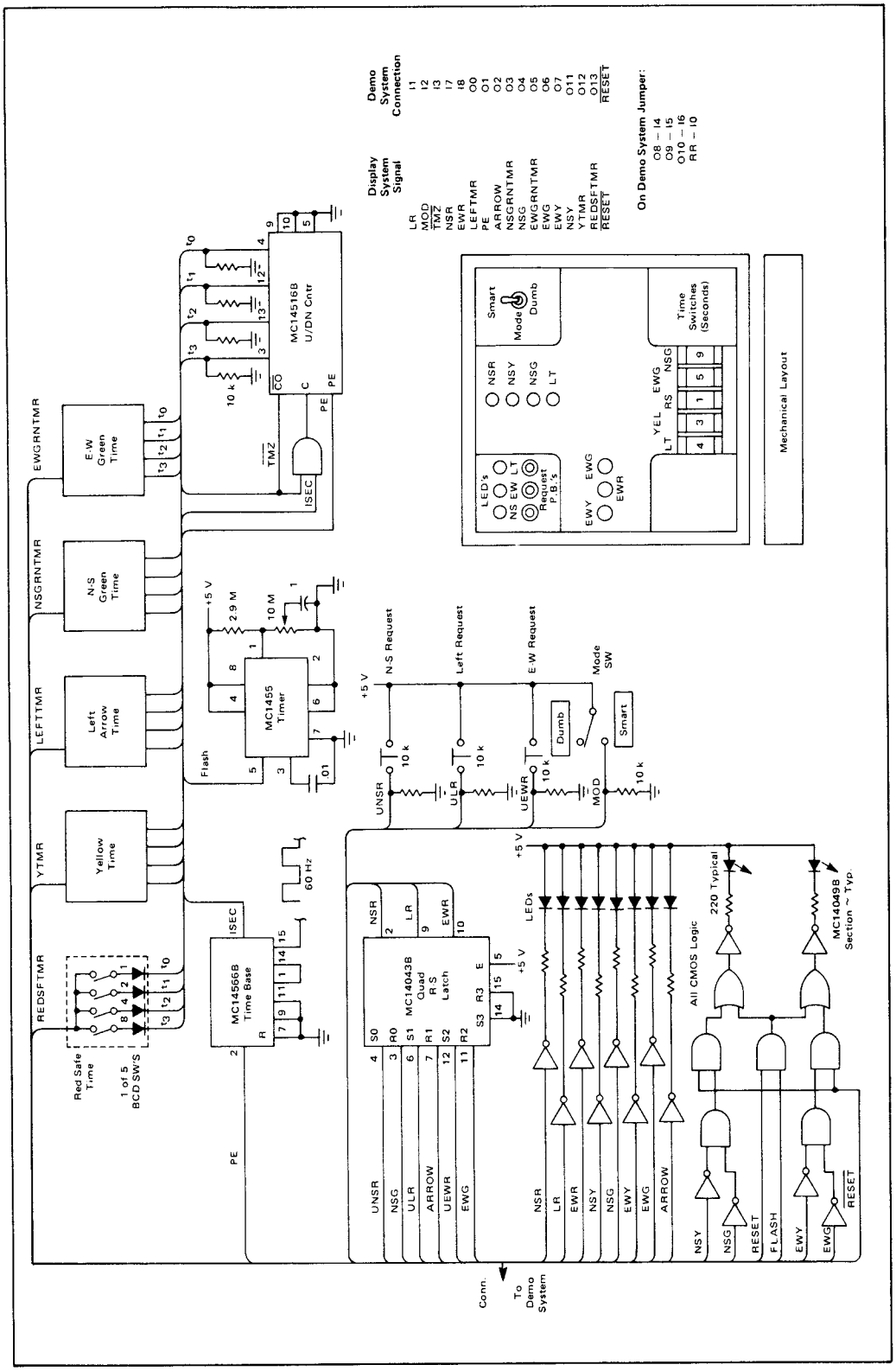


Figure 11.4 Traffic Intersection Controller

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.

