CHRIS WILLIAMS '85

Customer Order Number 424410284-001
NSC Publication Number 424410284-001A
February 1985

COPS™

The COPS Programming Manual

2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, California 95052-8090

CONTENTS

FIGURES

TABLES

Chapter 1

INTRODUCTION TO COPS MICROCONTROLLERS

1.1 SCOPE AND PURPOSE OF THIS MANUAL

How is an efficient COPS program written? The answer to this question begins with dividing the broad category of microcomputer into two areas: microcontrollers and microprocessors. This distinction is made because these are really two different types or classes of devices. Microcontrollers generally have a dual-bus architecture rather than the memory-mapped von Neumann architecture common in most microprocessors. For control applications, microcontrollers are generally more memory efficient than microprocessors. The microcontroller instruction set is quite different in nature than the microprocessor instruction set. Microcontrollers are invariably single-chip devices and microprocessors are, generally, multi-chip devices. Microcontrollers dominate the microcomputer marketplace in terms of volume. To be sure, the division between microcontroller and microprocessor is sometimes blurred but the distinction is real nonetheless.

COPS devices are microcontrollers. It is the intent of this manual to provide the user/programmer of COPS microcontrollers the requisite information to write an efficient COPS program — to take full advantage of the characteristics of the devices. To achieve that end, this manual is written from the programmer's perspective. The various characteristics of COPS microcontrollers are described in the context of the effect of those characteristics on the programming of the devices. The COPS architecture is discussed; the instruction set is described in detail; general techniques of COPS programming are explained; and standard programs are provided. The standard programs are commonly used as vehicles to illustrate various programming techniques. The user or reader would be well advised to carefully read the explanations associated with routines showing multiple implementations. The intent of providing multiple implementations is not to show how many different ways a routine can be written but rather to show techniques, "tricks", tradeoffs, considerations, etc. Therefore, a great deal of useful information is included in those explanations.

This manual does not attempt to explain the detailed physical or electrical characteristics of COPS microcontrollers. To the extent any such information is provided here, it is to explain some software effect or characteristic. Therefore, the physical details may be simplified to clarify the software explanation.

1.2 THE COPS MICROCONTROLLER FAMILY

1.2.1 General Description

COPS devices are general purpose, single-chip microcontrollers. These microcontrollers are complete microcomputers containing all system timing, internal logic, ROM, RAM, and I/O necessary to implement dedicated control functions in a wide variety of applications. The COP400 family presently consists of a large number of devices enabling the user to select the device best suited to his application. The software is upward compatible – programs written on one device may be transferred to the next larger device (in terms of memory capacity) with little or no change. The package pin configurations have also been selected so that movement up or down (using memory size as the variable parameter) within the family can be accomplished easily. All COPS microcontrollers, regardless of memory size or number of pins, have the same basic architectural structure. In addition to the large number and wide range of devices, all COPS microcontrollers have a number of I/O options, specified at the same time as the program, which allow the user to tailor, within limits, the I/O characteristics of the microcontroller to the system. Thus, the user can optimize the microcontroller for the system, thereby achieving maximum capability and minimum cost.

This manual deals with the basic functionality of COPS microcontrollers. It does not address electrical differences among the various devices. Thus, this manual does not distinguish between the COP400 and the COP300 series. These two series differ only in electrical characteristics and not in function. This manual further does not distinguish the high-speed devices from the low-power devices or from the CMOS devices except to the extent that some of the devices may have features that affect programming.

1.2.2 COPS ROMless Microcontrollers

Several COPS microcontrollers are designed to use external program memory. Basically, these devices have been created by removing the ROM from their single-chip counterparts. These devices are primarily intended to be used in program development and debug, device emulation, and low-volume production. Table 1-1 provides a list of COP400 ROMless devices currently available or in design. The devices are designed so that each COPS microcontroller has at least one ROMless device that can be used for accurate emulation. Since these devices are functionally equivalent to the single-chip microcontrollers, this manual does not generally distinguish the ROMless device from its single-chip counterpart.

1.2.3 COPS Single-Chip Microcontrollers

Table 1-2 provides a list of COPS single-chip microcontrollers currently available or in design. It is readily apparent that the list is quite extensive. Many of the variations are simply different packagings of the same device, *e.g.*, the COP441 is the COP440 in a 28-lead package; the COP442 is the COP440 in a 24-lead package; the COP440 is a 40-pin device. Another important characteristic is the commonality of the pinouts of the single-chip devices: all 40-pin devices have the same pinout; all 24-pin devices have the same pinout; the COP411L and COP411C have the same pinout; the COP422 and COP422L have the same pinout. See Figures 1-1 through 1-4.

TABLE 1-1. COPS ROMLESS MICROCONTROLLERS - GENERAL SOFTWARE OVERVIEW

COP	401L	402	402M	404C	404L	404	2404	409
External ROM x 8	Up to 512	Up to 1024		Up to 2048				Up to 32768
RAM x 4	32	64		128		160		512
Inputs	0	4		4		4		4
Bidirectional TRI-STATE® I/O	8	8		8		16		8
Bidirectional I/O	4	4		4		8		4
Outputs	4	4		4		4		4
Serial I/O and External Event Counter	Yes	Yes		Yes		Yes		Yes
Internal Time Base Counter	No	Yes		Yes		Yes		Yes
Time Base Counter Programmable	No	No		Yes	No	Yes		Yes
Interrupt	No	Yes	No	Yes		Yes - 4 sources		Yes
Stack Levels	2	3		3		4	4 per CPU	8
Microbus™ Option	No	No	Yes	Yes	No	Yes		No
Instruction Cycle (μs) min - max	15-40	4-10		4-DC	15-40	4-10		4-25
Package Size (pins)	40	40		48	40	48		40
Availability	Now	Now		Now	Now	Now		Future*

* These devices are NOT available as of this writing. The information on these devices is preliminary and subject to change. Advance information has been provided for completeness and as an aid to the user. Announcements will be made by National Semiconductor at the appropriate times regarding the availability and ultimate characteristics of these devices

TABLE 1-2. COPS MICROCONTROLLERS - GENERAL SOFTWARE OVERVIEW

<table>
<tr><th>COP:</th><th>410L</th><th>410C</th><th>411L</th><th>411C</th><th>413L</th><th>420</th><th>420L</th><th>424C</th><th>421</th><th>421L</th><th>425C</th><th>422</th><th>422L</th><th>426C</th><th>444L</th><th>444C</th><th>445L</th><th>445C</th><th>440</th><th>441</th><th>442</th><th>2440</th><th>2441</th><th>2442</th><th>484</th><th>485</th></tr>
<tr><td>ROM X 8</td><td colspan="5">512</td><td colspan="9">1024</td><td colspan="10">2048</td><td colspan="2">4096</td></tr>
<tr><td>RAM x 4</td><td colspan="5">32</td><td colspan="9">64</td><td colspan="4">128</td><td colspan="6">160</td><td colspan="2">256</td></tr>
<tr><td>INPUTS</td><td colspan="5">0</td><td colspan="3">4</td><td colspan="6">0</td><td colspan="2">4</td><td colspan="2">0</td><td colspan="2">4</td><td>0</td><td colspan="2">4</td><td>0</td><td>4</td><td>0</td></tr>
<tr><td>BIDIRECTIONAL TRI-STATE I/O</td><td colspan="5">8</td><td colspan="9">8</td><td colspan="4">8</td><td>16</td><td colspan="2">8</td><td>16</td><td colspan="2">8</td><td colspan="2">8</td></tr>
<tr><td>BIDIRECTIONAL I/O</td><td colspan="2">4</td><td colspan="2">3</td><td>4</td><td colspan="3">4</td><td colspan="3">4</td><td colspan="3">2</td><td colspan="4">4</td><td>8</td><td colspan="2">4</td><td>8</td><td colspan="2">4</td><td colspan="2">4</td></tr>
<tr><td>OUTPUTS</td><td colspan="2">4</td><td colspan="3">2</td><td colspan="3">4</td><td colspan="3">4</td><td colspan="3">2</td><td colspan="4">4</td><td colspan="6">4</td><td colspan="2">4</td></tr>
<tr><td>SERIAL I/O AND EXTERNAL EVENT COUNTER</td><td colspan="5">YES</td><td colspan="9">YES</td><td colspan="4">YES</td><td colspan="6">YES</td><td colspan="2">YES</td></tr>
<tr><td>INTERNAL TIME BASE COUNTER</td><td colspan="5">NO</td><td colspan="9">YES</td><td colspan="4">YES</td><td colspan="6">YES</td><td colspan="2">YES</td></tr>
<tr><td>TIME BASE COUNTER PROGRAMMABLE</td><td colspan="5">NO</td><td colspan="2">NO</td><td>YES</td><td colspan="2">NO</td><td>YES</td><td>NO</td><td colspan="2">YES</td><td>NO</td><td>YES</td><td>NO</td><td>YES</td><td colspan="8">YES</td></tr>
<tr><td>INTERRUPT</td><td colspan="5">NO</td><td colspan="3">YES</td><td colspan="6">NO</td><td colspan="2">YES</td><td colspan="2">NO</td><td colspan="2">YES 4 SOURCES</td><td>YES 2 SOURCES</td><td colspan="2">YES 4 SOURCES</td><td>YES 2 SOURCES</td><td>YES</td><td>NO</td></tr>
<tr><td>STACK LEVELS</td><td colspan="5">2</td><td colspan="9">3</td><td colspan="4">3</td><td colspan="3">4</td><td colspan="3">4 PER CPU</td><td colspan="2">4</td></tr>
<tr><td>MICROBUS OPTION</td><td colspan="5">NO</td><td>YES</td><td>NO</td><td>YES</td><td colspan="6">NO</td><td>NO</td><td>YES</td><td colspan="2">NO</td><td colspan="2">YES</td><td>NO</td><td colspan="2">YES</td><td>NO</td><td colspan="2">NO</td></tr>
<tr><td>INSTRUCTION CYCLE (MS) MIN-MAX</td><td>16-40</td><td>4-DC</td><td>16-40</td><td>4-DC</td><td>16-40</td><td>4-10</td><td>16-40</td><td>4-DC</td><td>4-10</td><td>16-40</td><td>4-DC</td><td>4-10</td><td>16-40</td><td>4µSEC TO DC</td><td>16-40</td><td>4-DC</td><td>16-40</td><td>4-DC</td><td colspan="6">4-10</td><td colspan="2">4-25</td></tr>
<tr><td>PACKAGE SRE (PINS)</td><td colspan="2">24</td><td colspan="3">20</td><td colspan="3">28</td><td colspan="3">24</td><td colspan="3">20</td><td colspan="2">28</td><td colspan="2">24</td><td>40</td><td>28</td><td>24</td><td>40</td><td>28</td><td>24</td><td>28</td><td>24</td></tr>
<tr><td>AVAILABILITY</td><td colspan="5">NOW</td><td colspan="3">NOW</td><td colspan="3">NOW</td><td colspan="3">NOW</td><td colspan="2">NOW</td><td colspan="2">NOW</td><td colspan="6">NOW</td><td colspan="2">FUTURE*</td></tr>
</table>

* THESE DEVICES ARE NOT AVAILABLE AS OF THIS WRITING. THE INFORMATION ON THESE DEVICES IS PRELIMINARY AND SUBJECT TO CHANGE. ADVANCE INFORMATION HAS BEEN PROVIDED FOR COMPLETENESS AND AS AN AID TO THE USER. ANNOUNCEMENTS WILL BE MADE BY NATIONAL SEMICONDUCTOR AT THE APPROPRIATE TIME REGARDING THE AVAILABILITY AND ULTIMATE CHARACTERISTICS OF THESE DEVICES.

