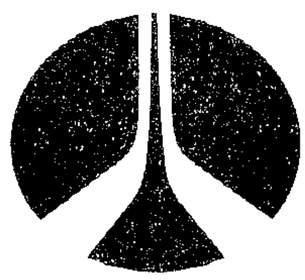
ROCKWELL PARALLEL PROCESSING SYSTEM (PPS)

PPS-4/1 ONE-CHIP MICROCOMPUTERS

MM76 MICROCOMPUTER PROGRAMMING MANUAL



Rockwell International

$ 5.00

Document No. 29410 N44
November 1977

PPS SUPPORTING DOCUMENTATION

The following documents provide additional PPS information for aiding in the understanding and implementation of your system:

- PPS-4/1 MM76 Product Description – Document No. 29410 N41
- PPS-4/1 MM76C Data Sheet – Document No. 29000 D46

LOCAL REPRESENTATIVE

MICROELECTRONIC DEVICES - REGIONAL SALES OFFICES

WESTERN REGION, U.S.A
3310 Miraloma Avenue
P.O. Box 3669
Anaheim, Ca. 92803
Phone: (714) 632-3698

EASTERN REGION, U.S.A.
Carolier Office Building
850-870 U.S. Route 1
North Brunswick, New Jersey 08902
Phone: (201) 246-3630

MIDWEST REGION, U.S.A
Contact:
John G. Twist Company
1301 Higgins Road
Elk Grove Village, Illinois 60007
Phone: (312) 593-0200

CENTRAL REGION, U.S.A.
2855 Coolidge Road, Suite 101
Troy, Michigan 48084
Phone: (313) 435-1638

FAR EAST
Rockwell International Overseas Corp.
Ichiban-cho Central Building
22-1 Ichiban-cho, Chiyoda-ku
Tokyo 102, Japan
Phone: 265-8808

EUROPE
Rockwell International GmbH
Microelectronic Device Division
Fraunhoferstrasse 11
D-8033 Munchen-Martinsried
Germany
Phone: (089) 859-9575

Rockwell International

HOME OFFICE SALES/MARKETING (714) 632-3729 ---------- *FOR ASSISTANCE, CALL OR WRITE THE OFFICE NEAREST YOU.*

INTRODUCTION

This manual provides the programming concepts and a description of techniques for application of the PPS-4/1 MM76 series of Microcomputers including software and hardware examples. Each instruction is described in detail and examples of its usage are shown by means of example routines and typical applications. All of the various versions of the MM76, including the MM75, use the same basic instruction set and programming disciplines.

Throughout the manual, extensive use is made of computer design aids for PPS-4/1 program assembly, source program editing, and input program emulation, simulation, and output control. These software design aids are available to the user for purchase in the form of PPS Universal Assemulator with the appropriate PPS-4/1 Personality Module. The assembly functions are available via the General Electric Information Services or as a FORTRAN IV source program for the IBM 360 or 370 computers. The development of a system using a PPS-4/1 Microcomputer is shown in the PPS Development Cycle figure. This manual is intended to be particularly useful in the functions shown in Steps 2, 3, and 4 of the PPS development cycle. It is this definition process which translates the requirements of the system as defined by the equipment functional specification, to a PPS-4/1 Microcomputer.

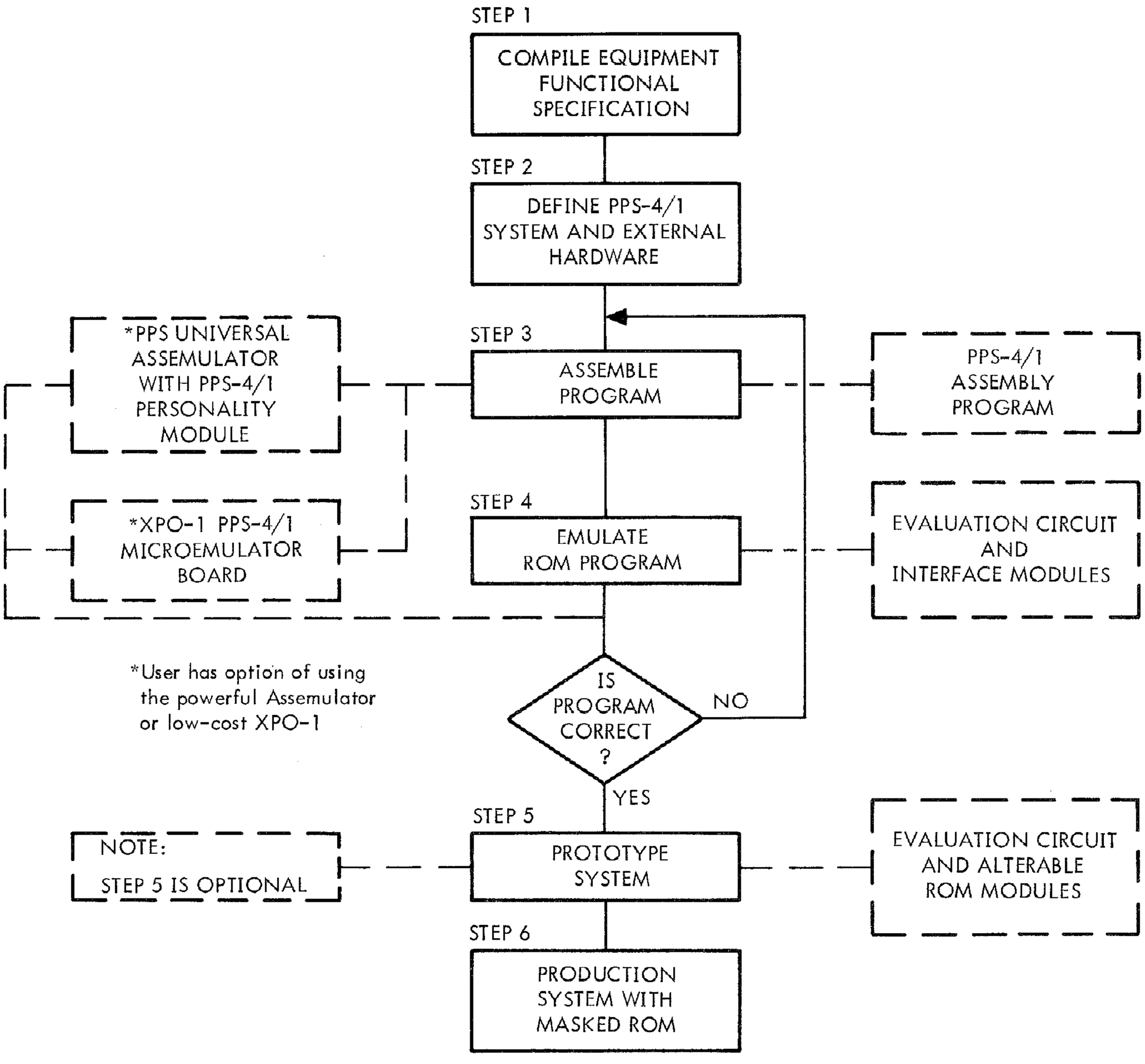
After the PPS-4/1 Microcomputer system is defined, the system designers have several options for completing the development of their system. The PPS-4/1 program may be assembled and checked out by the simulation program either by means of one of the timesharing systems or by means of an in-house computer system using the FORTRAN IV source program. Another option is the purchase of a Rockwell PPS Universal Assemulator which, in addition to doing the assembly and simulation functions, can also be used for the next step in the development of the user's system — emulation when the appropriate Personality Module is plugged in. During the emulation phase, the software and the hardware interfaces may be checked to be sure that all of the timing, logic levels, and other specific requirements are accomplished satisfactorily. The PPS Universal Assemulator interfaces with a TTY ASR 33 or a TI 733-ASR terminal. The Rockwell PPS Universal Assemulator also has an integral keyboard, display, and optional high-speed tape reader. The Assemulator with the Personality Module provides assembly, real-time emulation, and real-time debugging capability. It also provides ease of program modification and single-step instruction execution. The assembler in the Assemulator is nearly as sophisticated as the assembly program on the timesharing systems. Therefore, the ideal development arrangement is the Assemulator with a Personality Module. This approach allows the same Assemulator to be used for all PPS microcomputer systems.

A recent addition to the Rockwell development tools is XPO-1. XPO-1 is a low cost microemulator which performs many of the functions of the powerful Assemulator. XPO-1 provides a full PPS microcomputer system on a single PC board, complete with a 20-key keyboard and a five digit display. The XPO-1 is ideal for users with limited programming requirements or users who have multiple concurrent development programs.

Another possibility to aid in system development and prototyping is to use a development circuit which is identical to the production circuit except that it has no read-only memory. The development circuits in place of the ROM have some of the internal addresses and registers brought out of the package (it has extra leads) to be used with external program memory. The development circuits may be interfaced through suitable circuitry to PROM or other external memory or may be interfaced directly with PROM or other static memory modules.

This approach allows full real-time emulation but does not provide the flexibility of the Assemulator approach. The latter approaches provides aids for easier system debugging and also provides a capability for readily making corrections which may be immediately checked out.

Six weeks after the system is fully defined and the ROM code has been submitted to Rockwell, the production units with the masked ROM code and strap options can be available.



PPS DEVELOPMENT CYCLE

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I
THE PPS FAMILY

1.1 BASIC DIGITAL PROCESSORS

Any digital processing system whether it be a large high-speed computer system, a minicomputer, or a microcomputer, performs a number of basic functions as shown in Figure 1-1. Data must be supplied to the processor through some input interface. It must be processed. (This implies controlling of sequences of operations as well as numerical processing.) Memory must be provided in some form to hold data and intermediate results and also to hold the program sequence which defines the specific processing steps to be accomplished. Finally, the processed data must be transmitted out of the processor in a form which is usable outside of the processing system.

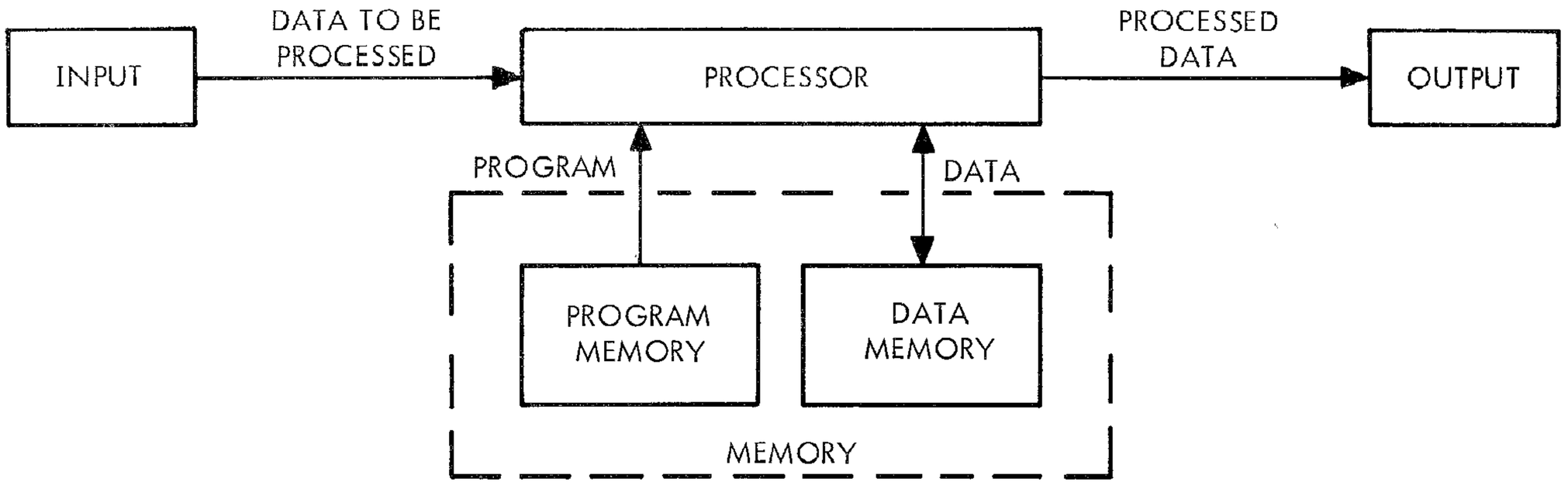


Figure 1-1. BASIC COMPUTER ORGANIZATION

1.2 ROCKWELL MICROCOMPUTERS

Rockwell has a number of large-scale integrated circuit systems for accomplishing digital processing as defined above. Each system is optimized for specific levels of processing complexity. The PPS-4/1 one-chip (single circuit) family of microcomputers has all of the input/output, storage, and processing capabilities within a single circuit chip. This family provides a capability of custom design for small systems and may be made available for customer design where quantities make the approach attractive. Currently there are two unique systems in the PPS-4/1 family, they are the MM76 and MM77 microcomputers.

In addition to the basic MM76 microcomputer, the MM76 is available in several specialized versions. All current and planned versions use the same basic instruction set and programming disciplines described in this manual. Any special instructions for a specific version of the MM76 will be covered in the Product Description document associated with that version. Currently, the five following versions of the MM76 are being offered: The basic MM76, the MM76C which incorporates a programmable high-speed counter, the MM76L which is a low-voltage, low power version of the MM76, the MM76E which has an expanded ROM program memory, and the MM75 which is a MM76 with fewer I/O lines so that it can be packaged in a 28-pin DIP.

Rockwell also has other families of microcomputers available for more complex system applications. These other systems include the PPS-4, PPS-4/2, PPS-8 and PPS-8/2 microcomputers as well as the R6500 family of microcomputers.

The PPS-4 and PPS-4/2 are multi-chip 4-bit parallel processing systems. The PPS-4 family provides a significant amount of computing capability and is low in cost — both in terms of the microcircuit costs, and the costs of the system implementing software and interfacing hardware.

The 8-bit parallel processing systems, the PPS-8 and the PPS-8/2, have even more capability. They process more rapidly due to processing 8-bits in parallel, have interrupt processing, have a faster basic clock rate, can perform eight channel direct memory access, and have an extremely flexible instruction set. The PPS-8 is intended for applications where data handling, real time control, and more extensive storage are important requirements. Though the PPS-8/2 is a low-cost, two-circuit system it offers most of the advantages of the PPS-8 system.

The purpose of this document is to describe software and hardware technique to aid in implementing all members of the MM76 series microcomputers in the PPS-4/1 family.

1.3 THE MM76 SYSTEM

The PPS-4/1, Series MM76 circuits have been designed to be used by themselves, or in conjunction with other PPS-4/1 circuits, or in conjunction with other PPS families of circuits (PPS-4/2, PPS-4, or PPS-8). The MM76 may be used as compact stand-alone microcomputer, as a low cost special controller, as a programmable peripheral controller for one of the larger PPS systems, as a sophisticated appliance controller, or as a universal logic element. The MM76 as a universal logic element can economically perform functions such as counting, time delays, comparsons, sequencing, function generating, etc., to control a set of output lines based upon conditions presented on a set of input lines.

Any of the PPS-4/1 systems may be operated in tandem to perform parallel processing functions in multi-microcomputer configurations.

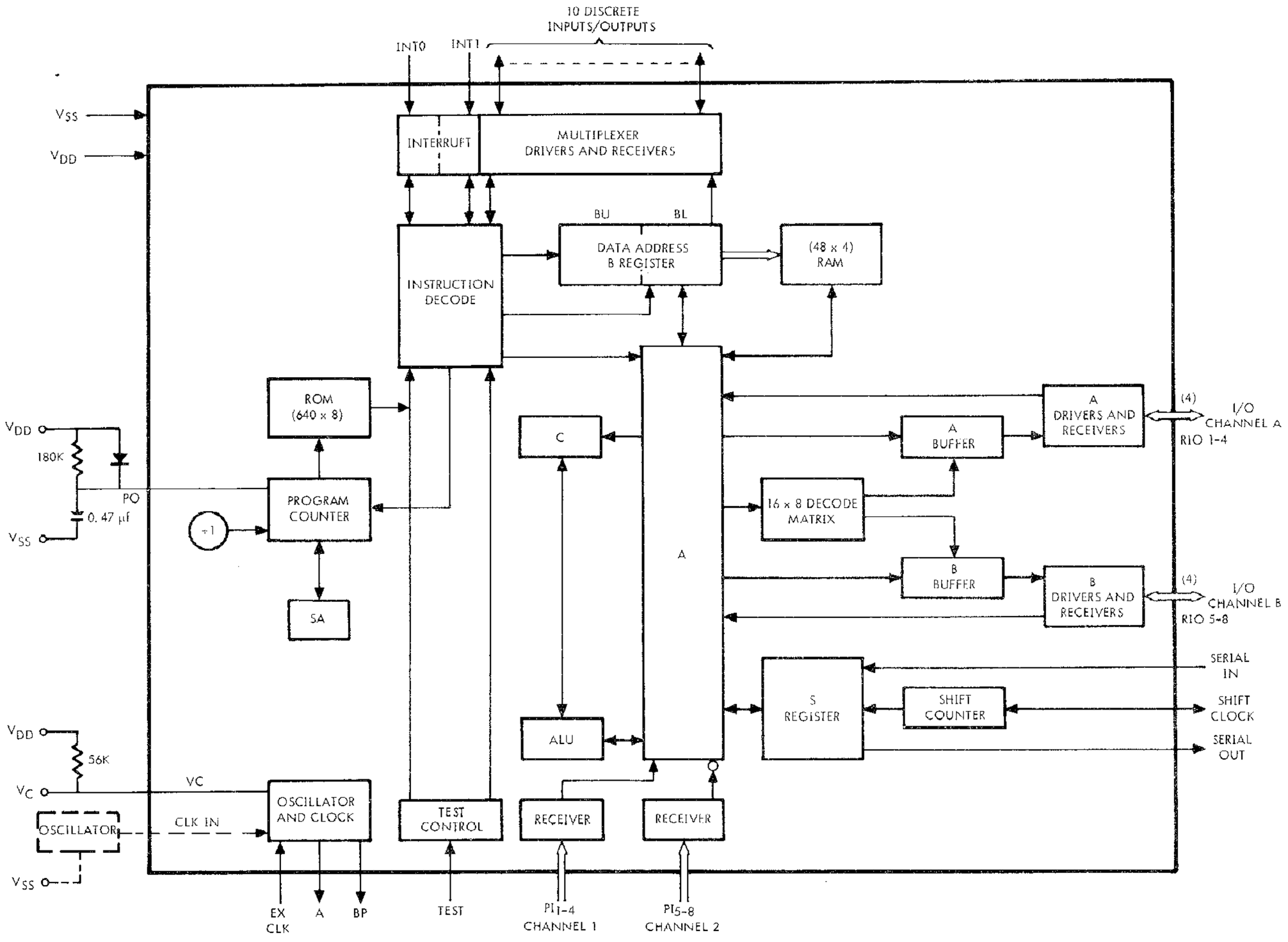
A block diagram of the basic PPS-4/1 MM76 is shown in Figure 1-2.

1.3.1 ACCUMULATOR AND ARITHMETIC LOGIC UNIT (A, ALU, AND C)

The primary working register in the PPS-4/1 MM76 is the Accumulator (A Register). It is the Accumulator which ties with the Arithmetic Logic Unit (ALU) and the carry flip-flop (C) to perform either binary or decimal arithmetic. Constants may be loaded into the Accumulator by appropriate instructions from the read only memory or variable data may be loaded from, or exchanged with the random access memory (RAM) under control of the Data Address Register (B). The Accumulator is also the primary path for 4-bit parallel or serial input or output data although the S Register may also be involved.

1.3.2 A BUFFER

The contents of the Accumulator may be output for control of data transfer purposes through the A Buffer via the Output A command. The A Buffer consists of four latched open drain circuits which hold the data output until new data is output or power is turned off.



NOTE: The evaluation circuit for the PPS-4/1 (Part No. A7699) is identical to the PPS-4/1 MM76 production circuits (Part No. A76XX) except that, in place of the ROM, terminals are brought out to an external memory and control system.

Figure 1-2. PPS-4/1 MM76 SYSTEM BLOCK DIAGRAM

1.3.3 DRIVER AND RECEIVER CIRCUITS

The outputs of each of the latches in the A Buffer, B Buffer and discrete input output flip-flops are through open drain drivers which drive to either the VSS logic level or float. When power is applied, the power on reset circuit (PO) causes all of the outputs to be automatically set to a float condition. (All output flip-flops are reset.)

Input signals may share the same lines as outputs for the A and B channels (RIO 1-8) and the discrete input/output lines. To use one of these lines as an input, it is necessary to set the output on that line into a float condition. The external signal may then either take the line high to VSS or to the appropriate low logic level. The input command, in effect, samples the logic level on the pin and inputs it appropriately.

The mechanization of the PPS-4/1 MM76 allows a software masked input capability. To mask out any input bits, the Accumulator bit in that bit position is set to 0. The external input in that position will always be input as a zero

while other bits input to bit positions pre-conditioned by a 1 in the Accumulator will be at logic one or zero levels as determined by the external system. This is in effect a logical AND between the initial contents of the Accumulator and the input signal.

1.3.4 B BUFFER

The output B instruction causes the contents of the Accumulator to be transferred to the B Buffer. The B Buffer comprises four latches which will output the last bit pattern loaded until either a new Output B command is executed or power is turned off. The power on reset signal resets all of the latches so the outputs float.

1.3.5 16 x 8 DECODE MATRIX

The MM76 microcomputer has the capability of connecting the four bit code in the Accumulator and its complement in addressed memory to any 8-bit code desired via the 16 x 8 Decode Matrix. The Carry flip-flop may also be used to select matrix terms. The contents of the Decode Matrix are specified by the user. When the SEG1 instruction is used, the four bit code in both the Accumulator and its complement in the memory selects the least significant four bits of the 8-bit code for that character and outputs it from I/O Channel A. Similarly, when the SEG2 instruction is used the upper four bits of the code in the matrix are output to I/O Channel B.

The user may define any code desired. The development circuit version of the MM76 has a BCD to seven segment conversion provided. The 0 through #F characters in the Accumulator produce 0 through 9, A, -, P, d, E and blank respectively. The Carry flip-flop controls RIO8.

1.3.6 S REGISTER — SERIAL INPUT/OUTPUT — SHIFT COUNTER

The S Register is a 4-bit parallel-in/parallel-out, serial shift register which is used as either an auxiliary storage register or a buffer for the simultaneous serial-in/serial-out capabilities in the microcomputer. The 4 bits to be serially output are loaded into the S Register either by exchanging the contents of the Accumulator and the S Register or directly loading the S Register from the Accumulator. The state of the serial output line is immediately set by the contents of the most significant bit position. When an Input/Output Serial instruction is executed, or an external shift clock input is provided the four bit contents of the S Register are shifted out (most significant bit first). The data shift rate is under control of the Shift Counter, and is one-half the rate of the internal clock frequency when the IOS instruction is used. The Input/Output Serial instruction also causes the Shift Counter to provide four shift clock signals to the external system. Under external shift control on the same shift clock line, the shift rate may be any value at or below the clock frequency. Both the serial data and shift clock outputs are open drain drivers which are set to the float state when power is turned on.

At the same time that the 4-bit data is being shifted out through the serial output line, 4 bits of data are shifted into the S Register from the serial input line. An exchange of the Accumulator and S Register brings the 4 bits of serial input data into the Accumulator where it can be processed or stored. The S Register may be simultaneously reloaded if more than 4 bits of data are being transmitted.

When the external clocking mode is used it may be necessary for the system designer to establish a handshake protocol to establish when data is to be moved and when the move is completed.

1.3.7 DISCRETE INPUT/OUTPUT PORTS (DI/00 THROUGH DI/09)

There are ten discrete input or output lines. Buffer flip-flops associated with all ten of these channels may be individually set, or reset under program control. They are all reset when power is applied. There is a buffer flip-flop associated with each of these channels which is selected by the least significant 4 bits of the Data Address (B) Register. A Set Output Selected instruction causes the selected output to be at the VSS level and a Reset Output Selected instruction causes it to float. When the output is floating, an input signal level on that port may be tested by a Skip on Input Selected Low instruction. When the Buffer flip-flops are not used specifically for input/output functions, they may be used as one bit status registers. In this case external pull up resistors connected to VDD must be used.

1.3.8 CONDITIONAL INTERRUPTS (INT0 AND INT1)

The conditional interrupt request lines may be used in a number of different ways. These ports are different from the discrete input/output channels in that they may be addressed directly and not by the B Lower portion of the B Register.

To test the state of the signal on the input line it is not necessary to set a flip-flop to any predetermined state as there is no output driver on these signal lines. The level on these two lines may be tested directly by an INT0L or INT1H instruction for INT0 and INT1 inputs respectively without any pre-conditioning. This gives the PPS-4/1 a pseudo interrupt capability by allowing a direct test of the input signal. The INT0L instruction causes the next instruction to be skipped if the input on INT0 is low and the INT1H instruction will cause it to skip if the signal on the INT1 line is high.

Another difference in these two signals is that they may be used to detect a pulse input of a duration longer than one clock cycle. In this case, for INT1 the associated flip-flop is preset to the set state by testing so that any subsequent incoming negative transition pulse on INT1 which lasts longer than one clock cycle will reset the flip-flop. The state of the flip-flop may then be tested by addressing it with a DIN1 instruction which will cause the next instruction to be skipped if the flip-flop is reset. Testing the flip-flop automatically restores it to the set state so that it is ready for the next test via the DIN1 instruction.
The DIN0 instruction similarly may be used to test for a positive transition or pulse on INT0.

1.3.9 CHANNEL 1

Channel 1 is a 4-bit input port which automatically loads the input value to the contents of the Accumulator.

1.3.10 CHANNEL 2

Channel 2 is a 4-bit input port which on command replaces the contents of the Accumulator with the complement of the value on the input lines. If the input value is from TTL or CMOS logic, the inversion causes the equivalent value to appear in the Accumulator.

1.3.11 PROGRAM COUNTER (P)

The 10-bit Program Counter is set to a specific initial value (hexadecimal address ICO) when power is applied to the microcomputer. The contents of the Program Counter addresses read-only memory to identify the specific instruction

to be executed. Then, unless the instruction is a transfer instruction, the contents of the Program Counter are incremented so that the next instruction may be selected. This process repeats until a transfer or transfer and mark instruction is executed. The transfer instruction may set a specific location into the least significant 6 bits (T) while leaving the upper bits fixed or may set the complete 10 bits with a Transfer Long (TL) instruction.

Similar alternatives are available for the transfer and mark instructions which are used to call subroutines. The TM instruction selects one of 64 locations in a specific area and the TML sets the complete 10 bits. These instructions, however, mark a return location so that the subroutines may return to the next instruction location after the one that called it. This is accomplished by incrementing the Program Counter prior to setting the new value into it, and saving the incremented value in the SA Register.

1.3.12 SA REGISTER

When a subroutine call is executed by one of the transfer and mark instructions, the contents of the SA Register are replaced by the incremented value of the Program Counter.

When a return instruction is executed in the subroutine, the contents of the SA Register are popped into the Program Counter.

1.3.13 READ ONLY MEMORY (ROM)

The Read Only Memory (ROM) provides the storage for instructions and constants (as immediate field portions of instructions) for the microcomputer. It contains 640 instruction bytes of 8 bits each. It is controlled by the Program Counter to read out each instruction to be executed.

1.3.14 INSTRUCTION DECODE

The instructions are decoded in the Instruction Decode circuits which then issue control signals to all appropriate portions of the microcomputer as necessary to perform the desired operations.

1.3.15 DATA ADDRESS REGISTER (BU AND BL)

The Data Address Register is 6 bits in length and is made up of two segments, B Upper (BU) and B Lower (BL). Register BU is two bits in capacity and BL is four bits. Data memory in RAM is addressed by all 6 bits and discrete input/output ports are addressed by the 4 bits in BL when the value in B Upper is three.

The BL portion may be automatically incremented or decremented and tested for overflow or underflow by the Exchange Increment and Skip, Exchange Decrement and Skip, Increment B, or Decrement B instructions.

1.3.16 DATA MEMORY (RAM)

The Random Access Memory (RAM) used for data memory consists of 48 characters of 4 bits each. This memory is used to buffer input or output values, hold intermediate results and also may be used as registers used for timers, counters, comparators, etc. when the microcomputer is used as a universal logic element.

1.3.17 CLOCK CONTROL (VC, CLKIN, EXCLK, AND OSCILLATOR)

When used in systems where precise timing is not important, the microcomputer may derive its internal clock signals very simply. A resistor connected between the VC input and VDD completes the circuit for an internal oscillator which is used to generate the four-phase clock system used for all the internal logic. The A and BP ($\overline{B}$) clock terms are brought out so external logic may be synchronized. A nominal 56k ohm resistor will set the clock frequency to the nominal 80 kHz value with a 50 percent tolerance. Closer frequency control may be achieved by resistor selection.

When precise timing is to be achieved, a precision external reference frequency may be used. A square wave oscillator input through the CLKIN pin when the EXCLK pin is tied to VDD will cause the internal clock generation to be slaved to an external reference at any frequency within the range of 40 kHz to 80 kHz.

1.3.18 PPS-4/1 TESTABILITY (TEST)

Another advantage of the PPS-4/1 microcomputer family is testability both at the factory and user levels. When a test state is indicated by the test input line, the PPS-4/1 goes into a test mode which tests ROM and allows testing of the RAM and instruction logic.

1.3.19 POWER SUPPLY

When inputs and outputs interface with other PMOS devices, or CMOS devices, VSS = GND and VDD = -15V ±5% provide proper interface levels. When interfacing with TTL devices, VSS should be +5 and VDD at -10 volts.

1.3.20 POWER ON RESET (PO)

The PO signal is derived from an external resistor, diode, and capacitor pulse shaping network which is tied to the power supply as shown in Figure 1-2. When power comes on, this circuit automatically sets the Program Counter to a fixed starting location and all outputs are set to a "float" (-V) state. The Program Counter then initiates the first instruction (which must be a Set Carry, Reset Carry or NOP instruction) to be read from the read only memory (ROM) into the instruction decode logic. After executing the first instruction the Program Counter increments so that the second and subsequent instructions may be recalled from memory and executed.

SECTION II
PROGRAMMING CONCEPTS

2.1 THE DATA WORD AND WHAT IT CAN REPRESENT

The PPS-4/1 is a 4-bit device, which means that data is handled in words consisting of four binary digit units. By convention, the binary digits (bits) of a word are numbered from right to left, as illustrated in Figure 2-1.

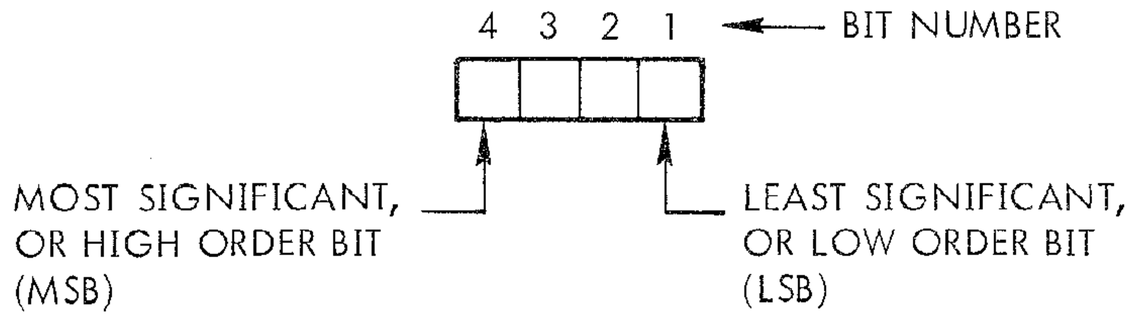


Figure 2-1. BIT NUMBERING WITHIN A BYTE

Since a data word consists of four binary bits, it may contain any one of 16 (2^4) possible combinations of zeros and ones. The contents of the word can, however, be interpreted in any one of the following ways:

1. As pure binary data.
2. As part of a character representation, ASCII codes and Binary Coded Decimal are character representations which may be used by the PPS-4/1.
3. As a special code representing static or other non-numeric non-character data.

Binary data can itself be interpreted in a variety of ways, as is described in Paragraph 2. 2.

How do you differentiate between characters, data, and special codes? Upon examining the contents of an isolated word, there is no way of differentiating. Only the manner in which data words are used determines how the word contents will be interpreted.

2.2 NUMBERS AND ARITHMETIC

There is no single assembly language instruction that will perform an operation such as:

multiply 462175. 34 by 2161. 2375

To begin with, the only arithmetic operations the computer can perform are complementing and addition; secondly, the "word size" of a computer determines the largest increment of a multidigit number that can be processed at one time. Yet, by suitably grouping instructions there is no limit to the arithmetic operations that can be performed.

2.2.1 NUMBER SYSTEMS

Binary numbers consist of the digits 0 and 1 only.

Decimal numbers consist of the familiar digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless otherwise identified in this manual, numbers are written in decimal form.

Hexadecimal numbers consist of the ten decimal digits, plus the letters A, B, C, D, E and F, representing 10, 11, 12, 13, 14 and 15, respectively. Hexadecimal numbers are identified in this manual by a preceding pound sign (#).

Hexadecimal numbers are the normal medium for representing data in PPS-4/1 program listings. Decimal data do not convert easily into binary equivalents, and binary numbers are clumsy to read and write; however, the four binary digits of a word are easily grouped into hexadecimal digits, (four bits per digit).

Decimal numbers are represented as one binary coded decimal digit per four-bit word. This is referred to as BCD. Groups of 4-bit words may be used to represent a decimal or binary number of any length. In this manual such a grouping will be referred to as a register or a memory register.

Table 2-1 shows the equivalence of various number systems. Observe from Table 2-1 that BCD uses just 10 of the possible 16 bit combinations of a word; the remaining bit combinations, as representations of decimal data, are unused. They may be used however to represent other symbols such as blank or minus, etc.

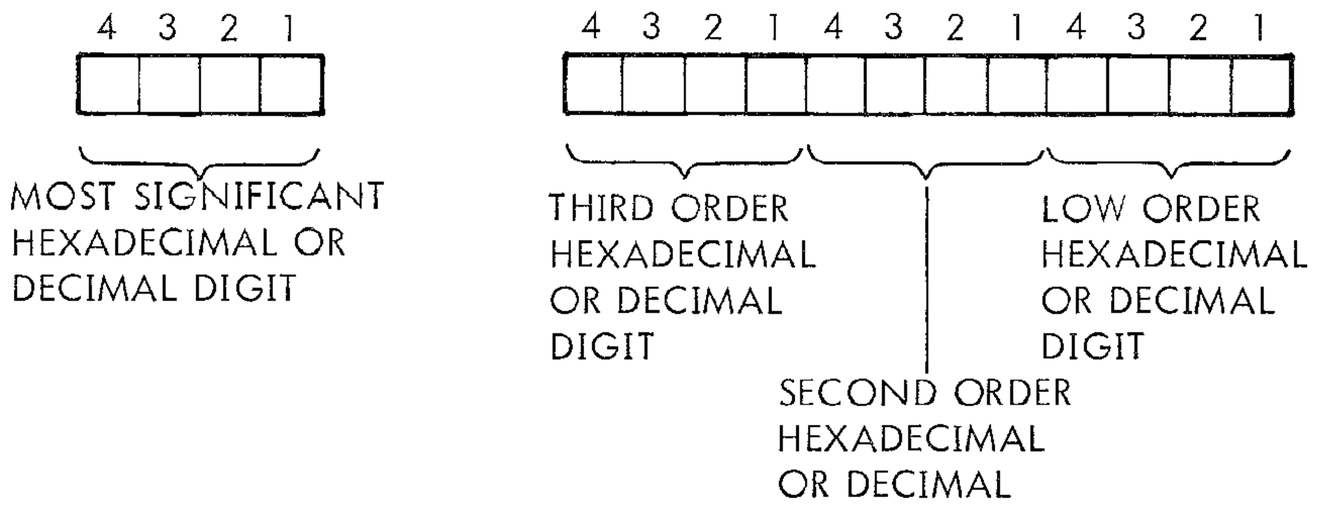


Figure 2-2. A STRING OF 4-BIT WORDS (REGISTER) VIEWED AS HEXADECIMAL OR DECIMAL (BCD) DIGITS

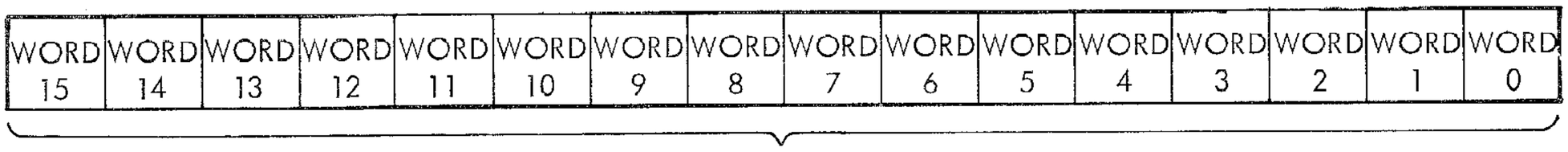
Table 2-1. BINARY, DECIMAL AND HEXADECIMAL NUMBERS

Binary	Decimal	Hexadecimal	Binary Coded Decimal	
0000	0	0	0000	0000
0000	1	1	0000	0001
0010	2	2	0000	0010
0011	3	3	0000	0011
0100	4	4	0000	0100
0101	5	5	0000	0101
0110	6	6	0000	0110
0111	7	7	0000	0111
1000	8	8	0000	1000
1001	9	9	0000	1001
1010	10	A	0001	0000
1011	11	B	0001	0001
1100	12	C	0001	0010
1101	13	D	0001	0011
1110	14	E	0001	0100
1111	15	F	0001	0101

2.2.2 NUMBER RANGES AND NEGATIVE NUMBERS

Four binary digits allow numbers in the range '0' to '1111' binary, or 0 to 15 decimal, or #0 to #F hexadecimal, or 0 to 9 BCD.

Positive numbers outside the range allowed by one word are generated by concatenating a number of sequential bytes into a number buffer (a section of memory), which is treated as a single unit; for example, a 64-bit number is maintained in a memory register which is 16 words long, (4X 16 = 64) as shown below.



Which end of the number register holds the high order digits, and which end holds the lower order digits is the programmer's choice; so long as a program is consistent in its handling of multiword numbers. There is no "correct" direction in which numbers should be processed.

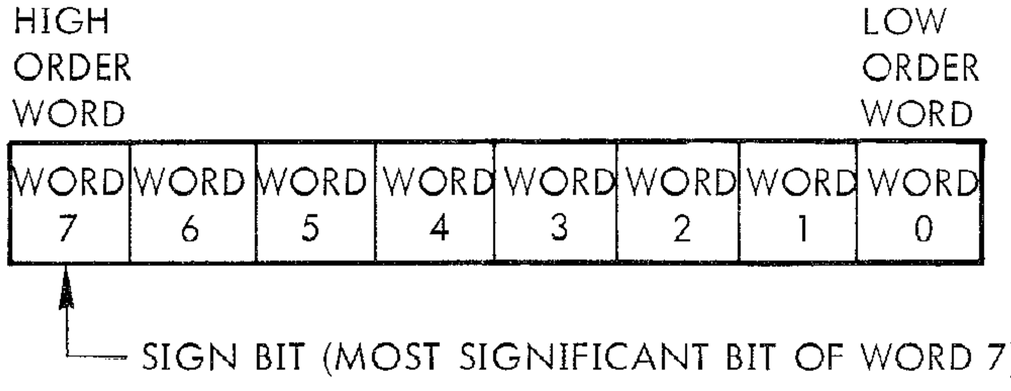
But what about negative numbers? A useful and workable convention used throughout the computer industry is to assign the high order bit to represent the sign of a number. Now a word will be interpreted numerically as shown in Table 2-2. The important word here is INTERPRETED. There is nothing about signed binary data that is hardware dependent, it is purely a question of how you choose to interpret a bit pattern.

Observe that there are two ways in which negative numbers may be represented. Two's complement is the natural fallout of signed binary arithmetic (described on the following pages). The sign and absolute value merely interprets the high order bit as a sign bit, but treats the remaining bits as though they coded a positive number; this scheme results in longer programs.

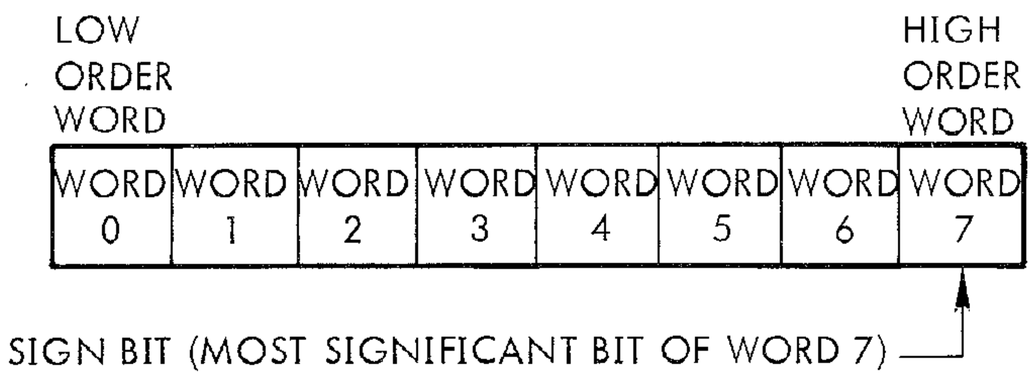
Table 2-2. SIGNED BINARY NUMERIC INTERPRETATIONS

Binary	Equivalent Decimal (Two's Complement)	Hexadecimal	Equivalent Decimal (Sign and Absolute Value)
10000000	-128	80	-0
10000001	-127	81	-1
10000010	-126	82	-2
–	–	–	–
–	–	–	–
–	–	–	–
11111110	-2	FE	-126
11111111	-1	FF	-127
00000000	0	0	0
00000001	1	1	1
00000010	2	2	2
–	–	–	–
–	–	–	–
–	–	–	–
01111101	+125	7D	+125
01111110	+126	7E	+126
01111111	+127	7F	+127

In multiword signed numbers, only the high order bit of the high order word is a sign bit; the high order bits of all other words are simple, numeric bits:



If the words are stored in memory with the order inverted, (the bit order in the word is not inverted) the sign bit would be located as follows:



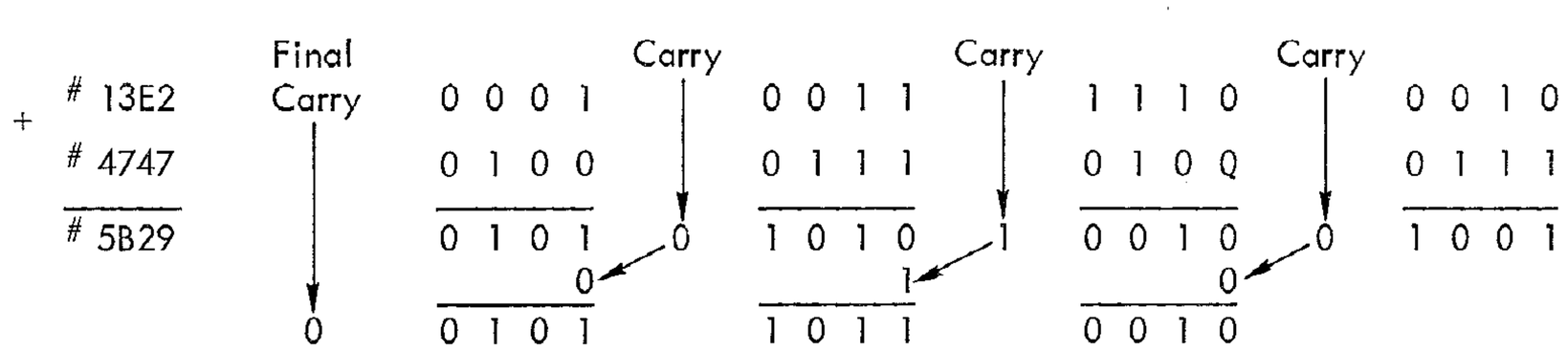
The sign bit is always considered to be the most significant bit in the most significant word. The sequence of words in memory is not important, but the sequence of processing them is.

2.2.3 BINARY AND DECIMAL ADDITION

Consider addition using positive numbers only.

When the result of the addition will not fit into 4 bits, the PPS-4/1 Carry bit (C) words like a fifth, high order bit, holding any carry from the binary addition. Here are some examples:

The carry is generally used when adding multibyte numbers, in which case a carry out of the high order bit of one word must be added to the low order bit of the next word. Here is a simple example for a four word (16-bit) number:



The PPS-4/1 adds decimal numbers by a technique which is described with the instruction set in Section 3.

2.2.4 BINARY AND DECIMAL SUBTRACTION

Subtracting a binary number is the same as adding the twos complement of the number.

The twos complement of a number is generated by forming a ones complement of the number (replace 0 with 1 and 1 with 0), then adding 1 to the result. Here is an example:

$^{\#}C$ =	1 1 0 0
Ones complement =	0 0 1 1
	1
Twos complement =	0 1 0 0

When subtracting positive numbers, the rules for interpreting the answer are:

1. If the carry bit is one, the answer is positive.
2. If the carry bit is zero, the answer is negative, and in its twos complemented form.

When adding and subtracting signed binary numbers, if the interpretations given in Table 2-2 are adhered to, rules differ from positive numbers subtraction only in that the answer is self-evident by inspection. For example, 11001100 is negative because the high order bit is set; therefore it represents the negative of its two's complement:

ones complement of 11001100 = 00110011 = $^{\#}33$
twos complement of 11001100 = 00110100 = $^{\#}34$

Here are some examples of subtraction involving positive numbers only:

	0 1 0 0 = $^{\#}4$	
	1 1 0 0 = Twos Complement of $^{\#}4$	
$^{\#}E$	$^{\#}E$ = 1 1 1 0	
$-\ ^{\#}4$	$-\ ^{\#}4$ = 1 1 0 0	
$^{\#}A$	[1] 1 0 1 0 = $^{\#}A$	
	↑ CARRY (1 INDICATES NO BORROW)	
	0 1 0 0	0 0 1 0 = $^{\#}42$
	1 0 1 1	1 1 1 0 = Twos Complement of $^{\#}42$
$^{\#}13$	$^{\#}13$ = 0 0 0 1	0 0 1 1
$-\ ^{\#}42$	$-\ ^{\#}42$ = 1 0 1 1	1 1 1 0
$-\ ^{\#}2F$	[0] = 1 1 0 1	0 0 0 1
	↑ CARRY (0 INDICATES BORROW)	
Twos Complement of $-\ ^{\#}2F$ =	0 0 1 0	1 1 1 1 = $^{\#}2F$

The PPS subtracts decimal numbers using a technique which is described in Paragraph 3.4.7.