SKIP IF RR = 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

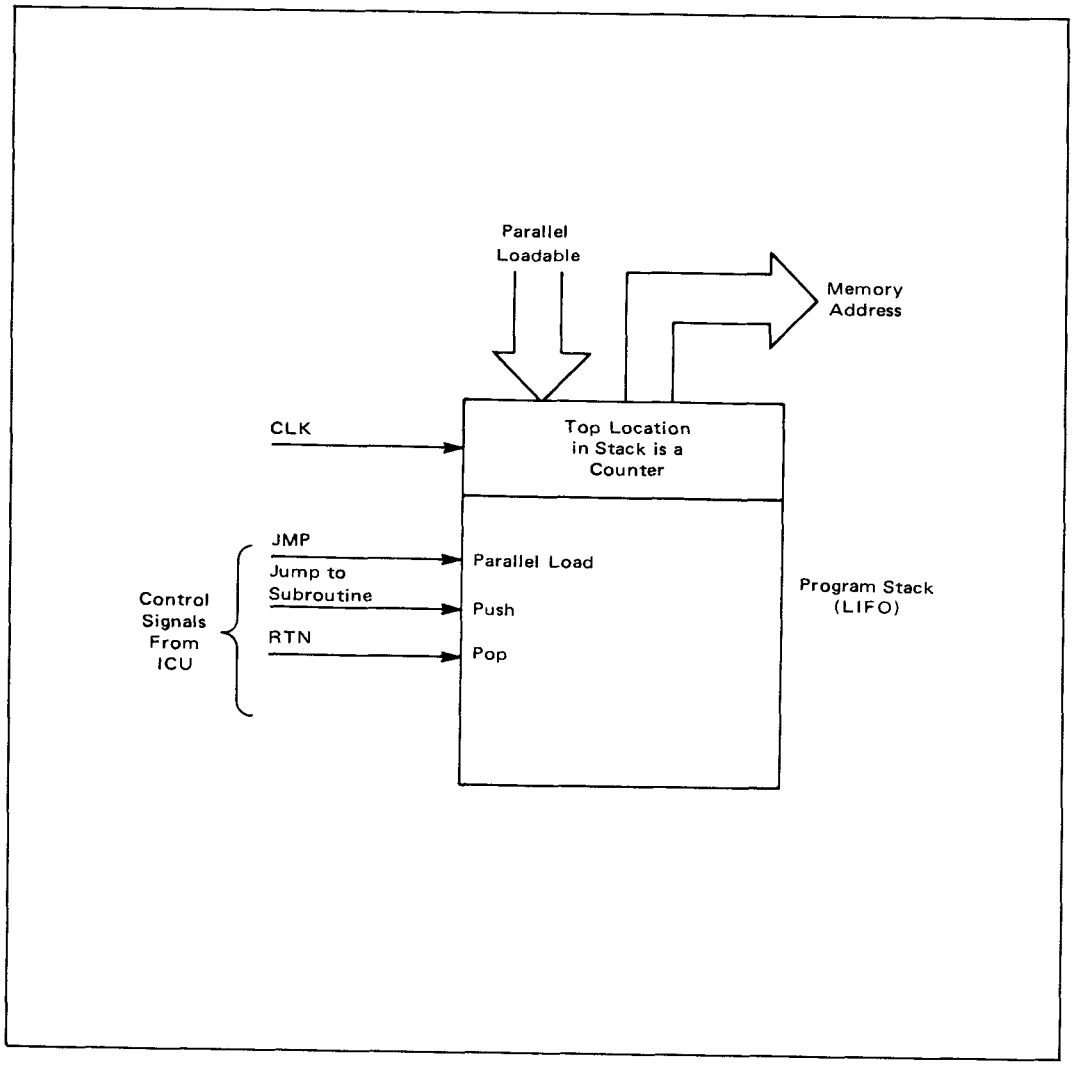


Figure 12.2 Block Diagram of Subroutine Structure

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.

