CL/CP Device) Other Outputs (V _{OH} = 9.5 Vdc) (V _{OL} = 0.5 Vdc)	Source I _{OH}	10	-2.25	mAdc
	Sink I _{OL}	10	2.25	mAdc
Input Current (RST)	I _{in}	15	150	μAdc
Input Current (AL Device)	I _{in}	15	±0.00001	μAdc
Input Current (CL/CP Device)	I _{in}	15	±0.00001	μAdc
Input Capacitance (DATA)	C _{in}	—	15	pF
Input Capacitance (All Other Inputs) (V _{in} = 0)	C _{in}	—	5.0	pF
Quiescent Current (AL Device) (Per Package)	I _{DD}	10	0.010	μAdc
Quiescent Current (CL/CP Device) (Per Package)	I _{DD}	10	0.010	μAdc

*T_{low} = -55°C for AL Device, -40°C for CL/CP Device
T_{high} = +125°C for AL Device, +85°C for CL/CP Device.

Noise Margin for both "1" and "0"
level = 2.0 Vdc min @ V_{DD} = 10 Vdc

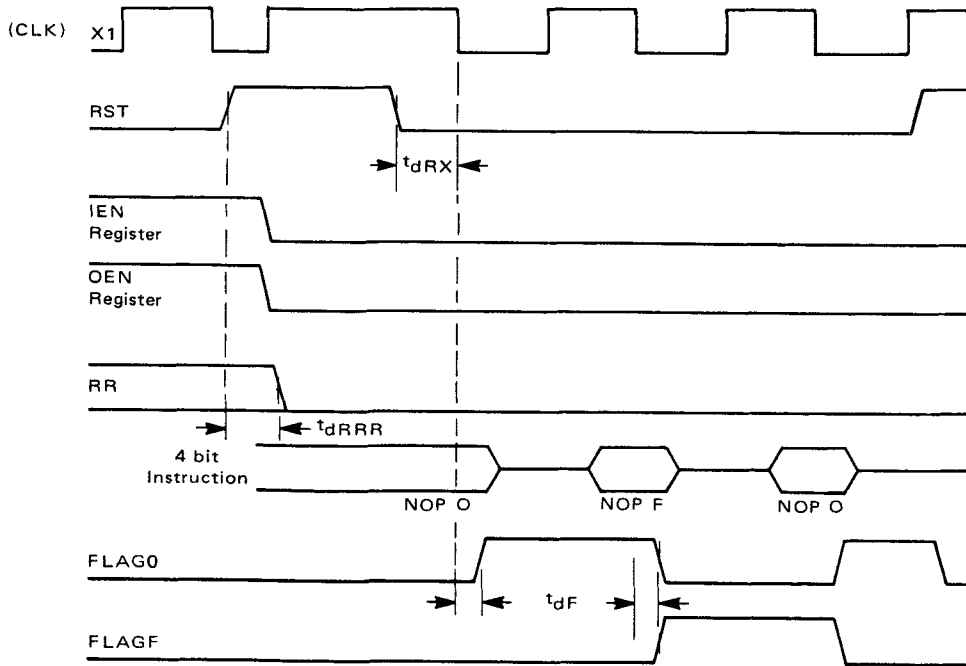
Figure 1.6

SWITCHING CHARACTERISTICS

Characteristic	Symbol	V _{DD} Vdc	All Types	Unit
			Typ	
Propagation Delay Time X1 to RR	t _{dR}	10	110	ns
X1 to FLAGF, FLAG0, RTN, JMP	t _{dF}	10	100	ns
X1 to WRITE	t _{dW}	10	125	ns
X1 to DATA	t _{dD}	10	120	ns
RST to RR	t _{dRRR}	10	110	ns
RST to X1	t _{dRX}	10	120	ns
RST to FLAGF, FLAG0, RTN, JMP	t _{dRF}	10	90	ns
RST to WRITE, DATA	t _{dRW}	10	110	ns
Minimum Clock Pulse Width, X1	PW _C	10	40	ns
Minimum Reset Pulse Width, RST	PW _R	10	50	ns
Setup Time Instruction	t _{IS}	10	125	ns
DATA	t _{DS}	10	50	ns
Hold Time Instruction	t _{IH}	10	0	ns
DATA	t _{DH}	10	30	ns

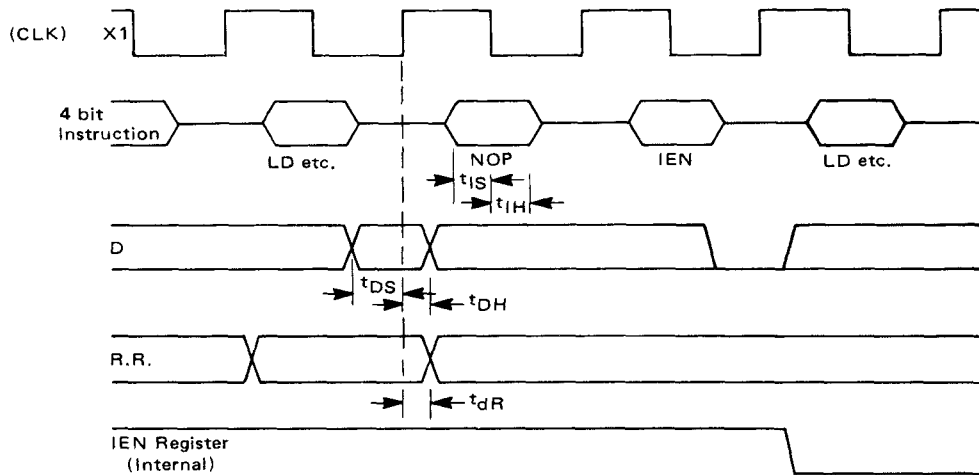
(T_A = 25°C; t_r = t_f = 20 ns for X and I inputs; C_L = 50 pF for JMP, X, RR, FLAG0, FLAGF; C_L = 130 pF + 1 TTL load for DATA and WRITE.)

Figure 1.7



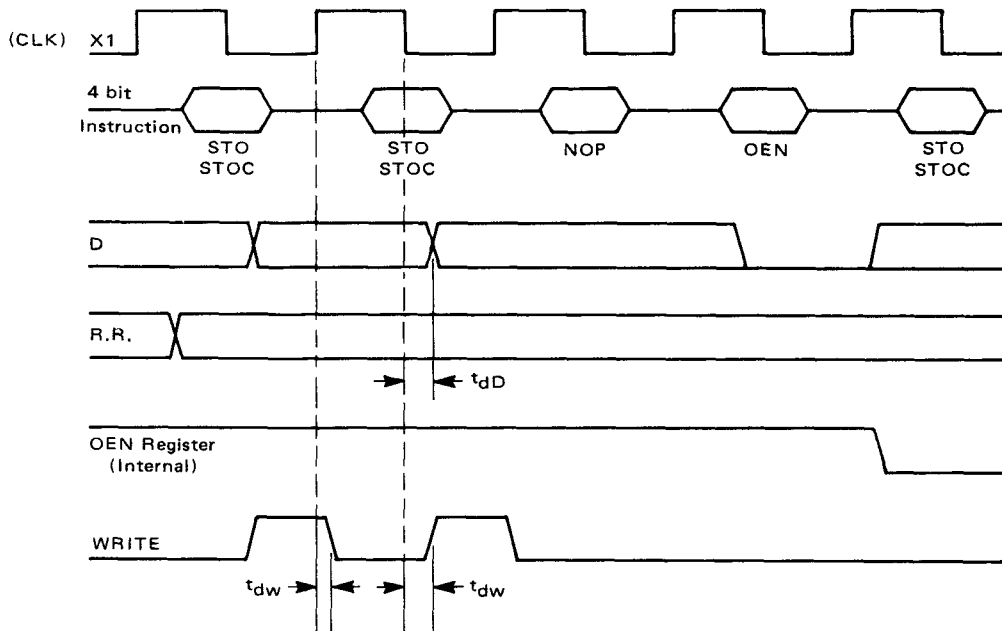
Instructions NOPO, NOPF
RR, IEN, OEN remain unaffected

Figure 1.8 Timing Waveforms



Instructions LD, LDC, AND, ANDC, OR, ORC, XNOR & IEN

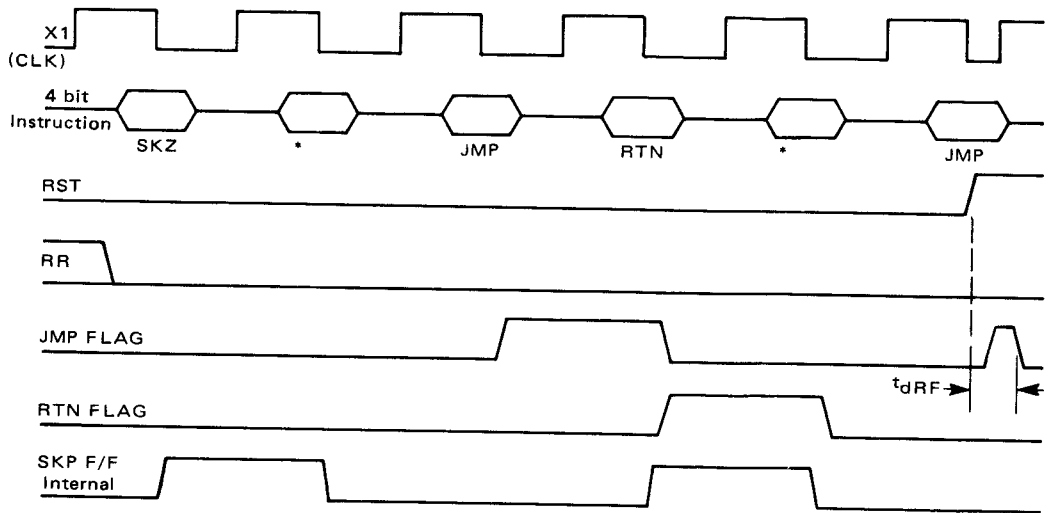
Valid when RST = L



Instructions STO, STOC, OEN

Valid when RST = L

Figure 1.8 Timing Waveforms (Continued)



* Instructions ignored.

Instructions SKZ, JMP, RTN

RR, IEN, OEN remain unaffected.

Figure 1.8 Timing Waveforms (Continued)

CHAPTER 2 BASIC CONCEPTS

The block diagram in Figure 2.1 shows an example of a small ICU-based PLC system. The components, in addition to the ICU, are composed of standard CMOS parts, except for the memory.

The ICU system operates on the principle of a stored program processor. A set of commands, called instructions, reside in the memory of the ICU system. Each command instructs the ICU system to perform one of 16 operations.

The system “fetches” a command, and the necessary information to execute the command, from memory, then “executes” the command. After executing a command, the next sequential command is “fetched” from memory, and the process is repeated ad infinitum.

LD & STO Commands

A typical command might be, LOAD (abbreviated LD). This command instructs the ICU system to read the logic level (logic 1 or logic 0) of an input and store this information in the Result Register within the ICU. To use the LD command, the user programs the memory with the LD instruction and the address of the input to be sampled. The operation of the

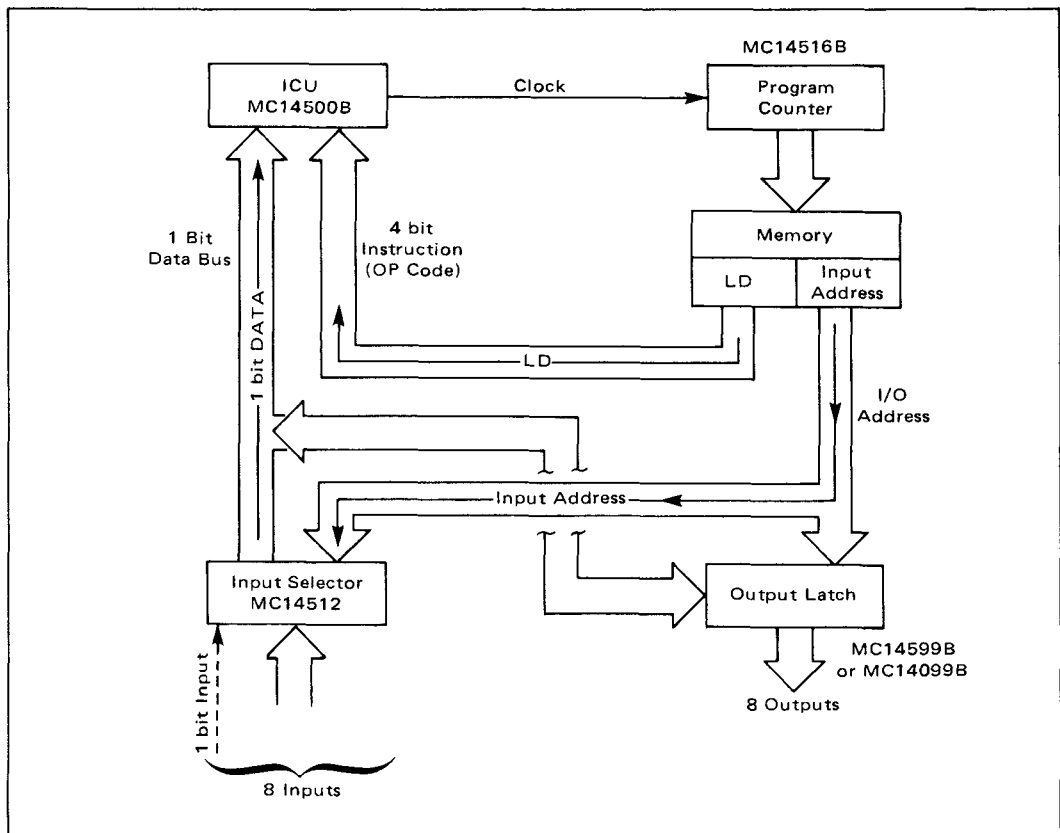


Figure 2.1 Typical Small System Organization & Data Flow

system is as follows: The system memory supplies the ICU with the LD instruction (the instruction is fetched), and supplies the input selector with the address of the input to be sampled. The logic level of the selected input is then transferred over the ICU's one bit data bus to the 1 bit Result Register. (See Figure 2.1).

Another typical command is STORE (abbreviated STO). This command instructs the ICU system to transfer the data contained in its 1 bit Result Register to an output latch. To use the STO command, the user programs the memory with the STO instruction and the address of the output latch which is to receive the data. The operation of the system is as follows. The system memory supplies the ICU with the STO instruction and supplies the output devices with the address of the selected output latch. The data in the ICU's Result Register is then routed to this latch over the ICU's one bit data bus. (See Figure 2.2).

Thus, data can be brought into the system, and also sent out of the system.

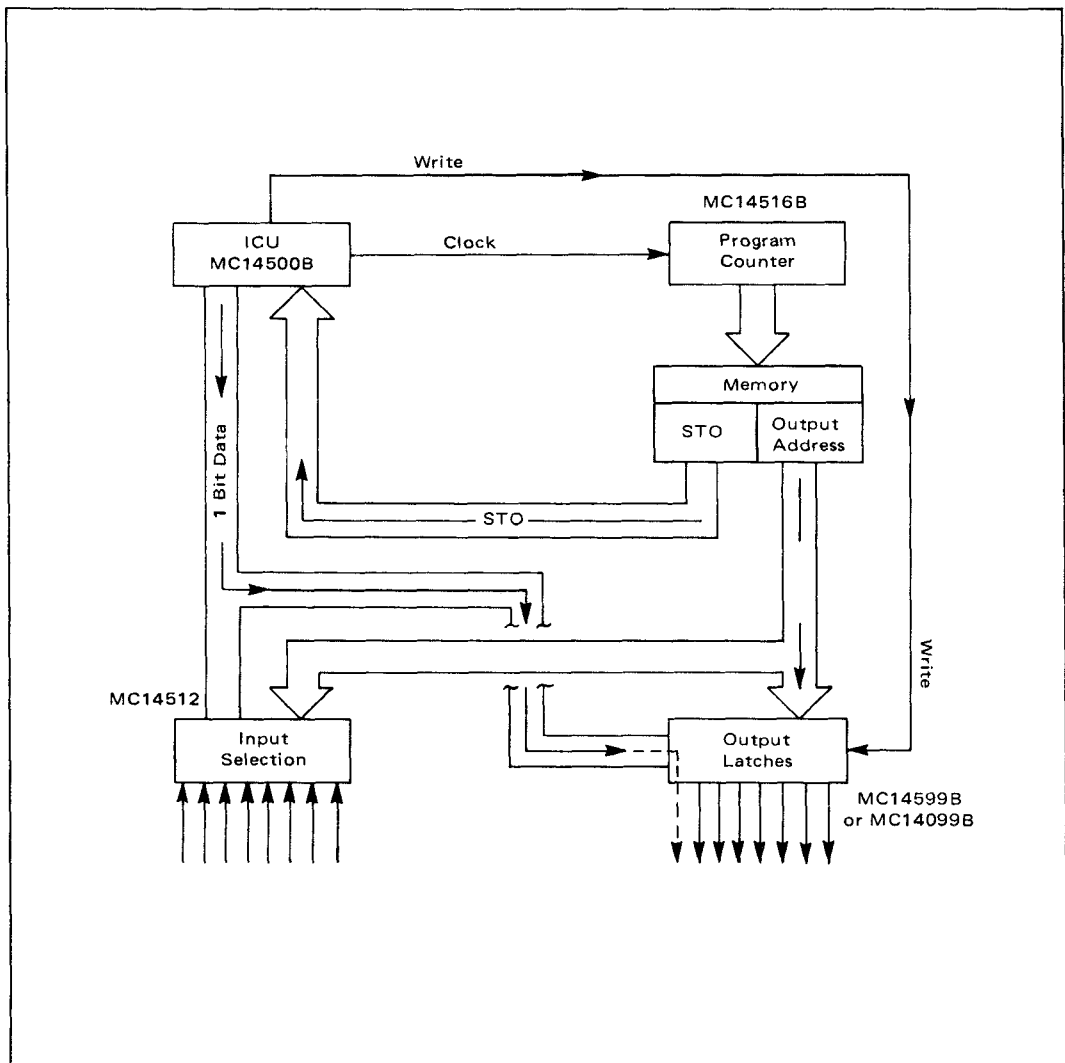


Figure 2.2 System Operation of STO Instruction

SYSTEM COMPONENTS

Memory

The system memory (see Figure 2.3) contains the program which instructs the system to perform its assigned tasks. This program consists of instructions to the ICU in the form of 4 bit operation codes (op-codes) and addresses. The addresses (in binary number form) route the data to and from the ICU's 1-bit bidirectional data bus to the input and output devices.

ICU

The ICU is the central control unit in the system. It controls the flow of data between its internal registers and its 1-bit bidirectional data line, performs logical operations between data in its Result Register and data on its 1-bit bidirectional data line, and sends control signals to the other system components to coordinate the operation of the system.

Program Counter

The program counter (PC) supplies the ICU system memory with the address of the command to be executed. The PC counts up sequentially in binary to its highest value and "wraps around" to zero and counts up again. This causes the sequence of commands in memory to be repeated creating what is known as a looping program.

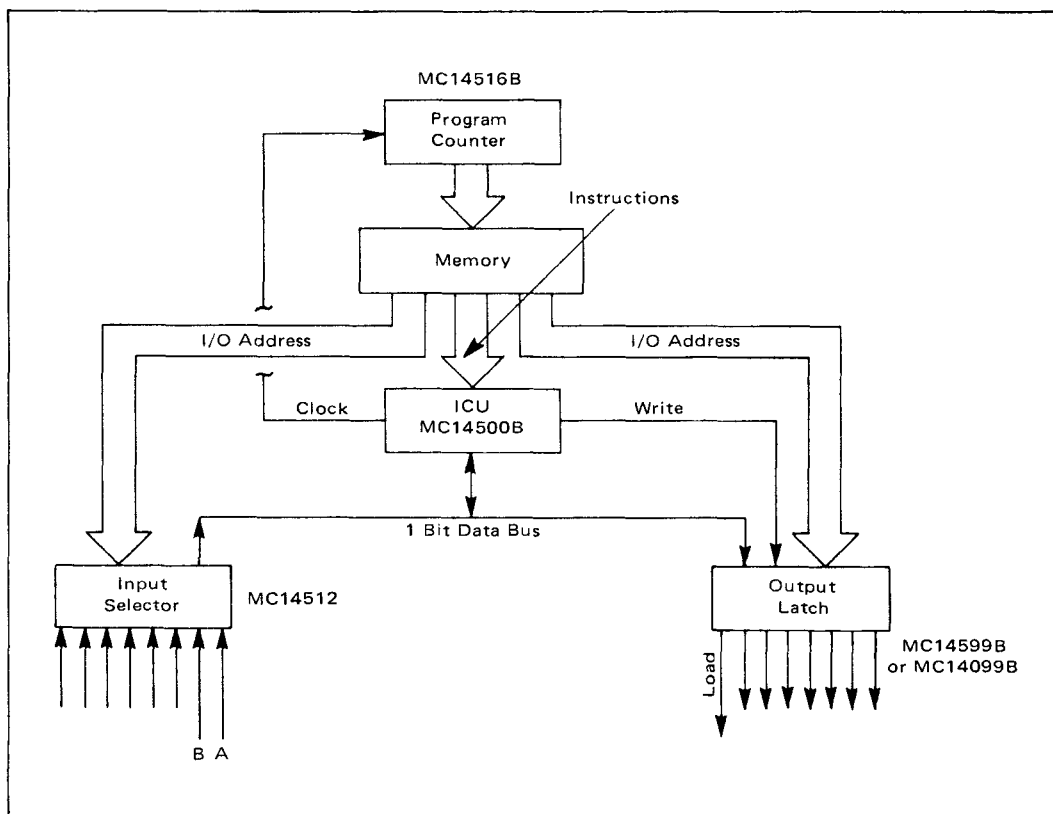


Figure 2.3 Basic System

Input Selectors

The input selectors are used to decide which of the inputs will be used in a particular operation. The ICU system memory supplies the input selectors with the address of the input, then the selector demultiplexes this data onto the ICU's 1-bit bidirectional data line for use by the ICU. Thus, one input is selected from the many inputs.

Output Selectors

The output latches are very similar to the input selectors except the data flow is reversed. When the ICU receives a command to store its Result Register data, it transfers this data to its 1 bit bidirectional data line and signals the output latch with the WRITE control line. The output device then routes this data to the latch specified by the address coming from memory.

The AND Instruction

Before continuing with an example, one more instruction is required — the AND instruction. The operation of the AND instruction is as follows. The system memory supplies the ICU with the AND instruction op-code and supplies the input selectors with the address of an input. The addressed input data is then demultiplexed onto the ICU's bidirectional data line. The information on this line is then logically "ANDed" with the data which is residing in the Result Register. The result of this operation becomes the new content of the Result Register. Notice that the final content of the Result Register will be a logic 1 if and only if the previous content of the Result Register was a logic 1 and the input data was a logic 1. The truth table is:

Input	"AND"	Initial Result Register Contents	=	New Result Register Contents
0		0		0
0		1		0
1		0		0
1		1		1

Example

The basic system of Figure 2.3 is well suited to solving problems presented in the form of relay ladders or solid state logic. Figure 2.4 shows the problem $LOAD = A \cdot B$ in both these forms.

Thus, when A and B are closed (or a logical 1), LOAD is energized (a logical 1).

The ICU solves this problem not once and once only, but once per program loop. Thus, if there are 1000 instructions in the program and the clock frequency is 500 KHz, the inputs will be sampled 500 times per second (every 2 mS) and the output will be energized or de-energized within 2 mS of an input changing. This is known as a looping control structure.

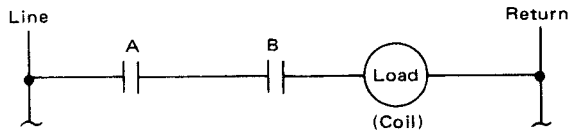


Figure 2.4A Relay Ladder Rung

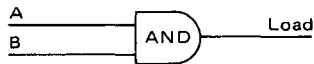
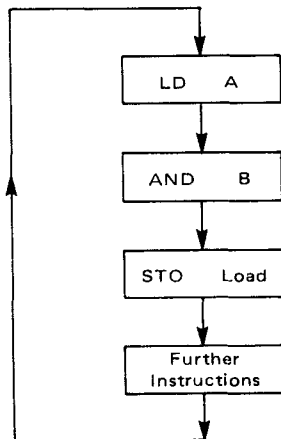


Figure 2.4B Solid State Equivalent of Figure 2.4A

Figure 2.4 Load = $A \cdot B$

Figure 2.5 shows the ICU program required to solve this problem.

Of course, the sequence could just as readily have been: LD B; AND A; STO LOAD.



Loads the state of input A onto the ICU's Result Register (RR)

Logically ANDs the state of input B with the data in the ICU Result Register (which now contains A). The result of this operation becomes the new contents of the Result Register.

Transfers the data in the RR to the output designated LOAD, thus activating or deactivating the load device

Performs remainder of program and loops back

Figure 2.5 Example LOAD = $A \cdot B$ Program

CHAPTER 3 BASIC PROGRAMMING AND INSTRUCTION SET

Accumulating Result Register

The reader will note that the AND instruction introduced the concept of an accumulating Result Register. In the execution of this instruction, the ICU logically performed an AND function on the data on its bidirectional data line with the data in its internal Result Register. The result of this operation became the new content of the Result Register. The point to be made here is that the Result Register always receives the result of any of the ICU's logical instructions. The Result Register therefore accumulates the logical result of each ICU logical instruction. This is analogous to an adding machine which always displays the subtotal after each operation.

Complement Instruction

It is sometimes desirable to activate an output when one input is in the logic 0 state and another input is in the logic 1 state. This situation occurs in relay controlled systems where "normally closed" relays are used, and occurs in solid state logic systems where inverters are present. Figure 3.1 shows an example of this situation.

The ICU instruction set is prepared for this event. Several logical "complement" instructions invert the logic level of the data on the ICU's bidirectional data line before operation on this data.

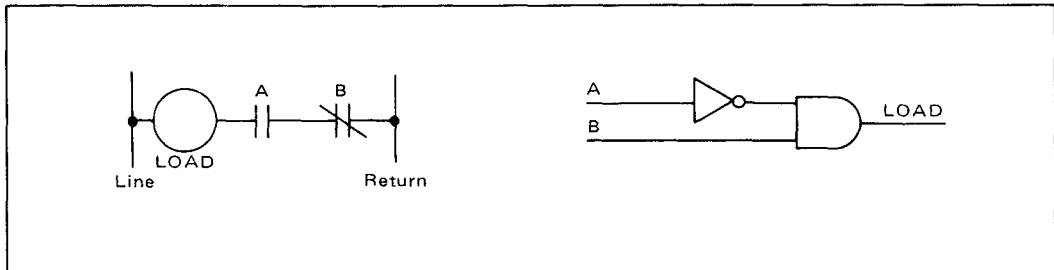


Figure 3.1 Examples of Complemented Signals

The LDC Instruction

An example of one of these instructions is the load complement instruction, abbreviated (LDC). The operation of this instruction is as follows. The ICU system memory supplies the ICU with the LDC instruction and the input selectors with the address of the input to be used in the operation. The input selector then demultiplexes the data of the selected input to the ICU's bidirectional data line. The ICU complements this data and stores the result in its one bit Result Register. The Result Register will receive a logic 1 if the selected input was in the logic 0 state. Figure 3.2 shows an ICU program which solves the problem shown in Figure 3.1, using the LDC command. The reader should be convinced of the operation of this program before reading further.

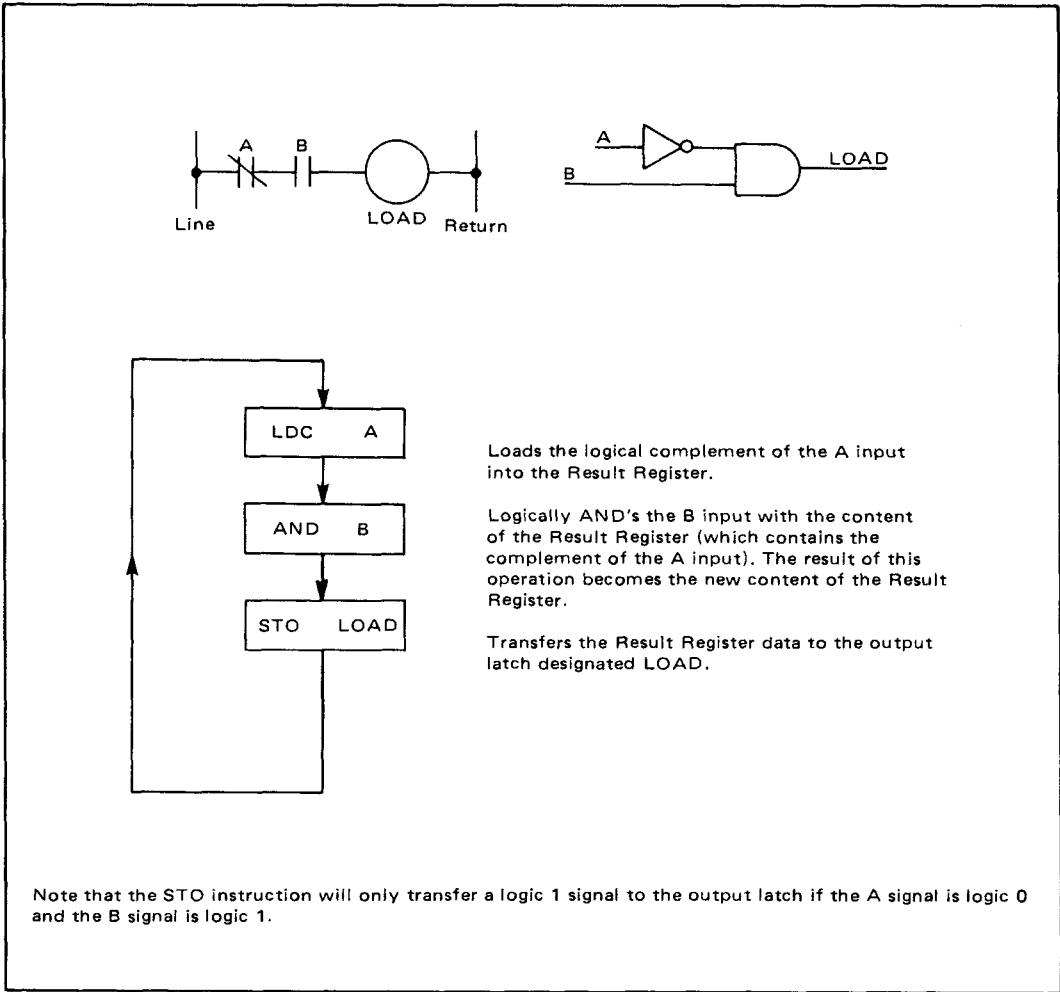


Figure 3.2 Using the LDC Command

The ANDC Instruction

Another example of a logical complement instruction is the “and complement” instruction abbreviated (ANDC). The operation of the ANDC instruction is as follows. The ICU system memory supplies the ICU with the ANDC instruction and the input selectors with the address of a selected input. The input selector then demultiplexes this data onto the ICU’s one bit bidirectional data line. The ICU complements this data and logically AND’s this data with the data in the Result Register. The result of this operation becomes the new content of the Result Register. The Result Register will receive a logic 1 if the input selected was at logic zero and the Result Register previously contained a logic 1.

With the addition of this instruction the ICU is able to attack some more complicated “chain” calculations. Figure 3.3 shows one such example. Figure 3.4 shows an ICU program which solves the problem depicted in Figure 3.3.

In reviewing the operation of these instructions, the reader should be convinced that the load device will only receive a logic 1 signal if $A = 1$, $B = 0$, $C = 1$, and $D = 0$.

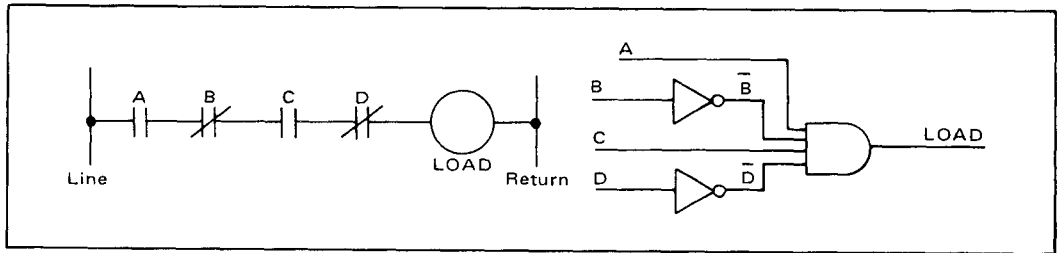


Figure 3.3 Example of a Chain Calculation

Statement	Operator	Operand	Comments
#1	LD	A	Result Register $\leftarrow A$
#2	ANDC	B	Result Register $\leftarrow A \cdot \bar{B}$
#3	AND	C	Result Register $\leftarrow A \cdot \bar{B} \cdot C$
#4	ANDC	D	Result Register $\leftarrow A \cdot \bar{B} \cdot C \cdot \bar{D}$
#5	STO	LOAD	Result Register = $A \cdot \bar{B} \cdot C \cdot \bar{D} \rightarrow \text{LOAD}$

Figure 3.4 Program to Solve the Chain Calculation of Figure 3.3

OR and ORC

In many cases, it is also desirable to activate an output when either input is in the logic 1 state. In this event, the “or” instruction, (OR), should be used. The operation of the OR instruction is as follows. The ICU system memory supplies the OR instruction to the ICU and the address of the input to be used in the operation to the input selectors. The input selector then demultiplexes the addressed data onto the ICU’s bidirectional data line. The ICU then logically OR’s this data with the content of the ICU’s Result Register and returns the result of the operation to the Result Register.

The ICU also has an “or complement” instruction, abbreviated ORC, in the event complement logic is needed. The operation of this instruction is exactly like the OR instruction *except* the incoming data is complemented before the OR operation is performed. Figure 3.5 shows some examples where the OR and ORC instructions may be used.

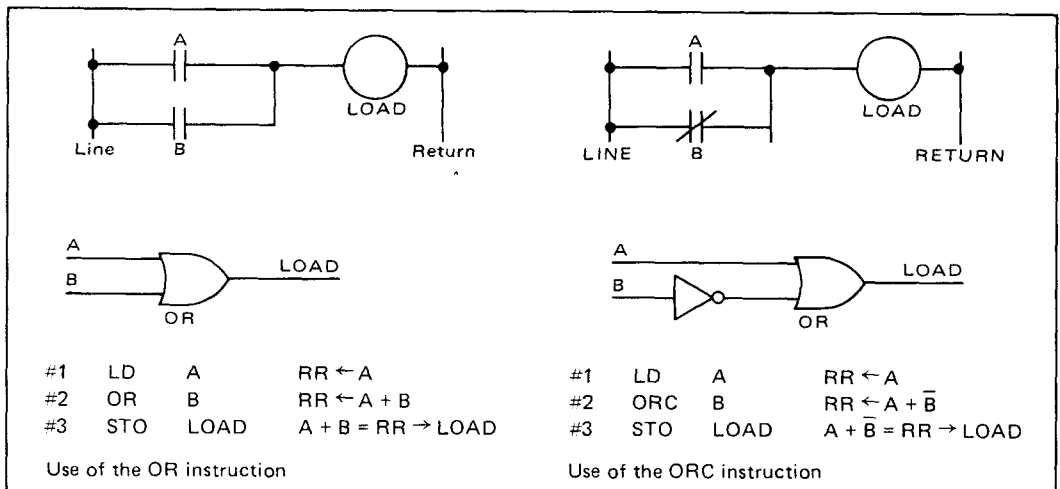


Figure 3.5 Use of the OR and ORC Instructions

In the example of using the OR instruction, the load device will receive a logic 1 signal if the A or B or both inputs are in the logic 1 state. In the example of the ORC instruction, the load device receives a logic 1 signal if the A input is in the logic 1 state or the B input is in the logic 0 state.

Use of Temporary Locations

Many of the logic structures found in the controls industry are branches of several series relays, in parallel with another branch of series relays. Figure 3.6 shows an example of this structure.