A summary of the resulting sign interpretation for all manipulations of signed numbers is shown in Table 2-3.

Table 2-3. INTERPRETATION OF SIGN BIT AFTER ARITHMETIC OPERATIONS

Operation	Original Signs		Final Sign Bit = 1	Final Sign Bit = 0
	A	B	Result	Result
A + B	+	+	Overflow	OK; answer positive
	+	-	OK; |B| > |A| so answer negative	OK; |A| ≥ |B| so answer positive
	-	+	OK; |A| > |B| so answer negative	OK; |B| ≥ |A| so answer positive
	-	-	OK; answer negative	Overflow
A - B	+	+	OK; |B| > |A| so answer positive	OK; |A| ≥ |B| so answer positive
	+	-	Overflow	OK; answer positive
	-	+	OK; answer negative	Overflow
	-	-	OK; |A| > |B| so answer negative	OK; |B| ≥ |A| so answer positive

2.3 SOFTWARE AIDS

There are a number of software aids which simplify the generation of a program for the PPS-4/1 family of microcomputers. These include software systems for assembling the program and editing the program to incorporate changes, and also includes a library of commonly used routines and programming techniques.

2.3.1 PPS-4/1 ASSEMBLERS

In the PPS-4/1 microcomputers, as in most computer systems, both instructions and data are represented by bit patterns of binary 1's and 0's. These are manipulated internally to form the desired computations and control functions and must be interpreted by any interfacing hardware. However, strings of 1's and 0's are extremely difficult to work with during the development phase of a system containing a microcomputer. The answer, of course, is an assembly program. An assembly program allows the programmer to write the instructions more conveniently by using alphanumeric labels for memory locations and mnemonic abbreviations for instructions. The program also allows comments to be appended to aid in the debugging and documenting processes.

There are three different assembly and editing systems which may be used for assembling and editing PPS-4/1 microcomputer programs. Programs may be assembled and edited on the General Electric Information Services system via a terminal, on IBM 360 or 370 systems or other suitable computer systems using FORTRAN IV, or the programs may be assembled, edited, debugged, and the system may be checked out using the Assemulator with a PPS-4/1 Personality Module or with the XPO-1 microemulator board. The Assemulator with the optional PROM programmer board can also transfer the working program from its memory to PROM's for use with the microcomputer evaluation circuit to be used for prototype systems or limited production runs.

Because of the greater flexibility provided by the Assemulator, it is anticipated that the Assemulator will be the primary development tool. For this reason, although all of the assembly capabilities are similar, the examples used in this document will use the Assemulator version of the Assembly Program.

2.4 PPS-4/1 ASSEMBLER FOR USE WITH THE PPS UNIVERSAL ASSEMULATOR

2.4.1 INTRODUCTION

This section describes the use of a cross assembler for the PPS-4/1 microcomputer which has been written in PPS-8 language and which is designed to execute on the PPS Universal Assemulator. A detailed description of the Assemulator and its use for editing and running PPS-4/1 programs is located in the PPS-4/1 Operator's Manual for PPS Universal Assemulator.

The purpose of the PPS-4/1 assembler is to allow the user to write programs using mnemonic codes in place of binary instruction codes and using symbols, decimal numbers, hexadecimal numbers and expressions in place of the binary operand fields of an instruction.

The assembler produces a listing which shows the original assembly language program as written and the corresponding machine language code generated along with identifying the specific memory cells occupied by the code. The assembler also produces an object tape in a format suitable for input to the Assemulator Utility program.

2.4.2 GENERAL OPERATION DESCRIPTION

The assembler is a two or three pass assembler, depending on whether or not the devices being used for the output listing and the object program can be operated independently. Pass 1 generates the symbol table and generates certain error diagnostics. Pass 2 generates the source/object listing and further error diagnostics. Pass 3 produces the object program in a format suitable for input to the supervisory program.

Detailed operating information can be found in the PPS-4/1 Operator's Manual for the Universal Assemulator.

2.4.3 INPUT OPERATION DESCRIPTION

2.4.3.1 Character Set

The assembler operates with the 128 character ASCII set as shown in Appendix A. Both the line feed (LF) character and the carriage return (CR) character are treated as statement delimiters. If a string of two or more such codes appears in the source program, all but the first are ignored. Consequently, each line on a teletype or terminal is terminated by either CR, LF or LF, CR.

2.4.3.2 Symbol

A symbol is used in the label field or operand field of an input statement. A symbol is defined as a character string of no more than four characters. Any character in the character set, except [+], [-], [(], [)], [!], and [ ] (space) may be used. The characters [#], [*], and ['] may not be used as the first character of a symbol. If a symbol longer than four characters is used only the first four will be interpreted. Care must be used when doing this to be sure the first four characters are unique.

Examples:

VALID	NOT VALID	REASON NOT VALID
LOOP	LOOP1	Confusion with LOOP symbol
TEST	+TST	+ improper character
TST2	TST 2	space is improper
ON#	#TWO	#improper start
ALB*	*ALB	*improper start
ALPHANUMERIC	ALPHABET	ALPH confusion

2.4.3.3 Decimal Number

A decimal number consisting of a string of from one to four numeric characters. All numbers defined as positive whole numbers.

VALID	NOT VALID	REASON NOT VALID
1	1.2	Interpreted as symbol
42	-42	Positive values only
4396	14396	Too many digits

2.4.3.4 Hexadecimal Number

A hexadecimal number is denoted by the character "#" followed by a string of from one to three characters in the range of 0-9 and A-F.

VALID	NOT VALID	REASON NOT VALID
#F	F	Interpreted as symbol
#23	23	Interpreted as decimal

2.4.3.5 Expressions

Two types of expressions are defined: data expression and program expression.

2.4.3.5.1 Data Expression

Data expressions are used to symbolically or actually define a value which is used as data. Other statements in the program define the value assigned to symbolic names for data.

A data expression may have one of the following forms:

1. Decimal number (2, 34, 412, etc.).
2. Symbol (SA2, DATA' VALU, etc.).
3. Symbol, followed by plus or minus sign, followed by decimal number (SA2+4, DATA-1, etc.).
4. Hexadecimal number (#A, #35, #3CB, etc.).

2.4.3.5.2 Program Expression

Program expressions are used to symbolically or actually define in absolute terms program memory locations which are entry or reentry points in the program. Consequently, program expressions are generally addresses for transfer instructions.

A program expression may have one of the following forms (shown as operands for transfer instructions):

1. Symbol (TL X24, T LOOP, TMB PLUS, etc.).
2. Symbol, followed by plus or minus sign, followed by decimal number (TL X24+2, T LOOP-3, etc.).
3. "*" This is a symbol meaning "this location" (*-3 means this location minus 3 bytes, *+2 means this location +2 bytes).

2.4.3.6 Input Statement Format

The assembly language source program is prepared by the user and represents the PPS-4/1 program, written in a symbolic notation. A statement may have any length. Each input statement consists of four fields: Label, Operation, Operand and Comments. The following paragraphs describe, for each field, its placement and its purpose.

Label Field — The label field begins with the first character of the statement and terminates with the first space character encountered. If the label is longer than four characters, only the first four are used.

When a symbol appears in the label field of an input statement, it establishes a symbolic address for a RAM location or ROM location or establishes a symbolic immediate field which can be referenced in the operand field of other statements.

Operation Field — The operation field may begin in any position past the first and must be separated from the label if it is present, by at least one space character. In all cases a space character must precede the operation field. The entry in the operation field is a mnemonic which identifies the instruction.

Operand Field — The operand field may begin in any position past the third and must be separated from the operation field by at least one space character. The form of the operand depends upon what instruction is being coded. Some instructions require an operand; a few (L, X, XDSK, XNSK, INCB, DECB) assume an operand of zero if none is coded; the rest do not use an operand at all.

Comment Field — The comment field begins after a single space character which follows the operand field for those statements which require an operand field. If an operand has not been coded in an L, X, XDSK, XNSK, INCB or DECB instruction, a "!" character must precede the comment.

This field is printed with the statement in the source listing but is otherwise ignored by the assembler. The user may place descriptive text in this field as he desires.

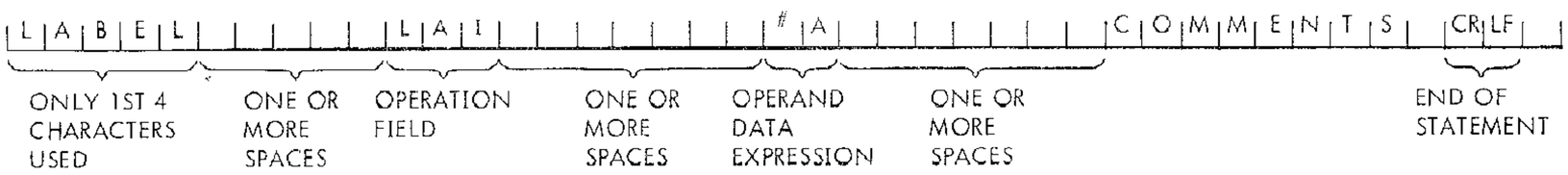
Figure 2-3 summarizes in pictoral form several examples of source assembly program formats.

2.4.3.7 Instructions

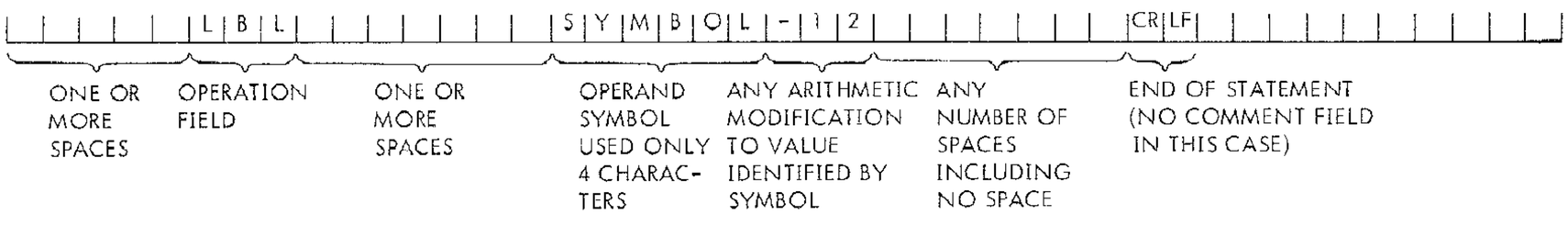
There are four classes of instructions accepted by the assembler.

1. CPU Instructions
2. CPU Macro Instruction
3. Assembler Instructions
4. Declaration Instructions

GENERAL FORM:



FORM WHEN NO LABEL IS USED:



FORM WHEN OPTIONAL OPERAND IS USED WITH COMMENT:

Figure 2-3. SOURCE ASSEMBLY PROGRAM FORMAT EXAMPLES

2.4.3.7.1 CPU Instructions and Macro Instructions

CPU instructions are assembled as binary words to be encoded into program memory for execution by the PPS-4/1 CPU. CPU macro instructions are predefined combinations of CPU instructions.

The instructions have been grouped according to operand requirements and each of these groups has been designated by a specific type number. Table 2-4 lists the instructions and macro instructions by type and identifies each instruction that falls within each specific type.

Additional information of interest to the user of the PPS-4/1 Assembler can be found in the tables in Appendix C. Appendix C Table C-1 lists the microcomputer instructions for the A76XX microcomputer. The table contains an alphabetical listing of the CPU instruction mnemonics, the hexadecimal codes, the lengths of the instruction (in 8-bit instruction words) and the instruction type. Table C-2 provides the same information for the macro instructions, but provides the macro expansions rather than the hexadecimal codes.

2.4.3.7.2 Assembler Instructions

Just as the CPU instructions are used to request the PPS-4/1 microcomputer to perform a sequence of operations during the program execution, so assembler instructions are requests to perform certain operations during the assembly.

The assembler instructions, in contrast to CPU instructions, are effective only during the assembly process. They generate nothing in the assembled program and have no effect on the program counter.

"*" Comment Statement — The comment statement is used, much like the comment field of other instructions, to introduce descriptive text into the listing. A line of text may be introduced into the listing by placing an asterisk, "*", in the first character position of the input statement. The remaining characters of that statement are then printed on the listing but otherwise ignored.

ORG — Origin Instruction — The ORG statement is used to establish the value of the instruction location counter of the assembly. Its operand must be a hexadecimal number. The value of this number becomes the new value of this counter. If no operand is specified, the location counter is set to the beginning of the next page of ROM.

EQU — Equate Instruction — The EQU instruction is used to define the symbol in the label field by assigning to it the value of the argument found in the operand field. The argument may be any expression. If the expression contains "*", it is treated as a program expression; otherwise it is treated as a data expression. A symbol which appears in an expression must appear in a previous label field. Examples of these assembler instructions are shown in Figure 2-4.

SPACE — Space Instruction — The SPACE instruction is used to insert a blank line into the listing. The space instruction itself is not printed.

END — End Instruction — The END instruction signals the assembler that all of the instructions of the source program have been read.

LOF — List-Off Instructions — This instruction signals the assembler to stop printing source statements until the LON instruction is encountered.

LON — List-On Instructions — This instruction signals the assembler to start printing source statements until the LOF instruction is encountered.

STOP — Stop Instruction — This instruction signals the assembler to stop reading from the input device until any key on the console is depressed. The instruction is normally used at the end of a tape when the source program is distributed over several tapes. Since the assembler always has read one statement ahead of the one being processed, it is required that a tape (other than the last) end with a STOP statement followed by one other statement (a comment or SPACE statement is recommended).

2.4.3.7.3 Declaration Instructions

Declaration instructions are special types of assembly instructions which define data in ROM.

DW — Data Word Instruction — The DW instruction is used to define an 8-bit data byte to be placed in ROM at the present value of the location counter.

Table 2-4. INSTRUCTIONS AND MACRO INSTRUCTIONS BY TYPE

Instructions	Acceptable Operand Forms D: Data Expression P: Program Expression N: None	Range of Operand	Role of Operand	Examples
L, XNSK, X, XDSK	D or N	0-3	Specifies modification of B-Upper	L XNSK 3
SB, RB, SKBF	D	1-4	Specifies bit	SB 2
EOB	D	0-3	Specifies modification of B-Upper	EOB 2
LB	D	0-F	RAM address (assembler takes operand modulo 15)	LB AX LB 6
LAI	D	0-F	Immediate data	LAI 6 LAI #B
AISK	D	1-F	Immediate data	AISK 2
LBL	D	0-3F	RAM address	LBL XDTA
INCB, DECB	D or N	0-3	Specifies modification of B-Upper	INCB DECB 2
SKBEI	D	0-F	Immediate data	SKBEI #A
SKAEI	D	0-F	Immediate data	SKAEI #A
TM	P	3C0-3FF	ROM address (DOIT is in SRO Page)	TM DOIT
T, TC, TNC, TE, TNE, TIH, TIL	P	Address on same page as instruction	ROM address	T *+4 T G24
TL, TML, TLC, TLNC, TLE, TLNE, TLIH, TLIL	P	000-37F	ROM address off page	TML ZAP TL XY-2
TBF, TBT	D, P	1-4, Address on same page as instruction	First operand is a bit specification; second operand is a ROM address	TBF 2, *+2
TLBF, TLBT	D, P	1-4, 000-37F	First operand is a bit specification, second operand is a ROM address	TLBT 1, BARC
All other CPU Instructions	N	NONE		SKNC I2C

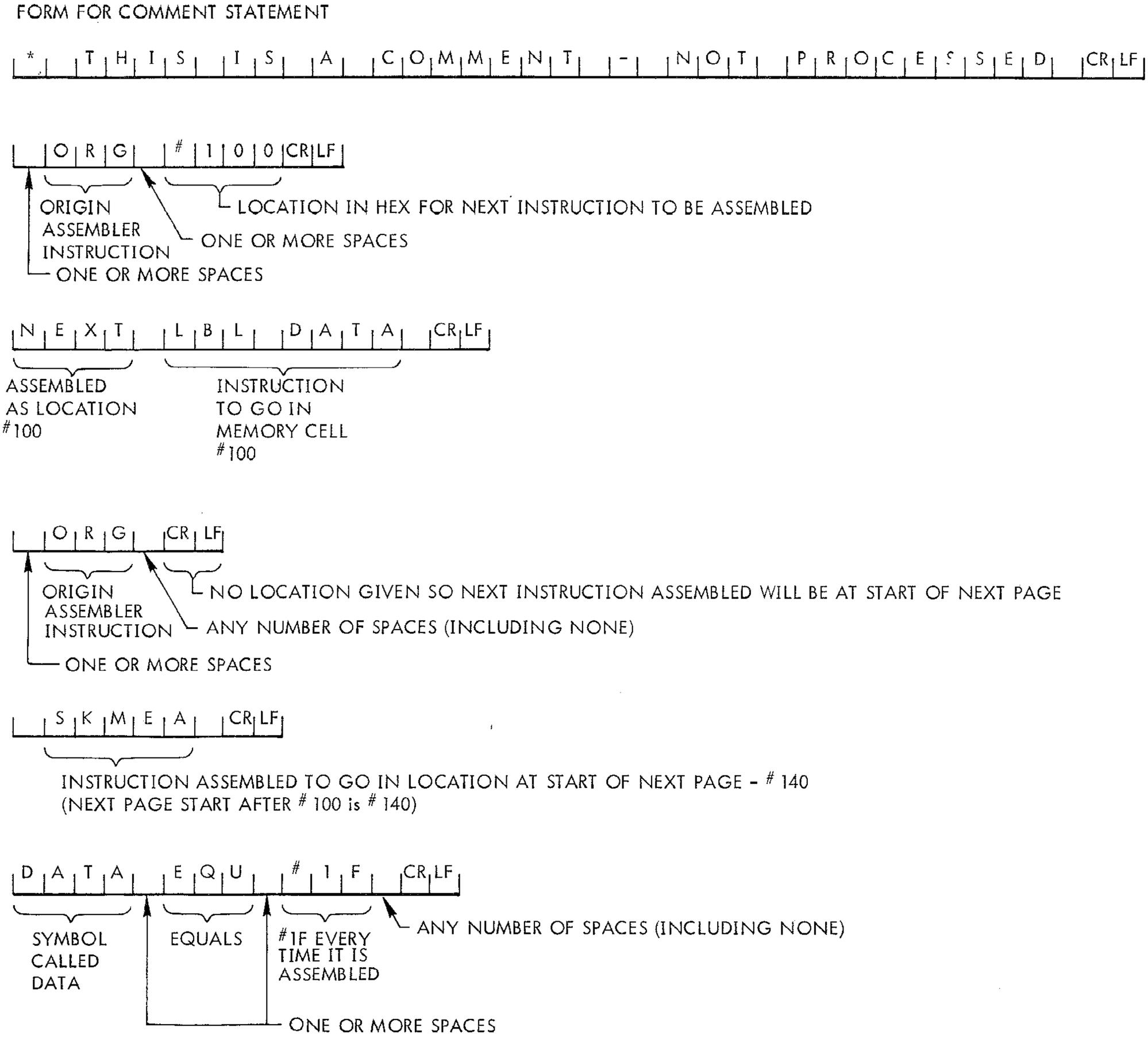


Figure 2-4. ASSEMBLER INSTRUCTION EXAMPLES

2.4.4 OUTPUT OPERATION DESCRIPTION

2.4.4.1 Assembly Listing

Each line of this listing contains four fields; address, generated code, error flag (if any) and source statement image. An example of an assembly listing is shown in Figure 2-5.

2.4.4.2 Object Program

The object program is produced (usually on punched or magnetic tape) in a format suitable for input to the Supervisory Program.

```
                *THIS IS AN 8-BIT A/D CONVERSION ROUTINE.
                *THIS ROUTINE USES THE SUCESSIVE APPROXIMATION
                *METHOD FOR THE CONVERSION.
                *TOTAL CONVERSION TIME:      275 CYCLES
                *INITIALIZATION TIME:         51 CYCLES
                *TIME PER BIT:                28 CYCLES
                *MEMORY LOCATIONS USED:       36 WORDS
                *
                *
                      ORG    #100
0100 27 AD0   LBL    #37    INITIALIZATION
0120 1F
0110 00       NOP
0108 14 AD0A  ROS           RESET DIO'S 0-7
0104 58       DECB
0102 5C
0121 F7       T      AD0A
0130 22 AD0B  LB     2
0118 77       LAI    7      SET N=7
010C 70 AD0C  LAI    0      SET RESULT REG=0
0106 5C       XDSK
0123 F3       T      AD0C
0111 22 AD2   LB     2
0128 53       L      3      LOAD N
0114 44 AD2A  LBA           POINT TO N
010A 00       NOP
0125 10       SOS           SET DIO N
0132 00       NOP
0139 04 AD3   INT0L         KEEP BIT?
013C C7       T      AD4    NO
011E 21 AD5   LB     1      YES, SET LSB RESULT
012F 10       SB     1
0117 22 AD6   LB     2
010B 50       L
0105 6F AD7   AISK   #F     DECR N, DONE?
0122 F1       T      AD8    NO
0131 02       RT            YES
0138 14 AD4   ROS           RESET PORT N
011C E8       T      AD6
010E 5C AD8   XDSK   0       STORE NEW N
0127 0D       RC            LEFT SHIFT RESULT
0113 50 AD8A  L
0109 40       AC
0124 5C       XDSK
0112 EC       T      AD8A
0129 EE       T      AD2    CONVERT NEXT BIT

 28 AVAIL

                      END
```

Figure 2-5. SAMPLE OF MM76 ASSEMBLY LISTING

2.4.5 ASSEMBLER ERROR DIAGNOSTICS

The basic function of the assembler is to translate the source program into machine instructions, and to assign each instruction to a memory location. However, a second function performed by the assembler is to flag programming errors so they may be readily corrected.

For those errors which are detected in Pass 1, the offending statement and the appropriate diagnostic flag are printed during Pass 1 (other statements are not printed during Pass 1). For those errors which are detected in Pass 2, the appropriate diagnostic flag is printed alongside each offending statement.

The following list specifies the errors which are detected, along with the diagnostic flags.

I	Invalid operand
L	TM instruction to location not between #7CO and #7FF for MM77
M	Symbol defined more than once
O	Invalid operation code
P	Program wraps around a page boundary
Q	No label on EQU instruction
S	Syntax error
W	Warning — program expression wraps around end of page
U	Undefined symbol
V	Symbol-table overflow
X	Two or more errors on this statement

2.4.6 STORAGE REQUIREMENTS FOR PPS-4/1 ASSEMBLY

The assembler program occupies 4K words of storage and requires approximately 256 words for working storage.

All data storage beyond the first 256 words is available for the symbol table. Each symbol table entry occupies five words.

Since the basic PPS Universal Assemulator comes with 6K bytes of storage, up to 358 symbols may be used in the source program without requiring any additional PPS-8 memory modules.

2.5 PPS-4/1 ASSEMBLER FOR USE ON A BATCH COMPUTING SYSTEM

2.5.1 INTRODUCTION

The One-Chip Assembler (OCA) is a computer program developed to run as a batch job, on an IBM S/360 or S/370, using the standard IBM operating system (OS). The OCA is written entirely in the FORTRAN IV language.

The capability exists to vary the size of a ROM page. This feature eliminates the problem of program fit during the initial states of development. System macro instructions are also provided to allow automatic generation of CPU instruction sequences with respect to microprogram branching.

2.5.2 GENERAL INFORMATION

2.5.2.1 Character Set

Except where specifically stated, all printable characters on the IBM 029 keypunch can be used in the preparation of source program statements for OCA.

2.5.2.2 Symbols

Symbols are specified by the user and may appear in the label and/or operand fields of source program statements. A symbol is defined as a character string of from one to six characters, one of which must be non-numeric. Any character in the character set may be used in a symbol except for the following:

b	blank
+	plus
-	minus
*	asterisk
'	apostrophe
,	comma
#	crosshatch

2.5.2.3 Self-Defining Terms

A self-defining term is one whose value is inherent in the term. For example, a decimal self-defining term, 12, represents a value of 12. Self-defining terms may appear in the operand fields of certain source statements.

Three different types of self-defining terms are acceptable to OCA: decimal, octal and hexadecimal. Decimal self-defining terms consist of from one to four decimal characters; octal self-defining terms consist of from one to four octal characters preceded immediately by 0' (e.g., 0'777); hexadecimal self-defining terms consists of from one to three hexadecimal characters preceded immediately by # (e.g., #'A6).

2.5.2.4 Expressions

The operand fields of certain instructions may contain expressions. An expression may have one of the following forms:

symbol
symbol + self-defining term
symbol - self-defining term
*
*+ self-defining term
*- self-defining term

OCA maintains its own location counter which is equivalent to the ROM address register in the CPU. It is used to assign ROM addresses to CPU instruction source statements. In the above, the "*" assumes the current value of the assembler location counter at the point in the program where the "*" appears.

2.5.3 DESCRIPTION OF INPUT

The OCA accepts three different types of input from the input stream (SYSIN): assembler instruction statements, system macro statements, and CPU instruction statements. A combination of these statements constitutes an OCA source program.

2.5.3.1 Input Statement Format

The OCA source program is prepared by the user and represents the one-chip microprogram written in a symbolic notation. The input statements are normally punched onto standard 80-column cards, with one statement on each

card. Each input statement consists of five fields: label field, operation field, operand field, comment field, and identification-sequence field. The first four fields are contained within card columns 1 to 72; the identification-sequence field is contained within card columns 73 to 80. The subparagraphs below describe, for each field, its position on the punched card and its purpose.

2.5.3.1.1 Label Field

The label field begins in card column 1. The "EQU" assembler instruction statement requires a symbol in the label field. For CPU instruction statements, a symbol may be placed in the label field at the option of the user.

When a symbol appears in the label field of an input statement, it establishes 1) a symbol address for a RAM or ROM location which can be referenced in the operand field of other input statements; or 2) a symbolic equivalence to a self-defining term which can be referenced in the operand field of other input statements that do not reference ROM or RAM addresses:

An example of the former:

```
RAMADR EQU 1
       :
       :
ROMADR LB RAMADR
       :
       :
       T ROMADR
```

An example of the latter:

```
CONSTI EQU 6
       :
       :
       LAI CONSTI
```

2.5.3.1.2 Operation Field

The operation field must be separated from the label field by at least one blank. If the label field is null (i.e., nonexistent), the operation field may begin in column 2.

The entry in the operation field is a mnemonic which identifies the instruction. The mnemonic may be that for an assembler instruction, system macro, or CPU instruction. Every statement input to OCA must have an operation field entry except for statements which have an "*" in column 1 (i.e., "*" comment statements).

2.5.3.1.3 Operand Field

The operand field begins to the right of the operation field and is separated from it by at least one blank.

Whether an operand field is required, and, if it is, the type of entry that is required, is dependent upon the type of instruction being coded. If an operand field entry requirement exists but the field is null (i.e., blank), a value of zero is assumed.

2.5.3.1.4 Comment Field

The comment field begins to the right of the operand field and is separated from it by at least one blank. If the statement being coded has no operand field requirement, the comment field begins to the right of the operation field and is separated from it by at least one blank. The comment field continues through card column 72. If a statement requires an operand field entry, but no entry is made (i.e., the field is left blank which evaluates to zero) and a comment field entry is desired, the user must precede the comment entry with an apostrophe '''' character. Otherwise, the comment field entry would be misinterpreted by the assembler as the operand field entry which, in most cases, would cause incorrect results.

This field is printed with the statement in the assembly listing, but is otherwise ignored by the assembler. The user may place descriptive text in this field as he desires.

2.5.3.1.5 Identification Sequence Field

This field is printed in the assembly listing but is otherwise ignored. It may be used, at the option of the user, for identification characters and/or sequence numbers.

2.5.3.2 Instructions

There are three classes of instructions accepted by OCA: CPU instructions, system macro instructions, and assembler instructions. The instructions are described in the subparagraphs below, and listed in Table 2-5.

2.5.3.2.1 CPU Instructions

CPU instructions are assembled as binary words to be encoded into the ROM for execution by the CPU. Table 2-5 contains an alphabetized list of all instruction mnemonics.

2.5.3.2.2 Assembler Instructions

Just as CPU instructions are used to request the one-chip microprocessor to perform a sequence of operations during program execution, so assembler instructions are requests to perform certain operations during the assembly process. Assembler instructions, in contrast to CPU instructions, are effective only during the assembly process; they generate nothing in the assembled program and have no effect on the location counter (i.e., other than to provide it with an initial value, see ORG statement).

Table 2-6 presents an alphabetized list of the assembler instructions with a brief description of their purpose. The subparagraphs which follow provide a detailed description of the instructions.

Table 2-5. CPU, MACRO, AND ASSEMBLER INSTRUCTIONS

INSTRUCTIONS								
CPU			MACRO			ASSEMBLER		
OPTAB	L	SKBF	B	TLE		EJECT		
A	LAB	SKISL	BM	TLIH		END		
AC	LAI	SKMEA		TLIL		FILL		
ACSK	LB	SKNC	DECB	TLNE		FLEX		
AISK	LBA	SOS	INCB	TLNC		LOF		
ASK	LSA	TAB	LBL	TM		LON		
COM	LXA	TB		TML		NAME		
DC	NOP		SKAEI	TMLB				
DIN0	OA	X	SKBEI	TNC		OPREF		
DIN1	OB	XAB		TNF		ORG		
DW	OX	XAS	T			PRTROM		
EOB		XAX				PUNCH		
EQU	RB	XDSK	TBF			RAID		
IAM	RC	XNSK	TBT					
IBM	ROS		TC			SECT		
INT0H	RT		TE					
INT1L	RTSK		TIH			SPACE		
IOA	SAG		TIL					
IOS	SB		TL			TITLE		
IX	SC		TLB					
I1	SEG1		TLBF					
I1SK	SEG2		TLBT					
I2C			TLC					

Table 2-6. ASSEMBLER INSTRUCTION DESCRIPTIONS

Assembler Instruction Mnemonic	Brief Description
END	End of source program
EJECT	Eject page on assembly listing
EQU	Equate symbol to value
FILL	Define value to fill unused ROM on emulation tape
FLEXP	Vary number of ROM pages available and/or size of a ROM page
NAME	Define one-chip device to which source program applies
OPREF	Request opcode cross-reference listing
ORG	Initialize OCA location counter
PRTROM	Request ROM pattern listing
PUNCH	Request punch output
SPACE	Print blank line on assembly listing
TITLE	Specify title for assembler-generated print output

In addition to the assembler instructions described below, a 'comment' statement is provided. The 'comment' statement has an asterisk '*' in card column 1 with descriptive text in the other columns. No processing is performed on these statements other than to print them in the assembly listing.

2.5.3.2.2.1 END Assembler Instruction The END statement defines the end of an OCA source program. It must be the last statement in a source program.

Label Field	Operation Field	Operand Field
blank	END	blank

If an end-of-file is encountered by the assembler while reading source statements, the assembler will insert an END statement and proceed. However, an operation field error will be indicated.

2.5.3.2.2.2 EJECT Assembler Instruction The EJECT statement causes a page advance on the assembly listing if lines have already been printed on the current page. The EJECT statement is not printed.

Label Field	Operation Field	Operand Field
blank	EJECT	blank

2.5.3.2.2.3 EQU Assembler Instruction The EQU statement equates a symbol (specified in the label field) to a value. The value that the symbol is equated to is specified in the operand field. The EQU statement can be used to equate symbols to RAM addresses, ROM addresses, and/or self-defining values.

Label Field	Operation Field	Operand Field
symbol	EQU	sdf expression

where sdf = self-defining value

For equating a symbol to a ROM or RAM address use the 'sdf' or 'expression' form. If the 'sdf' form is used, the following is assumed:

ROM page = sdf/64, and
ROM page word = sdf - (ROM page *64)

For example,

ROMADR EQU 0'277

is assumed to refer to ROM page 2, ROM page word 63. That is, if the statement

TR1 ROMADR

is encountered in the source program transfer is assumed to the 63rd word on page 2 (63 is not assumed to be a polynomial counter value).

If the 'expression' form is used and a 'symbol' appears in the expression, the symbol must have appeared in the label field of a previous source statement (i.e., the symbol must be previously defined).

If the operand field is null (i.e., blank), it is assumed to have a value of zero. If the operand field contains only a comma, the symbol in the label field is equated to RAM register 0, word 0.

2.5.3.2.2.4 FILL Assembler Instruction The FILL statement specifies the value to be inserted into unused ROM locations on the emulation tape (see PUNCH statement). If an emulation tape is generated and no FILL statement is encountered, the fill value is zero.

Label Field	Operation Field	Operand Field
blank	FILL	self-defining value

The self-defining value has a range requirement of 0 to 255 decimal. For example.

FILL #FF

specifies that each unused 8-bit ROM location is to be filled with 11111111_2.

2.5.3.2.2.5 FLEXP Assembler Instruction The FLEXP statement permits the number of ROM pages available to be increased by eight and/or the maximum number of words per ROM page to be changed.

The number of available ROM pages is increased by encoding a symbol in the label field of the statement. The maximum number of words per ROM page is changed by specifying the number (via a self-defining value) in the operand field of the statement.

Label Field	Operation Field	Operand Field
blank	FLEXP	blank or self-defining value

A FLEXP statement which contains both a blank label field and blank operand field is ignored. If an operand field entry is encoded, punched output requests (see PUNCH statement), excepting for PUNCH SOURCE, are ignored.

The range of the operand field value, if specified, is 1 to 512 decimal. A value of 64 is equivalent to no operand field specification excepting that punch output requests are ignored.

The FLEXP statement, if used, should immediately follow the NAME statement in the source program.

2.5.3.2.2.6 NAME Assembler Instruction The NAME statement defines the one-chip device to which the source program pertains. The statement must be the first or second statement in the source program; if it is the second statement, it should be preceded by a TITLE statement.

Label Field	Operation Field	Operand Field
blank	NAME	MM76

2.5.3.2.2.7 OPREF Assembler Instruction The OPREF statement requests that an opcode cross-reference listing be produced. The listing provides a concise list of where specified CPU instructions and/or system macro instructions appeared in the source program. From one to ten CPU and/or system macro instruction mnemonics may be selected, by the user, for cross-referencing. The instruction mnemonics selected are encoded in the operand field and separated from one another by a comma.

Label Field	Operation Field	Operand Field
blank	OPREF	One or more CPU and/or system macro instruction mnemonics

For example,

OPREF BM, AISK, LBL

requests that the BM system macros, and DSPY and STRB CPU instructions used in the source program be recorded and a cross-reference listing, which indicates where they were used, be printed following assembly.

2.5.3.2.2.8 ORG Assembler Instruction The ORG statement establishes the ROM location for the next word of text generated by OCA. Subsequent words of text are assigned to ROM locations which follow that specified by the ORG statement.

The value specified in the operand field initializes the OCA location counter. It is expressed as a self-defining value.

Label Field	Operation Field	Operand Field
blank	ORG	self-defining value

The ROM page and word on the ROM page is calculated from the self-defining value (sdf) as follows:

ROM page = sdf/64, and
ROM page word = sdf - (ROM page *64)

It should be noted that the 'ROM page word' is not a polynomial counter value but a binary counter value (see PADR and BADR values respectively in the Assembly Listing description).

If the operand field value is found to be out of range, the ORG statement is ignored. The allowable range is dependent on the device to which the source program pertains. The permissible ORG operand range is #0000 to #03FF.

2.5.3.2.2.9 PRTROM Assembler Instruction The PRTROM statement requests that a ROM pattern in binary listing be output by OCA. Unused ROM locations are printed with zeros.

Label Field	Operation Field	Operand Field
blank	PRTROM	blank

2.5.3.2.2.10 PUNCH Assembler Instruction The PUNCH statement requests that punch card and/or punch tape be produced by OCA.

Label Field	Operation Field	Operand Field
blank	PUNCH	One or more punch keywords separated by a comma

The punch keywords that may appear in the operand field are presented in the table below.

Punch Keyword	Description
ROMBIN	Requests that the ROM pattern be punched onto cards in a binary format acceptable for layout. Unused ROM will contain zeros.
ROMHEX	Requests that the ROM pattern be punched onto cards in a hexadecimal format acceptable to SIMSTRAN. Unused ROM locations will contain zeros.
TAPE	Requests that the ROM pattern be punched onto tape in a format acceptable for input to the SMS ROM emulator. Unused ROM locations will contain the fill code (see FILL statement).
	Note: If the TAPE keyword appears, the PUNCH statement must contain a non-blank comment field or the request will be ignored. The comment field entry will be punched as eyeball characters in the label field on the tape and is limited to 40 characters beginning with the first non-blank.
SOURCE	Requests that a sequenced copy of the source program be punched onto cards.

If the user selects the variable length ROM page size option (see FLEXP statement with non-blank operand field entry), punch requests (excepting for SOURCE) are ignored.

All punch output requests may be contained on one PUNCH statement.

For example:

```
PUNCH ROMHEX, ROMBIN, TAPE   TAPE X12
```

where 'TAPE X12' will be punched in the label field of the punched tape requested by the TAPE keyword. The same results would occur if the requests were made using separate PUNCH statements. For example,

```
PUNCH TAPE TAPE X12
PUNCH ROMBIN
PUNCH ROMHEX
```

2.5.3.2.2.11 SPACE Assembler Instruction The SPACE statement requests that a blank line be printed in its place on the assembly listing. If the SPACE statement is encountered as the first statement following an assembly listing page advance, the statement is ignored.

Label Field	Operation Field	Operand Field
blank	SPACE	blank

2.5.3.2.2.12 TITLE Assembler Instruction The TITLE statement specifies a character string that is to be used on the assembler-generated print output. The TITLE statement, when encountered, causes the current assembly listing page to be ejected if lines have been printed on the current page. The contents of the operand field becomes the title for the new and subsequent pages of print output until another TITLE statement is encountered. The operand field is restricted to 64* characters beginning with the first non-blank and extending through column 72; the field can contain any combination of printable characters and blanks. The TITLE statement is not printed.

Label Field	Operation Field	Operand Field
blank	TITLE	Any character-string

A TITLE statement may be the first statement in a source program (i.e., it may precede the NAME statement).

*For the timesharing systems — 36 characters.

2.5.4 DESCRIPTION OF OUTPUT

The One-Chip Assembler (OCA) produces four types of output: print, punch card, punch tape and simulation file. Each type of output is described below.

2.5.4.1 Print Output

The print output consists of seven listings. The illustration below indicates the listings generated by OCA and the order in which they are output on FORTRAN unit 6 (FT06F001).

ASSEMBLY LISTING
SYMBOL CROSS-REFERENCE LISTING
OPCODE CROSS-REFERENCE LISTING[1]
ROM OVERWRITE LISTING[2]
UNUSED ROM LISTING[3]
ROM BIT PATTERN LISTING[4]
DIAGNOSTIC LISTING

1. Only printed if requested (see OPREF statement).
2. Only printed if ROM overwrite(s) detected.
3. Printed unless terminal error detected.
4. Only printed if requested (see PRTROM statement).

The seven listings, when referred to as a single entity, are known as the "assembler-generated print output". Each page of assembler-generated print output is titled as follows:

OCA	signifies the printed output was produced by the One-Chip Assembler
user-title	blank or characters read from the operand field of a TITLE source statement
mm/dd/yy	date of assembly printed as month/day/year
hh.ss	time of assembly printed as hour.second
PAGE	page number which begins with 1 and is incremented by 1 for each page of printed output

The remainder of what appears on each page of printed output is described on the following page.

2.5.4.1.1 Assembly Listing

BADR — binary address for CPU instruction printed as 4 hex digits.

Note: If a flexible page size is requested (see FLEXP statement) and the word number exceeds the physical maximum for the device, the binary address is expressed by 5 hex digits.

PADR — polynomial address for CPU instruction printed as 4 hex digits.

Note: If the BADR value goes to 5 hex digits (see note above), the PADR will be blank since no polunomial address exists.

TXT — object text for CPU instruction printed as 2 hexadecimal digits (the first digit represents the data for the upper 4 bits of the ROM location, the latter digit represents the data for the lower 4 bits of the ROM location).

ARG —

(a) "BADR value" associated with effective operand address for TR0 and TR1 CPU instructions,

(b) "RAM register, RAM register word" in decimal associated with LB CPU instruction,

(c) "page number" resolved for associated TR CPU instruction,

(d) decimal equivalent to operand field specification for all other CPU instructions having operand field requirements, or

(e) blank for all CPU instructions not having operand field requirements.

STMT — source program statement number in decimal showing the relative position in assembly listing for the source statement.

Note: The source statement is used to reference the assembly listing from the symbol cross-reference listing, opcode cross-reference listing, ROM overwrite listing, and diagnostic listing.

error characters (4 character positions to right of start field) — blank if no errors or warnings associated with statement; otherwise from 1 to 4 characters where:

Character	Meaning
A	operand field entry is wrong type (e.g., LBI instruction references a ROM address)
D	symbol in label field is a duplicate (this usage is ignored)
E	transfer is illegal
I	illegal operation field entry
L	illegal label field entry
M	required label field entry is missing
R	operand field value is out of range
U	undefined symbolic operand field entry
V	illegal operand field entry
W	transfer may be incorrect

SOURCE STATEMENT — source statement image

Note: TITLE, EJECT and SPACE statements are not printed.

2.5.4.1.2 Symbol Cross-Reference Listing

SYMBOL	alphabetical list of symbols appearing in the label fields of source statements
DEFN	number of the source statement in the assembly listing where the symbol appeared in label field (i.e., where the symbol was defined)
REFERENCES	a list of assembly listing source statement numbers where the symbol is used in the operand field.

2.5.4.1.3 Opcode Cross-Reference Listing

OPCODE	mnemonic operation code
REFERENCES	a list of assembly listing source statement numbers where the mnemonic operation code appears in the operation field.

2.5.4.1.4 ROM Overwrite Listing

The ROM overwrite listing is only printed if ROM overwrites are detected during the assembly process (i.e., if more than one word of object text is assigned to the same ROM location). The listing provides a list, by assembly listing statement number, of the overwrites detected; an example is presented below.

STMT 68 OVERWROTE STMT 4

The above example indicates that the object text associated with statement 68 in the assembly listing has replaced the object text associated with statement 4 (i.e., they both are associated with the same ROM address).

A total of overwrites is printed at the end of the ROM overwrite listing.

The message "ROM OVERWRITE(S) DETECTED will appear in the diagnostic listing if overwrites are detected.

2.5.4.1.5 Unused ROM Listing

The unused ROM listing specifies, by ROM page, the number of words that are unused. An example line from the listing is shown below.

PAGE 4: 12 WORDS

A total of unused ROM words is printed at the end of the unused ROM listing.

2.5.4.1.6 ROM Bit Pattern Listing

The ROM bit pattern listing is produced if a PRTROM statement appears in the source program. The listing contains the contents of ROM printed in binary. Eight words of ROM are printed per line. In addition, the page number and word number (in decimal) associated with the first word on the line is printed in the left-hand margin; the right-hand margin of each line contains a card sequence number. Every eight lines, which constitutes one ROM page, are followed by a line containing only a semicolon.

The ROM bit pattern listing, with the exception of the ROM address portion in the left-hand margin, contains data which is identical to that punched on cards when a PUNCH statement appears in the source program with ROMBIN encoded in its operand field.

2.1.4.1.7 Diagnostic Listing

A diagnostic listing is always printed which lists errors or possible errors detected during the assembly process. Table 2-7 lists the diagnostic messages. If no errors or possible errors were detected, the message "***NO ERRORS OR WARNINGS" is printed. Otherwise, a list is printed which contains the following headings:

STMT — Blank or the number of the statement (in the assembly listing) being processed when the error or possible error was detected.

LVL — The severity of the error where;
- 4 = warning
- 8 = error
- 12 = terminal error

Note: If a terminal error is detected:
1) all punch requests excepting for "PUNCH SOURCE" are deleted;
2) all print listings excepting for assembly, cross-reference and diagnostic are deleted; and
3) simulation file output, if requested, is deleted.

DIAGNOSTIC MESSAGE — A brief description of the error detected.

Table 2-7. DIAGNOSTIC LISTING

Message	LVL	Description
OPREF MAX EXCEEDED	4	More than 10 unique CPU operation mnemonics have been requested to be cross-referenced, only the first 10 will be referenced.
CROSS-REFERENCE TABLE OVERFLOW	4	More than 1600 references to ROM/RAM labels and/or CPU operation mnemonics have been made; the reference at this source statement (STMT) and subsequent references are ignored (i.e., no entry is made for them in the cross-referenced table).
DIAGNOSTIC TABLE OVERFLOW	4	More than 200 errors and warnings have been noted; the 200th has been replaced by this entry. Refer to error flags on the assembly listing for further diagnostics.
TRANSFER MAY BE INCORRECT	4	See section 3.2.2 for a complete description.
LSI OUTSIDE SR0, SR1	4	See section 3.2.2 for a complete description.

Table 2-7. DIAGNOSTIC LISTING (continued)

Message	LVL	Description
SYMBOL TABLE OVERFLOW	8	More than 750 symbols defining ROM/RAM locations have appeared in the label fields; this statements label field entry and those associated with subsequent statements will not be entered into the symbol table.
ROM OVERWRITE(S) DETECTED	8	See the ROM overwrite listing.
ILLEGAL LABEL FIELD	8	The label field of the statement contains an illegal, missing or duplicate symbol.
DUPLICATE LABEL	8	The symbol in the label of this statement has already been used; this usage is ignored.
ILLEGAL OPERATION FIELD	8	The mnemonic appearing in the operation field of this statement is unrecognizable; the instruction will be assembled as if it were a no operation "NOP". Note: If no "END" statement is found at the end of a source program, OCA inserts one but flags it as an illegal operation; however, the assembly proceeds as if the "END" statement was present.
ILLEGAL OPERAND FIELD	8	The operand field entry is illegal; see requirements for this statement.
UNDEFINED SYMBOLIC OPERAND	8	The symbol appearing in the operand field of this statement has not appeared in the label field of a source statement.
OPERAND VALUE OUT OF RANGE	8	The operand field entry had evaluated to a value which is outside of the allowable range; a value of 0 will be used.
REQUIRED LABEL MISSING	8	The label field entry for the "EQU" has been found to be illegal, missing or a duplicate.
OPERAND IS WRONG TYPE	8	The operand field makes reference to a RAM address when a non-RAM reference is required, or the operand field requires a RAM reference and a non-RAM reference is made.
TRANSFER IS ILLEGAL	8	See section 3. 2. 2 for a complete description.
NAME STATEMENT ERROR	12	The "NAME" statement operand field incorrect, more than one "NAME" statement or "NAME" statement not 1st or 2nd statement of source program.
ASSEMBLER NOT INITIALIZED	12	"NAME" statement error or "NAME" statement missing from source program; OCA not initialized properly.
ROM TEXT TABLE OVERFLOW	12	More than 2048 actual ROM locations used with flexible page size option selected (see FLEXP).