BA-37-C

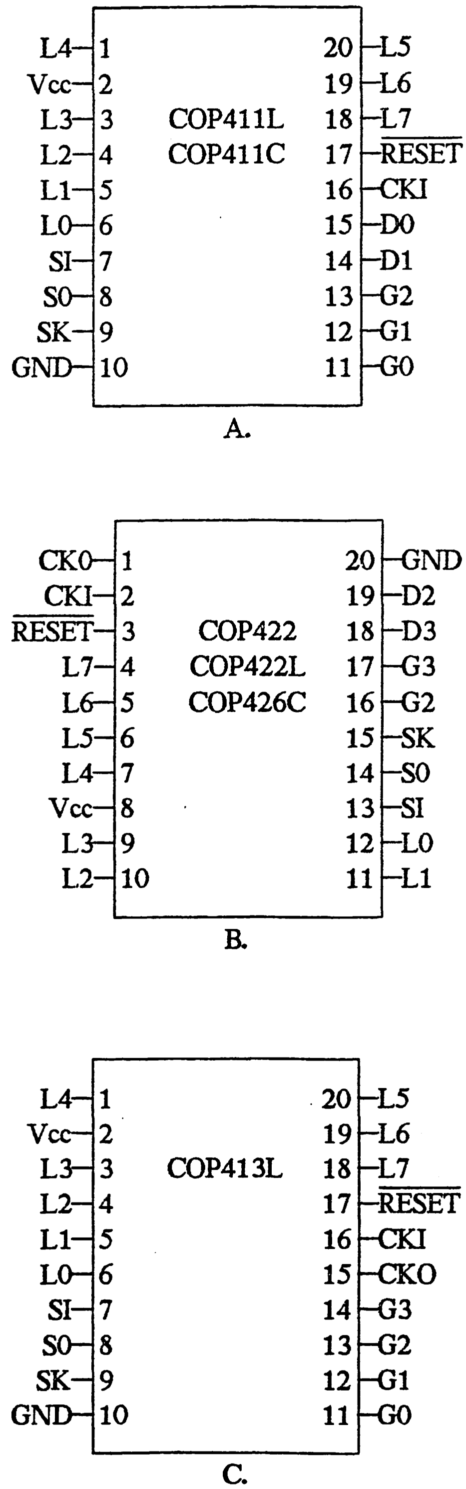


Figure 1-1. Pinouts for 20-Pin COPS Microcontrollers

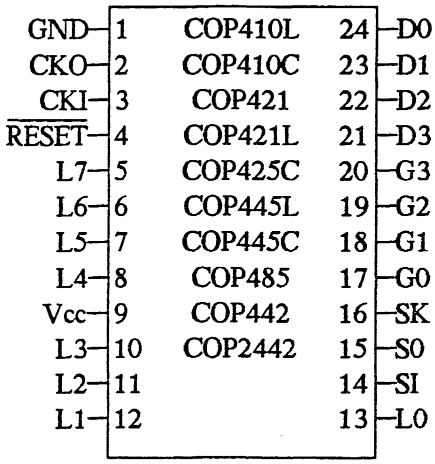


Figure 1-2. Pinout for 24-Pin COPS Microcontrollers

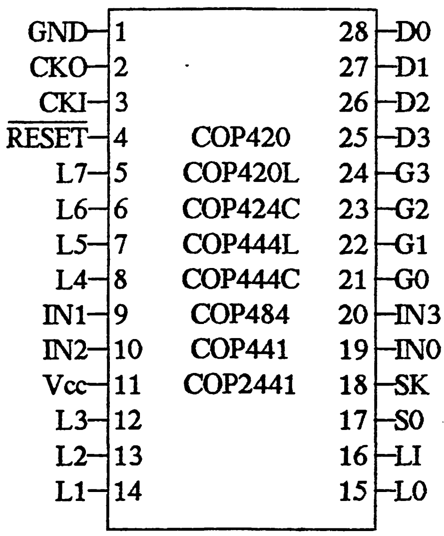


Figure 1-3. Pinout for 28-Pin COPS Microcontrollers

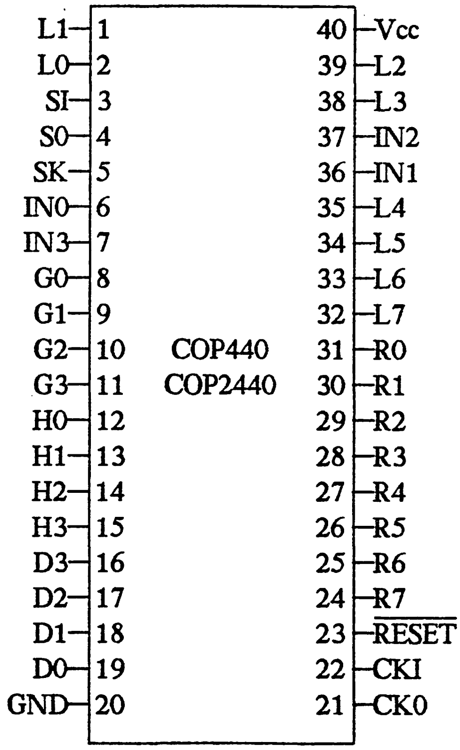


Figure 1-4. Pinout for 40-Pin COPS Single-Chip Microcontrollers

1.2.4 Conclusion

COPS microcontrollers comprise a broad, general purpose, powerful, and flexible family of devices. The hardware and software compatibility of the devices allow the user to move easily within the family as the need arises or the application dictates. Many ROMless devices are available to aid in emulation and development. The applications of COPS devices are unlimited. COPS microcontrollers have been used in automotive (trip computer, seat position controller, electronic instrument cluster, ignition systems, diagnostic systems), appliance (ovens, microwave ovens, vacuum cleaners, sewing machines, washers, driers, food processors), home electronic (electronically tuned radios, cassette recorders, video cassette recorders, stereo systems), security system, timekeeping, energy management, industrial/commercial (utility meters, keyboard encoders, cash registers, dictation equipment, coin changers, vending machines, jukeboxes), telephone (repertory dialers, simple phone dialers, call timers), exercise equipment (exercise bicycle, jogging machine), miscellaneous home (garage door openers, lawn sprinklers, Christmas ornaments, cable television), toy, game, and many other applications.

Chapter 2

ARCHITECTURE OF COPS MICROCONTROLLERS

2.1 INTRODUCTION

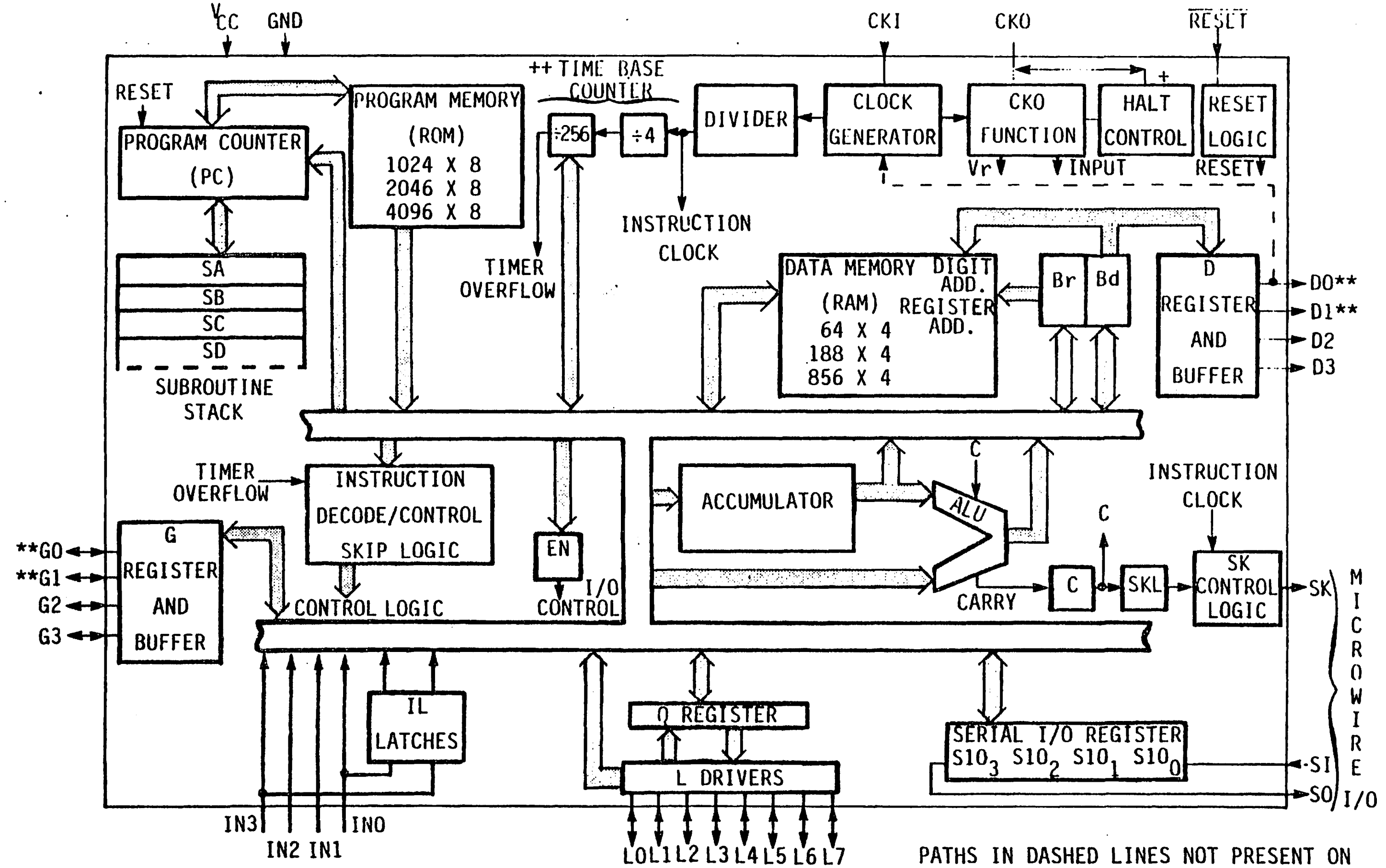
This section deals with the architecture of COPS microcontrollers. Figure 2-1 is the generic block diagram for COPS microcontrollers. The diagram is accurate as is for the COP420/421/422, COP420L/421L/422L, COP424C/425C/426C, COP444L/445L, COP444C/445C, and COP484/485 devices. The addition or deletion of certain elements creates the other microcontrollers in the COPS family. Figure 2-2, the block diagram of the COP410L/411L/413L and COP410C/411C, Figure 2-3, the block diagram of the COP440/441/442, and Figure 2-4, the block diagram of the COP2440/2441/2442, illustrate this fact. It is clear, even from a cursory examination, that all COPS microcontrollers possess the fundamental architecture that is indicated in Figure 2-1. Therefore, Figure 2-1 is the focal point for the discussion of the COPS architecture. The additions or deletions that lead to the other block diagrams are discussed where appropriate.