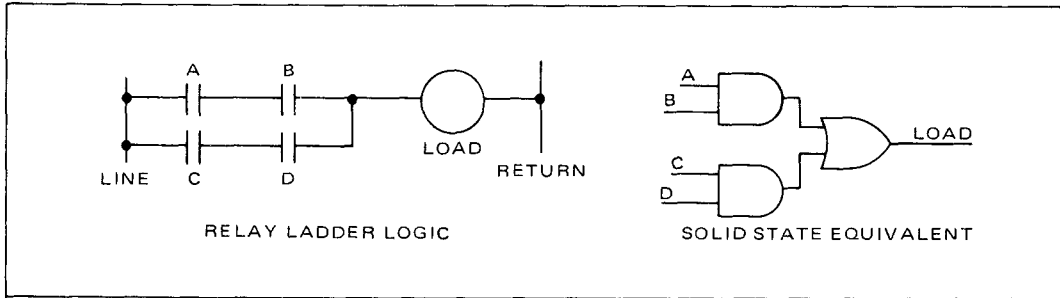


Figure 3.6 Series-Parallel Combinations

When dealing with this type of problem, it is not always possible to directly “chain” a series of LD, LDC, AND, ANDC, OR, and ORC instructions together to correctly evaluate the logic function required. In some cases, it may be necessary to temporarily store the intermediate results before processing the remainder of the problem. In these cases, the programmer must evaluate the series branches using LD, AND, and ANDC instructions as necessary to evaluate the expression and then store the result in a temporary location. The second series branch must then be evaluated and ORed with the data saved in the temporary location. The result of this operation should then be used to activate or deactivate the load device. Figure 3.7 shows a common error in programming this type of problem and Figure 3.8 describes and the correct approach to the problem. Figure 3.8 shows the correct method for solving this problem by using a temporary storage location.

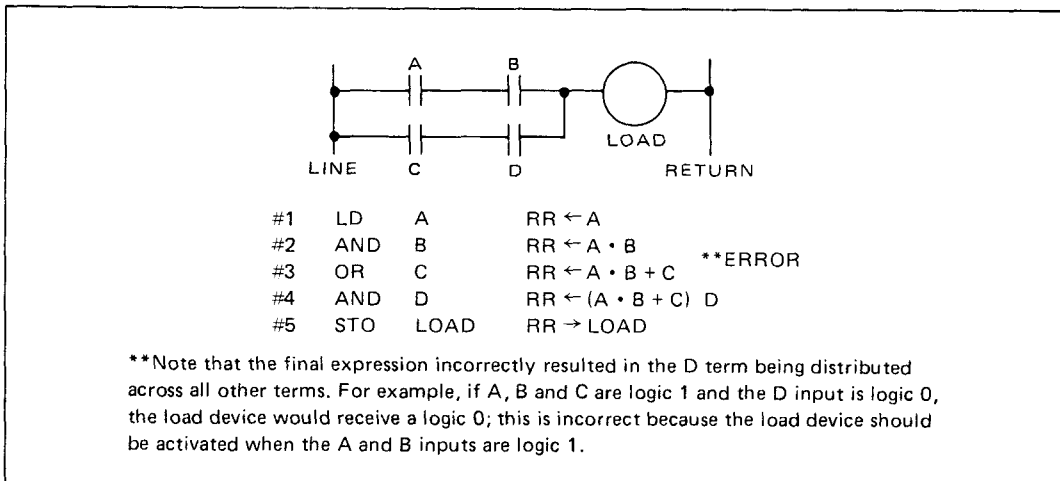
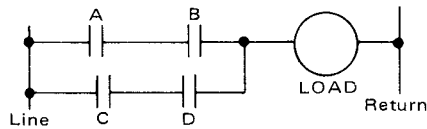


Figure 3.7 Example of Incorrect Programming



```

#1 LD A RR ← A
#2 AND B RR ← A · B
#3 STO TEMP RR = A · B → TEMP
#4 LD C RR ← C
#5 AND D RR ← C · D
#6 OR TEMP RR ← C · D + (TEMP = A · B) = A · B + C · D
#7 STO LOAD RR = A · B + C · D → LOAD

```

In this program, the logical result of ANDing A and B is stored temporarily, then the logical AND of C and D is ORed with the data previously stored in the temporary location. The correct logical signal is then transferred to the load device. This example demonstrates the need for temporary storage locations before proceeding.

Figure 3.8 Correct Method of Solving the Problem

The XNOR Instruction

The “exclusive nor” instruction, abbreviated (XNOR), is the final logical instruction in the ICU’s repertoire of logical instructions. The XNOR instruction can be thought of as a “match” instruction. That is, whenever the input data is identical to the data in the Result Register, the new content of the Result Register will be a logic 1. Figure 3.9 shows the truth table for the XNOR function and Figure 3.10 an example using the XNOR function. Note the reduction in code that may result from the use of this instruction.

Input Data	Old Result Register Data	New Result Register Data
0	0	1
0	1	0
1	0	0
1	1	1

Figure 3.9 XNOR Truth Table

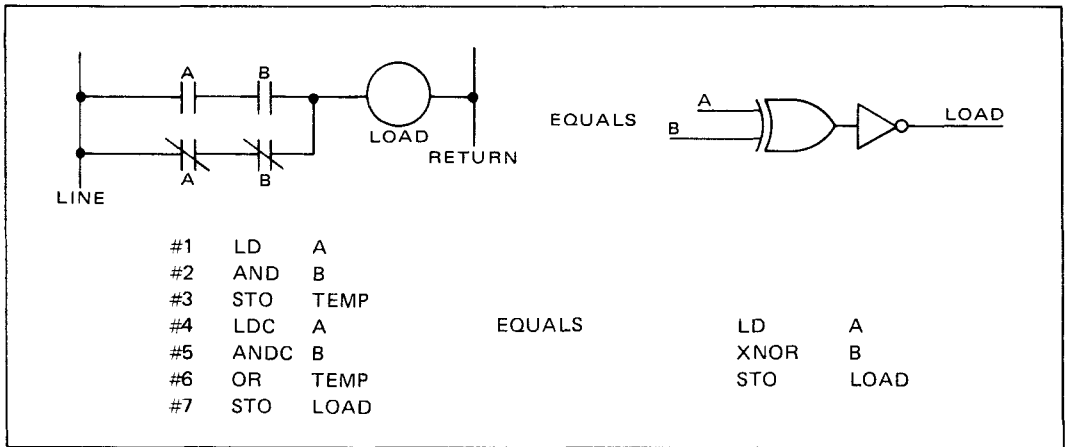


Figure 3.10 Example of use of XNOR Instruction

The STOC Instruction

When transferring a signal to activate a load device, it is very useful to be able to store the logical complement of an expression. The ICU therefore has a “store complement” instruction, abbreviated (STOC). The STOC instruction is exactly like the store (STO) instruction, except the logical complement of the Result Register is transferred to the output latch. It should be pointed out that the Result Register retains its original value (i.e. the STOC does not change the Result Register value, it merely transfers the complement of the Result Register to the bidirectional data line for routing to the output latches). This instruction is quite useful when dealing with negative logic or so called “low active” devices. Figure 3.11 shows an example usage of the STOC instruction. Figure 3.12 shows a problem in both the relay ladder and logic formats. Figure 3.13 shows the problem reduced to code.

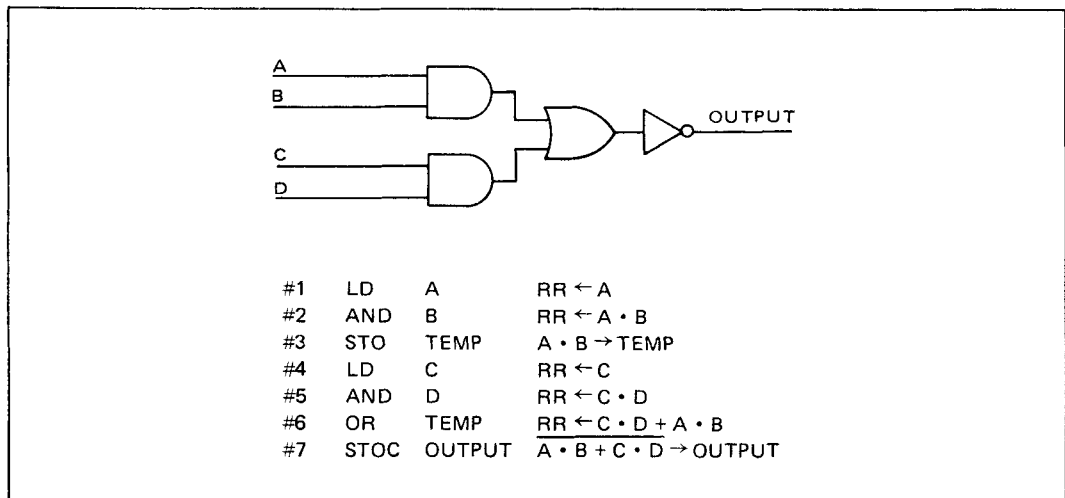


Figure 3.11 Example of the STOC Instruction

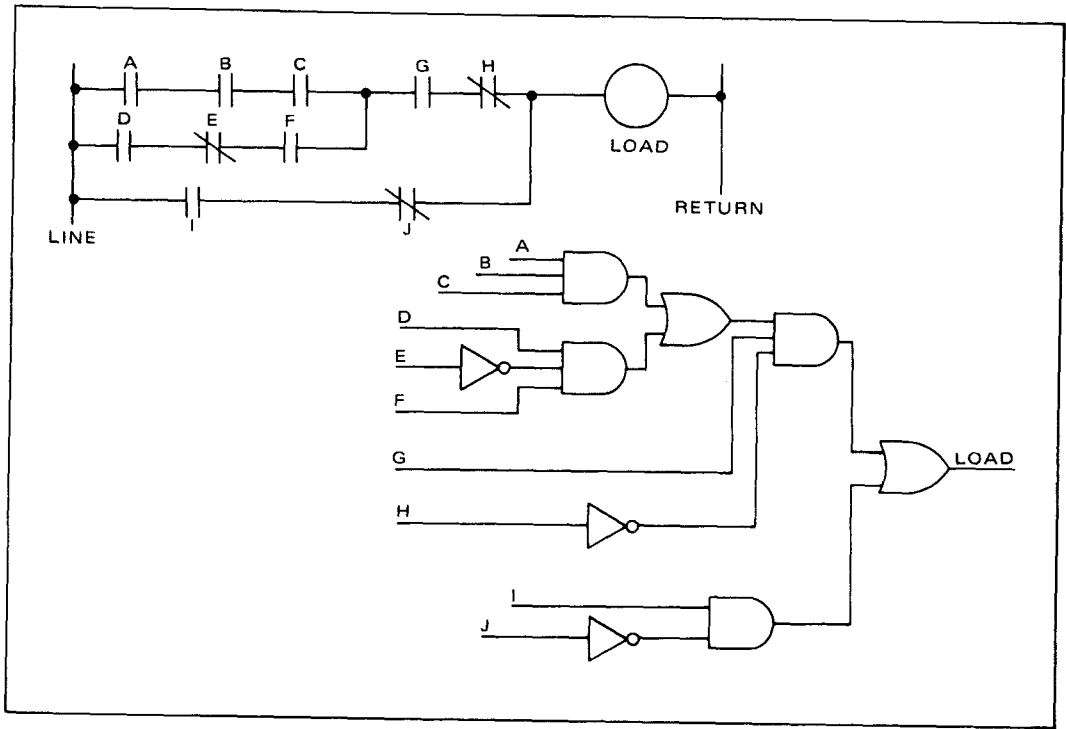


Figure 3.12 Complex Problem

```

#1  LD      A      RR ← A
#2  AND     B      RR ← A · B
#3  AND     C      RR ← A · B · C
#4  STO     TEMP   A · B · C → TEMP
#5  LD      D      RR ← D
#6  ANDC    E      RR ← D · E̅
#7  AND     F      RR ← D · E · F
#8  OR      TEMP   RR ← A · B · C + D · E · F
#9  AND     G      RR ← (A · B · C + D · E · F) · G
#10 ANDC   H      RR ← (A · B · C + D · E · F) · G · H̅
#11 STO     TEMP   (A · B · C + D · E · F) · G · H̅ ← TEMP
#12 LD      I      RR ← I
#13 ANDC   J      RR ← I · J̅
#14 OR      TEMP   RR ← (A · B · C + D · E · F) · G · H̅ + I · J̅
#15 STO     LOAD   (A · B · C + D · E · F) · G · H̅ + I · J̅ → LOAD

```

Figure 3.13 Complex Example Problem Code

The Enabling Instructions, IEN and OEN

In addition to the ICU's logic instructions, the ICU provides two instructions for controlling the program flow in a looping control structure. The reader will remember that, in a looping control structure, each instruction is fetched from memory in sequential order. In some instances, it may be desirable to effectively "jump" over a certain section of the ICU program or to inhibit input data from effecting the system's output.

IEN

The first of these instructions is the "input enabling" instruction, abbreviated (IEN). The operation of the input enabling instruction is as follows. The ICU system memory supplies the ICU with the IEN instruction and the input selectors with the address of the selected input to be used. The input selector demultiplexes the data of the addressed input onto the ICU's bidirectional data line. The ICU then latches the input data into its "input enabling" register. If the input enabling register is loaded with a logic 0, all future input data will be interpreted as logic 0 until the IEN register is loaded with a logic 1 by another IEN instruction. This instruction can be used in a manner similar to the way "master contacts" are used in relay ladder logic. Figure 3.14 shows an example usage of the IEN instruction.

Note, (statement #5), that if the IEN register was loaded with a logic 0 the Result Register can only receive logic 0 data because only LD and AND instructions are used to decide if the load device will be activated.

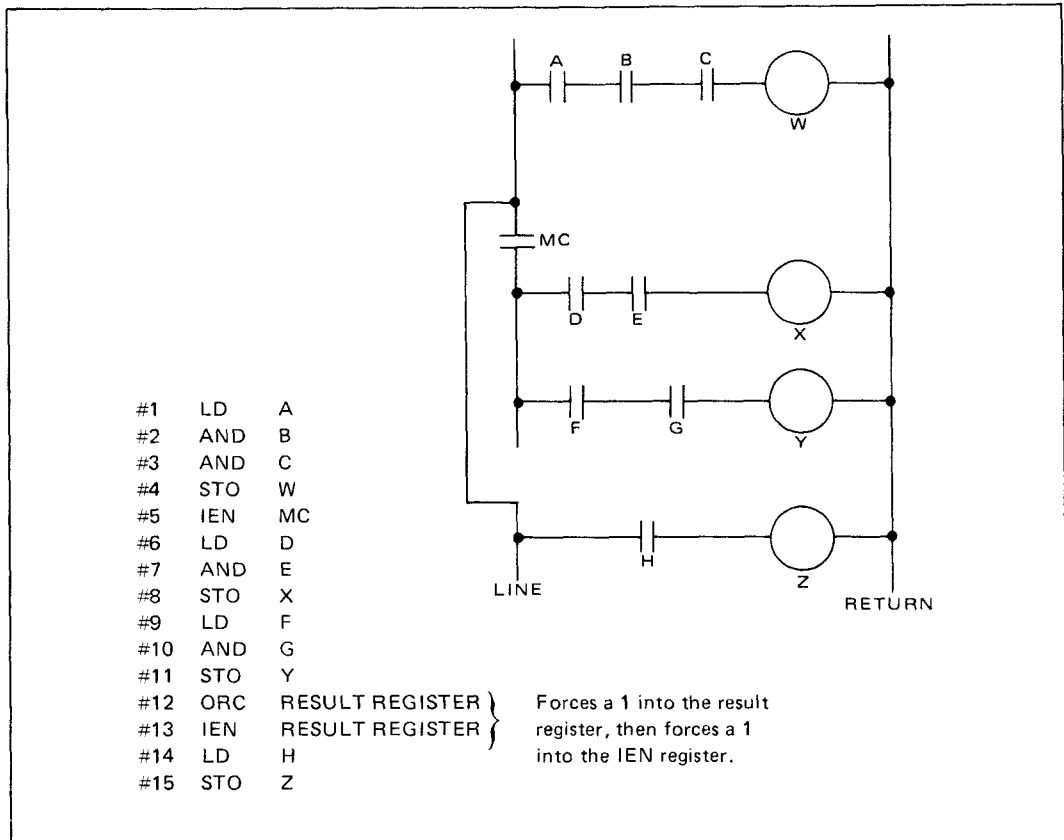


Figure 3.14 Example of Using the IEN Instruction

Caution

Care must be taken using the IEN instruction properly; remember that when the IEN register contains a logic 0 all input data for the ICU will be interpreted as logic 0. This can be tricky. For example, assume the IEN register contains a logic 0. If either an LDC or an ORC instruction is executed, the Result Register will receive a logic 1 regardless of the actual state of the inputs. Additional care must be taken to reload the IEN register with a logic 1 after executing the block of code to be controlled by the IEN register. In the example of Figure 3.14, this is done in statements 12 and 13. Statement 12 forces the Result Register to logic 1 and statement 13 loads the IEN register from the Result Register. Notice that the Result Register data is “pinned out” on the MC14500B and is here assumed to be connected to one of the inputs of the system. In most systems, this connection will be made.

OEN

The second ICU instruction for controlling the operation of programs in a looping control structure is the “output enabling instruction,” abbreviated OEN. The operation of the OEN instruction is as follows. The ICU system memory supplies the ICU with the OEN instruction and the input selectors with the address of a selected input. The input selectors then demultiplexes the data of the addressed input onto the ICU’s bidirectional data line. The ICU then latches this data into its output enabling register (OEN). If a logic 0 is loaded into the OEN register, the WRITE control signal from the ICU is inhibited. Therefore, the output selectors cannot be instructed to receive new data and remain unchanged by STO and

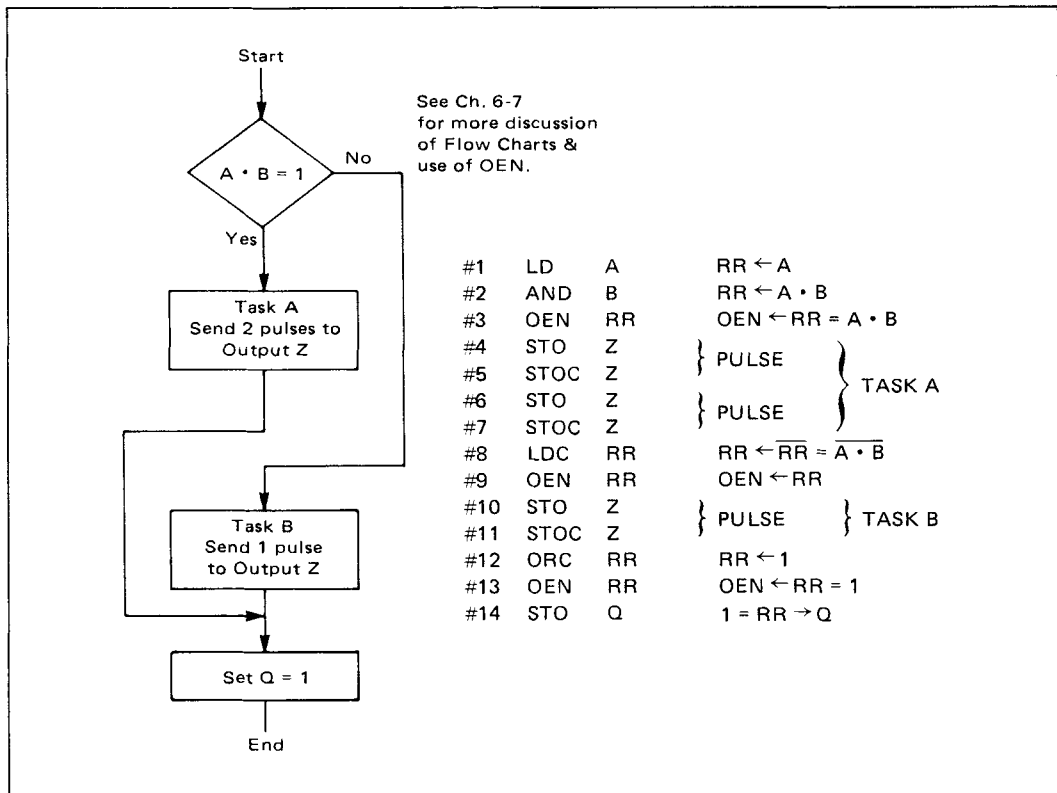


Figure 3.15 Example Use of the OEN Instruction

STOC instructions. The key point is that once the OEN register is loaded with a logic 0, the system outputs remain in their present state until the OEN register is loaded with a logic 1 by another OEN instruction. Then and only then can the system outputs be changed by STO and STOC instructions. Using the OEN instruction, the programmer can effectively “jump” over a block of code by conditionally setting the OEN register to logic 0, causing subsequent instructions to have no effect on the system outputs. The programmer can then set the OEN register back to logic 1 so that future ICU code will operate in normal fashion. Figure 3.15 shows an example use of the OEN instruction. In the example, the program again assumes that the Result Register (RR) is available as a system input. Chapters 7 through 10 describe the OEN structures in greater detail.

In the example of Figure 3.15, if A and B are both true statements 4 through 7 send two pulses to output Z and statements 10 and 11 will not influence output Z effecting a “pseudo” branch around these instructions. If the tested condition fails, statements 4 through 7 will have no effect on the output and statements 10 and 11 will send one pulse to output Z. Statements 12 and 13 return the OEN register to logic 1 so that the Q output will receive a logic 1, and future code will operate in normal fashion. Much more will be said about the use and advantages of an OEN instruction in Chapters 7, 8, and 9.

Thus far, we have studied the LD, LDC, AND, ANDC, OR, ORC, XNOR, STO, STOC, IEN, and OEN instructions. Of the remaining five instructions, two are no operation (NOP) instructions and the other three are for optional use in larger systems which do not have a looping control structure. These will be discussed later.

CHAPTER 4 HARDWARE SYSTEMS

The purpose of this chapter is to begin to acquaint systems designers with the components which are used in a basic ICU looping control system. The system illustrated was not specifically intended to be used in a practical design, however, it illustrates how the components, which comprise the building blocks of an ICU system, might be used. From this point, the system designer can delete, augment or otherwise modify the system illustrated to his own particular needs.

Figure 4.1 is a schematic diagram of a small ICU based system. The system has a looping control structure (i.e. the program counter is never altered by any operation of the ICU.)

System Features

The scheme depicted on Figure 4.1 is a PLC-like system, designed to operate on the principle of a looping control structure. It has 8 inputs, 8 outputs and 8 additional outputs which can be "read" back by the ICU. These outputs can be used for temporary storage. The system memory is capable of holding two separate ICU programs; each individual program can be 256 ICU statements long.

Program Counter

The program counter is composed of two MC14516B binary up-counters chained together to create 8 bits of memory address. This gives the system the capability of addressing 256 separate memory words. The counters are configured to count up on the rising edge of the ICU clock (CLK) signal and reset to zero when the ICU is reset. Notice that the program counter count sequence cannot be altered by any operation of the ICU. This confirms that the system is configured to have a looping control structure.

Memory

The memory for this system is composed of one MCM7641 512-word by 8 bit PROM memory. Because the program counter is only 8 bits wide, only 256 words, (half of the memory), can be used at any one time. However, by wiring the most significant bit of the memory's address high or low, the system designer can select between two separate programs with only a jumper option. This might be a desirable feature if extremely fast system changes are required. Optionally, the designer could chain another counter chip or a single, divide-by-two of flip-flop to the program counter and use the additional memory space for more programming statements. If less than 256 program statements are needed and fast turn around is not needed, a smaller memory may be more economical.

Figure 4.2 shows the format of each memory word. The most significant 4 bits contain the instruction operation code which is routed to the ICU. The 4 least significant bits are routed to the system's input selectors and output latches to address the system's inputs, outputs, and "readable" outputs.

Memory Options

There are, of course, many ways to configure the memory of an ICU system.

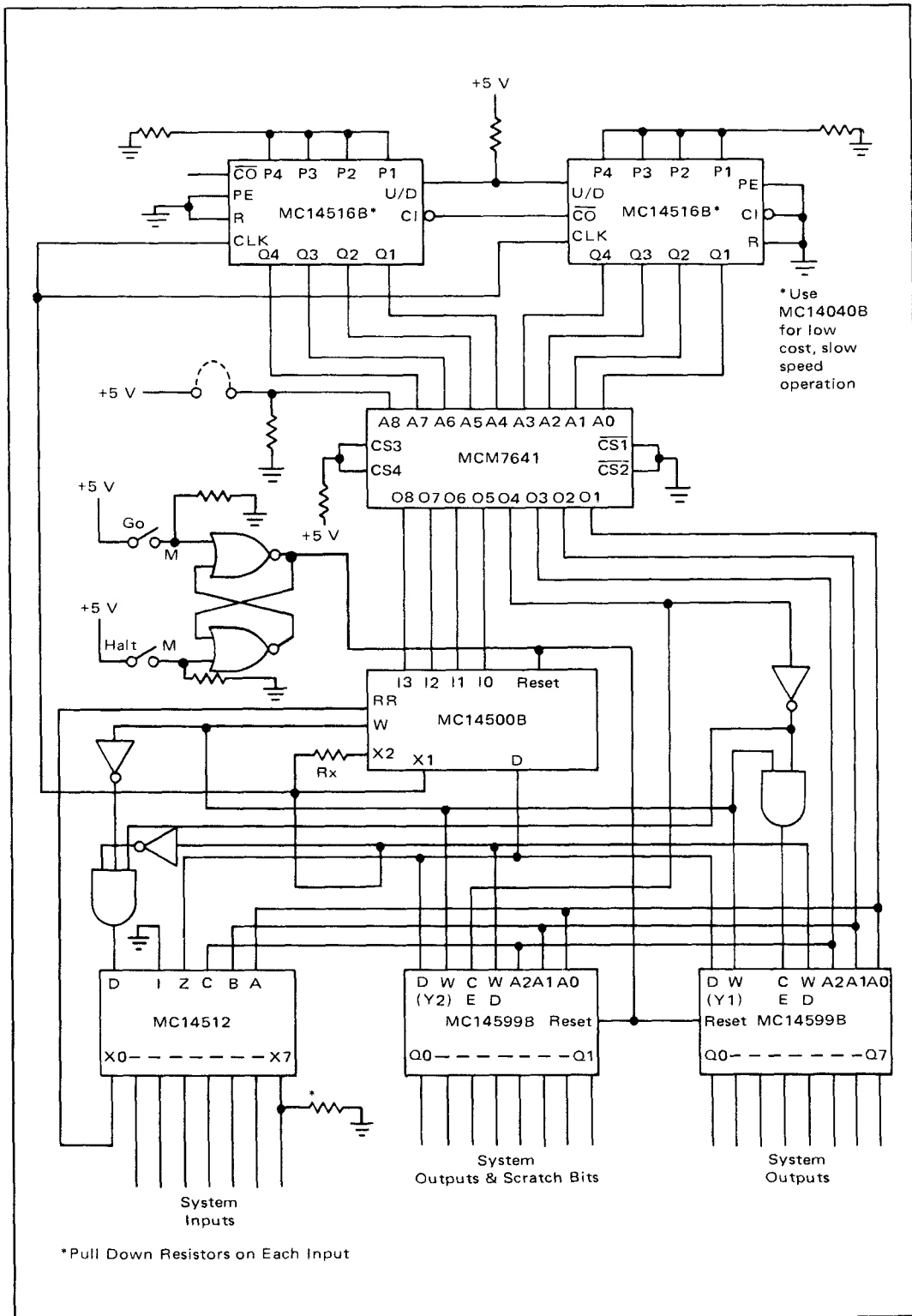


Figure 4.1 A Minimal ICU System

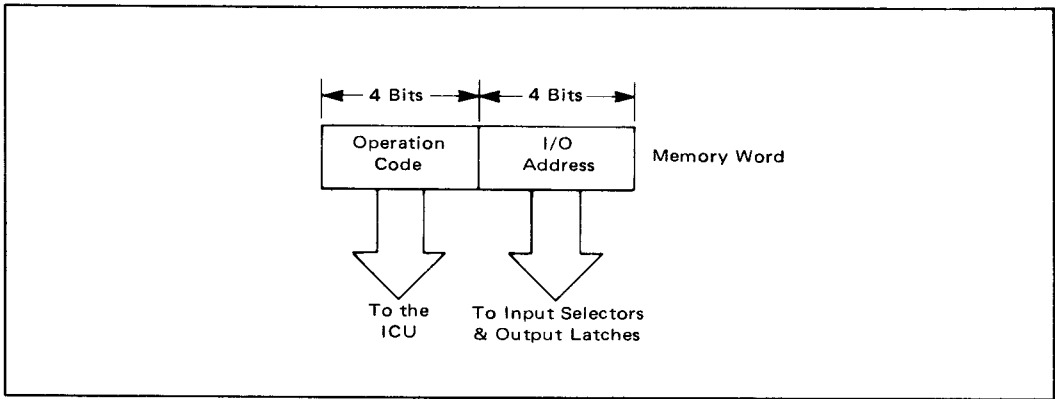


Figure 4.2 Parallel Memory Word Format

Expansion

Figure 4.3 shows a simple approach to expanding the I/O address capability of an ICU system. In this approach, the system memory is broken into two separate sections which share common address lines and bring their data out in parallel. The first of these memories is an N by 4 bit "instruction memory," used to hold only ICU instructions. The MCM7643 1K by 4 bit PROM is capable of holding 1024 ICU instructions and would be a good choice for problems requiring moderate length programs. The second memory is an N word by M bit "address memory" used to hold the address of the operand for each ICU instruction. The MCM7641 512 by 8 bit PROM is a good choice for this application. Two MCM7641, 512 by 8 bit memories and one MCM7643 1K by 4 bit memory would comprise an ICU system memory capable of holding 1024 complete ICU program statements and be capable of addressing 256 inputs and 256 outputs.

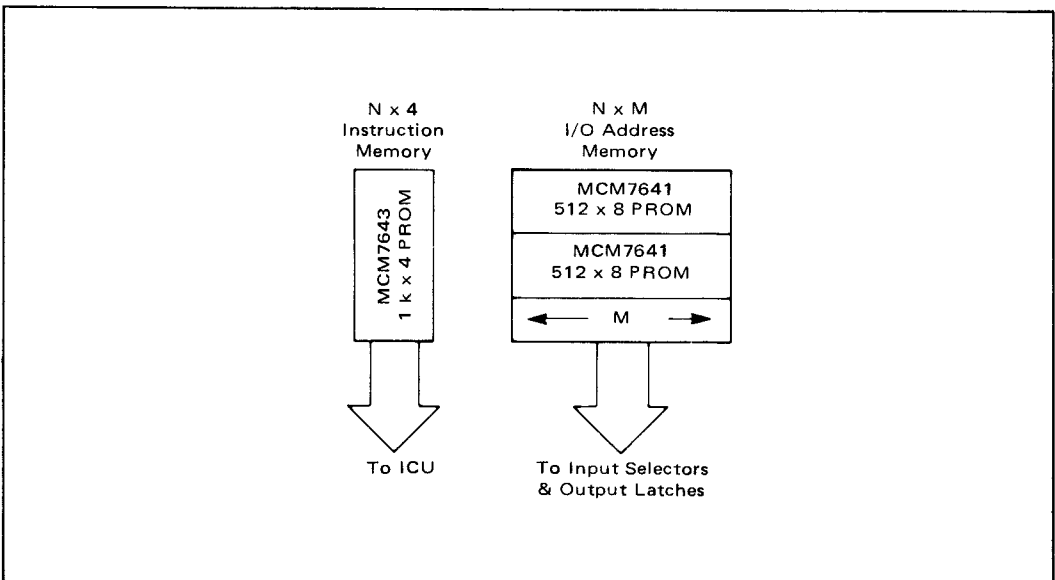


Figure 4.3 An Approach to I/O Address Expansion

Using 4-Bit Wide Memories

It is also possible to “interlace” the instruction operation codes with the I/O addresses in the same memory. In this type of structure, the CLK signal will become the least significant address bit. When the clock signal is high, the memory supplies the ICU with an instruction which will be latched into the ICU on the falling edge of the CLK signal. The memory is then free to supply the I/O sections of the ICU system with an address when the clock signal is low. Figure 4.4 shows this. Thus, a 4 bit wide memory may contain the instructions and addresses for 16 inputs and 16 outputs. This method is used in the demonstration system. Note that as the clock-high and clock-low signals are still used, there is no time penalty involved.

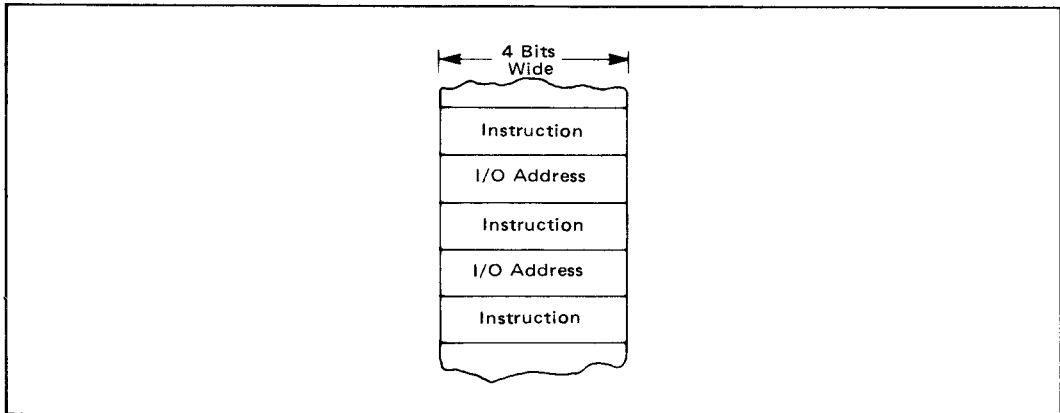


Figure 4.4 Interlaced Memory

Hybrid Expansion

It is also possible to interleave with 8 bit wide memory and thus create a 12 bit wide (4096) I/O structure. See Figure 4.5.

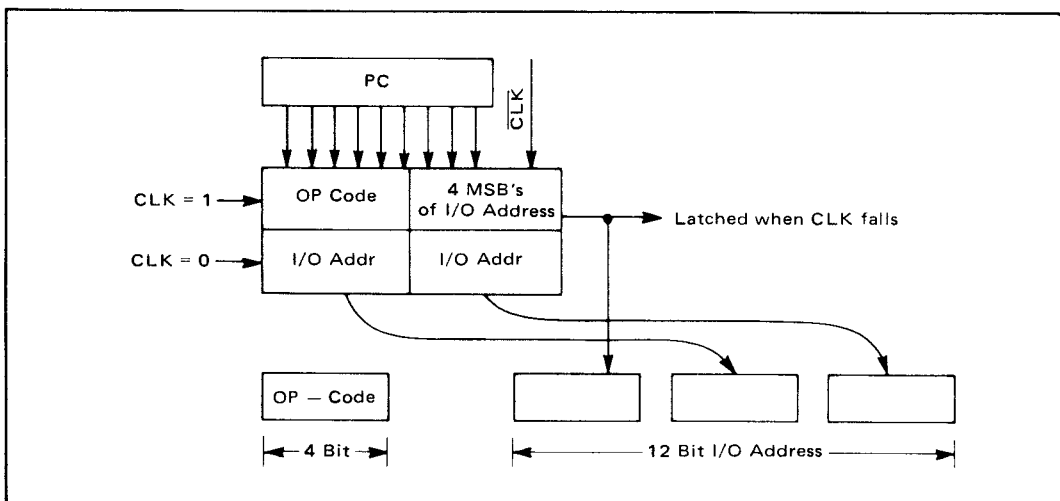


Figure 4.5 Interlaced 8 Bit Memory

Input/Output Structure

The system shown in Figure 4.1 will be considered in more detail here. Figure 4.6 shows the complete I/O map.

Input Selectors

The input selectors used are MC14512 8-channel data selectors. In the example system of Figure 4.1, there is only 1 MC14512 supplying the system with 8 inputs. These inputs occupy addresses 0 through 7 (see Figure 4.6). The input selectors multiplex the addressed input onto the ICU's bidirectional data during the CLK low phase of each ICU machine cycle for all instructions except the STO and STOC instructions. The number of inputs can be expanded easily by adding additional address lines, the proper address decode, and timing.

Output Latches

The output latches are composed of MC14599B 8 bit (bidirectional data port) latches. In the example system of Figure 4.1, the MC14599B labeled Y1 is used strictly as an output latch supplying the system with 8 outputs. These outputs occupy addresses 0 through 7. (See Figure 4.6.) The MC14599 labeled Y2 is configured as a "readable" output latch. In this configuration the part can be thought of as an 8 bit RAM with the outputs of each location pinned out. Because this chip has the read/write feature implemented, it occupies space in both the input and output sections of the I/O address map. The assigned addresses are 8 through 15.