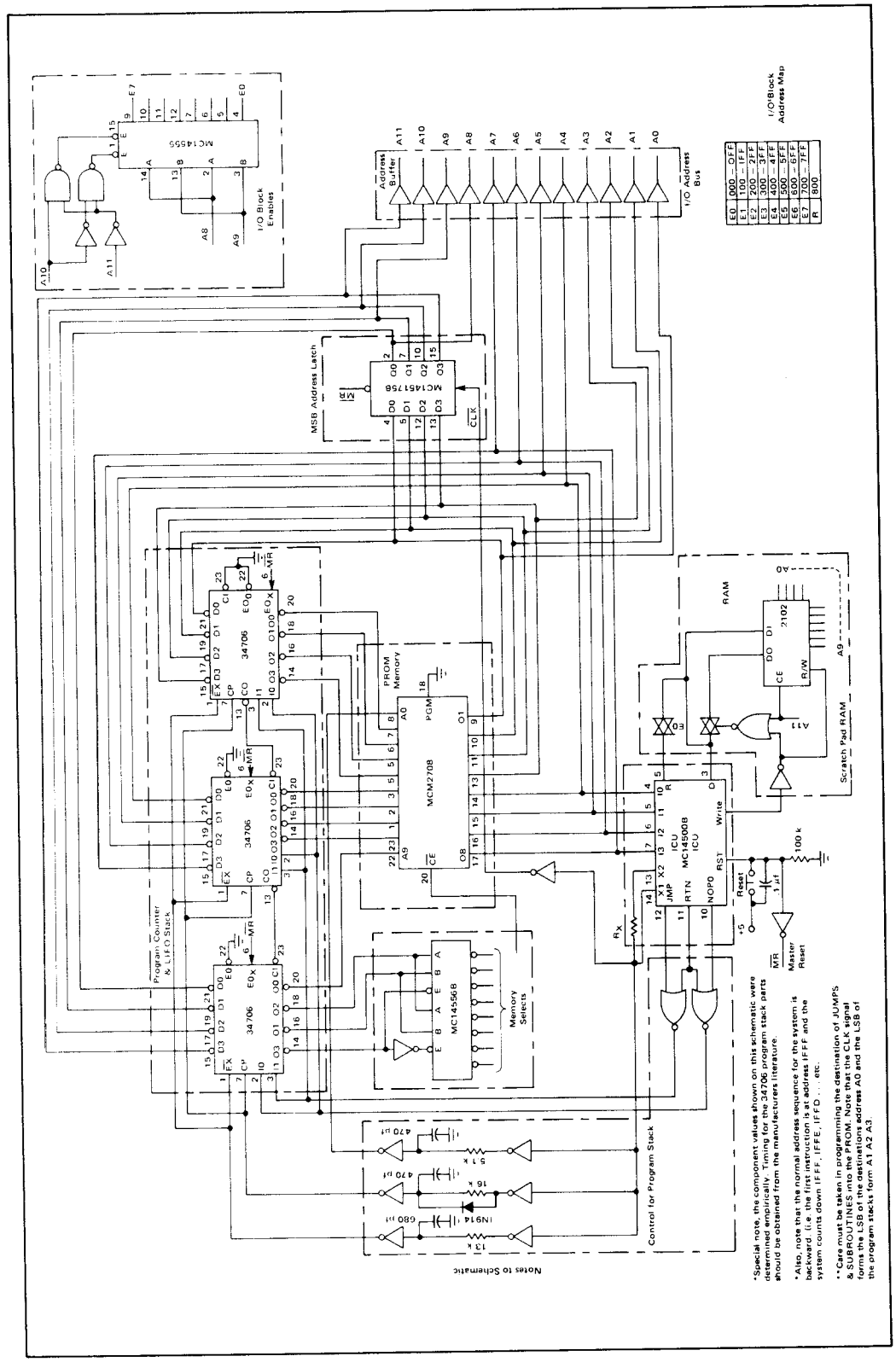


Figure 12.3 ICU Based System with Subroutine Capability

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.