2.2 COPS MEMORY STRUCTURE

2.2.1 Program Memory – ROM

The program memory in COPS microcontrollers is a read-only memory (ROM) organized as a number of eight-bit words. COPS microcontrollers with ROM capacities of 512, 1024, and 2048 words are presently available. Devices with ROM capacity of 3072 and 4096 words are currently in design. The ROM words are addressed sequentially by a binary program counter (in ROMless devices, the program counter is brought out to pins to address external memory). The program counter starts at zero and, if there are no jumps or subroutines or table lookups, will increment to the maximum value possible for the device and rolls over to zero and begins again.

Internally, COPS microcontrollers have a semi-transparent page, block, and chapter structure to the ROM. A page is composed of 64 contiguous ROM words. The lower six bits of the program counter are zeroes at the first address of a page and ones at the last address of a page. A block, which is significant only in the table lookup and indirect jump operations, is composed of four contiguous pages (256 contiguous ROM words). The lower eight bits of the program counter are zeroes at the first address of a block and ones at the last address of a block. The first address of a block is also the first address of a page and the last address of a block is also the last address of a page. The chapter division is relevant only in COPS devices with more than 2048 ROM words or ROMless devices capable of addressing more than 2048 ROM words. The lower 11 bits of the program are zeroes at the first address of a chapter and ones at the last address of a chapter. The first address of a chapter is also the first address of a block and the last address of a chapter is also the last address of a block. Table 2-1 lists the hexadecimal address and the corresponding page/chapter/block divisions.



PATHS IN DASHED LINES NOT PRESENT ON ALL DEVICES.

+SD REGISTER AVAILABLE ONLY ON 36- AND 48-LEAD DEVICES.

*THESE PINS NOT AVAILABLE ON 24- AND 28-LEAD DEVICES.

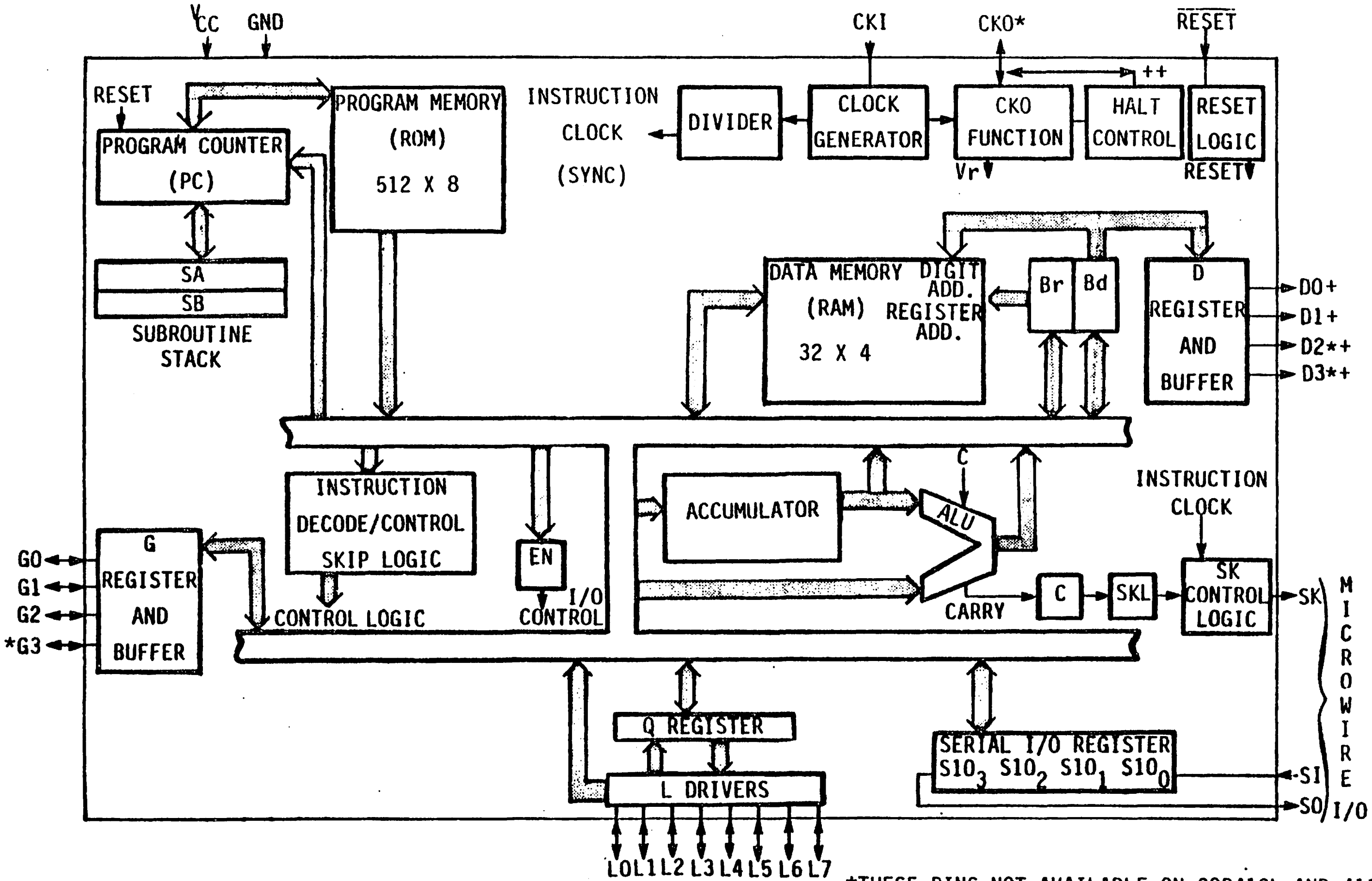
**THESE PINS NOT AVAILABLE ON 20-LEAD DEVICES.

+ONLY CMOS COPS EXCEPT 411C.

++NOT AVAILABLE ON ¼K DEVICES.

BA-01-0

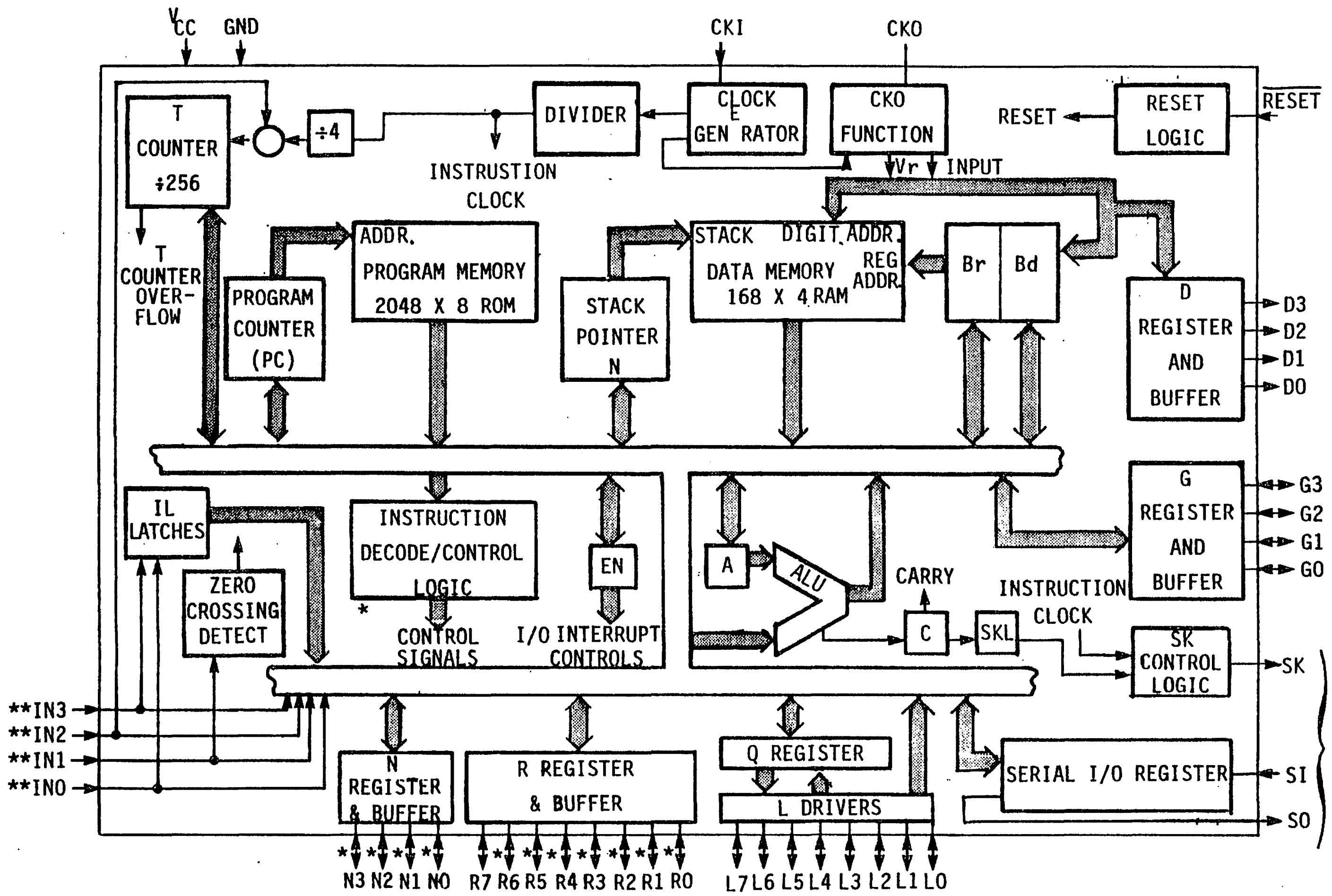
Figure 2-1. Basic Block Diagram for COPS Microcontrollers