The output selectors receive the data coming from the ICU over the ICU's bidirectional data line. The information is transmitted during the clock low phase of a machine cycle when the ICU executes an STO or STOC instruction, provided the OEN register contains a logic 1. The ICU signals the output latches that a STO or STOC instruction is being executed. The addressed output latches then receive the data and retain its value until the latch is once again addressed and changed. Again, the number and configuration of the output latches can be expanded easily by adding additional address lines, the proper address decode and timing.

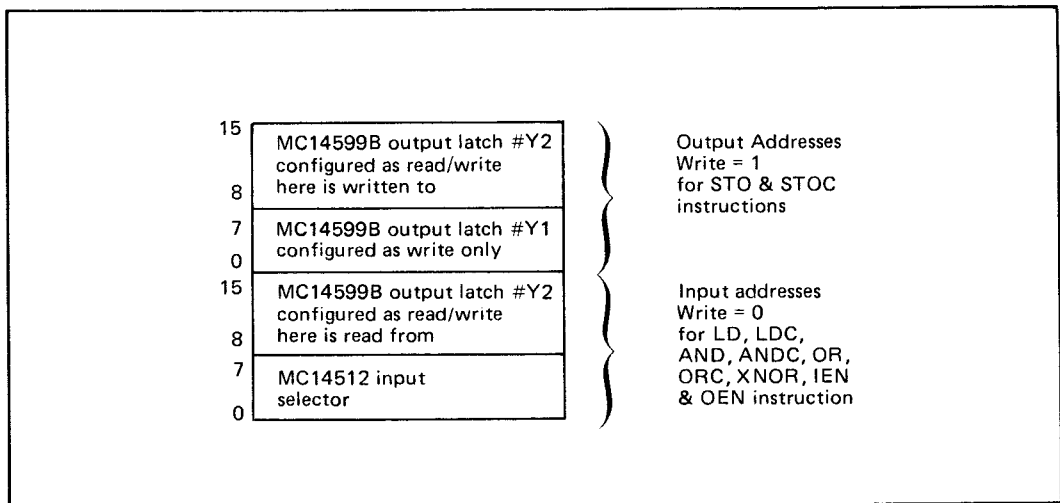


Figure 4.6 I/O Map

I/O Options

In the system shown in Figure 4.1, it may be more desirable to have more system inputs and less temporary storage bits. In this event, the designer can reconfigure the Y2 MC14599B to be a “write-only” output latch. This action would free 8 locations on the I/O address map for 8 more inputs; another MC14512 could be used. The system would then have 16 inputs and 16 outputs. The designer could create temporary storage bits by tying outputs back to inputs. The memory options description showed how memory, and therefore, I/O, may be expanded.

Adding RAM

If the system requires a large number of inputs, outputs and temporary storage bits, it may be more economical to put an N by 1 bit RAM on the data bus rather than using the output latches and input selectors to effect temporary storage bits. See Figure 4.7.

ICU

The MC14500B is the central control element within the system. It coordinates the actions of all the system’s components. The system of Figure 4.1 was designed to use the looping control structure of the ICU. In this type of structure, the Result Register is usually tied to one of the system’s inputs and in this example, the Result Register is returned to input X₀. The ICU’s RESET line is connected to a latch, which is set or reset by two momentary contact switches, giving the system a HALT/RUN feature. Note that when the ICU is halted, the output latches are cleared to zero.

Because the ICU is to be used in a looping control structure, the pulses created by the JMP and RTN instructions are not required. Also, the pulses created by the NOP instructions are not used.

Notice that the ICU has NOP instructions of all 1’s or all 0’s. This was done because the unprogrammed states of PROM’s are all 0’s or all 1’s. Therefore, in a looping control structure, the ICU can be allowed to sequence through these unprogrammed locations without affecting the logical operation of the system.

Chapter 5 contains an example of an “interlaced” memory system and Chapter 12 contains an example of a hybrid (parallel/interlaced) memory system with a scratchpad RAM.

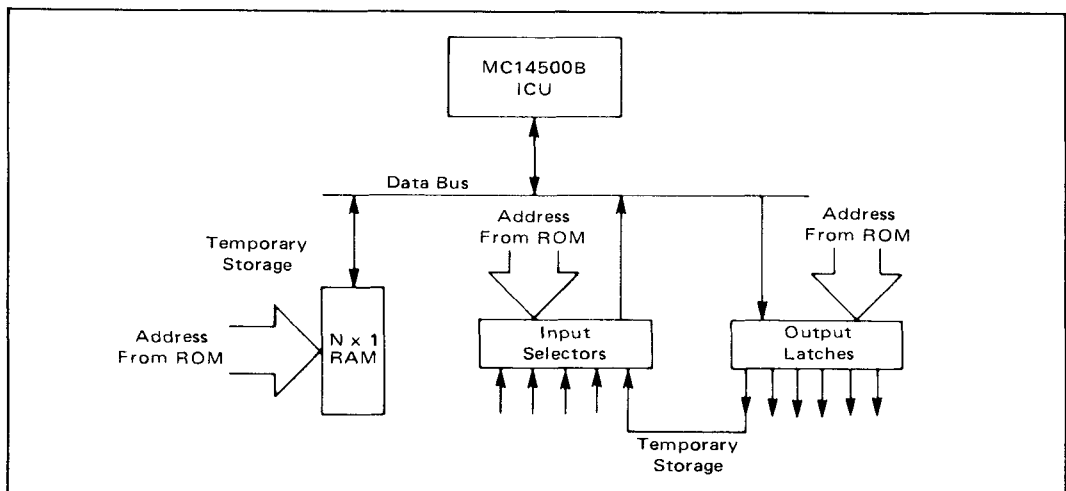


Figure 4.7 Adding RAM to a System

CHAPTER 5 DEMONSTRATION SYSTEM

General Description and Capability

This chapter describes a 16 input and 16 output PLC (Programmable Logic Controller) demonstration system featuring the Motorola MC14500B Industrial Control Unit as the main control element within the system. The system is primarily designed to be used as an educational tool to illustrate the simplicity and power of the Motorola MC14500B ICU. The system illustrates the power of the “looping control structure” found in PLC systems. Therefore, the jumping, conditional branching and subroutine capabilities available in the ICU are not implemented in the system. (However, the programmer will discover that those conventional program control techniques are not necessary, even when writing programs to solve complex control problems.) The unit may also be used as a model for a small system implementation.

The system has 16 inputs and 16 outputs, each numbered from 0 to 15, and a RAM capable of holding 128 ICU program statements. The user is able to examine or change the contents of any location in memory, and has the option of running or single-stepping programs. Alternatively, programmed PROM may be installed in the socket available, and the system run from the PROM. In addition, the demonstration unit displays on LED's, the content of the program counter, the 4 memory data lines, the content of the ICU's Result Register and the current phase of each machine cycle when loading and single-stepping programs. These features provide an easy means to understand the operation of the ICU system and to verify and trouble-shoot ICU programs.

Figure 5.1 shows the basic block diagram of the demonstration system. A schematic of the system is shown in Figure 5.7.

Memory

To reduce cost, a 4-bit wide memory rather than an 8-bit wide memory has been used. This means that the demonstration system is configured with “interlaced” memory such that alternate locations in the memory contain the instruction and its corresponding operand address. During the clock-high phase of a machine cycle, the memory supplies the ICU with an instruction which is latched into the ICU when the clock signal falls. The address of the operand is found in the next memory location and is supplied to the I/O circuitry during the clock-low phase of the machine cycle. This address is used in the execution phase of the ICU instruction. Therefore, the CLK signal is used as the least significant bit of the memory address. The 256 X 4 bit RAM installed in the demonstration unit will hold 128 complete 8-bit ICU program statements. Most statements will result in a 4-bit op-code and a 4-bit operand address being loaded into memory. Note that not all ICU instructions require a corresponding I/O address. In these cases, the I/O address location in memory may be left unprogrammed.

In Figure 5.2 the progression from a normal Instruction-Operand in parallel, to Instruction-Operand in series, and to actual RAM Operation code is shown. Note that there is no difference in program time between the two structures, since both clock phases are used in each case.

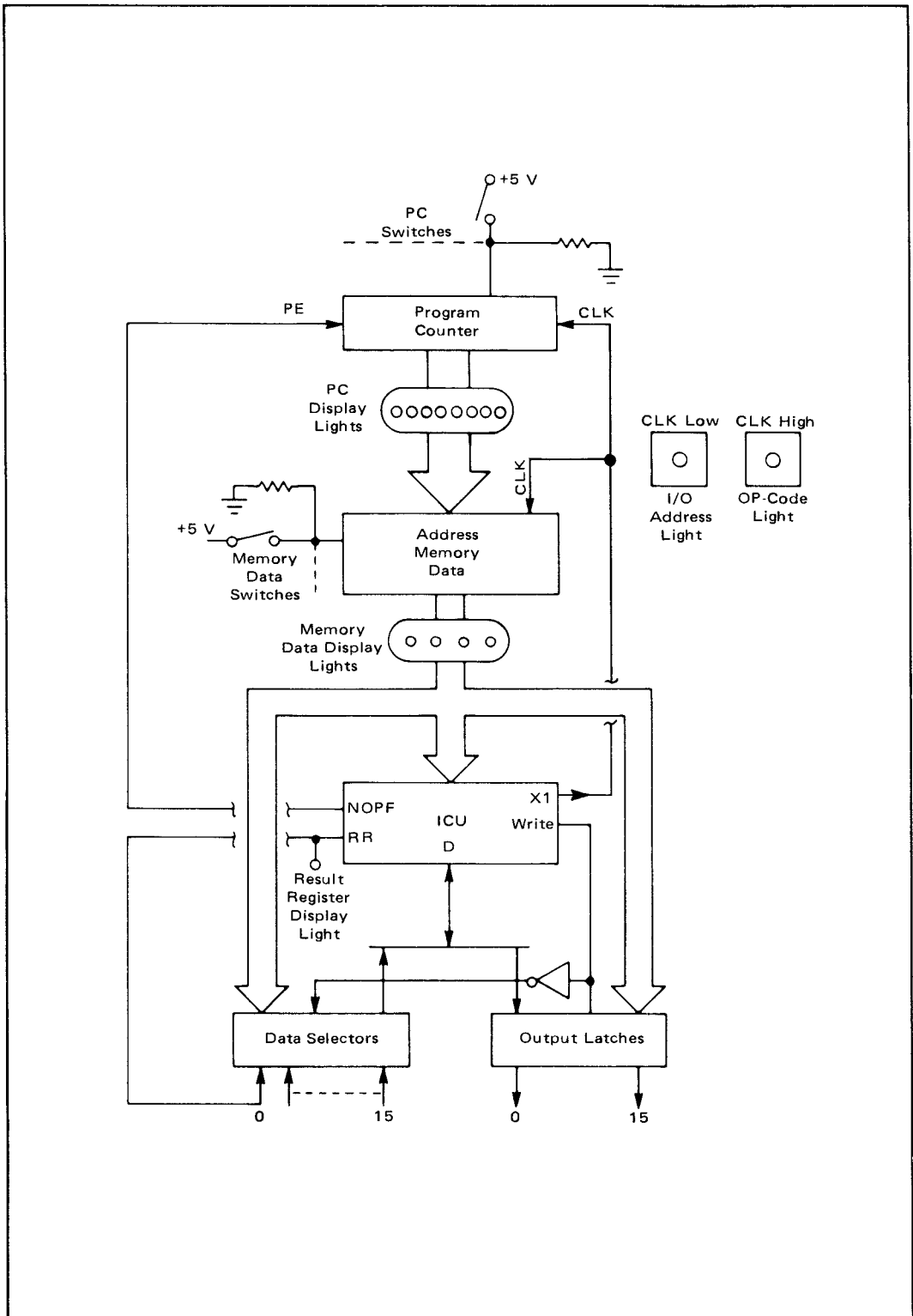


Figure 5.1 Demonstration System Block Diagram

Instruction	Operand	Mnemonic Code	OP Code in RAM
LD	Input # 2	LD	0001
		# 2	0010
AND	Input # 1	AND	0011
		# 1	0001
SKZ	(N/A)	SKZ	1110
		Don't Care	XXXX
STO	Output # 7	STO	1000
		# 7	0111

Figure 5.2 Interlaced Memory Structure of Demonstration System

Program Counter

The program counter supplies the memory of the ICU system with its most significant address bits. The least significant address bit is supplied by the clock (CLK) signal, as explained above. The program counter normally increments on the rising edge of each clock pulse, sequencing the ICU through the programmed instructions in memory. In a non-jumping, non-branching system, the count sequence of the program counter is not altered by the ICU program statements. Therefore, the control program statements are executed in order, until the program counter “wraps around,” and the sequence is repeated. This is known as a “looping control structure.”

The program counter can be thought of as a statement counter; for each unique count, the clock signal will be high and low, causing the memory to supply the ICU system with an instruction and its operand address. This constitutes 1 machine cycle and the completion of 1 ICU instruction.

In the demonstration system, the NOPF instruction (which causes the FLAGF output to pulse for one clock cycle, when the NOPF instruction is encountered) is used to by-pass unprogrammed memory space, to avoid tediously coding to NOPO's and stepping through unused locations. This is done by using the FLAGF output from Pin 9 to preset the program counter to the setting of the program counter switches. Figure 5.3 is an illustration of this, with the program counter toggle switches to zero.

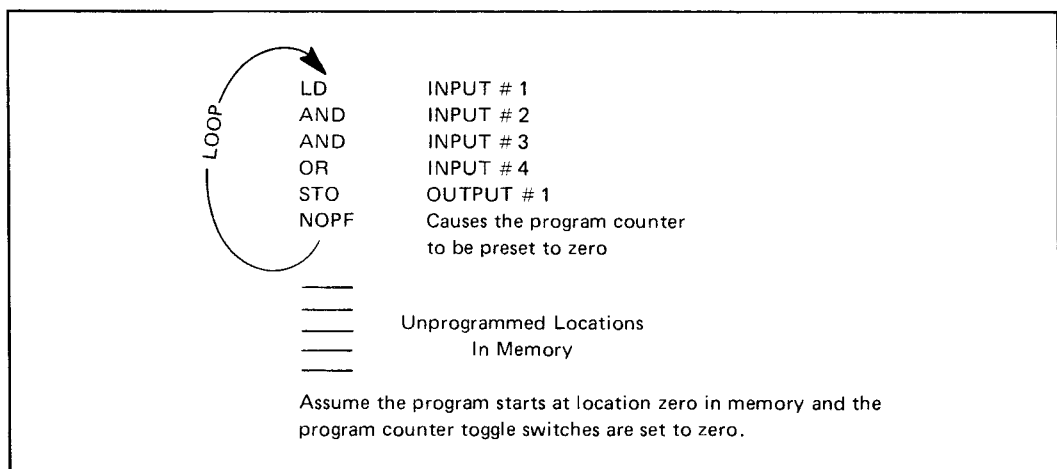


Figure 5.3

ICU and Input/Output System

The MC14500B operates synchronously with a single phase clock which divides the ICU machine cycle into two phases. The first phase (CLK HIGH) is the “fetch” phase — the ICU fetches an instruction from the memory. When the clock signal falls from the high level, the instruction is latched into the ICU’s instruction register. Then, during the second phase, (CLK LOW), the instruction is executed.

There are three types of I/O related instructions—logical, input, and output. During the execution phase of input or logical instructions the operand of the instruction is demultiplexed onto the ICU’s data bus by the input data selectors. The memory supplies the input selectors with the address of the bit to be used in the operation. During the execution phase of an output instruction, the ICU puts the data in its Result Register (or its complement) on its data bus and raises the (WRITE) control line. The data bit is then multiplexed to an output line where it is latched on the rising edge of the clock signal. The memory supplies the address of the output latch, to which, the data is to be routed.

Display Lights

The Program Counter lights show, in binary, the current count of the program counter. These lights can be used to determine which ICU statement is currently being executed when single stepping, and are also useful in keeping track of ICU statements when loading program.

The memory data lights show the content of the memory location currently addressed by the program counter and the clock signal. After data has been loaded into memory, it is displayed by the memory data lights. The lights are also useful in verifying programs entered in memory. This can be done by resetting the ICU, then single stepping through the memory locations with the single step push-button and observing the memory data lights.

The OP-CODE and I/O ADDRESS lights actually reflect the state of the clock (CLK) signal. The OP-CODE light indicates that the clock is high and the I/O ADDRESS light indicates that the clock is low. These lights are very useful when loading programs into memory. The lights indicate to the user whether the operation code of an instruction or the operand address should be entered. The lights also indicate the state of the system, (Fetch or Execute), when single stepping programs.

The Result Register light indicates the content of the Result Register. This is useful in understanding the operation of the ICU logical instructions in the single-step mode.

Functional Switches

RAM/PROM selects which memory, the RAM or the PROM, will be enabled for use by the ICU.

RUN/SINGLE STEP selects which mode the ICU will operate in when the ICU’s RESET line is pulled to logic zero.

DATA switches set the data, either instruction op-code or I/O address, to be loaded into the memory.

PROGRAM COUNTER switches set the memory location to which the data is sent.

LOAD loads the data selected by the data switches into the RAM location indicated by the program counter display lights and the op-code, I/O address lights. After loading data into RAM, the data entered will be displayed by the data display lights.

SINGLE STEP advances the ICU’s clock (CLK) one half cycle per depression. (i.e. the single step push button toggles the clock signal.) The present state of the CLK signal is indicated by the op-code and I/O address lights. (op-code light → CLK = 1, I/O address light → CLK = 0.)

LOAD PC enters the data selected by the program counter switches, into the program counter. After loading the program counter, the value loaded will be displayed by the PC display lights.

RUN latches the ICU's RESET line to logic zero. The ICU will then sequence through the program in memory or the program may be "single stepped" using the single step push button.

HALT/RESET latches the ICU's RESET line high, resetting the ICU. In addition, the system's output latches and program counter are cleared to zero.

Example Problem

The following example shows a typical problem that the ICU may be used to solve. The example illustrates how a problem is reduced to code, and how, using the demonstration system, the code is entered into memory, verified, and executed. The example problem illustrates how an ICU program solves a typical relay ladder logic network, shown in Figure 5.4. In this problem the load device is to be activated if relay A and relay B are closed or if relay C is closed. For the purpose of illustration relays A, B, and C will be represented by switches and the load device activation will be indicated by an LED.

For this problem the following assignments are made:

INPUT # 6	IS TIED TO LOGIC 1	SWITCH # 6 IS ALWAYS HIGH
INPUT # 1	REPRESENTS RELAY A	SWITCH # 1
INPUT # 2	REPRESENTS RELAY B	SWITCH # 2
INPUT # 3	REPRESENTS RELAY C	SWITCH # 3
OUTPUT # 1	REPRESENTS THE LOAD DEVICE	LED # 1

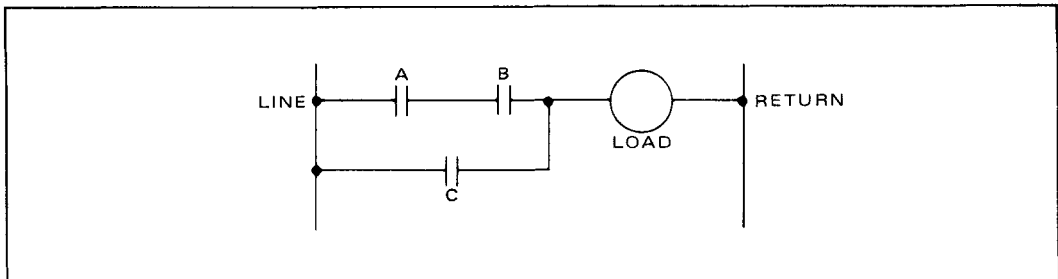


Figure 5.4

Figure 5.5 shows an ICU program which will implement this function and the code to be loaded into memory. The "A" portion of Figure 5.5 shows the ICU interpretation, the "B" portion shows the programming steps.

CAUTION: Note that input zero (0000) is reserved for the Result Register. Therefore, input zero must not be used; if violated, improper system operation will result.

Explanation of Program

Statement # 1 loads the IEN register with a logic 1. If the IEN register contained a logic 0, all future input data for the logical instructions would be interpreted as logic 0.

Statement # 2 loads the OEN register with a logic 1 to enable the output instructions. If the OEN register contained a logic 0, the WRITE strobe from the ICU would be inhibited and the output latches could not be signalled to activate the load.

A: ICU Interpretation

	Instruction	Operand	Notes
1 START	IEN	LOGIC 1	Enable the input register
2	OEN	LOGIC 1	Enable the output register
3	LD	A	Load the state of switch A into the Result Register
4	AND	B	Logically "AND" switches A and B
5	OR	C	Logically "OR" A • B with switch C
6	STO	LOAD	Transfer the result to the load to activate/deactivate it
7 END	NOFF		Causes the program to repeat this sequence

B: Programming Steps

Program Counter	Clock State	Op-Code I/O Address Hex 4-Bits	Binary	Notes
PC = 0	{ CLK High	A	1010	IEN Instruction
	{ CLK Low	0	0110	Address of Logic 1
PC = 1	{ CLK High	B	1011	OEN Instruction
	{ CLK Low	0	0110	Address of Logic 1
PC = 2	{ CLK High	1	0001	LD Instruction
	{ CLK Low	1	0001	Address of A
PC = 3	{ CLK High	3	0011	AND Instruction
	{ CLK Low	2	0010	Address of B
PC = 4	{ CLK High	5	0101	OR Instruction
	{ CLK Low	3	0011	Address of C
PC = 5	{ CLK High	8	1000	STO Instruction
	{ CLK Low	0	0001	Address of Load
PC = 6	{ CLK High	F	1111	NOP Instruction
	{ CLK Low	*	xxxx	No Address needed

*Don't Care

Figure 5.5 Solution to Typical Problem

Statement # 3 loads the Result Register with the state of switch A.

Statement # 4 logically AND's the state of switch B with the contents of the Result Register; this result is then returned to the Result Register. The Result Register will now contain a logic 1 if and only if switches A and B were both high.

Statement # 5 logically OR's the state of switch C with the content of the Result Register; this result is then returned to the Result Register. The Result Register will now contain a logic 1 if and only if switches A and B were high or switch C was high.

Statement # 6 stores the content of the Result Register in the output latch. If the Result Register contained a logic 1, the output latch would receive a logic 1 to activate the load. The STO instruction does not alter the content of the Result Register.

Statement # 7 creates a pulse on pin # 9 of the ICU chip. This signal is used to preset the program counter to the beginning of the program. The entire sequence is then repeated.

The following is a detailed procedure for entering, verifying, single stepping and running the example program.

1. Entering the program to RAM.

- A. Set the RAM/ROM and RUN/SINGLE STEP switches to RAM and SINGLE STEP RESPECTIVELY.
- B. Set all the PC switches to zero.
- C. Press the HALT/RESET push button. This resets the PC to zero, resets the ICU, the output latches and sets the CLK signal high. The OP-CODE light will indicate that the CLK signal is high and an instruction should be loaded into memory.
- D. Set the data switches to hex A (OP-CODE of the first instruction), binary 1010 and press the LOAD push button. The 1010 pattern will be displayed by the data lights.
- E. Press the SINGLE STEP push button once. This toggles the CLK. The I/O address lights will indicate that the CLK is low and an I/O address should be loaded into memory.
- F. Set the data switches to hex 6 (ADDRESS of switch six), binary 0110 and press the LOAD push button.
- G. Press the SINGLE STEP push button once. Note the PC has incremented and the CLK is high indicating the next complete statement should be entered.
- H. Set the data switch to the bit pattern of the next piece of data to be entered — 1011 in this case.
- I. Press the LOAD push button.
- J. Press the SINGLE STEP push button once.
- K. REPEAT STEPS H, I, J UNTIL THE ENTIRE PROGRAM HAS BEEN ENTERED.
- L. NOTE: The NOPF instruction does not require that an I/O address be entered in memory. The I/O address location in memory for this instruction may be left unprogrammed.
- M. Press the HALT/RESET push button.
STOP

2. Verifying the program entered in RAM.

- A. Press the HALT/RESET push button. The PC will be reset to zero, the CLK will be high and the first piece of data entered, (1010), will be displayed by the data lights.
- B. Press the SINGLE STEP push button once. The second piece of data entered, (0110), will be displayed by the data lights. The entire program may be verified by sequencing through memory with the single step feature, while observing the data display lights. The PC lights and the OP-CODE and I/O address light will aid in keeping track of particular ICU statements.
- C. Press the HALT/RESET push button.
STOP

3. Single stepping the program.

Set switch # 6 and switch # 3 high. Setting these switches high will cause light # 1 to activate on the (CLK LOW) phase of the 6th (STO) instruction.

- A. Press the HALT/RESET push button.
- B. Press the RUN push button.

The processor may now be sequenced through the program entered in memory by using the single step feature. Each depression of the SINGLE STEP push button advances the CLK 1/2 cycle. The display lights will aid in understanding the operation of the system as it is single stepped.

- C. Press the HALT/RESET push button.
- STOP

4. Running the program.

- A. Press the HALT/RESET push button.
- B. Set the RUN/SINGLE STEP switch to RUN.
- C. Press the RUN push button.

Switch # 6 should be set high. This enables the IEN and OEN registers. The reader will now note that light # 1 is activated when switches 1 and 2 are both high or when switch # 3 is high. The processor may be halted by pushing the HALT/RESET push button. The following Figure 5.6 is a program the reader may implement as an exercise. (The ANDC and ORC instruction will be useful). Figure 5.7 is the schematic of the system, with the major areas partitioned and labeled.

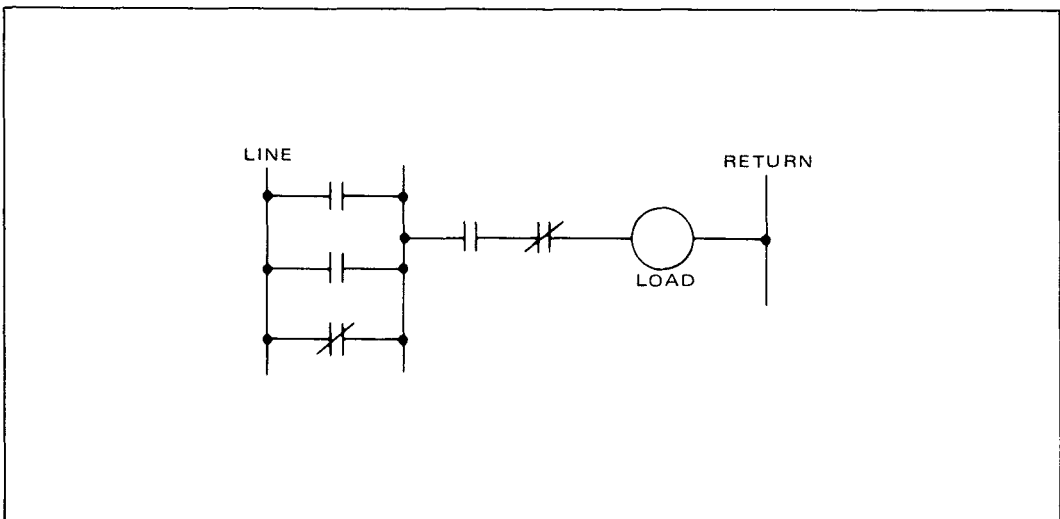


Figure 5.6 Reader's Problem

Device	+5 V	Gnd
MC14011B	14	7
MC14013B	14	7
MC14023B	14	7
MC14095B	1	5
MC14501B	14	8
MC14512	16	8
MC14516B	16	8
DM74571	16	8
MM2101.1	22	8
MC14095E	16	8

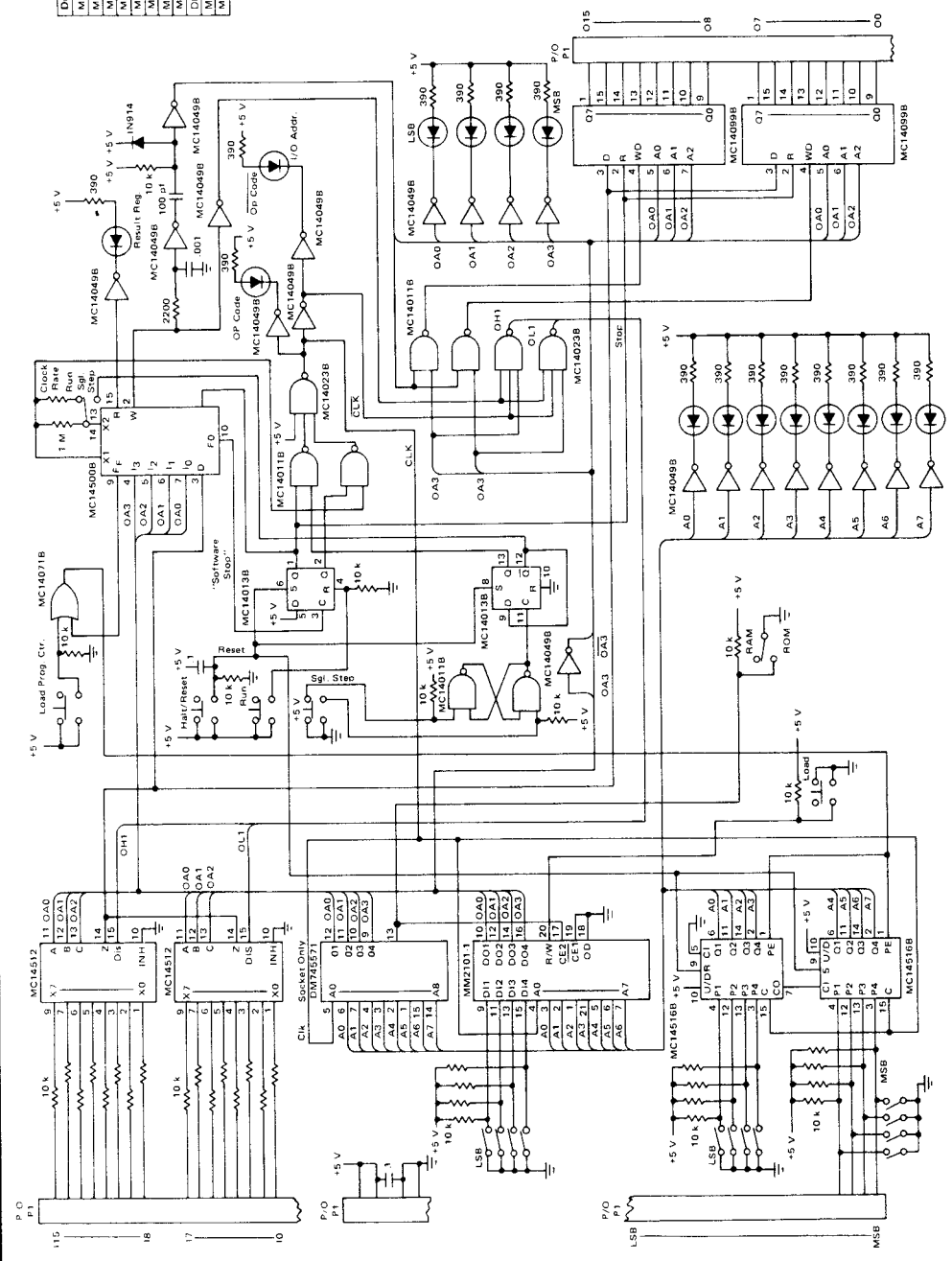


Figure 5.7 ICU Demonstration Unit Schematic

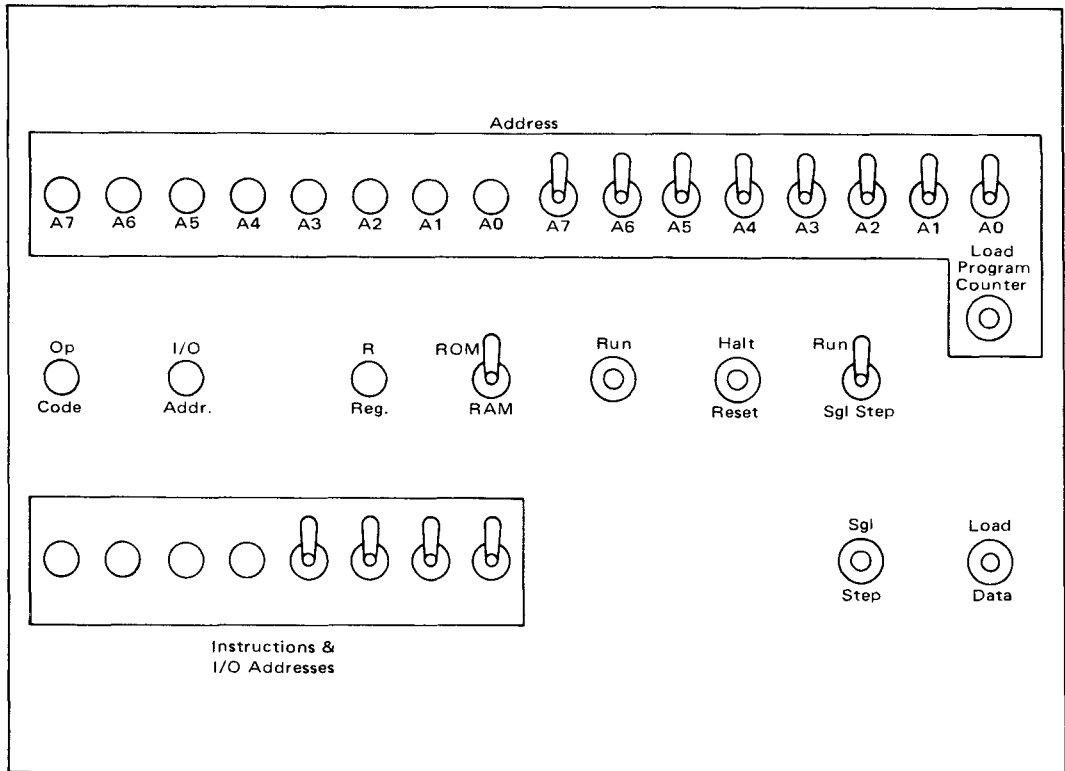


Figure 5.8 ICU Demonstration Unit Panel

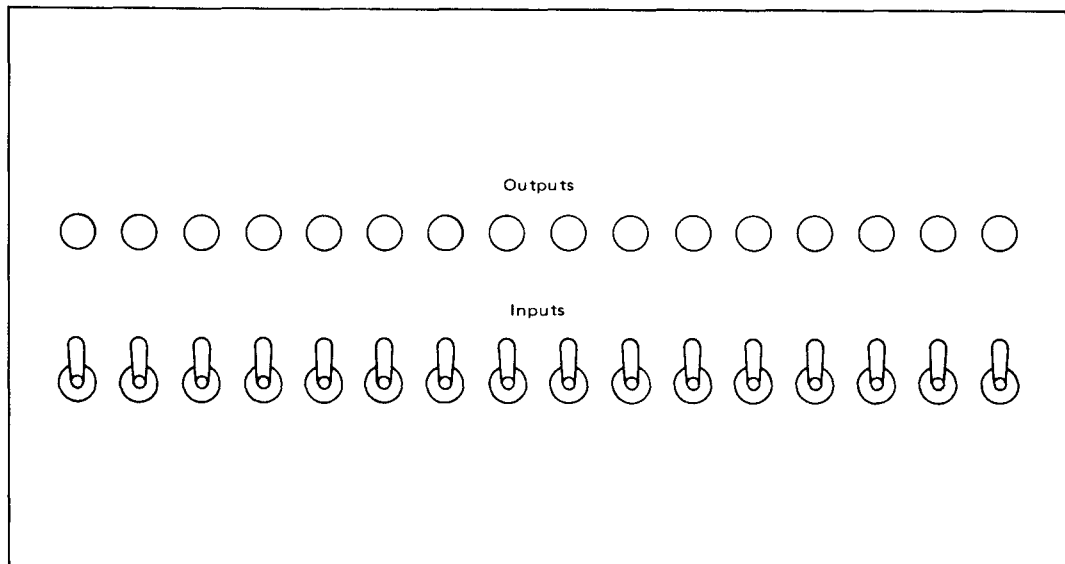


Figure 5.9 I/O Simulator

CHAPTER 6 TIMING, SIGNAL CONDITIONING, AND I/O CIRCUITS

The ICU can make use of a variety of circuits to do timing information gathering and distribution of information. This chapter is an assortment of such circuits.