2.5.4.2 Punch Card Output

Three types of punch card output can be produced by OCA: a copy of the source program, the ROM pattern in binary for layout, and the ROM pattern in hexadecimal. The punch card output, which is output on FORTRAN unit 14 (FT14F001), is requested by the "PUNCH" statement.

2.5.4.2.1 Source Program Punch Card Output

A copy of the source program input to OCA will be punched onto cards if a "PUNCH" statement, with "SOURCE" encoded in the operand field, is encountered. The new source program will be sequenced in card columns 73-80 by 10 beginning with 10.

2.5.4.2.2 ROM Pattern in Binary Punch Card Output

The assembled ROM pattern will be punched in binary if a "PUNCH" statement, with "ROMBIN" as an operand, is processed. Each card is sequenced and contains data for eight ROM locations punched as binary characters. The first card begins with the data associated with word 0 of page 0; the data which follows is that for filling contiguous ROM words. Every eight data cards (which contain data sufficient to fill one ROM page, are followed by a semi-colon card. The last semi-colon card is followed by a blank card.

An exact replica of the ROM pattern in binary punch card output can be printed by using a "PRTROM" statement.

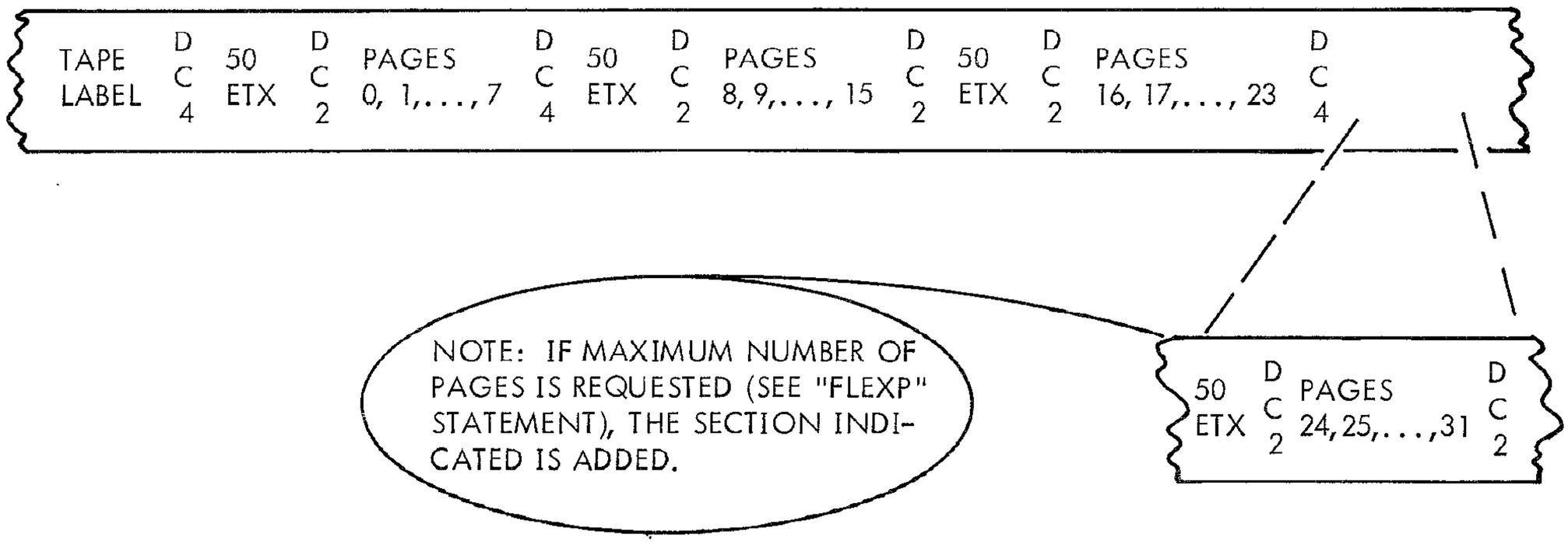
2.5.4.2.3 ROM Pattern in Hexadecimal Punch Card Output

The assembled ROM pattern will be punched in hexadecimal if a "PUNCH" statement, with "ROMHEX" encoded in the operand field, is encountered in the source program. The first card punched contains "L000". The subsequent cards each contain data for 16 ROM locations; the data for an individual ROM location being punched as 2 hexadecimal characters. The ROM page and page word (in decimal), associated with the first of the 16 data words on the card, are punched in card columns 77,78 and 79,80 respectively. The last data card is followed by a blank card.

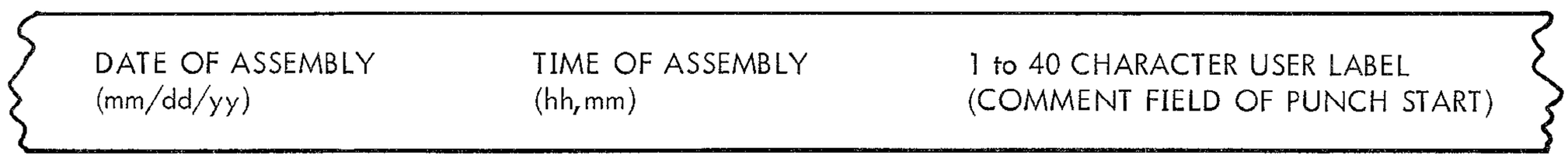
2.5.4.3 Punch Tape Output

The assembled ROM pattern can be punched onto tape for use on the Signetics Memory System's (SMS) ROM emulator which has 512 X 8 bit memory units. This output is selected by the "PUNCH" statement with "TAPE" encoded in the operand field. The ROM information is encoded on the tape as a pair of hexadecimal ASCII coded characters. Each character pair is separated by an apostrophe character, the apostrophe representing a load command. Figure 2-6 depicts the tape formats for the punch tape which is output on FORTRAN unit 13 (FT13F001).

MICROCOMPUTE DEVICE



TAPE LABEL FORMAT (280 FRAMES)

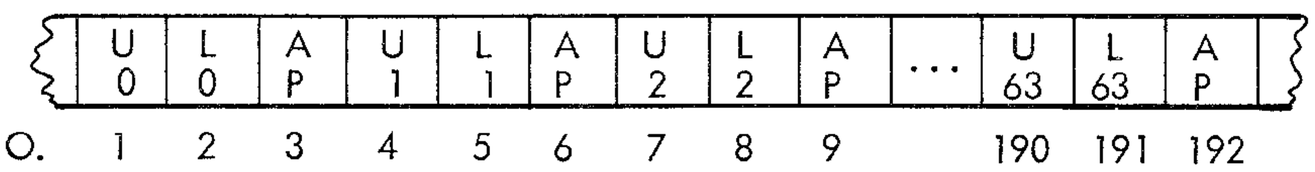


DC4 = 1 FRAME (ASCII DC4, TAPE OFF)

50 ETX = 50 FRAMES (ASCII ETX's)

DC2 = 1 FRAME (ASCII DC2, TAPE ON)

ROM PAGE FORMAT (192 FRAMES/PAGE)



WHERE UX = UPPER 4-BITS OF A WORD,

LX = LOWER 4-BITS OF A WORD,

AP = ASCII APOSTROPHE

Figure 2-6. OCA PUNCH TAPE FORMATS

SECTION III
PPS-4/1 PROGRAMMING TECHNIQUES

3.1 INTRODUCTION

Any application of the MM76 microcomputer system will require a program to be written to make the system function.

3.1.1 FUNCTIONAL GROUPS

The material in this section is organized into seven basic functional groups. These groups include: (1) data address modifications — showing how the B Register can be modified automatically or selectively so that the desired data can be manipulated, (2) data transfer techniques — methods of moving data from one point to another for processing or organizing it, (3) arithmetic operations — techniques for both binary and decimal arithmetic operations, (4) bit operations — methods for manipulating data as individual bits and methods for obtaining logical combinations of those bits, (5) control transfers — techniques for modifying the normal counting sequence of the program counter for unconditional branching and conditional branching — which includes methods for changing the sequence of computations depending upon the results of tests executed as part of the program, (6) CPU register manipulation and comparison functions — techniques for transferring information from point to point within the CPU — including techniques for temporarily storing data, and techniques for comparing internal registers, and (7) basic input/output operations — including the use of the input/output instructions and discrete input and output capabilities. Subroutine usage (including techniques for calling routines, setting up data addresses, returning from subroutines and nesting of subroutines) and techniques for efficient memory assignment are covered by various examples used to simultaneously show other capabilities of the MM76 instruction set. These examples occur where most appropriate and consequently are scattered through the text.

The final part of this section shows examples of a number of typical software functions including keyboard displays, 12-hour clock, time delays, etc.

3.1.2 MULTIFUNCTION INSTRUCTIONS

Because the MM76 instructions were designed to facilitate the most often needed functions, many of the instructions will be included in more than one of the above categories. For instance, many of the instructions in the MM76 set will automatically test and modify the program sequence as necessary while performing other functions such as moving data or arithmetic operations, etc. Other instructions have been designed to minimize the amount of memory required for executing common sequences of operation. Thus, the MM76 system is efficient not only in interfacing conveniently with external systems, but also is organized to effectively be a much faster microcomputer than either the 4-bit data word length or the nominal 80,000 complete clock cycles per second would indicate. Additional data on multifunction instructions is provided in Paragraph 3. 2. 9.

3.1.3 MM76 INSTRUCTION SET DESCRIPTION

The MM76 instruction set is summarized in Table 3-1.

3.2 DATA ADDRESS MODIFICATION

The key to the operation of any microcomputer including the MM76 microcomputer is the operation of the data memory since efficient data memory functioning generally leads to efficient processing. Before discussing the specifics of the MM76 memory it may be useful to discuss data memory concepts.

3.2.1 MEMORY REGISTER CONCEPT

The data memory may be used in a variety of ways. It may be used as individual bits to indicate status or logic states; it may be used as a single four-bit word to indicate a count, or a single binary coded decimal (BCD) character; or it may be used in groups of four bit words to represent binary or BCD numbers or to represent coded data such as alphanumeric information in ASCII or EBCDIC codes.

Most representations of parameters used in performing a computational or timing function in a microcomputer will use groups of words to represent each parameter. Such a group of words (of any length) is termed a memory register in this document. Typical memory register assignments are shown in Table 3-2.

The MM76 microcomputer memory may be used in any manner desired by the programmer, but the instruction set is extremely efficient if the memory register assignments for parameters less than 16 words long are arranged in a continuous set of words in one row of the memory. Thus, a five word parameter could be assigned to many locations in memory. For instance, it could be in words 0 thru 4, 1 thru 5, 2 thru 6, 3 thru 7, 4 thru 8, 5 thru 9, 6 thru A, 7 thru B, 8 thru C, 9 thru D, A thru E, or B thru F of any row in memory and still be efficiently handled. Some of the possible assignments are shown in Figure 3-1.

Table 3-1. MM76 INSTRUCTION SET

Op Code	Bytes	Cycles	Description
RAM Addressing Instructions			
XAB	1	1	Exchange Accumulator with B Lower (least significant 4 bits)
LBA	1	1	Load B Lower from Accumulator
*LB**	1	1	Load B Upper with zero and B Lower with immediate field
*EOB**	1	1	Exclusive OR B-Upper with two bit immediate field
†*LBL**	2	2	Load B Register Long with 6 bits (4 bits 1st byte, 2 bits 2nd byte) immediate field. This instruction should not be skipped.
†*INCB	2	2	Increment B Lower and modify B Upper with 2 bit immediate field; Skip if BL counts to 0. This instruction should not be skipped.
†*DECB	2	2	Decrement B Lower and modify B Upper with 2 bit immediate field; Skip if BL counts to 15. This instruction should not be skipped.
Bit Manipulation Instructions			
*SB	1	1	Set bit in word in memory. Specific bit designated by 2 bit immediate field and specific word addressed by B Register
*RB	1	1	Reset bit in word in memory. Specific bit designated by 2 bit immediate field and specific word addressed by B Register
*SKBF	1	1	Skip on designated bit in addressed memory when bit is false (zero). Bit is selected by 2-bit immediate field.

Table 3-1. MM76 INSTRUCTION SET (continued)

Op Code	Bytes	Cycles	Description
Register to Register Instructions			
XAS	1	1	Exchange Accumulator and S Register contents
LSA	1	1	Load S Register from Accumulator
Register Memory Instructions			
*L	1	1	Load Accumulator from memory and modify B Upper with 2-bit immediate field
*X	1	1	Exchange Accumulator with memory and modify B Upper with 2-bit immediate field
*XDSK	1	1	Exchange Accumulator with memory and modify B Upper with 2-bit immediate field; Decrement B Lower and skip if BL counts to 15
*XNSK	1	1	Exchange Accumulator with memory and modify B Upper with 2-bit immediate field; increment B Lower and skip if BL counts to 0
Arithmetic Instructions			
A	1	1	Add memory to Accumulator (carry not used or set)
AC	1	1	Add memory and carry to Accumulator; form sum and carry
ACSK	1	1	Add memory and carry to Accumulator; skip if No carry is generated
ASK	1	1	Add memory to Accumulator and skip if No overflow occurs (carry not used or set)
DC	1	1	Decimal correct (same as AISK 6 so it must always be followed by NOP)
COM	1	1	Complement Accumulator
RC	1	1	Reset carry
SC	1	1	Set carry
SKNC	1	1	Skip on no carry
*LAI***	1	1	Load Accumulator with contents of immediate field
*AISK	1	1	Add accumulator and immediate field, skip on no overflow. No carry is set or used.
ROM Addressing Instructions			
RT	1	2	Return from subroutine
RTSK	1	2	Return from subroutine and Skip first instruction of one or two bytes in length. Do not skip macro instructions marked with a †
T	1	2	Transfer on-page to 6 bit immediate field location
NOP	1	1	No operation

Table 3-1. MM76 INSTRUCTION SET (continued)

Op Code	Bytes	Cycles	Description
ROM Addressing Instructions (continued)			
TL	2	3	Transfer Long off page to pages 0 through 7 (addresses #000 thru #1FF)
TM	1	2	Transfer and Mark to special subroutine pages SR0 (addresses #3C0 thru #3FF)
TML	2	3	Transfer and Mark Long to subroutine on pages 0 through 7 (addresses #000 thru #1FF)
Logical Comparison Instructions			
SKMEA	1	1	Skip on memory equals Accumulator
*SKBEI	2	2	Skip on B Lower equals immediate field. ** Should not be used on subroutine pages.
*SKAEI	2	2	Skip on Accumulator equals immediate fiels. ** Should not be used on subroutine pages.
Input/Output Instructions			
SOS	1	1	Set output, bit selected by B Lower (B Upper = 3) to VSS
ROS	1	1	Reset output, bit selected by B Lower (B Upper = 3) to -V
SKISL	1	1	Skip on Input Selected Low on bit selected by B Lower (B Upper = 3) Bit sampled during prior cycle.
IBM	1	1	Input Channel B ANDed with Accumulator, results to Accumulator
OB	1	1	Output from Accumulator to Channel B
IAM	1	1	Input Channel A, ANDed with Accumulator, results to Accumulator
OA			Output from Accumulator to Channel A
IOS	1	1	Serial input and output from S – shifting takes 8 cycles concurrent with other instruction operations
I1	1	1	Input Channel 1 to Accumulator
I2C	1	1	Input Channel 2 to Accumulator and complement
INT1	1	1	Skip on INT1 equals VSS on input pad
DIN1	1	1	Skip if INT1 flip-flop is reset and set INT1 flip-flop
INT0	1	1	Skip on INT0 equals -V on input pad
DIN0	1	1	Skip if INT0 flip-flop is reset and set INT0 flip-flop
SEG1	1	1	Decode combined memory, Carry flip-flop, and Accumulator and output 4 bits to Channel A
SEG2	1	1	Decode combined memory, Carry flip-flop, and Accumulator and output 4 bits to Channel B

Table 3-1. MM76 INSTRUCTION SET (continued)

Op Code	Bytes	Cycles	Description
Conditional Transfer Instructions†			
† TC	2	3-2	Transfer within page on Carry Set
† TNC	3	4-3	Transfer within page on No Carry Set
† TLC	3	4-3	Transfer Long on Carry Set
† TLNC	4	5-3	Transfer Long on No Carry Set
† TBF	3	4-3	Transfer within page on Bit in Memory False
† TBT	2	3-2	Transfer within page on Bit in Memory True
† TLBF	4	5-3	Transfer Long on Bit in Memory False
† TLBT	3	4-3	Transfer Long on Bit in Memory True
† TE	3	4-3	Transfer within page on Accumulator Equals Memory
† TNE	2	3-2	Transfer within page on Accumulator Not Equal Memory
† TLE	4	5-3	Transfer Long on Accumulator Equals Memory
† TLNE	3	4-3	Transfer Long on Accumulator Not Equal Memory
† TIH	2	3-2	Transfer within page if input selected by B Lower is High
† TIL	3	4-3	Transfer within page if input selected by B Lower is Low
† TLIH	3	4-3	Transfer Long on Input Selected High
† TLIL	4	5-3	Transfer Long on Input Selected Low

*The immediate field (IF) is two, four or six bits which are included as part of the 8-bit instruction. If not specified otherwise the immediate field is 4 bits.

**When LB, EOB or LBL instructions appear in sequence as a string of LB, EOB or LBL or mixtures of LB and LBL instructions only the first one of them will be executed. The remainder of the LB, EOB or LBL instructions in the sequence will be ignored except that the first EOB after an executed LB instruction will also be executed.

***When more than one LAI instruction occurs in sequence, only the first LAI instruction encountered will be executed. The remainder of the LAI instructions in the string will be ignored.

†These are all macro instructions which must not be preceded by an instruction which executes a skip. For the conditional transfer instructions the first number in the cycles column indicates number of cycles when condition is met and second number indicates number of cycles when the condition for transfer is not met.

Numbers preceded by # are hexadecimal numbers.

Note 1: Whenever an instruction is skipped (i. e., ignored) the number of cycles required is equal to the number of bytes in the skipped instruction (e. g. TL takes two cycles to skip).

Note 2: A subroutine called by a TM may not execute an SKBEI or SKAEI when they are followed by a transfer from the extension page (SR1) to the entry page (SR0) prior to executing the return (RT or RTSK) instruction.

Table 3-2. MEMORY REGISTER CONCEPT

MEMORY REGISTER IS A GROUP OF WORDS CONSIDERED TO BE ONE PARAMETER. EXAMPLES:

- ONE DIGIT: 4 (0100) A (1010) 9 (1001) 0 (0000) } BCD OR HEX
- MANY DIGITS: 1 2 3 4 5 (0001 0100 0011 0100 0101) HOURS; MINUTES — 0 0 1 (0000 0000 0001) SECONDS — F F F F (1111 1111 1111 1111) PM
- CODED SIGNIFICANCE: 0 3 (0000 0011); 4 5 (0100 0101); 2 2 (0010 0010); 1 (0001) (0 = AM)

	E	D	I	G	H	
	5 3 (0101 0011)	4 D (0100 1101)	4 9 (0100 1001)	5 4 (0101 0100)	4 8 (0100 1000)	ASCII
	E 2 (1110 0010)	D 4 (1101 0100)	C 9 (1100 1001)	E 3 (1110 0011)	C 8 (1100 1000)	EBCDIC

+ 3 1 4 2 (0000 0011 0001 0100 0010) +3. 142 BCD FIXED DECIMAL

- 6 8 5 8 (1111 0110 1000 0101 1000) — + 2 0 (0000 0010 0000) -3.142×10^{20} 10's Complement Exponential

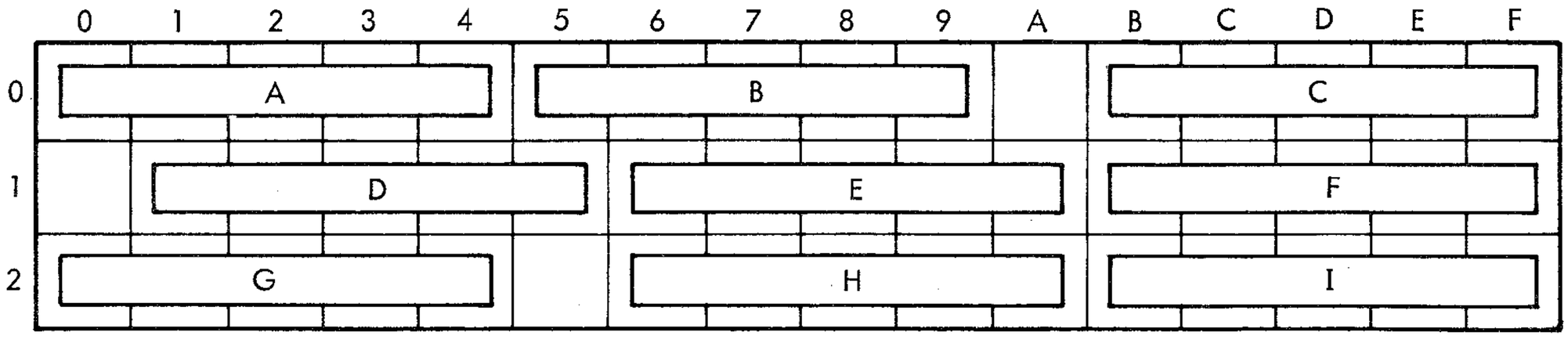


Figure 3-1. POSSIBLE ASSIGNMENTS FOR FIVE WORD MEMORY REGISTERS

3.2.2 DATA ADDRESSING

In the MM76 microcomputer the data in the memory registers are selected, four bits at a time, by the Data Address Register termed the B Register. The B Register is six bits long and consequently can address 64 unique addresses, even though there are only 48 actual memory cells. This is discussed in more detail in Paragraph 3.3.8.

The B Register for the MM76 is shown in Figure 3-2. The B Register is organized into two segments: BU (bits 6 and 5) and BL (bits 4 through 1). The upper portion, BU, can be thought of as addressing rows of memory and the lower portion, BL, can be considered as addressing one of the 16 words in that row. The bits in the B Register may be modified in groups of 2, 4 or 6 bits as shown in Table 3-3. It can be seen from the data memory map (Figure 3-3) that B Upper addresses a row and B Lower selects the column within that row.

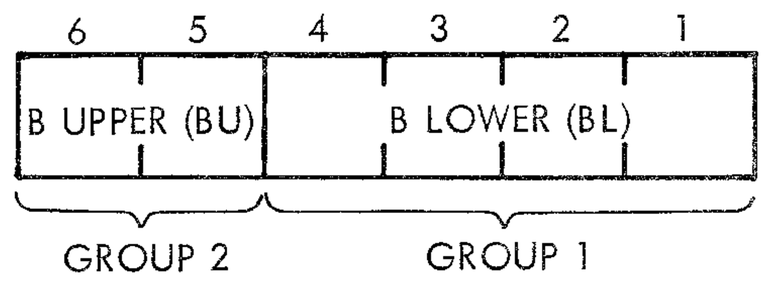


Figure 3-2. ADDRESS REGISTER (B) ORGANIZATION

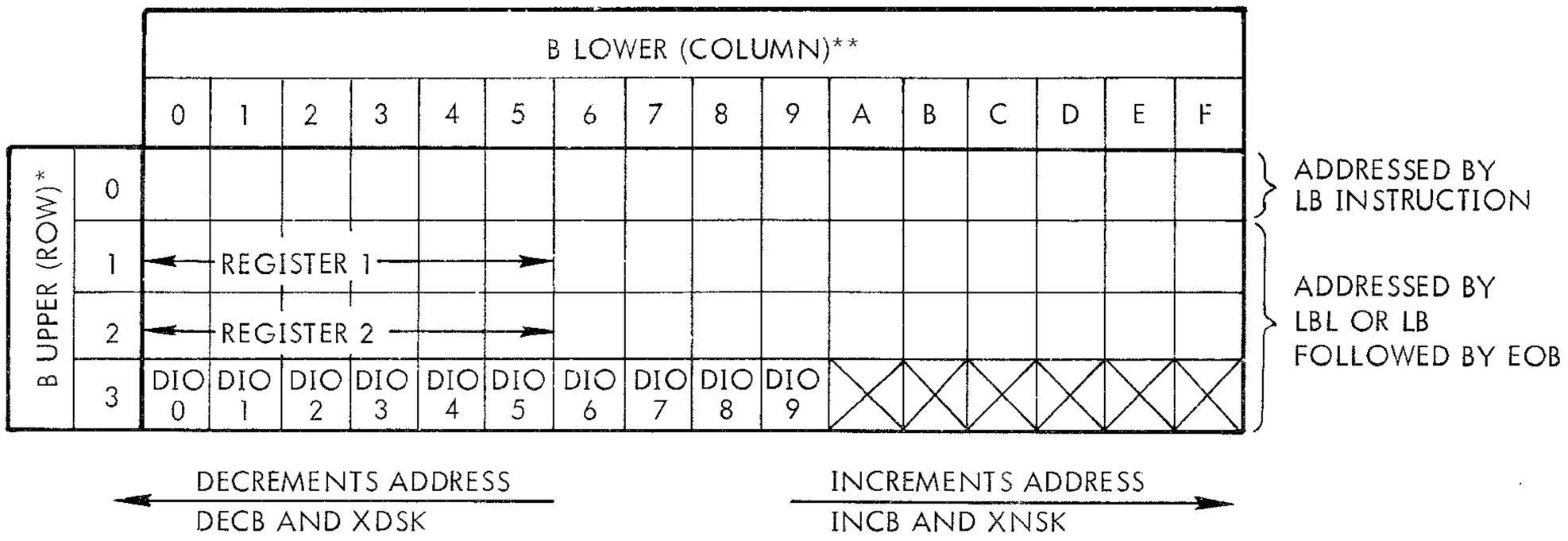
Table 3-3. DATA ADDRESS MODIFICATION

Instruction		Address Bits Modified		Condition to Automatically Terminate Process
Op Code	Immediate Field	Group 2 2 Bits of BU**	Group 1 BL	
XAB*	—	—	Exchanged with A	—
LBA*	—	—	Replaced with A	—
LBL	Upper/Lower	Replaced (2nd)	Replaced (first)	—
LB	Lower	Zero	Replaced	—
EOB	Upper	Exclusive OR	No Change	—
INCB*	2 bit Upper	Exclusive OR	Increment and Test for 0	BL counts to 0
DECB*	2 bit Upper	Exclusive OR	Decrement and Test for #F	BL counts to #F
L	2 bit Upper	Exclusive OR	No Change	—
X	2 bit Upper	Exclusive OR	No Change	—
XDSK*	2 bit Upper	Exclusive OR	Decrement and Test for #F	BL counts to #F
XNSK*	2 bit Upper	Exclusive OR	Increment and Test for 0	BL counts to 0

*In the cycle immediately following these instructions, the contents of BU and BL Registers are correct as modified. However, in the cycle immediately following the address and modification instruction, instructions which address memory will have B Upper modifications completed but the old value of BL will be used in forming the effective memory address.

**RAM-DI/O Timing. When changing BU from addressing RAM (0, 1, or 2) to DI/O (3), the B Register value is updated in the cycle immediately following the modification instruction, but neither RAM nor DI/O accessing instructions are valid. In the second cycle following, the DI/O selected by the modified BU and BL may be set, reset, or tested. When changing BU from addressing DI/O (3) to RAM (0, 1, or 2) the B Register value is updated in the cycle immediately following the modification and RAM addressing instructions are valid (subject to the timing related to changing BL) except for SB, RB, SKBF instructions. During the one cycle immediately following changing BU the SB and RB instructions will set or reset a bit in RAM as well as the DI/O bit. The SKBF instruction is undefined during this cycle. In the second cycle following, these three instructions are valid.

The primary instructions for initially loading the address into the B Register and doing some specialized modification of the addressing are shown in Table 3-4, Data Address Modification Instructions. Figure 3-3 shows the effective areas of data memory affected by each of the data address modification instructions.



*Row Address Modification can be performed by L, X, XDSK, or XNSK Instructions.
**BL may be directly loaded from the Accumulator by XAB or LBA without affecting BU.

Figure 3-3. DATA MEMORY OR DISCRETE I/O (DI/O) ADDRESSING

3.2.3 LOAD B (LB) INSTRUCTION

The Load B (LB) instruction provides a one-byte, one cycle capability of causing the B Register to point to the specified memory cell in row zero of data memory. Specifically, the upper portion of the B Register (BU) is set to zero and the lower portion (BL) is set to the value specified as part of this instruction. The upper four bits of the instruction (0010) identify that this is a LB instruction and the lower four bits (xxxx) specify the value that gets loaded into BL. Thus the LB instruction is identified in hexadecimal notation as any one of the following instructions: #20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 2A, 2B, 2C, 2D, 2E, or 3F. The #2A instruction causes the upper portion of the Data Address Register, BU to be set to 00, and the lower portion BL to be set to 1010. When instructions are used which reference data memory the contents of memory cell #0A would be involved until some other instruction changes the B Register address. The LB instruction may be used in concert with other instructions which have the capability of modifying the data address in the B Register. These other instructions include other LB instructions, EOB (Exclusive OR B) instructions and LBL (Load B Long) instructions are discussed in Paragraphs 3. 2. 4 and 3. 2. 5.

Table 3-4. DATA ADDRESS MODIFICATION INSTRUCTIONS

Mnemonics	Op Code (Hexadecimal & Binary	Name	Description	Symbolic Equation
LB	20-2F 0010 ----	Load B (1 cycle-1 byte)	Load BU with zero and BL with four bit immediate field. See notes.	BU ← 0 BL ← I(4:1)
EOB	1C-1F 0001 11--	Exclusive OR B (1 cycle-1 byte)	Exclusive OR the 2-bit immediate field with the two bits in BU. See notes.	BU ← I(2:1) ⊕ BU
LBL	1st word: 20-2F 0010 ---- 2nd word: 1C-1F 0001 11--	Load B Long (2 cycles-2bytes)	Load the 4 bit immediate field in I_1 into BL and the 2 bit immediate field in I_2 into BU. Do not skip this instruction, it is a macro instruction (LB, EOB), See note below.	BL ← I_1(4:1) BU ← I_2(2:1) Do not skip this instruction
XAB	46 0100 0110	Exchange A and BL (1cycle overlapped -1byte)	Exchange the four bit contents of A with the four bit contents of BL	BL ↔ A (BL → A 1st cycle A → BL, 2nd cycle)
LBA	44 0100 0100	Load BL from A (1 cycle overlapped -1 byte)	Load BL with the four bit contents of the A Register. Leave contents of A Register unchanged	BL ← A
INCB	1st word: 58 0101 1000 2nd word: 54-57 0101 01--	Increment B (2 cycles overlapped -2 bytes)	Increment BL, Skip after next cycle (ignore) next instruction if BL increments from 1111 to 0000. The RAM address in the B Register is modified by Exclusively ORing the two-bit immediate field in I_2 with two bits in the BU Register of B(6:5). Do not skip this instruction, it is a macro instruction (X, XNSK)	BU(6:5) ← I_2(2:1) ⊕ BU (6:5) BL ← BL + 1 after next cycle. Do not skip this instruction
DECB	1st word: 58 0101 1000 2nd word: 5C-5F 0101 11--	Decrement B (2 cycles overlapped -2 bytes)	Decrement BL after the next cycle, skip (ignore) next instruction if BL decrements from 0000 to 1111. The RAM address in the B Register is modified by Exclusively ORing the two-bit immediate field in I_2 with two bits in the BU Register, B(6:5). Do not skip this instruction, it is a macro instruction (X, XDSK).	BU(6:5) ← I_2(2:1) ⊕ BU(6:5) BL ← BL -1 after next cycle. Do not skip this instruction

Other instructions which modify data addresses addresses, as well as perform other primary functions are: L, X, XDSK, XNSK } See Table 3-5

NOTES: Only the first occurrence of an LB, EOB, or LBL instruction in a consecutive string of mixed LB, EOB, or LBL instructions will be executed except that EOB following an executed LB will also be executed, (= LBL). The program will ignore the remaining LB, EOB or LBL and execute the next valid instruction.

I (4:1) denotes bits 1 through 4 of the instruction.

B (6:5) denotes bits 5 through 6 of the B Register.

3.2.4 EXCLUSIVE OR B (EOB) INSTRUCTION

The Exclusive OR B (EOB) instruction is used to modify the upper portion of the B Register (BU) by "Exclusively ORing" the two bit immediate field, which is part of the 8-bit instruction, with the BU portion of the B Register. The Exclusive OR function is used since it provides more addressing flexiblity than would have been possible by another function such as replace. For instance, if the immediate field of the EOB instruction were 1 (01), replacement would only allow the upper register to be set to 1. However, by using the Exclusive OR function, the resulting row address can be any value from 0 through 3 depending upon the initial value of BU:

Initial BU	00	01	10	11
EOB 1	01	01	01	01
Resulting BU	01	00	11	10

The Exclusive OR function (when both bits are the same, the result is zero, and if they are different, the result is one) has an additional feature. By using the Exclusive OR function to sequentially point to two memory registers repeatedly, the same Exclusive OR value may be used to do the alternating. In the preceding example, 0 was Exclusively ORed with 1 to point to a register in row 1. When it is necessary to point back to the register in row 0, the EOB 1 instruction may also be used. This relationship is only true with the Exclusive OR function. Another function such as "replace" or "add" would require two different values to be used for the immediate field. An LB instruction which selects a column in data memory, followed by an EOB instruction which selects a row, causes the B Register to be set to any selected value.

3.2.5 LOAD B LONG (LBL) INSTRUCTION

The Load B Long (LBL) instruction is a combination of an LB instruction, followed by an EOB instruction. Because of this, the LBL instruction should not be skipped unless the programmer wishes to skip the LB protion and execute the EOB portion.

The LBL instruction is particularly useful to a programmer when symbolic addresses are being used for data memory. If DATA is defined as a specific memory cell to the assembler by a DATA EQU #27 assembler instruction, the instruction LBL DATA will cause an LB 7 and an EOB 2 instruction to be generated.

3.2.6 INTERMIXED LB, EOB AND LBL INSTRUCTIONS

All of the MM76 microcomputers have the property that when a string of intermixed LB, EOB, or LBL instructions are encountered, only the first one encountered will be executed. The subsequent instructions in the string will be examined and ignored until an instruction which is not an LB, EOB, or LBL is encountered.

The value of this feature can be illustrated with a few examples. Suppose, for instance, that a program which needs to shift one of three specific registers in RAM to the left by 4 bits is desired. One way of programming this would be to write a program to accomplish this three times, each with a different LBL argument, and then the programmer could transfer control to the appropriate set of instructions. Another way to do this would be to use an LBL instruction prior to transferring to the subroutine, but then each time the subroutine was used, the LBL instruction would have to be written, which would require two bytes of information for each usage.

By using the technique available in the MM76 in which only the first addressed LBL in a string is executed and the others are ignored, the routine has only to be written once and the particular LBL instruction has only to be written once, so that a significant savings in program memory is achieved. A program which would meet this objective is as follows:

```
SHFZ    LB      #F      SETUP
SHF1    LBL     #1F
SHF2    LBL     #2F
A1      XDSK            ! CHARACTER SHIFT
        T       A1
        :
        :
```

If the program transfers to SHFZ, the LB #F instruction is obeyed, but the LBL #1F, and the LBL #2F instructions are ignored so that the first digit to be loaded into the Accumulator would be the 4 bits in memory cell #0F and the process would continue from there. If time is important, the most frequently used entry should be the last LBL in the sequence as the microcomputer steps through each of the LBL instructions in sequence and then ignores it. The functioning of the actual character shift routine using the L, XDSK, and T is discussed in example 5 of Paragraph 3.3.6.

If time is important, a slightly faster routine which illustrates the intermixing more completely may be written as follows:

```
SHFTC   LBL     #27
SHFTB   LBL     #34
SHFT1   EOB     1
SHFT2   EOB     2
SHFT3   EOB     3
SHFTA   LB      6
SHFTZ   L
A1      XDSK
        T               A1
        :
        :
```

In this example, the program must load the B Register with an LB #F instruction prior to executing a SHFT1, SHFT2, SHFT3, or SHFTZ subroutine call. For these entries, the set up routine is faster because only one-byte EOB instructions are ignored rather than the two-byte LBL instructions used in the earlier illustration. Also in this example, it is possible to shift shorter registers than the full 16 characters by entering the program at the SHFTA, SHFTB, or SHFTC entry points. With these entry points it is not necessary to have initialized the B Register with the LB #F instruction since the starting point is in columns 6, 4, or 7 respectively, not F. The placement of the SHFTA entry suggests that the seven character register in memory cells #00 through #06 might be referenced more frequently than most of the other registers since no time will be lost in sequencing through ignored instructions.

3.2.7 ACCUMULATOR AND B LOWER MANIPULATION INSTRUCTIONS (XAB AND LBA)

There are two instructions which allow the BL portion of the B Register to be loaded from the Accumulator. One of these, the Load B Lower from Accumulator (LBA) instruction, does not save the prior contents of BL; the other one, Exchange the Accumulator with B Lower (XAB) does save it by replacing the contents of the Accumulator with the prior value in BL.

These instructions are generally used in PPS-4/1 applications when digits in a register are manipulated individually with some processing which references other portions of data memory interspersed between the manipulations. Typical operations of this type might be for keyboard/display routines. The saved BL value may be stored in the S Register without having to reference memory.

In a way these instructions differ greatly from the LB or LBL manner of loading BL since the LB or LBL instructions load fixed value BL directly from ROM. The XAB and LBA instructions may load computed values. The XAB instruction initially loads Accumulator from BL while the contents of Accumulator are routed through the arithmetic unit. The original contents of Accumulator are then inserted into BL so that the exchange is complete after one cycle so the old value of BL will apply for 1 cycle then the new value may be used as follows:

N	XAB	
N + 1	[ ]	old BL is in A and (old A) is in BL, but old BL addresses data memory or I/O
	[ ]	new BL (old A) may access data memory

This is an example of what is termed overlapped instructions. The full process defined by the instruction is overlapped with the next instruction in sequence so that the effective execution time is one instruction cycle although the process is not complete until the next cycle.

The LBA instruction, however, makes use of the same path so its operation is as follows:

N	LBA	
N + 1	[ ]	(old A) is in BL, but old BL addresses data memory or DI/O's
	[ ]	new BL (old A) may access data memory or DI/O's

Examples of programs using these instructions will be found in later sections.

3.2.8 INSTRUCTIONS WHICH COUNT IN THE B LOWER REGISTER (INCB AND DECB)

The Increment B Lower (INCB) and Decrement B Lower (DECB) instructions cause the BL Register to count up or to count down respectively. Both of these instructions are actually macro instructions made up from two machine instructions. The INCB instruction is an X instruction followed by an XNSK instruction and the DECB macro is actually an X instruction followed by an XDSK instruction. These instructions which make up the two macros are discussed in detail in paragraphs starting at Paragraph 3. 3.

The primary use for the INCB and DECB instructions is to cause the B Register to point to the adjacent memory cell without transferring any data. If data is to be transferred between the Accumulator and the addressed memory cell and the address counted up or down, then one of the multifunction instructions XDSK or XNSK is used instead.

In addition to just counting, these instructions also have the capability of modifying the row address and automatically testing for counter overflow. The multifunction aspect of the INCB and DECB instructions is discussed in more detail in Paragraphs 3. 3. 1, 3. 3. 2, and 3. 3. 3.

Since these instructions are macro instructions any attempt to skip a DECB or INCB instruction will only skip the first of the two instructions which make up the macro. Consequently, these instructions should never be preceded by

an instruction which will actually execute a skip. As described in Paragraph 3.3.2, the incremented or decremented address is not available until the second cycle time after the instruction as these both use overlapped instructions. Any use made of the B Register during the first cycle time will reference the old value of BL.

3.2.9 MULTIFUNCTION INSTRUCTIONS DATA MANIPULATION

Much of the power of the MM76 instruction set comes from a group of multifunction instructions which in addition to performing a basic function such as load or exchange also set up the next address and perhaps test to see if the function is complete so a computational loop may be terminated automatically.

Suppose that a number is in a register, Register 2, which occupies Row 2 Columns 0 through 5. The program to initially set the B Register address to Row 2, Column 5, and then shift the contents of the register left one decimal place with a zero loaded into the vacated position is as follows:

```
ONE     LBL     #25
TWO     LAI     0
LSFT    XDSK    0
FOUR    T       LSFT
```

Instruction ONE sets the B Register to hexadecimal 25 (the # symbol indicates a hexadecimal number rather than a decimal number). Instruction TWO loads zero into the Accumulator. The instruction in location LSFT (XDSK) causes the contents of the Accumulator to be exchanged with the addressed memory cell (initially #25), decrements the B Lower address to point to #24 and leaves the B Upper register alone since 0 Exclusively ORed with anything leaves it unchanged.

The XDSK instruction also tests to see if the complete register has been shifted by checking the decremented B Lower. If B Lower decrements from zero to #F, then the next instruction will be skipped. Since, after completion of the 1st XDSK instruction, B Lower is 4, no skip occurs. Consequently the transfer instruction is not skipped and the process is repeated. The prior contents of #25 which are in the Accumulator are exchanged with #24 and the address is decremented and tested again. Subsequently the contents of #24 are moved to #23, #23 to #22, #22 to #21 and #21 to #20. Note that when this last operation is performed the B Lower register will decrement from 0 to #F causing the transfer instruction to be skipped and the process terminated.

3.3 DATA TRANSFER TECHNIQUES

The five basic data transfer instructions are shown in Table 3-5. These instructions have been designed to provide great flexibility in loading the Accumulator with data from memory or constants and exchanging information between the Accumulator and memory. The L, X, XDSK and XNSK instructions are all multi-function instructions which have the capability of modifying the B Register for the next instruction if it is desired by the programmer. This feature eliminates the need, in many cases, for writing additional instructions to modify the data address in B Register. The LAI instruction is used to load a constant into the Accumulator.

Table 3-5. DATA TRANSFER INSTRUCTIONS

Mnemonics	I/D Bus Op Code Hex & Binary	Name	Description	Symbolic Equation
L	50-53 0101 00--	Load Accumulator from Memory (1 cycle-1byte)	The 4-bit contents of RAM currently addressed by B Register are placed in the Accumulator. The RAM address in the B Register is then modified by the result of an Exclusive-OR of the 2-bit immediate field I(2:1) and B(6:5)	A ← M; B(6:5) ← B(6:5) ⊕ I(2:1)
X	58-5B 0101 10--	Exchange Accumulator and Memory (1 cycle-1 byte)	Same as L except the contents of Accumulator are also placed in currently addressed RAM location.	A ↔ M; B(6:5) ← B(6:5) ⊕ I(2:1)
XDSK	5C-5F 0101 11--	Exchange Accumulator with Memory, decrement BL and Skip if BL underflows. (1 cycle overlapped-1 byte)	Same as X except RAM address in B Register is further modified by decrementing BL by 1. If the new contents of BL is 1111, the next instruction will be ignored.	A ↔ M B(6:5) ← B(6:5) ⊕ I(2:1) BL ← BL-1 after one cycle Skip on BL = 1111
XNSK	54-57 0101 01--	Exchange Accumulator with Memory, Increments BL and Skip if BL overflows (1 cycle overlapped -1 byte)	Same as XDSK except B Lower increments and the skip will occur when BL counts to 0000.	A ↔ M B(6:5) ← B(6:5) ⊕ I(2:1) BL ← BL+1 after one cycle Skip on BL = 0000
LAI	70-7F 0111 ----	Load Accumulator Immediate (1 cycle-1 byte)	The 4-bit contents, immediate field I(4:1), of the instruction are placed in Accumulator. (See Note below)	A ← I(4:1) See Note

NOTE: Only the first occurrence of an LAI instruction in a consecutive string of LAI's will be executed. The program will ignore the remaining LAI's and execute the next valid instruction.

⊕ indicates Exclusive OR function.

3.3.1 ROW ADDRESS MODIFICATION WITH THE L, X, XDSK, AND XNSK INSTRUCTIONS

The address modification capability of the L, X, XDSK, and XNSK instructions allows the programmer to modify the row designation (BU) postion of the B Register. The next instruction which uses data memory then may refer to a different row in the RAM for the next desired data. For this reason, registers within memory which interact and consist of more than one 4-bit word are usually organized so that each complete register is organized along a row using the same columns whenever possible. With this organization of memory, the PPS-4/1 programs for manipulations between registers can be very efficient.

3.3.2 BL ADDRESS MODIFICATION BY THE XDSK AND XNSK INSTRUCTIONS

In addition to modifying the row address, BU, the XDSK (Exchange, Decrement, and Skip on Underflow) instruction has the further capability of modifying the column by automatically decrementing the BL Register each time the XDSK instruction is obeyed. Consequently, in addition to having the capability of moving to any other row or register within RAM, the address can also be made to select the next adjacent lower numbered column for the next data. All of this occurs after exchanging the contents of the Accumulator and memory. When the data is organized so that a register occupies a row or part of a row and each of the next most significant four bits in a register occupy the word in the next lower numbered column, this instruction allows automatic addressing of the next four significant bits in sequence in the designated register after the data exchange.

The XNSK (Exchange, Increment, and Skip on Overflow) instruction operates in a similar manner except that the address selects the next adjacent higher numbered column for the next data address. When using this instruction for numerical processing, the least significant digit occupies the lowest numbered column address for the memory register with subsequent digits in the next higher address column. The Register BL modification effects of these instructions are shown in Figure 3-4.