*THESE PINS NOT AVAILABLE ON COP410L AND 411C.
+THESE PINS NOT AVAILABLE NO THE COP413L.
++ONLY CMOS COPS EXCEPT 411C

BA-02-0

Figure 2-2. COP410L/411L/413L and COP410C/411C Block Diagram

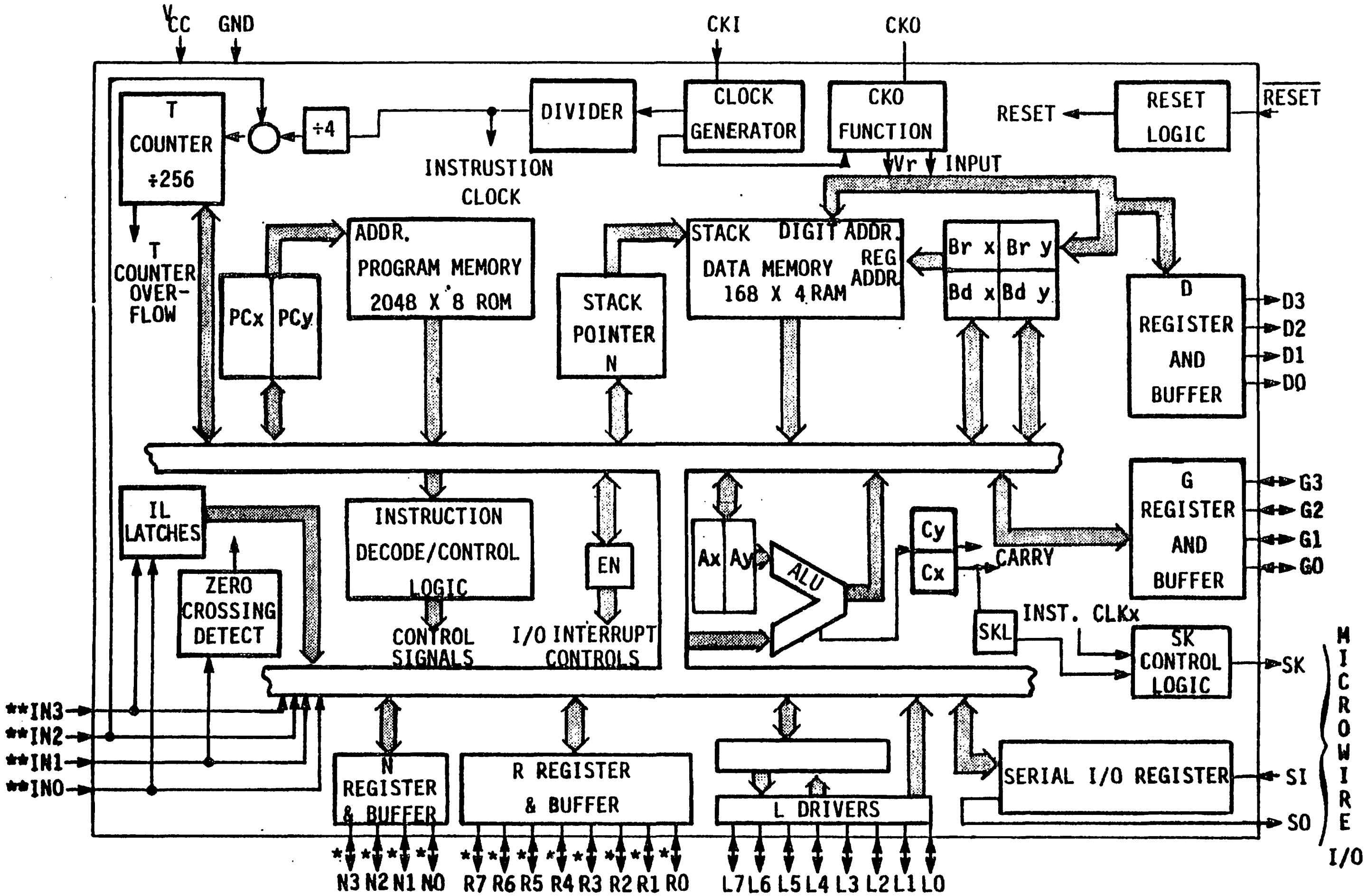


*THESE PINS NOT AVAILABLE ON 28- OR 24- LEAD DEVICES.

**THESE PINS NOT AVAILABLE ON 24- LEAD DEVICES.

BA-03-0

Figure 2-3. COP440/411/442 Microcontrollers Block Diagram



*THESE PINS NOT AVAILABLE ON 28- OR 24- LEAD DEVICES.

**THESE PINS NOT AVAILABLE ON 24- LEAD DEVICES.

BA-04-0

Figure 2-4. COP2440/2411/2442 Dual CPU Microcontrollers - Block Diagram

TABLE 2-1. ADDRESS-PAGE-BLOCK-CHAPTER MAPPING

HEX ADDRESS	PAGE	BLOCK	CHAPTER
000-03F 040-07F 080-0BF 0C0-0FF	0 1 2 3	0	0
100-13F 140-17F 180-1BF 1C0-1FF	4 5 6 7	1	
200-23F 240-27F 280-2BF 2C0-2FF	8 9 10 11	2	
300-33F 340-37F 380-3BF 3C0-3FF	12 13 14 15	3	
400-43F : 4C0-4FF	16 : 19	4	
500-53F : 5C0-5FF	20 : 23	5	
600-63F : 6C0-6FF	24 : 27	6	
700-73F : 7C0-7FF	28 : 31	7	

TABLE 2-1. (Cont)

HEX ADDRESS	PAGE	BLOCK	CHAPTER
800-83F : 8C0-8FF	32 : 35	8	
900-93F : 9C0-9FF	36 : 39	9	
A00-ACF : AC0-AFF	40 : 43	10	
B00-B3F : BC0-BFF	44 : 47	11	
C00-C3F : CC0-CFF	48 : 51	12	1
D00-D3F : DC0-DFF	52 : 55	13	
E00-E3F : EC0-EFF	56 : 59	14	
F00-F3F : FC0-FFF	60 : 63	15	
1000-103F : .	64 : .		

This internal structure is semi-transparent. Only some jumps, some subroutine calls, and table lookups are affected by this structure. As indicated earlier, the block divisions come into play only in the table lookups and indirect jumps. The page and chapter divisions affect some direct jumps and subroutine calls. Chapter 4 explains the effects of these divisions on the pertinent instructions. Complete operational programs can be written without consideration of this internal structure. Such a program, however, will use more code, and therefore require larger ROM capacity, than a program written with this structure in mind. Chapter 4 will address this in greater detail. This page/block/chapter structure has no effect on the program counter. **The binary program counter will freely increment through page, block, or chapter boundaries.**

2.2.2 Data Memory – RAM

The data memory (RAM) in COPS microcontrollers is organized as a matrix. Each row in the matrix is called a register; each column in the matrix is called a digit. A digit is 4 bits wide. As shall be seen, this particular structure contributes to the general efficiency of COPS microcontroller. All RAM addressing is based on this register-digit (or row-column) organization. The RAM address register identifies a specific digit in the RAM matrix. COPS devices with RAM sizes of 32 digits (4 registers by 8 digits, 128 bits), 64 digits (4 registers by 16 digits, 256 bits), 128 digits (8 registers by 16 digits), and 160 digits (10 registers by 16 digits) are presently available. A device with RAM sizes 256 digits (16 registers by 16 digits) is in design. A ROMless device with 512 digits (32 registers by 16 digits) of RAM is also in design.

The RAM in COPS microcontrollers is not in the program memory space. The RAM is not addressed by the program counter but has its own address register, the B register. The B register can be loaded directly or through the accumulator. Since the RAM has its own address register, most COPS instructions which access RAM do not contain an address field. This tends to promote ROM code efficiency. The B register is divided into two distinct parts: Br – the row or register address and Bd – the column or digit address. Bd is 4 bits wide in all COPS microcontrollers. Br is between 2 and 5 bits wide depending on the particular device. Bd, in addition to being the digit address, is the source for the D output register. On software command, the contents of Bd can be transferred to the D port where the information is latched.

The data memory digit addressed by the B register is normally accessed through the accumulator. The contents of the RAM digit may be directed, under software command, to one of several output ports as well as used in the normal program flow. Two instructions, LDD and XAD, carry a RAM address with them. These instructions operate (load or exchange) on the specified RAM digit without modifying the B register.

2.2.3 Subroutine Stack

COPS microcontrollers have a subroutine stack of two, three, or four (eight on the COP409) save registers. On all COPS microcontrollers with two or three save registers in the subroutine stack, a physical transfer of register contents within the stack occurs on all operations affecting the stack, primarily calls and returns. On these devices, the stack is physically and logically separate from data RAM. The user does not have access to the stack and, therefore, may not read or write the stack in these devices.

On COPS devices with four or more stack levels, the stack is located in data RAM. Four stack levels use up one data register. The user has access to the stack since the data RAM contains the stack. However, in **no** case does the stack expand beyond its assigned area into the rest of the data memory. These devices contain a stack pointer which is incremented or decremented on operations affecting the stack. Overflowing the stack merely causes the stack pointer to wrap around from its maximum value back to zero. On the COP440 and COP2440 series, **only** the user also has access to the stack pointer and may read or write the pointer. In all of these devices which permit stack access, the programmer has increased versatility. However, caution is recommended. Increased power brings with it increased risk, and the programmer should exercise care that the stack is not accidentally accessed in these devices.

2.3 THE ARITHMETIC LOGIC UNIT

The arithmetic logic unit (ALU) in COPS microcontrollers is a 4-bit parallel binary adder. It performs all the arithmetic and logic functions in the microcontrollers. The destination for all such operations is the 4-bit accumulator, and one input to the ALU is always the accumulator. The other input is either an immediate operand as specified by an instruction or, more commonly, the data RAM digit addressed by the B register. The one-bit C register sometimes is a third input to the ALU. The ALU outputs a carry bit which, depending on the instruction being executed, can be loaded into the C register. See the instruction set description and Section 4.4 for more details on carry and the C register.

2.4 INPUT/OUTPUT

2.4.1 Inputs

Only one input port, the IN port, is available on COPS microcontrollers. This port is available only on devices with 28 or more leads. On software command, the four IN lines are read, as a group, into the accumulator. In addition to the the direct inputs, IN0 and IN3 have latches associated with them. These latches capture a high to low transition on the particular line. The status of the latches is read into the accumulator on software command. Thus, the programmer can read the present status of the IN lines directly or can read the status of the latches associated with IN0 and IN3.

The IN1 input can, under software control, serve as an interrupt input. The enabling or disabling of interrupts is a software decision. As such, in a given program, interrupts may be always enabled, never enabled, or sometimes enabled. On the COP440/441 and COP2440/2441 devices only, IN1 may be mask programmed to be a zero crossing input. As such, interrupts may be generated at each zero crossing. Note that the zero crossing option is a mask, *i.e.*, hardware option and not a software option.