Timing

Nearly all control tasks have a timing function. An important feature of the MC14500B system is the ease with which any number of timers and timing functions can be incorporated into a system.

In an ICU system, timing can be implemented with either software or hardware. Software timing requires the use of Incrementation or Decrementation routines, which are described in Chapter 14. Hardware timers can be quite simple; a variety of them will be described. They all tend to have one thing in common — an output is used to start a time interval, and an input to the ICU system is used to monitor the timer output so that the ICU will “know” when the interval has ended. A typical timer is shown in Figure 6.1.

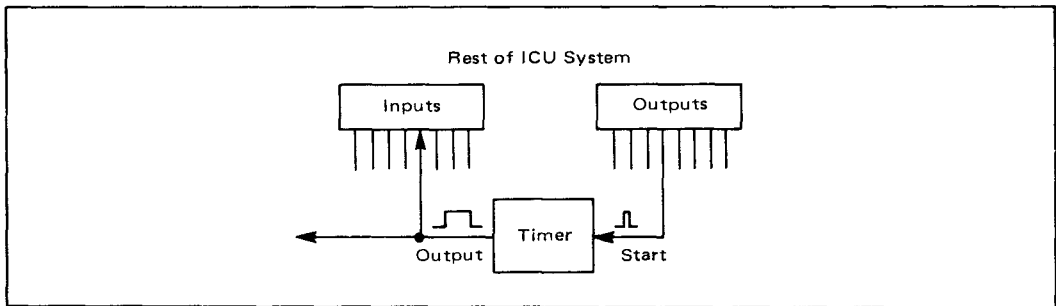


Figure 6.1 A Typical Hardware Timer

No mention has been made of the kind of timer used since a variety of choices are available: motor timers, clocks, timer delay relays, and both analog and digital timers. The following examples are in order of increasing complexity.

The code required to control and use the analog timer in Figure 6.2 is short and simple.

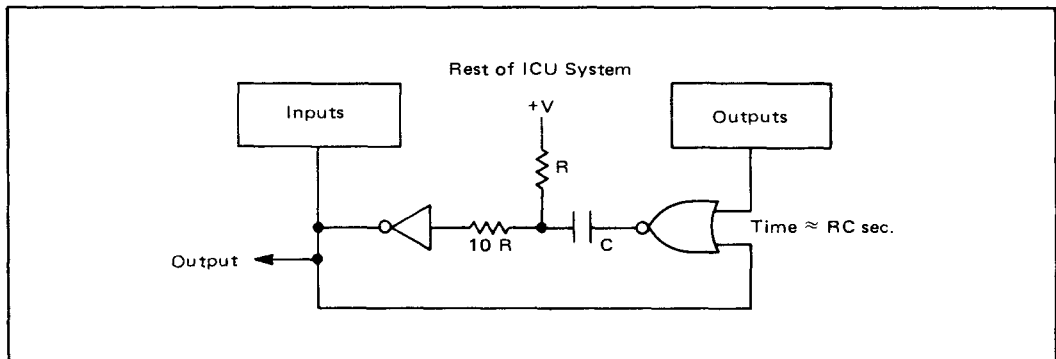


Figure 6.2 CMOS Monostable Timer

Assume that the time interval is to start whenever ‘‘A’’ and ‘‘B’’ are both high and the timer is quiescent. Only four instructions are required.

```
LD      A          LOADS THE A SIGNAL
AND     B          AND B WITH A
ANDC    INPUT     AND WITH INPUT, THE TIMER'S OUTPUT,
                  TO VERIFY THAT TIMER IS NOT RUNNING.

STO     OUTPUT    OUTPUT SIGNAL TO START THE TIMER
```

On the next pass through the program, the input signal will be at the 1 level and a 0 will be stored at the output, which ends the timer’s start pulse. Also notice that, although $A \cdot B$ is a simplistic starting condition, in actual practice, the condition for starting the timer might be very complex.

In industrial applications, simple timers are often inadequate for the actual task. Typically, one needs control time delays as well as interval timing. In Figure 6.3, a few of the commonly-used delay functions are depicted and have the following relationships:

Turn On Delay

Turn on U, X seconds after A goes high.

Turn Off Delay

Turn off V, Y seconds after B goes low.

Delay On — Delay Off

Turn on W, X seconds after C rises.

Turn off W, Z seconds after C falls.

The coding for the functions is as follows:

Turn-On Delay

```
X Start = A · U · TX
Store U = TX · U
assume IEN = OEN = 1
```

```
LD      A
ANDC    U
ANDC    TX
STO     X START
LD      TX
ANDC    U
OEN     R
STO     U
XNOR    R
OEN     R
```

Turn-Off Delay

```
Y Start = V · V · TY
V Store = V · TY
```

```
LD      V
AND     B
ANDC    TY
STO     Y START
LD      TY
AND     V
OEN     R
STO     V
XNOR    R
OEN     R
```

Delay-On/Delay-Off

This function simply chains the Turn-On Delay and Turn-Off Delay codes into a single routine.

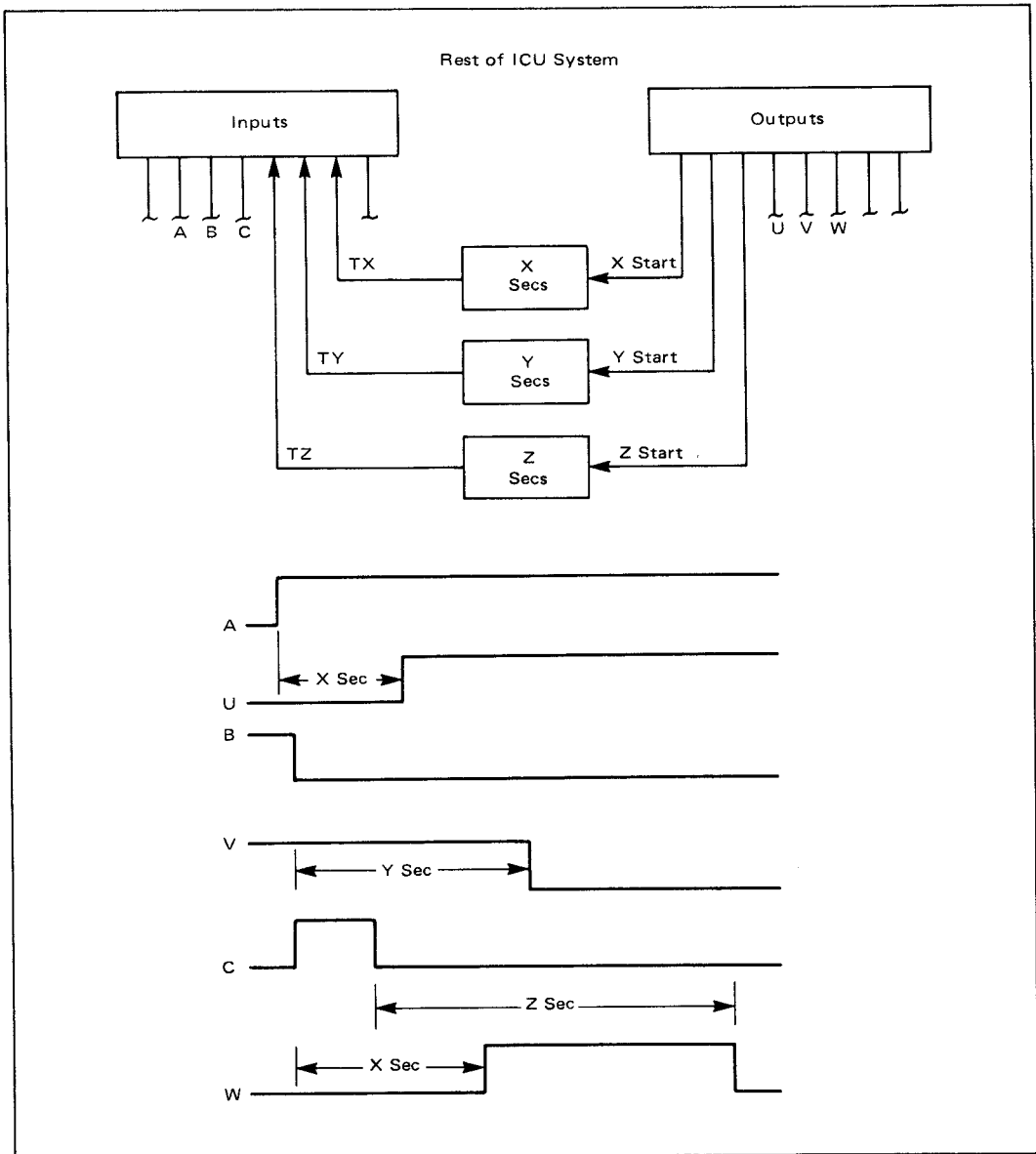


Figure 6.3 Complex Timing Waveforms

Digital Timing

A simple and straight forward method of digital timing is shown in Figure 6.4. A single four bit CMOS up/down counter is used, although any number of these devices may be cascaded to form counters for very long time intervals. The counter shown is used in the count down mode. At rest, the counter's reset line is held high. When a time interval is to be started, the reset signal is removed and a pulse is put on the counter's parallel enable pin, (PE), which loads the counter with the time set by the digital switches. From this point on, the circuit works like an analog timer, with its output taken from the CO pin. This pin will go to the high state during the timed interval and fall when the interval terminates. The ICU

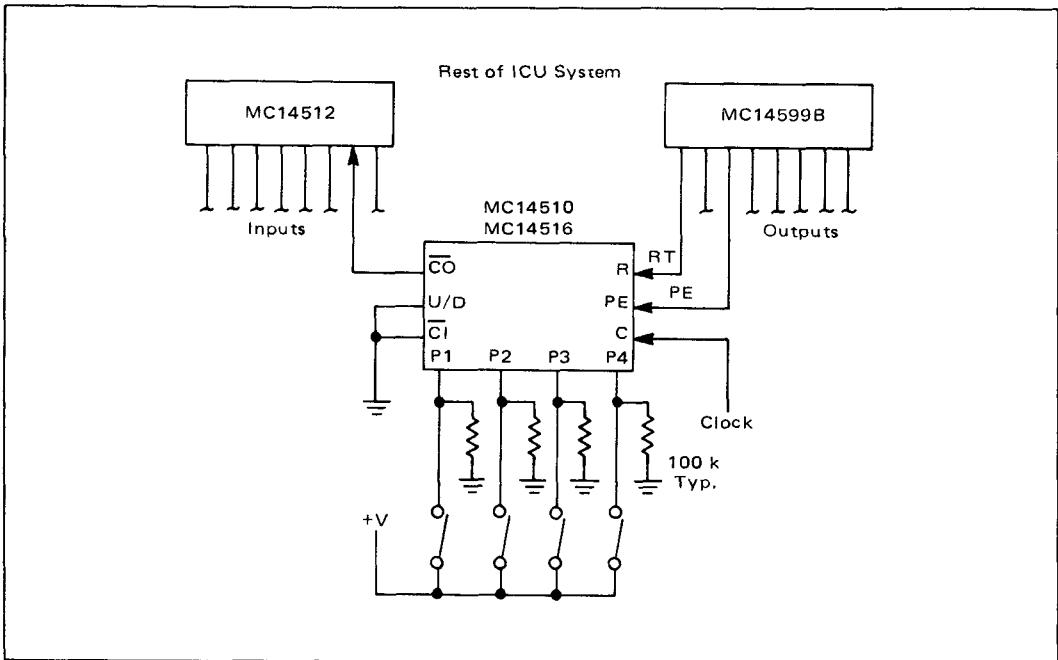


Figure 6.4 Simple Digital Timer

monitors the C0 signal and reapplies the reset signal when the interval has ended. A possible code sequence for this set of tasks is listed below.

Assume that a Flag "T" is set to 1 by previous code execution whenever the digital timer is required to operate. The timer will be started whenever the timer is at reset ("END" = 0) and $T = 1$. T will be set to 0 when the interval starts to insure that two intervals do not overlap.

START	LD	T	TEST START FLAG
	ANDC	END	TEST TIMER RESET
	OEN	RR	ENABLE OUTPUTS
	STOC	RESET	REMOVE TIMER RESET
	STO	LOAD	START LOAD PULSE
	STOC	LOAD	REMOVE LOAD PULSE
	STOC	T	CLEAR T
	ORC	RR	RR GETS 1 ($RR + RR = 1$)
FIN	OEN	RR	ENABLE OUTPUTS

Assume that the time interval should be started if $A \cdot B = 1$. The result will be used as Output Enable to control the start of the timing sequence.

LD	A	LOAD A
AND	B	RESULT IS NOW $A \cdot B$
OEN	R	OUTPUT ENABLED IF $A \cdot B = 1$
STOC	RT	REMOVE COUNTERS RESET
STO	PE	PULL PE HIGH
STOC	PE	RESTORE PE LOW

With the timer operating, the end of the interval is detected by finding CO signal low. The OEN should be restored for use by other routines. As follows, XNOR R will force a 1 into the Result Register, which is then used to load OEN.

XNOR	RR	FORCE RR = 1
OEN	RR	OEN = 1

Multiple Interval Timing

It is frequently necessary to time a number of different intervals in a controller, with no two intervals running simultaneously. This means that a number of different sources can be used to load a digital timer, as shown in Figure 6.5. The switches shown are coded switches. The isolation diodes are required in order to “OR” the switch outputs into the data inputs.

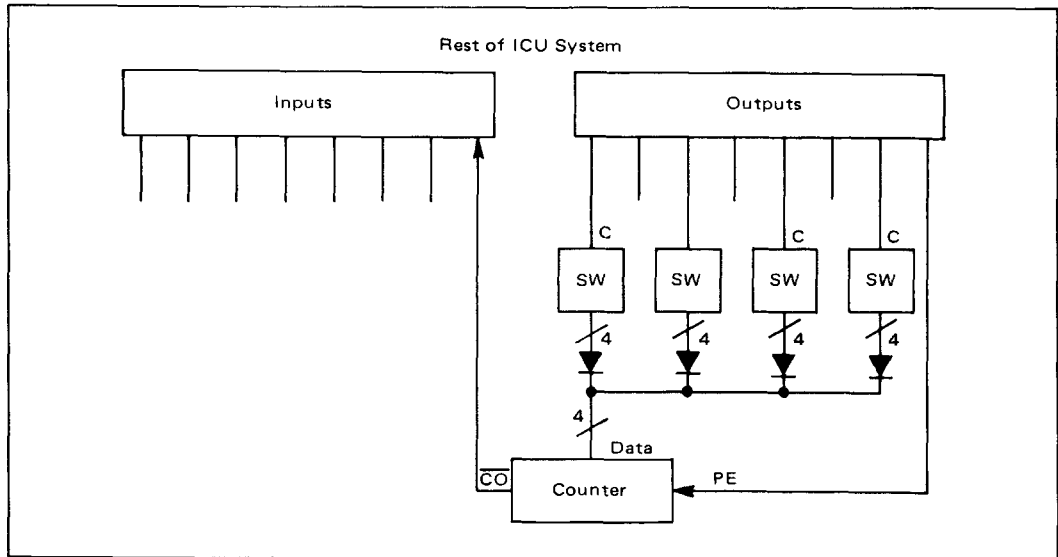


Figure 6.5 Multiple Interval Timer

Control By Time-Of-Day

Many users want to control loads by time-of-day and/or day-of-week. An example is a retail store which is open two nights a week and wants to control lighting, heating, air conditioning and a security system in conformity with the store's schedule. The store has a "morning routine" which turns on air conditioning, lights and removes the security system at the proper times. The "morning routine" does not operate on Sunday when the store will be closed.

Such applications require a clock system which has a coded output that will be read by an ICU routine. When the ICU determines that clock time matches a key stored in the ICU's memory, a single pass will be made through a routine which sets or clears output bits as programmed. At the end of the single pass, a flag is set, thereby telling the ICU that the task is completed. This flag will be cleared as part of the "housekeeping" when a subsequent routine is started. The flag condition is part of the key for starting the routine.

The whole concept for a time-of-day control is one of enabling a block of code when certain conditions have been met. An example of such a routine is shown in Figure 6.6.

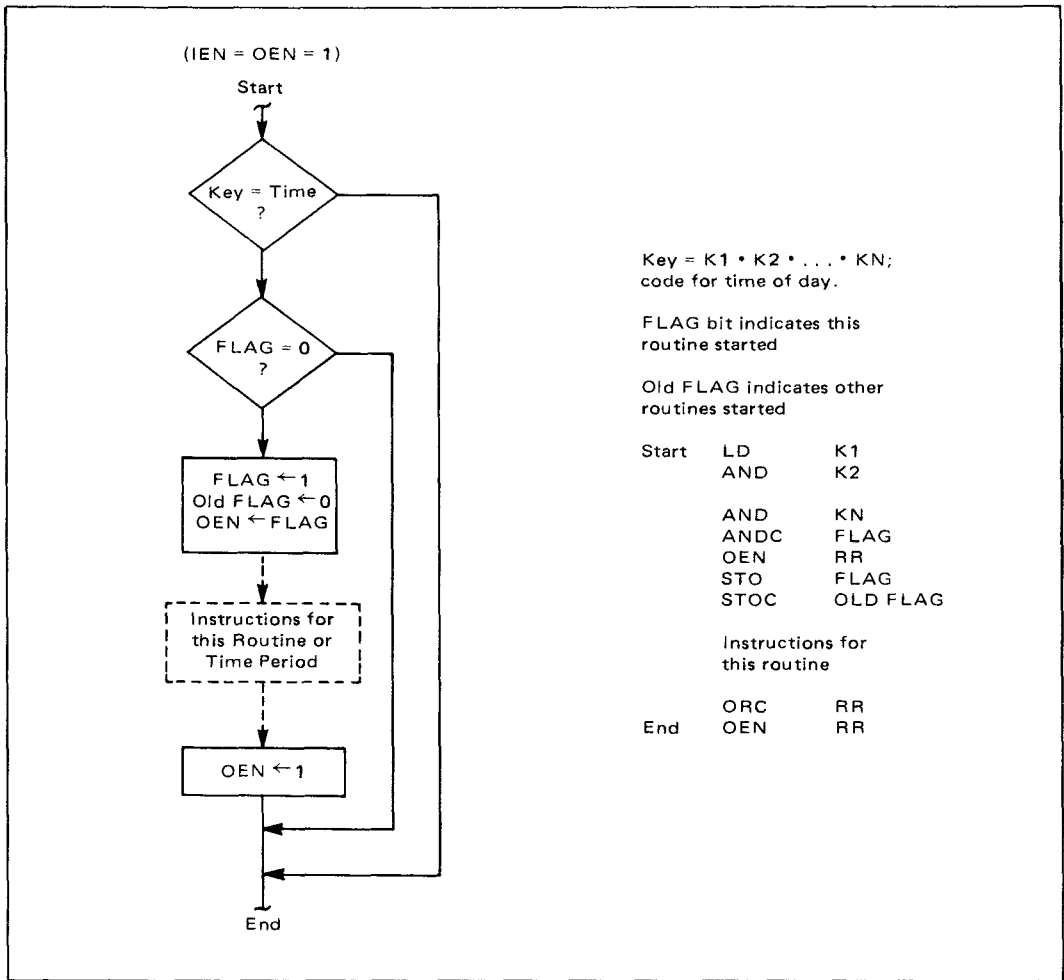


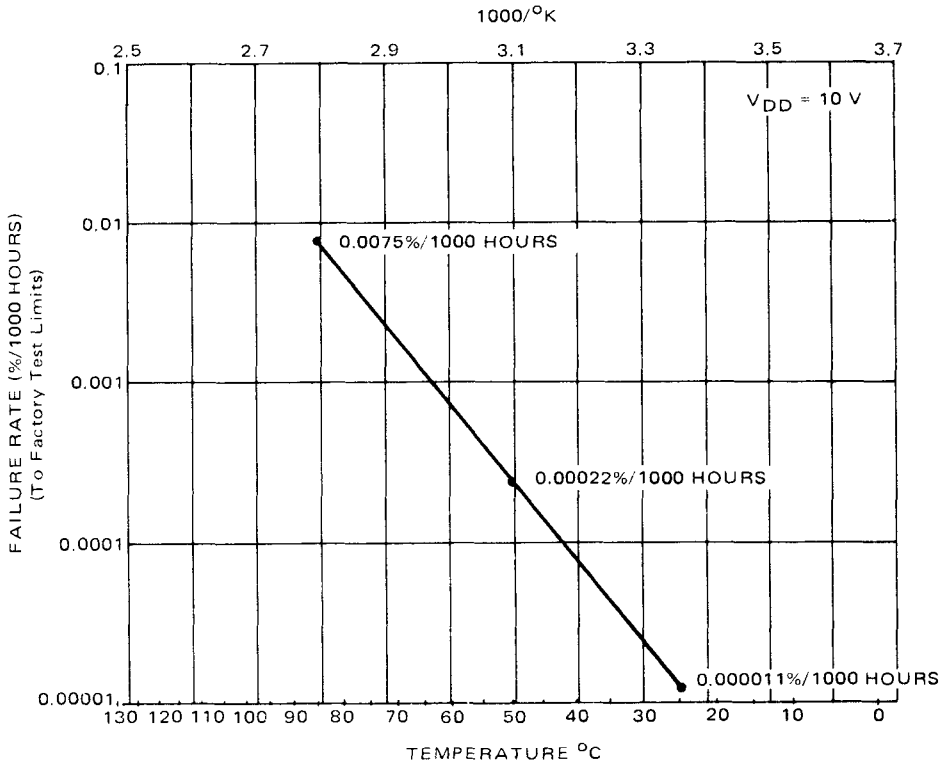
Figure 6.6 Time-of-Day Routine

McMOS RELIABILITY AND DEVICE HANDLING PROCEDURES

Confident use of a family of components requires assurance of the reliability of a component under normal operating conditions and the ability of the device to survive abnormal conditions that may occur. CMOS, and specifically Motorola McMOS, has achieved the high confidence level for equipment usage that has been enjoyed by many other semiconductor products.

RELIABILITY

Figure 6.7 shows the composite failure rate of commercial ceramic and plastic packaged McMOS integrated circuits as a function of temperature. Note that CMOS devices dissipate little power and work nominally close to ambient temperature. This feature adds to CMOS reliability. The data shown represent over 40 million equivalent device hours and give failure rates to the factory set of test limits. This standard of failure is more severe than a catastrophic failure rate.



This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit. For proper operation it is recommended that V_{in} and V_{out} be constrained to the range $V_{SS} \leq (V_{in} \text{ or } V_{out}) \leq V_{DD}$.

Unused inputs must always be tied to an appropriate logic voltage level (e.g., either V_{SS} or V_{DD}).

Figure 6.7 – Failure Rate of Commercial CMOS Integrated Circuits (Ceramic and Plastic Packaged Devices)

HANDLING PRECAUTIONS

All CMOS devices have diode input protection against adverse electrical environments such as static discharge. The following statement is included on each data sheet:

Unfortunately, there can be severe electrical environments during the process of handling. For example, static voltages generated by a person walking across a common waxed floor have been measured in the 4 to 15 kV range (depending on humidity, surface conditions, etc.). These static voltages are potentially disastrous when discharged into a CMOS input considering the energy stored in the capacity (≈ 300 pF) of the human body at these voltage levels.

Present CMOS gate protection structures can generally protect against overvoltages. This is usually sufficient except in the severe cases. Following are some suggested handling procedures for CMOS devices, many of which apply to most semiconductor devices.

1. All MOS devices should be stored or transported in materials that are somewhat conductive. MOS devices must not be inserted into conventional plastic “snow” or plastic trays.
2. All MOS devices should be placed on a grounded bench surface and operators should ground themselves prior to handling devices, since a worker can be statically charged with respect to the bench surface.
3. Nylon clothing should not be worn while handling MOS circuits.
4. Do not insert or remove MOS devices from test sockets with power applied. Check all power supplies to be used for testing MOS devices to be certain there are no voltage transients present.
5. When lead straightening or hand soldering is necessary, provide ground straps for the apparatus used.
6. Do not exceed the maximum electrical voltage ratings specified by the data sheet.
7. Double check test equipment setup for proper polarity of voltage before conducting parametric or functional testing.
8. Cold chambers using CO₂ for cooling should be equipped with baffles, and devices must be contained on or in conductive material.
9. All unused device inputs should be connected to V_{DD} or V_{SS}.

When external connections to a PC board address only an input to a CMOS integrated circuit, it is recommended that a resistance 10 kΩ or greater be used in series with the input. This resistor will limit accidental damage if the PC board is removed and wiped across plastic, nylon carpet or inserted into statically charged plastic “snow”.

The input protection circuit, while adding some delay time, provides protection by clamping positive and negative potentials to V_{DD} and V_{SS}, respectively. Figure 6.8 shows the internal circuitry for the diode-resistor protection.

The input protection circuit consists of a series isolation resistor R_S, whose typical value is 1.5 kΩ, and diodes D1 and D2, which clamp the input voltages between the power supply pins V_{DD} and V_{SS}. Diode D3 is a distributed structure resulting from the diffusion fabrication of R_S.

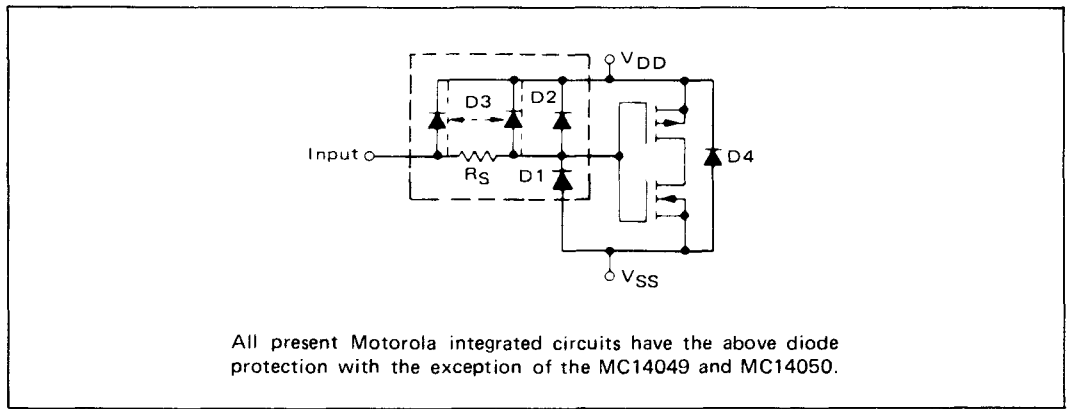


Figure 6.8 – Schematic Diagram, Diode-Resistor Input Protection

Isolating Inputs

Many applications require electrical isolation between a signal source and the control logic. There are four usual ways of doing so: Optical isolators, transformer and capacitive coupling and relay contacts. The relay contacts are simply used as a switch closure to the logic supply or to ground. The other schemes require more discussion.

Optical Isolated Inputs

Figure 6.9 shows two typical examples of opto isolation.

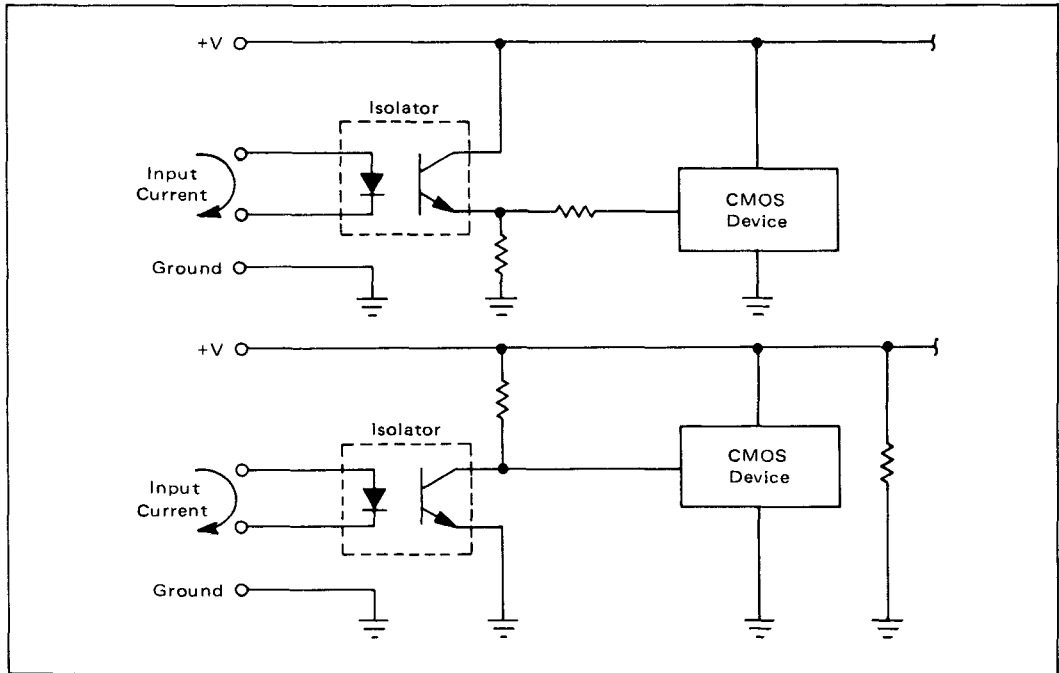


Figure 6.9 Optically Isolated Inputs

Transformer Coupled Inputs

Transformer coupling is used, most often, for detecting the phase or amplitude of a power line derived signal. Figure 6.10 shows a voltage level-sensing scheme. Figure 6.11 has a connection for detecting the phase of an AC signal.

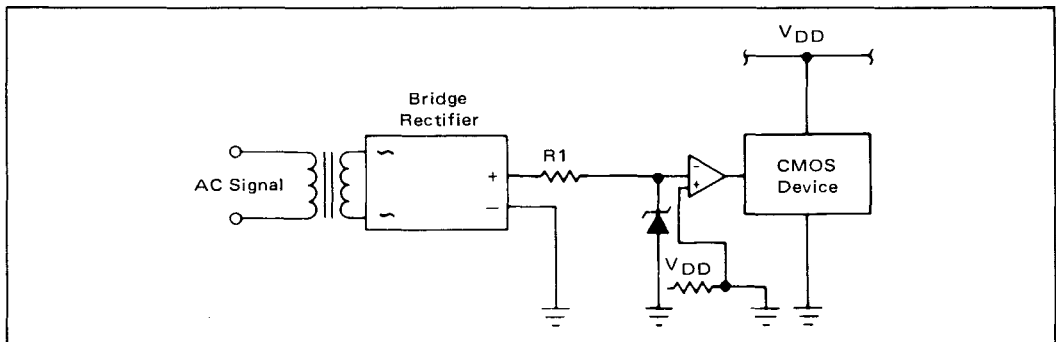


Figure 6.10 Amplitude Detection of AC Signal

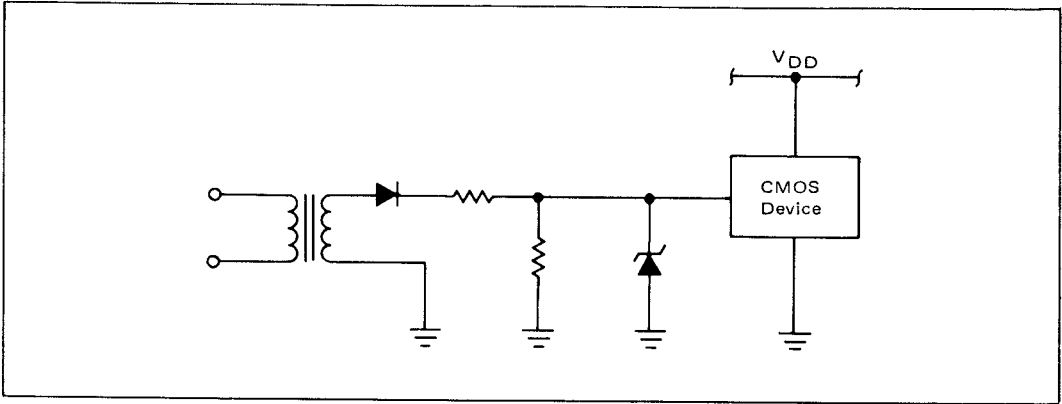


Figure 6.11 Phase Detection of AC Signals

Capacitively Coupled AC Signals

For convenience or economy, a designer can sometimes replace a transformer with a capacitor. Figure 6.12 shows the way this might be done. The capacitive divider technique might be preferred for true ratioing of voltages up to the zener value.

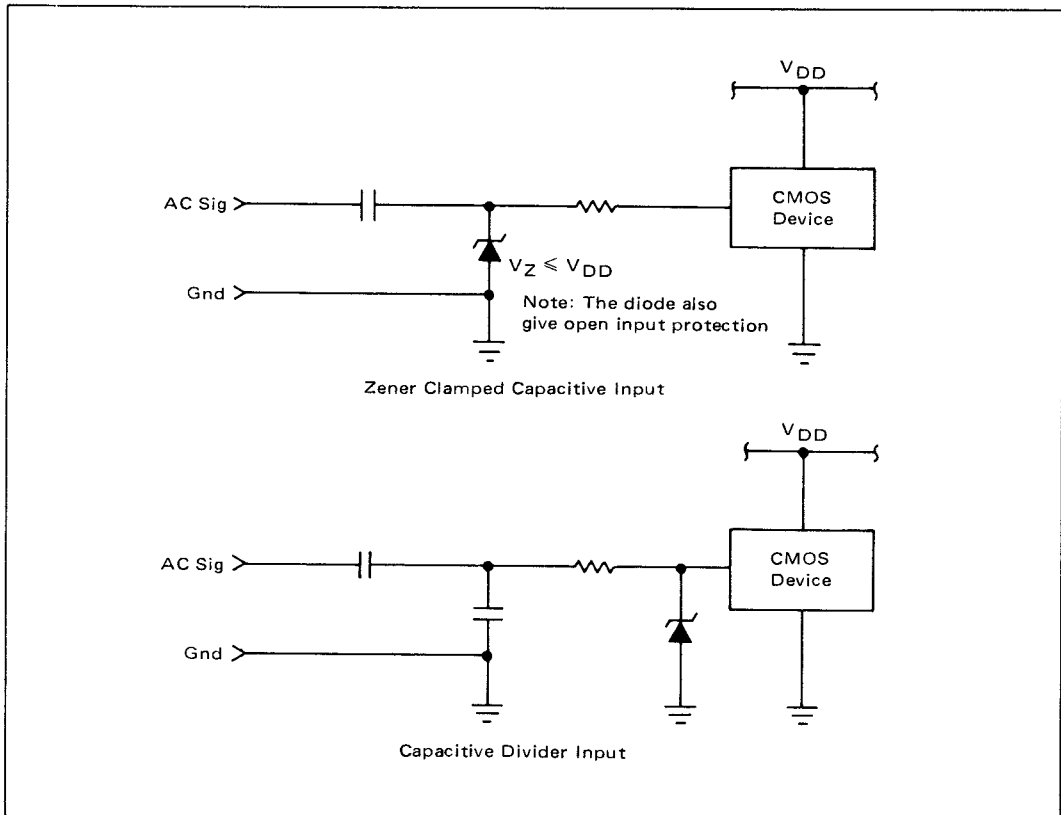


Figure 6.12 Capacitive Input Schemes

Sampling Inputs

It is possible that a signal might change status during a loop of a program. If an initial sample of a signal and a later sample of the same signal were of different values, some undesired result might be obtained. The simple avoidance of this problem is to sample all the variables at the beginning of a program and to store them in temporary stores. Whenever the values are needed later in the program, the temporary store contents should be used in lieu of the input signal. The rule is:

Only sample an input signal once in any program.

OUTPUT CONDITIONING

High Current

Figure 6.13 shows an interface between a low impedance load and an MC14599B ICU output device. The MC1413 interface ports will drive 300 mA loads when saturated, with V_{CC} up to 50V. Inrush currents of 600 mA may be handled by the MC1413, which allows incandescent lamp loads of 300 mA to be driven without derating for inrush. The internal diodes are useful for damping inductive load switching transients.