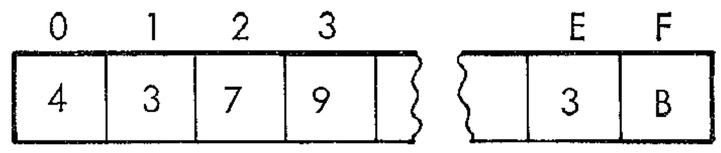
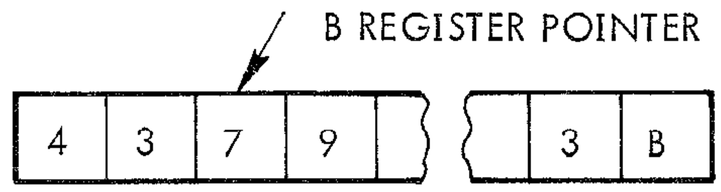
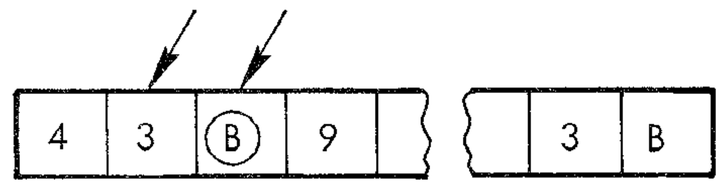
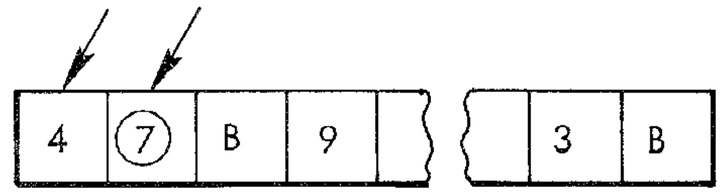
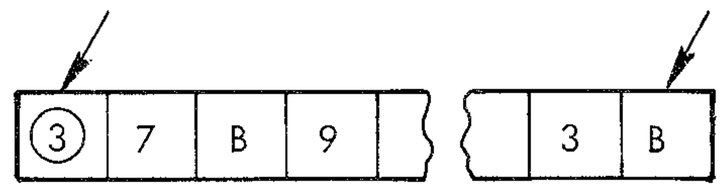
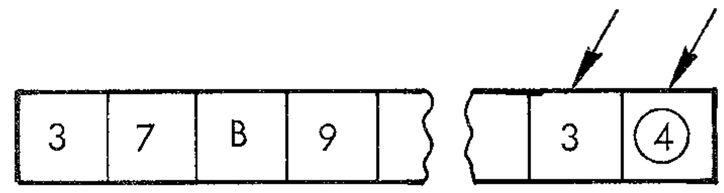
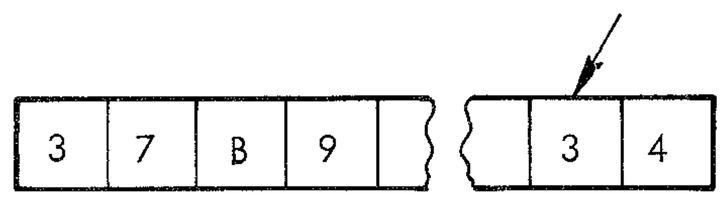
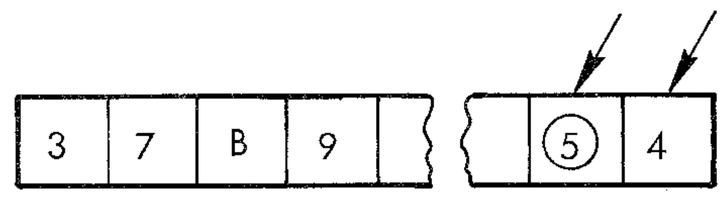
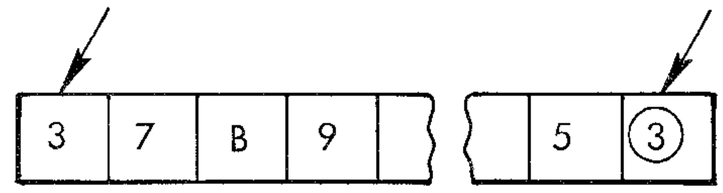
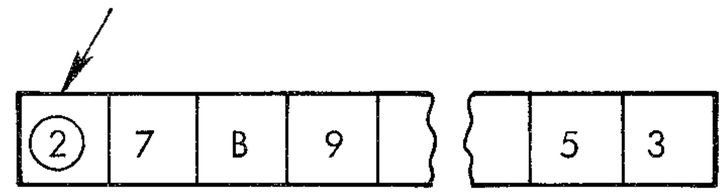
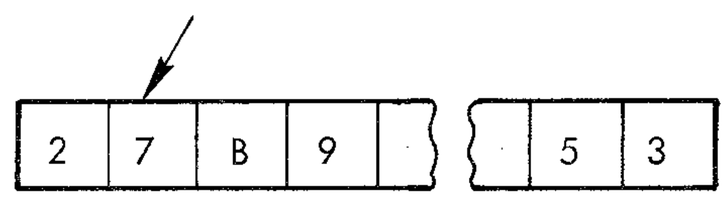
Both of these instructions (and the INCB and DECB instructions also) use the ALU to accomplish the increment or decrement functions. Because of this, there is a delay of one cycle before the incremented or decremented address may actually address memory or a discrete I/O port. If an instruction is executed which references memory or a discrete I/O port immediately after an increment or decrement function, the old value of BL will be used. The new value of BL is available, however, for instructions, such as XAB, which do not reference memory or the discrete I/O ports, but do make use of BL. This overlapping capability may be useful in handling keyboard or display strobe. The following program is an example of this:

```
LBL     #33     Point to discrete I/O port 3
XDSK
ROS             Reset I/O port 3
SOS             Set I/O port 2
```

In almost all (probably all) data moves, the XNSK or XDSK instruction is followed by a T or SKBEI instruction so that when the next L instruction is executed, the address is properly set up. This will be illustrated in the next section.

3.3.3 USE OF IMMEDIATE FIELD TO DESIGNATE BU CHANGES

In many functions which are to be performed by a computer, it is necessary to first address a source of data and then address a memory cell destination for the data after it has been processed. Thus the capability to readily shift addresses from one register to another is important for efficient computations. In the PPS-4/1 this shifting of addresses is accomplished automatically by selectively modifying B Upper. To implement BU Register modification the two least significant bits of an L, X, XDSK or XNSK are used to identify the relationship between the current BU Register value and the desired next value. The two bits which the programmer writes as an operand are

INSTRUCTION	BU	BL	A	ROW 2
				
LBL #22	2	2	B	
XDSK 0	2	1	(7)	
[*]				
XDSK 0	2	0	(3)	
[*]				
XDSK 0	2	F	(4)	
[IGNORE]				
XDSK 0	2	E	(B)	
LAI 5	2	E	(5)	
XNSK 0	2	F	(3)	
[*]				
XNSK 0	2	0	(4)	
[*]				
LAI 2	2	0	(2)	
X	2	0	(3)	
INCB	2	1	3	

[*] MAY BE ANY OPERATION THAT DOES NOT CHANGE B REGISTER OR ACCUMULATOR TO GET RESULTS SHOWN

Figure 3-4. BL MODIFICATIONS AND DATA TRANSFERS WITH XDSK, XNSK, X AND LAI INSTRUCTIONS

designated as the immediate field and become part of the instruction as it is stored in the program memory. The actual modification is accomplished by Exclusively ORing the immediate field (bits 2 and 1 of the instruction) with bits 6 and 5 of the B Register.

In addition to using the automatic address modifications available as part of the L, X, XDSK, and XNSK, there is another possibility available for manipulating the BU Register: the EOB (Exclusive OR BU Register) which uses an additional instruction, but gives complete flexibility in modifying the BU Register.

Examples of some of these address modification techniques are shown in Figure 3-4.

The Exclusive OR function was chosen by the designers of the PPS-4/1 because it is the simplest logic mechanization to provide complete flexibility of interchanging addresses within each of the groups of rows. Furthermore, the same immediate field value used a second time with the load or exchange types of instructions will cause the original row address to be restored.

Figure 3-5 provides a convenient reference for specifying the argument portion of the instruction. From this argument the assembly routine will generate the required immediate field bit pattern which will accomplish the desired address modification. For example, if the BU Register is 1 (bit pattern 01) and it is desired to convert the address of the BU Register to 0 (bit pattern 00), the programmer can determine from Figure 3-5 that 1 (bit pattern 01) is the proper argument. Similarly, if BU Register was 1 (bit pattern 01), and it was desired to convert it to 3 (bit pattern 011), it is necessary to write 2 as the argument. Note that with the Exclusive OR argument, to convert from 0 back to 1, or 3 back to 1, the same argument is used again. Similarly, if it is desired to change BU Register from 3 to 2, then Figure 3-5 shows that 1 is the proper argument. If BU Register is to be left unchanged for the next data access, 0 is used as the argument. The assembly routine interprets a blank argument as a zero so BU Register would remain unchanged if no argument is used.

FINAL B UPPER

ORIGINAL B UPPER	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

- TOTAL AREA MAY BE USED FOR EOB, L, X, XDSK, XNSK, INCB, AND DECB INSTRUCTIONS.
- IF NO ARGUMENT IS DESIGNATED, IT WILL BE ASSUMED ZERO.

Figure 3-5. VALUES OF ARGUMENT FOR L, X, XDSK, XNSK, INCB, AND DECB INSTRUCTIONS

3.3.4 USE OF XDSK AND XNSK TO AUTOMATICALLY TERMINATE DATA TRANSFER SEQUENCE

The XDSK and XNSK instructions each have one additional feature which even further simplifies programming of the PPS-4/1 and reduces the amount of program memory which is required. This capability provides an automatic determination that all of the data from the addressed row has been exchanged with the Accumulator and provides an automatic capability for terminating the sequence of operations when this has occurred.

Using XDSK to terminate operation is accomplished in the following manner. As the BL Register is decremented successively, it will eventually reach a value of zero, which will cause the most significant 4-bits in the row of data to be exchanged with the Accumulator and then BL Register is decremented. At that time, the value of BL Register will change to 1111 condition which the CPU logic recognizes and causes the next instruction to be ignored. Usually the next instruction after an XDSK is a transfer instruction which causes the data transfer process to be repeated until the last segment of the data is exchanged. Then the transfer instruction is effectively skipped to go on the next phase of the processing.

The XNSK instruction functions similarly except that BL Register increments until after exchanging the contents of the Accumulator with the memory in column F (1111), the BL Register will increment to 0000 and the next instruction will be skipped. Note that there are a few instructions which should not be skipped. These are LBL, INCB, DECB and all of the conditional transfer instructions. All other one, two or three byte instructions may be skipped.

3.3.5 USAGE OF THE LOAD ACCUMULATOR IMMEDIATE (LAI)

The Load Immediate Instruction (LAI) is used to load the 4-bit number identified in the immediate field into the Accumulator directly. This instruction is normally used to load the value of a constant into the Accumulator. The constant can be zero so that this instruction is also used to clear the Accumulator.

There is, however, another basic use of the Load Immediate instruction, and that is to load one and only one of a string of constants into the Accumulator, depending upon where the string of LAI instructions is entered. This is accomplished because the PPS-4/1 recognizes only the first of a string of LAI instructions are valid, and ignores the following LAI instructions.

3.3.6 DATA TRANSFER EXAMPLES

The first set of examples consists of a series of short programs which set various areas in RAM to zero. The programs are written as follows:

```
* INSERT ZERO'S(ROW 0,COL 0 TO 5)
*LABEL  OPCODE OPERAND COMMENT
EX1     LB     5      POINT TO ROW 0 COL 5
A1      LAI    0      LOAD ACC WITH 0
        XDSK          !EXCH ACC WITH MEMORY,DECR B AND TEST
        T      A1     IF NOT DONE DO NEXT DIGIT
*
* INSERT SIX'S(ROW 1,COL 0 TO F)
EX2     LBL    #1F    POINT TO ROW 1 COL F
A2      LAI    6      LOAD ACC WITH 6
        XDSK          !EXCH ACC WITH MEMORY,DECR B AND TEST
        T      A2     IF NOT DONE DO NEXT DIGIT
*
* INSERT ZERO'S(ROW 1 AND 2,COL A TO F,CHECKERBOARD FASHION)
EX3     LBL    #1A    POINT TO ROW 1 COL A
A3      LAI    0      LOAD ACC WITH 0
        XNSK   3      EXCH,MODIFY ROW,INCR B AND TEST
        T      A3     IF NOT DONE DO NEXT DIGIT
*
* INSERT ZERO'S(ROW 0 AND 2,COL 0 TO 7)
EX4     LB     7      POINT TO ROW 0 COL 7
A4      LAI    0      LOAD ACC WITH 0
        X      2      EXCH AND MODIFY ROW TO 2
        LAI    0      LOAD ACC WITH 0
        XDSK   2      EXCH,MODIFY ROW,DECR B AND TEST
        T      A4     IF NOT DONE DO NEXT DIGIT
```

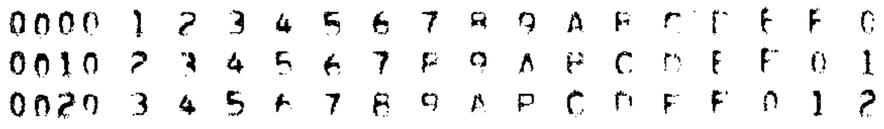
The symbol # identifies the argument as an absolute hexadecimal address and not an alphanumeric symbol address. The A1 through A4 symbols in the label and argument fields are interpreted as labels since they are not preceded by the # symbol. The EX1 through EX4 symbols are also interpreted as labels.

Both the initial contents of RAM and the contents after executing the above program are shown in Figure 3-6.

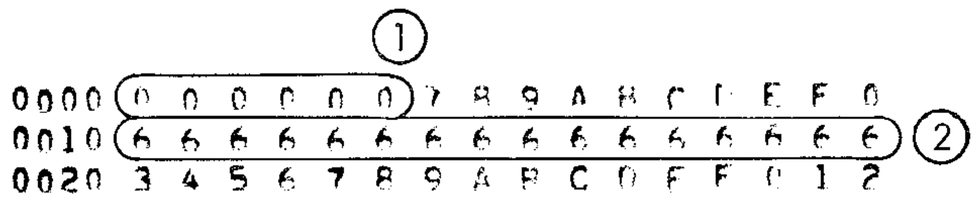
The first example loads the B Register with #05 (in hex), sets the Accumulator to zero, and exchanges the Accumulator with the addressed memory cell.

Since the value of BL Register is then set to 4, the program does not skip an instruction and a transfer to load the Accumulator immediate with zero instruction causes the process to repeat. This process is automatically terminated after the last zero is transferred. The result of this sequence is identified by (1) in Figure 3-6.

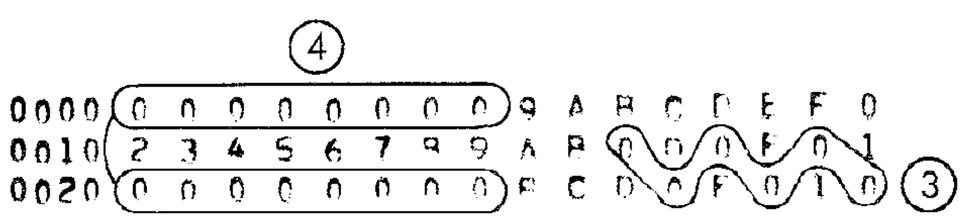
The second example is identical except that the initial address is set to #3F (in hex) and the Accumulator is loaded with 6 so that a complete row is set to 6's. The result of this sequence is shown as (2) in Figure 3-6.



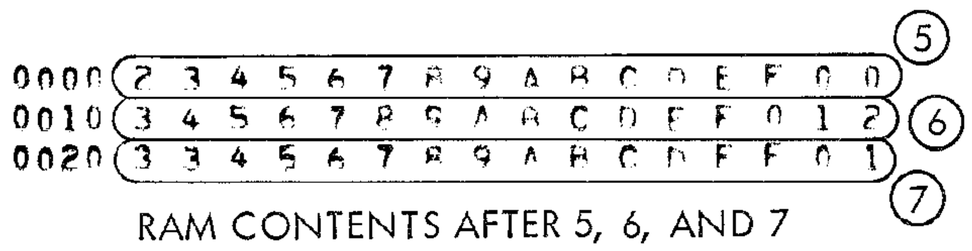
INITIAL CONTENTS OF RAM



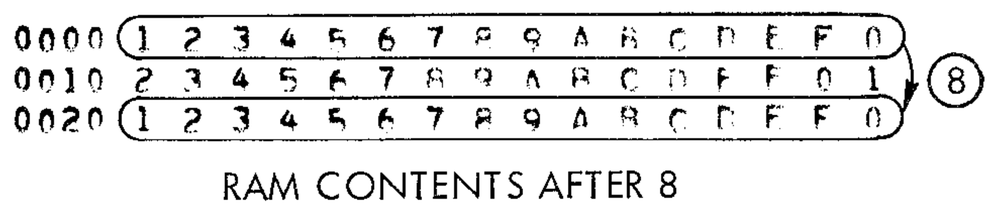
RAM CONTENTS AFTER 1 AND 2



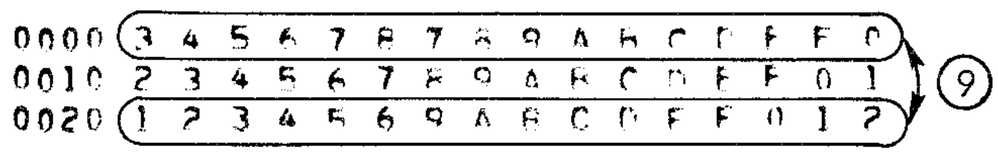
RAM CONTENTS AFTER 3 AND 4



RAM CONTENTS AFTER 5, 6, AND 7



RAM CONTENTS AFTER 8



RAM CONTENTS AFTER 9

Figure 3-6. PRINTOUTS BEFORE AND AFTER EXECUTING EXAMPLE PROGRAMS

The third example is identical to the first except that the starting address is #1A (hex) and the capability of modifying the BU Register is demonstrated using the increment form of the Exchange and Skip instruction. When the immediate field value of 3 (11) is Exclusively ORed with the corresponding bits in BU, which in this example are equal to 1, (01) the result is 2 (10) for the next digit in sequence. (Note that the table in Figure 3-5 can be used to obtain this result.) On the next digit, BU is restored to 1 so the zig-zag pattern of zeros results. Since the XNSK BU instruction is used, the zero pattern moves from the initial setting to the right.

In example 4 the initial zero is loaded into #07 and then the BU value is changed to 2 by the 2 in the X 2 instruction. Zero is again loaded into the Accumulator because the Accumulator has the value #D in it (the original contents of #07). The zero is then transferred to #27 and the address modified to #06 by the XDSK 2 instruction. Again the process is repeated until the XDSK causes a skip out of the transfer sequence.

The following programs show several useful examples of data transfer programs.

```
* LEFT DIGIT SHIFT(ROW0)...(MULTIPLY BY 16)
EX5     LAI     0       LOAD ACC WITH 0
        LB      #F      POINT TO ROW 0 COL F
A5      XDSK            !EXCH ACC WITH MEMORY,DECR B AND TEST
        T       A5      IF NOT DONE DO NEXT DIGIT
*
* END AROUND LEFT SHIFT(ROW 1)
EX6     LBL     #10     POINT TO ROW 1 COL 0
        XDSK            !EXCH ACC WITH MEMORY,DECR B AND TEST(SKIP)
A6      XDSK            !EXCH ACC WITH MEMORY,DECR B AND TEST
        T       A6      IF NOT DONE DO NEXT DIGIT
*
* RIGHT DIGIT SHIFT(ROW 2)
EX7     LBL     #20     POINT TO ROW 2 COL 0
        L               !LOAD ACC FROM MEMORY
A7      XNSK            !EXCH ACC WITH MEMORY,INCR B AND TEST
        T       A7      IF NOT DONE DO NEXT DIGIT
*
* MOVE ROUTINE(ROW 0 TO ROW 2)
EX8     LB      #F      POINT TO ROW 0 COL F
A8      L       2       LOAD ACC FROM MEMORY AND MODIFY ROW
        XDSK    2       EXCH,MODIFY ROW,DECR B AND TEST
        T       A8      IF NOT DONE DO NEXT DIGIT
*
* EXCHANGE ROUTINE(ROW 0 WITH ROW 2)
EX9     LB      5       POINT TO ROW 0 COL 5
A9      L       2       LOAD ACC FROM MEMORY AND MODIFY ROW
        X       2       EXCH ACC WITH MEMORY AND MODIFY ROW
        XDSK            !EXCH,DECR B AND TEST
        T       A9      IF NOT DONE DO NEXT DIGIT
```

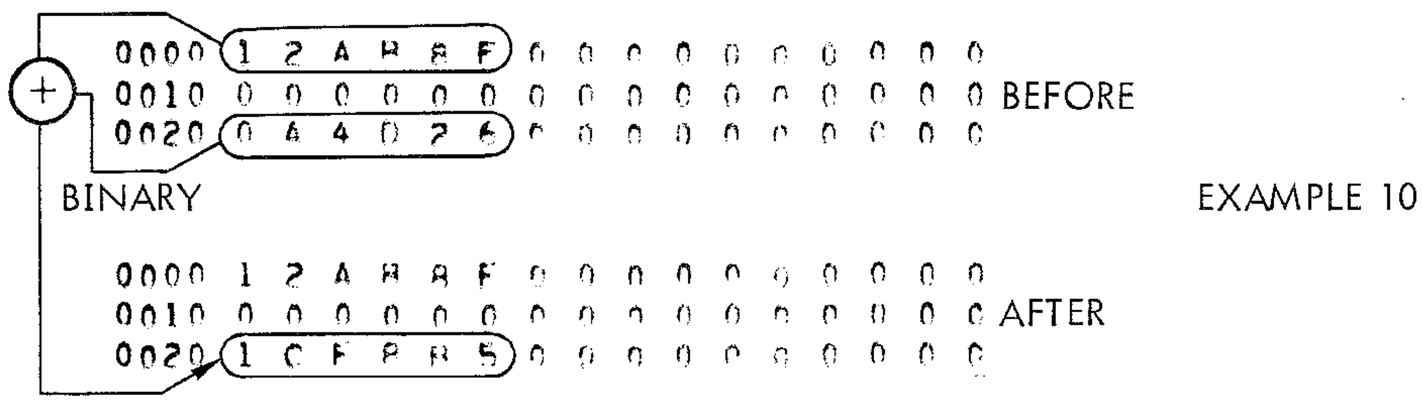
The results of these examples are shown in Figure 3-6. Example 5 causes a 16-word register to be shifted left by one word with the least significant word loaded with zero from the Accumulator. The value 2 is left in the Accumulator at the end.

Example 6 causes an end-around or circular left shift of a 16-word register.

Example 7 causes a 16-word register to be shifted right by one word with the most significant digit (#20) left unchanged.

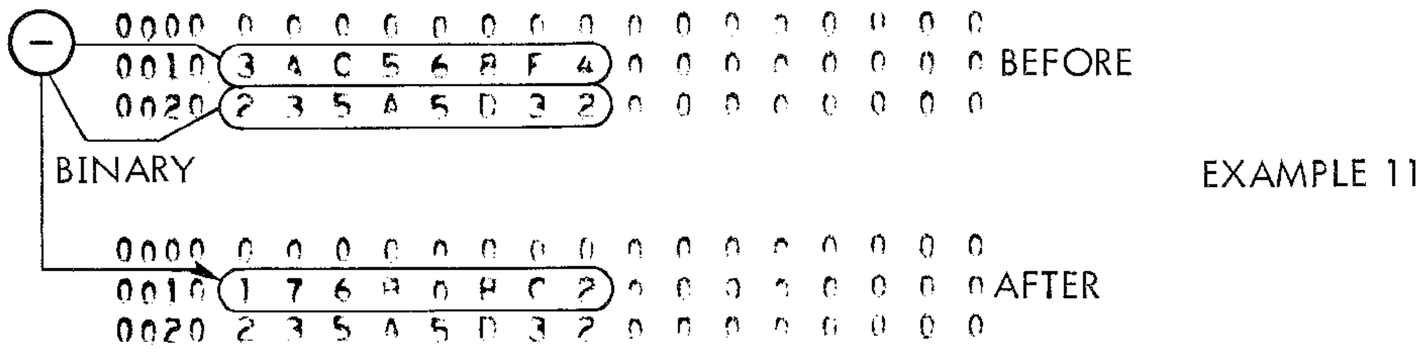
Example 8 merely replaces the contents of Row 2 with the contents of Row 0.

Example 9 causes the contents of Rows 0 and 2 to be exchanged.



EXAMPLE 10

RAM CONTENTS AFTER EXAMPLE 10



EXAMPLE 11

RAM CONTENTS AFTER EXAMPLE 11

Figure 3-7. RAM PRINTOUT AFTER EXECUTING EXAMPLE PROGRAM

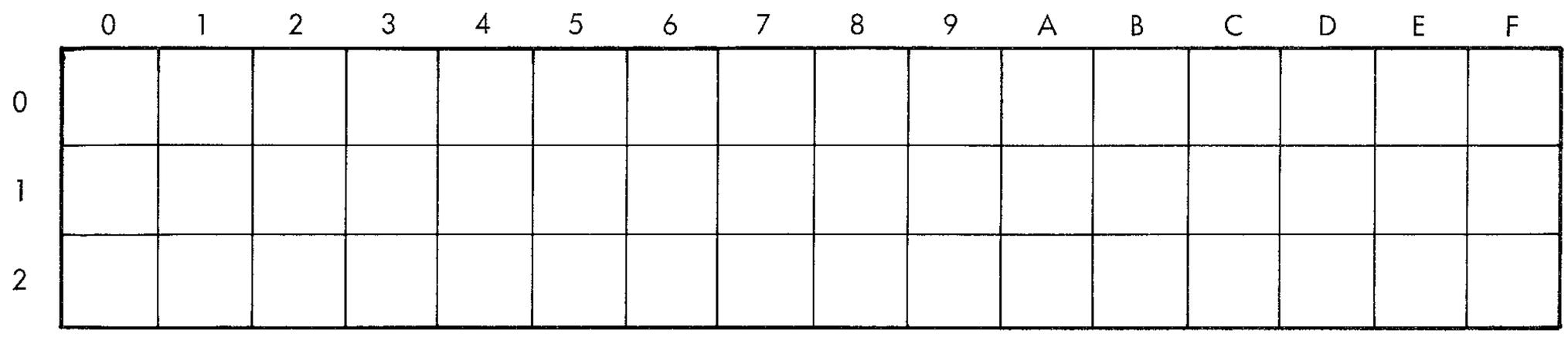


Figure 3-8. ADDRESS ALLOCATIONS FOR THE 48 RAM MEMORY CELLS

3.3.7 INDEXING

The PPS-4/1, like most other microcomputers, does not have true index registers in the classical sense of adding the contents of a separate index register to a specified address and manipulating that index register to change addresses and control terminating sequences. However, the PPS-4/1 provides similar capability in its flexible addressing, its automatic address modification, and its automatic sequence termination techniques. By carefully organizing the way the data is stored in the data memory so that advantage may be taken of the automatic addressing shifting by switching BU; and by organizing registers in memory so that their relative significant digit positions are aligned, very efficient programs can be written for the PPS-4/1.

3.3.8 DATA ADDRESS MODIFICATION INSTRUCTIONS SUMMARY

The data address modification capability of the PPS-4/1 is extremely flexible. Table 3-6 summarizes all of the instructions which directly modify or aid in modifying addresses. It shows all of the principal microcomputer registers involved and identifies each of the instructions and conditions which influence modifying data addresses.

Table 3-6. DATA ADDRESS MODIFICATION SUMMARY

Instruction		Address Bits Modified		Condition to Automatically Terminate Process
Op Code	Immediate Field	2 LSBs of BU	BL	
XAB*			Exchanged with A	
LBA		—	Replaced	—
LBL	Upper/Lower	—	Replaced (first)	—
LB	Lower	—	Replaced	—
EOB	Upper	—	No Change	—
INCB*	2 bit Upper	Exclusive OR	Increment and Test for 0	BL counts to 0
DECB*	2 bit Upper	Exclusive OR	Decrement and Test for #F	BL counts to #F
L	2 bit Upper	Exclusive OR	No Change	—
X	2 bit Upper	Exclusive OR	No Change	—
XDSK*	2 bit Upper	Exclusive OR	Decrement and Test for #F	BL counts to #F
XNSK*	2 bit Upper	Exclusive OR	Increment and Test for 0	BL counts to 0

*The B Lower portion of the address is modified during the next cycle following instruction so complete new address may be used one cycle later. Instructions immediately following the address modification instruction which address memory will have any B Upper address modifications completed, but the old value of BL will be used.

3.4 ARITHMETIC OPERATIONS

The basic arithmetic instructions in the PPS-4/1 system are summarized in Table 3-7. It will be noted that there is perhaps, a surprising number of different instructions for adding 4 bits in parallel to the Accumulator. As was the case with the XDSK type instruction, many of the arithmetic instructions perform additional functions beyond just performing the basic function. The purpose of each of these additional capabilities is discussed below.

3.4.1 ADD (A) AND ADD WITH CARRY IN (AC) — BINARY ADDITION

The Add (A) instruction and the Add with Carry (AC) instructions are the most conventional of the arithmetic instructions contained in the PPS-4/1 set. The Add instruction is used for modulo 16 addition, (addition that does not extend beyond 4 bits) or for shifting left without affecting the Carry flip-flop. For this instruction, any carry which may have existed from prior operations is ignored and just the contents of the Accumulator and the addressed memory cell are added to produce a new 4-bit sum with the carry unchanged.

The Add With Carry instruction is used to add successive groups of 4 bits using the carry from the previous 4 bits as an input to the add operations. Consequently, the prior carry, the contents of the addressed memory cell, and the contents of the Accumulator are added together to form a new 4-bit sum and new carry bit.

The AC instruction can be used for a complete binary add. For the least significant 4 bits, the Carry flip-flop must be set to zero prior to the first addition in the sequence. The instruction, Reset Carry (RC), which will be discussed later, is used to set the Carry flip-flop to zero.

3.4.2 BINARY SUBTRACTION

In order to perform a binary subtraction, the Complement instruction is used to form the complement of the number to be subtracted. Then this complement is added to the minuend and the process is repeated 4 bits at a time until the final difference is obtained. In this case, it is necessary to use the AC instruction for all of the digit additions and to initially set the Carry flip-flop to 1 so that, in effect, the 2's complement is used. This is done by executing a Set Carry (SC) instruction prior to starting the subtraction sequence.

3.4.3 ADD IMMEDIATE AND SKIP (AISK)

The Add Immediate and Skip instruction (AISK) may be used for adding a fixed constant to the Accumulator. This is used in decimal arithmetic since selectively adding a constant 10 ($^{\#}$A) is part of the process of converting what is basically binary arithmetic to decimal arithmetic. The Add Immediate and Skip instruction can also be used to set up tests on the magnitude of the Accumulator. For instance, if AISK 7 is executed and no overflow is produced, the original number in the Accumulator must have been 8 or less. The microcomputer will then skip (ignore) the next instruction. If the value in the Accumulator prior to the AISK 7 instruction was 9 or more the next instruction in sequence would be executed instead of being ignored.

A different characteristic of the AISK instruction is that if an overflow is generated, it does not disturb the Carry flip-flop. The contents of the Carry flip-flop are unchanged.

Table 3-7. ARITHMETIC INSTRUCTIONS

Mnemonics	I/D Bus Op Code Hex & Binary	Name	Description	Symbolic Equation
A	42 0100 0010	Add (1 cycle-1 byte)	The result of binary addition of contents of Accumulator and 4-bit contents of the RAM currently addressed by B Register, replaces the contents of Accumulator. Carry is not set or used.	A ← A+M C ← C
AC	40 0100 0000	Add with Carry-In (1 cycle overlapped -1 byte)	Same as A except the C flip-flop serves as a carry-in to the adder and changing carry is delayed one cycle. The carry is set or reset as a result of the arithmetic with one cycle delay.	C, A ← A+M+C
ACSK	41 0100 0001	Add with Carry-In and Skip on No Carry (1 cycle overlapped-1 byte)	Same as AC except if the carry is not set as a result, the next instruction is skipped (ignored). New carry is available after one cycle delay.	C, A ← A+M+C Skip if C = 0
AISK	60-6F 0110 ----*	Add Immediate and Skip on No Carry-Out (1 cycle-1 byte)	The result of binary addition of contents of Accumulator and 4-bit immediate field of instruction word replaces the contents of Accumulator. The next instruction will be skipped (ignored) if no overflow occurs. This instruction does not use or change the C flip-flop.	A ← A+I(4:1) C ← C Skip if No overflow
ASK	43 0100 0011	Add and Skip on No Overflow	Same as A except the next instruction is skipped if no overflow occurs.	A ← A+M C ← C Skip if No overflow
COM	45 0100 0101	Complement (1 cycle-1 byte)	Each bit in the Accumulator is complemented (inverted).	A ← [$\overline{A}$] C ← C

*---- Indicates restrictions on bit patterns allowable in immediate field as specified in the symbolic equation description.

The AISK instruction is useful for testing the most significant bit in the Accumulator for a 0. A sequence of instructions: AISK 8, AISK 8 and T RSET will cause the program to transfer to RSET if the most significant bit is a zero, but will reset the Accumulator to its original value without disturbing the Carry flip-flop and continue with the next in line instruction following the location of the transfer instruction if the bit is a one.

3.4.4 ADD AND SKIP ON NO OVERFLOW (ASK)

The Add and Skip on No Overflow instruction is like the Add instruction in that it adds two 4-bit numbers together to obtain another 4-bit number and does not use or affect the Carry flip-flop. This instruction is also useful in performing a test since it ignores the next instruction if no overflow is produced by the addition. It is especially convenient when it is desired to perform a test without changing the state of the carry.

3.4.5 ADD WITH CARRY IN AND SKIP ON NO CARRY (ACSK) — DECIMAL ADDITION

The Add with Carry In and Skip On No Carry instruction is the primary arithmetic instruction for decimal arithmetic. The resulting sum from adding two 4-bit numbers using this instruction is a binary 4-bit number with or without carry depending upon the digits added. If no carry is produced, this instruction causes the next instruction to be skipped (ignored). However, this instruction used together with an Add Immediate and Skip #6 and an Add Immediate and Skip 10 instruction produces, a very efficient, decimal addition technique. This is shown in the following examples.

Example: 4 + 5 = 9

```
                  0  = Carry in
4          0100      = 4
            110      = Corrective 6 added
           1010        (Skip)
+5          101      = +5
           1111      = 15 (No Carry = Skip)
           1010      = +10 (Never Skips - no Carry is set)
9    1     1001      = Binary 25 but with Carry bit (16) ignored
     0     1001      = 9 with no Carry = correct decimal answer
```

No Carry Out is actually generated

```
RC ———— BU Modifier
L ( )
AISK 6
NOP
ACSK
T          *+2
AISK 10
```

```
          ———— BU Modifier
XDSK ( )
T          *-5
:
```

Example: 7 + 8 = 15

```
                  0  = Carry In
7          0111      = 7
           0110      + Corrective 6 (Skip)
           1101
+8         1000        +8
+15   1    0101      = 5 with a Carry = correct decimal answer
```

```
RC ———— BU Modifier
L ( )
AISK 6
NOP
ACSK
T          *+2
AISK 10 (skipped)
XDSK ( )
T          *-5
:
```

In the latter case, a carry is produced which causes the AISK 10 instruction to be skipped so the proper value is stored and the process repeated for the full decimal number. Note that independent of whether or not a carry was generated and the AISK 10 instruction skipped or not, the time to execute is constant.

3.4.6 MAGNITUDE TESTING VIA ACSK AND ASK

The Add with Carry-In instruction (ACSK) may also be used for magnitude testing. To determine if the number in the Accumulator is equal to or greater than some test value, V, all that is necessary is to reset the carry and add an appropriate value from memory. This value from memory must be the two's complement of the test value. This will cause an overflow to occur which sets the Carry flip-flop and causes the next instruction to be performed if the test condition is met.

```
        RC
        ACSK
        T           NOC     A ≥ V
        [ ]                 A < V
        :
```

If the magnitude in the Accumulator was equal to or greater than the value being compared, the program would execute the next instruction. This next instruction would normally be the transfer to the point which continues the path if the magnitude was greater than or equal to the tested value. If a carry was not produced, the program would skip the transfer instruction to continue the program.

The same test can be performed without affecting the state of the Carry flip-flop using ASK instead of ACSK. If the test value is a constant, the AISK instruction is the preferred magnitude test. If the test value is variable, then the form of magnitude testing shown here may be used. Another form of magnitude testing using the SKAEI instruction is discussed later.

3.4.7 DECIMAL SUBTRACTION

Decimal subtraction is similar to binary subtraction in that the Carry flip-flop is initially set to 1, the subtrahend is loaded into the Accumulator and complemented and the minuend is then added with an ACSK instruction to obtain a trial result. If a carry is produced, the resulting number is the correct difference, and if no carry is produced, then it is necessary to perform a correction. Some examples of decimal subtraction are shown below:

Example:

```
          7 - 9     = 8 with a borrow
            1       = Carry In
         1001       = 9
         0110       = Complement of 9 in binary
            1       = Carry In
         0111       = + 7
    0    1110       = 14 with No Carry
         1010       = +10 (never Skips - Carry not set)
    1    1000       = 24 but with Carry bit ignored = 8 and
                      the Carry is not set so a borrow is indicated
    0    1000       = correct decimal answer
```

No Carry Out is actually generated

```
        SC          <-- BU Modifier
        L   ( )
        COMP
        ACSK
        T           *+2
        AISK 10
        XDSK ( )
        T           *-5
        :
```

Example:

```
      8 - 3     = 5
        1       = Carry In
     0011       = 3
     1100       = Complement of 3 in binary
        1       = Carry in
     1000         +8
1    0101       = 5 with Carry = correct decimal answer
```

```
SC  ┌──── BU Modifier
L   (↓)
COMP

ACSK
T       *+2
AISK 10 (skipped)
XDSK  ( )
T       *-5
```

3.4.8 MULTIPLY AND DIVIDE

Both binary multiplication and division and decimal multiplication and division make use of repeated additions and subtractions of the same forms as shown above. The digits must be shifted and added or subtracted an appropriate number of times depending upon the specific numbers involved. This is an efficient process on the PPS-4/1 and will be described in a later section after a larger repertoire of PPS-4/1 instructions have been discussed.

3.4.9 BINARY AND DECIMAL ARITHMETIC PROGRAMS

A useful series of arithmetic examples is indicated in the series of short programs below.

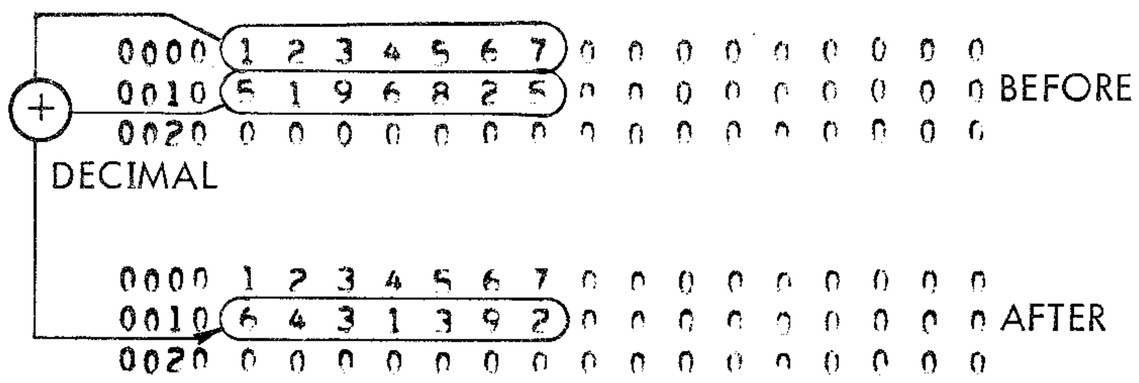
```
* BINARY ADD (REG #00-05 + REG #20-25 --> REG #20-25)
EX10    RC              RESET CARRY
        LB      5       POINT TO ROW 0 COL 5
A10     L       2       LOAD ACC FROM MEMORY AND MODIFY ROW
        AC              ADD MEMORY TO ACC
        XDSK    2       EXCH,MODIFY ROW,DECR B AND TEST
        T       A10     IF NOT DONE,DO NEXT DIGIT
*
* BINARY SUBTRACT (REG #10-17 - REG #20-27 --> REG #10-17)
EX11    SC              SET CARRY
        LBL     #27     POINT TO ROW 2 COL 7
A11     L       3       LOAD ACC FROM MEMORY AND MODIFY ROW
        COM             COMPLEMENT ACC
        AC              ADD MEMORY TO ACC
        XDSK    3       EXCH,MODIFY ROW,DECR B AND TEST
        T       A11     IF NOT DONE,DO NEXT DIGIT
*
* DECIMAL ADD (REG #00-06 + REG #10-16 --> REG #10-16)
EX12    RC              RESET CARRY
        LB      6       POINT TO ROW 0 COL 6
A12     L       1       LOAD ACC FROM MEMORY AND MODIFY ROW
        AISK    6       DECIMAL CORRECT(ADD 6)
        NOP             THIS INSTRUCTION WILL ALWAYS BE SKIPPED
        ACSK            ADD WITH CARRY AND SKIP IF NO CARRY OUT
        T       *+2     IF  CARRY, DECIMAL CORRECT NEEDED
        AISK    10      ADD 10 TO ACC(SUBTRACT 6)
        XDSK    1       EXCH,MODIFY ROW,DECR B AND TEST
        T       A12     IF NOT DONE,DO NEXT DIGIT
*
* DECIMAL SUBTRACT (REG #00-07 - REG #20-27 --> REG #00-07)
EX13    SC              SET CARRY
        LBL     #27     POINT TO ROW 2 COL 7
A13     L       2       LOAD ACC FROM MEMORY AND MODIFY ROW
        COM             COMPLEMENT ACC
        ACSK            ADD WITH CARRY AND SKIP IF NO CARRY OUT
        T       *+2     IF CARRY, DECIMAL CORRECT NEEDED
        AISK    10      ADD 10 TO ACC(SUB 6)
        XNSK    2       EXCH,MODIFY ROW,DECR B AND TEST
        T       A13     IF NOT DONE,DO NEXT DIGIT
*
* BINARY ADD (REG #09-0F + REG #19-1F --> REG #29-2F)
EX14    RC              RESET CARRY
        LB      9       POINT TO ROW 0 COL 9
A14     L       1       LOAD ACC FROM MEMORY AND MODIFY ROW
        AC              ADD WITH CARRY
        X       0       EXCHANGE
        X       3       EXCH AND MODIFY ROW TO 2
        XNSK    2       EXCH,MODIFY ROW TO 0,INCR B AND TEST
        T       A14     IF NOT DONE,DO NEXT DIGIT
```

The results of the above programs are shown in Figure 3-9 with the contents of RAM shown before and after the sequence of instructions is executed.

In example 10, two 24-bit binary words are added together and the contents replace the contents of one of the original 24-bit registers.

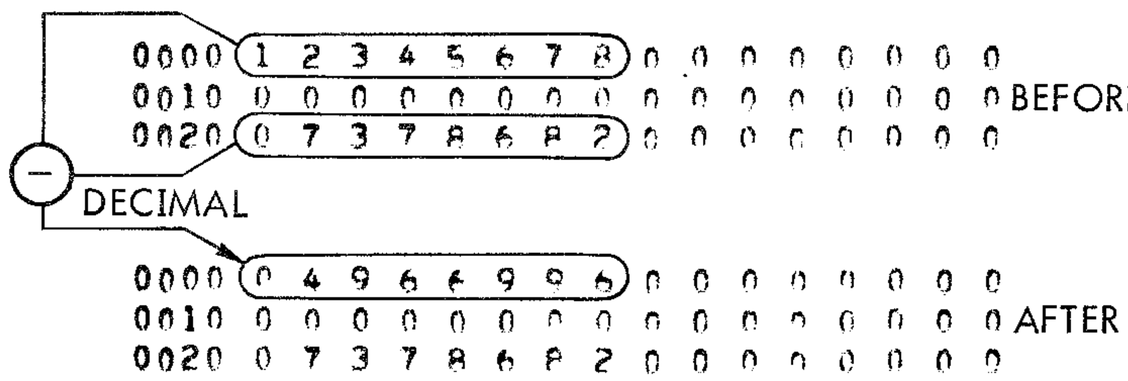
Similarly, example 11 works with two 32-bit registers and subtracts the register in row 2 from row 1, and stores the result back into row 1.

Example 12 works with 7 decimal digits to add two decimal numbers and example 13 works with 8 decimal digits to subtract two decimal numbers. In both of these cases the results of the computation replace the contents of one of the original registers.



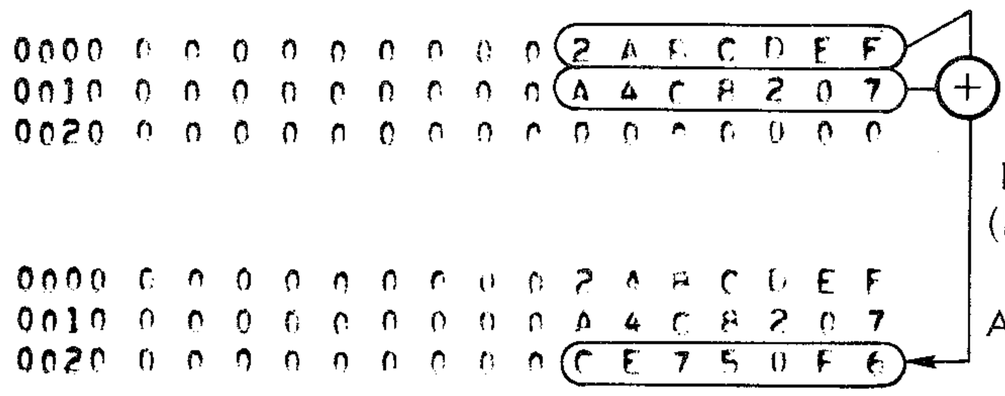
EXAMPLE 12

RAM CONTENTS AFTER EXAMPLE 12



EXAMPLE 13

RAM CONTENTS AFTER EXAMPLE 13



EXAMPLE 14

RAM CONTENTS AFTER EXAMPLE 14

Figure 3-9. RAM PRINTOUTS BEFORE AND AFTER EXECUTING EXAMPLE PROGRAMS

Example 14 is similar to example 13 except that the register length is 28 bits and that the results of the computation are placed into a third memory register. This is accomplished in the following manner.