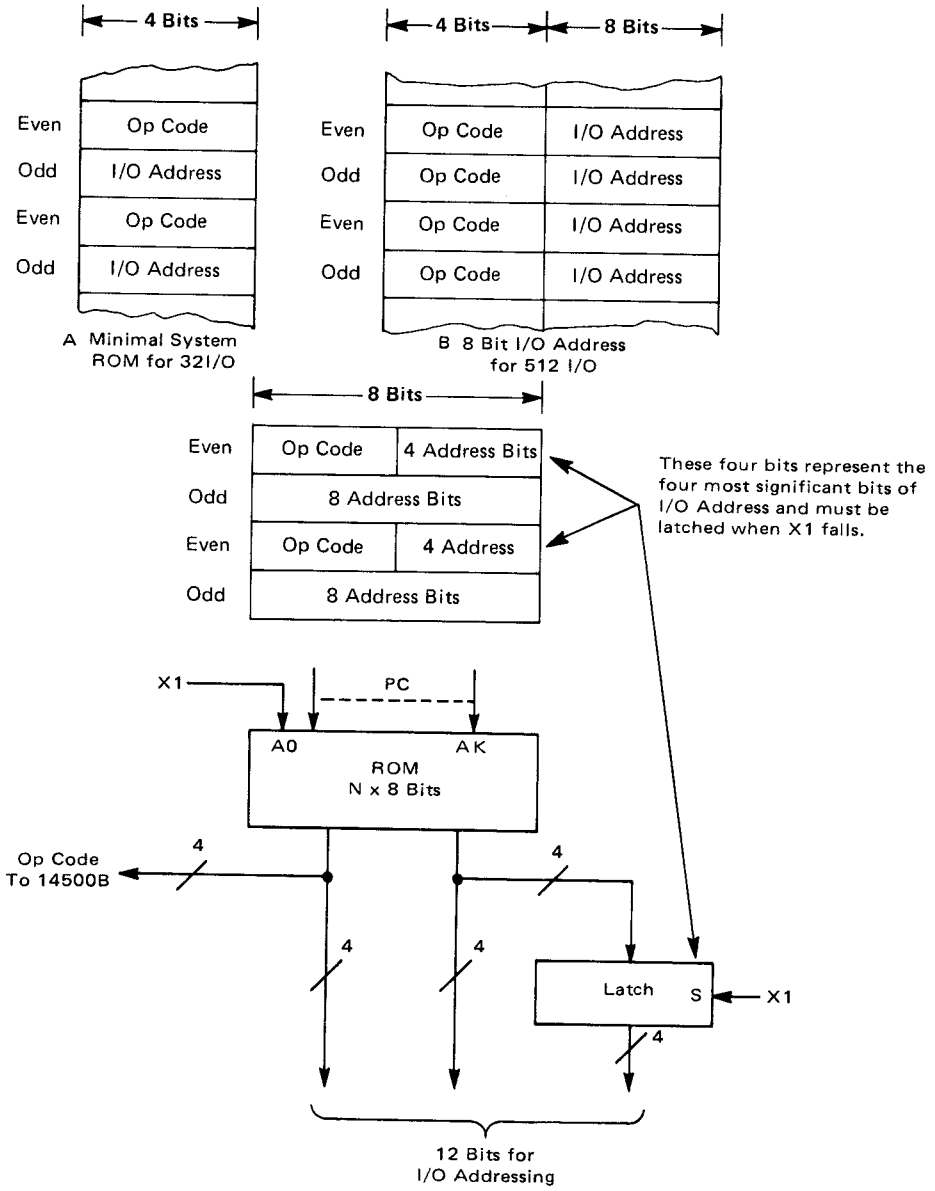


Figure 13.1 ROM Organizations for MC14500B System

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.