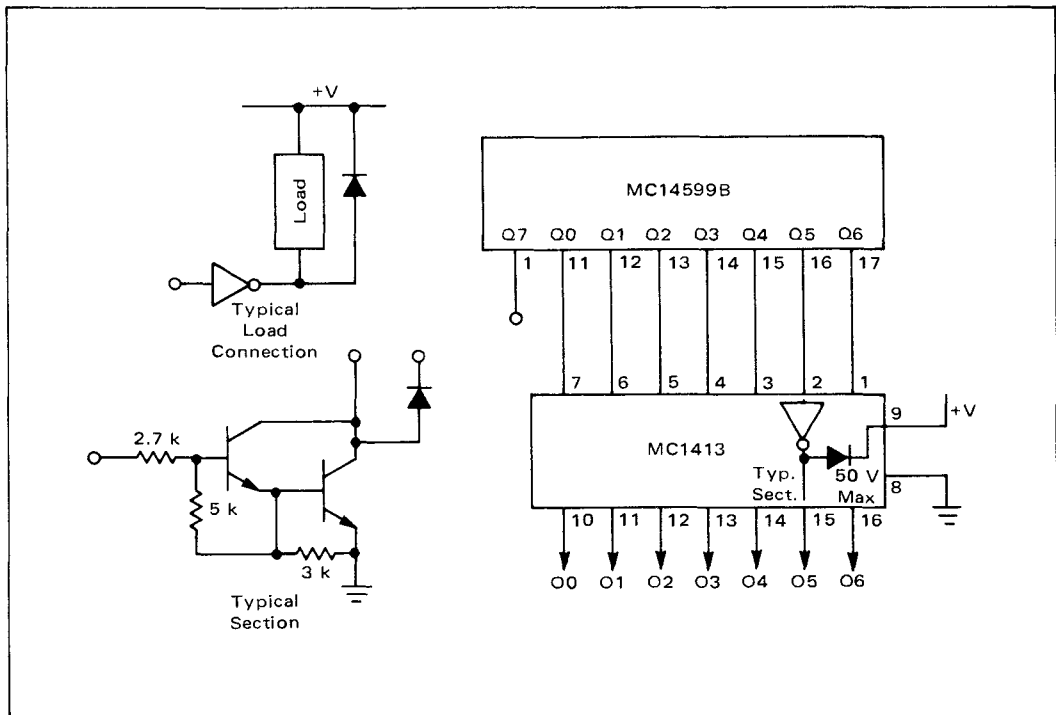


Figure 6.13 High Current Output Buffers

Relay Driving

A typical interface to a machine must often be made using electromechanical relay contacts for loading switching. This occurs because the original wiring of the machine was designed before electronic control was contemplated. As shown on Figure 6.14, the MC1413 can also serve as a relay coil driver.

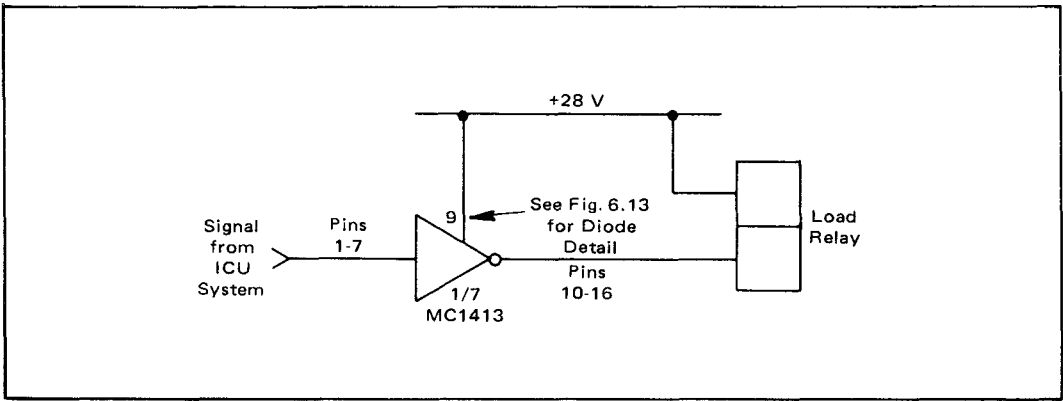


Figure 6.14 Driving Relays

LED Driving

Driving LED's or opto-couplers is much like driving a relay load, except that an external current limiting resistor is used to control the current through the LED, coupler or solid-state-switch. See Figure 6.15. High-efficiency LEDs can be driven directly from CMOS circuits.

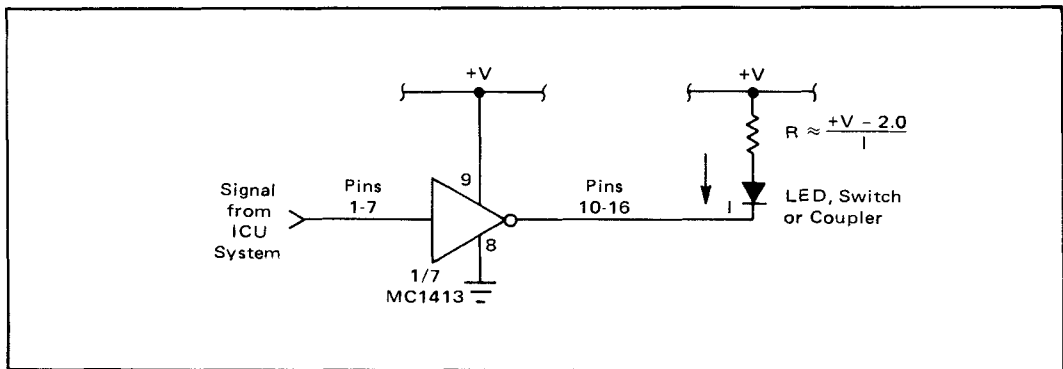


Figure 6.15 Driving LED Loads

Driving Thyristors (SCRs & Triacs)

There are many different means for driving thyristors. One of the simplest and most reliable will be shown here. The method uses pulse transformers and is called the "picket fence" technique. The name is due to the scope trace of a pulse transformer's secondary voltage when the primary is pulsed many times each millisecond.

Given an adequate supply voltage and a resistive load, a triac or SCR will usually turn on when driven by a single gate pulse. However, combinations of low voltage and reactive loads can keep a device from reaching the on (latched), state. The picket fence approach is to supply a train of gate pulses so that the SCR or triac will latch on during the first pulse time when sustaining conditions are met. This allows the devices to turn on as close to zero crossings of load voltage as possible. Additionally, the technique is quite economical.

There are three key parameters to satisfy: Minimum pulse width, maximum pulse width and gate current requirements. Additionally, a check of the insulation specifications of pulse transformers is in order.

Minimum pulse width and thyristor gate current numbers are obtained from device data sheets. After defining minimum pulse width, a rule-of-thumb is to exceed the minimum, say by a factor of two. The maximum pulse width requirement is dependent upon the volt-second-product (VSP) of the transformer. The importance of VSP is to insure that the pulse transformer will not saturate, the driver will not burn up, and a current limiting resistor need not be used.

The maximum pulse width as a function of supply voltage is:

$$PW_{MAX} = \frac{VSP}{V_{supply}} \text{ seconds}$$

An ICU system that has a clock period of four times PW_{MAX} can pulse a triac driver by storing a "1" in the driver and removing it three clock periods later:

```
(RR = 1, OEN = 1)
STO          TRIAC
NOP          DON'T CARE
NOP          DON'T CARE
NOP          DON'T CARE
STOC        TRIAC
```

This code would need to be repeated many times; to conserve memory space, a subroutine could be called over and over, to make a picket fence gate drive effective. As shown on Figure 6.1, the output of a pulse oscillator is "ANDed" with the ICU output.

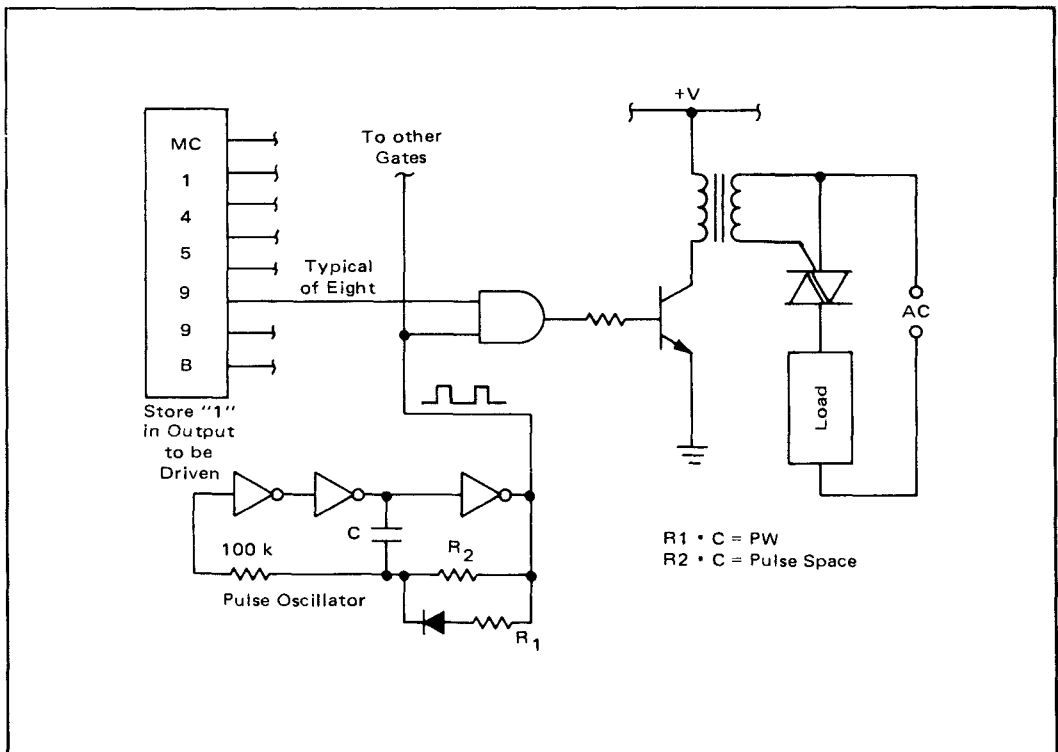


Figure 6.16 Picket Fence Triac Firing

Adding Hysteresis to Inputs by Using Outputs

A simple, but useful, trick in an ICU system is to add hysteresis to an input signal, under program control. In Figure 6.17, the input pin on an MC14512 device can be an input and an output signal. The ICU reads the input signal, stores the result in the output (positive feedback) and then reads the input again.

LD	INPUT
STO	OUTPUT
LD	INPUT

Gang Transfer of Outputs

While looping through a program, the ICU addresses each output bit in a sequential manner. If a particular controller configuration requires that all output bits be available simultaneously, then the output values can be stored in additional latches or flip-flops. A simple routine can be added to the program to strobe the ICU outputs into the latches. The latch outputs are then available to the rest of the system. The strobe, or pulse-generating routine is:

START	ORC	RR	FORCE RR TO 1
	STO	PULSE	PULSE GOES HIGH
STOP	STOC	PULSE	PULSE GOES LOW

End of Program

Often, one wants to return to the top of the program immediately after completing the last written instruction. Usually, there is a pulse output on the MC14500B device that is not being used in the system. For example, if FLAG 0 is not being used elsewhere in the system, then it can simply be OR'd with the program counter's reset signal.

In other words, the PC will jump to 0 whenever a FLAG 0 instruction (NOP 0) is executed. The NOP commands do not need an address (operand); hence, the ROM's operand field can be AND'ed with NOP's (FLAG pulses) to generate user defined functions.

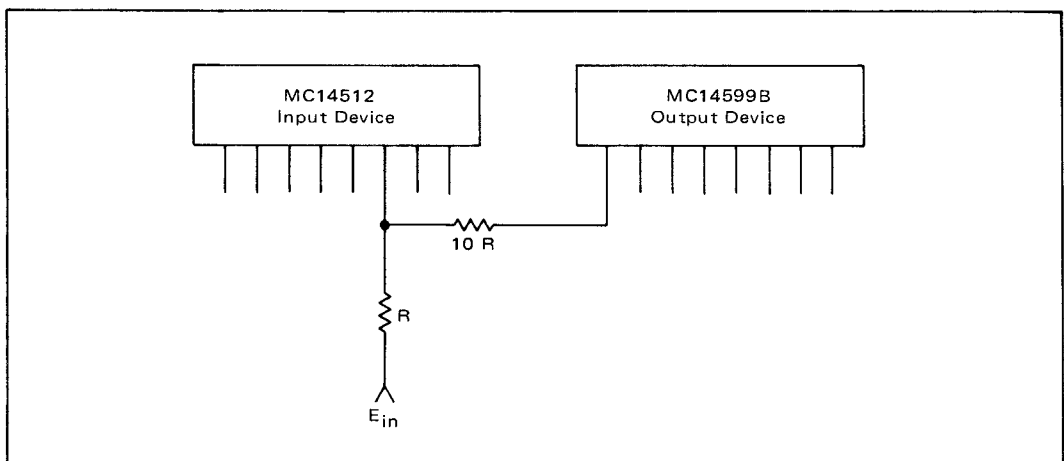


Figure 6.17 Adding Input Signal Hysteresis

CHAPTER 7 OEN AND THE IF-THEN STRUCTURE

The OEN Instruction

The Output Enable (OEN) instruction is the most unusual and powerful instruction in the MC14500B. This is the instruction that makes the looping, (as opposed to jumping), program flow powerful and practical.

All the concepts required to exploit the OEN's power are shown in Figure 7.1. The four important ideas are:

1. If the OEN register holds a 1, the ICU can write to output or memory devices. If OEN holds a 0, the WRITE pulse cannot be generated and no output device or memory content will change. OEN thus controls whether or not the ICU system is "working" at any moment.
2. Any input signal can be directly loaded into the OEN register. An input can be wired to the supply or ground to give an addressable 1 or 0. The Result Register output can also be used as an input signal.
3. The physical connection allowing the Result Register to be addressed as an input is so useful it should always be utilized.
4. A block of instructions can be used that calculates whether or not a subsequent block of code is to be executed. As this result resides in the Result Register, the Output Enable Register should be loaded with OEN RR.

IF-THEN (OEN Step 1)

In this section, the Output Enable instruction (OEN) will be used to simplify the logic controlling the program's execution. The title IF-THEN implies exactly what we want to do. If a condition is satisfied THEN a block of code will be enabled. If not, the code should be disabled (ignored) so it cannot change the state of any output or internally stored bits.

For example, if overtemperature switch OTS is closed (= 1), sound horn H, turn off oven power OP, and turn on oven temperature light OTL. The ICU routine is as follows:

1	START	LD	OTS	LOAD SWITCH STATE
2		OEN	RR	ENABLE OUTPUTS IF 1 IN RESULT REGISTER
3		STO	H	TURN ON HORN
4		STOC	OP	TURN OFF POWER
5		STO	OTL	TURN ON LIGHT
6		ORC	RR	FORCE RR TO 1 ($RR + \overline{RR} = 1$)
7	END	OEN	RR	ENABLE OUTPUTS

The first two statements *disabled* the outputs if the overtemperature switch was not closed. Two statements are used, one to load the state of OTS into the Result Register, the other to enable the output or WRITE signal.

This is an example of a conditional program. The program can affect outputs *only* when the OEN register has a stored 1. Otherwise, nothing is changed by the ICU's execution of the seven instructions. The same number of clock cycles are used in either case. This is an

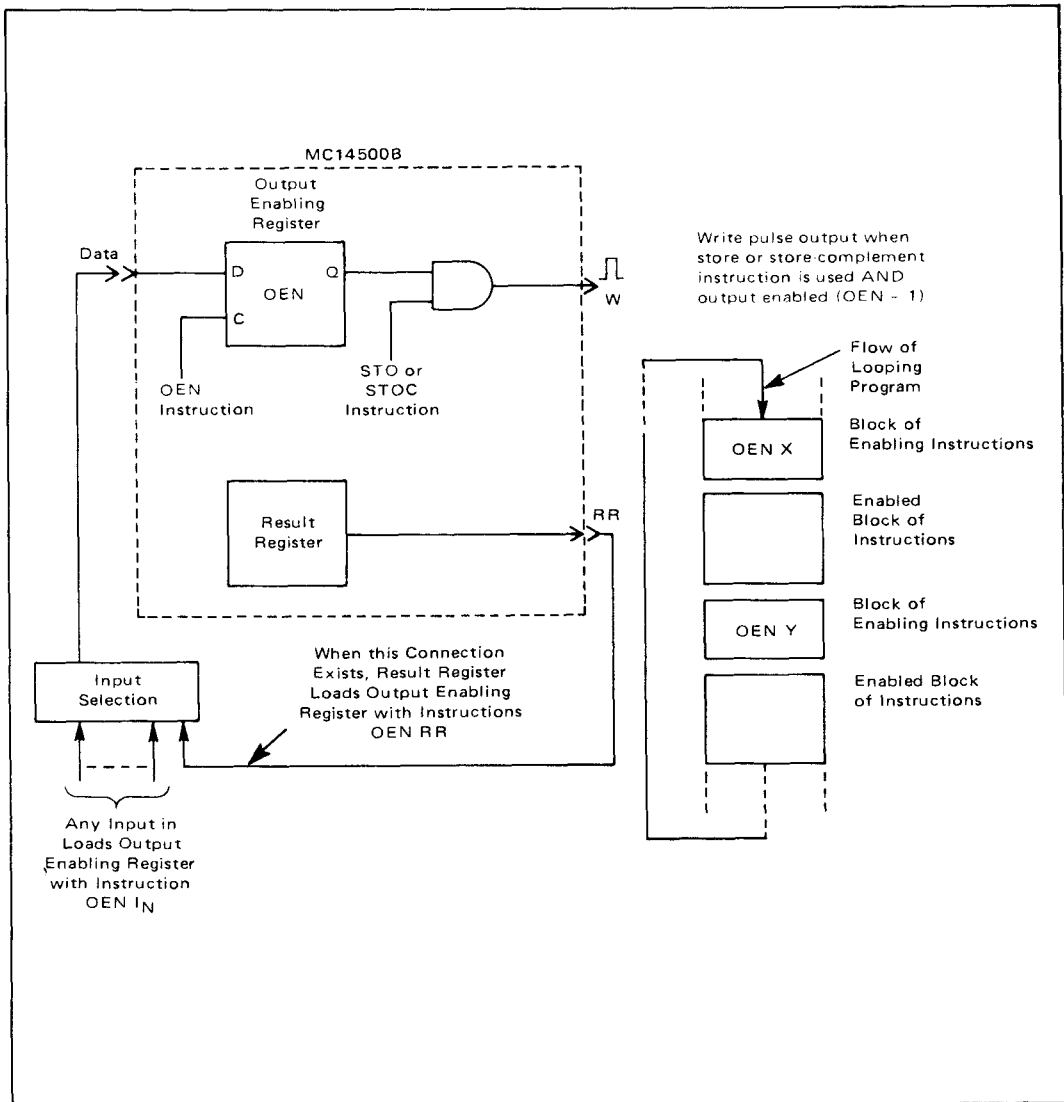


Figure 7.1 Output Enable Concepts

example of an IF-THEN block of code. The last two statements re-enable OEN for use by other blocks or sections of code. Figure 7.2 shows a flow chart representation of this IF-THEN block. The instructions that are executed in each block are written beside the blocks.

It is important to notice that when the IF test fails, *nothing* happens. This distinguishes IF-THEN blocks from other code or the flow chart structures we will examine in other chapters.

To Review: IF-THEN code blocks ask a question. If the question is answered yes (OEN = 1), the code following the question is enabled and the programs section is enabled. Otherwise, the code following is not workable because the WRITE pulse is not produced by the ICU when OEN = 0.

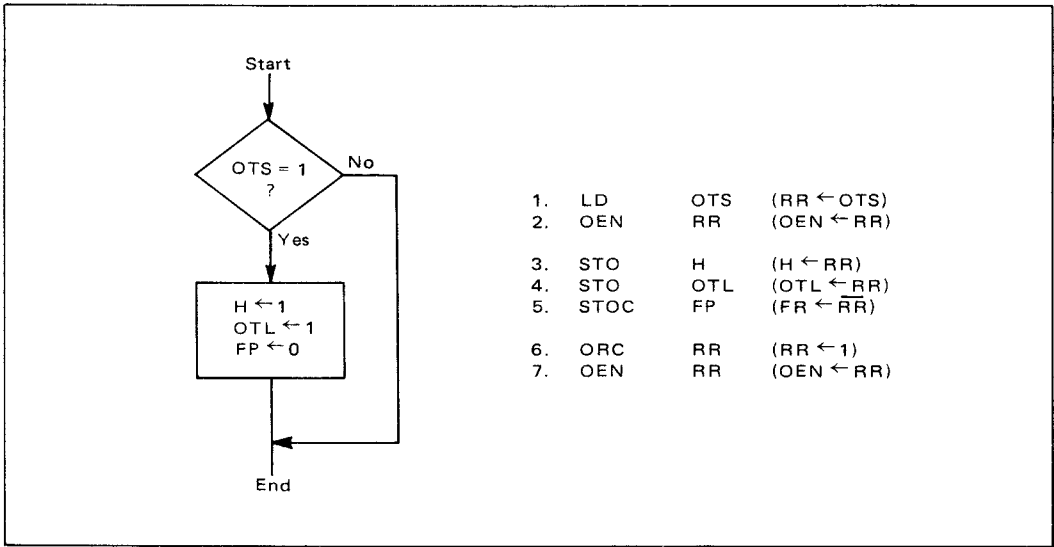


Figure 7.2 An If-Then Program Block

There is no restriction on the structure of the block of code or instructions enabled by the IF question in an IF-THEN structure. The block can contain other IF-THEN structures, as shown in Figure 7.3, or any of the other two program structures to be described in the next two chapters.

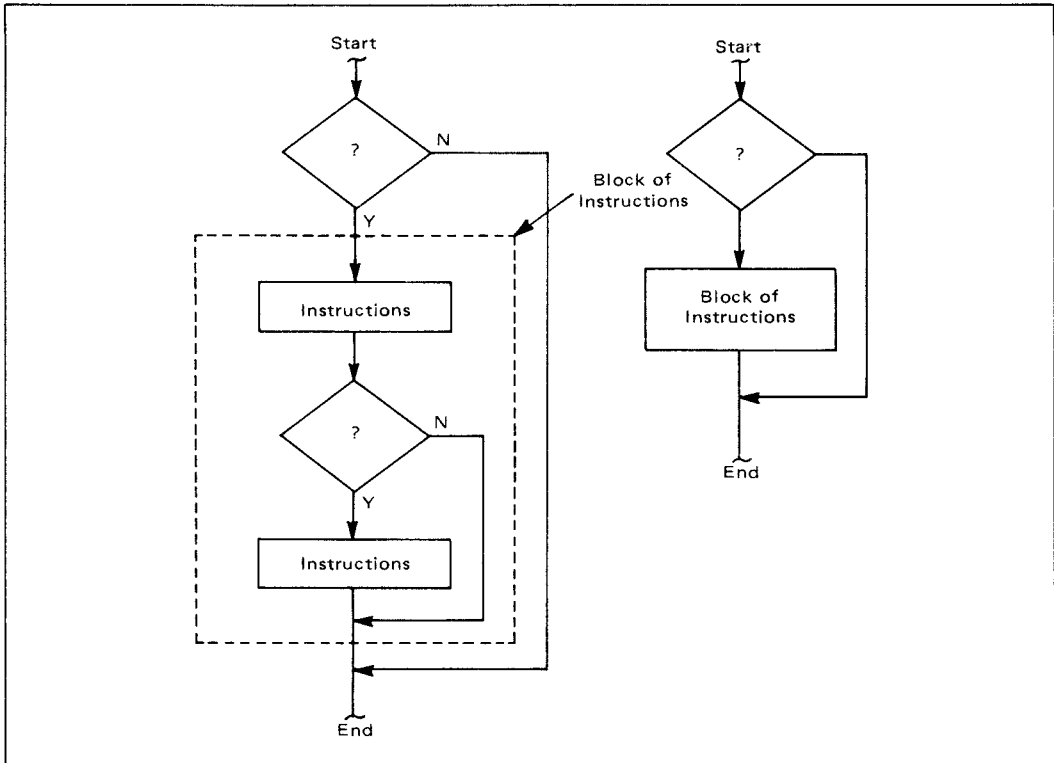


Figure 7.3 The Instruction Block May Be Complex

CHAPTER 8 THE IF-THEN-ELSE STRUCTURE (OEN STEP 2)

In the IF-THEN arrangement of Chapter 7, we saw that an action could be taken if a condition was satisfied. For example, if the limit switch is closed, turn off the motor. *There is no statement about what is to happen if the switch is not closed.* The IF-THEN-ELSE has the alternate action instructions not provided for in the IF-THEN structure.

The IF-THEN-ELSE structure is shown in Figure 8.1. A question is asked, if the answer is “no,” block B’s instructions are enabled and executed. We can see now that if block B contains no instructions, we once again have the simple IF-THEN structure. Once again, the output enable OEN is used. The “A” block is enabled by loading the OEN register from the Result Register (assuming $RR = 1$). To enable B, the complement of RR is stored, to be recalled later to either enable or not enable B. Thus the IF-THEN-ELSE sequence is as follows:

1. Resolve the enabling condition.
2. Store the *complement* of the result in some temporary location (“TEMP”).
3. Load the Output Enable Register OEN from the Result Register RR.
4. Do the A Block.
5. Load the Output Enable Register from “TEMP.”
6. Do the B Block.
7. Restore the OEN’s condition to enable ($OEN = 1$) to allow subsequent code to be enabled.

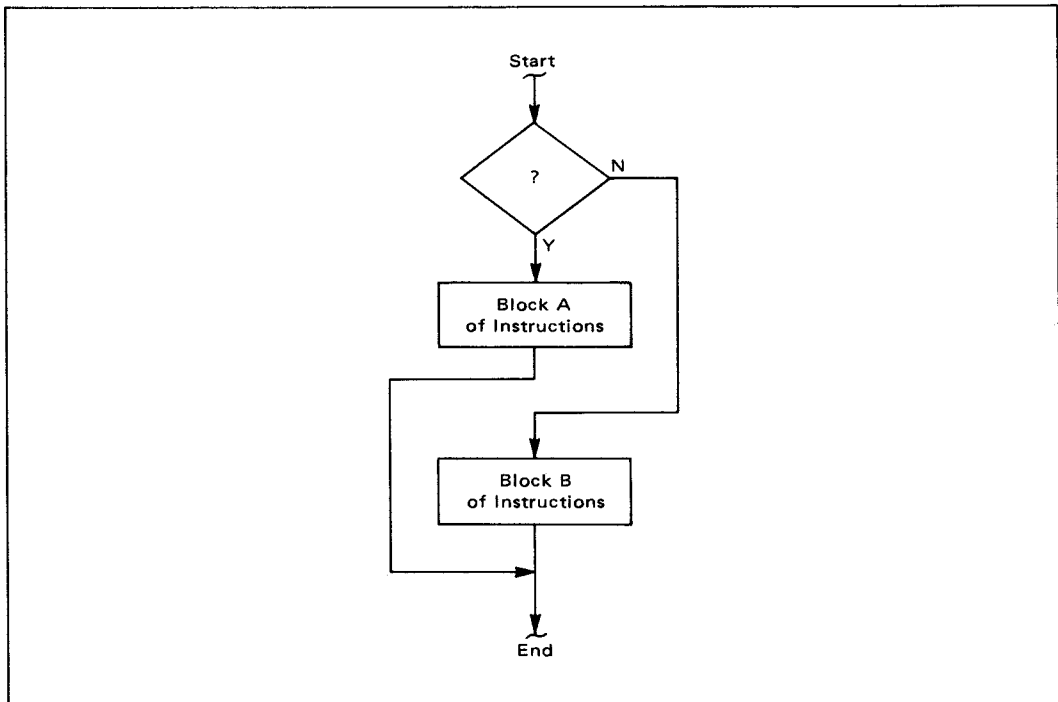


Figure 8.1 IF-THEN-ELSE

An example follows:

A simple IF-THEN-ELSE usage is to “turn on” a load if a condition exists, and to “turn off” the load otherwise. Such a function can be directly done, without the IF-THEN-ELSE structure. However, it is the enabling logic that is to be illustrated, not the control function.

The example is illustrated in Figure 8.2. The motor M is to run if the A and B contacts are both closed. Otherwise, the motor is not to run. If $A \cdot B = 1$, then $M = 1$, else $M = 0$. To start the routine, assume $IEN = 1$ (input enabled).

START	1A.	LD	A	$RR \leftarrow A$
	1B.	ANDC	B	$RR \leftarrow A \cdot \overline{B}$
	2.	STOC	TEMP	$TEMP = A \cdot \overline{B} = \overline{A} + B$
NOTICE: THE RESULT REGISTER STILL CONTAINS THE ENABLE CONDITION AND ITS COMPLEMENT ENABLE IS IN “TEMP”				
	3.	OEN	RR	ENABLE “RUN” CODE
	4.	STO	M	THE MOTOR RUNS
	5.	OEN	TEMP	$OEN \leftarrow TEMP$
	6.	STO	M	THE MOTOR STOPS
	7A.	ORC	RR	$RR \leftarrow RR + RR = 1$
	7B.	OEN	RR	$OEN \leftarrow 1$

In this example, the executable blocks consisted of single store instructions, which took advantage of the fact that the Result Register RR contained a 1 in the first “block” if the motor was to run, and a 0 in the second “block” if the motor was to stop.

LD	A	$RR \leftarrow A$
ANDC	B	$RR \leftarrow A \cdot \overline{B}$
STO	M	$M \leftarrow A \cdot \overline{B}$

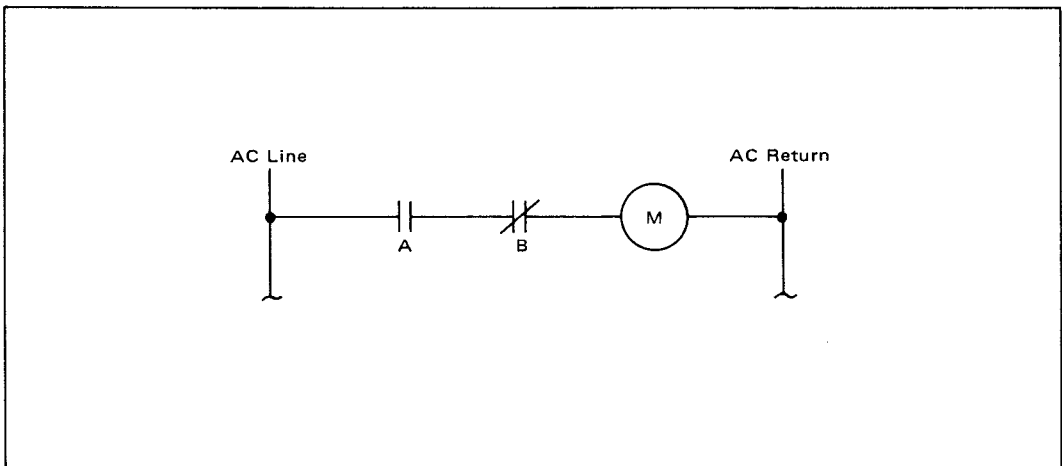


Figure 8.2 Function for IF-THEN-ELSE

It would have been six instructions briefer, but our example would be lost. The extra statements will allow us to write very complex programs in a straight forward and organized fashion. Notice, in Figure 8.3, that a "block" of instructions in our IF-THEN-ELSE structure could contain other IF-THEN or IF-THEN-ELSE structures. In that case, we would say the structures were "nested".

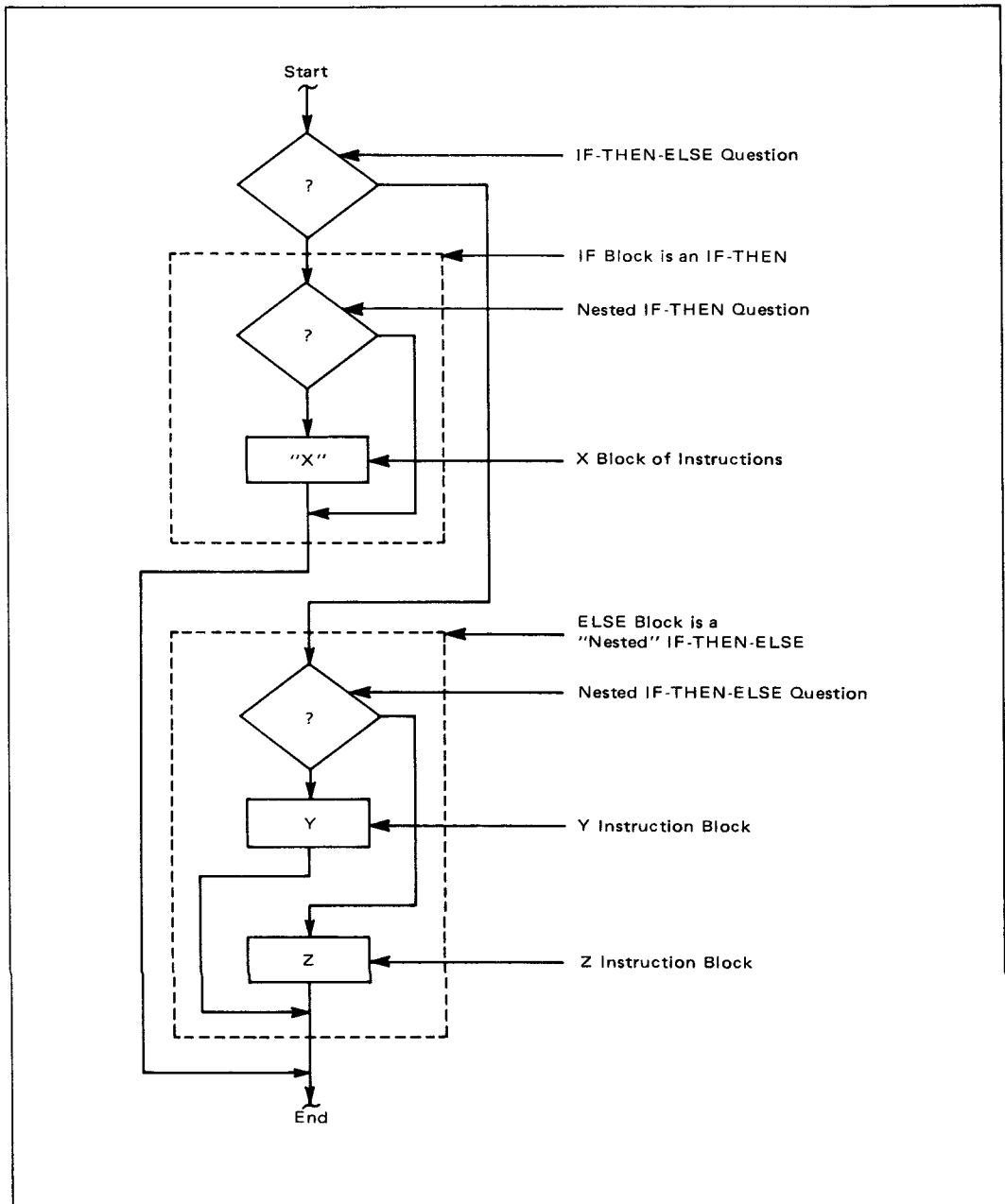


Figure 8.3 IF-THEN-ELSE With Nested Structures

CHAPTER 9 THE WHILE STRUCTURE

The WHILE structure is the last of three program structures required to write programs powerful enough to handle problems of any complexity. The WHILE structure allocates the entire power of the machine to a single section of the problem *while* some condition exists.

The WHILE structure is flow charted as shown in Figure 9.1.

In this chart, we see that after leaving the Executable Block, the program returns to the while condition test. Each machine or computer can have its own unique way to return the program to the test question. In general, these are two ways of doing so: After leaving the executable block, reset the program counter to the entry of the while test question, or, disable the execution of the other blocks of code and let the program counter step through the non-executable instructions until the program loops back to this while test.

The first method, jumping the program counter back, is the most common, fastest and most costly. It will be discussed at the end of this chapter. The second method is less expensive, in that hardware to jump the program counter is not required. It does require a clear understanding of programming requirements and hence, will now be discussed in detail.