On the new COP424C, the COP444C, and the COP484, IN2 may be mask programmed to be an input to the time base counter. Again, this is a hardware option and is not software alterable. On the COP440/441 and COP2440/2441 devices, IN2 may also be selected as an input to the time base counter. On these devices, however, the choice is controlled in software by the programmer.

2.4.2 Bidirectional Tri-State I/O

All COPS microcontrollers have at least one eight-bit bidirectional I/O port. This is the L port. In output operations, the L lines output the contents of the eight-bit Q register. The input path is from the pins to the accumulator and RAM. Note that the L lines are drives only: they do not retain any data. Output data for the L port is stored in the Q register. The L drivers can be placed in the high impedance, or TRI-STATE mode for ease in interface to a system bus.

The COP440 and COP2440 have and additional eight-bit bidirectional I/O port, the R port. The R port contains latches and drivers. Data to be output is latched into the R register. The input path is from the pins to the accumulator and RAM. Input data at the R pins is not, and cannot be, latched into the R register by any external signal. This must be done indirectly by the program. The R drivers, like the L drivers, can be put into a high impedance, or TRI-STATE, condition for simple bus interface.

Both the L port and the R port can be inputs. There is no input state per se. If used as inputs, either port may be put into a high impedance, or TRI-STATE, condition. In this case, the external signal must drive the line both high and low and guarantee the valid "0" and "1" logic levels. Alternatively, for both ports, the Q register or the R register can serve as a pullup for the L and R lines respectively. The programmer may write "1's" to the input positions and enable the drivers. In this case, the external signal need only pull the line down to a valid low level.

2.4.3 Bidirectional I/O

The G port is a four-bit bidirectional I/O port. The G outputs are latches and drivers. Therefore, data can be saved in the G port. The input path is from the pins to the accumulator. In addition to reading the port, the G lines can be directly tested, either individually or as a four-bit group, in software. Note, the latches on G are for output only; input signals are not latched into the G port.

The COP440 and COP2440 devices have an additional bidirectional four-bit port, the H port. The H port is essentially a duplicate of the G port except that H cannot be directly tested.

There is no restriction on H or G as to which lines may be inputs or outputs. All G lines may be inputs; all G lines may be outputs; any G line, or group of G lines, may be outputs with the remaining G lines inputs. The same is true of the H lines.

2.4.4 Outputs

The D port is an output-only port. The outputs are latched. On software command, the contents of Bd, the digit address portion of the RAM address register, are copied to the D port. These outputs will remain in that state until the next write to D. The D port is loaded only from Bd.

2.4.5 The SIO Register

The SIO register is a dual-purpose four-bit register. Depending on the status of the EN register, whose contents are user alterable, this register may be a four-bit binary down counter or a four-bit serial shift register. When SIO is a down counter, SI is the counter input, the counter decrements on the high to low transition, provided that the input remains low for two instruction cycles of the signal at the SI output. SO and SK are logic level outputs which can be directly controlled by the program. When SIO is a shift register, SI is the input to the 4-bit shift register and SO is the shift register output. SK is a serial clock running at the instruction cycle rate. By means of the EN register, and while SIO remains enabled as a shift register, SO can be disabled, *i.e.*, forced to zero. Similarly SK can also be forced to zero in this mode. Note that when SIO is enabled as a shift register and SO enabled as a shift register output, whatever is at SI will appear at SO four instruction cycles later unless the program alters the contents of SIO. When enabled as a shift register, SIO is always shifting at the instruction cycle rate regardless of the status of SO or SK.

MICROWIRE™ I/O

The MICROWIRE concept provides a simple, easy to use serial interface between COPS microcontrollers and various peripheral devices. The MICROWIRE interface is, essentially, the serial I/O port on COPS microcontrollers, the SIO register in the shift register mode. SI is the shift register input, the serial input line to the microcontroller. SO is the shift register output, the serial output line to the peripherals. SK is the serial clock, data is clocked into or out of peripheral devices with this clock. MICROWIRE is available on all COPS microcontrollers.

MICROWIRE Peripherals

For MICROWIRE interface, a peripheral device requires some or all of the following:

DI Data Input. This is the serial input to the peripheral. This is connected to SO on the microcontroller. All MICROWIRE peripherals **must** have this pin.

SK Serial Clock. This is the serial clock connected to SK of the microcontroller. All MICROWIRE peripherals **must** have this pin.

CS Chip Select. This merely selects a particular device. It may be connected to any convenient microcontroller output. Chip Select is required in any multiple peripheral systems. In a single peripheral system, whether or not Chip Select must be connected to a microcontroller output depends on the peripheral itself and its design.

DO Data Output. This is the serial output from the peripheral. It is connected to SI of the microcontroller. DO is required only on peripherals that communicate back to the microcontroller.

2.4.6 Microbus™

Microbus is a universal eight-bit parallel system bus. Certain COPS microcontrollers have a mask option permitting them to be used as Microbus-compatible peripheral devices. As far as the COPS device is concerened, the Microbus is composed of the following elements:

- An eight-bit bidirectional data bus
- Data Strobes - a read strobe and a write strobe
- Chip Select - to identify the device
- Interrupt/Acknowledge - return line to main CPU

In COPS microcontrollers, the data bus is the Q register-L drivers combination. If the device is selected and a write strobe occurs, data is transferred from the bus-L directly into the Q register. Similarly, if the device is selected and a read strobe occurs, data is copied from the Q register onto the bus-L. Input IN_1 becomes $\overline{RD}$, the read strobe. Input IN_2 becomes CS, the chip select. Input IN_3 becomes WR, the write strobe. Note that these three inputs are all active low. A logical "0" on $\overline{CS}$ (IN_2) selects the COPS device and enables operation of $\overline{RD}$ and $\overline{WR}$. A logical "0" on $\overline{RD}$ (IN_1) or $\overline{WR}$ (IN_3) when $\overline{CS}$ is also a logical "0" will cause the data read or write as described above. I/O pin G0 serves as an interruppt/acknowledge or ready pin back to the main CPU. G0 is normally high-ready. It is set high by the user program. The occurrence of a write strobe while the device is selected automatically sets G0 to the low or busy state. The user program sets G0 high again.

The Microbus option on COPS microcontrollers is completely compatible with the Microbus standard. The timing and timing relationships are those defined by that standard.

The Microbus option is a mask option, *i.e.*, a hardware option. The functions of IN_1, IN_2, IN_3, G0, and L drivers and the Q register are physically altered by this option. The Microbus option is available on the following COPS microcontrollers only: COP420, COP424C, COP444C, COP440, COP441, COP2440, and COP2441.

2.5 THE ENABLE REGISTER

The ENABLE (EN) register is an internal four- or eight-bit register loaded under program control. The state of the individual bits of this register selects or deselects certain features in the microcontroller.

2.5.1 EN_0 through EN_3

These four bits of the EN register are present on all COPS microcontrollers. Their function is as follows:

EN_0, the least significant bit of the enable register, controls the status of the SIO register. With EN_0 set, a logical "1", the SIO register is a four-bit asynchronous binary down counter decrementing its value by one upon each low going pulse at the SI input. The pulse must be low at least two instruction cycles. With EN_0 equal to "1", SO and SK are logic signals. SK outputs the value of SKL. SO outputs the value of EN_3. With EN_0 reset (low), the SIO register is a four-bit serial shift register that shifts left, from SI toward SO, one bit each instruction cycle time. Data is shifted into the least significant bit of SIO from SI. SO can be enabled to output the most significant bit of SIO. With EN_0 reset, SK becomes a logic controlled clock whose period is the instruction cycle time.

EN_1 controls the interrupt. With EN_1 set, the interrupt is enabled. If a signal meeting the timing requirements appears at the interrupt input when EN_1 is set, the interrupt will be recognized. With EN_1 reset, the interrupt is disabled, the signal at the interrupt input is ignored. Obviously, the status of EN_1 is significant only in those COPS microcontrollers having interrupt capability.

EN_2 controls the L drivers. With EN_2 set, the L drivers output the data in the Q register to the L I/O port. With EN_2 reset, the L drivers are disabled thereby placing the L I/O port into a high impedance, or TRI-STATE, condition. EN_2 has no effect on the L drivers in devices that have the Microbus option implemented.

On the COP440, COP441, COP2440, and COP2441 devices which have the Microbus option selected, EN_2 serves a different function. In this case, EN_2 set will disable any writing, by the program, into G0 which is the ready signal back to the main CPU.

EN_3, in conjunction with EN_0, controls the SO output. As stated above, if EN_0 is set, SO outputs the value of EN_3. If EN_0 is reset and EN_3 is set, SO is the output of the SIO serial shift register. If EN_0 is reset and EN_3 is reset SO is set to logical "0". SIO remains a shift register shifting data in from SI; SO is merely held low by internal logic. Table 2-2 provides a summary of the SIO modes associated with EN_0 and EN_3.

2.5.2 EN_4 through EN_7

These "extra" four bits of the enable register are present only in the following devices: COP440, COP441, COP442 COP2440, COP2441, and COP2442. Obviously, therefore, the information in this section applies to those devices **only**.

EN_4 - In conjunction with EN_5, EN_4 selects the interrupt source. See Table 2-3.

EN_5 - In conjunction with EN_4, EN_5 selects the interrupt source. See Table 2-3.

EN_6 - With EN_6 set (high), IN_2 becomes the input to the internal eight-bit T counter. With EN_6 reset (low), the input to the eight-bit T counter is the output of a divide by 4 prescaler from the instruction cycle frequency, thus providing a ten-bit time base counter.

On the COP442 and COP2442, IN_2 is not available as an input. Therefore, on these devices, EN_6 functions as a T counter disable: EN_6 set disables further counting, and EN_6 reset produces the ten-bit time base counter.

EN_7 controls the R I/O port. With EN_7 set, the contents of the R register are output to the R I/O port. With EN_7 reset, the R I/O port is placed into a TRI-STATE, or high impedance, condition. The contents of the R register are not affected.

TABLE 2-2. EFFECTS OF EN_3, EN_0, ON SIO, SI, SO, AND SK

EN_3	EN_0	SKL	SIO	SI	SO	SK
0	0	0	Shift Register	Input to Shift Register	0	0
0	0	1	Shift Register	Input to Shift Register	0	Clock
1	0	0	Shift Register	Input to Shift Register	Serial Out	0
1	0	1	Shift Register	Input to Shift Register	Serial Out	Clock
0	1	0	Binary Down Counter	Input to Binary Counter	0	0
0	1	1	Binary Down Counter	Input to Binary Counter	0	1
1	1	0	Binary Down Counter	Input to Binary Counter	1	0
1	1	1	Binary Down Counter	Input to Binary Counter	1	1

NOTE: SKL not affected by EN_3 or EN_0, but SKL does affect SK status.