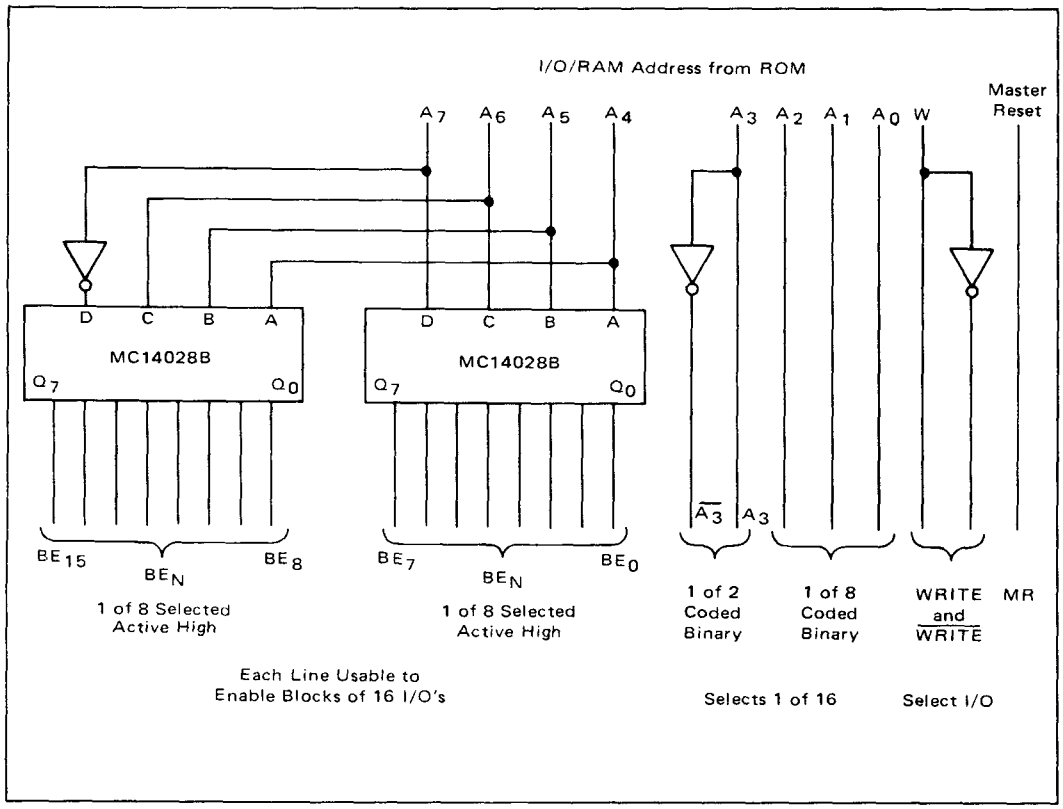


Figure 13.2 I/O Card Enables for 8 Bit Address

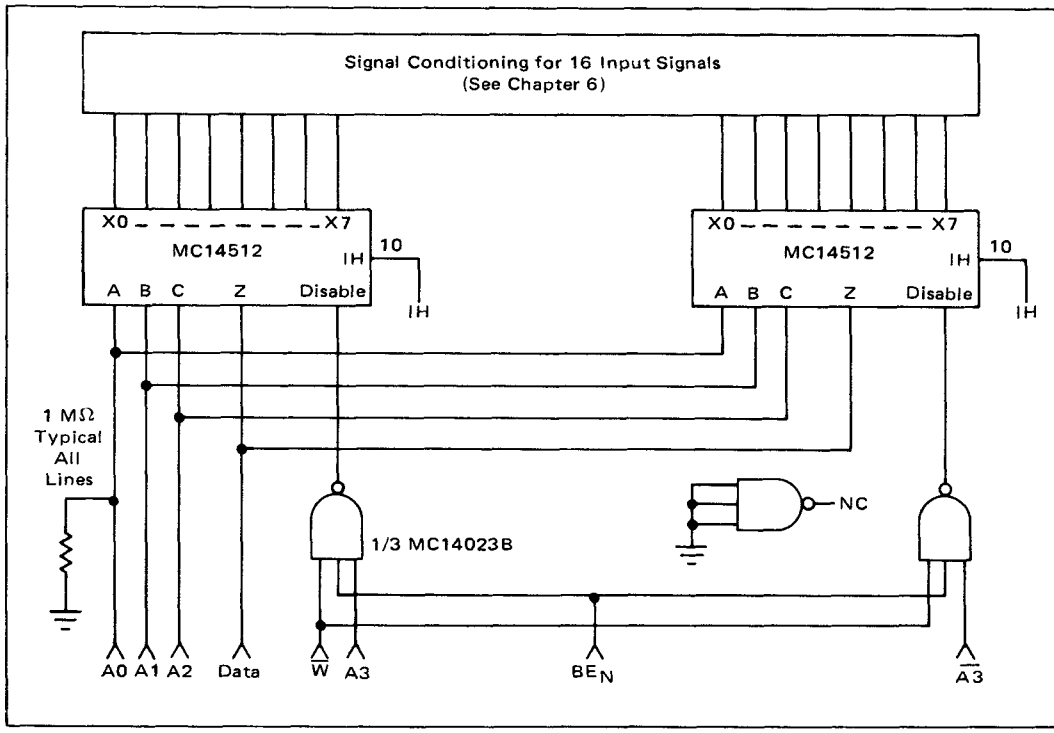


Figure 13.3 A 16 Input Board

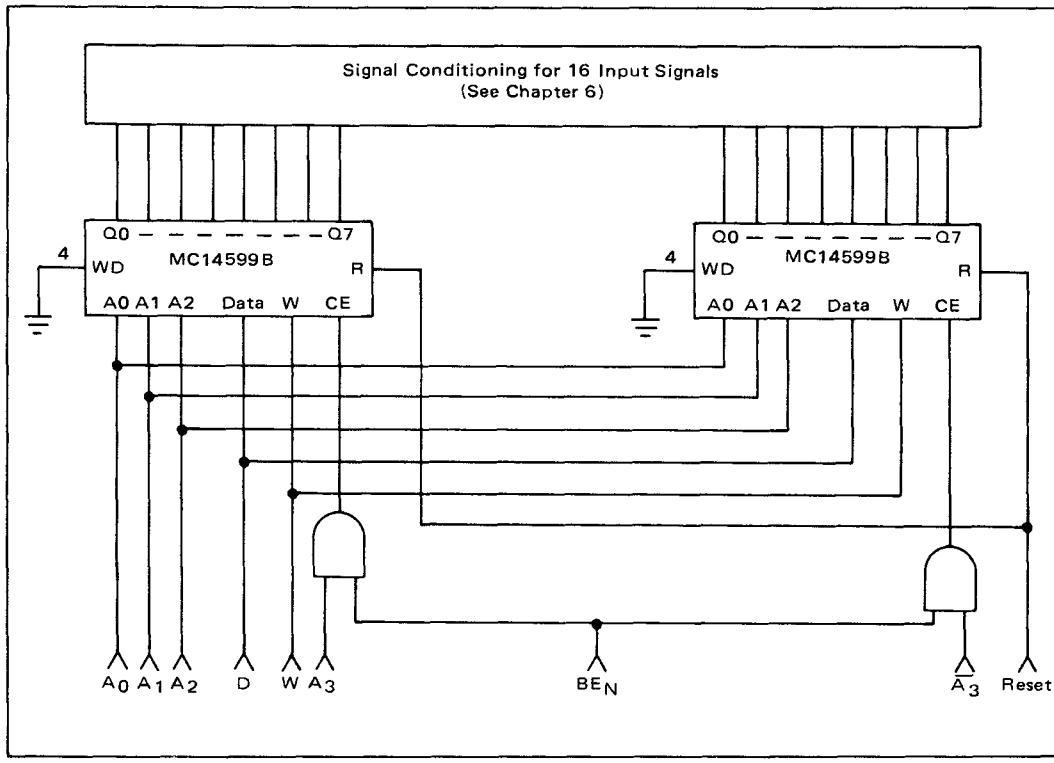


Figure 13.4 A 16 Output Board

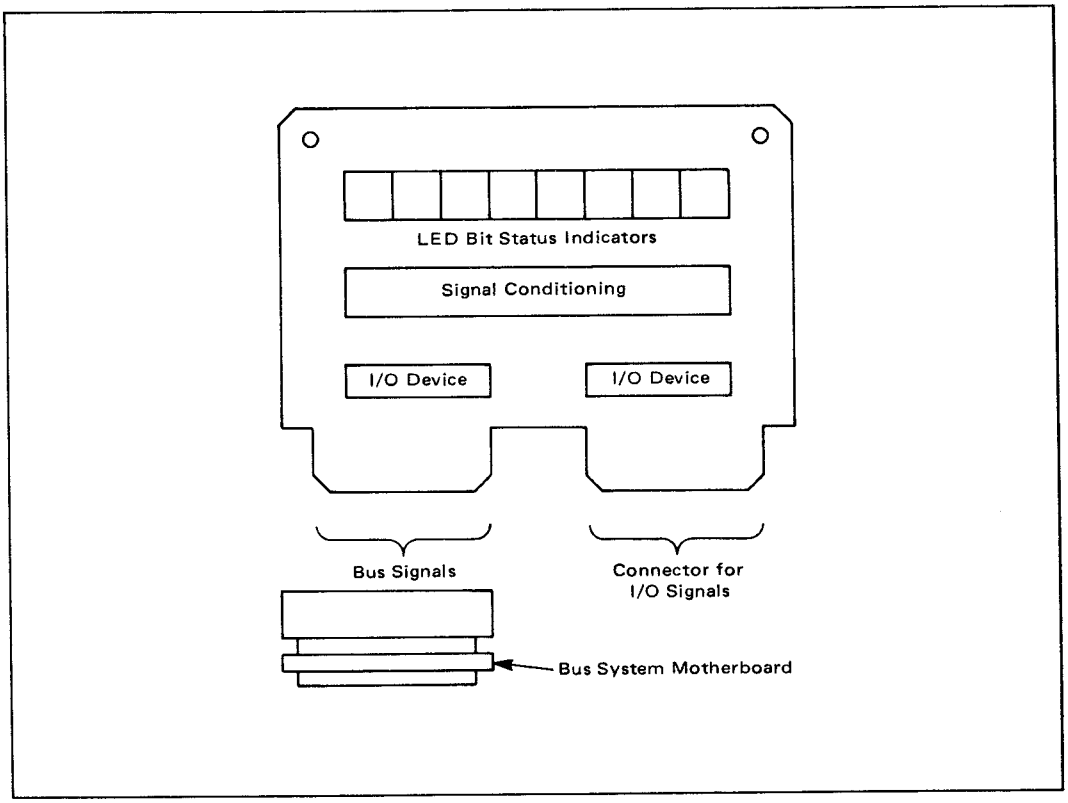


Figure 13.5 Possible Mechanical I/O Card Arrangement

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.

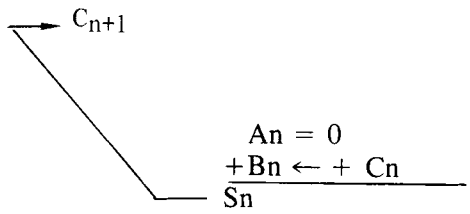
LD	Cin	GENERATING THE SUM
XNOR	B	$S = A \oplus (B \oplus C)$
XNOR	A	$= A \oplus (\overline{B \oplus C})$
STO	SUM	SIMILAR TO GENERATING PARITY
LD	B	$Co = A \cdot B + A \cdot Ci + B \cdot Cin$
OR	Cin	$= A \cdot (B + Cin) + B \cdot Cin$
AND	A	$RR \leftarrow A \cdot (B + Cin)$
IEN	B	ACTUALLY PERFORMS $B \cdot Cin$
OR	Ci	
STO	CARRYout	$RR = A \cdot (B + Cin) + B \cdot Cin \rightarrow Co$
ORC	RR	RESTORES THE IEN MASK
IEN	RR	

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 Sn. The carry in for the Nth Sn bit is denoted Cn. 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.



$$Sn = \overline{Bn \oplus Cn}$$
$$Cn+1 = \overline{(Bn \oplus Cn)} \cdot Bn$$

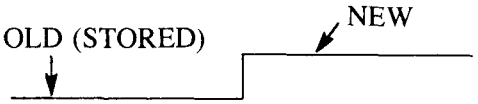
```
LD      Bn
XNOR    Cn
STOC    Sn
AND     Bn
STO     Cn+1
```