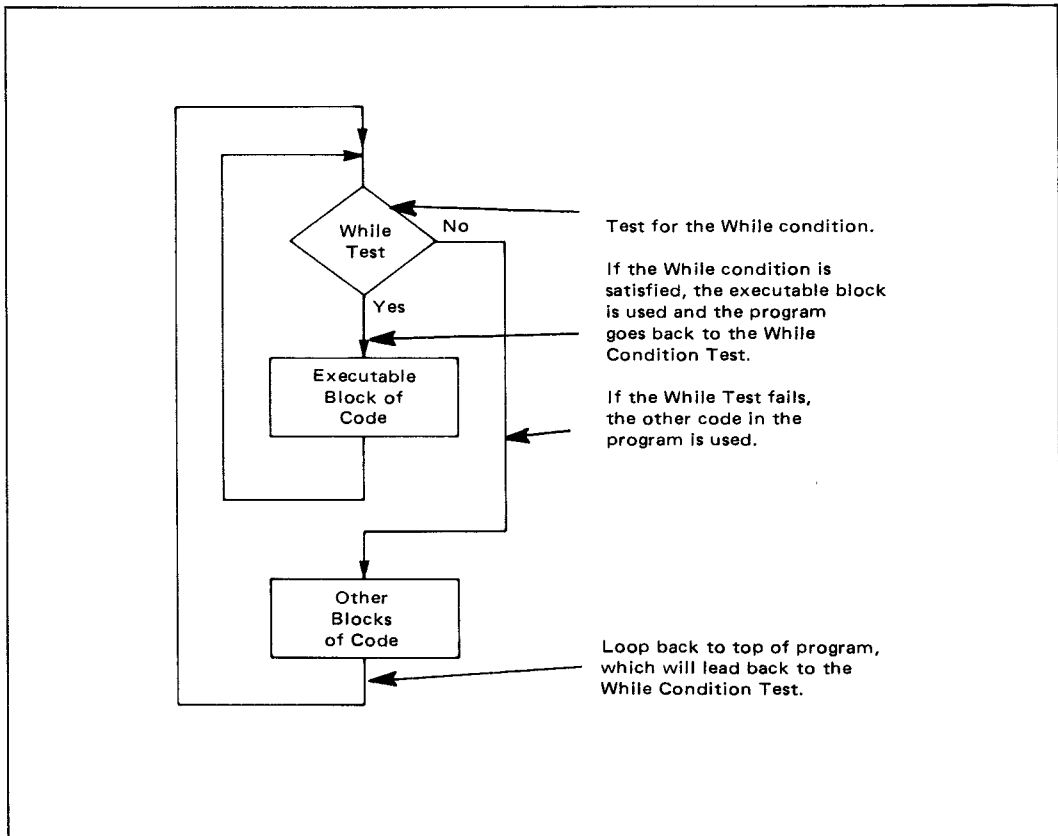


Figure 9.1 The WHILE Structure

Let's call the two ways of implementing the WHILE structure the Jump-Back-While and the Loop-Around-While. Both accomplish the purpose that only one block of code is executable during the time the while condition is satisfied. The Loop-Around-While then puts a condition on the rest of the program: NO code outside the WHILE structure can be executable if the WHILE is enabled. The means of doing this is provided by the OEN instruction which can let the program counter step through instructions without affecting any outputs or memory. This means that the programming problem is one of using OEN to disable other code if a WHILE block is active.

A way of doing this is to count the number of WHILES in a flow chart and assign a unique number to each one. A number called STATE is maintained in memory or temporary storage locations. A while block is enabled only when the blocks' number agrees with STATE. STATE can only be changed by the instructions in the enabled block, or by conditions external to the program, such as timers or other input signals. The Traffic Control application in Chapter 11 is a good example of time being a factor in STATE.

When the Loop-Around-While is used, each while structure's test for enabling the following code tests STATE as part of the while test. This is the same as implementing the WHILE as on IF-THEN-ELSE. "IF the WHILE test condition is satisfied, enable the while block, ELSE enable another block of code. Chapter 10 treats this possibility in detail.

The Jump-Back-While uses two jumps to build the While structure. The first jump is a conditional jump that can move the program counter past the while block if the while test fails. It is at the bottom of the While question block. The second jump is written at the bottom of the executable block and jumps the program counter back to the While question. This jump is not conditional. See Figure 9.2.

An example of a program with a Jump-Back-While is listed next. With this program we want to run a pump until a tank is full. During the pumping time, we want no other action to take place. A flag switch, FS, will close, (FS = 1), when the tank is full.

1	START	LD	FS	LOAD FLOAT SW SIGNAL
2		SKZ		SKIP NEXT INST IF FS = 0.
3		JMP	NEXT	JUMP TO NEXT
4		STO	PUMP	TURN ON PUMP
5		JMP	START	JUMP BACK TO START
6	NEXT	STOC	PUMP	TURN OFF PUMP
7				NEXT INSTRUCTION

Instruction 1 loads the FS signal into the ICU Result Register. If the switch is open, instruction 2 will cause instruction 3 to be ignored. Instructions 1-3 comprise the While question. The executable block is instructions 4 and 5, which drives the pump and jumps the program counter back to instruction 1. Instruction 6 turns off the pump when the while condition fails. After instruction 6, the program counter steps through the balance of the program. The Jump-Back-While is easy to write. Aside from cost, it has a very important limitation: Only one block — the While block — can be executed if the While condition is satisfied.

The Loop-Around-While has two important advantages. First, jumping hardware is not required. Second — and unique to this structure — *two or more completely independent programs can run simultaneously in a single ICU system*. The cost is slower speed of execution and a more complex set of block enabling conditions.

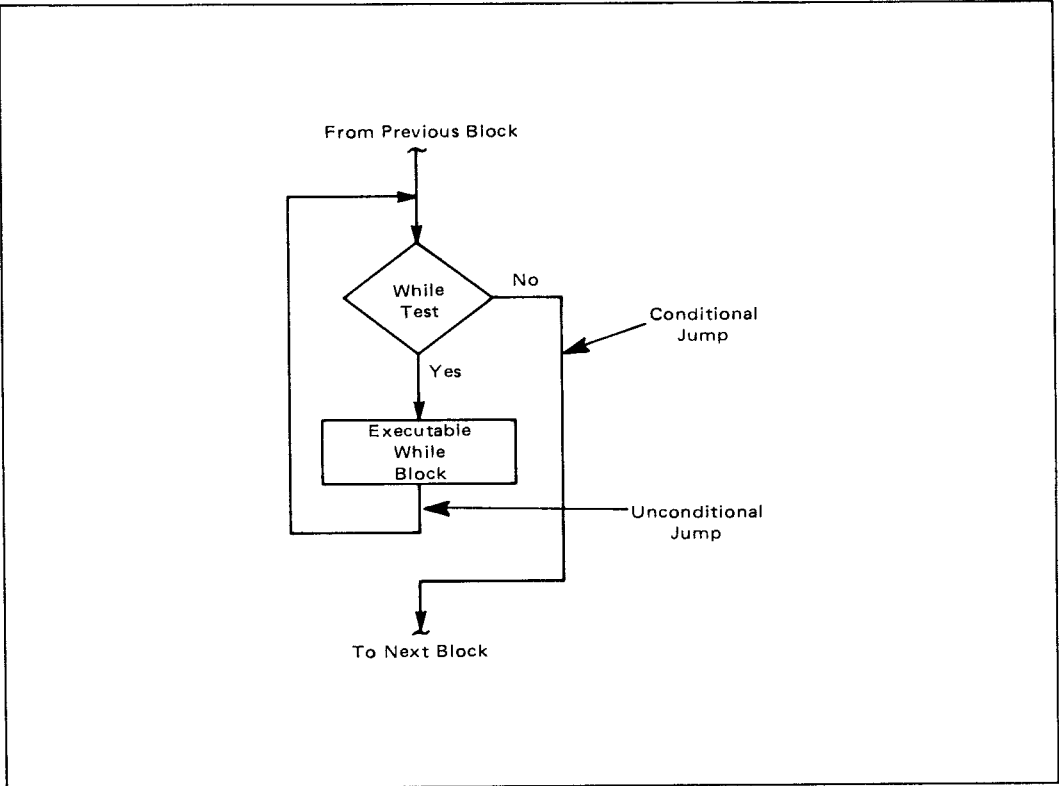


Figure 9.2 Jump – Back – While Structure

CHAPTER 10 COMPLETE ENABLING STRUCTURES

We have seen three important program structures: IF-THEN, IF-THEN-ELSE and WHILE. It is possible to write any program of interest using these three structures. To do so, we must realize that any of these three structures may be used at anyplace within a block of executable instructions. This will lead to nesting. Often, to keep complex programs under control, one will want to draw flow charts.

The flow charts will be most useful if they are conceived as modules that are linked together. Each module has one starting point and one ending point. They are linked by joining the end of one module to the start of another. Thus, the modules will link into one long chain. When the program finishes with the last module, it simply goes back and starts the first module again.

The flow chart should follow a single vertical line. This makes it convenient and easy to add or remove modules when developing programs. This concept is shown in Figure 10.1. For convenience, the flow chart can be drawn as a column which is continued in an adjoining column. This conforms with realistic paper dimensions, as shown in Figure 10.2.

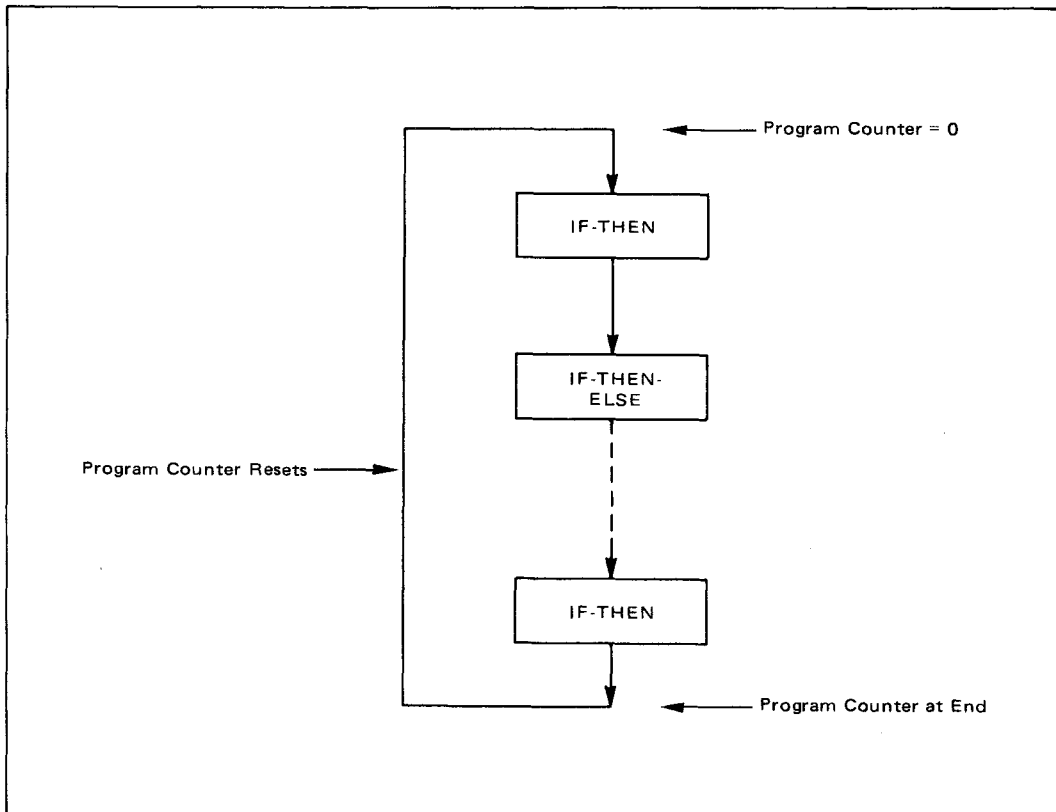


Figure 10.1 A Master Flow Chart Showing Looping and Structured Blocks

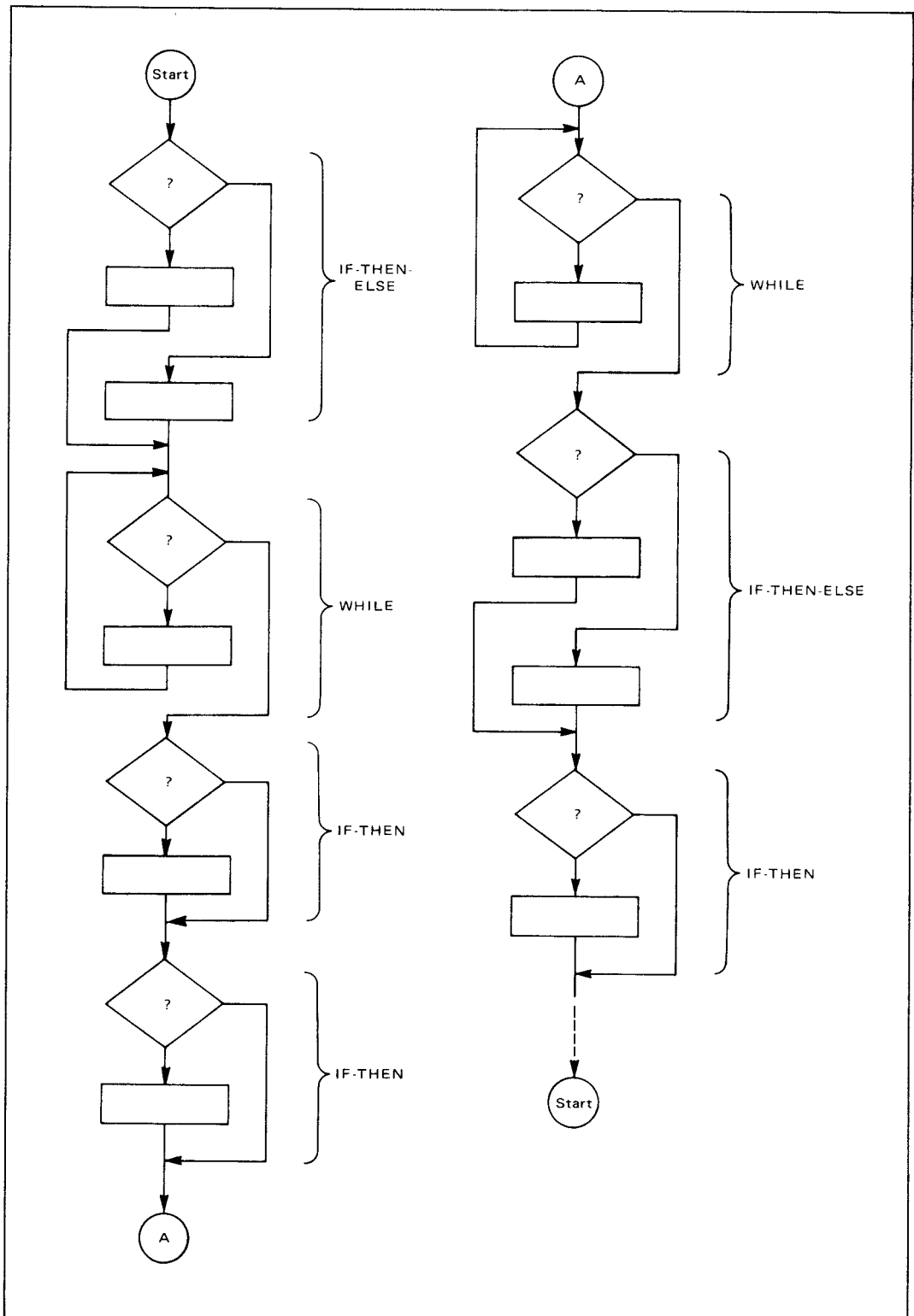


Figure 10.2 A Structured Looping Program

Structured flow charting and programming is a technique for organizing solutions to problems or algorithms. This approach can be used with any programming language or hardware set. Hence, control programs which are structured are convertible between, say, a large Fortran program to the MC14500B, in either direction.

In the MC14500B, one wants to examine the structure to see if WHILEs are present. If WHILEs are present, then no other statements are to be executed until the program exits WHILE block. This may be accomplished in two ways: by using the JUMP instruction from Chapter 12, or by disabling all other code during the time a WHILE block is active. If there are only a few WHILEs in the program, then it is easy to "FLAG" each WHILE and enable blocks only if no WHILEs are active. Otherwise, the state counter technique from the last chapter is recommended. An example of the WHILE test will be shown next, and the "state enable" technique will be shown in the Traffic Controller of Chapter 11.

Example

Shuttle Motor Problem

A motor driven carrier, on a weaving machine, shuttles between a left and a right hand stop each of which have limit switches. When a limit switch closes, the drive motor stops. After a T second pause, the motor runs in the opposite direction until the other limit switch closes, whereupon the cycle repeats.

The various signals have the following designations and characteristics:

LLS	LEFT LIMIT SWITCH	0 = SWITCH CLOSED
RLS	RIGHT LIMIT SWITCH	0 = SWITCH CLOSED
TO	TIMER OUTPUT	1 = TIMING
TI	TIMER INPUT	1 = START INTERVAL
MR	MOTOR RIGHT	1 = RUN TO RIGHT
ML	MOTOR LEFT	1 = RUN TO LEFT

The Task

Document an ICU control system for the shuttle motor.

Solution

We notice that the motor direction is controlled by the last limit switch that closed. So a one bit location called LAST will be used to "remember" the last limit switch closed. Arbitrarily, we will say that LAST = 1 when the last switch closed was the left limit switch LLS.

The I/O and timer connections to the ICU system are shown in Figure 10.3, and Figure 10.4 shows a structured flow chart for the shuttle motor's control. The flow chart uses all three of the structures we have examined. Both the THEN and the ELSE branches of the IF-THEN-ELSE port contain nested IF-THEN structures. So the problem is a good example of what one is likely to have to do in conditionally enabling different blocks of code.

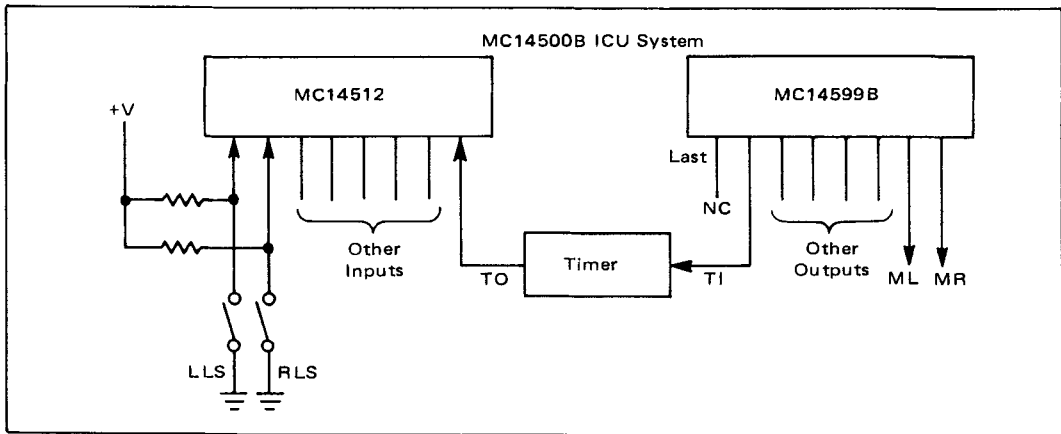


Figure 10.3 Shuttle Motor Signal Connections

In writing code for a flow chart, one often has best results by starting on the innermost structure. So let's look at the instructions for the IF-THEN, which looks for the right limit switch closure.

IF RIGHT LIMIT SW CLOSED

THEN LAST GETS 0

TIMER START GETS PULSED

START	LDC	RLS	RR ← RLS
	STOC	LAST	LAST ← RLS
	STO	TI	TIMER PULSE ON
	ANDC	RR	CLEAR RR TO ZERO
END	STO	TI	TIMER PULSE OFF

This is direct execution. If the same structure were to be coded using Output Enable instructions it would like like:

START	LDC	RLS	RR ← RLS
	OEN	RR	OEN ENABLE BY RR = 1
	STOC	LAST	CHANGE LAST
	STO	TI	PULSE TIMER ON
	STOC	TI	END TIMER PULSE
	ORC	RR	FORCE RR TO 1
END	OEN	RR	RESTORE OEN

This block of code is probably ENABLED by using the OEN instruction ahead, so we do not want to use OEN within the block. The direct method is preferred.

However, other ways of writing code for this block are possible, for example, using the Skip If Zero instruction.

START	LDC	RLS	RR ← RLS
	SKZ		SKIP NEXT IF ZERO
	STOC	LAST	LAST ← 0
	SKZ		SKIP NEXT IF ZERO
	STO	TI	TIMER PULSED ON
	ANDC	RR	RR FORCE TO ZERO
END	STO	TI	TIMER PULSED OFF

Many ways of writing short blocks of code are possible. Again, direct methods are preferable.

Now, let's look at code for the IF-THEN-ELSE block that controls the motor's run direction. OEN is 1 and the block is enabled when we start.

```
OEN = 1
IF-THEN-ELSE
IF LAST = 1
    THEN MOTOR RUNS RIGHT
        IF RLS = 0
            THEN LAST 0, TIMER STARTS
    ELSE MOTOR RUNS LEFT
        IF LLS = 0
            THEN LAST = 1, TIMER STARTS
```

Noting on the flow chart that the IF-THEN-ELSE structure is active whenever TO = 0, we can combine TO into the branching decision of this structure.

START	LDC	TO	LOAD \overline{TO}
	AND	LAST	RR = LAST · \overline{TO}
	OEN	RR	OEN ← LAST · \overline{TO}
	STO	MR	RUN RIGHT ON
	STOC	ML	RUN LEFT OFF

·
·
·

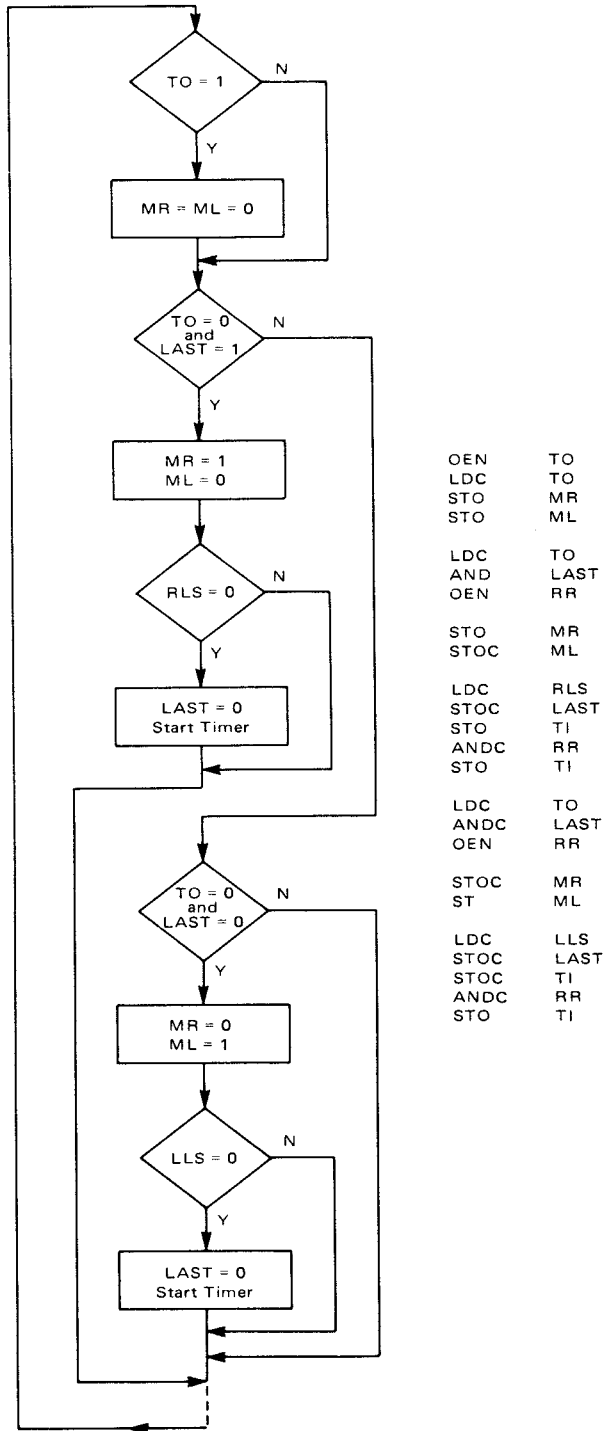
INNER IF-THEN BLOCK FROM BEFORE FOR MTR RT

LDC	TO	LOAD \overline{TO}
ANDC	LAST	RR = \overline{LAST} · \overline{TO}
OEN	RR	OEN ← \overline{LAST} · \overline{TO}
STOC	MR	RUN RIGHT OFF
STO	ML	RUN LEFT ON

·
·
·

INNER IF-THEN BLOCK FROM BEFORE FOR MTR LFT

We have now “collapsed” the code and flow chart to the simpler form shown in Figure 10.5. One of the points of this section is that the “work” is not in writing code, but in understanding, organizing and documenting a problem and a solution procedure or algorithm. When a problem and its solution are well understood, flow charting and documentation are often expedited by the immediate writing of code.



OEN	TO
LDC	TO
STO	MR
STO	ML
LDC	TO
AND	LAST
OEN	RR
STO	MR
STOC	ML
LDC	RLS
STOC	LAST
STO	TI
ANDC	RR
STO	TI
LDC	TO
ANDC	LAST
OEN	RR
STOC	MR
ST	ML
LDC	LLS
STOC	LAST
STOC	TI
ANDC	RR
STO	TI

Figure 10.5 Final Structured Flow Chart and Motor Control Code

CHAPTER 11 TRAFFIC INTERSECTION CONTROLLER

In this chapter, many of the concepts previously examined are consolidated in an example of a traffic intersection controller. The controller is developed around the 16 input/16 output demonstration system that was described in Chapter 5. This example illustrates the power of the ICU's instruction set and exemplifies the new concept of branching code that is fetched and executed entirely sequentially, thus, eliminating the need for conventional branching or jumping by modifying the contents of program counter.

State Diagram of the Controller

One of the many ways to visualize a problem or task is by means of a state diagram. This method presents the maximum information, regarding the task, in very compact form. Figures 11.1 and 11.2 depict the controller states in simplified and complete diagrams, respectively.

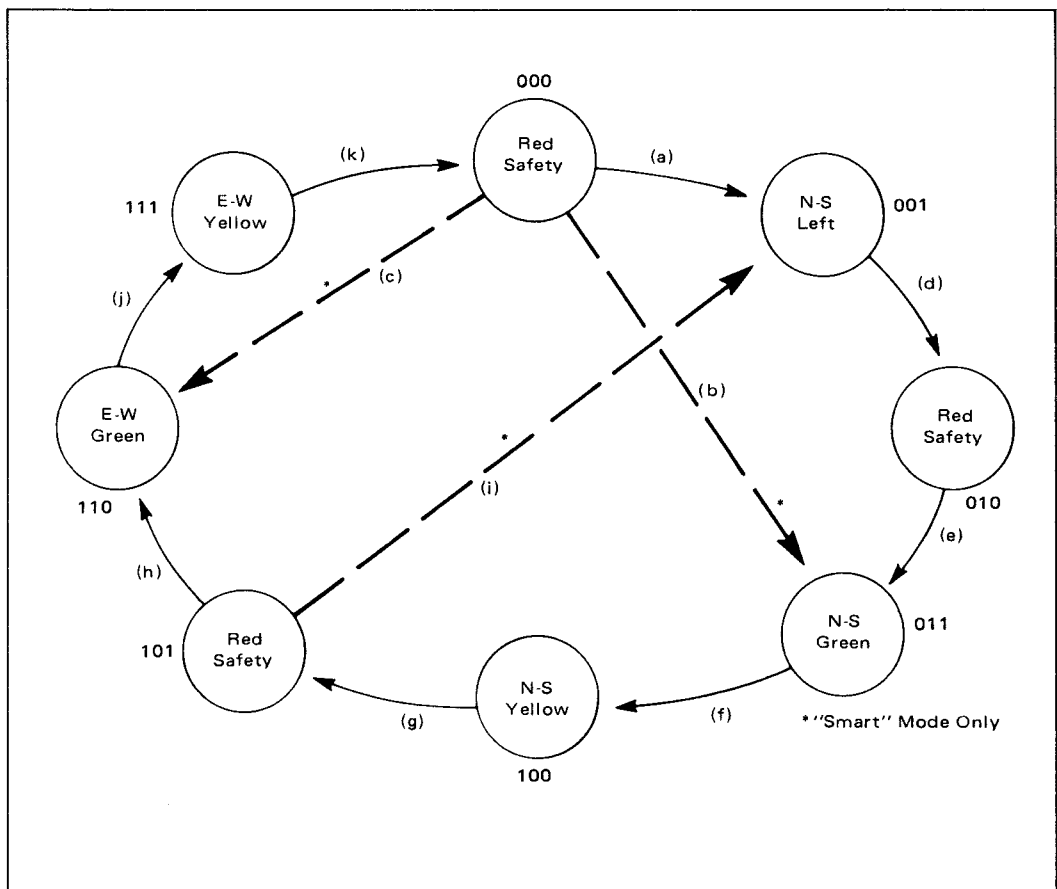


Figure 11.1 Traffic Controller – Simplified State Diagram

As shown in the figures, the traffic controller has 8 separate states. The conditions for leaving a state and advancing to another are shown as labels for the lines connecting each state pair on Figure 11.2. Priority is assigned in the following manner: if there are several paths that can be taken upon leaving a state the controller will advance to the nearest neighboring state, following a clockwise path around the outside of the state diagram circle. For example, if the controller were in State 0 and in its “smart” mode, (to be described later), and if there are “North-South” and “East-West” traffic requests, but no left turn request, the controller would advance directly to State 3, skipping States 1 and 2.

Features of the Intersection Controller

The intersection to be controlled consists of a major highway (North-South) and a minor cross street (East-West). The N-S highway has left-turn arrows operating simultaneously. The controller can operate in either of two modes: sequenced (dumb) and responsive (smart). The mode of operation is selected by a switch on the demonstration unit.

In the sequenced mode, the controller “steps” through each state (clockwise path around the state diagrams). In this mode, the sequence is repeated endlessly, without regard to traffic build-up.

In the responsive mode, the controller will “rest” with the N-S green light on and “answer” requests for left turns and E-W traffic. After servicing these requests, the controller returns to the N-S green state. If many requests are made simultaneously, the controller selects which request has priority. Each request will be serviced once in each cycle. This is done so that no request will be denied because of a large amount of traffic in a high priority direction.

The controller also has various time delays, (N-S Green time, etc.) that are programmable by switches. Other features are a hard-wired constraint that if N-S green is off and N-S yellow is off, N-S red must be on. A similar condition exists in the east-west directions; also, a hard-wired “flash all reds” function when the ICU is in the reset mode.

Intersection Controller Flow Chart

The state diagram is a universal, generalized representation of the task. The flow chart is a more specific representation that allows design tradeoffs between system hardware and software to be resolved. Figure 11.3 is the complete flow chart for the implementation of the controller. Table 11.1 lists and describes the input/output function and the mnemonic terms assigned to the controller system.

The resultant program for the controller consists of eleven IF-THEN structures. Each IF-THEN structure implements one of the arrows and states shown on Figures 11.1 & 11.2. In any of the structures, the IF part tests to see if the conditions exist for entering a new state. The THEN parts change Flag bits to indicate which new state the controller has entered. Next, the THEN parts execute the time and output-state code that corresponds to the new state.

The Flag bits B2, B1 and B0 signify which state the controller is occupying. For instance, if B2, B1 and B0 equal binary 0, 1 and 1, respectively, the controller is in State 3 (011₂). During start-up, or after a reset, the controller proceeds from State 0.

Intersection Controller Software

Table 11.2 is the controller software, based on the flow chart of Figure 11.3.