Initially, the B Register is loaded with the location #09, the contents of the Accumulator are loaded with the memory cell contents in that location, and the BU Register value is converted to 1 by the argument (1) of a load instruction, so that the B Register now points to #19. The contents of that memory cell are added to the Accumulator with the AC instruction. The next three steps accomplish the storage operation in a third register. The first exchange instruction interchanges the original value in register #19 with the Accumulator. The second exchange instruction exchanges them back so that the original value of #19 is back in #19. The resulting sum is back in the Accumulator except that on completion, the second exchange instruction modifies the BU Register with 3, so that the address in the B Register now points to the memory cell #26.

Finally, the exchange and increment instruction stores the four bit result in the designated third memory register. This instruction also uses the argument 2 to translate the 2 in the B Upper address back to a 0 and increment BL so that the process is ready to be repeated beginning with the word located in memory cell #0A.

The argument table for the Load, Exchange, Exchange and Increment and Skip, and Exchange and Decrement and Skip instructions as shown in Figure 3-5 is very useful for tracing B Register modification process in these examples. For instance, the original 0 in BU Register locates a column which can be scanned until the argument 1 is located and then the intersecting row shows a resulting value of #1 for the BU Register address. Similarly, starting with BU equals #1, again at the left of the figure, the argument of 3 shows the translation to a BU Register value of 2 and finally, scanning the 2 row for 2, it confirms that the address is restored to the initial B Register value of zero.

3.5 BIT OPERATIONS

The PPS-4/1 microcomputer family provides instructions for using any bit in memory as a status flag or for any other single bit storage purpose. A group of 12 instructions allows any bit in the addressed memory cell to be set, reset, or tested. The Carry flip-flop may also be used in this same way as can the discrete I/O flip-flops when they are not required for other functions.

The primary bit operation instructions are summarized in Table 3-8.

3.5.1 SET BIT (SB)

The Set Bit instruction sets the bit in the addressed memory as selected by the 2-bit immediate field of the instruction. The 00 code selects bit 1, 01 selects bit 2, 10 selects bit 3, and 11 selects bit 4. With instructions formed by any of the assemblers, the code will automatically be selected. Thus

```
LB   9
SB   3
```

causes the word in memory cell #09 to have bit 3 set to 1 and the other bits are left unchanged.

3.5.2 RESET BIT (RB)

The Reset Bit instruction operates in a similar fashion except that the selected bit in the addressed memory cell is set to zero. Thus

```
LBL      FLAG
RB       2
```

resets bit 2 in the memory cell designated FLAG.

3.5.3 TEST BIT (SKBF)

The Skip on Bit False (SKBF) instruction addresses a bit in memory in a similar manner and skips (ignores) the next instruction if the selected bit was zero. Thus a routine such as

```
          LBL     MSDG     POINT TO DATA WORD MOST SIGNIFICANT DIGIT
          SKBF    4        TEST BIT 4
          T       POS      1 = POSITIVE
NEG       [ ]              0 = NEGATIVE
```

can be used to test the sign of a register when the most significant bit of the most significant digit represents the sign.

Other uses for the Set Bit, Reset Bit, and Skip if Bit False instructions are for performing the logical operations of AND, OR, and Exclusive OR.

3.5.4 LOGICAL AND

The Logical AND operation is usually used to cause the contents of an addressed memory cell to be modified depending upon the contents of a mask. Whenever there is a 1 in corresponding bit positions of both 4-bit words that bit position in memory will be retained as a 1. Any other combination of bits in the mask and memory for that particular bit position will cause a zero to be placed in that corresponding bit position in memory.

Example:

1100	Memory
0110	Mask
0100	Logical AND in Memory

Table 3-8. BIT OPERATION INSTRUCTIONS

Mnemonics	Op Code (Hexadecimal & Binary)	Name	Description	Symbolic Equation
SB*	10-13 0001 00__	Set Bit (1 cycle-1byte)	Set bit in the memory cell addressed by the B Register (M_B) and selected by the two-bit immediate field to 1. I(2:1) = 00 selects bit 1 01 selects bit 2 10 selects bit 3 11 selects bit 4	$M_B(I) \leftarrow 1$ I = bit 1, 2, 3 or 4 See table at left
RB*	14-17 0001 01__	Reset bit (1 cycle-1 byte)	Reset bit in the memory cell addressed by the B Register (M_B) and selected by the two-bit immediate field to 0. See bit selection table in SB instruction description.	$M_B(I) \leftarrow 0$ I = bit 1, 2, 3 or 4 See table above
SKBF*	08-0B 0000 10__	Skip on Bit False (1 cycle-1 byte)	Test the bit in the memory cell addressed by the B Register (M_B) and selected by the two-bit immediate field; if it is false (zero) skip (ignore) the next instruction. See bit selection table above.	Ignore next instruction of $M_B(I) = 0$ I = bit 1, 2, 3 or 4 See table above
SC	0C 0000 1100	Set Carry (1 cycle-1 byte)	Set the Carry flip-flop to the one state.	$C \leftarrow 1$
RC	0D 0000 1101	Reset Carry (1 cycle-1 byte)	Reset the Carry flip-flop to the zero state.	$C \leftarrow 0$
SKNC	01 0000 0001	Skip on No Carry (1 cycle-1 byte)	If the Carry flip-flop is in zero state, skip (ignore) the next instruction. Carry status is sampled on prior instruction cycle.	Ignore next instruction if C = 0 during prior cycle

*RAM-DIO Timing. When changing BU from addressing RAM (0, 1, or 2) to DI/O (3), the B Register value is updated in the cycle immediately following the modification instruction, but neither RAM nor DI/O accessing instructions are valid. In the second cycle following, the DI/O selected by the modified BU and BL may be set, reset, or tested. When changing BU from addressing DI/O (3) to RAM (0, 1, or 2) the B Register value is updated in the cycle immediately following the modification and RAM addressing instructions are valid (subject to the timing related to changing BL) except for SB, RB, SKBF instructions. During the one cycle immediately following changing BU the SB and RB instructions will set or reset a bit in RAM as well as the DI/O bit. The SKBF instruction is undefined during this cycle. In the second cycle following, these three instructions are valid.

Note that this result may be obtained in the PPS-4/1 by addressing the memory cell and setting the bit positions corresponding to zero in the mask to zero. In this case

```
RB  1
RB  4
```

accomplishes the AND function corresponding to the above example. It was not necessary to set the middle two bits since anything ANDed with a 1 is unchanged.

3.5.5 LOGICAL OR

The Logical OR instruction is usually used to cause the contents of an addressed memory cell to be logically ORed with the contents of a mask. In corresponding bit positions of either the mask or memory, if there is a 1, the contents of the memory in that bit position will result in a 1. Only if the corresponding bit positions of both the mask and memory are zero will the contents of the memory cell remain zero in that bit position. This is shown in the following example.

Example:

```
1100    Memory
0110    Mask
----
1110    Logical OR in Memory
```

This result is obtained in the PPS-4/1 by addressing the memory cell and setting the bit positions corresponding to ones in the mask to ones in memory.

```
SB  2
SB  3
```

The untouched bits remain what they were and the other bits are set to one so that the result is the same as the OR operation gave.

3.5.6 LOGICAL EXCLUSIVE OR

With the Exclusive OR instruction the contents of the address memory cell are Exclusively ORed with the mask. The appropriate bit in the memory will be set to a 1 only when the corresponding bit positions in the memory and Accumulator are different. Either a 10 or a 01 combination will cause the corresponding bit position in the memory to be set to 1. If the corresponding bit position in both words has 0 or 1 in both words, a 0 is placed into the memory cell in that bit position. An example of Exclusive OR is shown below:

Example:

```
1100    Memory
0110    Mask
----
1010    Exclusive OR in memory
```

This result is obtained a little differently than in the case of the AND and OR functions since the resulting bit pattern in memory is conditional. For the resulting bit to be a one it must be a one if the mask bit is 0 or a zero if the mask bit is 1. Otherwise, the bit is zero. Consequently, all four bits in memory must be examined and each bit set accordingly. An efficient process for forming Exclusive OR function is discussed in Paragraph 3.9.3. Note that the above example is not the general use for the Exclusive OR function in microcomputers which have the

Exclusive OR instruction. The general use is to see if two memory cells are exactly the same by Exclusively ORing them together. If they are the same, a test for zero will show the identity.

```
            [EOR]       RESULT IN ACCUMULATOR
            AISK 15     SKIP IF ACCUMULATOR IS ZERO
            T DIFF      DIFFERENT
SAME        [   ]       SAME
```

However, the PPS-4/1 microcomputer has a simpler procedure since it has the Skip If Memory Equals Accumulator instruction:

```
            SKMEA
            T DIFF      DIFFERENT
SAME        [   ]       SAME
```

3.5.7 SET CARRY (SC)

Although the primary purpose for the Carry flip-flop is for use in arithmetic operations for propagation of a carry or a borrow from one digit to the next, it may also be used for a temporary flag bit when it is available.

The Set Carry instruction has two basic purposes. One is to set up an initial condition for subtraction by adding in complement process and the other is to set a flag to the "one" state.

3.5.8 RESET CARRY (RC)

The Reset Carry instruction performs the opposite function to Set Carry. Namely, it resets the Carry flip-flop to the "zero" state. Its use for setting initial conditions for add operations has been illustrated earlier. Its other use is to set a flag to the "zero" state for later interrogation by the SKNC instruction.

3.5.9 SKIP IF NO CARRY (SKNC)

The Skip if No Carry (SKNC) instruction causes the next instruction in the execution sequence to be ignored if the Carry flip-flop is in the "zero" state. If the carry bit is a "one", the next instruction in the sequence will be executed normally. After arithmetic operation (AC and ACSK) there should be one cycle delay before attempting to test the carry with a SKNC instruction.

Basically the operation of the SC, RC, and SKNC instructions is similar to the operation of the SB, RB, SKBF instructions when the carry is not being used for arithmetic operations, except that the B Register does not have to be used in any way.

Like all of the other "skip" instructions, if the instruction is skipped (ignored), the microcomputer still takes the time to cycle past the ignored instruction. Consequently, if the microcomputer is operating with a precision external clock, all computation times are predictable since the time to execute generally does not change whether or not the instruction is executed.

3.5.10 BINARY LEFT SHIFT

Although there is no direct instruction for binary shift left in the PPS-4/1, the function may be readily performed. Binary left shift is functionally identical to multiplying by two or adding the number to itself. This is accomplished with the following routine:

```
        LBL     WORD    POINT TO LEAST SIGNIFICANT CHARACTER
        RC
BLSH    L               ! LOAD CHARACTER
        AC              ADD IT TO ITSELF
        XDSK            STORE IT AND POINT TO NEXT CHARACTER, TEST
        T       BLSH    IF NOT THRU, REPEAT
        :
```

This routine will shift up to 64 bits left one bit position. It fills the least significant bit position which was vacated by the shift with a zero. The most significant bit from the original data occupies the Carry flip-flop when the process is completed. Longer shifts than 64 bits may be accomplished by performing the routine as shown until the process is complete, then, leaving the Carry flip-flop alone, pointing to the next character in the sequence, and then repeating the steps shown above starting at BLSH.

3.5.11 BINARY RIGHT SHIFT

The process for accomplishing a binary right shift is similar since shifting a character left four times through the Carry flip-flop causes the initial contents of the carry to be shifted into the most significant bit position and leaves the carry loaded with the information that was originally in the least significant position. This is illustrated in the following example:

Example:

N_0 = ORIGINAL NUMBER
C = CARRY

MEMORY CONTENTS	C	WORD	
N_0	0	1 1 0 1	
		1 1 0 1	
N_1	1	1 0 1 0	N_0 SHIFTED LEFT
		1 0 1 0	
		1	CARRY
N_2	1	0 1 0 1	
		0 1 0 1	
		1	CARRY
N_3	0	1 0 1 1	
		0	CARRY
		1 0 1 1	
N_4	1	0 1 1 0	BINARY RIGHT SHIFTED WORD

ORIGINAL LEAST SIGNIFICANT BIT LEFT IN CARRY FOR PROPAGATION INTO MOST SIGNIFICANT BIT POSITION OF CHARACTER TO RIGHT FOR CONTINUING SHIFT.

The binary right shift is the equivalent of dividing a binary number by two. Usually for right shifts the most significant bit is copied into the vacated bit position so that a number expressed in the two's complement form for negative numbers will still be correct. If the number was positive the zero in the most significant bit position is left there. Similarly if the number was negative, the one is in effect left in the most significant bit position. This is accomplished in the following code by testing the most significant bit and setting the carry appropriately, prior to starting the shifting process.

```
        LBL     MSCH    POINT TO MOST SIGNIFICANT CHARACTER
        RC              CARRY ZERO IF MS BIT IS ZERO
        SKBF    4       TEST MOST SIGNIFICANT BIT
        SC              MS BIT IS ONE SO SET CARRY, OTHERWISE IGNORE
BRSH    L               ! LOAD CHARACTER
        AC              DOUBLE IT (LEFT SHIFT ONCE)
        X               ! SAVE DOUBLED CHARACTER
        L               ! LOAD DOUBLED CHARACTER N 1
        AC              DOUBLE IT (FORM N 2)
        X               ! STORE N 2
        L               ! LOAD N 2
        AC              DOUBLE IT (FORM N 3)
        X               ! SAVE N 3
        L               ! LOAD N 3
        AC              DOUBLE IT (FINAL RESULT N 4)
        XNSK            SAVE ANSWER, POINT TO NEXT CHARACTER, TEST
        T       BRSH    IF NOT THRU, REPEAT
```

3.6 PROGRAM REGISTER MODIFICATIONS (TRANSFER INSTRUCTIONS)

In the PPS-4/1 as in any other computer system, the Program Register (also referred to as P Register and Program Counter) will advance with each instruction causing the instructions in memory to be executed in a predictable sequence until either an unconditional or conditional transfer is commanded. Unconditional transfers are those that are not dependent upon the results of any computations, the results of any tests, or similar events dependent upon the data or program path. The unconditional transfer instructions in the PPS-4/1 are shown in Table 3-9.

3.6.1 PROGRAM COUNTER OPERATION

In the PPS-4/1 microcomputer system MM77, the Program Register is a 10-bit register which controls the sequence of instruction execution. It consists of two parts, the lower 6 bits are a counter which normally increments one count each clock cycle in a polynomial count sequence. The upper 4 bits are made up of a static register which must be set externally or under program control.

3.6.1.1 Program Counter Operation During Power Turn On

External setting of both the static register and the counter portion of the Program Register occurs during power turn on. The power-on reset initializes the Program Register to a predetermined state so that the program can start from a known location. The starting location when power is turned on is #1C0.

To ensure that all power-on transients have settled, it is recommended that the instruction stored in this initial location be a NOP (no operation) instruction.

Table 3-9. UNCONDITIONAL TRANSFER INSTRUCTIONS

Mnemonics	Op Code (Hexadecimal & Binary)	Name	Description	Symbolic Equation
NOP	00 0000 0000	No Operation (1 cycle-1 byte)	No Operation — do nothing. Ignore this instruction.	
T	C0-FF 11__ _____ (If in primitive subroutine area and transferring to page 15 80-BF 10__ ____)	Transfer on page (2 cycles-1 byte)	Set Lower 6 bits of Program Register (PL) to complement to 6 bit immediate field. If executed on pages 0-#13 leave Upper 5 bits (PU) unchanged. If executed from primitive subroutine area (pages 14 & 15) Set PU to 14 if addressing page 14, or PU to 15 if addressing page 15.	PL ← $[\overline{I(6:1)}]$ If executed from address in range #000-#2FF PU ← PU. If executed from address in range #380-#3FF if addressing address in range #380-#3BF PU ← 1110 or if addressing address in range #3C0-#3FF PU ← 1111
TL	1st byte (I^1): 30-3F 0011 _____ 2nd byte ($\overline{I_2}$): C0-FF 11__ ____	Transfer Long (3 cycles-2 bytes)	Transfer to an address on page 0 thru 7B (#000 thru #37F).	P(10:7) ← $[\overline{I_1(4:1)}]$ P(6:1) ← $[\overline{I_2(6:1)}]$
TM	80-BF 10__ ____	Transfer and Mark (2 cycles-1 byte)	Subroutine call to primitive subroutine page (#3C0 thru #3FF). (SA prior contents lost.) Push incremented P Register to SA. Transfer to primitive subroutine page location defined by complemented 6-bit immediate field.	SA ← P+1 Original SA lost PU ← 15 PL ← $[\overline{(6:1)}]$
TML	1st byte (I_1): 30-3F 0011 _____ 2nd byte ($\overline{I_2}$): 80-BF 10__ ____	Transfer and Mark Long (3 cycles-2 bytes)	Subroutine call to pages 0 thru 13 (#000 thru #37F). Push incremented P Register into SA. (SA prior contents lost.)	SA ← P+1 P(10:7) ← $[\overline{I_1(4:1)}]$ P(6:1) ← $[\overline{I^2(6:1)}]$
RT	02 0000 0010	Return (2 cycles-1 byte)	Return from subroutine. Pop SA into P.	P ← SA
RTSK	03 0000 0011	Return and Skip (2 cycles-1 byte)	Return from subroutine and skip (ignore) 1st instruction	P ← SA Ignore 1st instruction

3.6.1.2 No Operation (NOP) Instruction

The No Operation (NOP) instruction is used when no actual operation is desired, but when space must be allocated in memory for the Program Counter to count through a location for timing or other similar purposes. It is usually used when a skip function is known to always occur. The ignored instruction is loaded with a NOP to fill the unexecuted memory space. The NOP execution causes no other function than allowing the Program Counter to advance while not disturbing any register, memory or input output port.

3.6.2 THE PAGE CONCEPT IN PROGRAM MEMORY

Because only the lower 6 bits of the Program Register actually perform a counting operation, the Program Register after the power on reset, would cycle through an equivalent count of zero to 63 in decimal, then recycle through the same addresses unless the upper 4 bits were changed by one of the transfer instructions. Similarly, after a transfer instruction the lower 6 bits would continue to count from wherever they were set by the transfer instruction until the maximum count was reached and then the next count would reset back to zero and the Program Register would continue to count up from that point until the same sequence of instructions is cycled through again. Each group of 64 instructions which are addressed by the 6 bit counter portion of the Program Register is termed a page. Page 0 consists of Hex addresses 000 through 03F. Page 1 consists of addresses 040 through 07F. Page 2 consists of addresses 080 through 0BF and page 3 consists of addresses 0C0 through 0FF, etc. In the MM76 there are 16 pages. Pages 0-13 are the general program area, while pages 14 and 15 make up the primitive subroutine page. Page 15 is the entry page (SR0) and page 14 is the extension page (SR1).

Table 3-10. PAGE TRANSFER ADDRESSES

TML TL I_1	Page	Address Hex Range From	 To
3F	0	000	03F
3E	1	040	07F
3D	2	080	0BF
3C	3	0C0	0FF
3B	4	100	13F
3A	5	140	17F
39	6	180	1BF
38	7	1C0	1FF
37	8	200	23F
36	9	240	27F
35	10	280	2BF
34	11	2C0	2FF
33	12	300	33F
32	13	340	37F
T TM	Primitive Subroutine Page	380 3C0	3BF 3FF

TL = I_1 11XX XXXX
TML = I_1 10XX XXXX

3.6.3 PROGRAM REGISTER

The Program Reset organization is shown in Figure 3-10. The program address is the 10-bit word contained in the Program Register.

The figure shows organization of the program address in binary and also shows the program address in hexadecimal notation because this is the common format for referencing addresses. By convention, the hexadecimal format uses 00 as the most significant bits to complete the 12 bits indicated by three hexadecimal characters.

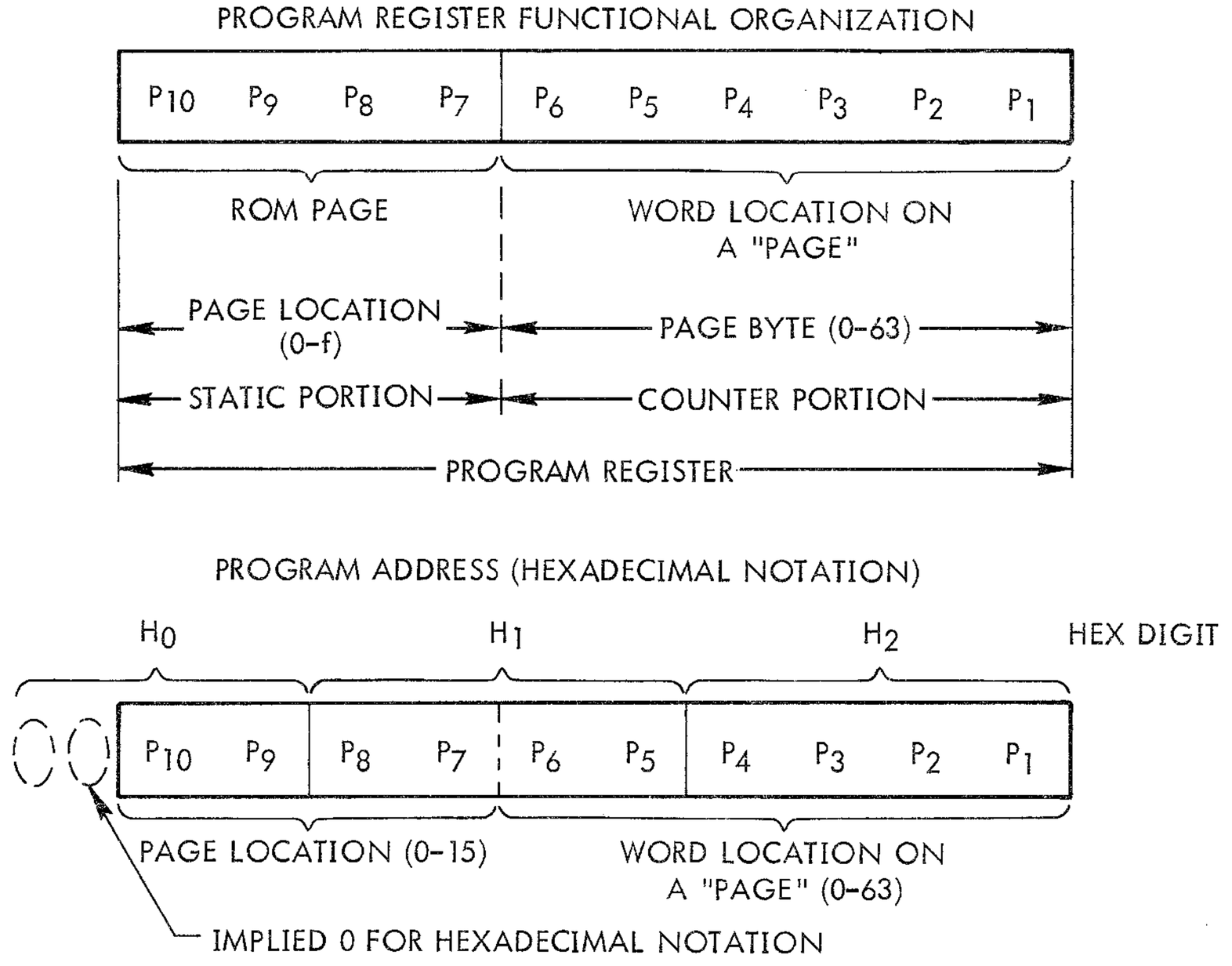


Figure 3-10. PROGRAM ADDRESS WORD — RELATIONSHIPS BETWEEN FUNCTIONAL ORGANIZATION AND PAGE, AND HEXADECIMAL NOTATION

3.6.4 UNCONDITIONAL TRANSFERS — TRANSFER (T), TRANSFER LONG (TL) OR BRANCH (B)

As in the case of several other PPS-4/1 instructions, there is a short-hand form of unconditional transfer which provides a great deal of flexibility but uses only one word of program memory. This instruction is the Transfer (T) instruction. It has been used in all of the examples up to this point because it is the most commonly used form of transfer instruction and is appropriate for the examples which all would be coded completely on one page. The T instruction causes an unconditional transfer to an instruction located on the current page addressed by the Program Register. In this case, the 6 bits in the immediate field after being complemented, replace the counter portion of the Program Register. This process is shown in Figure 3-11.

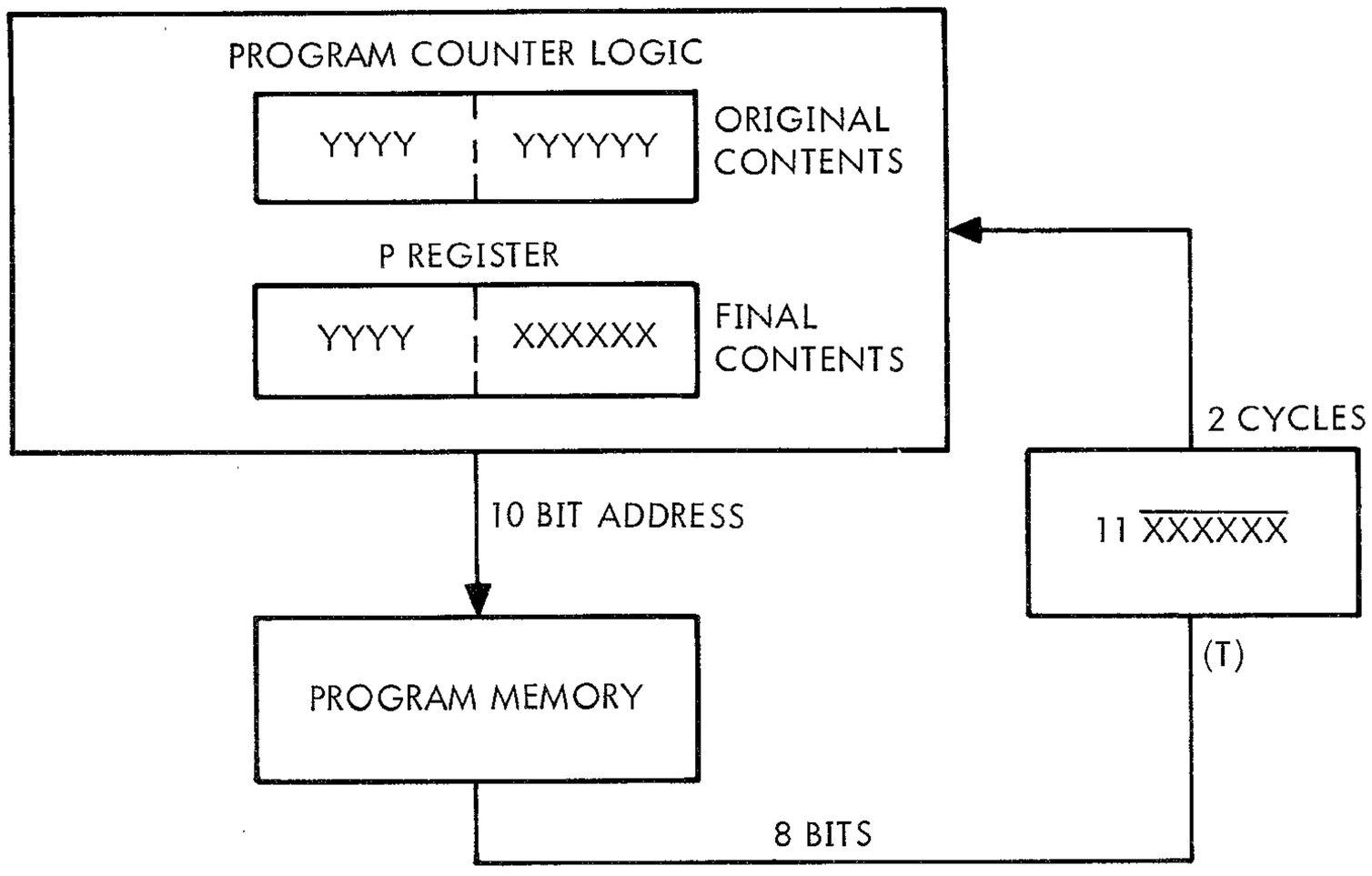


Figure 3-11. TRANSFER (T) ADDRESS FORMATION (INSTRUCTION TRANSFERS WITHIN A PAGE)

If the programmer does not insure that the location to which he wishes to transfer with this instruction is indeed on the same page any of the assemblers will flag this as an error.

The second unconditional transfer instruction is Transfer Long (TL). This instruction changes the complete contents of the Program Register so that it can transfer to any instruction in pages 0 through 13 (#000 through #37F). The four most significant bits of the Program Register are loaded from the complemented 4-bit immediate field portion of the first instruction of this two instruction command. The remaining bits are loaded from the second word which immediately follows in program memory. These relationships are shown in Figure 3-12.

The actual loading of the Program Register does not occur until all of the information has been received by the Program Register logic. In fact, if the bit patterns for I_1 as shown in Figure 3-12 are not followed by an acceptable code, that bit pattern will be interpreted as a Special No Operation (NOP) instruction. The actual NOP instruction, however, has no restrictions on its use. It always does nothing.

In TL instructions the I_2 byte code is identical to the T code for an on-page transfer. The I_1 code is designated TR for use in some of the reference tables. Such a table is the Transfer Instruction Coding Format shown in Table 3-11 which summarizes the use of each type of transfer instruction.

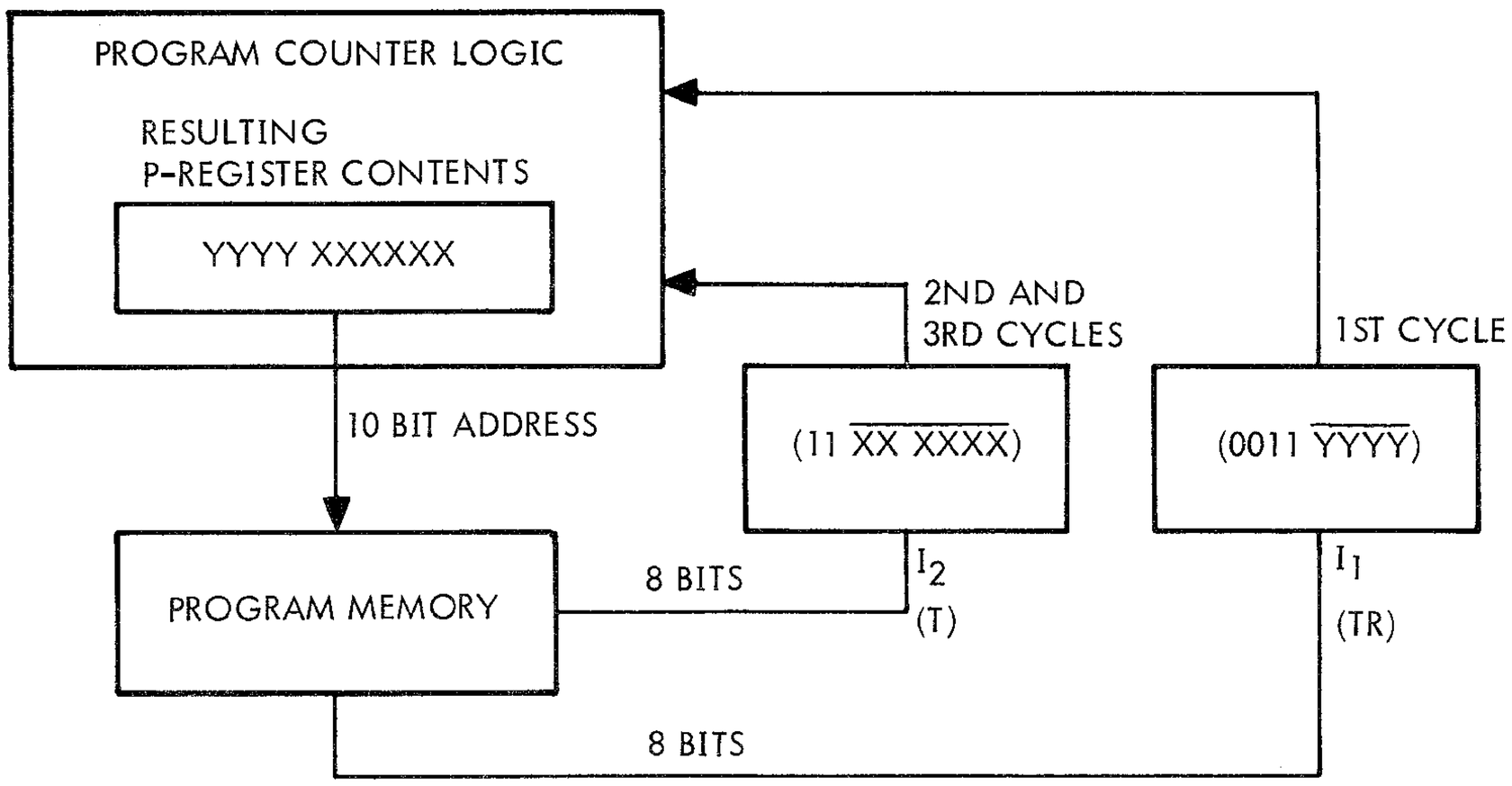


Figure 3-12. TRANSFER LONG (TL) ADDRESS FORMATION (GENERAL TRANSFER INSTRUCTION — FOR TRANSFER OFF OF PAGE TRANSFERS TO PAGES 0 THROUGH 13)(# THROUGH #37F)

Table 3-11. TRANSFER INSTRUCTION CODING FORMAT

		Operand Address Page			
	Mnemonic →	T	TL	T	TM
		0-13 (#000-#37F) Same Page	0-13 (#000-#37F) Different Page	14 (#380-#3BF) Subroutine Ext Page	15 (#3C0-#3FF) Subroutine Entry Page
Instruction Page	0-13 (#000-#37F)	T	I_1 = TR I_2 = T		
	14 (SR1) (#380-#3BF)		I_1 = TR I_2 = T	T	TM*
	15 (SR0) (#3C0-#3FF)		I_1 = TR I_2 = T	T	TM*

*When instructions are being executed on pages 14 or 15, TM is a Transfer — not a Subroutine Call.

3.6.5 TRANSFERS WITHIN PRIMITIVE SUBROUTINE PAGE

The special primitive subroutine page (occupying the block of memory from #380 through #3FF) has been reserved for storing the most basic, most often used subroutines. Therefore it is recommended that direct transfers to this region be avoided. This area is special from another aspect, it is twice as long as a normal 64 byte page, but the on-page transfer instruction, T, allows a one-byte transfer to any one of the 128 locations on the page. Since this is a primitive subroutine page, subroutine calls are not allowed to be made from this page. Consequently, the one-byte subroutine call code, TM, may be used to select the additional 64 locations.

The programmer stil uses the mnemonic, T, for the on-page transfer. If next instruction is to be in the area addressed by #3C0 to #3FF, the assembler will generate an instruction of the form 10XXXXXX. If the next instruction is to be in the area addressed by #380 to #3BF, the assembler generates an instruction of the 11XXXXXX type. The 10XXXXXX (or TM) form of the instruction always sets the upper 4 bits of the Program Register to 1111 (15). The 11XXXXXX (or T) form of the instruction, when executed within the primitive subroutine page, set the upper bits of the Program Register to 1110 (14). At all other locations in memory, the T operation code leaves the upper 4 bits of the Program Register untouched.

3.6.6 TRANSFER INSTRUCTION GENERATION VIA BRANCH (B)

It is not necessary for the programmer to be concerned about the code or format or address bit inversions or the specific hexadecimal address for any of the transfer instructions if any of the Rockwell PPS-4/1 MM76 assembly programs and other tools are used for program development. The programmer merely writes instructions such as T LOOP, TL NEXT, and the assembler program generates the proper code.

When the assembly is done on a system using the FORTRAN IV assembly, the programmer does not need to be concerned with using three different operation codes for the three types of transfers. In the FORTRAN system the operation code mnamonic B (Branch) causes the assembler to select the proper one, two, or three byte code to accomplish the transfer. Thus B LOOP and B NEXT will generate the same code as T LOOP or TL NEXT providing, of course, that the address corresponding to LOOP is on-page and the address corresponding to NEXT is on a different page with an address in the range of #000 to #37F.

When debugging a program, if there are minor changes, the programmer may wish to make them in machine language rather than edit and reassemble. For this reason a number of useful tables are provided in Appendix B. If the PPS Universal Assemulator is being used, a T or TM entry followed by the hexadecimal address desired will cause the proper code to be output.

3.6.7 THE SUBROUTINE CONCEPT IN THE PPS-4/1

One of the general programming techniques used in all computers is the technique of using subroutines for coding the processing functions which are used repeatedly. For instance, repeating the six instructions required for decimal add every time a decimal add is required in the processing, a subroutine may be written for the decimal add function, stored in memory in one section, and then called up for use when the program requires it. The subroutine technique, consequently, may save significant amounts of program memory, and significant amounts of coding time since the subroutine need only be written once.

Using subroutines, however, requires that some overhead be associated with the subroutine call to mark the place from which the call was made and to return to that place after the subroutine has been completed. This overhead may be handled by hardware (as it is in the PPS-4/1 family), or it may be accomplished by software (this option is also available in the PPS-4/1 when desired).

The PPS-4/1 microcomputer MM76 has four unconditional transfer instructions which have been particularly designed to operate subroutines efficiently. The two subroutine call instructions, Transfer and Mark (TM) and Transfer and Mark Long (TML) operate in a manner somewhat similar to the T and TL instructions except that both of these instructions also marks its place at the time of the transfer. Both of these instructions cause the address of the next instruction which would normally be executed, had not the transfer occurred, to be saved in the Save (SA) Register and whatever was in the SA Register before is lost.

The two return instructions used for getting back to the routine from which the subroutine was called are the Return (RT) and Return and Skip (RTSK) instructions. Both of these instructions cause the value that is in the SA Register to be popped into the Program Register (restoring the normal execution sequence of the calling routine). The RT instruction differs from the RTSK instruction in that when the RTSK instruction is used the first instruction after the return to the calling sequence is skipped (ignored). This gives the PPS-4/1 the capability to make a decision in the subroutine and branch back to the calling sequence at one of two places depending upon the results of the decision. When the RTSK instruction is used, the first instruction after the subroutine call is usually a transfer instruction of any of the unconditional transfer types.

Figure 3-13 shows the basic subroutine concept as implemented in the PPS-4/1.

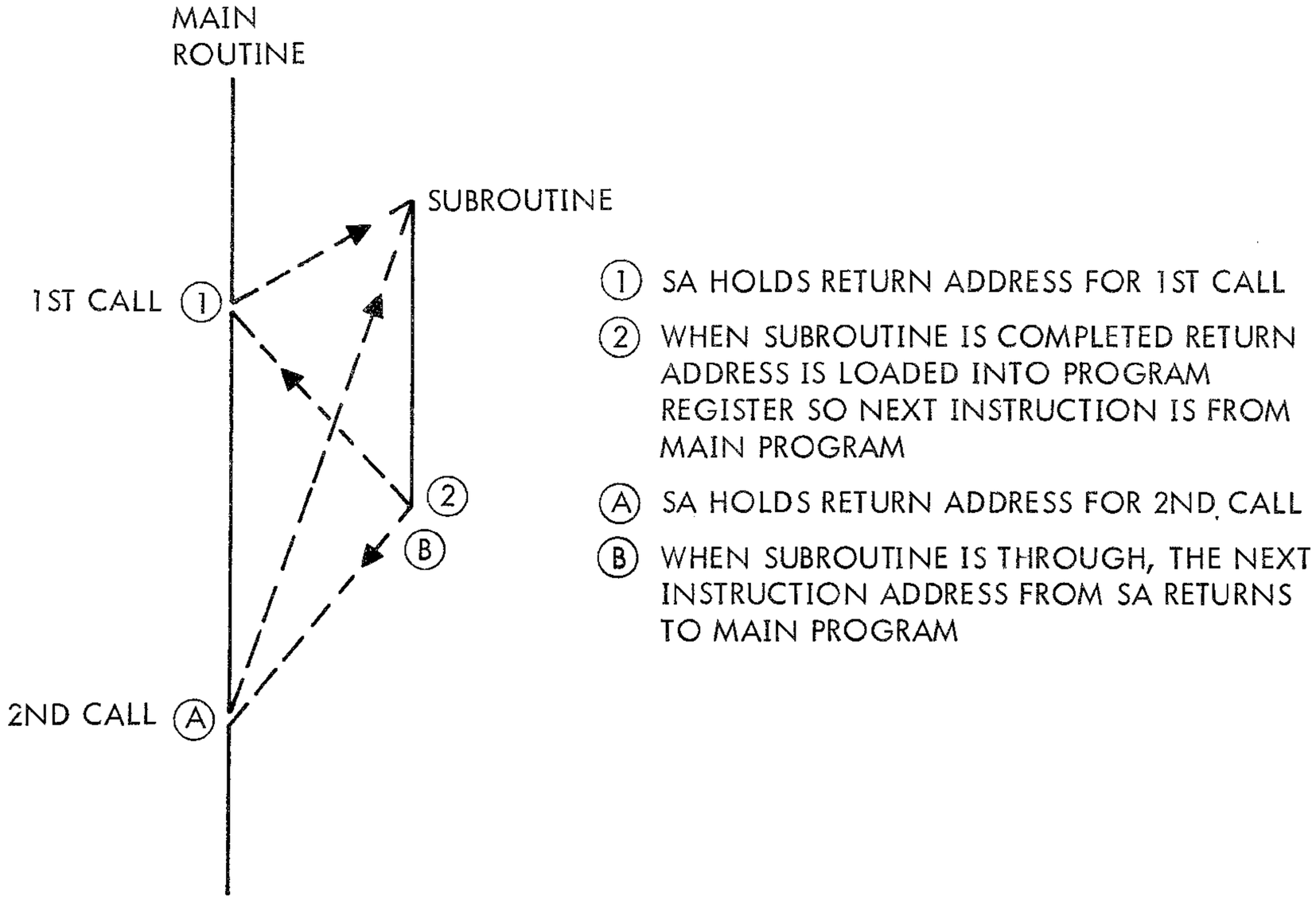


Figure 3-13. THE SUBROUTINE CONCEPT

When more than one level of subroutine is desired, the additional levels may be accomplished by software techniques. In this case prior to calling the subroutine, the programmer stores a unique code to identify the point to which the program should return. This code may be stored in a memory cell, in one of the internal registers such as S, or if some spare input output capabilities of the PPS-4/1 are available; the code may be output and then read back again when required. On completion of the extended nesting sburoutine, the extended subroutine can examine the stored code and execute a conventional transfer to the proper point. Note that in the case of the extended nesting, conventional transfers are used, not the Transfer and Mark type of instruction. If the transfer and mark type of instruction is used, the SA Register would be pushed and the address for the return to the main routine from the subroutine would be lost. A diagramatic example of an extended nesting subroutine is shown in Figure 3-14.

An example of a calling routine for an extended nesting subroutine is as follows:

```
        LBL     CALL    POINT TO SUBROUTINE RETURN IDENTIFICATION
        LAI     1       LOAD RETURN IDENTIFICATION (ONE IN THIS CASE)
        X               ! STORE
        TL      XSR1    TRANSFER TO EXTENDED SUBROUTINE
IRTN    (  )            RETURN POINT
        :
```

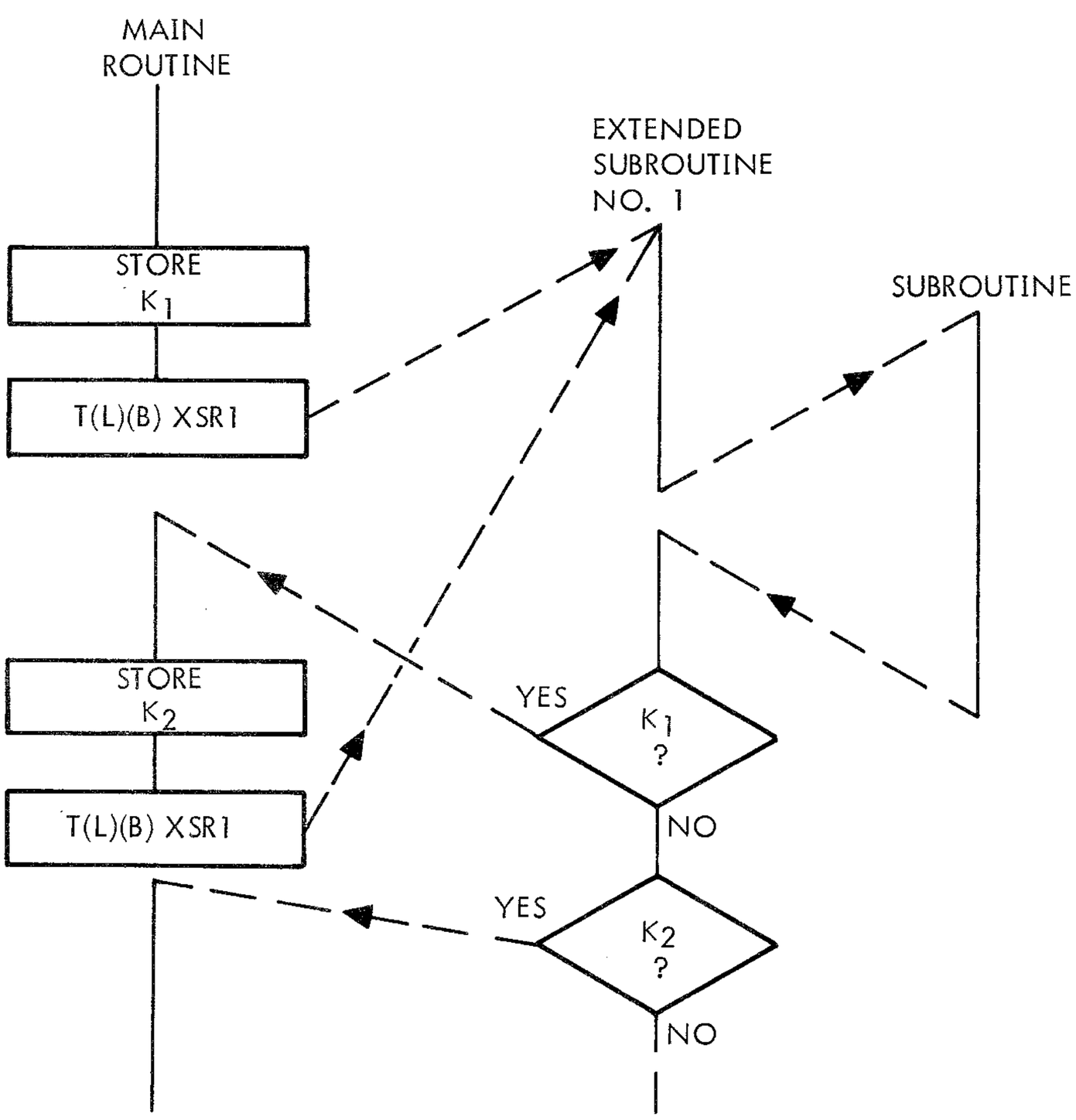


Figure 3-14. EXTENDED SUBROUTINE NESTING

Several methods for using the return identification code to point to the proper return location are available in the MM76. Some of these techniques using the SKMEA instruction for the table look up operation will be discussed in Section 3.7. However, a technique using the AISK instruction is shown below:

```
XSR1    (  )                EXTENDED SUBROUTINE ENTRY POINT
          .
          .
          .
        TML     1LSR        CALL SUBROUTINE
          .
          .
          .
        LBL     CALL        POINT TO IDENTIFICATION (ID) CODE
        L       0           PICK UP ID
        AISK    13          TEST ID FOR 3 CODE
        TL      3RTN        13+3 OVERFLOWS SO CODE WAS 3
        AISK    1           NO CARRY, SO TEST FOR ID OF 2
        TL      2RTN        14+2 OVERFLOWS SO CODE WAS 2
        AISK    1           NO CARRY, SO TEST FOR ID OF 1
        TL      1RTN        15+1 OVERFLOWS SO CODE WAS 1
        TL      0RTN        NO CARRY, SO CODE MUST BE 0
```

The XSR1 subroutine as depicted above, is one that was called from four different locations. All four were on different pages than XSR1 so they required the TL form of transfer.