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



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
```

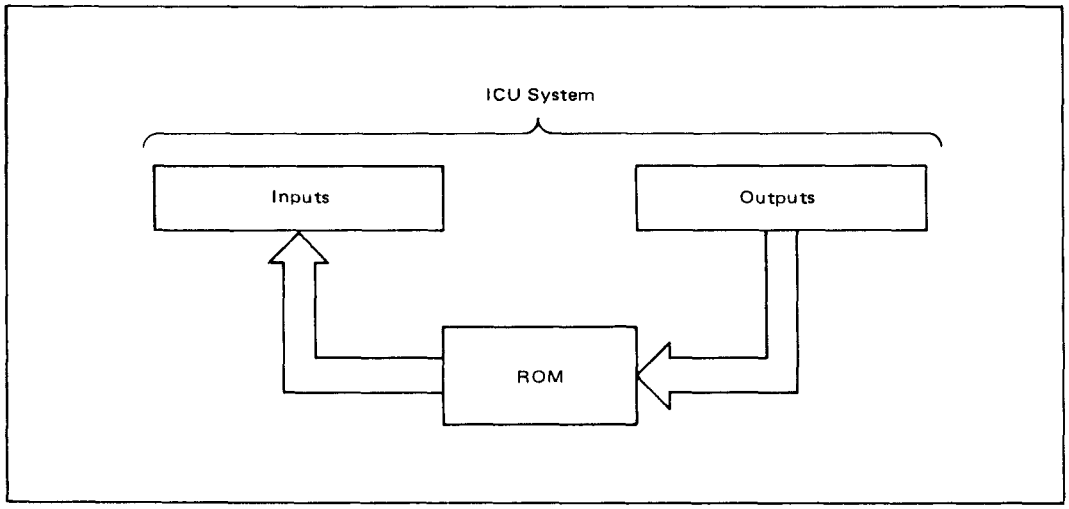


Figure 14.1 ROM for 1-Bit Add

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

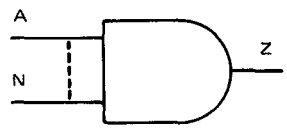
ADDRESS				DATA	
Operator (Add)	Operand (A)	Operand (B)	Operand (CI)	Result (Sum)	Result (CO)
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	1
ROM Address				ROM Content	

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

CHAPTER 15 TRANSLATING ICU CODE

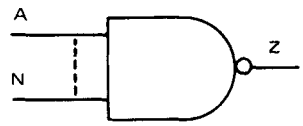
Repacing combinatorial logic with an ICU system is very simple and straightforward. All that is involved is the writing of the short codes which describe the logic devices. Logic functions and their associated codes are depicted in the following diagrams.

AND



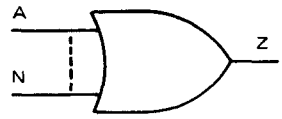

```
LD      A     Load A
AND     N     And each Input in turn
STO     Z     Store in Z
```

NAND



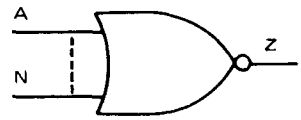

```
LD      A     Load A
AND     N     And each Input in turn
STOC    Z     Store complement in Z
```

OR



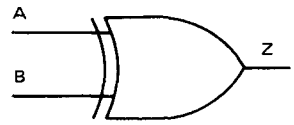

```
LD      A     Load A
OR      N     Or each Input in turn
STO     Z     Store in Z
```

NOR



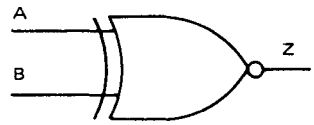

```
LD      A     Load A
OR      N     Or each Input in turn
STOC    Z     Store comp. in Z
```

XOR



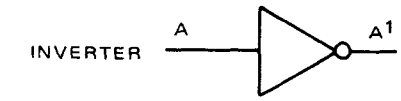

```
LD      A     Load A
XNOR    B     Compare to B
STOC    Z     Store comp. in Z
```

XNOR




```
LD      A     Load A
XNOR    B     XNOR B
STO     Z     Store in Z
```

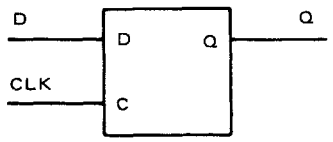
INVERTER




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

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

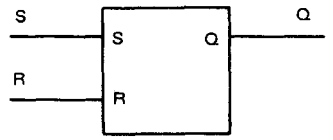
D FLIP FLOP



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        RR = CLK • OLD CLK
        OEN     RR          ENABLE STORE
        LD      D
        ST      Q
        ORC     RR          RESTORE OEN
End     OEN     RR          IF NO Q CHANGE
```