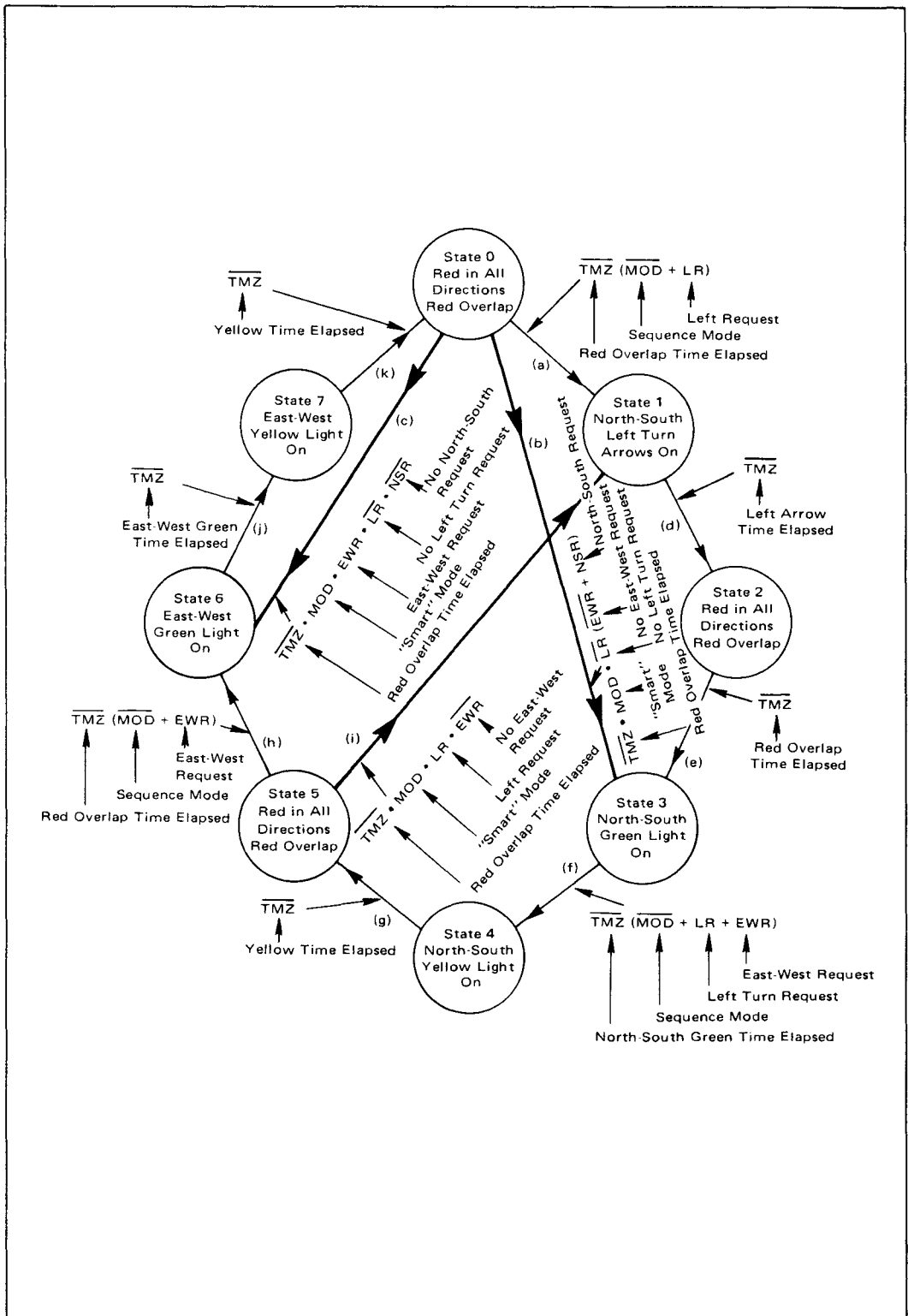


Figure 11.2 Complete State Diagram with Conditional Linkages

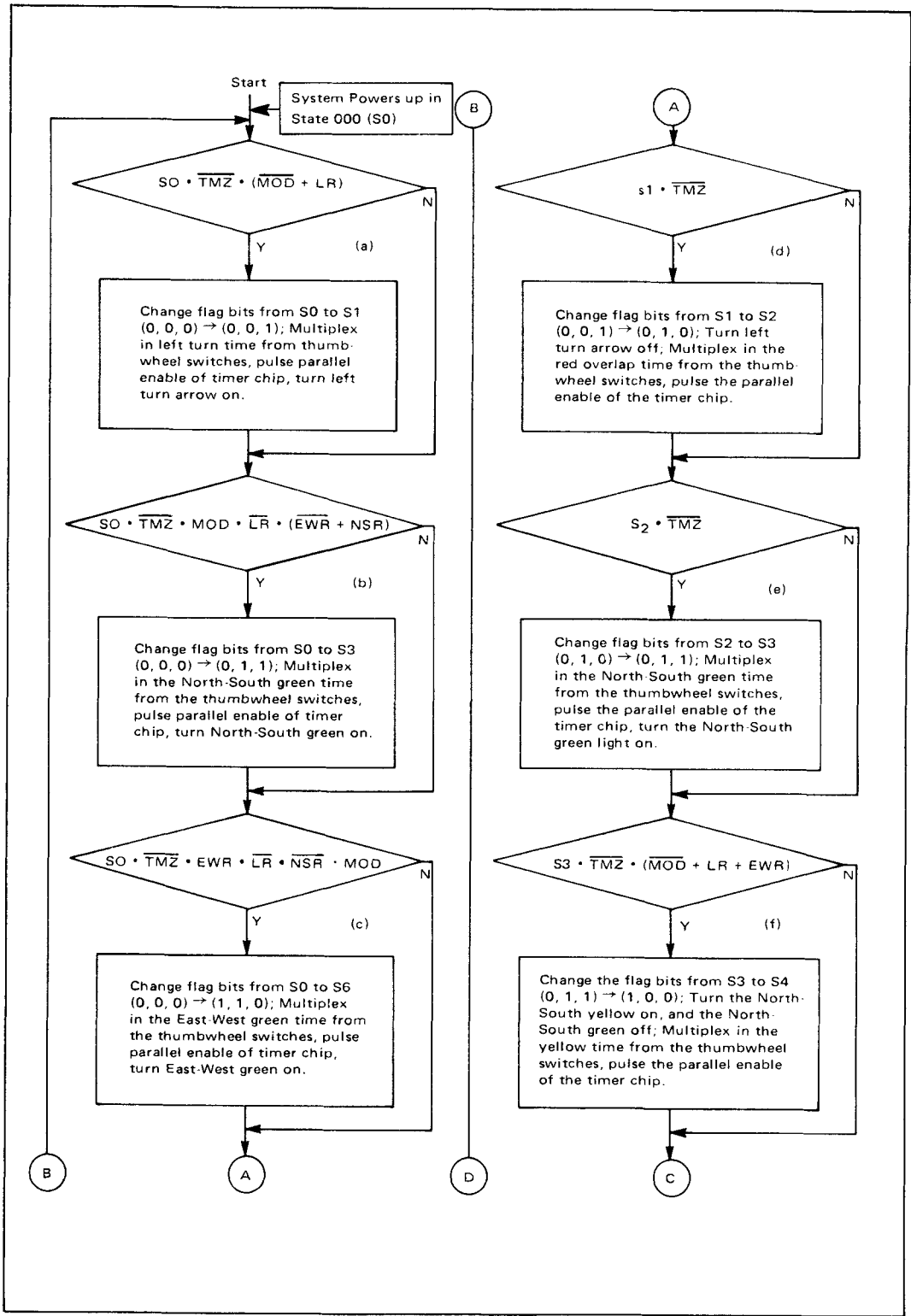


Figure 11.3 Flow Chart

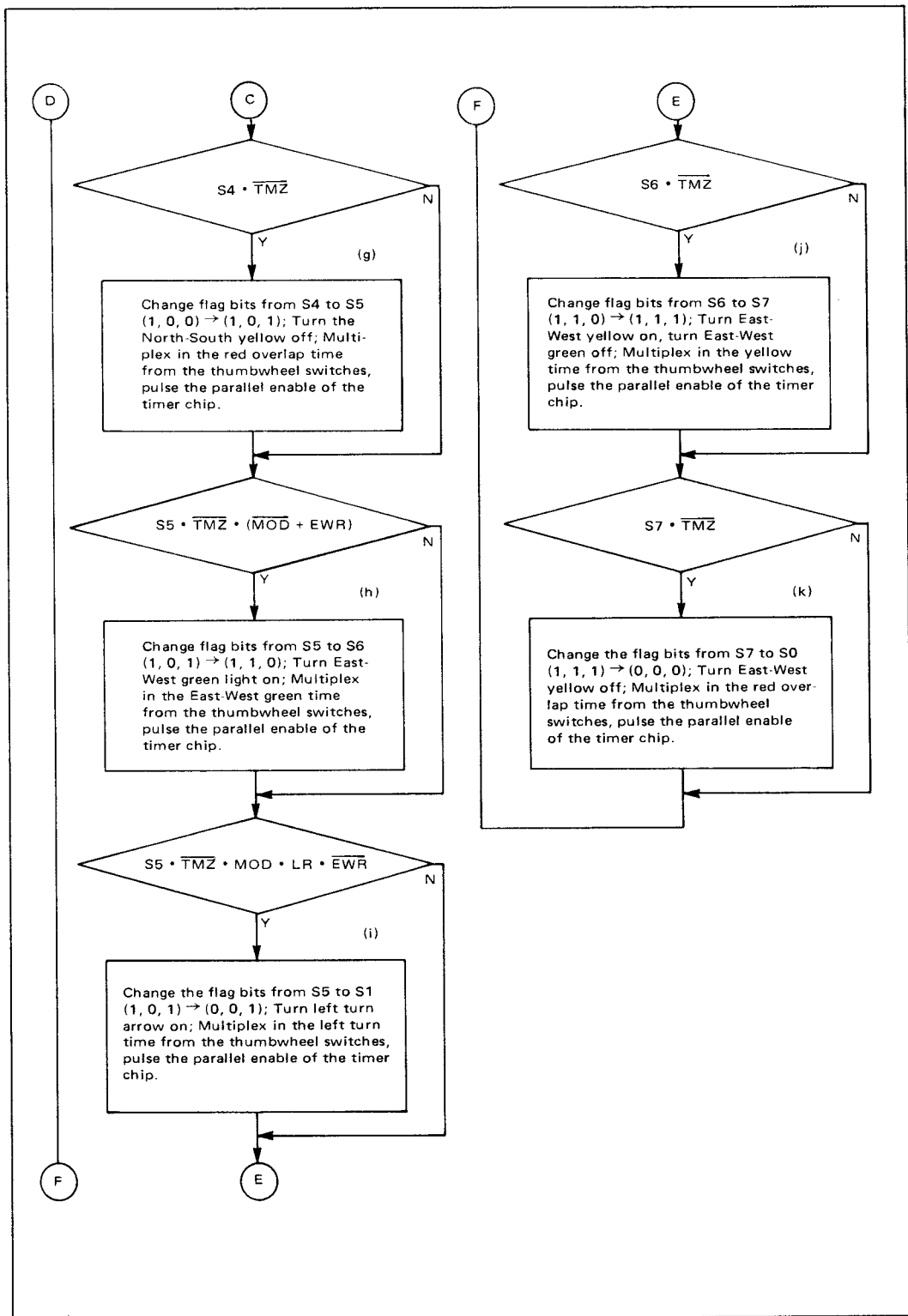


Figure 11.3 Flow Chart (continued)

Table 11.1 Intersection Controller Input/Output Listing

ICU Inputs

Input #	Name	Function
0	RR	The pinned out Result Register is connected to this input so the ICU can conditionally load the IEN and OEN register, and manipulate the result register content.
1	LR	Signal indicating that a request for a left turn has been made.
2	MOD	Selects the mode of operation the intersection will function in MOD = 1 smart; MOD = 0 sequence.
3	$\overline{\text{TMZ}}$	This is the (low active) carry out of the timer chip, the monitoring of this input determines when time has elapsed.
4		Inputs 4, 5, and 6 are tied to outputs 8, 9, and 10 respectively. The software uses these three bits as flags to determine which block of code will be "active" as the ICU sequences through the instructions in memory.
5		
6		
7	NSR	Signal indicates that a request for the North-South green light has been made.
8	EWR	Signal indicates that a request for the East-West green light has been made.

ALL SIGNALS ARE HIGH ACTIVE UNLESS OTHERWISE SPECIFIED

ICU Outputs

Output #	Name	Function
0	Left TMR	Multiplexes the left turn time to the inputs of the down counter.
1	PE	Parallel enable of the down counter.
2	ARROW	Left turn arrow light.
3	NSGRNTMR	Multiplexes the North-South green time to the inputs of the down counter.
4	NSG	North-South green light.
5	EWGRNTMR	Multiplexes the East-West green time to the input of the down counter.
6	EWG	East-West green light.
7	EWY	East-West yellow light.
8		Outputs 8, 9, and 10 are tied to inputs 4, 5, and 6 respectively. The software uses these three bits as flags to determine which block of code will be active as the ICU sequences through the instruction in memory.
11	NSY	
12	YTMR	
13	REDSFTMR	Multiplexes the red overlap time to the inputs of the down counter.

Table 11.2 Intersection Controller Program

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment
00	7	0	XNOR	RR	Force RR to 1
01	A	0	IEN	RR	Enable input

IF-THEN BLOCK (a)

02	1	1	LD	LR	Load LR
03	6	2	ORC	MOD	OR with MOD
04	4	4	ANDC	B0	AND with B0
05	4	5	ANDC	B1	AND with B1
06	4	6	ANDC	B2	AND with B2
07	4	3	ANDC	$\overline{\text{TMZ}}$	AND with TMZ (Low Active)
08	B	0	OEN	RR	Enable if R = 1
09	8	8	STO	B0	Change State to S1
0A	8	0	STO	LEFTMR	Enable Left Time SW
0B	8	1	STO	PE	Pulse Timer Load
0C	9	1	STOC	PE	Pulse Off
0D	9	0	STOC	LEFTMR	Disable Left Time SW
0E	8	2	STO	ARROW	Turn On Left Arrow

IF-THEN BLOCK (b)

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment
OF	1	7	LD	NSR	Code in this Block follows Comments in Block (a)
10	6	8	ORC	EWR	
11	4	1	ANDC	LR	
12	3	2	AND	MOD	
13	4	3	ANDC	TMZ	
14	4	4	ANDC	B0	
15	4	5	ANDC	B1	
16	4	6	ANDC	B2	
17	B	0	OEN	RR	
18	8	8	STO	B0	
19	8	9	STO	B1	
1A	8	3	STO	NSGRNTMR	
1B	8	1	STO	PE	
1C	9	1	STOC	PE	
1D	9	3	STOC	NSGRNTMR	
1E	8	4	STO	NSG	

IF-THEN BLOCK (c)

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment
1F	2	4	LDC	B0	Logic Flow is same as Block (a)
20	4	5	ANDC	B1	
21	4	6	ANDC	B2	
22	3	2	AND	MOD	
23	3	8	AND	EWR	
24	4	1	ANDC	LR	
25	4	7	ANDC	NSR	
26	4	3	ANDC	TMZ	
27	B	0	OEN	RR	
28	8	9	STO	B1	
29	8	A	STO	B2	
2A	8	5	STO	EWGRNTMR	
2B	8	1	STO	PE	
2C	9	1	STOC	PE	
2D	9	5	STOC	EWGRNTMR	
2E	8	6	STO	EWG	

IF-THEN BLOCK (d)

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment
2F	1	4	LD	B0	See Block (a) for Comments
30	4	5	ANDC	B1	
31	4	6	ANDC	B2	
32	4	3	ANDC	TMZ	
33	B	0	OEN	RR	
34	9	8	STOC	B0	
35	8	9	STO	B1	
36	9	2	STOC	ARROW	
37	8	D	STO	REDSFTMR	
38	8	1	STO	PE	
39	9	1	STOC	PE	
3A	9	D	STOC	REDSFTMR	

IF-THEN BLOCK (e)

3B	2	4	LDC	B0	Form Same as Block (a)
3C	3	5	AND	B1	
3D	4	6	ANDC	B2	
3E	4	3	ANDC	\overline{TMZ}	
3F	B	0	OEN	RR	
40	8	8	STO	B0	
41	8	3	STO	NSGRNTMR	
42	8	1	STO	PE	
43	9	1	STOC	PE	
44	9	3	STOC	NSGRNTMR	
45	8	4	STO	NSG	

IF-THEN BLOCK (f)

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment
46	1	8	LD	EWR	See Block (a)
47	5	1	OR	LR	
48	6	2	ORC	MOD	
49	4	3	ANDC	\overline{TMZ}	
4A	3	4	AND	B0	
4B	3	5	AND	B1	
4C	4	6	ANDC	B2	
4D	B	0	OEN	RR	
4E	9	8	STOC	B0	
4F	9	9	STOC	B1	
50	8	A	STO	B2	
51	8	B	STO	NSY	
52	9	4	STOC	NSG	
53	8	C	STO	YTMR	
54	8	1	STO	PE	
55	9	1	STOC	PE	
56	9	C	STOC	YTMR	

IF-THEN BLOCK (g)

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment
57	2	4	LDC	B0	Same Structure as Block (a)
58	4	5	ANDC	B1	
59	3	6	AND	B2	
5A	4	3	ANDC	\overline{TMZ}	
5B	B	0	OEN	RR	
5C	8	8	STO	B0	
5D	9	8	STOC	NSY	
5E	8	D	STO	REDSFTMR	
5F	8	1	STO	PE	
60	9	1	STOC	PE	
61	9	D	STOC	REDSFTMR	

IF-THEN BLOCK (h)

62	1	8	LD	EWR	Structure or Block (a) Repeated
63	6	2	ORC	MOD	
64	4	3	ANDC	TMZ	
65	3	4	AND	B0	
66	4	5	ANDC	B1	
67	3	6	AND	B2	
68	B	0	OEN	RR	
69	9	8	STOC	B0	
6A	8	9	STO	B1	
6B	8	6	STO	EWG	
6C	8	5	STO	EWGRNTMR	
6D	8	1	STO	PE	
6E	9	1	STOC	PE	
6F	9	5	STOC	EWGRNTMR	

IF-THEN BLOCK (i)

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment
70	1	4	LD	B0	Structure repeats again . . .
71	4	5	ANDC	B1	
72	3	6	AND	B2	
73	4	3	ANDC	TMZ	
74	3	2	AND	MOD	
75	3	1	AND	LR	
76	4	8	ANDC	EWR	
77	B	0	OEN	RR	
78	9	A	STOC	B2	
79	8	2	STO	ARROW	
7A	8	0	STO	LEFTTMR	
7B	8	1	STO	PE	
7C	9	1	STOC	PE	
7D	9	0	STOC	LEFTTMR	

IF-THEN BLOCK (j)

7E	2	4	LDC	B0	And again . . .
7F	3	5	AND	B1	
80	3	6	AND	B2	
81	4	3	ANDC	TMZ	
82	B	0	OEN	RR	
83	8	8	STO	B0	
84	8	7	STO	EWY	
85	9	6	STOC	EWG	
86	8	C	STO	YTMR	
87	8	1	STO	PE	
88	9	1	STOC	PE	
89	9	C	STOC	YTMR	

IF-THEN BLOCK (k)

Memory Location	Op Code	I/O Address	Mnemonic Op Code	Symbolic Address	Comment	
8A	1	4	LD	B0	Start Last Block	
8B	3	5	AND	B1		
8C	3	6	AND	B2		
8D	4	3	ANDC	TMZ		
8E	B	0	OEN	RR		
8F	9	8	STOC	B0		
90	9	9	STOC	B1		
91	9	A	STOC	B2		
92	9	7	STOC	EWY		
93	8	D	STO	REDSFTMR		
94	8	1	STO	PE		
95	9	1	STOC	PE		
96	9	D	STOC	REDSFTMR		End Last Block & Prog.
97	F	X	NOPF	No Address		Flag F can be used to reset program counter after last instruction if Flag F resets PC
97	F	X	NOPF			or the balance of ROM will contain NOP's when the rest of the locations are unprogrammed and the program will automatically loop around to the first instruction.
FF	F	X	NOPF			
97	0	X	NOPO			
FF	0	X	NOPO			

INTERSECTION CONTROLLER HARDWARE

Display Board

The traffic intersection display board is controlled by the ICU and the control program located in the demonstration board ROM. Two 16-wire cables interface the display board to the ICU system inputs and outputs. The display uses a separate power supply. The display board has a "hard wired" flash feature where the red lights flash in both directions when the ICU is in the reset state. The display board has three request buttons which are used to simulate traffic conditions. When the request button has been pushed, the request is latched and displayed by an LED. The request light will go off after the request has been serviced (i.e. that particular direction gets a green light).

Timing

There are five different time intervals in the traffic controller, each settable by a thumbwheel switch. The intervals are: N-S Green Time, E-W Green Time, N-S Left Turn Time, Common Yellow Time and Red Overlap Time. The Red Overlap or Red Safety interval allows the last car traveling on Yellow to clear the intersection before the next direction starts a Green interval. During red overlap or clearance time, all red lights are on.

When a state is entered, a common counter is loaded with the state of the proper thumbwheel switch. The thumbwheel switches are connected by diodes to a common four wire bus used to load a down counter. When a particular time is to be used, an ICU output is sent high to drive the common line of one of the switches. The counter's load pin is then pulsed by another ICU system output, and the switch's common line is returned to a low state. (See the section on timers, Chapter 6). The switches and the counter/timer are on the Display Board shown in Figure 11.4.

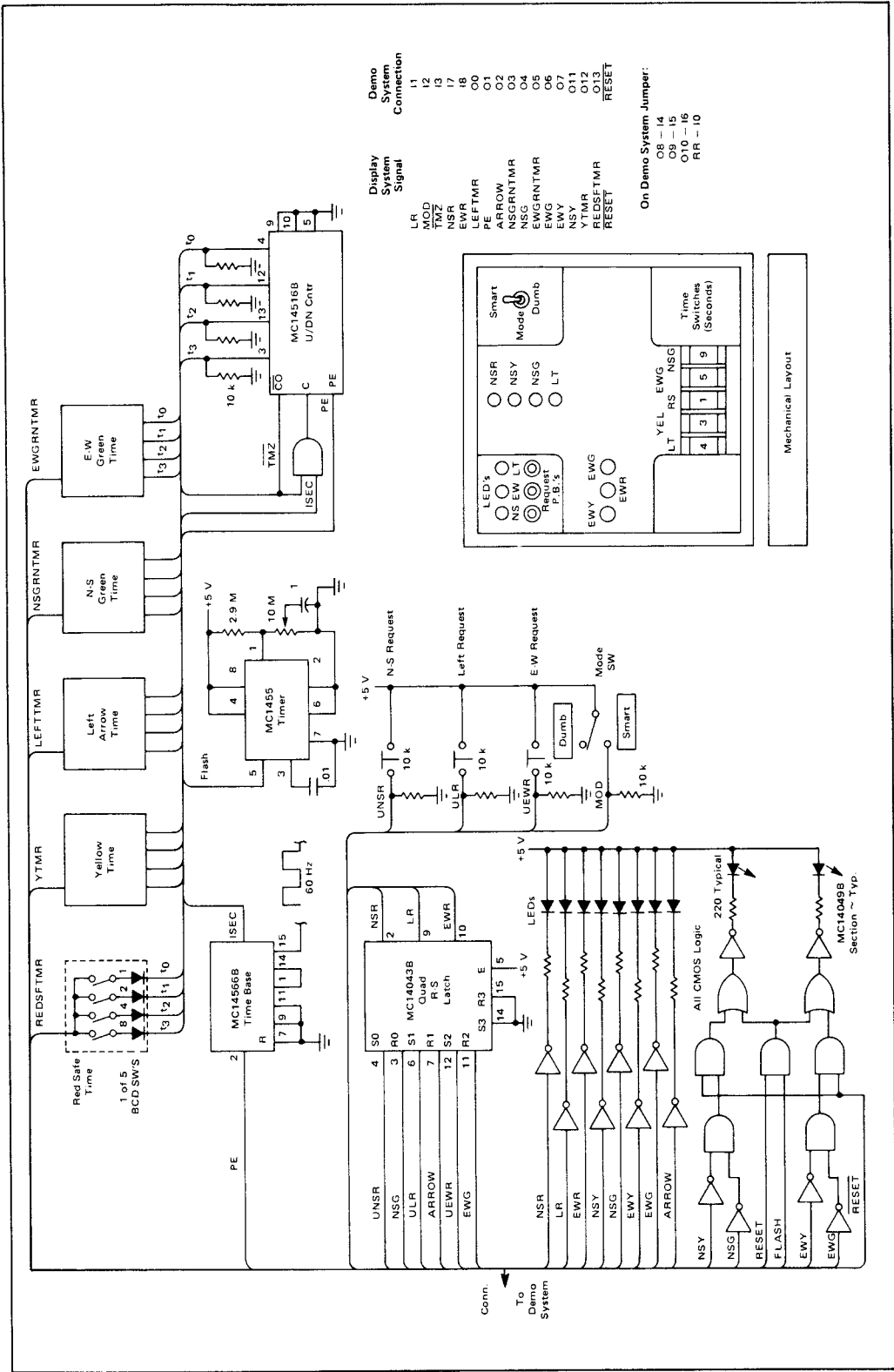


Figure 11.4 Traffic Intersection Controller

CHAPTER 12 ADDING JUMPS, CONDITIONAL BRANCHES, AND SUBROUTINES

In some control applications, it may be advantageous to have a control structure like that of a conventional processor, rather than a looping control structure. Jumping can reduce execution time and reduce software complexity. Having the capability to call subroutines also helps to modularize the software. It should be pointed out that subroutines can be implemented in a looping control structure; however, the overhead required (additional processing steps) may be disadvantageous in some cases. An ICU system can be readily designed to incorporate a jumping, conditional branching and subroutine capability.

The ICU has three program control instructions which are intended for the purpose of adding jumping, conditional jumping and subroutine capabilities to an ICU system. These instructions cause the ICU to take the appropriate action and provide the necessary control signals to external logic circuits that actually perform the address modifications.

Program Control Instructions

JUMP, (Mnemonic: JMP). The JMP instruction generates a one clock period pulse on the JMP pin of the ICU, beginning on the falling edge of the CLK signal. This pulse can be used to gate the jump address into the program counter.

SKIPIFRR = 0, (Mnemonic: SKZ). If the Result Register contains a logic 0 at the time the SKZ instruction is executed, the next instruction is ignored by the ICU. (i.e. no action is taken.)

Together the JMP and SKZ instructions give the ICU a conditional branch capability. See Figure 12.1.

To add subroutines to the ICU, a Last In, First Out Memory (LIFO "stack") is required. If the subroutine feature is required, the most economical method of implementing this feature is to have a LIFO stack in which the top location of the stack is a parallel-loadable counter, where the outputs of the top location (counter) are available as address lines. Figure 12.2 diagrams this situation. There are a number of excellent LSI CMOS parts available which exactly implement the function shown in Figure 12.2.

The ICU does not have a JSR (jump to subroutine) instruction; however, both of the NOP instructions create control signal pulses and either could be used as a JSR instruction. This pulse can be used to signal the program stack to perform the "push" (store binary states) operation while the address of the subroutine is parallel-loaded into the top location of the stack.

LD	BIT	LDC	BIT
SKZ		SKZ	
JMP	BITSET	JMP	BITSET
.		.	
.		.	
CODE FOR BRANCH IF		CODE FOR BRANCH IF	
BIT IS SET		BIT = 0	

Figure 12.1 Conditional Branching

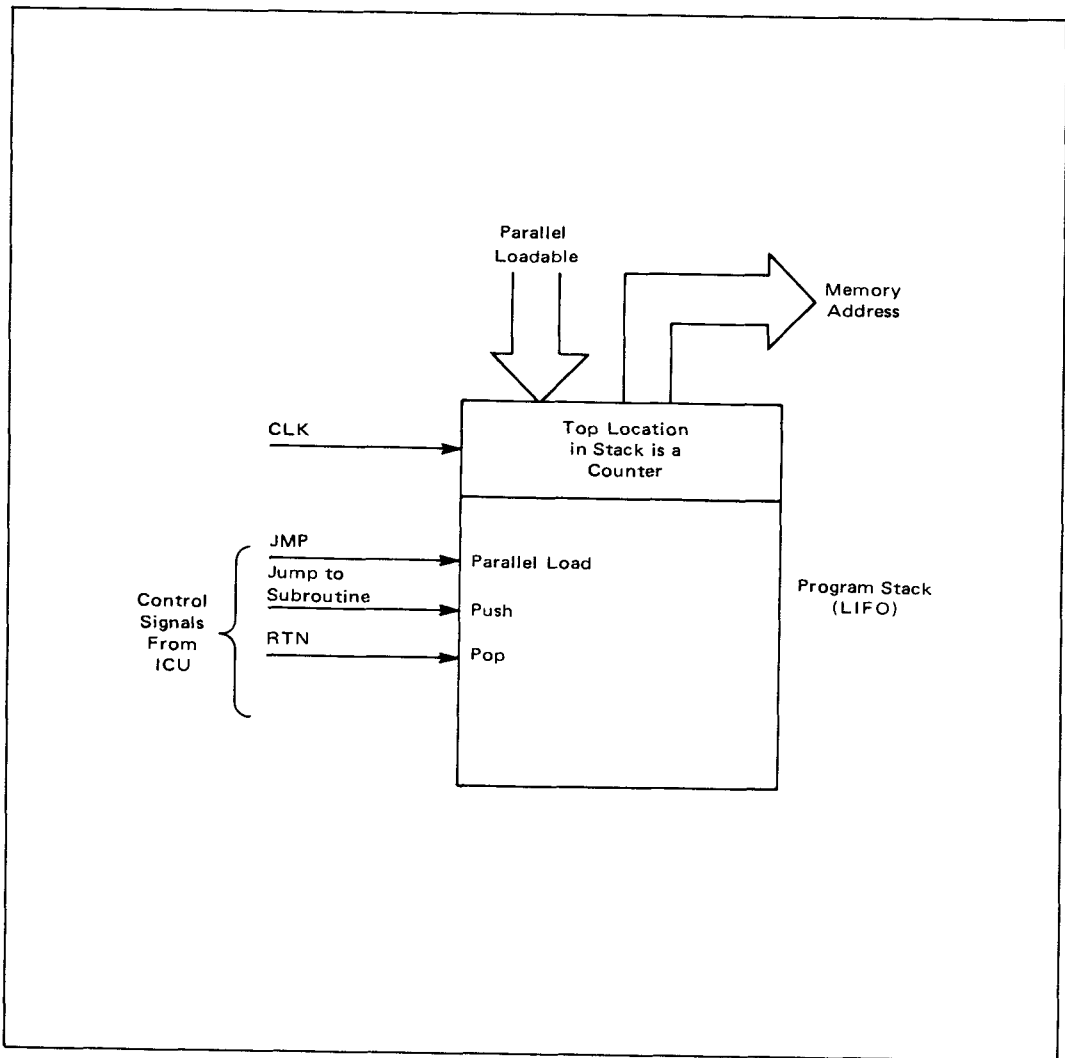


Figure 12.2 Block Diagram of Subroutine Structure

The ICU does provide a return (RTN) instruction. The RTN instruction creates a one clock period pulse on the RTN pin of the ICU. This pulse can be used to signal the program stack to perform the “pop” (return binary states) operation. After executing an RTN, the next instruction is ignored by the ICU. This is done because popping the stack returns the address of the JSR instruction to the top location in the stack. If this instruction was not skipped, the machine would be trapped in an infinite loop.

Figure 12.3 shows a schematic diagram of an ICU system with a parallel/interlaced memory structure, scratchpad RAM and a program stack. The system was designed to address 7 blocks of I/O ports, with each I/O block containing 256 inputs and 256 outputs, thereby providing a total of 1792 inputs and 1792 outputs. This system has 1024 by 1 bit scratchpad locations, a program stack which is 12 bits wide by 16 locations deep and the capability of holding 4096 ICU statements.

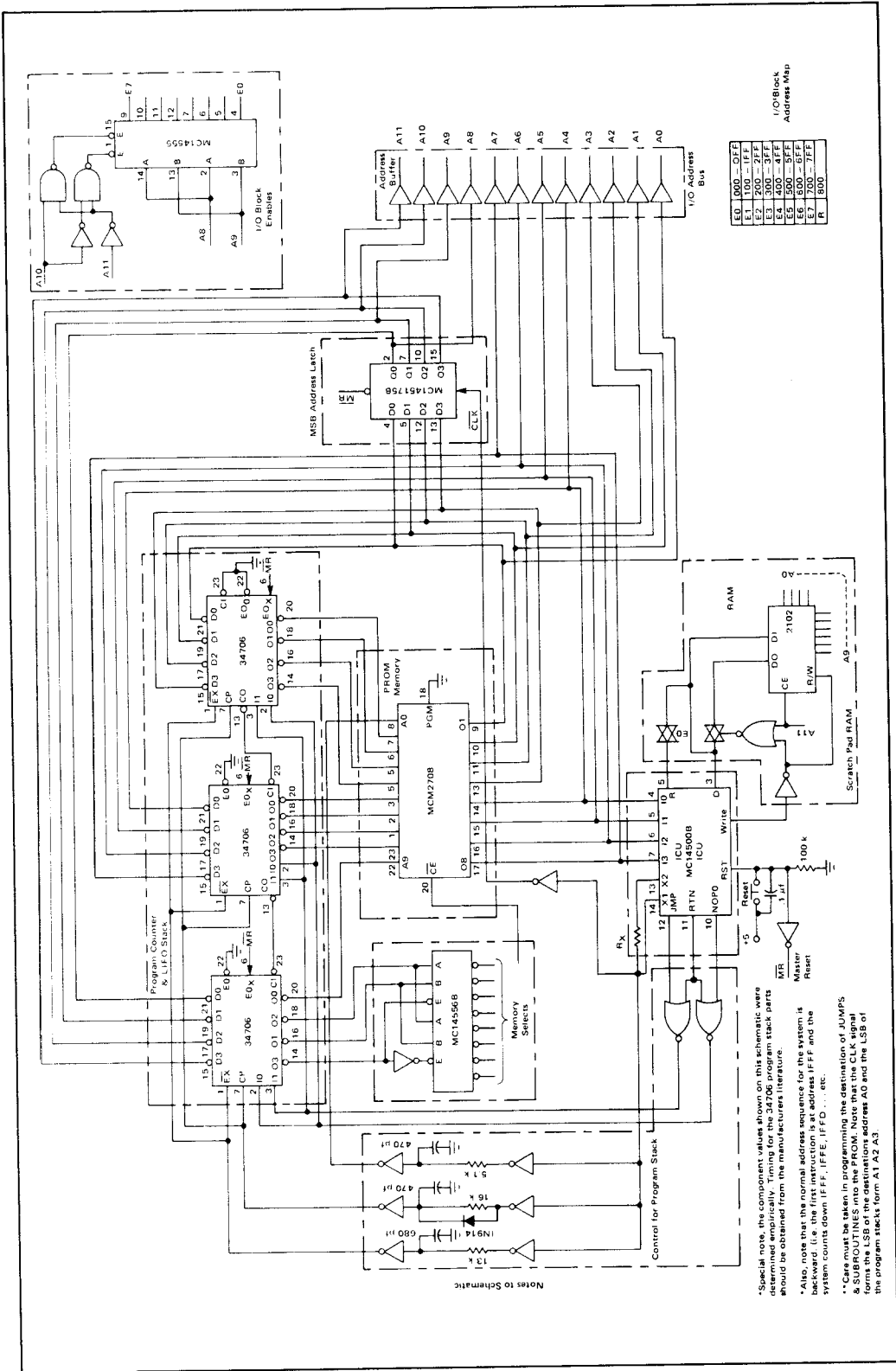


Figure 12.3 ICU Based System with Subroutine Capability

CHAPTER 13 MODULARIZING HARDWARE SYSTEMS

MC14500B ICU systems can be built in a variety of sizes, all of which depend upon specific applications. Many users will want to configure “standard” systems that can be used for a span of applications or end products. Others may want to expand a starter system into a large system. Concepts which highlight opportunities for economy and system partitioning are the subject of this chapter.

Stand-Alone Single-Card Systems

An ICU system, described earlier, had 16 I/O lines; 8 Inputs and 8 Outputs. Without changing any logic, one more input and another output device can be added, making a small system with 32 I/O lines.

The minimal system had 4 I/O address bits in memory. As the WRITE signal was used to differentiate between Input and Output, the 4 address bits can code $2^4 = 16$ inputs and 16 outputs. Increasing I/O past 32 lines, requires more memory bits for addressing. As memories are made in width multiples of 4 bits, the next practical number of address bits will be 8, which will handle 256 inputs and 256 outputs. This is more than adequate for the majority of applications. The next four bit increment of memory width takes us to 12 address bits, enough to code 4096 each of input and output. Let us now examine systems that have 4, 8 and 12 bits of I/O addressing.

The previously described ICU system used a 4 bit I/O address from a four bit wide memory. The memory words alternated Operator/Operand/Operator/Operand/. . . or Instruction/Address/Instruction/Address/. . ., which is of course, the same thing. The reason for this interleaving is to put a small program into the smallest ROM, 256 X 4 bits. The MC14500B was conceived to work with either an interleaved structure of Op-codes and Addresses or to have both appear on a single word of memory. The interleaved technique uses the Pin 14 clock as the least significant bit of ROM address, where as the single wider word uses the LSB of the program counter as the lowest ROM address bit.

Useful ROM organizations are shown in Figure 13.1. The configurations shown are not exhaustive, but represent the most popular choices. A ROM configuration, once chosen, is not readily changed. The choice is based upon the number of I/O signals plus storage bits that will require addressing.

System Partitioning

With 8 address bits plus WRITE, one quickly suspects that it is difficult to place a whole system on a single board. The next question is how should the system be partitioned between circuit boards? It seems advantageous to partition the system in two ways: by generic types of I/O devices, e.g. Triacs, Darlingtons, etc; or by “Feature Cards”; cards which can contain a small ROM and the I/O devices necessary to support a small optional function, such as pedestrian walk signals in a traffic controller. These possibilities will be discussed in turn.

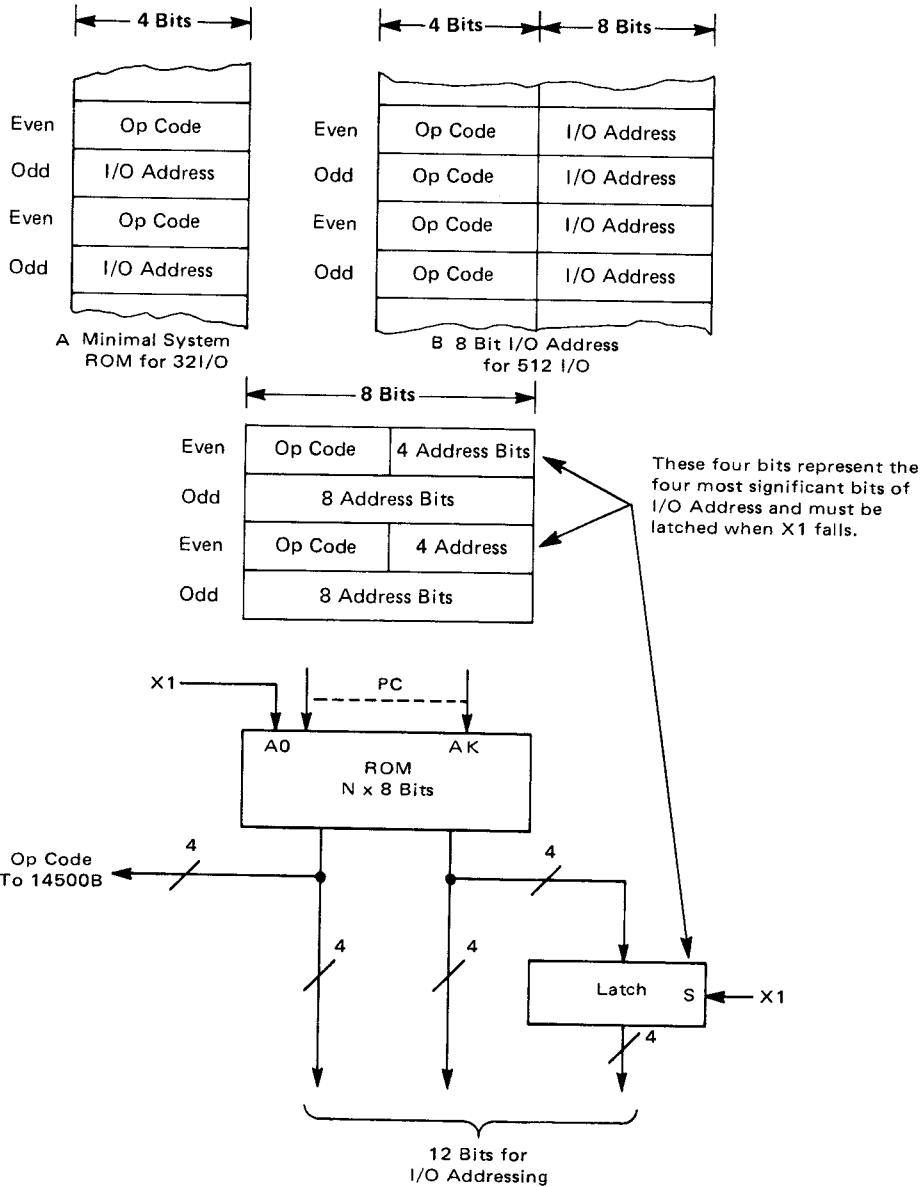


Figure 13.1 ROM Organizations for MC14500B System

I/O Cards by Circuit Type

Some of the different type devices one will use in different applications are input isolators, output opto isolators, Darlington and saturated switch drivers, LED drivers and SCR's or Triacs. Circuits for these different device types are described in the Interface Circuits section in Chapter 6. Here, we are concerned with enabling cards in an efficient manner.