TABLE 2-3. INTERRUPT SOURCE SELECTION

EN_5	EN_4	INTERRUPT SOURCE
0	0	IN_1 - low going pulse
0	1	CKO input (if CKO input option mask programmed)
1	0	Zero Crossing on IN_1 (or IN_1 level transition)
1	1	T counter overflows

2.6 INTERNAL TIMER

All COPS microcontrollers except the COP410L, COP411L, COP413L, COP410C, and COP411C have an internal time base counter. This counter is in the form of a ten-bit counter with the input being the instruction cycle frequency. Thus, this counter divides the instruction cycle frequency by 1024 or overflows once every 1024 instruction cycle times. A timer latch is set every time the counter overflows. This latch may be tested and reset (a single instruction) by the user's program.

2.6.1 Access to the Timer

All COPS microcontrollers that have the time base counter have the ability to test and reset the timer latch. Some devices, however, also have the ability to read and write the upper eight bits (the T counter) of the timer. The devices with this capability are as follows: The COP424C, COP425C, COP426C, COP444C/445C, COP440/441/442, COP2440/2441/2442, and COP484/485, and their associated ROMless devices. The timer overflow latch is still present and is still set when the counter overflows. These devices allow the user to modify, under program control, the overflow rate of the time base counter.

2.6.2 External Event Counter

On some devices, the COP424C, COP444C, COP440/441, COP2440/2441, and COP484, the upper eight bits or the T counter of the time base counter may be disconnected from the instruction cycle clock and connected to input IN_2. In this mode, the T counter counts external pulses. The timer overflow latch is set whenever the T counter overflows. The latch is tested in the normal manner. This characteristic is a mask option on the COP424C, COP444C, and COP484 devices. Thus, on these devices, the T counter may be connected to form the ten-bit time base counter or the T counter may be connected to IN_2 to count external events. On the COP440/441 and COP2440/2441 devices, this characteristic is a software option. The user's program controls the connection of the T counter via EN_6. There is no restriction, in these devices, on changing the T counter connection during program execution. The user is free to alternate between a time base counter and an external event counter if doing so is useful in his or her application.

2.7 OSCILLATOR AND BASIC TIMING

2.7.1 Clock Generator and Divider

The clock generator on COPS microcontrollers is extremely versatile and, by means of mask options, will work with a variety of oscillators: crystal, external, simple RC, or more involved RC, RLC, or LC networks. Furthermore, the clock generator will usually operate over a fairly large range in order to give the user maximum flexibility in selecting the oscillator frequency. Several divider (prescaler) options are available, as mask options, to insure that the COPS microcontroller is operating within its valid range with the oscillator frequency being used. See the various device data sheets for precise details regarding the oscillator frequency, clock generator, and divider.

2.7.2 The Instruction Cycle

The instruction cycle frequency is the frequency after the divider or prescaler. The period of this frequency, or the instruction cycle time, is the basic timing reference in COPS microcontrollers. Minimum pulse widths, for counter inputs, interrupt, etc., are expressed in terms of instruction cycle times. The highest degree of resolution with which a COPS microcontroller can read input pulses or generate output pulses is the instruction cycle time.

The instruction cycle time or frequency can be measured by the user. The period of the SK output when the microcontroller is reset ($\overline{\text{RESET}}$ low) or when SK is enabled as a clock output is the instruction cycle time.

2.8 INITIALIZATION

On power up, providing the timing parameters in the data sheets are met, the following registers are cleared on all COPS microcontrollers: A, B, C, D, EN, G, and the program counter PC. The SK latch, SKL, is set on all devices. In addition the T counter is cleared on the COP440/COP2440 series, the COP424C series, the COP444C series, and the COP484 series devices. In the COP440/COP2440, and COP484 series devices, the IL register and the Q register are also cleared. (Note that these two registers are not cleared on other devices.) The R, H, and N registers are also cleared on reset in the COP440/COP2440 series devices.

Reset, or initialization, occurs on power up and whenever a logical "0" at least three instruction cycles wide appears at the $\overline{\text{RESET}}$ input. On the COP440/COP2440 series, the COP484 series, and the COP424C/COP444C series devices, the T counter is cleared within these three instruction cycle times. On other COPS microcontrollers, the logical "0" at the $\overline{\text{RESET}}$ input must be ten cycles wide to clear the time base counter. In this situation, the timer overflow latch is set.

The reset condition of COPS microcontrollers is as follows:

- The program counter, PC, is set to 0.
- The accumulator, A, is 0.

- The RAM address register, B, is set to 0,0.
- The carry, C, is set to 0.
- The D register and D output port are set to 0.
- The enable register is set to 0 and SKL set to 1.
    1. SIO is a shift register.
    2. SI is shift register input.
    3. SO is 0.
    4. SK is clock output.
    5. Interrupts are disabled.
    6. The L port is put into a high impedance, or TRI-STATE condition.
- The G output port is set to 0.
- On the COP440/COP2440 series, the following is also true:
    1. The Q register is set to 0.
    2. The H register and I/O port are set to 0.
    3. The R register is set to 0.
    4. The R I/O port is put into a high impedance, or TRI-STATE condition.
    5. The interrupt source is IN_1, low-going pulse.
    6. The T counter is cleared and connected to form the time base counter.
    7. The IL register is set to 0.
- The Q register and IL register are also cleared in COP464/COP484 series devices.

Chapter 3

THE COPS INSTRUCTION SET

3.1 BASIC CHARACTERISTICS

The instruction set of COPS microcontrollers is designed to take maximum advantage of the COPS dual-bus architecture. The COPS instruction set, merged with the COPS architecture, provides the user with the power, versatility, and efficiency to achieve the maximum function and capability in minimum memory.

Since COPS microcontrollers are not memory-mapped devices, most instructions do not have the burden of carrying some form of address field. Therefore, most instructions are one byte in length. This, in turn, increases program efficiency. ROM space is devoted to performing a function rather than pointing to the locations of various items.

It is quite common for a COPS instruction to contain a multiplicity of function. This obviously creates program efficiency by performing in a single instruction a number of functions that would otherwise require several instructions.

The test instructions, like most COPS instructions, do not contain an address. Therefore, a successful test causes the next instruction to be skipped. It is quite common for one or both of the instructions following the test to be jumps. More importantly, however, this skipping characteristic allows the programmer to do a number of "unusual" things. Tests without following jumps are common. B or A or other parameters can be altered in line without jumping by judicious use of the test instructions. Examples of this, and further details, are provided in Section 4. Furthermore, the skip feature has been "built into" a number of arithmetic functions thereby eliminating the need to make separate tests.

3.2 DETAILED INSTRUCTION DESCRIPTION

For purposes of discussion and explanation, the COPS instructions are loosely grouped into the following six categories:

1. Arithmetic/Logic Instructions
2. Transfer of Control Instructions
3. Memory Reference Instructions
4. Register Reference Instructions
5. Test Instructions
6. Input/Output Instructions

This section provides a detailed description of all COPS instructions. This description includes the following information:

- The instruction mnemonic.
- A written description of the instruction.
- The data or program flow associated with the instruction.
- The instruction opcode in hex and binary.
- The instruction execution time - expressed in instruction cycles.
- Skip conditions associated with the instruction.
- Any restrictions on the instruction or its use; any "special effects" of the instruction.
- The COPS microcontrollers which have or do not have the instruction.

For ease and simplicity of description, the COPS microcontrollers are divided into the following four groups:

Group 1 devices:	COP401L, COP410L, COP411L, COP413L, COP410C, COP411C
Group 2 devices:	COP402, COP402M, COP404L, COP404C, COP420, COP421, COP422, COP420L, COP421L, COP422L, COP424C, COP425C, COP444L, COP445L, COP444C, COP445C
Group 3 devices:	COP404, COP440, COP441, COP442, COP2404, COP2440, COP2442
Group 4 devices:	COP408, COP484, COP485, COP409

The following list defines the symbols used in the descriptions of the instructions.

A	4-bit accumulator
B	RAM address register
Br	Upper bits of B, register address
Bd	Lower 4 bits of B, digit address
C	1-bit carry register
D	4-bit data output port
EN	Enable register
G	4-bit register to latch data for G I/O port
H	4-bit register to latch data for H I/O port
IL	Two 1-bit latches associated with IN3 and IN0 inputs
IN	4-bit input port
IN_1 Z	Zero crossing input
L	8-bit TRI-STATE I/O port
M	4-bit contents of RAM addressed by B

N	Subroutine stack pointer
PC	ROM address register, program counter
Q	8-bit register to latch data for L I/O port
R	8-bit register to latch data for R I/O port
SIO	4-bit shift register and counter
SK	Logic controlled clock output
T	8-bit binary counter register
RAM(B)	4-bit contents of RAM addressed by B
RAM_N	Contents of RAM location addressed by stack pointer N
ROM(t)	Contents of ROM location addressed by t
$PC_{a:b}$	Bits a through b of program counter PC

3.2.1 Arithmetic/Logic Instructions

ASC

Binary add, with carry, the accumulator with the memory location specified by the B register. The result is placed in the accumulator. If result $> 15_{10}$, generate a skip.

A <– A+C+RAM(B) C: Set or reset according to carry from bit three

Hex Code	7	6	5	4	3	2	1	0
30	0	0	1	1	0	0	0	0

Execution Time: 1 Instruction Cycle

Skip Conditions: If 1 -> C, skip

Restrictions: None

Availability: All COPS microcontrollers

ADD

Binary add the accumulator with the memory location specified by the B register. Result is placed in the accumulator.

A <– A+RAM(B)   C: Not used or affected

Hex Code	7	6	5	4	3	2	1	0
31	0	0	1	1	0	0	0	1

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: None

Availability: All COPS microcontrollers

ADT

Binary add 10 to the accumulator. Instruction used for decimal adjust.

A <–A+10_{10} C: Not used or affected

Hex Code	7	6	5	4	3	2	1	0
4A	0	1	0	0	1	0	1	0

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: None

Availability: Not available on: Group 1 and Group 4 devices

AISC y

Binary add the immediate value y to the accumulator and place the result in the accumulator. Generate a skip if there is a carry out of bit 3.

A <– A+y   C: Not used or affected

Hex Code	7	6	5	4	3	2	1	0
5y	0	1	0	1	y_3	y_2	y_1	y_0

Execution Time:	1 Instruction Cycle
Skip Conditions:	If carry from bit 3, skip
Restrictions:	$y \neq 0, 0 < y \leqslant FH$
Availability:	All COPS microcontrollers

CASC

Binary add, with carry, of the one's complement of the accumulator with the data in the memory location specified by the B register. Generate a skip if result $> 15_{10}$. This is the basic subtract instruction.

$\overline{A} \leftarrow A+RAM(B)+C$ C: Set or reset according to carry from bit three

Hex Code	7	6	5	4	3	2	1	0
10	0	0	0	1	0	0	0	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	If 1 -> C, skip
Restrictions:	None
Availability:	Not available on: Group 1 devices

CLRA

Clear the accumulator.