When using the hardware subroutine calls, there is no need for the programmer to be concerned with the form of the return since the complete 10-bit Program Register is saved.

3.6.8 SUBROUTINE CALL INSTRUCTIONS — TRANSFER AND MARK (TM) AND TRANSFER AND MARK LONG (TML). ALSO BRANCH AND MARK (BM)

The Transfer and Mark (TM) instruction resembles the T instruction in that it is a one byte instruction, but the effect of the TM instruction is significantly different. The TM instruction, in addition to marking its place by pushing the Program Register into the hardware stack, causes a transfer to the upper half of the special primitive subroutine page. All entries to subroutines in this area (addresses #3C0 through #3FF) must use the TM instruction. No subroutines may be called from addresses in the primitive subroutine page (address #380 through #3BF) although an artifice of transferring off-page with a TL instruction and then calling the subroutine may be used.

Functionally, the TM instruction sets the upper 4 bits to 1111 and the lower 6 bits are taken from the immediate field in complement form. Figure 3-15 shows this in pictorial form.

Functionally, the TML instruction is identical to the TL instruction except that the TML instruction pushes the Program Register into the hardware stack. The loading of the Program Register occurs for this instruction exactly as shown in Figure 3-12 except that the I_2 operation code is 10XXXXXX instead of 11XXXXXX as shown. The operation of the stack is exactly the same as shown in Figure 3-15. The format for coding Transfer and Mark type instructions is shown in Table 3-12.

When one of the FORTRAN IV assemblers is being used it is permissible to use the mnemonic code BM for Branch and Mark. The assembler in that case will examine the transfer address and generate the appropriate form of the transfer and mark operation.

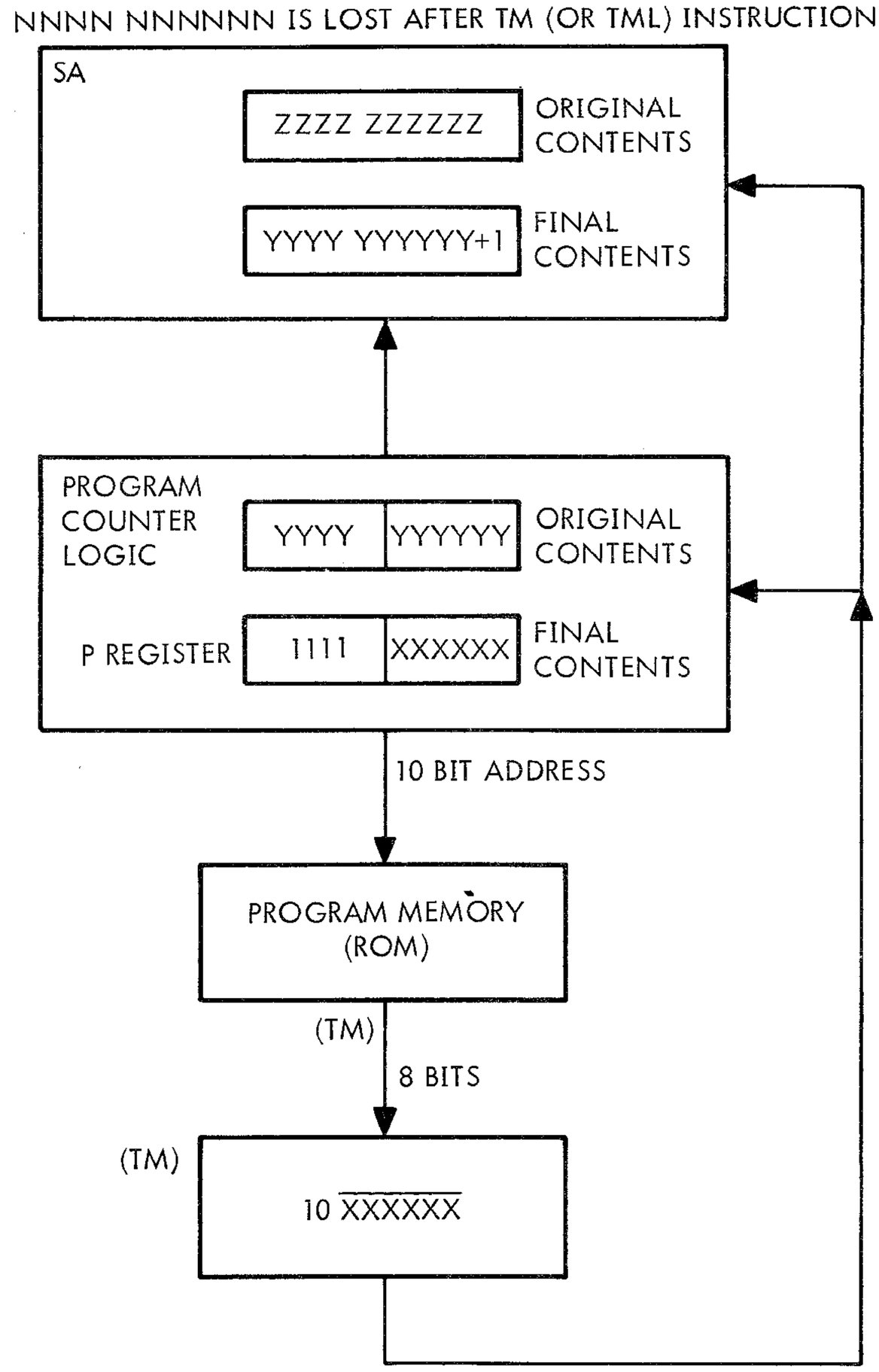


Figure 3-15. TRANSFER AND MARK (TM) ADDRESS FORMATION AND HARDWARE STACK OPERATION

Table 3-12. SUBROUTINE CALL INSTRUCTION CODING FORMAT

		Operand Address Page			
Mnemonic →		TML	TML		TM
		0-13 (#000-#37F) Same Page	0-13 (#000-#37F) Different Page	14 (#380-#3BF) Subroutine Ext Page	15 (#3C0-#3FF) Subroutine Entry Page
Instruction Page	0-13 (#000-#37F)	I_1 = TR I_2 = TM	I_1 = TR I_2 = TM		TM
	14** (SR1) (#380-#3BF)				
	15** (SR0) (#3C0-#3FF)				

** Subroutine calls are not allowed from Pages 14 and 15.

3.6.9 SUBROUTINE RETURN INSTRUCTIONS (RT AND RTSK)

The utility of the Return (RT) and Return and Skip (RTSK) instructions was discussed in Paragraph 3.6.7. This section discusses the internal operation of the PPS-4/1 when these instructions are encountered. The return process is shown in block diagram form in Figure 3-16.

The only difference between the RT and the RTSK instructions occurs after the process shown in Figure 3-16 is complete. The RTSK instruction will cause the instruction which starts at location YYYY YYYYYY+1 (the return address) to be ignored. The instruction at the return point may be any instruction, but if it is one of the two byte transfer instructions, the complete transfer will be ignored and actual execution will start after the extra byte is ignored.

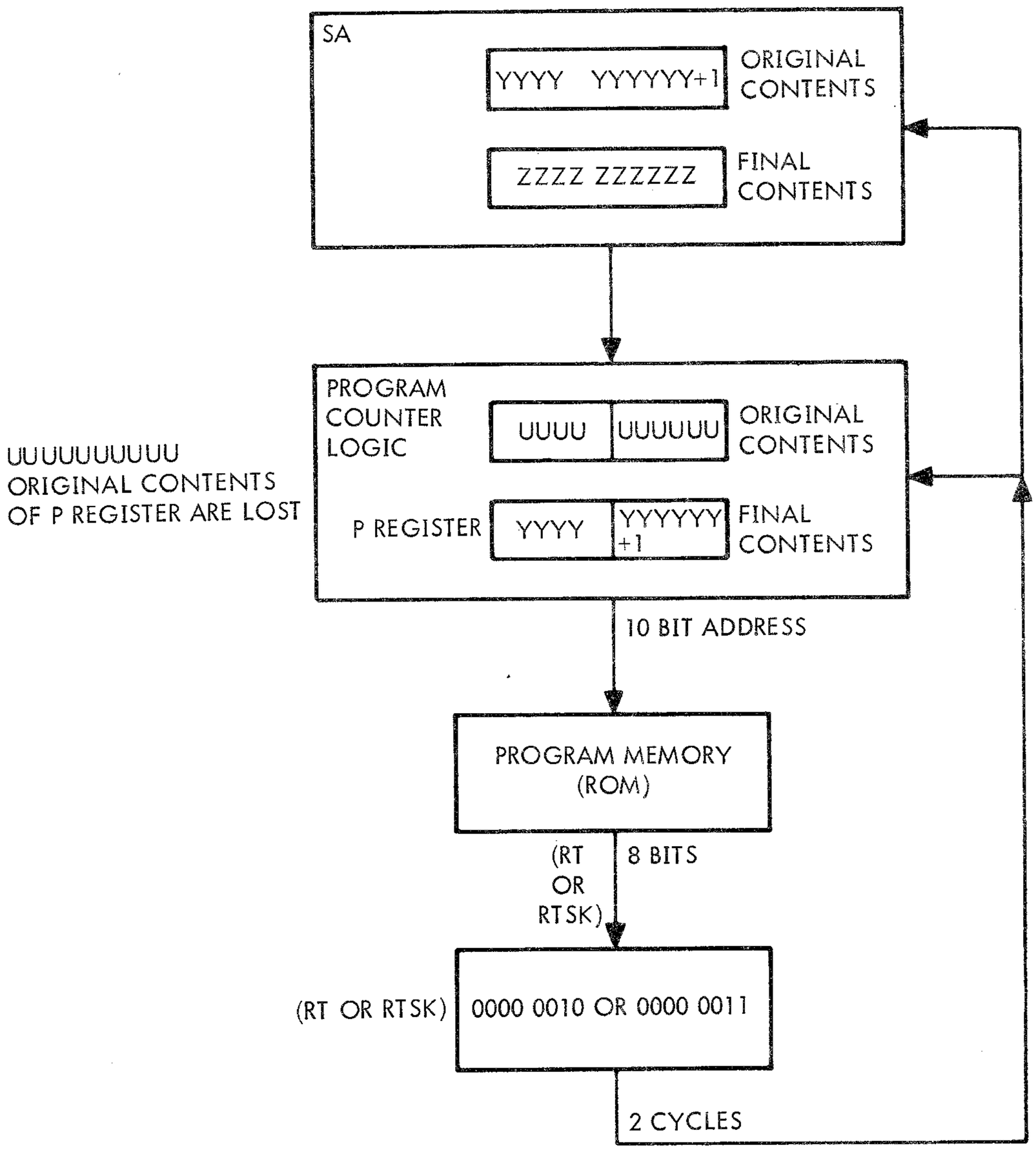


Figure 3-16. HARDWARE STACK OPERATION WITH RETURN (RT) OR RETURN AND SKIP (RTSK) INSTRUCTIONS

The capability in the PPS systems of performing a test in the subroutine and then returning to a location depending upon the results of the test is unique. Other microcomputer systems set a flag in the subroutine which identifies the results of the test and then test the flag after returning. This is another example of the efficiency of the PPS multifunction instructions.

3.6.10 SUBROUTINE EXAMPLES

All of the prior examples could have been written as subroutines. To convert these routines to subroutines it is only necessary to follow the last instruction in the set (generally a transfer back to repeat a loop) with an RT instruction. Each subroutine would be entered by the appropriate TM or TML instruction depending upon where in memory the programmer had chosen for the subroutine to reside.

As an example of the use of both th RT and RTSK instructions in a subroutine, a decimal add subroutine will be used. If at the completion of the addition, the Carry flip-flop is set it indicates that an overflow error has occurred. To perform the error test in the subroutine the following steps are coded:

```
DADD    L       1       DEC ADD ROWS 0+1
        AISK    6
        NOP
        ACSK
        T       *+2
        AISK    10
        XDSK    1
        T       DADD
        SKNC            TEST CARRY
        RT              CARRY, RETURN TO ERROR TRANSFER POINT
        RTSK
```

If the following subroutine call has been used, the subroutine will add the decimal digits in locations #00 through #04 to the locations #10 through #14 and leave the result in the register in row 1. If an overflow occurs, the subroutine should initiate a transfer to a subroutine called GOOF which is located in the area #000-#37F.

```
LB      #4
TM      DADD    DADD IN PRIMITIVE SUBROUTINE PAGE
TML     GOOF    RETURN FOR OVERFLOW ERROR
(               ) NORMAL RETURN POINT
```

3.6.11 PROGRAM MEMORY ALLOCATION FOR PRODUCTION AND EVALUATION DEVICES

In all of the discussions of program memory up to this point, it has been assumed that the memory is continuous from location #000 through #3FF. This capacity is available when the evaluation circuit is being used, but the production version of the PPS-4/1 MM76 has 640 bytes, not the 1024 bytes that it is capable of addressing with 10 bits. The production circuit is capable only of addressing pages 0-7 (000-1FF) and pages 14 and 15 (3C0-3FF). The active ROM memory which may be used in the production circuit is shown in Table 3-13.

Table 3-13. ROM MAP FOR PRODUCTION CIRCUITS

Hex Address Range	Page No.	Program Memory (ROM) Allocation		
#000-03F	0	64 bytes general program area	on page transfers (T) — one byte	TML Subroutine area
040-07F	1	64 bytes general program area		
080-0BF	2	64 bytes general program area		
0C0-0FF	3	64 bytes general program area		
100-13F	4	64 bytes general program area		
140-17F	5	64 bytes general program area	off page transfers (TL) — two bytes	
180-1BF	6	64 bytes general program area		
1C0-1FF	7	byte 0 = power on location — remainder general area		
- -	8-13	Not Used		
380-3BF	14	Extension of primitive subroutine area from page 15		
3C0-3FF	15	TM address area (1 byte subroutine calls start here and extend to page 14 or to other sections of memory)		

However, the evaluation circuit, since it can address all 1024 bytes, can make effective use of 1024 bytes of memory rather than the 640 bytes of the production circuit. This additional memory capacity is useful during the system development phase when patches may have to be inserted for debugging purposes and for convenience of not having to pack all the portions of the program together as densely as the final production program may require.

3.6.12 THE POLYNOMIAL COUNTER

The count sequence of the 6-bit polynomial counter is shown in Table 3-14. The count is obtained by Exclusively ORing the least two significant bits together, appending the bit generated at the most-significant-bit end and shifting all the bits right one bit position. There are two special cases. The initial count is set to 000000 and for the next count the value is forced to a 100000 configuration. The Exclusive OR technique is used until the last count value of 000001 is reached, then the count is forced to the 000000 configuration to repeat the count sequence.

The programmer generally need not be aware of the polynomial counter or its count sequence. If, for instance, a binary subtract subroutine is written in the following manner, the proper code will be generated by the assembler.

```
BSUB    SC
        L       1
        COM
        AC
        XDSK    1
        T       *-4
        RT
```

In the case the assembler does not arithmetically subtract four from the location of the transfer to get the location of the fourth instruction which was executed in sequence prior to the transfer instruction. The assembler generates the proper code for that polynomial counter address.

Table 3-14. PPS-4/1 POLYNOMIAL COUNT SEQUENCE

Polynomial Count Sequence			Polynomial Count Sequence		
Sequence	Binary Value	Equivalent Hex Value	Sequence	Binary Value	Equivalent Hex Value
0	0 0 0 0 0 0	00	32	0 0 1 0 0 1	09
1	1 0 0 0 0 0	20	33	1 0 0 1 0 0	24
2	0 1 0 0 0 0	10	34	0 1 0 0 1 0	12
3	0 0 1 0 0 0	08	35	1 0 1 0 0 1	29
4	0 0 0 1 0 0	04	36	1 1 0 1 0 0	34
5	0 0 0 0 1 0	02	37	0 1 1 0 1 0	1A
6	1 0 0 0 0 1	21	38	1 0 1 1 0 1	2D
7	1 1 0 0 0 0	30	39	1 1 0 1 1 0	36
8	0 1 1 0 0 0	18	40	1 1 1 0 1 1	3B
9	0 0 1 1 0 0	0C	41	0 1 1 1 0 1	1D
10	0 0 0 1 1 0	06	42	1 0 1 1 1 0	2E
11	1 0 0 0 1 1	23	43	1 1 0 1 1 1	37
12	0 1 0 0 0 1	11	44	0 1 1 0 1 1	1B
13	1 0 1 0 0 0	28	45	0 0 1 1 0 1	0D
14	0 1 0 1 0 0	14	46	1 0 0 1 1 0	26
15	0 0 1 0 1 0	0A	47	1 1 0 0 1 1	33
16	1 0 0 1 0 1	25	48	0 1 1 0 0 1	19
17	1 1 0 0 1 0	32	49	1 0 1 1 0 0	2C
18	1 1 1 0 0 1	39	50	0 1 0 1 1 0	16
19	1 1 1 1 0 0	3C	51	1 0 1 0 1 1	2B
20	0 1 1 1 1 0	1E	52	0 1 0 1 0 1	15
21	1 0 1 1 1 1	2F	53	1 0 1 0 1 0	2A
22	0 1 0 1 1 1	17	54	1 1 0 1 0 1	35
23	0 0 1 0 1 1	0B	55	1 1 1 0 1 0	3A
24	0 0 0 1 0 1	05	56	1 1 1 1 0 1	3D
25	1 0 0 0 1 0	22	57	1 1 1 1 1 0	3E
26	1 1 0 0 0 1	31	58	1 1 1 1 1 1	3F
27	1 1 1 0 0 0	38	59	0 1 1 1 1 1	1F
28	0 1 1 1 0 0	1C	60	0 0 1 1 1 1	0F
29	0 0 1 1 1 0	0E	61	0 0 0 1 1 1	07
30	1 0 0 1 1 1	27	62	0 0 0 0 1 1	03
31	0 1 0 0 1 1	13	63	0 0 0 0 0 1	01

When machine language coding is desired for machine language corrections, the tables in Appendix C may be used to help generate the code. If the assembler and editor functions are always used, the programmer would never need to know that the Program Register count was accomplished by means of a polynomial counter. The polynomial counter was used by the designers because the hardware mechanization is very much simpler than a binary counter.

3.6.13 CONDITIONAL TRANSFERS

The conditional transfer instruction which may be executed by the PPS-4/1 microcomputer family are combinations of skip instructions followed by a Transfer or Transfer Long instruction.

Thus a Transfer on Carry (TC) macro instruction is a Skip on No Carry (SKNC) followed by a Transfer (T) instruction. Similarly a Transfer on No Carry (TNC) macro instruction is made up of the following sequence:

```
SKNC
T          *+2        CARRY SO CONTINUE ON
T          ADDR       TRANSFER TO TNC ADDRESS
```

All of the macro instructions formed in this way are shown in Table C-2 in Appendix C.

3.7 REGISTER MANIPULATION AND COMPARISON

Many times the programmer will find it convenient to temporarily store information in the CPU portion of the PPS-4/1 without having to set up an address to store the information in memory. It is also convenient to be able to make decisions based upon the contents of various registers without having to perform any prior arithmetic or logical operations. The instructions summarized in Table 3-15 perform these register manipulation and comparison functions.

3.7.1 REGISTER EXCHANGE INSTRUCTION — EXCHANGE A AND S (XAS)

The Exchange A and S (XAS) instruction performs the basic exchange capability for moving the contents of the Accumulator to the S Register while at the same time bringing the prior contents of the S Register into the Accumulator.

Although the S Register is used in the serial input/output function of the PPS-4/1, it also may be used as an auxiliary storage register between serial input/output operations. In the MM76 the XAS instruction provides the only access to the S Register from the Accumulator. Once a serial input/output operation is under way, it is possible to perform an XAS operation while the data in the S Register is shifting. If an Exchange A and S instruction is executed while a serial input/output function is being performed, the resulting data in the S Register and Accumulator will vary depending upon the relative timing of the two instructions. The data shifted out will be a mixture of the original S Register contents and the data that was exchanged from the Accumulator. The specific results for each possible XAS instruction timing is shown in Table 3-17.

Table 3-15. REGISTER MANIPULATION AND COMPARISON INSTRUCTIONS

Mnemonics	Op Code (Hexadecimal & Binary)	Name	Description	Symbolic Equation
XAS	4E 0100 1110	Exchange A and S (1 cycle-1 byte)	Exchange the contents of the Accumulator and S Registers	A ◄► S
LSA	4C 0100 1100	Load S from A (1 cycle-1 byte)	Load the S Register from the Accumulator. Leave contents of Accumulator undisturbed.	S ← A A ← A
SKMEA	47 0100 0111	Skip if memory equals A (1 cycle-1 byte)	Skip (ignore) the next instruction if the contents of the memory cell addressed by the B Register equals the contents of the Accumulator. Memory and Accumulator unchanged.	If A = M ignore next instruction A ← A M ← M
SKBEI	1st byte (I_1): 30 0011 0000 2nd byte (I_2): 70-7F 0111 ____	Skip if BL equals immediate (2 cycles-2 bytes)	Skip (ignore) next instruction if BL equals 4-bit immediate field of second byte. BL is unchanged. This instruction should not be used on subroutine pages.	If BL = $I_2(4:1)$ ignore next instruction BL ← BL
SKAEI	1st byte (I_1): 30 0011 0000 2nd byte (I_2): 60-6F 0110 ____	Skip if A equals immediate (2 cycles-2 bytes)	Skip (ignore) next instruction if Accumulator equals complemented 4-bit immediate field of second byte. Accumulator is unchanged. This instruction should not be used on subroutine pages.	If A = $[\overline{I_2(4:2)}]$ ignore next instruction A ← A

Also see XAB, LBA in Table 3-4.

3.7.2 LOAD S FROM A (LSA)

The Load S from A instruction provides the capability of transferring the contents of the Accumulator to the S Register without disturbing the data in the Accumulator. This instruction is used when the prior contents of S may be destroyed and the contents of the Accumulator need to be retained.

3.7.3 MEMORY COMPARE — SKIP IF MEMORY EQUALS ACCUMULATOR (SKMEA)

The Skip if Memory Equals Accumulator (SKMEA) instruction provides a direct capability of comparing the contents of the Accumulator with the contents of the addressed memory cell. This instruction may be used in a variety of ways. It is used in the PPS-4/1 systems to perform the same function for which an Exclusive OR instruction is normally used. An Exclusive OR between the contents of an addressed memory cell and the Accumulator will produce all zeros in the Accumulator if the two values are identical. The Accumulator then would be tested for zero (in the PPS-4/1 systems the SKAEI 0 instruction discussed in Paragraph 3.7.5) to determine if the values are

equal. Using the SKMEA instruction it is only necessary to address the memory cell to be tested and execute SKMEA instruction. If the contents are not identical, the next instruction in sequence would be executed, but if they are, the next instruction is skipped (ignored).

One application of the SKMEA instruction could be to use it with extended subroutine nesting to determine the proper return point.

The SKMEA instruction is also used when it is desired to make comparisons between memory and the Accumulator without disturbing the contents of either. This differs from most of the testing techniques discussed to this point. For instance, if testing of a magnitude is accomplished by an AISK instruction, the contents of the Accumulator are modified each time a test is performed.

The SKMEA instruction may also be used for efficient table searches. For instance, if it is desired to search a row of memory for a particular value and identify the location of the first match, the following routine may be used:

```
        LBL     VALU    POINT TO MATCH VALUE
        L               LOAD IT
        LBL     TABL    POINT TO TABLE
SRCH    SKMEA           COMPARE VALUE WITH TABLE
        T       NOTV    NOT VALUE SO TRY AGAIN
        XAB             VALUE SO LOAD BL INTO ACCUMULATOR
        T       HERE    TRANSFER TO NEXT ROUTINE
NOTV    DECB            POINT TO NEXT TABLE VALUE
        T       SRCH    REPEAT SEARCH
        [        ]
```

3.7.4 SKIP ON B LOWER EQUALS IMMEDIATE (SKBEI)

Up to this point in this manual all of the routines which move data in some way have been automatically terminated by the contents of the B Register counting from zeros to all ones (XDSK or DECB) or counting from all ones to all zeros (XNSK or INCB). Thus it has been necessary to utilize memory registers which occupy either the left end of a row or the right end of a row in memory. With the Skip on B Lower Equals Immediate (SKBEI) instruction the register processing may be terminated automatically at any point within a row. For instance, in the following routine:

```
        LBL     #19     POINT TO INITIAL CHARACTER
REPT    L       2       LOAD AND POINT TO ROW 3
        XNSK    2       STORE AND SET UP NEXT VALUE
        SKBEI   #C      TEST FOR THRU
        T       REPT    NOT THRU, REPEAT UNTIL THRU
        [        ]
```

the three characters occupying memory cells #19, #1A, and #1B are moved to the corresponding locations in row 3. When the exchange with the last character stores results into memory cell #3B, the XNSK 2 instruction increments B Lower to the value #C. The SKBEI instruction compares B Lower with its immediate field, #C, and since they are equal, the transfer instruction which had been executed on all prior passes through the loop is skipped to terminate the process.

The SKBEI instruction is a two-byte instruction. The bit pattern for the first byte of the SKBEI instruction is identical to the TR byte used in the TL and TML instructions and the bit pattern used for the second byte which is identical to a Load A Immediate instruction.

3.7.5 SKIP IF ACCUMULATOR EQUALS IMMEDIATE VALUE (SKAEI)

The Skip on Accumulator Equals Immediate (SKAEI) instruction is similar to the SKBEI instruction in that it compares the immediate field with a CPU register. The SKAEI instruction compares the contents of the Accumulator with the inverted 4-bit immediate field of the instruction and skips if the two are equal. However, the SKAEI instruction is unlike the SKBEI instruction in one respect.

The SKAEI instruction is a hardware macro made up of 2 bytes. The first byte is the 8-bit code for the TR code used to signify the banked form of transfers in the TL and TML instructions. The second byte is identical to the 8-bit pattern for AISK instruction. Note that even though the immediate field for this instruction is inverted, the assembler produces the proper format to test the Accumulator for zero; the programmer writes SKAEI 0.

If SKAEI and SKBEI are used on the short subroutine pages, they will reset the flip-flop which allows the program to transfer from an address in this area (#380-#3FF) to an address on SR0 (#3C0-#3FF) without affecting the SA Register. Therefore, if an SKBEI or SKAEI instruction is executed on one of these pages followed by a TM, the contents of the SA Register will be replaced with an address in this area. The return instruction which follows well then cause the new value of the SA Register to be placed in the Program Register, which would force the program into a continuous loop on the Subroutine Page.

3.8 INPUT/OUTPUT INSTRUCTIONS

All of the members of the PPS-4/1 family have an extensive input/output capability. The instructions for utilizing this capability in the MM76 are summarized in Table 3-16.

3.8.1 DISCRETE INPUT/OUTPUT INSTRUCTIONS — SET OUTPUT SELECTED (SOS), RESET OUTPUT SELECTED (ROS) AND SKIP ON INPUT SELECTED LOW (SKISL)

The ten discrete input/output ports and the two flip-flops associated with the two conditional interrupt input ports may be set (tied to VSS) by a Set Output Selected (SOS) instruction, reset (put into a float situation so that it may be pulled to a -V condition) by a Reset Output Selected (ROS) instruction or tested by a Skip on Input Selected Low (SKISL) instruction.

The selection of the particular discrete input/output port or flip-flop associated with the pseudo interrupt lines is accomplished by addressing the port using BL and setting BU to 3. In all of the ten discrete input/output lines the output must be reset via an ROS instruction to the float state before the SKISL instruction may be used to input the signal state of the particular line addressed. When the SKISL instruction is used, the level of the signal on the input port will determine whether or not the next instruction in the sequence is to be skipped. If the port voltage is high (VSS), the next instruction sequence will be executed, and if the signal on the port is low, the SKISL instruction will cause the next instruction in the sequence to be skipped (ignored).

Table 3-16. INPUT/OUTPUT INSTRUCTIONS

Mnemonics	Op Code (Hexadecimal & Binary)	Name	Description	Symbolic Equation
SOS*	10-13 0001 00__	Set Output Selected (1 cycle-1 byte)	Set the Discrete input/output line DI/O (BL) selected by BL (BU must be 3) to VSS. If BL equals 10 set the discrete input 1 (DIN1) flip-flop. If it equals 11 set the discrete input 0 (DIN0) flip-flop.	DI/O (BL)← VSS by Ø2 of next instruction If BL = 10 (INT1) DIN1 F/F ← 1 (set) If BL = 11 (INT0) DIN0 F/F ← 1 (set)
ROS*	14-17 0001 01__	Reset Output Selected (1 cycle-1 byte)	Reset the discrete input/output line selected by BL (BU must be 8) to -V. If BL equals 10 reset the discrete input 1 (DIN1) flip-flop. If it equals 11 reset the discrete input 0 (DIN0) flip-flop.	DI/O (BL)← -V by Ø2 (float output) of next instruction If BL = 10 (INT1) DIN1 F/F ← 0 (Reset) If BL = 11 (INT0) DIN1 F/F ← 0 (Reset)
SKISL*	08-0B 0000 10__	Skip on input selected low (1 cycle-1 byte)	Skip (ignore) next instruction if the discrete input/output line selected by BL (BU must be 3) is low (-V). The discrete input/output line must be in float state so that external signal may pull line to VSS or -V condition for testing.	When DI/O (BL) is in float state skip next instruction if the corresponding input port = -V. Sample at Ø3 of prior cycle.
IBM	18 0001 1011	Input Channel B Masked (1 cycle-1 byte)	Input the signals on RIO5 thru RIO8 ANDed with the Accumulator to the Accumulator.	A ← RIO(8:5) VSS = 0 in A -V = 1 in A RIO(8:5) sampled during Ø2
OB	19 0001 1001	Output Channel B (1 cycle-1 byte)	Output the Accumulator to RIO8 thru RIO5 by latching corresponding bits in the B Buffer. Output is stable by Ø2 of the next instruction.	RIO(8:5) ← B Buffer by Ø2 ← A 1 = -V 0 = VSS
IAM	1A 0001 1010	Input Channel A Masked (1 cycle-1 byte)	Input the signals on RIO4 thru RIO1 ANDed with the Accumulator to the Accumulator.	new A ← A and RIO(4:1) 1 = -V, 0 = VSS RIO(4:1) sampled at Ø2
OA	18 0001 1000	Accumulator to Channel A (1 cycle-1 byte)	Output from Accumulator to Channel A	RIO(4:1) ← A 1 = -V, 0 = VSS

*RAM-DI/O Timing. When changing BU from addressing RAM (0, 1 or 2) to DI/O (3), the B Register value is updated in the cycle immediately following the modification instruction, but neither RAM nor DI/O accessing instructions are valid. In the second cycle following, the DI/O selected by the modified BU and BL may be set, reset, or tested. When changing BU from addressing DI/O (3) to RAM (0, 1 or 2) the B Register value is updated in the cycle immediately following the modification and RAM addressing instructions are valid (subject to the timing related to changing BL) except for SB, RB, SKBF instructions. During the one cycle immediately following changing BU the SB and RB instructions will set or reset a bit in RAM as well as the DI/O bit. The SKBF instruction is undefined during this cycle. In the second cycle following, these three instructions are valid.

Table 3-16. INPUT/OUTPUT INSTRUCTIONS (continued)

Mnemonics	Op Code (Hexadecimal & Binary)	Name	Description	Symbolic Equation
IOS	4D 0100 1101	Input/Output Serial (1 cycle overlapped -1 byte)	Simultaneously shift out the contents of the S Register over serial output line Data 0 while loading the contents of the S Register thru serial input line Data I. Most significant bit in S Register is output at all times. Shift at 1/2 clock rate. Shift completed after 8 cycles overlapped with any instructions.	Shift Data 0 ← S ← Data I, Most significant bit first. 1/2 clock rate shift pulses → clock 1 = -V, 0 = VSS Data I sampled at Ø4. Data O stable by Ø2
I1	4A 0100 1010	Input Channel 1 (1 cycle-1 byte)	Input Channel 1 [PI(4:1)] to Accumulator.	A ← PI(4:1) -V =1, VSS = 0
I2C	4B 0100 1011	Input Channel 2 and complement (1 cycle-1 byte)	Input PI(8:5), complement and load into Accumulator	A ← [PI(8:5)] Therefore VSS on PI line = 1 in A -V on PI line = 0 in A. PI(8:5) sampled at Ø3
INT0L	04 0000 0011	Interrupt 0 High (1 cycle-1 byte)	Skip (Ignore) next instruction if signal on INT0 line is low (-V)	If INT0 is low (VDD), ignore next instruction. Sampled at Ø3.
INT1H	05 0000 0100	Interrupt 1 Low (1 cycle-1 byte)	Skip (ignore) next instruction if signal on INT1 line is high (VSS)	If INT1 is high (VSS), ignore next instruction. Sampled at Ø1.
DIN0	08 0000 0110	Discrete Input 0 (1 cycle-1 byte)	Skip if INT0 flip-flop is reset and set INT0 flip-flop	↑ = INT0 F/F reset
DIN1	07 0000 0111	Discrete Input 1 (1 cycle-1 byte)	Skip if INT1 flip-flop is reset and set INT1 flip-flop	↓ = INT1 F/F reset

Note that the input port is actually sampled during the previous cycle and the information is held internally to be added upon if there is a SKISL instruction. Consequently the B Register must be pointing to the desired port during the sample time. The following example will illustrate this.

```
T1      LB      3       POINT TO #03
        L       1
        X       2       EXCHANGE
        LB      7       POINT TO #07
T2      SKISL           TEST DI/O 3
        T       3HI
```

The SKISL instruction at T2 skips if DI/O 3 was low even though the B Register was pointing to #07 when the SKISL instruction was executed. However the sampling occurred while the B Register was still pointing at #33.

For an explanation of DIN0 and DIN1 instructions refer to Paragraph 1. 3. 8.

3.8.2 INPUT/OUTPUT ACCUMULATOR (IAM, OA)

The input and output to/from the Accumulator is accomplished by means of a single instruction, Input Accumulator. When power is applied to the PPS-4/1 the flip-flops in the A Buffer register are automatically all set to the float state (1111). Consequently, an external signal on the RIO1 through RIO4 input/output ports may pull the voltage level on the pin to a VSS or to a -V level. If the level is high (VSS), the input will be interpreted as a zero, and if the input is low (-V), the input will be interpreted as a one. If the contents of A Buffer are not 1111, any bit input in a position corresponding to a zero will automatically be input as a zero. This allows a masked input capability. This process may be thought of as a logical AND between the four bits in the A Buffer and the four signal levels applied to the input pins. The circuit designer must design his external circuits so that it may be connected to VSS when making use of this masking capability. These inputs are sampled at phase 3.

At the same time that the data are being input, the contents of the Accumulator are output to the latches in the A Buffer. These latches control the output on the RIO4 through RIO1 input output ports. The outputs will either float (in bit positions which were originally 1 in Accumulator) or will be tied to VSS depending on the data transferred to the latches.

The outputs are stable by phase 2 of the next instruction cycle although minimum impedance is not achieved until after phase 4.

3.8.3 CHANNEL ONE INPUT

Channel 1 consists of a 4-bit parallel input port tied to pins PI1 through PI4. This input port is designed in the PPS-4/1 MM76 system to be used as a simultaneous input and test value port. The instruction that inputs information through this port is the Input Channel 1 instruction. The 4 bits input by this instruction are put into the Accumulator.

A typical use for this instruction would be as part of a keyboard input routine. In the example below the concept is to react to a key input by jumping to a decode routine (designated KEYR) or to determine that there is no contact closure in which case the program will execute a no key (NKEY) program. The program is as follows:

```
I1
AISK    #F      SKIP IF NO RETURNS
T       KEYR    ANY INPUT OVERFLOWS
NKEY    [ ]     NO KEY INPUT
```

The channel 1 inputs are sampled at phase 1.

3.8.4 CHANNEL TWO INPUT

Channel 2 consists of the 4 bits tied to input pins PI5 through PI8. The input instruction for this channel is Input Channel 2 and Complement (I2C). This instruction causes the information which is input on pins PI5 through PI8 to be complemented and replace the contents of the Accumulator. On all of the other 4-bit input ports the signal level of VSS will be interpreted as a zero and signal level of -V will be interpreted as a one. On this port, however, because of the complement function, and input signal of VSS will be interpreted as a one and an input signal of -V will be inserted into the Accumulator as a zero.

The channel 2 inputs lines are sampled at phase 3.

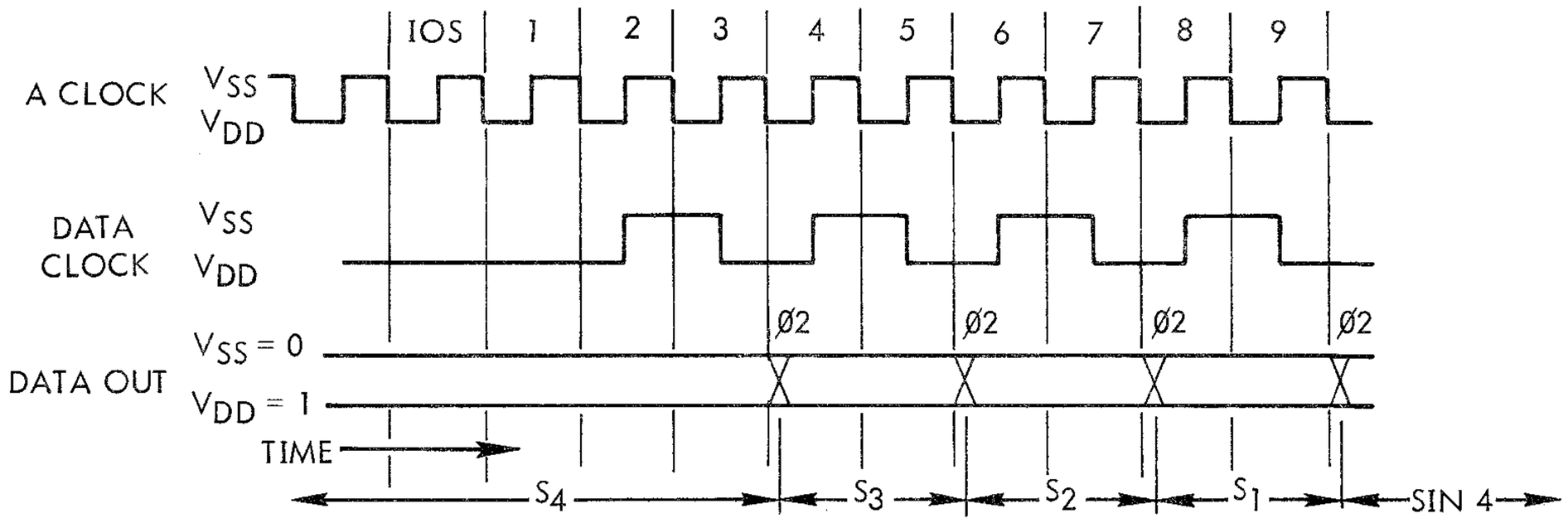
3.8.5 CONDITIONAL INTERRUPTS

All the PPS-4/1 systems have the capability of reacting to a conditional interrupt. The signal input on lines INT0 and INT1 may be tested directly by means of the instruction Interrupt 1 High (INT1H) instruction for the INT0 line or the instruction Interrupt 0 Low (INT0L) for the INT0 line. When the INT1H instruction is executed, if the signal on the INT1 input port is at a VSS during phase 1, the next instruction in sequence will be skipped. If it is high, of course, the next instruction will be executed. Similarly, the INT0L instruction interrogates the input port INT0 but in this case the skip occurs if the signal on the input line is at the VDD potential during phase 2 and no skip occurs if the signal is positive. These two signals present an easy to use, zero set up time technique for testing the status of two input signals.

3.8.6 SERIAL INPUT/OUTPUT

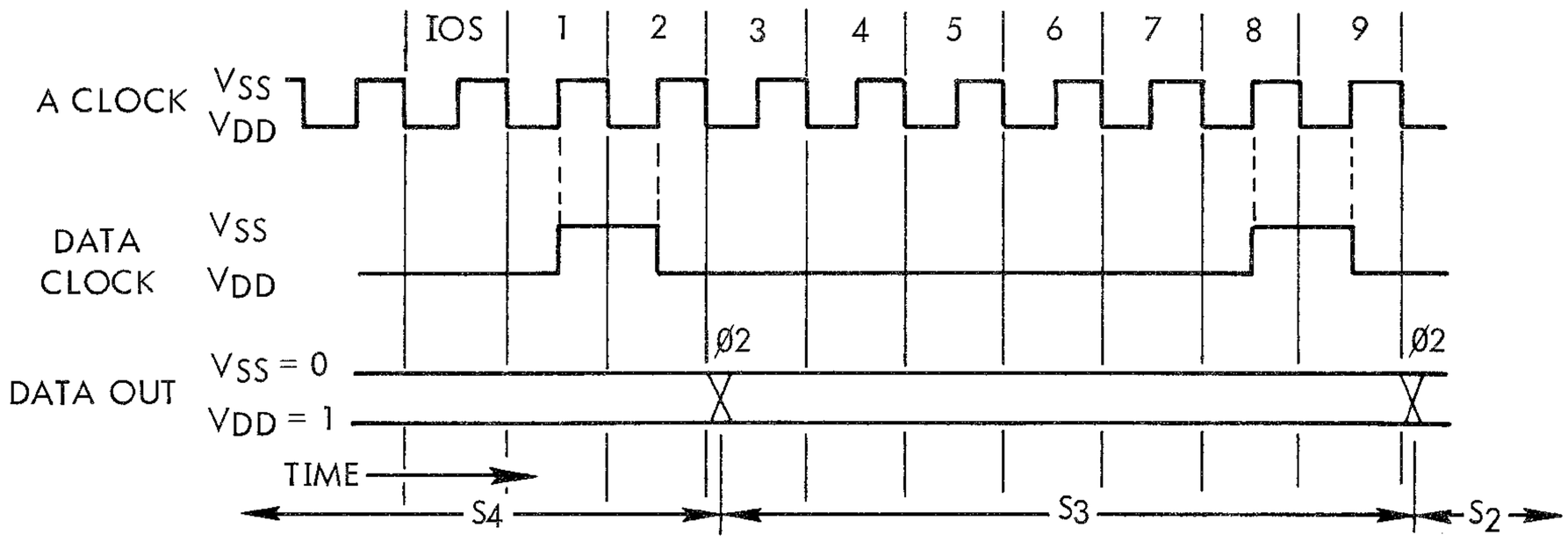
The serial input/output capability in the PPS-4/1 system is extremely flexible. The S Register which serves as a parallel-in parallel-out serial shift register is the primary buffer between the Accumulator and the Serial Input/Output ports. The Input/Output Serial (IOS) instruction causes the simultaneous serial input/output through the 4-bit S Register. When the IOS instruction is executed, the shift clock counter is set to a zero state. Then it successively counts through 8 cycle times. The shift clock, during the 8 cycle times produces 4 shift pulses as shown in Figure 3-17. Consequently, the shift rate when the IOS instruction is executed is one-half the clock rate of the microcomputer.

The shift clock signal may also be driven externally. When the shift clock is driven from a -V to VSS back to -V condition, the contents of the S Register will automatically shift left one bit position for each clock time that the signal is high as shown in Figure 3-18. Under either internal or external clock control the signal level on the serial input line will be interpreted to load zero into the least significant bit position of the S Register if the input line was at VSS. If it was at -V it will be interpreted as a "one". Simultaneously, the bit which was formerly in the bit-3 position of the S Register becomes the most significant bit and is output each time the register shifts. A zero in the most significant bit of the S Register will produce a VSS output and a one will produce a -V output. When a total of four shifts have occurred the entire original contents of the S Register will have been shifted out and the signal provided on the serial input line will have produced four input bits in the S Register. The operation of the serial input/output function is identical for both the internally clocked and externally clocked situation.