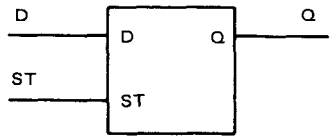
SR FLIP FLOP



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

```
LD      S       LOAD S
ANDC    R       AND WITH R̄
STO     Q
```

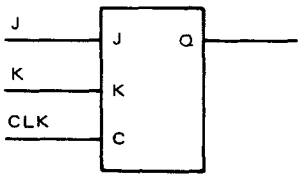
SINGLE LATCH



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

```
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
```

JK FLIP FLOP



$Q_{n+1} = Q_n \cdot \overline{K} + \overline{Q}_n \cdot J$,
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 • K̄
        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., $A \cdot B + C \cdot D \cdot E + \cdots + X \cdot Y \cdot Z$) or a product of sums form (e.g., $(A + B) \cdot (C + D + E) \cdot \ldots \cdot (X + Y + Z)$.
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

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

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

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

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.

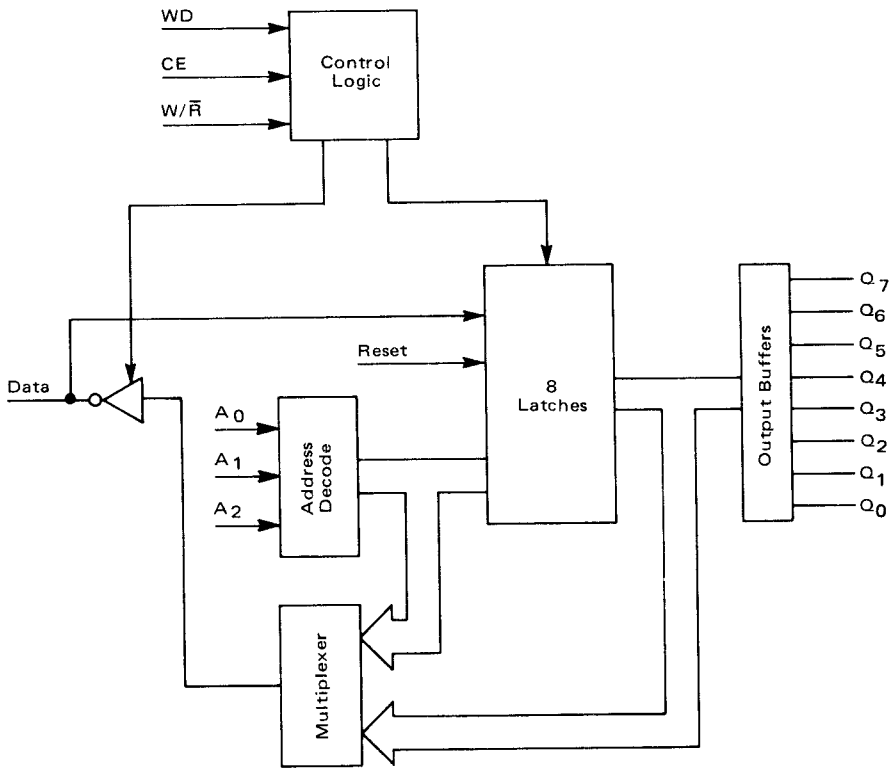
```
START   ORC   RR     RR GETS 1
        IEN   RR     IEN = 1
END     OEN   RR     OEN = 1
```

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/$\overline{\text{READ}}$ Control
* Write Disable
* Chip Enable
* B Series CMOS



MC14599B Block Diagram

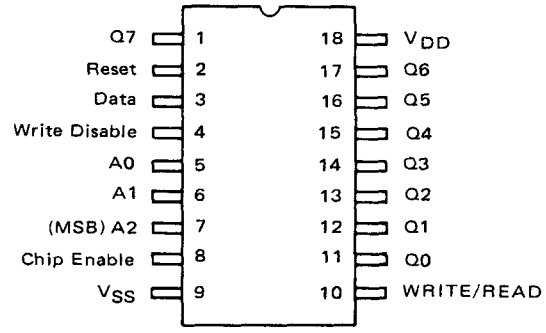
MC14599B Truth Table						
Inputs				Internal States & Data		
R	CE	WD	W	Addressed Latch	Other Latches	Data Pin
1	X	X	X	0	0	Z
0	0	X	X	NC	NC	Z
0	1	X	0	NC	NC	Q_N (Output)
0	1	1	1	NC	NC	Z
0	1	0	1	Data	NC	Input

X = Don't Care

NC = No Change

Z = Open Circuit

Q_N = State of Addressed Cell



MC14599B

Printed in Switzerland 33-B78/8.0