$A \leftarrow 0$ C: Not affected

Hex Code	7	6	5	4	3	2	1	0
00	0	0	0	0	0	0	0	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

COMP

Replace the value in A with its one's complement.

$A <- \overline{A}$ C: Not affected

Hex Code	7	6	5	4	3	2	1	0
40	0	1	0	0	0	0	0	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

NOP

No operation. C: Not affected

Hex Code	7	6	5	4	3	2	1	0
44	0	1	0	0	0	1	0	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

OR

Logical OR of accumulator with contents of memory location specified by the B register. Result in accumulator.

A <– A v Ram(R)   C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
1A	0	0	0	1	1	0	1	0

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	Group 3 devices

RC

Reset/clear the one-bit carry register

C <– 0   A: Not affected

Hex Code	7	6	5	4	3	2	1	0
32	0	0	1	1	0	0	1	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

SC

Set the one-bit carry register.

C <– 1 A: Not affected

Hex Code	7	6	5	4	3	2	1	0
22	0	0	1	0	0	0	1	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

XOR

Exclusive OR, bit by bit, of accumulator with contents of memory location specified by the B register. Result placed in accumulator.

A <– A $\oplus$ RAM(R) C: Not affected

Hex Code	7	6	5	4	3	2	1	0
02	0	0	0	0	0	0	1	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

3.2.2 Transfer of Control Instructions

JID

Jump Indirect. This involves a two-step modification of the program counter. First, load the lower eight bits of the program counter with the contents of the accumulator (upper four bits) and the memory location specified by the B register. The data addressed by this modified program counter is then loaded into the lower eight bits of the program counter. Execution continues at this second address.

(1) PC	<–PC+1	C: Not affected	
(2) $PC_{7:0}$	<–A,RAM(B)	A: Not affected	
(3) PC	<–PC+1		
(4) $PC_{7:0}$	<–ROM ($PC_{10:8}$,A,RAM(B))		

Hex Code	7	6	5	4	3	2	1	0
FF	1	1	1	1	1	1	1	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	JID at last word of block looks to next block for vector addresses (step one above). Vector address at last word of block points into next block (Step 3, 4 above).
Availability:	All COPS microcontrollers

JMP a

Jump Direct. Load the program counter (lower 11 bits) with the address specified in the instruction. Continue program execution at this address.

$PC_{10:0}$ <– a

C: Not affected
A: Not affected

Hex Code	7	6	5	4	3	2	1	0
6-	0	1	1	0	0	a_{10}	a_9	a_8
–	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	a_{10}=0, a_9=0 Group 1 devices a_{10}=0 in 1K devices JMP in last two words of chapter jumps to next chapter
Availability:	All COPS microcontrollers

JMPL a

Long Jump Direct. Load the program counter with the address as specified in the instruction. Continue program execution at this address.

PC <-- a

C: Not affected
A: Not affected

Hex Code	7	6	5	4	3	2	1	0
4A	0	1	0	0	1	0	1	0
–	1	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8
–	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Execution Time:	3 Instruction Cycles
Skip Conditions:	None
Restrictions:	a_{14}= 0, a_{13}= 0, a_{12}= 0 in COP484, COP485
Availability:	Group 4 devices

JP a

Jump within Page.

(1) PC <-- PC+1
(2) $PC_{6:0}$ <-- a - pages 2, 3 only
or
(2) $PC_{5:0}$ <-- a - all other pages

C: Not affected
A: not affected

Hex Code	7	6	5	4	3	2	1	0
—	1	a_6	a_5	a_4	a_3	a_2	a_1	a_0

(above for pages 2,3 only)

1	1	a_5	a_4	a_3	a_2	a_1	a_0

(all other pages)

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: May not JP to last word of a page. JP in last word of a page jumps to next page (Step 1 above).

Availability: All COPS microcontrollers

JSRP a

Jump to subroutine within Page 2.

1) PC <– PC+1
2) SB <– SA <– PC — Group 1 devices
or
2) SC <– SB <– SA <–PC — Group 2 devices
or
2) RAM_N <– PC — Group 3 and Group 4 devices
   N <– N+1
3) $PC_{5:0}$ <– a — address within Page 2
   $PC_{8:6}$ <– 010 — load Page 2
   $PC_{\text{all other}}$ <– 0

Hex Code	7	6	5	4	3	2	1	0
—	1	0	a_5	a_4	a_3	a_2	a_1	a_0

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: May not be used within Pages 2 and 3
May not JSRP to last word of Page 2

Availability: All COPS microcontrollers

JSR a

Jump to subroutine direct. Load lower 11 bits of the program counter with the address a. Push the subroutine stack. Continue execution at the address specified by the instruction.

1) PC <– PC+2
2) SB <– SA <– PC Group 1 devices

or

2) SC <– SB <– SA <– PC Group 2 devices

or

2) RAM_N <– PC; N <– N+1 Group 3 and Group 4 devices
3) $PC_{10:0}$ <– a

Hex Code	7	6	5	4	3	2	1	0
6-	0	1	1	0	1	a_{10}	a_9	a_8
–	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: a_{10}= 0, a_9= 0 in Group 1 devices
a_{10}= 0 in 1K devices
JSR in last two words of chapter calls subroutine in next chapter.

Availability: All COPS microcontrollers

JSRL a

Long jump to subroutine direct. Load the program counter with the address a. Continue execution at this address. Push the subroutine stack.

1) PC <– PC+3
2) RAM_N <– PC, N <– N+1
3) PC <– a

Hex Code	7	6	5	4	3	2	1	0
4A	0	1	0	0	1	0	1	0
–	1	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8
–	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Execution Time:	3 Instruction Cycle
Skip Conditions:	None
Restrictions:	a_{14}= 0, a_{13}= 0, a_{12}= 0 in COP484 and COP485
Availability:	Group 4 devices

RET

Return from subroutine and return control to the main program at the instruction following the JSR, or JSRP, or JSRL.

PC <– SA <– SB Group 1 devices

or

PC <– SA <– SB <–SC Group 2 devices

or

N <– N-1 Group 3 and Group 4 devices
PC <– RAM_N

Hex Code	7	6	5	4	3	2	1	0
48	0	1	0	0	1	0	0	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

RETSK

Return from subroutine. Return control to the main program and always skip the instruction following the JSR, JSRP, or JSRL.

PC <– SA <– SB   Group 1 devices

or

PC <– SA <– SB <– SC   Group 2 devices

or

N <– N-1, PC <– RAM_N Group 3 and Group 4 devices

Hex Code	7	6	5	4	3	2	1	0
49	0	1	0	0	1	0	0	1

Execution Time: 1 Instruction Cycle

Skip Conditions: Always skip on return

Restrictions: None

Availability: All COPS microcontrollers

HALT

Stop all internal operation of the device. Retain all internal status. Resume operation as result of external stimulus.

A, B, C, PC, G, L, Q, EN, RAM, T: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
38	0	0	1	1	1	0	0	0

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	Requires Hardware external restart
Availability:	COP410C, COP411C, COP424C, COP425C, COP426C, COP444C, COP445C, and COP404C

NOTE: This instruction places the eight microcontrollers mentioned above in their minimum power dissipation state.

IT

Stop all internal operation, except the timer, of the device. Resume operation at the instruction following IT when the timer overflows.

PC <– PC   A, B, C, G, L, Q, EN, PC, RAM: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
39	0	0	1	1	1	0	0	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	COP424C, COP425C, COP426C, COP444C, COP445C, COP404C

3.2.3 Memory Reference Instructions

CAME

Copy the eight-bit contents of A and the memory location addressed by the B register to the eight-bit enable register (Note: the enable register is eight bits long in COP440 and COP2440 series only). This is the inverse of the CEMA instruction in function and with respect to the four bits of the enable register with which A and RAM(B) communicate.

$EN_{7:4}$ <– $A_{3:0}$ A: Not affected

$EN_{3:0}$ <– $RAM(B)_{3:0}$ C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
1F	0	0	0	1	1	1	1	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	Group 3 devices

CAMQ

Copy the eight-bit contents of the accumulator and the memory location addressed by the B register to the eight-bit Q register. This is the inverse of the CQMA instruction in function and with respect to the four bits of Q with which A and RAM(B) communicate.

$Q_{7:4}$ <– $A_{3:0}$	A: Not affected
$Q_{3:0}$ <– $RAM(B)_{3:0}$	C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
3C	0	0	1	1	1	1	0	0

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

CAMT

Copy the eight-bit contents of the accumulator and the memory location addressed by the B register to the eight-bit timer register (T). This is the inverse of the CTMA instruction in function and with respect to the four bits of T with which A and M communicated.

$T_{7:4}$ <-- $A_{3:0}$	A: Not affected
$T_{3:0}$ <-- $RAM(B)_{3:0}$	C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
3F	0	0	1	1	1	1	1	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	Group 3 devices, COP424C, COP425C, COP426C, COP444C, COP445C, COP404C, Group 4 devices

CEMA

Copy the contents of the eight-bit enable register (COP440 and COP2440 series only) to the memory location addressed by the B register and to the accumulator. This is the inverse of the CAME instruction in function and with respect to the four bits of the enable register with which A and RAM(B) communicate.

$A_{3:0}$ <-- $EN_{3:0}$	C: Not affected
$RAM(B)_{3:0}$ <-- $EN_{7:4}$	

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
0F	0	0	0	0	1	1	1	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	Group 3 devices

CQMA

Copy the contents of the eight-bit Q register to the memory location addressed by the B register and to the accumulator. This is the inverse of the CAMQ instruction in function and with respect to the four bits of the Q register with which A and RAM(B) communicate.

$A_{3:0}$ <-- $Q_{3:0}$ C: Not affected

$RAM(B)_{3:0}$ <-- $Q_{7:4}$

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
2C	0	0	1	0	1	1	0	0

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	Not available on COP410L, COP411L, COP401L, COP410C, COP411C

CTMA

Copy the eight-bit contents of the timer register to the memory location addressed by the B register and to the accumulator. This is the inverse of the CAMT instruction in function and with respect to the four bits of T with which A and RAM(B) communicate.

$A_{3:0}$ <-- $T_{3:0}$ C: Not affected

$RAM(B)_{3:0}$ <-- $T_{7:4}$

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
2F	0	0	1	0	1	1	1	1

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: Group 3 devices, COP424C, COP425C, COP426C COP444C, COP445C, COP404C, Group 4 devices

LD n

Load the accumulator with the contents of the memory location addressed by the B register. Also, exclusive-OR the upper part of the B register (Br) with the n value.

A <-- RAM(B) C: Not affected
Br <-- Br ⊕ n

Hex Code	7	6	5	4	3	2	1	0
n5	0	0	n_1	n_0	0	1	0	1

n = 0, 1, 2, or 3

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: n = 0, 1, 2, 3 only

Availability: All COPS microcontrollers

LDD r,d

Load the accumulator with the contents of the memory addressed by the operand field r,d. The B register is not used or altered.