NOTE: S4, S3, S2, S1 = Original contents of S Register

Figure 3-17. TIMING OF INTERNALLY CONTROLLED SERIAL DATA INPUT/OUTPUT



NOTE: S4, S3, S2, S1 = Original contents of S Register

Figure 3-18. TIMING OF EXTERNALLY CONTROLLED SERIAL DATA INPUT/OUTPUT

The serial input/output capability of the PPS-4/1 system may be used in a variety of ways. Since the most significant bit in the S Register is always available on the serial output port it provides the capability of an additional discrete input/output line of the S Register is not being used as a temporary storage register. Similarly the serial output port can be used to generate pulses of 2, 4, or 6 cycle times in width merely by loading the Accumulator with a 0100, a 0110, or a 0111 code and recircling the serial output to the serial input. In this situation each time an IOS instruction is executed, the output will transition from zero to -V for the time determined by the bit pattern. Another option, of course, would be to put 0101 in the S Register and two pulses would be obtained each time an IOS instruction is executed.

Although the IOS instruction takes 9 cycle times to complete its operation, these 9 cycle times may be used by the programmer for any function the programmer desires. The shift operation goes on independently of the remainder of the control operations in the PPS-4/1. The programmer may, if he wishes, execute an XAS instruction while the shift operation is going on. This is discussed in Paragraph 3.7.1 and illustrated in Table 3-17. The final results, as shown, are dependent on the timing of the execution of the XAS instruction.

Table 3-17. RESULTS OF XAS PERFORMED DURING IOS OPERATION

Data Shifted Out	Resulting S	Resulting A	
	S_4 S_3 S_2 S_1	A_4 A_3 A_2 A_1	Initial Condition of IOS
A_4 A_3 A_2 A_1	I_4 I_3 I_2 I_1	S_4 S_3 S_2 S_1	Results after XAS at IOS + 1
A_4 A_3 A_2 A_1	I_4 I_3 I_2 I_1	S_4 S_3 S_2 S_1	Results after XAS at IOS + 2
S_4 A_4 A_3 A_2	A_1 I_3 I_2 I_1	S_3 S_2 S_1 I_4	Results after XAS at IOS + 3
S_4 A_4 A_3 A_2	A_1 I_3 I_2 I_1	S_3 S_2 S_1 I_4	Results after XAS at IOS + 4
S_4 S_3 A_4 A_3	A_2 A_1 I_2 I_1	S_2 S_1 I_4 I_3	Results after XAS at IOS + 5
S_4 S_3 A_4 A_3	A_2 A_1 I_2 I_1	S_2 S_1 I_4 I_3	Results after XAS at IOS + 6
S_4 S_3 S_2 A_4	A_3 A_2 A_1 I_1	S_1 I_4 I_3 I_2	Results after XAS at IOS + 7
S_4 S_3 S_2 S_1	A_4 A_3 A_2 A_1	I_4 I_3 I_2 I_1	Results after XAS at IOS + 9

3.8.7 INPUT/OUTPUT EXAMPLES

In the following paragraphs a number of examples will be shown using different capabilities of the input/output system to perform the same basic function. In these examples negative logic is assumed where a -V signal relative to VSS will be interpreted as a binary one. If positive logic is to be used, minor adjustments in the program will also work. The basic problem is to detect and react to three signal inputs A, B, and C. A decision is to be made if all three signals are in the true (-V) state.

3.8.7.1 Detect and React to Three True Signals (A, B and C)

Using I1SK in this example, if signals A, B and C are true, perform function: FUNC. Otherwise perform function: NOTT. Signal A is connected to the most significant bit of Channel 1 (PI4), B is connected to the next most significant bit (PI3) and C is connected to the third (PI2). The least significant bit (PI1) is tied to VSS.

PROGRAM:

```
            I1          INITIAL CONDITION
            AISK   2    CARRY OVERFLOW ONLY WHEN INPUT = 111X
            T(L)   FUNC OVERFLOW — ALL TRUE
NOTT        [ ]         NO OVERFLOW — NOT ALL TRUE
```

This is a good example of the efficiency of multifunction instructions designed to facilitate particular operations since this program is much shorter than the following examples using techniques other than the AISK multifunction.

3.8.7.2 Same Problem Using IAM

In this case the signals A, B and C are tied to RIO8, RIO7, and RIO6 respectively. The least significant bit from RIO5 will be masked out by the X Buffer bit 1.

PROGRAM:

```
        LAI  #E         FORM MASK
        OA              OUTPUT MASK
        NOP             DELAY
        IAM             INPUT SIGNALS TO ACCUMULATOR
        AISK 2          WOULD OVERFLOW ONLY IF A, B, C TRUE
        T(L)  FUNC      TRUE SO PERFORM FUNC
NOTT    [ ]             NOT TRUE
```

The T(L) FUNC instruction format used here and in the case above indicates that the programmer may use either a T or TL instruction as appropriate for the actual location of FUNC. This is a descriptive technique only. The assembler will accept only one of these unless the B form may be used.

3.8.7.3 Same Problem Using Discrete Inputs (Inputs Already Floating)

In this case, the A, B, and C signals are tied to descrete inputs DI/O6, DI/O7, and DI/O8 respectively.

PROGRAM:

```
        LBL     #36     POINT TO A INPUT
        INCB            ! START POINTING TO B INPUT
        SKISL           STILL POINTING TO A
        T       NOTT    A HIGH SO EXIT
        INCB            ! START POINTING TO C INPUT
        SKISL           TEST B
        T       NOTT    B HIGH SO EXIT
        SKISL           TEST C
NOTT    T(L)    NOTTZ   C HIGH SO EXIT
FUNC    [ ]             A, B, AND C TRUE
```

The routine as written has assumed that the outputs from DI/O6, 7 and 8 have been set to the float state prior to the execution of this program. If not, then the following approach may be used.

3.8.7.4 Same Problem Using Discrete Inputs While Simultaneously Floating DI/O Channels

If the DI/O6, 7 and 8 flip-flops have not previously been put into a float state, it may be done at the same time as executing the inputs.

PROGRAM:

```
        LBL     #36     POINT TO A
        NOP
        ROS             FLOAT A CHANNEL
        INCB            ! START POINTING TO B
        SKISL           TEST A
        T       NOTT    A HIGH SO EXIT
        ROS             FLOAT B CHANNEL
        INCB            ! START POINTING TO C
        SKISL           TEST B
        T       NOTT    B HIGH SO EXIT
```

```
         ROS                  FLOAT C CHANNEL
         SKISL                TEST C
NOTT     T(L)      NOTTZ      C HIGH SO EXIT
FUNC     [ ]                  A, B, AND C TRUE
```

3.8.7.5 Same Problem Using Discrete Inputs With Inverted Signals (Inputs Already Floating)

In this case A is tied to DI/O6, B to DI/O7 and C to DI/O8. A, B and C are at VSS potential when A, B, and C are true.

PROGRAM:

```
         LBL      #36         POINT TO INPUT A
         INCB                 ! START POINTING TO B
         SKISL                STILL POINTING TO A
         T        TRYB
         T        NOTT
TRYB     INCB                 ! POINTING TO B START C
         SKISL                STILL POINTING TO B
         T        TRYC
         T        NOTT
TRYC     SKISL                POINTING TO C
         T(L)     FUNC        ALL TRUE, GO TO FUNC
NOTT     [ ]                  NOT TRUE
```

3.8.7.6 Same Problem Using I2C Instruction

In this case, the A, B, and C inputs are tied to input pins PI8, PI7, and PI6. Pin PI5 could be used for another input for some other purpose if desired.

PROGRAM:

```
         I2C                  INPUT A B C
         AISK     14          IF INPUT IS 000X, SKIP
         T        NOTT        OVERFLOW SO INPUT NOT 000X
         T(L)     FUNC        A B C TRUE, GO TO FUNC
NOTT     [          ]         NOT TRUE
```

3.8.7.7 Same Problem Using OA and IAM Instructions. Also Test Fourth Bit Separately

In this situation the signals A, B, and C are tied to RIO8, RIO7, and RIO6 respectively. The least significant bit (D) will have other information which is evaluated after storing the input in memroy and performing FUNC as a subroutine if it is required. It is assumed for this example that the A Buffer is continually set to 1's as this I/O path is to be used for inputs only.

PROGRAM:

```
         LBL      BIT         POINT TO MEMORY CELL FOR LSB
         LAI      #F          READY OUTPUT BUFFER TERM
         OA                   LOAD BUFFER, INPUT A B C AND D
         NOP
         IAM
         AISK     2
         TM(L)    FUNC        CALL FUNC SUBROUTINE
EVLD     X        0           SEND TO MEMORY FOR LSB TEST
         SKBF     1           EVALUATE D
         T(L)     DIS1        D IS EQUAL TO 1, GO TO DIS1
DIS0     [          ]         D IS EQUAL TO 0
```

3.8.7.8 Same Problem Using OA and IAM Separately Testing Each Bit

This is very similar to the above example except that instead of using the AISK instruction to test bits A, B, and C, the SKBF instruction is used on complemented inputs.

PROGRAM:

```
        LBL     BIT     POINT TO MEMORY CELL FOR INPUT
        LAI     #F      READY OUTPUT BUFFER TERM
        OA              INPUT A, B, C AND D, AND RESTORE BUFFER
        NOP
        IAM
        COM             COMPLEMENT FOR SHORTER PROGRAM
        X       0       SAVE IN MEMORY
        SKBF    4       TEST Ā
        T       NOTT    Ā TRUE SO EXIT
        SKBF    3       TEST B̄
        T       NOTT    B̄ TRUE SO EXIT
        SKBF    2       TEST C̄
        T       NOTT    C̄ TRUE SO EXIT
        TM(L)   FUNC    A, B AND C TRUE SO GO TO SUBROUTINE
NOTT    LBL     BIT     FUNC MAY HAVE MODIFIED B REG, SO POINT TO BIT
        SKBF    1
        T(L)    DIS0    ORIGINAL D WAS ZERO (COMPLEMENT IS ONE)
DIS1    [       ]       ORIGINAL D WAS ONE (COMPLEMENT IS ZERO)
```

3.9 SOME USEFUL PROGRAMMING EXAMPLES

This section contains a number of useful routines which utilize a broad spectrum of the PPS-4/1 MM76 instructions and can be used or adapted for applications requiring such capabilities as keyboard displays, 12-hour clocks with hours, minutes and seconds being calculated, logical operations where both operators are obtained as a result of data inputs or data processing, and multiply and divide examples.

3.9.1 KEYBOARD/DISPLAY EXAMPLE

There are many possible keyboard and display configurations which can be mechanized. This example (Figure 3-19) shows an eight digit display and a 64-key keyboard but up to 20 display digits and 80 keys can be easily accommodated. Smaller keyboard and/or display combinations will permit more of the I/O to be used for other required functions.

Keyboard strobing and display refresh are performed simultaneously in an endless loop which is interrupted only by the depression of a key (or other control input). When a key is depressed, a key decoding routine is entered that determines which key is depressed and branches to the processing routine for that key.

Keyboard strobing and display refreshing routines are usually suspended during key processing in the interests of coding simplicity, although this is not a requirement. The resulting display "blink" is often desirable as visual feedback to the operator that the key has been recognized. If lengthy processing routines are required, such as printer control functions, it may be necessary to code the Keyboard/Display routine as a subroutine which can be

called many times during a process. In this case, "key stacking" would be required where input keys would be saved in a RAM queue for later processing. Programs of this type can become quite complex, with ROM and RAM requirements considerably greater than the more straightforward routine described here.

The procedure for the keyboard/display routine is outlined in the Figure 3-20 flow chart and is described below:

1. The keyboard flags are initialized
2. The strobe value is initialized
3. The selected strobe is activated
4. Segment data for this display position is output
5. Keyboard returns are sampled for a key depression
6. After an appropriate delay, the segment data is blanked
7. The strobe is deactivated
8. The strobe value is decremented and execution returns to step 2 until the last strobe is output
9. Following the last strobe, return to step 1 and start over
10. If key return is present at step 5, adjust flag
11. If new key, set key down flag
12. Turn off display
13. Debounce key
14. Decode key
15. Process key function and return to step 1

Several clarifying comments are required. Note that the strobes are shared between the keyboard and the display. The display buffer in RAM must be arranged so that the BL values for each display digit correspond to the BL values for the related strobe. In the example, the least significant (rightmost) digit of the display is connected to strobe 7 (DI/O7), and must therefore occupy a RAM location with BL = 7. (See Figure 3-19.)

If a key depression is detected at step 5, the process is suspended until the key processing routine is completed. Note that the key processing routine may take only a few milliseconds to complete, in which case the key will probably still be depressed when the keyboard routine is re-entered. The "KD Flag" (key down flag) prevents reprocessing the same key.

The "KBC Flag" (keyboard clear) permits normal display refresh during key-down periods. The delay in step 6 is optional and is used to adjust display brightness. The brightness of the display (particularly in the case of LED's) is largely determined by the segment duty cycle, which is defined as the ratio of segment-on time to segment-off time. Note that the segment-on time is the period when both the selected strobe driver and segment drivers for the selected digit are activated. In this case, on time is from step 4 until the completion of step 6. The delay in step 6 lengthens the on-time period, thus increasing the segment duty cycle and brightness. Table 3-17 shows the effect on timing by various values of N and M.

It might seem redundant to turn off both the strobe and segments at the completion of each digit refresh, but this is required for certain types of gas discharge and fluorescent displays. A quiescent period is required between digits when both the strobe and segments are turned off to prevent "ghosting", a phenomenon where off segments can be seen to be slightly on. Severe ghosting can make the display unreadable. LED displays are, in general, not subject to ghosting, allowing some minor simplifications of the refresh routine when used.

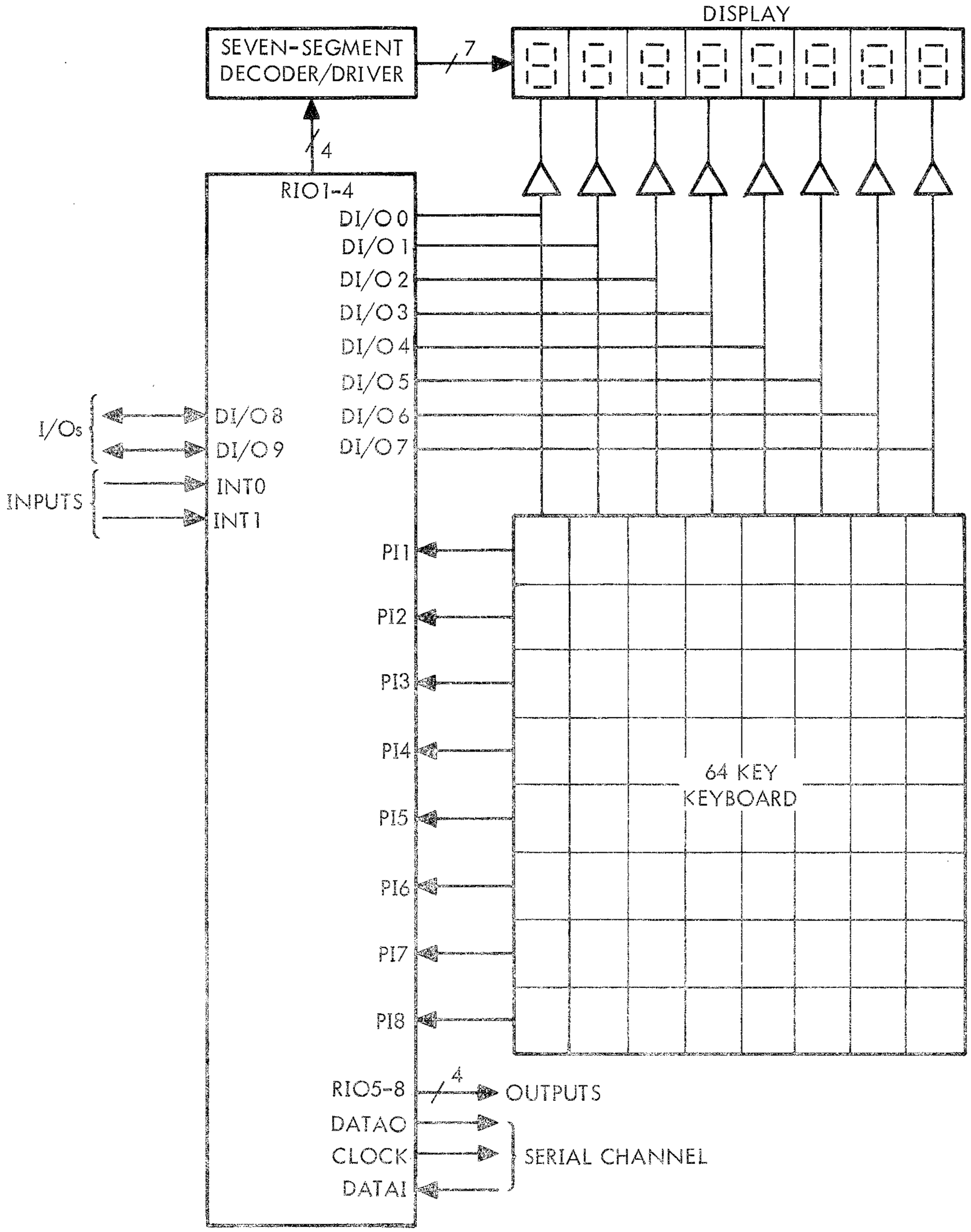


Figure 3-19. KEYBOARD DISPLAY EXAMPLE

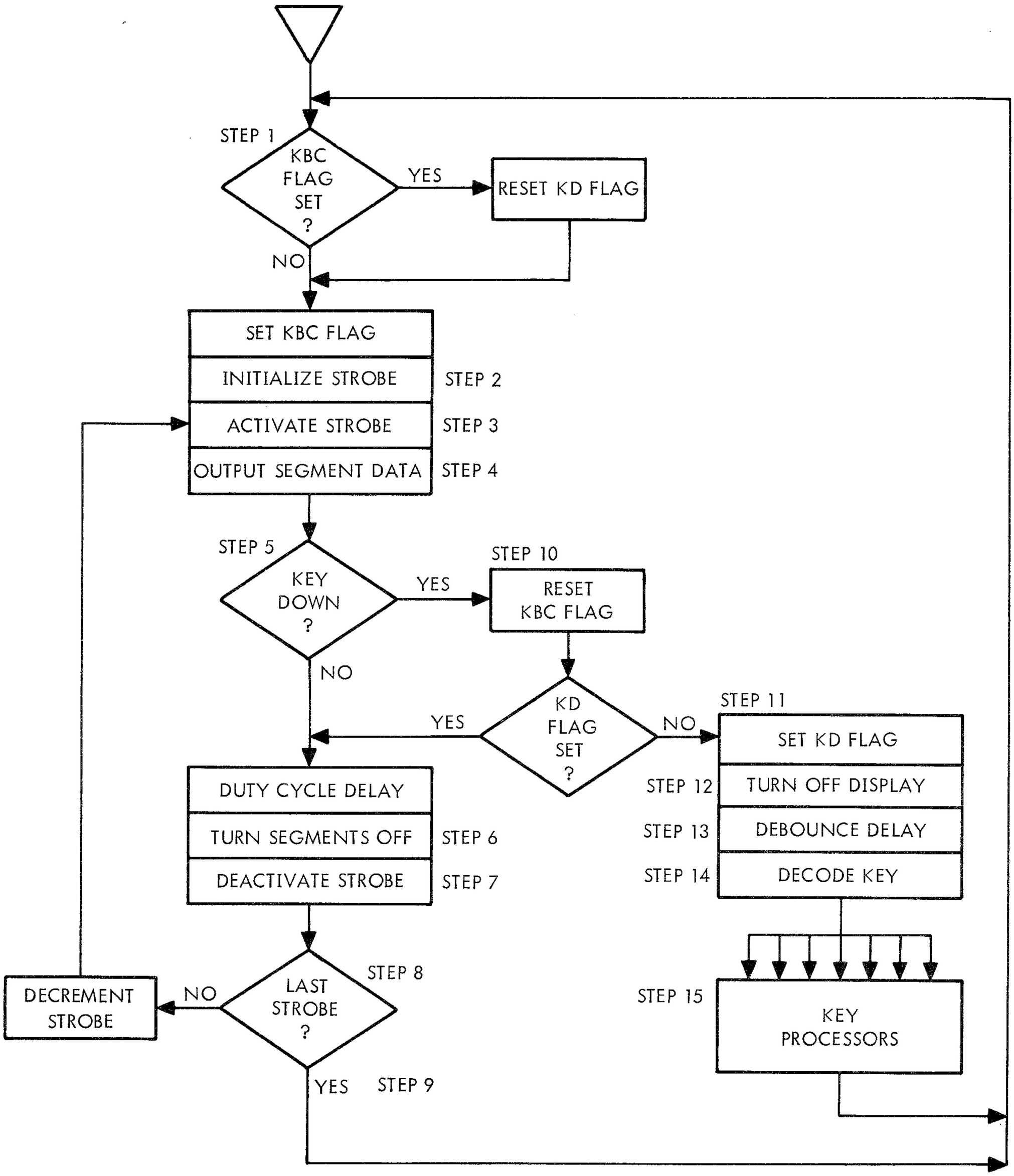


Figure 3-20. KEYBOARD/DISPLAY FLOW DIAGRAM

Table 3-18. KEYBOARD SCAN VS. DISPLAY DUTY CYCLE

N	M	ON Cycles	OFF Cycles	TOT/DIG Cycles	TOT Cycles	KB Scan Time (MS) 40 kHz	KB Scan Time (MS) 120 kHz	Duty Cycle DIG. ON/TOT
15	13	59	8	67	542	13.55	4.52	10.88%
14	12	56	8	64	518	12.95	4.32	10.81%
13	11	53	8	61	494	12.35	4.12	10.73%
12	10	50	8	58	470	11.75	3.92	10.64%
11	9	47	8	55	446	11.15	3.72	10.54%
10	8	44	8	52	422	10.55	3.52	10.43%
9	7	41	8	49	398	9.95	3.32	10.30%
8	6	38	8	46	374	9.35	3.12	10.16%
7	5	35	8	43	350	8.75	2.92	10.00%
6	4	32	8	40	326	8.15	2.72	9.82%
5	3	29	8	37	302	7.55	2.52	9.60%
4	2	26	8	34	278	6.95	2.32	9.35%
3	1	23	8	21	254	6.35	2.12	9.06%
2	0	20	8	28	230	5.75	1.92	8.70%
1	Bypass*	16	8	24	198	4.95	1.65	8.09%
NO DELAY**		10	8	18	150	3.75	1.25	6.67%

*DELETE LAI M — DISPLAY BRIGHTNESS WILL CHANGE WHEN SOME KEYS ARE DOWN

**DELETE LAI N
LAI M
AISK 15
T *-1

The debounce delay preceding key processing is required to prevent switch bounce on the keyboard switches from being interpreted as multiple key depressions. The length of the delay depends on the bounce characteristics of the keyboard selected, and is generally in the range of 5-10 milliseconds.

A final comment on the flowchart of Figure 3-20. This routine mechanizes a "two-key lockout" type of keyboard entry which is common in calculators. Two-key lockout means that once a key is depressed, a second key will not be recognized (is "locked out") until the first key is released and a clear keyboard is detected for at least one complete cycle of the keyboard/display loop. Some applications where rapid data entry is desirable require a "rollover" type keyboard routine. This can also be mechanized with a slight increase in complexity of the keyboard flag logic.

KEYBOARD/DISPLAY PROGRAM:

```
KBD     LB      FLAG    STEP 1
        SKBF    1       KBC FLAG SET?
        SB      2       YES, RESET KD FLAG
        SB      1       SET KBC FLAG
        LBL     #37     INITIALIZE STROBE STEP 2
        NOP
KBD1    SOS             OUTPUT SELECTED STROBE STEP 3
        L       0       GET DISPLAY DATA FROM RAM
        OA              OUTPUT SEGMENT DATA STEP 4
        I1
        AISK    15      KEY DOWN ON CH1? STEP 5
        T       KBD2    YES, GO TO KBD2
        T       KBD3    NO, TEST REST OF KEYS
KBD2    XAB             SAVE BL (STROBE) IN A
        LB      FLAG
        RB      1       RESET KBC FLAG
        SKBF    2       KD FLAG SET?
        T       NEWK    NO, NEW KEY DETECTED
        LBA             YES, KEY ALREADY PROCESSED, RELOAD BL
        T       KBD4
KBD3    I2C
        COMP
        AISK    15      KEY DOWN ON CH2? STEP 5 (CONT)
        T       KBD2    YES
        LAI     N       NO
KBD4    LAI     M       ADJUST M AND N FOR PROPER DISPLAY BRIGHTNESS
        AISK    15      GENERATE
        T       *-1     DELAY
        LAI     15
        OA              TURN SEGMENTS OFF STEP 6
        ROS             DEACTIVATE STROBE STEP 7
        L
        XDSK            DECREMENTED STROBE, LAST? STEP 8
        T       KBD1    NO, DO NEXT
        T       KBD     LAST, START OVER AT STEP 1
NEWK    RB      2       SET KEY DOWN FLAG STEP 11
        LBA             RELOAD BL (STROBE)
        LAI     15      SEGMENTS STEP 12
        OA              OFF
        ROS             STROBE OFF
        XAB             SAVE STROBE IN A
        LB      STR     SAVE FOR KEY DECODE IF NEEDED
        X       0       SAVE STROBE VALUE IN RAM
        LAI     15
        LXA             STEP 13
        XAX
        AISK    15      DEBOUNCE DELAY
        T       *-1
        XAX
        AISK    15
        T       *-5
        TL      DECODE  DECODE AND PROCESS KEY STEP 14
```

*Strobe value is in A, strobe and segments are still activated.

3.9.2 TWELVE HOUR CLOCK SUBROUTINE

This subroutine increments an 8-digit RAM register containing four digit pairs corresponding to Hours, Minutes, Seconds and 1/60 second fractions. The Minutes, Seconds and fractions are incremented on a modulo-60 basis, while the Hours counter is modulo 12 for a normal 12 hour clock. With some additional code, the Hours counter can be programmed to operate also in modulo 24 for 24 hour clock applications.

The program is coded as a subroutine which would be entered every 1/60 second upon detection of a transition of a 60 Hz square wave derived from the input power line frequency and connected to one of the discrete inputs (INT0 or INT1). The input would be interrogated using the edge-detection logic and associated instructions. The sub-routine is terminated by an RT instruction located at one of three possible exits from the subroutine.

The flowchart for the program is straightforward. Note that all four digit pairs are incremented modulo 60 and then the hours are corrected only if required.

DIGITAL CLOCK ROUTINE

BL =	0	1	2	3	4	5	6	7
	H	R	M	IN	S	EC	1/	60 FR

Clock Update Routine

```
CKUP    LBL     FR          POINT B AT LEAST SIGNIFICANT DIGIT (BL = 7)
        SC
CK1     LAI     6           INCREMENT EACH
        ACSK                DIGIT
        T       *+2
        AISK    10          PAIR
        XDSK                MODULO
        LAI     10          60
        ACSK
        T       *+3
        AISK    6
        NOP
        XDSK
        T       CK1
        LBL     H           CORRECT HOURS (BL = 0)
        L                   VALUE IF
        XNSK                REQUIRED
        AISK    15          HR MSD = 0?
        T       *+2         NO, 1 OR GREATER
        RT                  YES
        L
        AISK    13          HR LSD > 2?
        AISK    1           YES, SET VALUE TO 1, SKIP
        RT                  NO
        XDSK
        LAI     0
        X
        RT
```

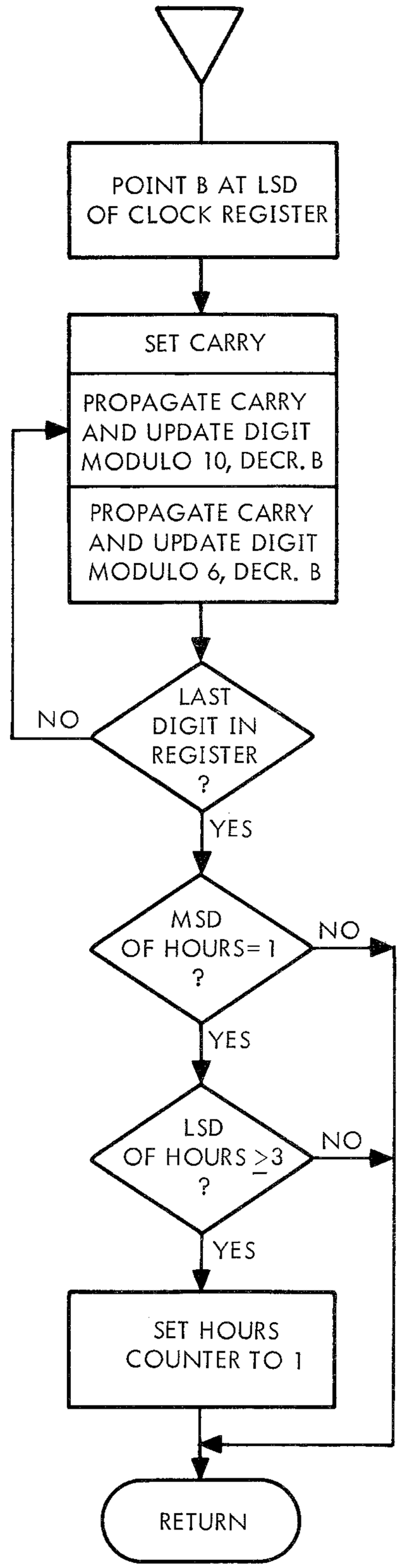


Figure 3-21. CLOCK ROUTINE FLOW DIAGRAM

3.9.3 GENERAL LOGIC SUBROUTINES

This section describes a general AND routine, general OR routine, and general Exclusive OR routine where both values involved in the logical operation are variables. All three routines make use of a subroutine termed CYCL which shifts both the Accumulator and the addressed memory cell left one position as shown in Table 3-19.

The CYCL subroutine is as follows:

```
CYCL    L                   !LOAD N VALUE FROM MEMORY
        AC                  DOUBLE IT
        X       (R)         STORE AND POINT TO M VALUE
        L                   !LOAD M VALUE
        A                   DOUBLE IT WITH NO CARRY
        X                   !AND RESTORE TO M
        XAS                 COUNTER VALUE FROM S TO A
        AISK    15          COUNT DOWN AND TEST
        RT                  NOT THRU — REPEAT
        RTSK                THRU SO EXIT TO TERMINATE STATEMENT
```

The two values to be operated on (N and M) are in corresponding column positions where (R) can alternate between them for X (R) and EOB (R) instruction.

The AND routine then is as follows:

```
LAND    LBL     M           POINT TO ONE VARIABLE
        RC                  START WITH CARRY FLIP-FLOP = 0
        LAI     4           LOAD COUNTER VALUE
LOOP    XAS                 SAVE COUNT
        SKBF    4           TEST BIT 4 OF M
        T       PASS        BIT IS TRUE SO LEAVE N UNCHANGED
        EOB     (R)         POINT TO N
        RB      4           BIT 4 = 0 (ANYTHING ANDED WITH 0 = 0)
NEXT    TM      CYCL        GET NEXT BIT ALIGNED
        T       LOOP        REPEAT
        T       EAND        COMPLETE SO EXIT
PASS    EOB     (R)
        T       NEXT
EAND    :                   END OF AND ROUTINE
        .
```

The OR routine is similar and is as follows:

```
LOR     LBL     M
        LAI     4           SAME AS AND
ORLP    XAS
        SKBF    4           SEQUENCE
        T       SET1
        EOB     (R)
SHFT    TM      CYCL
        T       ORLP
        T       THRU
SET1    EOB     (R)         BIT IS TRUE SO SET
        SB      4           N = 1
        T       SHFT
THRU    :                   END OF OR ROUTINE
        .
```

The Exclusive OR subroutine is shown below:

```
LEOR    LBL     M
        LAI     4
ADVN    XAS
        SKBF    4
        T       TRUB        MBIT = 1
        EOB     (R)         MBIT = 0 POINT TO N
        SKBF    4           TEST N BIT
        T       SET         MBIT = 0; NBIT = 1; F(MN) = 1
ZERO    RB      4           MBIT = 0; NBIT = 0; F(MN) = 0
        TM      CYCL
        T       ADVN
        T       EEOR
TRUB    EOB     (R)         MBIT = 1, POINT TO N
        SKBF    4           TEST NBIT
        T       ZERO        MBIT = 1; NBIT = 1; F(MN) = 0
SET     SB      4           MBIT = NBIT; F(MN) = 1
        T       ZERO+1
EEOR    :                   END OF EXCLUSIVE OR ROUTINE
        :
        .
```

Table 3-19. CYCLE OPERATION

		Memory Cell M					Memory Cell N			
	Count	4	3	2	1	C	4	3	2	1
Start	4	M_4	M_3	M_2	M_1	0	N_4	N_3	N_2	N_1
End of 1st Pass	4	M_3	M_2	M_1	–	$Y_4 = f(N_4M_4)$	N_3	N_2	N_1	0
2nd Pass	3	M_2	M_1	–	–	$Y_3 = f(N_3M_3)$	N_2	N_1	0	Y_4
3rd Pass	2	M_1	–	–	–	$Y_2 = f(N_2M_2)$	N_1	0	Y_4	Y_3
4th Pass	1	–	–	–	–	$Y_1 = f(N_1M_1)$	0	Y_4	Y_3	Y_2
5th Pass	0	–	–	–	–	0	Y_4	Y_3	Y_2	Y_1

3.9.4 TIME DELAYS

There are many ways of performing time delays in the PPS-4/1 system.

One technique which is useful when a delay is required, but the period of the delay is not important, is to make use of subroutines which are used in combination so that the net effect is only a time delay. For instance, if the routine has both a Character Right Shift (RSFT) and a Character Left Shift routine (LSFT), then the sequence is:

```
LBL     LSD     POINT TO LEAST SIGNIFICANT DIGIT
TM      LSFT    LEFT SHIFT ROW
LBL     MSD     POINT TO MOST SIGNIFICANT DIGIT
TM      RSFT    RIGHT SHIFT ROW
```

This sequence delays while the contents of one row are all shifted left one character and then shifted back. This technique assumes that the shift routines do not destroy any data and leave the Accumulator loaded with the value which has been shifted out. This technique uses a total of 108 cycles.

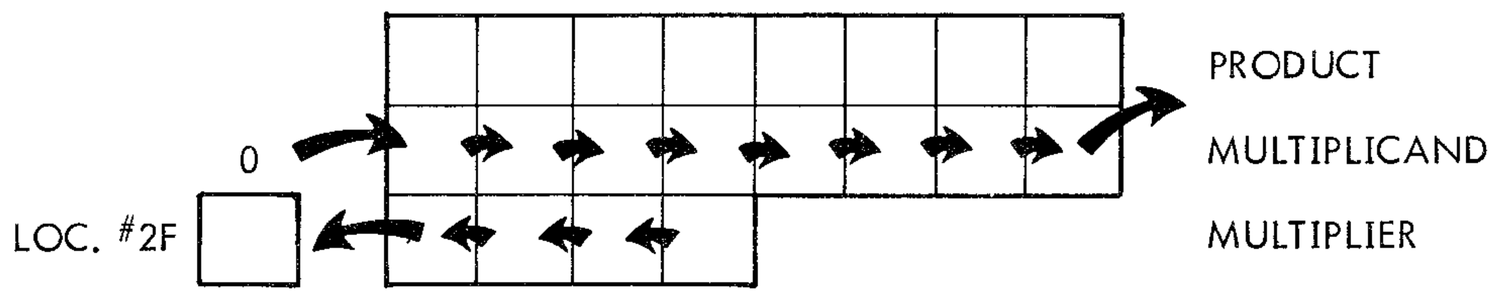
When more precise timing is required a constant value may be loaded into several words in a register. A loop which counts one for each iteration can be adjusted by choosing the length of the register and the initial constant to get a more precise value. When the counter overflow (carry after the most significant digit is processed) is one the time has elapsed. Finer resolution down to one cycle precision may be accomplished by selecting the counter conditions to be just short of the desired time and then finishing the timing by a series of NOP's with a final output at exactly the desired time.

3.9.5 MULTIPLY

In the PPS-4/1 System, multiply is accomplished by repeatedly adding the multiplicand under the control of the multiplier. It is also necessary to appropriately shift the multiplicand and to appropriate shift the multiplier register so that each successive multiplier digit is used to control the add operation. A counter which is initially set to 16 minus the number of digits to be used as a multiplier must also be counted up to overflow so that the process may be terminated.

The following routine multiplies a 4-digit multiplicand in RAM locations #10-#13 by a 4-digit multiplier in RAM locations #20-#23. The 8-digit product is generated in RAM locations #0 through #7. Circled values in the program identify addresses or counters which would be changed for different numbers of digits.

The process consists of successive addition of the multiplicand to the product register (which is initialized to zero). The addition loop is controlled by the multiplier digit which is shifted out of the multiplier register. After each addition loop, the multiplicand is shifted right one digit. The process is repeated four times, once for each multiplier digit. The digit counter is retained in the S Register and the multiplier digit is held in LOC. #2F during the addition loop.

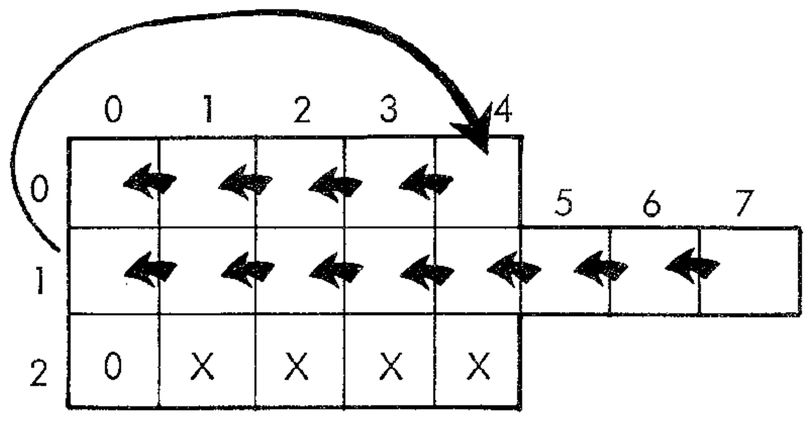


```
MULT    LB      (7)
        LAI     0       ZERO
        XDSK    0       PRODUCT
        T       *-2     REGISTER
        LAI     (12)    INITIALIZE DIGIT COUNTER (16-4)
M1      XAS             SAVE DIGIT COUNTER IN S
        LBL     #10
        LAI     0
        XNSK    0       RIGHT
        SKBEI   (8)     SHIFT
        T       *-3     MULTIPLICAND
        LBL     #(23)   LEFT
        XDSK    0       SHIFT MULTIPLIER
        T       *-1     REGISTER, GET MSD
M2      AISK    15      MULTIPLIER DIGIT ZERO?
        T       M3      YES
        X       0       NO, SAVE MULTIPLIER DIGIT
        XAS             YES, DIGIT IS ZERO
        AISK    1       BUMP DIGIT COUNTER
        RT              DONE
        T       M1      DO NEXT DIGIT
M3      X       0       SAVE MULTIPLIER DIGIT IN #2F
        LBL     #(17)
        RC
M4      L       1       ADD
        AISK    6       MULTIPLICAND
        NOP
        ACSK            TO
        AISK    10      PRODUCT
        T       *+2
        XDSK    1
        T       M4
        EOB     3
        L       0       GET MULTIPLIER DIGIT
        T       M2
```

3.9.6 DIVIDE

This routine divides an 8-digit dividend by a 4-digit divisor and generates an 8-digit quotient. The quotient is generated in the dividend register, so the dividend value is lost as a result of the divide operation. The divisor remains unchanged.

The process consists of successive subtraction of the divisor from a portion of the dividend which has been shifted into a working register. When underflow occurs, indicating that the divident portion is negative, the divisor is added back in to restore the interim remainder value. The counter for the subtraction loop becomes a digit of the quotient and is shifted into the dividend register as each dividend digit is shifted out into the working register. The digit counter is retained in the S Register during the subtraction — addition loop and is incremented after each loop is completed. Circled values are those that would change in order to increase or decrease register sizes.




```
DIVD    LB      (4)     ZERO
        LAI     0       WORKING
        XDSK            REGISTER
        T       *-2
        LAI     (8)     SET DIGIT COUNTER (N̄)
D1      LSA             STORE DIGIT COUNTER IN S REGISTER
        LBL     (#17)   LEFT
        LAI     0       SHIFT
        XDSK            DIVIDEND
        T       *-1
        LB      (4)     PUT MSD
        XDSK            OF DIVIDEND
        T       *-1     IN WORKING REGISTER
D2      LBL     (#24)   SUBTRACT
        SC              DIVISOR
        L       2       FROM
        COM             WORKING
        ACSK            REGISTER
        T       *+2
        AISK    10
        XDSK    2
        T       *-6
        SKNC            RESULT NEGATIVE?
        T       D3      NO
        LBL     (#24)   YES,
        RC              RESTORE
        L       2       REMAINDER
        AISK    6       IN
        NOP             WORKING
        ACSK            REGISTER
        T       *+2
        AISK    10
        XDSK    2
        T       *-7
        XAS             GET DIGIT COUNTER
        AISK    1       INCREMENT. LAST?
        RT              YES, DONE
        T       D1      NO, DO NEXT
D3      LBL     (#17)
        LAI     1       INCREMENT
        A               QUOTIENT
        X               DIGIT
        T       D2
```

3.9.7 COMBINATION SUBROUTINES

Many times portions of subroutines which accomplish basically different functions may be combined to reduce program memory requirements and still provide a high degree of utility. For example, in many programs it is necessary to increment one or more registers, decrement others, add values to others, to load a value into memory and clear the Accumulator to zero.

The following primitive subroutine with multiple entry points accomplishes all of these things.

```
R1MI    LB    R1    DECREMENT R1 ENTRY
R2MI    LBL   R2    DECREMENT R2 ENTRY
RAMI    LBL   RA    DECREMENT RA ENTRY
MIN1    LAI   15    DECREMENT MEMROY ENTRY
PLS1    LAI   1     INCREMENT MEMORY ENTRY
ADDM    A           ADD TO MEMORY ENTRY
XCLR    X           STORE AND CLEAR ENTRY
        LAI   0     CLEAR ACCUMULATOR
        RT          RETURN
R1PL    LB    R1    INCREMENT R1 ENTRY
R2PL    LBL   R2    INCREMENT R2 ENTRY
RBPL    LBL   RB    INCREMENT RB ENTRY
RCPL    LBL   RC    INCREMENT RC ENTRY
        T     PLS1  TRANSFER TO INCREMENT MEMORY ENTRY
```

To use this subroutine to decrement the one-word register termed R2 it is only necessary to code TM R2MI. The primitive subroutine will point to R2, Load 15 ($^{\#}$F) into the Accumulator, add the addressed memory cell, store the result back in memory, clear the Accumulator to zero, and return to the calling routine. To increment the R1 Register a TM R1PL is used.

The routine may decrement and test the result in memory for zero by the following calling sequence.

```
TM      RAMI    DECREMENT RA AND RESTORE
SKMEA           COMPARE 0 IN A WITH RA
T       NOTZ
[ ]
```

The subroutine may be used to add a larger increment (3 for instance) to a one-word register, RD, by the following sequence:

```
LBL     RD
LAI     3
TM      ADDM
```

The subroutine may also be used just to store a value and load zero into the Accumulator by executing a TM XCLR instruction.

3.9.8 STRAIGHT LINE PROGRAMMING FOR HIGHER SPEED

All of the examples used in this manual have made use of loops to accomplish repetitive processes so that the minimum program memory capacity would be used. However, if memory capacity is not as significant as speed of execution a straight line programming technique which does not make use of loops may be used.

For instance if a fast character left shift for four characters is desired, the following technique may be used:

```
LBL     RO      POINTER     MEMORY      ACCUMULATOR
X               RO          A           RO
XDSK            RO          RO          A
XDSK            R1          RO          R1
XDSK            R2          R1          R2
X               R3          R2          R3
[  ]            R4
```

This program shifts the Accumulator into the R0 memory location and moves the next three digits left one position. The value left in the Accumulator at the conclusion is the value shifted out of the fourth word (R3 contents). It requires a total of 6 cycle times for the shift while the conventional loop program:

```
LBL     RO
XDSK
T       *-1
```

requires a total of 11 cycle times.

APPENDIX A
ASCII CODE

BIT POSITION								HEXADECIMAL CODE	FIRST GROUP	SECOND GROUP	THIRD GROUP	FOURTH GROUP	FIFTH GROUP	SIXTH GROUP	SEVENTH GROUP	EIGHTH GROUP
								NOTES								
B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1									
				COL #				*	*	*	*	*	*	*	*	
				ROW #				0	0	0	0	1	1	1	1	
								0	0	1	1	0	0	1	1	
								0	1	0	1	0	1	0	1	
				COL HEX				0	1	2	3	4	5	6	7	
				ROW HEX												
				0	0	0	0	0	NUL	DLE	SP	0	@	P	\	p
				0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
				0	0	1	0	2	STX	DC2	"	2	B	R	b	r
				0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
				0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
				0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
				0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
				0	1	1	1	7	BEL	ETB	/	7	G	W	g	w
				1	0	0	0	8	BS	CAN	(	8	H	X	h	x
				1	0	0	1	9	HT	EM	)	9	I	Y	i	y
				1	0	1	0	A	LF	SUB	*	:	J	Z	j	z
				1	0	1	1	B	VT	ESC	+	;	K	[	k	┤[
				1	1	0	0	C	FF	FS	,	<	L	\	l	¦
				1	1	0	1	D	CR	GS	–	=	M	]	m	]├
				1	1	1	0	E	SO	RS	.	>	N	^	n	~
				1	1	1	1	F	SI	US	/	?	O	–	o	DEL

NUL – Null
SOH – Start of Heading
STX – Start of Text
ETX – End of Text
EOT – End of Transmission
ENQ – Enquiry
ACK – Acknowledge
BEL – Bell
BS – Backspace
HT – Horizontal Tabulation
LF – Line Feed
VT – Vertical Tabulation
FF – Form Feed
CR – Carriage Return
SO – Shift Out
SI – Shift In
DLE – Data Link Escape
DC – Device Control
NAK – Negative Acknowledge
SYN – Synchronous Idle
ETB – End of Transmission Block
CAN – Cancel
EM – End of Medium
SUB – Substitute
ESC – Escape
FS – File Separator
GS – Group Separator
RS – Record Separator
US – Unit Separator
SP – Space
DEL – Delete
* – Parity Bit: 1 = Odd, 0 = Even
MARK = 1
SPACE = 0

APPENDIX B
USEFUL TABLES FOR DEBUGGING

This appendix provides a number of tables which are useful in performing machine language modifications to a program for debugging purposes.