Figure 13.2 shows a scheme for decoding board/chip enables for a system with 8 I/O address bits. The drawing shows all the lines, except Data, that need be bussed to I/O boards in a system. A Board Enable signal, BE, activates a group of 16 inputs or outputs. A_3 is used to split the block into groups of 8, or to the device level. The A_0 to A_2 lines are used by all I/O devices to identify 1 of 8 bits. An Input Board for such a system is shown on Figure 13.3 and an Output Board is on Figure 13.4.

To make such a system practical, one wants a means of interfacing the CMOS bus to the "real world." Figure 13.5 shows the normal card edge split to accept two edge connectors. The system signals travel on a mother board to the small board-mounted edge connector on the left. The second edge connector is connected to a wire bundle tied to the system's connection to the outside world, such as a barrier strip. The signal conditioning could be any or all of the methods described in Chapter 6.

The LED status bit indicators are not detailed, as their design is common and straightforward. The convenience of the bit indicators, their low cost and the common usage of their feature suggest they should be considered for any modular system design.

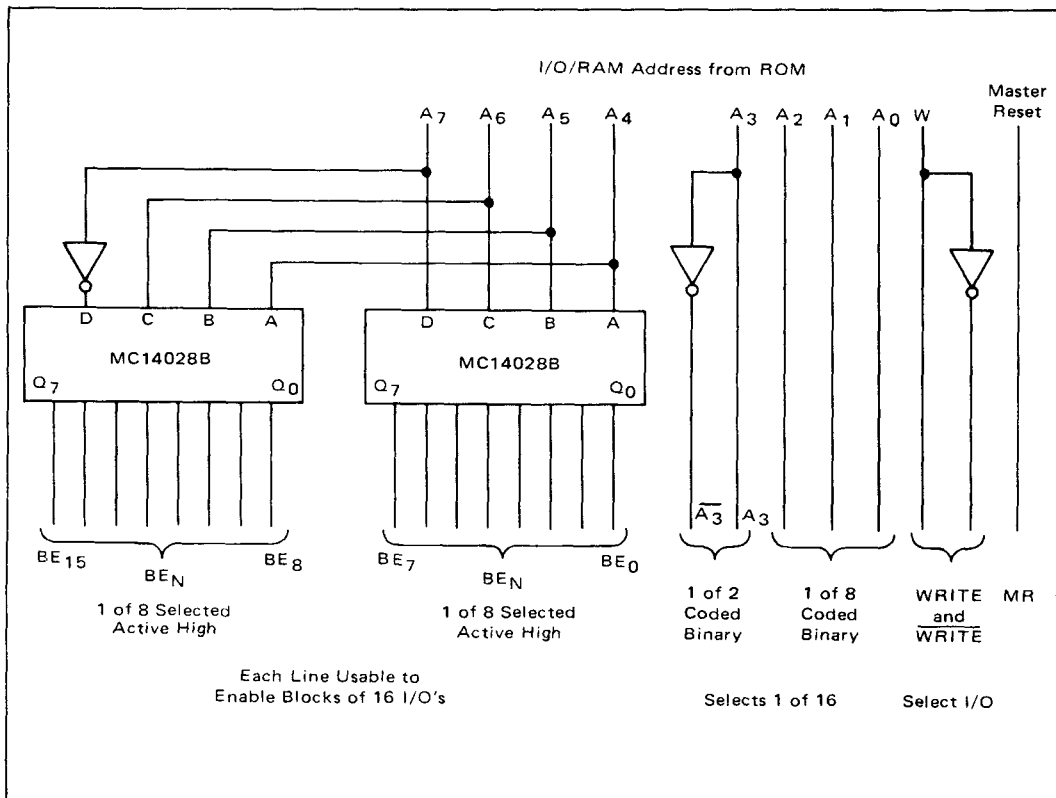


Figure 13.2 I/O Card Enables for 8 Bit Address

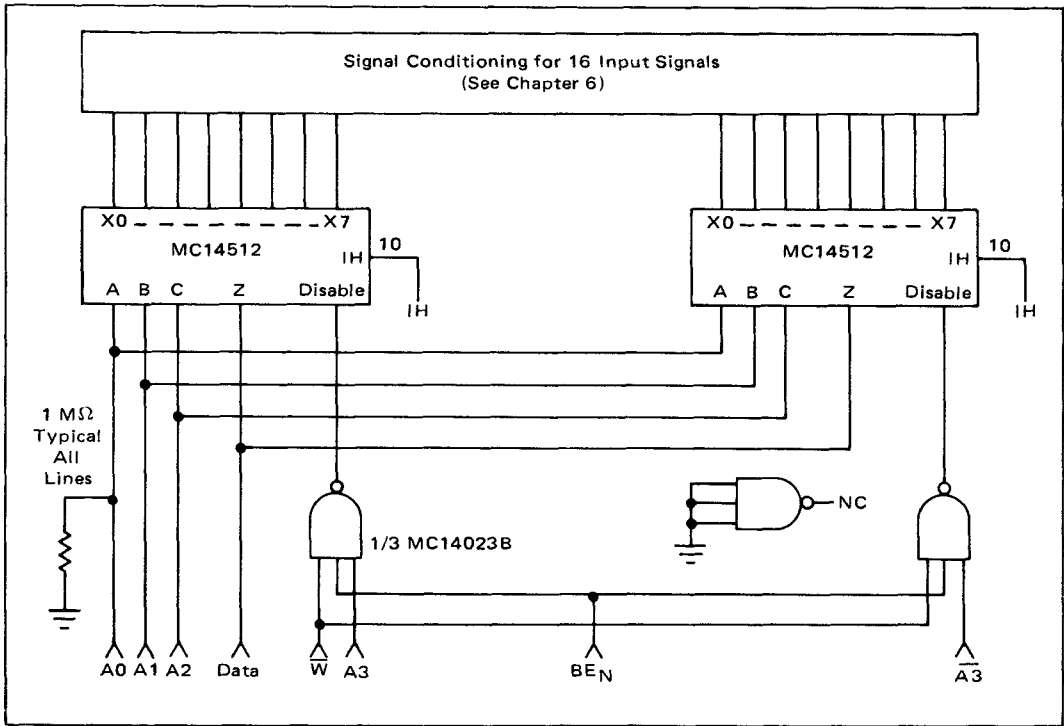


Figure 13.3 A 16 Input Board

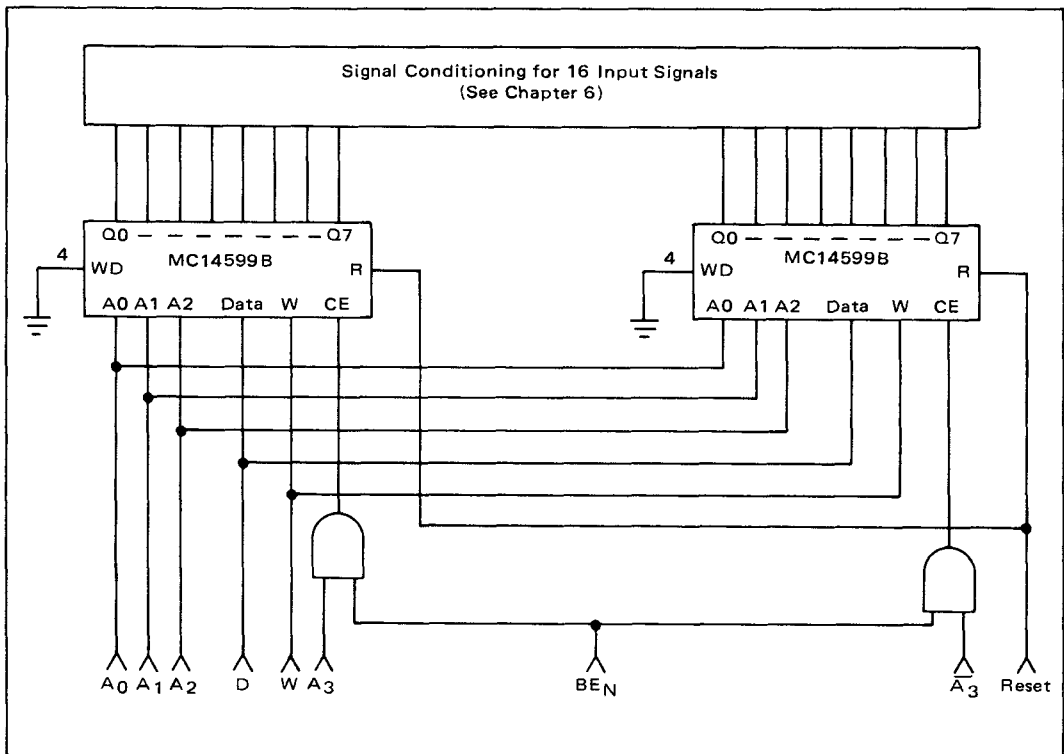


Figure 13.4 A 16 Output Board

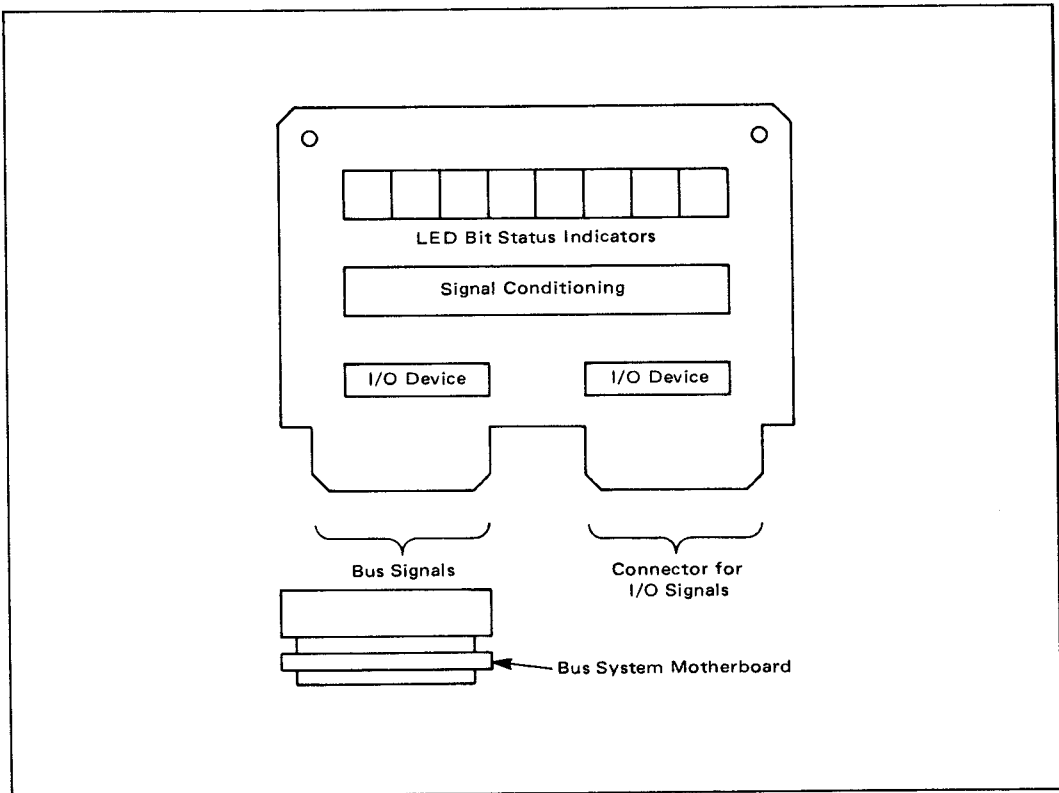


Figure 13.5 Possible Mechanical I/O Card Arrangement

Feature Cards

The MC14500B ICU was defined in such a way that ROM could be disabled, (or not present), in a system and the “missing instructions” would be interpreted as NOP’s (code 0 or F). This assumes the instruction lines do not “float,” but are tied to +V or ground through high value (> 100k) resistors. This provides for another way to modularize an ICU system. ROM can be placed on a card together with the I/O devices required to perform a function. The ROM is addressed from the central program counter and enabled by an enable decoder.

If the “feature card” is installed in the system, the feature card’s ROM is enabled during some interval of the program count and the ROM controls the system. All other ROM’s in the system are, of course, disabled at this time. If the feature card is missing from the system, the program counter increments through the states assigned to the feature’s program, but receiving no instructions, the ICU does “NOP’s” until some ROM that is in the system furnishes the ICU with instruction codes. The only restriction to the use of a feature card is that of “Jumping” the program counter off the feature ROM’s enabled block. Users who write such a jumping command must therefore exactly understand the implications of their code.

CHAPTER 14 ARITHMETIC ROUTINES

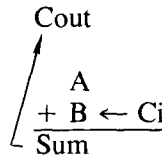
Occasionally, in a decision oriented controller, some arithmetic may be required for timing, parts counting or part of the enabling routine for some control functions. A nucleus of arithmetic coding follows. Programs which do large amounts of arithmetic can be assembled by building with the listed routines.

Binary Addition

Binary addition is an operation involving five bits: two bits to be added or operands, carry-in and carry-out bits and a sum bit. About 12 operations are required to do a one bit add. Addition, as well as other more complex functions, can be sent to a companion microprocessor or calculator. For example, if addition were the only arithmetic function, relegating the task to a CMOS adder might be appropriate. If the percentage of processing time required for addition is small, it is generally more economical to do the task completely with the ICU system. This is an instance of effective usage of the ICU's sub routine capabilities.

The code for single bit add with carry follows.

Cout
Sum
A
+ B Cin



LD Cin
XNOR B
XNOR A
STO SUM

GENERATING THE SUM

$$S = A \oplus (B \oplus C)$$

$$= A \oplus (\overline{B \oplus C})$$

SIMILAR TO GENERATING PARITY

LD B
OR Cin
AND A
IEN B
OR Ci
STO CARRYout

$$Co = A \cdot B + A \cdot Ci + B \cdot Cin$$

$$= A \cdot (B + Cin) + B \cdot Cin$$

$$RR \leftarrow A \cdot (B + Cin)$$

ACTUALLY PERFORMS $B \cdot Cin$

$$RR = A \cdot (B + Cin) + B \cdot Cin \rightarrow Co$$

ORC RR
IEN RR

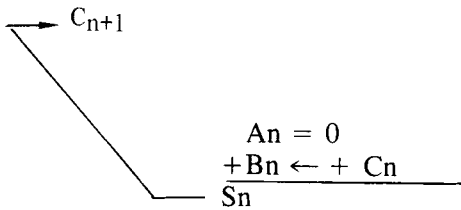
RESTORES THE IEN MASK

ONE BIT ADD WITH CARRY

Incrementation

Adding 1 to a stored number, or incrementing by 1, is perhaps the simplest and most common arithmetic function. It is used in parts counting, measuring frequency, etc.

In the code below we operate upon a single bit position at a time. For the Nth sum bit the variables' name is S_n . The carry in for the Nth S_n bit is denoted C_n . The carry out for the next bit position is denoted $(n+1)$. Notice that incrementing is analogous to forcing the initial carry in to 1 and adding zero to the number to be incremented. When the routine starts, Carry is set to 1 if the incrementation is to start. Otherwise, the initial value for Carry is 0.



$$S_n = B_n \oplus C_n$$

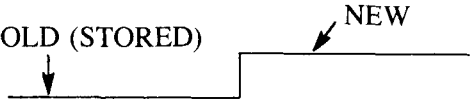
$$C_{n+1} = (B_n \oplus C_n) \cdot B_n$$

LD	B_n
XNOR	C_n
STOC	S_n
AND	B_n
STO	C_{n+1}

The routine is repeated N times for an N bit incrementation.

Counting Rising Edges

As a matter of practicality, counting rising signal edges is a simple and straightforward method of incrementing a sum.

OLD (STORED)	NEW	OLD (STORED)	
The code is:			
START	LD	NEW	
	XNOR	OLD	COMPARE OLD/NEW; 1 IF EQUAL
	OR	OLD	1 IF OLD WAS HIGH
	STOC	CARRY	CARRY ZERO IF NO RISING EDGE
	LD	NEW	
END	STO	OLD	PUT NEW IN OLD FOR NEXT TEST

Notice that NEW is sampled twice. To avoid this, use a Temp Store, e.g.

START	LD	NEW	
	STO	TEMP	
	ANDC	OLD	
	STO	CARRY	CARRY GETS RESULT
	LD	TEMP	AVOIDS 2nd SAMPLING
END	STO	OLD	

Magnitude Comparison

The Algorithm: Magnitude comparison compares two binary numbers to see which is greatest or if they are equal. Only three results are possible.

To compare two binary words, it is convenient to start with the most significant bits. In each bit position a comparison is made to see if the bits are identical. If they are, continue to the next bit position. If the bits are different, set EQUAL to 0 and set a flag indicating that the word with the 1 is greatest.

Three variables or flags are used, AGTR, BGTR and EQU. These correspond to A Greatest, B Greatest, and Equal. Initially set AGTR = 0, BGTR = 0 and EQ = 1.

Assume IEN = OEN = 1

START	ORC	RR	FORCE RR TO 1
	STO	EQ	INIT EQ
	STOC	AGTR	INIT AGTR
	STOC	BGTR	INIT BGTR
NTH BIT	OEN	EQ	ENABLE IF EQ = 1
	LD	AN	LOAD NTH A BIT
	XNOR	BN	COMPARE TO NTH B BIT
	STO	EQ	NEW VALUE TO EQ
	OR	AN	BGTR = EQ + AN
	STOC	BGTR	STORE NEW BGTR
	LD	EQ	LOAD EQ
	OR	BN	AGTR = EQ + AN
END NTH BIT	STOC	AGTR	STORE NEW AGTR
N-1 ST BIT	OEN	EQ	ENABLE IF EQ 1

REPEAT FOR EACH BIT POSITION

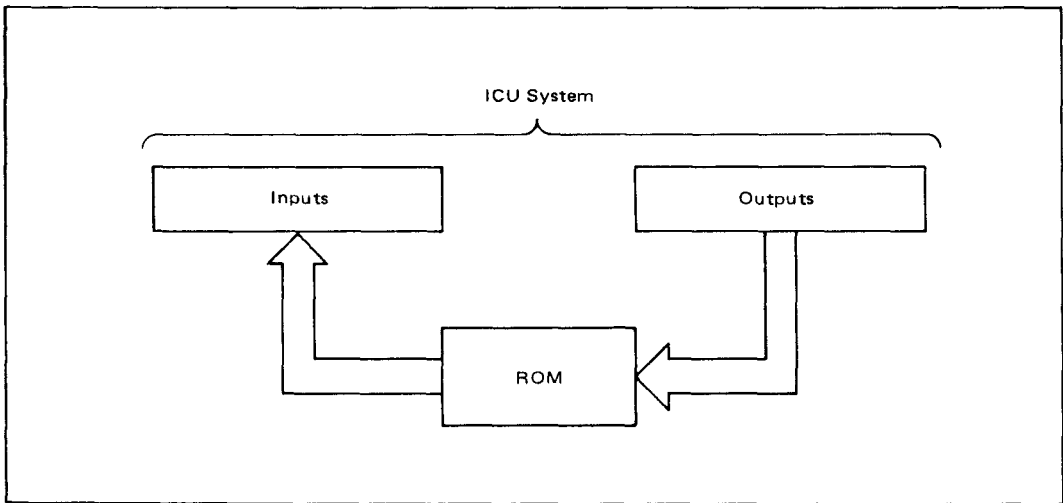


Figure 14.1 ROM for 1-Bit Add

Look-Up Tables

The processor overhead “expense” of a 1 bit add shows the need for a better implementation. One answer is a LOOK-UP TABLE as shown on Table 14.1. The operands and operator in an arithmetic expression are used as the address to a ROM. The ROM supplies the answer to the input pins in an ICU system.

As an example, a “1 bit ADD with Carry” will be examined. There are three operands — A, B and Carry-In; the operator is Add; the results are Carry-Out and Sum.

The ROM organization is summarized in Table 14.1. The binary addition of three single-bit operands can only result in $2^3 = 8$ possible outcomes. The sum and carry outputs of the ROM are simply the known results of any possible combination. The operator, ADD “vectors” (points) the ICU to the addition look-up table in system memory. The Look-Up ROM needs, at the most, 16 bits! The Look-Up Table idea can be extended to nearly any type function. Look-Up Tables for sine values, as an example, have long been standard semiconductor parts.

Table 14.1 The ROM Look-up Table

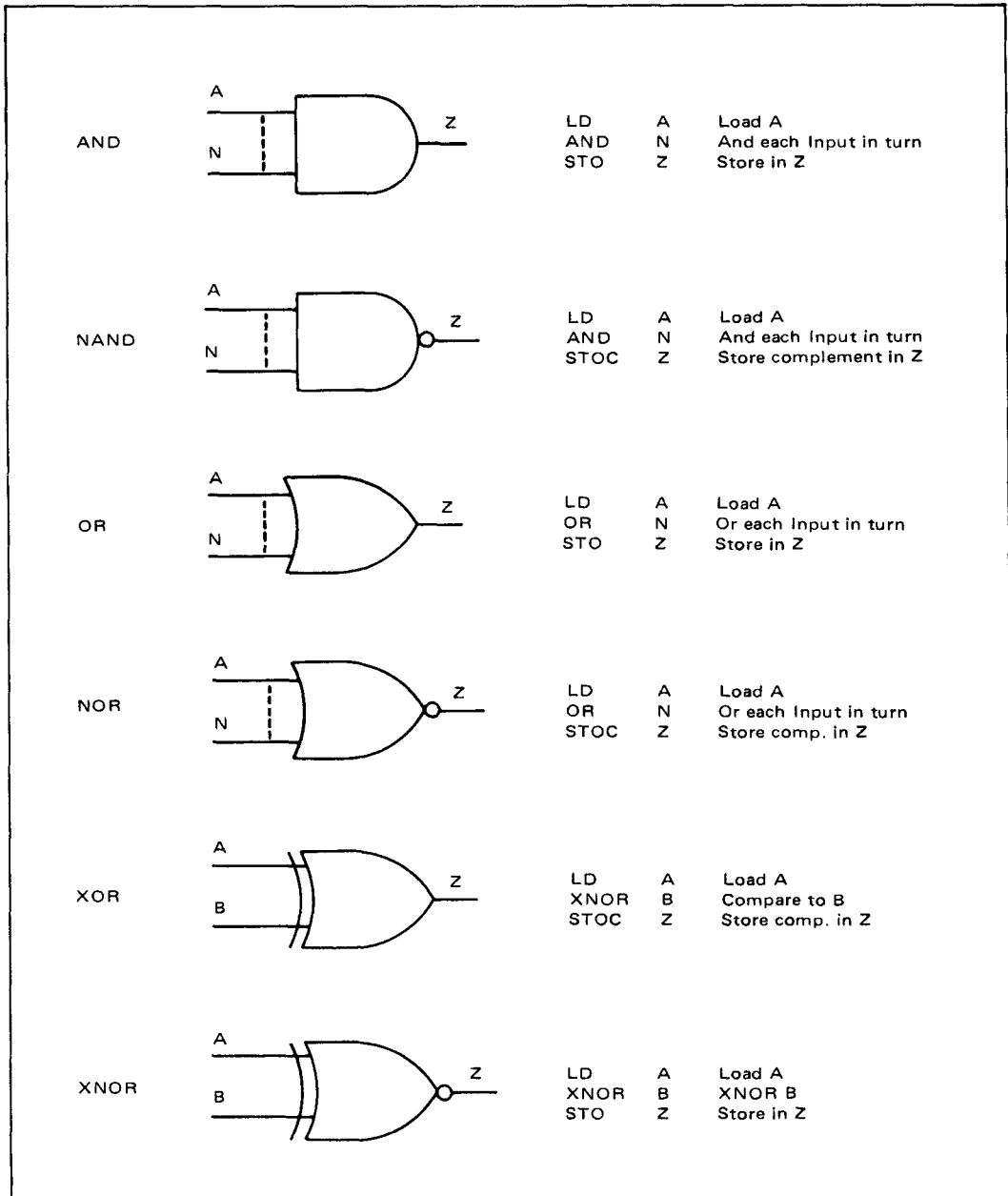
ADDRESS				DATA	
Operator (Add)	Operand (A)	Operand (B)	Operand (CI)	Result (Sum)	Result (CO)
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	1

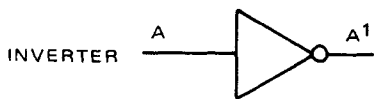
ROM Address ROM Content

Note that Operator (Add) = 0 could easily “vector” the ROM to a Subtract Table

CHAPTER 15 TRANSLATING ICU CODE

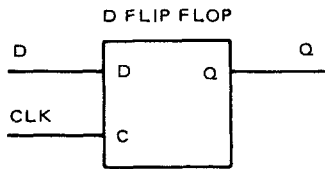
Repacing combinatorial logic with an ICU system is very simple and straightforward. All that is involved is the writing of the short codes which describe the logic devices. Logic functions and their associated codes are depicted in the following diagrams.





```
LD A      Load A
STOC A1   Store in A1
```

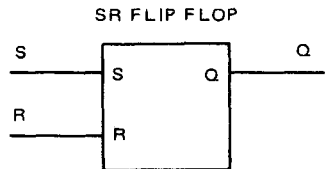
Notice: This code is never required as the ICU can load and store complements.



To clock on rising edges, clock is stored in old CLK to compare with current CLK.

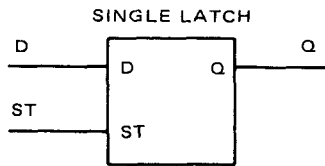
```
Start  LD      OLD CLK
        STO     TEMP
        LD      CLK
        STO     OLD CLK
        ANDC   TEMP
        OEN    RR
        LD      D
        ST     Q
        ORC   RR
End     OEN    RR
        RR     RESTORE OEN
        RR     IF NO Q CHANGE
```

$RR = CLK \cdot \overline{OLD\ CLK}$
ENABLE STORE



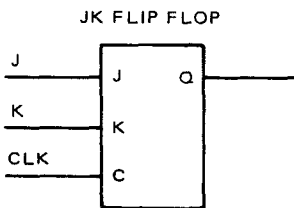
```
LD      S      LOADS
ANDC   R      AND WITH R
STO    Q
```

$$Q = S \cdot \overline{R}$$



```
LD      D      LOAD D
AND     ST     AND WITH STROBE
STO    TEMP   STORE IN TEMP
LD      Q      LOAD Q
ANDC   ST     AND WITH STROBE
OR     TEMP   OR WITH TEMP
STO    Q      STORE IN Q
```

$$Q = D \cdot ST + Q \cdot \overline{ST}$$



$Q_{n+1} = Q_n \cdot \overline{K} + \overline{Q}_n \cdot J$,
CLOCK ON RISING EDGES, CLOCK STORED IN OLD CLK.

```
Start  LD      OLD CLK  } MOVE OLD CLK
        STO     TEMP    } TO TEMP.
        LD      CLK     } FIND RISING
        STO     OLD CLK } EDGE.
        ANDC   TEMP    } ENABLE OUTPUT
        OEN    R       } IF EDGE FOUND.
        LD      Q       } AND Q WITH
        ANDC   K       } K COMP.
        STO     TEMP    } STORE IN TEMP.
        LDC    Q       } AND Q COMP.
        AND    J       } WITH J.
        OR     TEMP    } OR WITH Q \cdot K
        STO    Q       } STORE NEW Q.
        ORC   R       } RE ENABLE
End     OEN    R       } OUTPUTS.
```

Reducing Boolean Equations to ICU Code

The following procedure is a straightforward way of writing ICU Code for evaluating Boolean expressions. One temporary storage location, "TEMP", is used. It is generally possible to avoid the use of "TEMP", however, the code will not be as easy to read.

Procedure:

1. Reduce the Boolean expression. The result will be a "Sum of Products" form (e.g., $A \cdot B + C \cdot D \cdot E + \dots + X \cdot Y \cdot Z$) or a product of sums form (e.g., $(A + B) \cdot (C + D + E) \cdot \dots \cdot (X + Y + Z)$).
2. Use the Sum of Products Procedure or Product of Sums Procedure, both below.

Sum of Products Procedure

- A. Factor common terms from the Sum of Products Expression, giving an Expression in the form

$$J \cdot K \cdot L (A \cdot B \cdot C + D \cdot E + \dots + X \cdot Y \cdot Z).$$

The distributed term ($J \cdot K \cdot L$) which was factored from the Sum of Products form will be used as an "INPUT ENABLE TERM". That is, if the INPUT ENABLE TERM is not 1 or true, then everything following will be evaluated as 0 or FALSE.

- B. Evaluate the INPUT ENABLE TERM and store in INPUT ENABLE.

```

START   ORC   RR   SET RR TO 1
        IEN   RR   ENABLE INPUT
        LD    J    LOAD 1st ELEMENT
        AND   K    AND WITH NEXT
        .
        .
        AND   L    AND WITH LAST
END     IEN   RR   STORE RESULT in IEN
    
```

- C. Reduce the first INNER TERM and store in "TEMP".

```

START   LD    A    RR GETS A
        AND   B    AND WITH B
        AND   C    AND WITH C
END     STO   TEMP  STORE IN TEMP
    
```

- D. Reduce the next INNER TERM and/or with TEMP, store result in TEMP.

```

START   LD    D    RR GETS D
        AND   E    AND WITH E
        OR    TEMP
END     STO   TEMP
    
```

TEMP now has $A \cdot BC \cdot + DE$, providing $IEN = 1$. If $IEN = 0$, $TEMP = 0$.

- E. Repeat D. for all the remaining inner terms.

- F. The Sum of Products value is now in the Result Register and stored in TEMP. To unconditionally enable the ICU for other routines, restore IEN and OEN to the 1's state.

```

START   ORC   RR   RR GETS 1
        IEN   RR   IEN GETS 1
END     OEN   RR   OEN GETS 1
    
```

Product of Sums Procedure

- A. Factor common terms from the Produce of Sums form, giving an expression in the form

$$(J + K + L) (A + B + C) \cdot (D + E) \cdot \dots \cdot (X + Y + Z).$$

- B. The distributed term which was factored out will be used as an "INPUT ENABLE TERM".

```
START  LD    J      RR GETS J
        OR    K      OR WITH K
        OR    L      OR WITH L
END     IEN  RR     IEN GETS RR
```

- C. Reduce the first INNER TERM and store in "TEMP".

```
START  LD    A      RR GETS A
        OR    B      OR WITH B
        OR    C      OR WITH C
END     STO  TEMP   STORE IN TEMP
```

- D. Reduce the next INNER TERM, and with TEMP, store result in TEMP.

```
START  LD    D      RR GETS D
        OR    E      OR WITH E
        AND  TEMP
END     STO  TEMP
```

- E. Repeat D. for each of the other INNER TERMS.

- F. The evaluated product of sums is in RR and stored in TEMP. The following routine will completely enable the ICU for other uses.

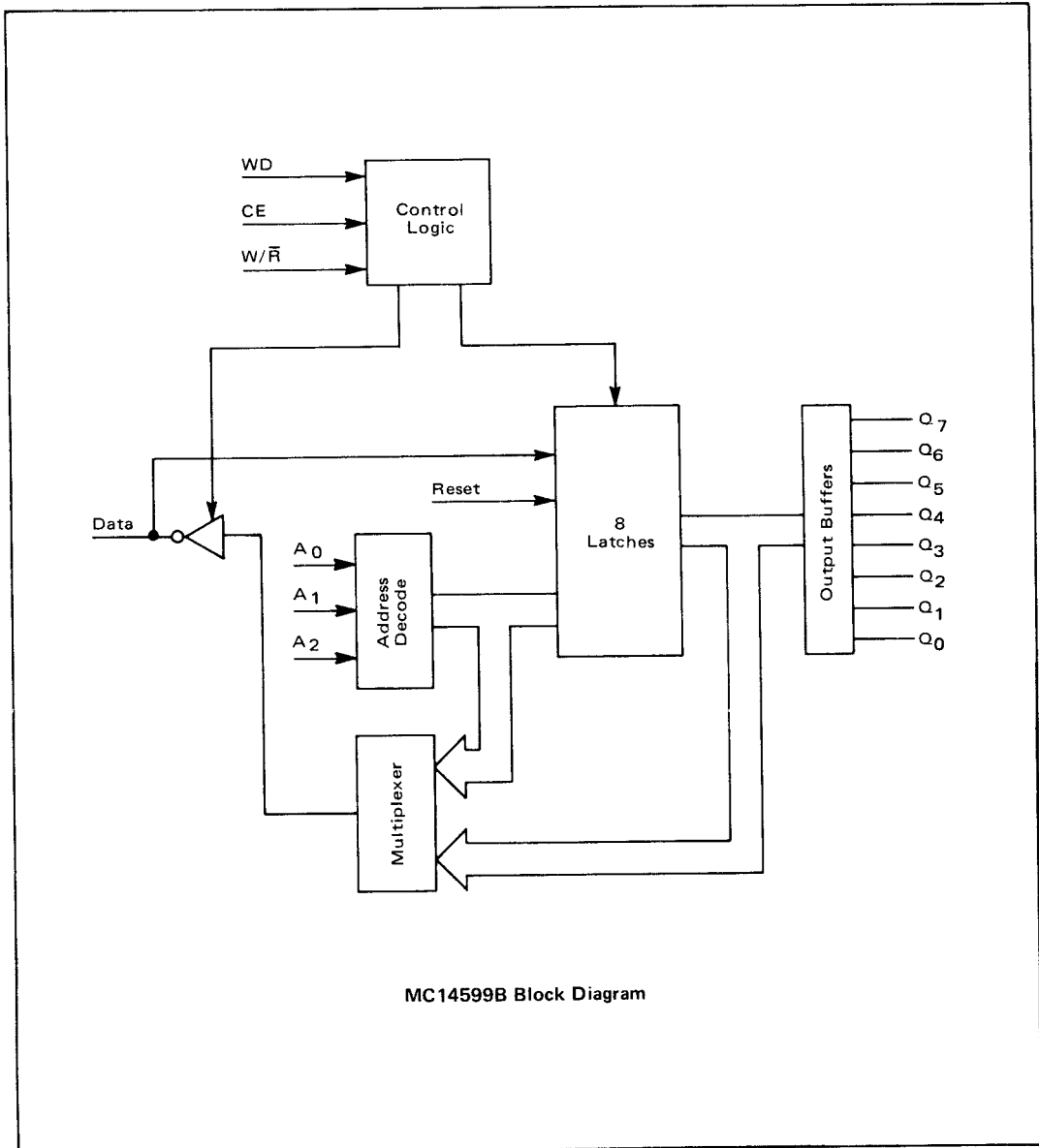
```
START  ORC  RR      RR GETS 1
        IEN  RR      IEN = 1
END     OEN  RR      OEN = 1
```

APPENDIX A. THE MC14599B 8-BIT ADDRESSABLE LATCH

The MC14599B is an 8 bit addressable latch capable of reading previously stored data. The device has a chip enable input for easy address expansion, buffered outputs, and a master reset pin for system clears.

Features

- * Parallel Buffered Output
- * Bidirectional Addressable Input/Output
- * Master Reset
- * WRITE/READ Control
- * Write Disable
- * Chip Enable
- * B Series CMOS



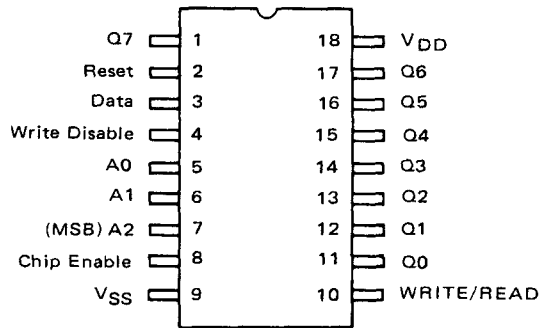
MC14599B Truth Table						
Inputs				Internal States & Data		
R	CE	WD	W	Addressed Latch	Other Latches	Data Pin
1	X	X	X	O	O	Z
0	0	X	X	NC	NC	Z
0	1	X	0	NC	NC	Q _N (Output)
0	1	1	1	NC	NC	Z
0	1	0	1	Data	NC	Input

X = Don't Care

NC = No Change

Z = Open Circuit

Q_N = State of Addressed Cell



MC14599B