B.1 POLYNOMIAL COUNT SEQUENCE

Since the PPS-4/1 microcomputers all use a polynomial program counter, the execution sequence of the instructions is not in binary address sequence. Tables B-1 and B-2 have been provided to indicate the address of the next instruction to be executed and to specify the number of words remaining on the page.

Example Using Table B-1:

If the current address is #42F, the Table is entered at Row Y (corresponding to the 2 in the H_1 position) and column F (corresponding to F in the H_2 position). The table gives X7 (42). The X can signify an H_1 character of D or 9 or 5 or 1 depending upon the entry address. Since the entry address was 2, the value from the same H_1 column is 1, which is the value of X in X7. Thus the next instruction address is #417. (Note the first digit (H_0) is never modified by counting.) The number in ( ) indicates there are 42 more instruction bytes on the page. The next instruction after this would be at location #40B and (41) indicates there are 41 bytes remaining.

Table B-1. NEXT ADDRESS IN POLYNOMIAL COUNT SEQUENCE

Row	H_1 Next Least Significant Hex Digit	H_2 (Least Significant Hex Digit) 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Row
W	C-8-4-0	Y0* (63)	**	Y1 (58)	W1 (1)	W2 (59)	Y2 (39)	Y3 (53)	W3 (2)	W4 (60)	Y4 (31)	Y5 (48)	W5 (40)	W6 (54)	Y6 (18)	Y7 (34)	W7 (3)	W
X	D-9-5-1	W8 (61)	Y8 (51)	Y9 (29)	W9 (32)	WA (49)	YA (11)	YB (13)	WB (41)	WC (55)	YC (15)	YD (26)	WD (19)	WE (35)	YE (22)	YF (43)	WF (4)	X
Y	E-A-6-2	X0 (62)	Z0 (57)	Z1 (38)	X1 (52)	X2 (30)	Z2 (47)	Z3 (17)	X3 (33)	X4 (50)	Z4 (28)	Z5 (10)	X5 (12)	X6 (14)	Z6 (25)	Z7 (21)	X7 (42)	Y
Z	F-B-7-3	X8 (56)	Z8 (37)	Z9 (46)	X9 (16)	XA (27)	ZA (9)	ZB (24)	XB (20)	XC (36)	ZC (45)	ZD (8)	XD (23)	XE (44)	ZE (7)	ZF (6)	XF (5)	Z

Notes:

In the table:

Y0 thru ZF = Next value of H_1 H_2; H_0 is unchanged

( ) = Number of bytes remaining on page.

*First entry on page

**Last entry on page

$H_0H_1H_2$ = The hex characters representing the Program counter contents

W, X, Y and Z identify row — actual value will be in same column as original H_1 entry point.

B.2 TRANSFER INSTRUCTIONS

Tables B-2, B-3 and B-4 have been provided to simplify generating machine language codes for T, TM, TL, and TML instructions. The code for T instructions is found in Table B-2. If an on page transfer from location #13C to location #11A is desired, enter the table at Row D-9-5-1 since $H_1 = 1$ and read the value E5 in column A. The on-page transfer instruction code is E5. The TM format table, Table B-3, is used in a similar manner.

Table B-4 is used in conjunction with Tables B-2 and B-3 for forming the TL and TML instructions respectively.

If it is desired to transfer to instruction #230, the first byte of the instruction is determined from Table B-4 to be 37 since #230 is in the range #200-23F. The next byte is CF from Table B-2. Thus, the instruction is 37,CF which is a TL instruction. If it is desired to transfer to #19B, 39 is obtained from Table B-4 and E4 from Table B-2. Thus a transfer to #19B is a TL instruction with the code 39,E4.

Similarly, a subroutine call to location #052 would be written 3E,AD (a TML format) where the 3E byte information was obtained from Table B-4 and the AD from Table B-3.

Table B-2. TRANSFER (T) FORMAT FOR HEX LOCATIONS H_0 H_1 H_2

H_1 \ H_2	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C-8-4-0	FF	FE	FD	FC	FB	FA	F9	F8	F7	F6	F5	F4	F3	F3	F1	F0
D-9-5-1	EF	EE	ED	EC	EB	EA	E9	E8	E7	E6	E5	E4	E3	E2	E1	E0
E-A-6-2	DF	DE	DD	DC	DB	DA	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
F-B-7-3	CF	CE	CD	CC	CB	CA	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0

Table B-3. TRANSFER AND MARK (TM) FORMAT FOR HEX LOCATIONS H_0 H_1 H_2

H_1 \ H_2	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C-8-4-0	BF	BE	BD	BC	BB	BA	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
D-9-5-1	AF	AE	AD	AC	AB	AA	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
E-A-6-2	9F	9E	9D	9C	9B	9A	99	98	97	96	95	94	93	92	91	90
F-B-7-3	8F	8E	8D	8C	8B	8A	89	88	87	86	85	84	83	82	81	80

Table B-4. PAGE TRANSFER ADDRESSES

TML TL I_1	Page	Address Hex Range
3F	0	000-03F
3E	1	040-07F
3D	2	080-0BF
3C	3	0C0-0FF
3B	4	100-13F
3A	5	140-17F
39	6	180-1BF
38	7	1C0-1FF
37	8	200-23F
36	9	240-27F
35	10	280-2BF
34	11	2C0-2FF
33	12	300-33F
32	13	340-37F
	14	380-3BF
	15	3C0-3FF

I_2 for TL = $11\overline{XXXXXX}$
I_2 for TML = $10\overline{XXXXXX}$

An alternate method of determining the polynomial sequence and transfer codes is provided in Table B-5. Table B-5 combines some of the information from Tables B-2, B-3, and B-4.

There are two ways of entering Table B-5. One method starts by the programmer determining the hexadecimal equivalent of the least significant 6 bits of the last address used. The programmer then scans down the polynomial column until he finds the corresponding polynomial value and enters the table at that point. The successive 6-bit entries when combined with the 4-bit static portion of the Program Register give the successive addresses.

The second way of entering Table B-5 uses Table B-1 to avoid the initial separation of the bits in the address and the need to search Table B-5 for the corresponding hexadecimal codes representing the polynomial value. After locating the next byte address from Table B-1, the bytes remaining value as indicated in ( ) in Table B-1, can be used to enter Table B-5. Table B-5 then provides the least significant 6 bits of the Program Register in the polynomial count sequence. Successive values of these 6 bit may be determined by preceding directly down the column line by line. The transfer format for T, TL, TM, and TML instructions have been repeated for convenience.

As an example of the use of Table B-5 assume that the last location used on a page is #1AF. Refer to Table B-1 and locate the intersection of the H1 hexadecimal character "A" with the H2 hexadecimal character F. The table value at this point will be seen to be "X7 (42)". Using the X7 (42) data from Table B-1 the next address in polynomial sequence can be determined to be #197 (1 unchanged, X = 9, and 7 from table) and the remaining number of bytes is 42. Using 42 as an entry in Table B-5, shows the least significant 6 bits in the Program Register are #2F which confirms the least significant 6 bits of #1AF, the entry address. Moving down one line from the entry point in Table B-5 shows that the least significant 6 bits of the next location is 17 which agrees with the least significant 6 bits of #197 as obtained from Table B-1.

For the programmer to determine the least significant 8 bits of the address from the 6 bit polynomial it is necessary to append a 10 in binary, (or add 8 in hexadecimal) to the second digit (H1). For example, #2F becomes #AF, #17 becomes #97, etc. Therefore, the address of the next six locations are #18B, #185, #1A2, #1B1, #1B8, and #19C.

Table B-5. BRANCH CODES

Table B-5a. Next Address in Polynomial Count

Bytes Remaining	Polynomial (6 Bits)	T	TM	Bytes Remaining	Polynomial (6 Bits)	T	TM
63	00	FF	BF	31	09	F6	B6
62	20	DF	9F	30	24	DB	9B
61	10	EF	AF	29	12	ED	AD
60	08	F7	B7	28	29	D6	96
59	04	FB.	BB	27	34	CB	8B
58	02	FD	BD	26	1A	E5	A5
57	21	DE	9E	25	2D	D2	92
56	30	CF	8F	24	36	C9	89
55	18	E7	A7	23	3B	C4	84
54	0C	F3	B3	22	1D	E2	A2
53	06	F9	B9	21	2E	D1	91
52	23	DC	9C	20	37	C8	88
51	11	EE	AE	19	1B	E4	A4
50	28	D7	97	18	0D	F2	B2
49	14	EB	AB	17	26	D9	99
48	0A	F5	B5	16	33	CC	8C
47	25	DA	9A	15	19	E6	A6
46	32	CD	8D	14	2C	D3	93
45	39	C6	86	13	16	E9	A9
44	3C	C3	83	12	2B	D4	94
43	1E	E1	A1	11	15	EA	AA
42	2F	D0	90	10	2A	D5	95
41	17	E8	A8	9	35	CA	8A
40	0B	F4	B4	8	3A	C5	85
39	05	FA	BA	7	3D	C2	82
38	22	DD	9D	6	3E	C1	81
37	31	CE	8E	5	3F	C0	80
36	38	C7	87	4	1F	E0	A0
35	1C	E3	A3	3	0F	F0	B0
34	0E	F1	B1	2	07	F8	B8
33	27	D8	98	1	03	FC	BC
32	13	EC	AC	0	01	FE	BE

Table B-5b. Page Transfer Addresses TL and TML

Block Address Range$_{16}$		
00	00-3F	3F
01	40-7F	3E
02	80-BF	3D
03	C0-FF	3C
04	100-13F	3B
05	140-17F	3A
06	180-1BF	39
07	1C0-1FF	38
08	200-23F	37
09	240-27F	36
10	280-2BF	35
11	2C0-2FF	34
12	300-33F	33
13	340-37F	32
14	380-3BF	
15	3C0-3FF	

APPENDIX C
TABLES OF MICROCOMPUTER INSTRUCTIONS

Table C-1. A76XX MICROCOMPUTER INSTRUCTIONS

A76 Insruction	Hexadecimal Code			Length
	Word 1	Word 2	Word 3	
XAB	46			1
LBA	44			1
LB	20-2F			1
EOB	1C-1F			1
LBL	20-2F	1C-1F		2
INCB	58	54-57		2
DECB	58	5C-5F		2
SB	10-13			1
RB	14-17			1
SKBF	08-0B			1
XAS	4E			1
LSA	4C			1
L	50-53			1
X	58-5B			1
XDSK	5C-5F			1
XNSK	54-57			1
A	42			1
ASK	43			1
AC	40			1
ACSK	41			1
DC	66			1
COM	45			1
RC	0D			1
SC	0C			1
SKNC	01			1
LAI	70-7F			1
AISK	61-65, 67-6F			1
RT	02			1
RTSK	03			1
T (operand 7C0-7FF)	80-BF			1
T (other operand)	C0-FF			1
NOP	00			1
TL	30-3F	C0-FF		2
TM	80-BF			1
TML	30-3F	80-BF		2
SKMEA	47			1
SKBEI	30	20-2F		2
SKAEI	30	61-6F		2

Table C-1. A76XX MICROCOMPUTER INSTRUCTIONS (continued)

A76 Instruction	Hexadecimal Code			Length
	Word 1	Word 2	Word 3	
SOS	10			1
ROS	14			1
SKISL	08			1
OB	19			1
OA	18			1
IOS	4D			1
I1	4A			1
I2C	4B			1
INT1H	05			1
INT0L	04			1
DIN0	07			1
DIN1	06			1
IAM	1A			1
IBM	1B			1
SEG1	0E			1
SEG2	0F			1

Table C-2. A76XX MACRO INSTRUCTIONS

Macro Instruction	Macro Expansion			Length
	Instr 1	Instr 2	Instr 3	
TC L	SKNC	T L		2
TNC L	SKNC	T *+2	T L	3
TLC L	SKNC	TL L		3
TLNC L	SKNC	T *+3	TL L	4
TBT B, L	SKBF B	T L		2
TBF B, L	SKBF B	T *+2	T L	3
TLBT B, L	SKBF B	TL L		3
TLBF B, L	SKBF B	T *+3	TL L	4
TNE L	SKMEA	T L		2
TE L	SKMEA	T *+2	T L	3
TLNE L	SKMEA	TL L		3
TLE L	SKMEA	T *+3	TL L	4
TIH L	SKISL	T L		2
TIL L	SKISL	T *+2	T L	3
TLIH L	SKISL	TL L		3
TLIL L	SKISL	T *+3	TL L	4

Table C-3. MM76 OPERATION CODE SUMMARY

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	NOP	SKNC	RT	RTSK	INTOL	INT1H	DIN1	DIN0	SKBF 1 / SKISL	SKBF 2 / SKISL	SKBF 3 / SKISL	SKBF 4 / SKISL	SC	RC	SEG1	SEG2	0
1	SB1 / SOS	SB2 / SOS	SB3 / SOS	SB4 / SOS	RB1 / ROS	RB2 / ROS	RB3 / ROS	RB4 / ROS	ØA	ØB	IAM	IBM	EOB 0	EOB 1	EOB 2	EOB 3	1
2	LB 0	LB 1	LB 2	LB 3	LB 4	LB 5	LB 6	LB 7	LB 8	LB 9	LB A	LB B	LB C	LB D	LB E	LB F	2
3	TR F	TR E	TR D	TR C	TR B	TR A	TR 9	TR 8	TR 7	TR 6	TR 5	TR 4	TR 3	TR 2	TR 1	TR 0	3
4	AC	ACSK	A	ASK	LBA	COM.	XAB	SKMEA			I1	I2C	LSA	IOS	XAS		4
5	L 0	L 1	L 2	L 3	XNSK 0	XNSK 1	XNSK 2	XNSK 3	X 0	X 1	X 2	X 3	XDSK 0	XDSK 1	XDSK 2	XDSK 3	5
6	AISK 0	AISK 1	AISK 2	AISK 3	AISK 4	AISK 5	AISK 6	AISK 7	AISK 8	AISK 9	AISK A	AISK B	AISK C	AISK D	AISK E	AISK F	6
7	LAI 0	LAI 1	LAI 2	LAI 3	LAI 4	LAI 5	LAI 6	LAI 7	LAI 8	LAI 9	LAI A	LAI B	LAI C	LAI D	LAI E	LAI F	7
8	TM 3F	TM 3E	TM 3D	TM 3C	TM 3B	TM 3A	TM 39	TM 38	TM 37	TM 36	TM 35	TM 34	TM 33	TM 32	TM 31	TM 30	8
9	TM 2F	TM 2E	TM 2D	TM 2C	TM 2B	TM 2A	TM 29	TM 28	TM 27	TM 26	TM 25	TM 24	TM 23	TM 22	TM 21	TM 20	9
A	TM 1F	TM 1E	TM 1D	TM 1C	TM 1B	TM 1A	TM 19	TM 18	TM 17	TM 16	TM 15	TM 14	TM 13	TM 12	TM 11	TM 10	A
B	TM F	TM E	TM D	TM C	TM B	TM A	TM 9	TM 8	TM 7	TM 6	TM 5	TM 4	TM 3	TM 2	TM 1	TM 0	B
C	T 3F	T 3E	T 3D	T 3C	T 3B	T 3A	T 39	T 38	T 37	T 36	T 35	T 34	T 33	T 32	T 31	T 30	C
D	T 2F	T 2E	T 2D	T 2C	T 2B	T 2A	T 29	T 28	T 27	T 26	T 25	T 24	T 23	T 22	T 21	T 20	D
E	T 1F	T 1E	T 1D	T 1D	T 1B	T 1A	T 19	T 18	T 17	T 16	T 15	T 14	T 13	T 12	T 11	T 10	E
F	T F	T E	T D	T C	T B	T A	T 9	T 8	T 7	T 6	T 5	T 4	T 3	T 2	T 1	T 0	F

SKBEI N = TR FOLLOWED BY LB N (30, 20-2F)

SKAEI N = TR FOLLOWED BY AISK $\overline{N}$ (30, 60-6F)

LBL #NM = LBM FOLLOWED BY EOB N

INCB N = X FOLLOWED BY XNSK N

DECB N = X FOLLOWED BY XDSK N

APPENDIX D
ASSEMBLER OPERATING PROCEDURE FOR GENERAL ELECTRIC INFORMATION SERVICES

D.1 GENERAL

The General Electric information service provides an international communications network with direct timeshared access to a large scale computer system. Through the use of local terminals, the computer system can be accessed and the necessary programs executed in either the foreground or background mode.

The G.E. information service is the key to the PPS-4/1 Assembler program. By implementing a number of simple commands the user can directly activate the PPS-4/1 applications program. The GE information service has a wide range of applications and the features are defined in detail in the General Electric manuals.

> Command System, Reference Manual No. 3501.01A. (In particular, refer to Section 1, pages 35, 40, 42, 79, 83, 88, 91, 105, 111, 115, 118 and 129.)
>
> Editing Commands, Reference Manual No. 3400.01
>
> Foreground/Background Interface, Reference Manual No. 2000.01-2.

If additional information or manuals are required, contact the nearest Microelectric Device Division office, or General Electric Information Service Division Sales Office.

Some of the functions which are directly applicable to the PPS programs include:

- Entering and leaving the system.
- Creating, deleting, and renaming files.
- Setting file access controls.
- Sending and receiving mail or system messages.
- Determining terminal connect time, machine, number, etc.

In most cases, the user can access all programs by using a few simple commands. To abort an operation, the user types the BREAK key and (the) system aborts the operation in process and returns to the ready status.

The user may edit GE information service commands using the back space to delete single characters and Control X to delete the entire line.

D.2 SYMBOLIC NOTATION

The symbol used in the following examples to indicate user-typed carriage return is:

Control characters are denoted by a superscript c. For example, Xc denotes Control X. (Control X is used to delete the line being types and can be used any time prior to depressing the carriage return.) The method of typing a control character depends on the type of terminal being used. On most terminals control characters are entered by depressing the control key (CTRL) while typing the applicable character. For specific information on your terminal consult the manual for your terminal or contact the General Electric representative.

In most of the examples of terminal printouts used in this manual, data typed by the operator has been underlined to clearly differentiate it from information typed out by the system. In addition, underlining has been used to indicate where the operator should type in information. For example, the statement INPUT FILE NAME _____ indicates that the system caused INPUT FILE NAME to be typed out and that the operator should type in a file name in the area underlined. Remember, underlining is only used in the examples and does not appear during actual operation.

D.3 ENTERING THE GE SERVICE SYSTEM

Entering the GE information service requires the following three steps: (1) Calling the GE information service, (2) identifying the type of terminal, and (3) identifying the user. Each of the steps is explained in the following steps and examples.

Step 1. Call the GE information service by dialing the local GE information service telephone number and connecting the telephone to the terminal.

Step 2. Identify the type of terminal by typing one or more H characters immediately after carrier detect indicates connection has been made. The system will return the carriage and be ready to accept the user identification input.

Step 3. Identify the user as shown in the following examples. In the following examples assume that the user number is UBQ70009, the password is ABCD. In examples A and B assume that the account has been set up for no project identification.

The eight-character user number is assigned by General Electric; and each user is assigned a unique number. The user number identifies the following three items: (1) The first character (U in this example) identifies the GE system being used, (2) the next four characters (BQ70 in this example) identifies the user catalog, and (3) the last three characters (009 in this example) identifies a specific user.

Log in example A:

```
HHH
U#=UBQ70009,ABCD,SAMP
PPS PROBLEM REPORTING SYSTEM

ANY PPS PROBLEMS/SOLUTIONS TO REPORT? N
USED     .17 UNITS
```

See paragraph on PPS. PROBLEM REPORTING SYSTEM for details on how to use the reporting system.

Log in example B:

```
HHH

U#=UBQ70009,
PASSWORD
XEOWSBJKBA5
ID:SAMP
ABCD
```

System will overprint several sets of characters on this line and wait for user to overprint his password. This overprinting makes it impossible for unauthorized personnel to read the password during or after it has been typed.

The remainder of the printout same as in example A.

Now, assuming the account was set up for project identification log it would be accomplished as shown in examples C and D. Project identification is most often used for accumulating costs against a specific project.

Log in example C:

```
HHH↵
U#=UBQ70009,ABCD,SAMP↵
```

Project identification will accept any code including all blank spaces.

Same as in example A.

Log in example D:

```
HHH↵
U#=UBQ70009,↵
PASSWORD
[illegible]↵
ID:JOE↵
```

Same as in example A.

D.4 PPS PROBLEM REPORTING SYSTEM

The PPS Problem Reporting System provides an exchange system between users under catalog number UBQ70. The main purpose of this reporting system is to provide a fast, efficient method for users to relate problems to the personnel at Rockwell who are responsible for user support of the PPS programs. This system may be used to obtain assistance or to point out trouble areas.

The PPS problem reporting system is largely automatic and only requires operator responses and/or information entries. During the normal log-in sequence if there are no reports waiting for delivery, the system will print out a question asking if the user has any "problems/solutions" to report as shown in Figure D-1. If there are no reports waiting and the user wishes to transmit a report, the user types "Y" and proceeds as shown in Figure D-1. If the user has no reports to transmit, he types "N" and proceeds with his regular operation. If there are reports in the system, the system will first print out a statement indicating that a report is waiting for delivery as shown in Figure D-2. The question, NOW (Y OR N)?, asks do you want the report to be displayed at this time? If the user types in Y for yes, the report will be printed out as shown in Figure D-2.

The question, DESTROY (Y OR N)?, asks do you want the report retained in the sender's report bank or do you want it destroyed. Reports, once read, should be destroyed as soon as practical since each user's report bank can only hold a maximum of five reports.

After the report has been displayed, and saved or destroyed, the system will print out a question asking if the user has any problems/solutions to report.

```
U#=UBQ70002
PASSWORD
**********
ID:DAVE

PPS PROBLEM REPORTING SYSTEM

ANY PPS PROBLEMS/SOLUTIONS TO REPORT?Y

FOR WHOM ?004

IS REPORT TO BE ENTERED VIA "FILE" OR "TTY"?TTY

LINES ARE LIMITED TO 80 CHARACTERS.  ENTER ONE LINE AT A TIME
FINISH WITH "END*"
 ?THIS IS AN EXAMPLE OF THE USE OF THE PPS PROBLEM REPORTING
?SYSTEM.  I CAN TYPE AS MANY LINES AS REQUIRED TO EXPLAIN ANY
?PROBLEM I MAY BE HAVING WITH THE USE OF THE PPS PROGRAMS.
?PPS PROBLEMS SHOULD BE ADDRESSED TO USER 002, BUT A MESSAGE
?CAN BE SENT TO ANY USER IN THE BQ70 CATALOG.
?NEVER START TYPING ON A NEW LINE UNTIL THE ? IS PRINTED.
?END*

HOW MANY FILES DO YOU WISH TO HAVE INSPECTED?0

REPORT ENTERED-- ANOTHER REPORT?N
```

Identifies user number 2.

Log in sequence.

Printout of this sentence indicates that no reports are waiting for delivery, so you may proceed with any you have.

Typing Y indicates you have a report to transmit.

Report to user 4.

Figure D-1. EXAMPLE OF SENDING A REPORT

```
U#=UBQ70004
PASSWORD
**********
ID:COPE

PPS PROBLEM REPORTING SYSTEM

        1  MESSAGES WAITING FOR YOU

NOW (Y OR N)?Y
----------------------------------------

PPSIR FROM:      2    15:49PDT    06/17/76

THIS IS AN EXAMPLE OF THE USE OF THE PPS PROBLEM REPORTING
SYSTEM.  I CAN TYPE AS MANY LINES AS REQUIRED TO EXPLAIN ANY
PROBLEM I MAY BE HAVING WITH USE OF THE PPS PROGRAMS.
PPS PROBLEMS SHOULD BE ADDRESSED TO USER 002, BUT A MESSAGE
CAN BE SENT TO ANY USER IN THE BQ70 CATALOG.
NEVER START TYPING ON A LINE UNTIL THE ? IS PRINTED.
END*

----------------------------------------

DESTROY REPORT (Y OR N)?Y

ANY PPS PROBLEMS/SOLUTIONS TO REPORT?N

USED    .26 UNITS
```

Identifies user number 4.

Log in sequence.

Printout of this sentence indicates that one or more reports are waiting for delivery, and asks if you want the report.

Typing Y indicates you want the report now.

This line identifies who originated the report; in this example - user number 2.

Report from user 2 to user 4.

If an answer to the report is to be made, or if a new report is to be originated, you would type Y and proceed as shown in Figure D-1.

Figure D-2. EXAMPLE OF RECEIVING A REPORT

D.5 CREATING THE INPUT DATA FILE

All user initiated files are created at the GE information service level. Files may be created without line numbers, however, it is recommended that line numbers be used since they are required if the file is to be updated. The line numbers are ignored by the assembly program (PPSBA) since the program recognizes leading numeric characters as line numbers. The assembly program will consider the first non-numeric character as the first character of the record unless the character is a blank. If this character is blank, it will be ignored, and the next character will be the first character of the record even if it is a blank. This allows the user to immediately follow a line number with a label or to follow the line number with one blank and then the label field. For statements without labels the line number must be followed by at least two blanks.

If line numbers are not used, the assembly program will consider the first character as the first character of the record.

NOTE: The assembly program will accept the file without line numbers, but care must be taken if the GE service desequence command is used. The desequence command will remove the line number and the first blank. Therefore, to obtain a leading blank, two blanks must be used.

An example of file generation is shown in Figure D-3.

```
HHH
U#=UBQ70009,ABCD,JOE

PPS PROBLEM REPORTING SYSTEM

ANY PPS PROBLEMS/SOLUTIONS TO REPORT?N

USED     .16 UNITS
NEW SAMP

READY
100       TITLE.          TEST
110       ORG             #100
120   *   DEFINE RAM
130   A   EQU             #0
140   B   EQU             #1
150   *   MOVE 1          CHAR
160       LBL             A
170       L
180       LBL             B
190       X
                                  Balance of program
340       T               *
350       END
SAVE

READY
```

Log in sequence.

Typing SAVE will cause the above program to be stored a new SAMP

Figure D-3. EXAMPLES OF FILE GENERATION

D.6 CREATING THE JOB CARD FILE

To run any job in this catalog a job control file must be created and saved in your foreground catalog. Create this file as follows:

```
NEW  FN
100//BQ700** JOB (BQ700**,PW,PID),"ID",CLASS-B,
SAVE
```

NOTE: *FN* ... *Any name you choose*
**** ... *Your user number (ie, 02, 16, 51, etc.)*
PW ... *The password for the above user number*
,PID ... *Project ID (if any)*
ID ... *Any string of 1 to 20 characters (blanks included).*

Example:

```
100//BQ70002 JOB (BQ70002,MYPASS),"TEST CASE",CLASS=B,
```

D.7 RULES FOR NAMING FILES

When using the PPS program, the user must identify a file with a file name consisting of four to eight characters, with any combination of letters and digits. The first character must be alphabetic. The following are typical of acceptable file names:

SAMP LDAT TEST-32 FPROGRAM

The user will find that the PPS program will create files and manufacture file names using the first four characters of the input file followed by a period and three alphabetic characters. The period and the last three alphabetic characters can be considered as an extension of the basic file name.

D.8 REFERRING TO FILES IN COMMANDS

The user specifies a file by its name, and the system automatically finds the named file.

D.9 RUNNING THE ASSEMBLER

The assembler is accessed by entering "RUN OCA". You will then be asked for the names of your job card file and source file. The assembler will then create a file in your catalog with the name "XXXX.JOB". (XXXX is the first four characters of the source file name.) You will be informed of the job ID number for this run. The status of the job may be obtained by using this number. When the job is "DONE" the XXXX.JOB file may be purged.

Example:

```
RUN OCA↵

OCA          13:35PST  04/08/76

ONE CHIP ASSEMBLER
JOB CARD FILE NAME IS.. ?JC

OCA SOURCE FILE NAME IS ?TESTOCA↵

JOB ID = UMI3

USE THE COMMAND "BST UMI3" TO CHECK THE STATUS OF THIS JOB

USED   4.43 UNITS
```

D.9.1 OBTAINING JOB RESULTS

To obtain the results of the job, the following example is given:

```
BST IDID↵
IDID DONE↵
00156 RETURNED                    File Content
00103 SUBMITTED                   ----------------------------------------
 SYSOUT REPORTS
IDID01$$-RETURNED ←—— System information as to the running of the job
IDID0101-RETURNED ←—— Assembler report file
IDID0102-RETURNED ←—— Object tape file
IDID0103-RETURNED ←—— Object card file
                  ←—— System file name (SFN)
```

NOTE: *IDID ... The job ID number*
SFN ... The system file name

D.9.2 LISTING FILES

To list the system files enter the following:

```
BLI SFN;L
```

D.9.3 PURGING THE SYSTEM FILES

If you wish to keep any of the files created by the "OCA" program place them in your catalog using the following procedure:

```
OLD BATC:SFN
SAVE ANYNAME
```

This is necessary as the GE system will keep the system files for only 36 hours. Once you have "SAVED" the wanted files execute the following to purge the system:

```
PURGE JOB:IDID
```

This action removes the files from the system and allows yourself or others to use this space. This action is requested by GE Services.

D.9.4 TAPE PUNCH

To obtain a punched tape at your terminal the following procedure should be followed. Enter "RUN PUNCH". You will then be asked for the name of your file to be punched. The tape will then begin to be punched at your terminal.

Example:

```
RUN PUNCH

PUNCH          15:47PDT      07/02/76

 PUNCHED TAPE ØUTPUT PRØGRAM  (7/01/76)
 TYPE IN FILE NAME TØ BE PUNCHED?TAPE

 IS IT TØ BE PUNCHED IN ASCII ØR BINARY? (A ØR B) ?A

 WHEN PUNCH STØPS - TURN ØFF PUNCH AND DEPRESS RETURN KEY

PUNCH          15:48PDT      07/02/76


PUNCHED TAPE ØUTPUT PRØGRAM  (7/01/76)
TYPE IN FILE NAME TØ BE PUNCHED?TAPE

IS IT TØ BE PUNCHED IN ASCII ØR BINARY? (A ØR B) ?B

ENTER SIZE ØF RØM IN THØUSANDS (1 ØR 2) ?2

TØ INSURE VALID PAPER TAPE ØUTPUT,
TAPE PUNCHING EQUIPMENT MUST BE CØNFIGURED AS FØLLØWS:
  1. AUTØMATIC TAPE-ØN (DC2 ØR CNTL 'R') DISABLED
  2. AUTØMATIC TAPE-ØFF (DC4 ØR CNTL 'T') DISABLED
  3. ANSWER-BACK (ENQ ØR CNTL 'E') DISABLED
IS THIS MACHINE PRØPERLY CØNFIGURED (Y ØR N)?Y

WHEN PUNCH STØPS - TURN ØFF PUNCH AND DEPRESS RETURN KEY
```

NOTE: The file name entered must be in your catalog, not a System Report file name (SFN). Save the System Report file as described above.

D.10 ASSEMBLY COST DETERMINATION

At some time the operator may wish to determine cost of assembly he has just executed by determining the amount of system units used. When working in foreground the amount of time used is printed out by the system following completion of the program. However, in the case of working with background only costs for that part of the operation performed in foreground as printed out by the system. The background operation costs must be determined by using a special inquiry as shown in the example in Figure D-4. The total cost would be the sum of the units used in the foreground portion of the operation as shown in Figure 4-6 and the units shown in Figure D-4. Dollar rates for units may be obtained from your G.E. representative.

Typing BRE and job ID prints out number of background units used.

```
BRE UON5
ACTIVITY          APPROXIMATE CRU'S

  01                   0.0041
                    ----------
TOTAL                  0.0041

READY
```

Figure D-4. BACKGROUND ASSEMBLY COST DETERMINATION

APPENDIX E
ANALOG TO DIGITAL CONVERSION EXAMPLES

Analog to digital conversion is accomplished by doing a digital to analog conversion and then comparing this D/A to the analog input.

The successive approximation method of analog to digital conversion is an iterative process, therefore an ideal method for use with microcomputers. This process may be implemented in the following manner (see Figure E-1): Set the MSB of the D/A (DI/O7), and test the comparator output (INT0). If the comparator output is at the -V state, leave the MSB set, and set a corresponding bit in data memory to indicate that the bit has been left set. If the comparator output is at the +V state, reset the MSB. In either case, continue this process with the next MSB, down to the LSB. When done the digital value of the analog signal will be stored in data memory for further use.

The flowchart (Figure E-2) and assembly listing (Figure E-3) shows how the analog to digital conversion was done in this example.

The hardware necessary for the analog to digital conversion (Figure E-1) is only a resistor ladder network and an LM 311 voltage comparator. The I/O necessary is nine ports, in this example eight DI/O's (DI/O 0-7) for the outputs, and INT0 for the input. The use of the eight DI/O's in the A/D routine does not limit their use for other things in a total system design. For example, these same DI/O's when not being used by the A/D routine could also be used to scan a keyboard and strobe a display.

The program was written for the MM75XX and MM76XX. It requires 36 instruction memory locations and three data memory locations. If used with the MM77 or MM78, the program memory requirement would be reduced to 33 instruction memory locations.

MM75 and MM76

```
AD0     LBL     #37
        NOP
        ROS
        :
        :
AD2A    LBA
        NOP
        SOS
        :
        :
        T       AD2
        END
```

MM77 and MM78

```
AD0     LB      7
        ROS
        :
        :
        :
AD2A    LBA
        SOS
        :
        :
        :
        T       AD2
        END
```

Data memory used is #00, 01, #02. #00 and 01 are used for storing the 8-bit result of the conversion. #02 is used for storing the bit pointer (N).

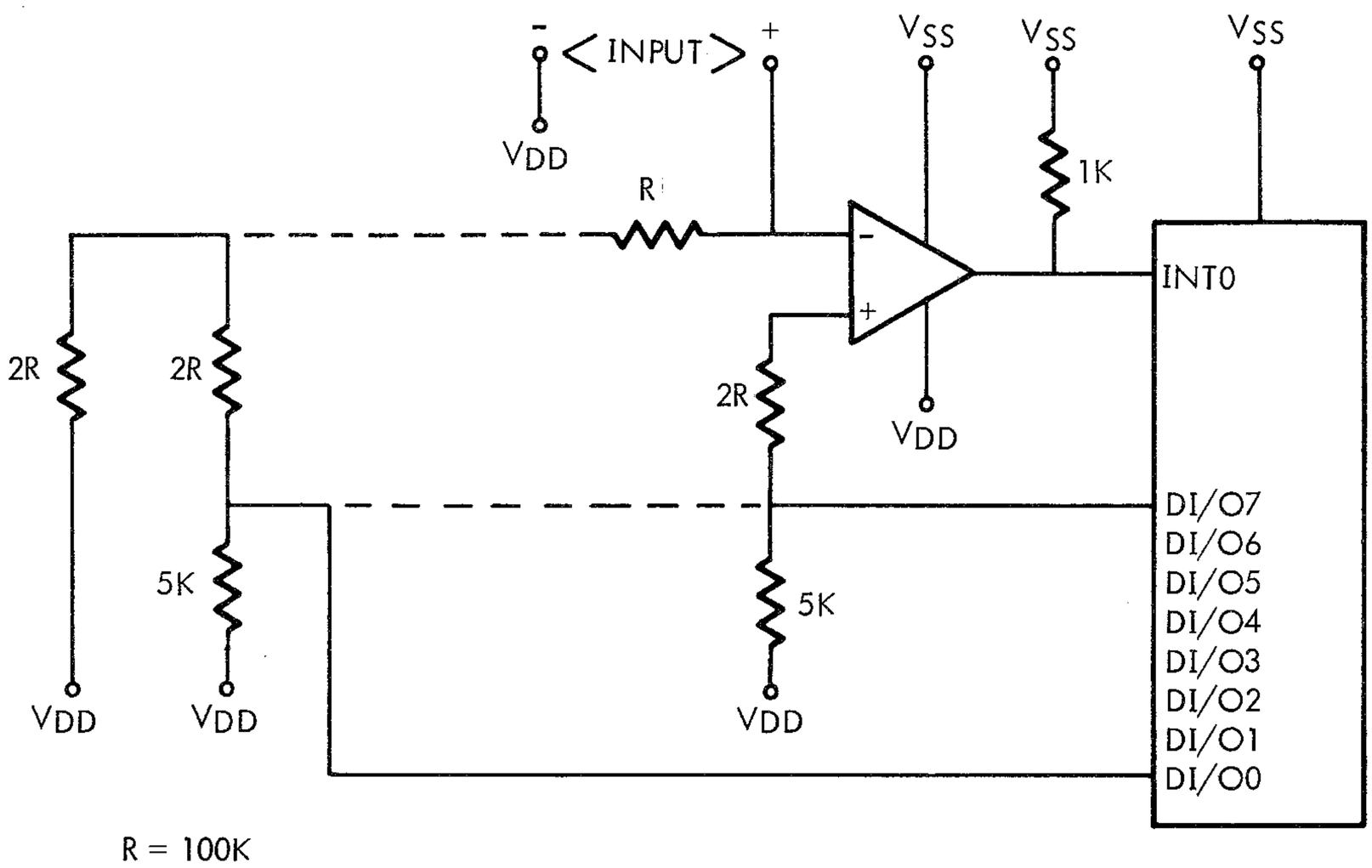


Figure E-1. ANALOG TO DIGITAL CONVERSION SCHEMATIC

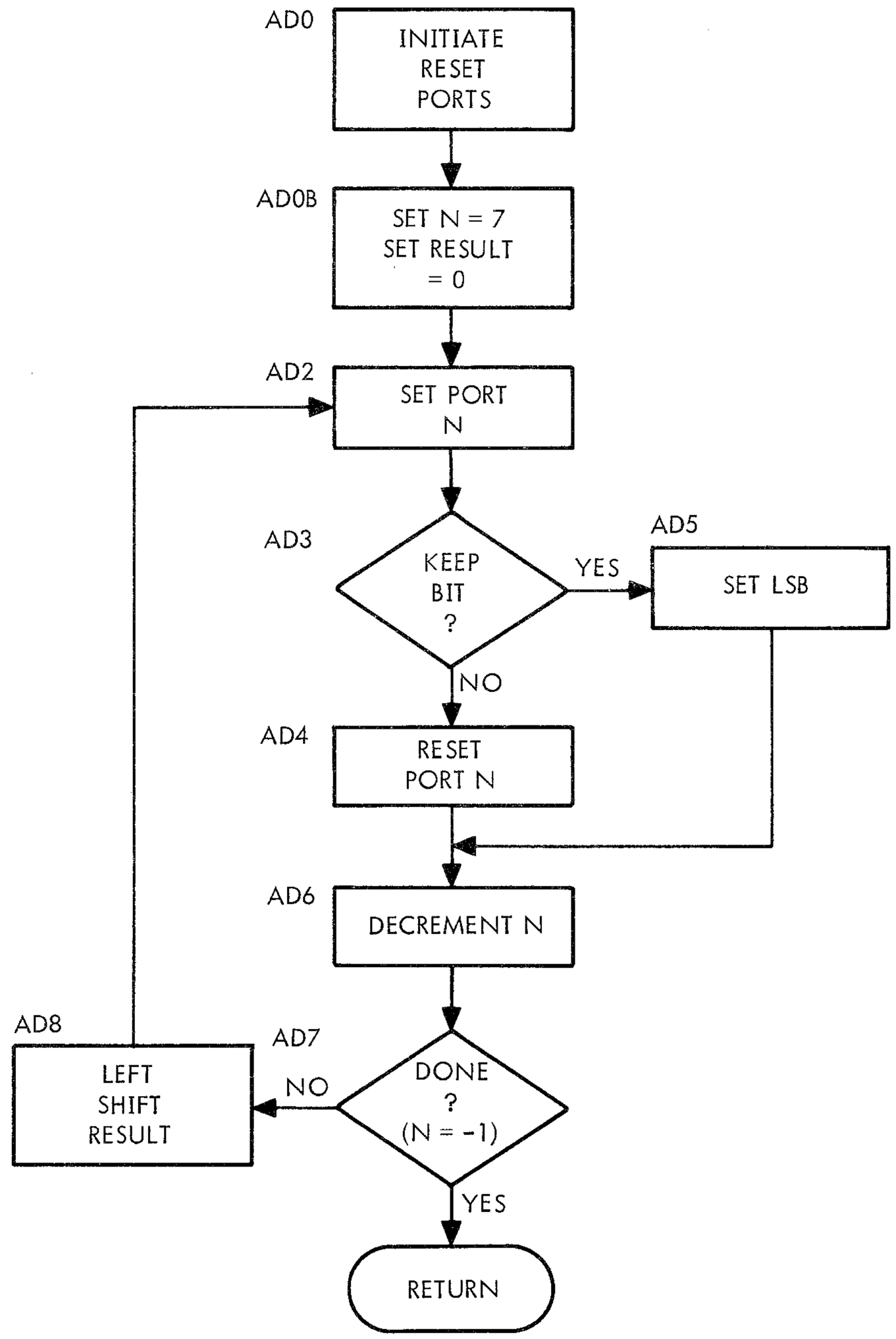


Figure E-2. A/D ROUTINE FLOW DIAGRAM

```
*THIS IS AN 8-BIT A/D CONVERSION ROUTINE.
*THIS ROUTINE USES THE SUCESSIVE APPROXIMATION
*METHOD FOR THE CONVERSION.
*TOTAL CONVERSION TIME:      275 CYCLES
*INITIALIZATION TIME:         51 CYCLES
*TIME PER BIT:                28 CYCLES
*MEMORY LOCATIONS USED:       36 WORDS
*
*
                ORG     #100
0100 27 AD0     LBL     #37     INITIALIZATION
0120 1F
0110 00         NOP
0108 14 AD0A    ROS             RESET DIO'S 0-7
0104 58         DECB
0102 5C
0121 F7         T       AD0A
0130 22 AD0B    LB      2
0118 77         LAI     7       SET N=7
010C 70 AD0C    LAI     0       SET RESULT REG=0
0106 5C         XDSK
0123 F3         T       AD0C
0111 22 AD2     LB      2
0128 53         L       3       LOAD N
0114 44 AD2A    LBA             POINT TO N
010A 00         NOP
0125 10         SOS             SET DIO N
0132 00         NOP
0139 04 AD3     INT0L           KEEP BIT?
013C C7         T       AD4     NO
011E 21 AD5     LB      1       YES, SET LSB RESULT
012F 10         SB      1
0117 22 AD6     LB      2
010B 50         L
0105 6F AD7     AISK    #F      DECR N, DONE?
0122 F1         T       AD8     NO
0131 02         RT              YES
0138 14 AD4     ROS             RESET PORT N
011C E8         T       AD6
010E 5C AD8     XDSK    0       STORE NEW N
0127 0D         RC              LEFT SHIFT RESULT
0113 50 AD8A    L
0109 40         AC
0124 5C         XDSK
0112 EC         T       AD8A
0129 EE         T       AD2     CONVERT NEXT BIT

 28 AVAIL

                END
```

Figure E-3. A/D CONVERSION ROUTINE EXAMPLE

APPENDIX F
PPS-4/1 MACHINE LANGUAGE CODING AID

P COUNTER * (ORIGIN + VALUE)	ON PAGE TRANSFER CODE	INSTRUCTIONS LABEL	CODE	OPERAND	COMMENT	INSTRUCTION HEX CODE
00	FF					
20	DF					
10	EF					
08	F7					
04	FB					
02	FD					
21	DE					
30	CF					
18	E7					
0C	F3					
06	F9					
23	DC					
11	EE					
28	D7					
14	EB					
0A	F5					
25	DA					
32	CD					
39	C6					
3C	C3					
1E	E1					
2F	D0					
17	E8					
0B	F4					
05	FA					
22	DD					
31	CE					
38	C7					
1C	E3					
0E	F1					
27	D8					
13	EC					

P COUNTER * (ORIGIN + VALUE)	ON PAGE TRANSFER CODE	INSTRUCTIONS LABEL	CODE	OPERAND	COMMENT	INSTRUCTION HEX CODE
09	F6					
24	DB					
12	ED					
29	D6					
34	CB					
1A	E5					
2D	D2					
36	C9					
3B	C4					
1D	E2					
2E	D1					
37	C8					
1B	E4					
0D	F2					
26	D9					
33	CC					
19	E6					
2C	D3					
16	E9					
2B	D4					
15	EA					
2A	D5					
35	CA					
3A	C5					
3D	C2					
3E	C1					
3F	C0					
1F	E0					
0F	F0					
07	F8					
03	FC					
01	FE					

NOTE: INCB, DECB, TL, TML, LBL, SKBEI and SKAEI are 2 byte instructions.
TLB and TMLB are 3 byte instructions.
**HOO, H40, H80, or HCO + value shown will give P Register (H is a hex character from 0 thru 7)*

PPS 607 732 gS

Rockwell

READER COMMENT FORM

We are interested in providing our customers with the best documentation possible, and will appreciate any comments you may have on this manual. If you are reporting an error, please be as specific as possible; a marked-up photocopy of the applicable page(s) would be helpful.

Return this form and any supporting material to:

Documentation Manager
D727, RC55
Rockwell International
Microelectronic Devices
3310 Miraloma Ave.
Anaheim, CA 92803

DOCUMENT NAME: XPO-1 USER'S MANUAL

DOCUMENT NUMBER: 29410 N45 **REVISION:** 1

Comments:

(Name)

(Company)

(Street Address)

(City) (State) (Zip)

CUT ALONG THIS LINE

DOCUMENT REGISTRATION FORM

Please fill out and return this card to automatically receive all updates to your manual.

DOCUMENT NAME:

DOCUMENT NUMBER: **REVISION:**

(Name)

(Company)

(Street Address)

(City) (State) (Zip)

PLACE
STAMP
HERE

Documentation Manager
D727, RC55
Rockwell International
MICROELECTRONIC DEVICES
3310 Miraloma Ave.
Anaheim, CA 92803