A <-- RAM(r,d) B: Not affected
C: Not affected

Hex Code	7	6	5	4	3	2	1	0
23	0	0	1	0	0	0	1	1
rd	0	r_2	r_1	r_0	d_3	d_2	d_1	d_0

r = 0:7; d = 0:15

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: r = 0, 1, 2, 3, 4, 5, 6, or 7 **only**

Availability: Not available in Group 1 devices

LID

Load the accumulator and the memory location addressed by the R register with the eight-bit ROM word addressed by the upper bits of the program counter, A and RAM(B).

PC <– PC+2 C: Not affected

RAM(B) <– $ROM(PC_{10:8},A,RAM(B))_{7:4}$

A <– $ROM(PC_{10:8},A,RAM(B))_{3:0}$

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
19	0	0	0	1	1	0	0	1

Execution Time: 3 Instruction Cycles

Skip Conditions: None

Restrictions: LID in last word of block will access next block (#1 above)

Availability: Group 3 devices

LQID

Load the Q register with the eight-bit ROM word addressed by the upper bits of the program counter, the accumulator and the memory location addressed by the B register.

PC <– PC+1	A: Not affected
$Q_{7:4}$ <– ROM($PC_{10:8}$,A,RAM(B))$_{7:4}$	C: Not affected
$Q_{3:0}$ <– ROM($PC_{10:8}$,A,RAM(B))$_{3:0}$	

Hex Code	7	6	5	4	3	2	1	0
RF	1	0	1	1	1	1	1	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	LQID in last word of a block accesses next block (#1 above). One level of subroutine stack is used by this instruction in Group 1 and Group 2 devices.
Availability:	All COPS microcontrollers

RMB 0, RMB 1, RMB 2, RMB 3

Reset the bit specified in the instruction in the memory location addressed by the B register.

RAM(B) n <– 0	C: Not affected
n = n,1,2,3	A: Not affected

		Hex Code	7	6	5	4	3	2	1	0
RMB	0	4C	0	1	0	0	1	1	0	0
RMB	1	45	0	1	0	0	0	1	0	1
RMB	2	42	0	1	0	0	0	0	1	0
RMB	3	43	0	1	0	0	0	0	1	1

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

SMB 0, SMB 1, SMB 2, SMB 3

Set the bit specified in the instruction in the memory location addressed by the B register.

RAM(B)n <– 1	C: Not affected
n = 0,1,2,3	A: Not affected

		Hex Code	7	6	5	4	3	2	1	0
SMB	0	4D	0	1	0	0	1	1	0	1
SMB	1	47	0	1	0	0	0	1	1	1
SMB	2	46	0	1	0	0	0	1	1	0
SMB	3	4B	0	1	0	0	1	0	1	1

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

STII y

Store the immediate value y into the memory location addressed by the B register. Then increment the lower four bits of the B register (Bd). The upper portion of the B register (Br) is not affected.

RAM(B) <– y	A: Not affected
Bd <– Bd + 1	C: Not affected

Hex Code	7	6	5	4	3	2	1	0
7y	0	1	1	1	y_3	y_2	y_1	y_0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

X n

Exchange the contents of the accumulator with the contents of the memory location addressed by the B register. Then replace Br with the exclusive OR of Br and n. Bd is not affected.

A <-> RAM(B) C: Not affected
Br <– Br ⊕ n

Hex Code	7	6	5	4	3	2	1	0
n6	0	0	n_1	n_0	0	1	1	0

n =0,1,2,3

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	n = 0,1,2, or 3 **only**
Availability:	All COPS microcontrollers

XAD r,d

Exchange the contents of the accumulator with the contents of the memory location addressed by r,d. The B register is not affected.

A <-> RAM(r,d) B: Not affected
C: Not affected

	Hex Code	7	6	5	4	3	2	1	0
Group 1 devices	23	0	0	1	0	0	0	1	1
	BF	1	0	1	1	1	1	1	1

All Others	23	0	0	1	0	0	0	1	1
	r,d	1	r_2	r_1	r_0	d_3	d_2	d_1	d_0

r = 0:7; d = 0:15

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: On Group 1 devices; r=3, d = 15 **only**
All other COPS microcontrollers:
r = 0,1,2,3,4,5,6, or 7 **only**

Availability: All COPS microcontrollers

XDS n

Exchange the contents of the accumulator with the contents of the memory location addressed by the B register. Replace Br with the exclusive OR of Br and n. Decrement Bd by 1. Generate a skip if Bd decrements from 0 to 15.

A <--> RAM(B) C: Not affected
Br <-- Br ⊕ n
Bd <-- Bd - 1

Hex Code	7	6	5	4	3	2	1	0
n7	0	0	n_1	n_0	0	1	1	1

n = 0,1,2, or 3

Execution Time: 1 Instruction Cycle

Skip Conditions: Generate a skip if Bd-1 = 15

Restrictions: n = 0,1,2, or 3 **only**

Availability: All COPS microcontrollers

XIS n

Exchange the contents of the accumulator with the contents of the memory location addressed by the B register. Replace Br with the exclusive OR of Br and n. Increment Bd by one. Generate a skip if Bd increments from 15 to 0.

A <--> RAM(B)	C: Not affected
Br <-- Br $\oplus$ n	
Bd <-- Bd + 1	

Hex Code	7	6	5	4	3	2	1	0
n4	0	0	n_1	n_0	0	1	0	0

n = 0,1,2,3

Execution Time:	1 Instruction Cycle
Skip Conditions:	Generate a skip if Bd+1 = 0
Restrictions:	n = 0,1,2, or 3 only
Availability:	All COPS microcontrollers

3.2.4 Register Reference Instructions

CAB

Copy the contents of the accumulator to the lower four bits of the B register.

Bd <-- A	A: Not affected
	C: Not affected
	Br: Not affected

Hex Code	7	6	5	4	3	2	1	0
50	0	1	0	1	0	0	0	0

Execution Time:	1 Instruction Cycle
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

CBA

Copy the lower four bits of the B register to the accumulator.

A <– Bd   C: Not affected
B: Not affected

Hex Code	7	6	5	4	3	2	1	0
4E	0	1	0	0	1	1	1	0

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: None

Availability: All COPS microcontrollers

LBI r,d

Load the B register immediate with the values r (to the upper portion of the B register). Skip all subsequent LBI instructions until an instruction that is not an LBI is encountered.

Br <– r   A: Not affected

Bd <– d   C: Not affected

Hex Code	7	6	5	4	3	2	1	0
r(d-1)	0	0	r_1	r_0	(d - 1)			

r = 0:3; d = 0, 9:15

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
rd	1	r2	r1	r0	d3	d2	d1	d0

r = 0:7; d = 0:15

33	0	0	1	1	0	0	1	1
7-	0	1	1	1	0	0	0	r_4
rd	r3	r2	r1	r0	d3	d2	d1	d0

r = 0:31; d = 0:15

Execution Time:
1 Instruction Cycle (One-byte form)
2 Instruction Cycles (Two-byte form)
3 Instruction Cycles (Three-byte form)

Skip Conditions: Skip until not an LBI

Restrictions:
One-byte form:
r = 0,1,2,3 **only**
d = 0,9,10,11,12,13,14,15 **only**
Two-byte form: r = 0,1,2,3,4,5,6,7 **only**
Three-byte form: None

Availability:
One-byte form: All COPS microcontrollers
Two-byte form: Not available on Group 1 devices
Three-byte form: Available on Group 4 devices **only**

LEI y

Load the enable register (lower four bits on COP440 and COP2440 series) with the immediate value y.

$EN_{3:0}$ <– y

A: Not affected
C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
6y	0	1	1	0	y_3	y_2	y_1	y_0

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: In COP2440, 2441, 2442, processor Y loads EN_2 **only**

Availability: All COPS microcontrollers

XABR

Exchange the contents of the accumulator with the contents of the upper part of the B register (Br). If Br is less than four bits wide, zeroes are placed in the corresponding bits of the accumulator.

Br <–> A, A_3 <– 0, A_2 <– 0	Devices with 64 or 32 RAM digits
Br <–> A, A_3 <– 0	COP404L, COP404C, COP444L, COP445L, COP444C, COP445C
Br <–> A	Group 3 and Group 4 devices

Hex Code	7	6	5	4	3	2	1	0
12	0	0	0	1	0	0	1	0

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: None

Availability: Not available on Group 1 devices

XABX

Exchange the contents of the accumulator with the contents of the upper part of the Br register. Zeroes are placed in the upper bits of the accumulator.

Br_{upper} <--> A, A_3 <-- 0, A_2 <-- 0, A_1 <-- 0

C: Not affected
Br_{lower}: Not affected
Bd: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
1D	0	0	0	1	1	1	0	1

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: COP409 only

XAN

Exchange the contents of the accumulator with the contents of the two-bit subroutine stack pointer. The lower two bits of A go into the stack pointer and the same two bits of A are loaded with the pointer value. The upper two bits of A are cleared.

$A_{1:0}$ <-->N

C: Not affected

A_2 <-- 0, A_3 <-- 0

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
0B	0	0	0	0	1	0	1	1

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: Group 3 devices

3.2.5 Test Instructions

SKC

If the one-bit carry register (C) is equal to "1", skip the next program instruction.

A: Not affected
C: Not affected

Hex Code	7	6	5	4	3	2	1	0
20	0	0	1	0	0	0	0	0

Execution Time: 1 Instruction Cycle

Skip Conditions: Skip if C = 1

Restrictions: None

Availability: All COPS microcontrollers

SKE

If the contents of the accumulator are equal to the contents of the memory location addressed by the B register, skip the next program instruction.

A: Not affected
C: Not affected

Hex Code	7	6	5	4	3	2	1	0
21	0	0	1	0	0	0	0	1

Execution Time: 1 Instruction Cycle

Skip Conditions: Skip if A = RAM(B)

Restrictions: None

Availability: All COPS microcontrollers

SKGZ

If all four G lines are low ("0"), skip the next program instruction.

A: Not affected
C: Not affected
G: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
21	0	0	1	0	0	0	0	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	Skip if $G_{3:0} = 0$
Restrictions:	None
Availability:	All COPS microcontrollers

SKGBZ n, 7 = 0,1,2,3

If GN is zero, skip the next program instruction.

A,C,G: Not affected

		Hex Code	7	6	5	4	3	2	1	0
SKGBZ	0	33	0	0	1	1	0	0	1	1
		01	0	0	0	0	0	0	0	1
SKGBZ	1	33	0	0	1	1	0	0	1	1
		11	0	0	0	1	0	0	0	1
SKGBZ	2	33	0	0	1	1	0	0	1	1
		03	0	0	0	0	0	0	1	1
SKGBZ	3	33	0	0	1	1	0	0	1	1
		13	0	0	0	1	0	0	1	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	Skip if specified G bit is zero
Restrictions:	None
Availability:	All COPS microcontrollers

SKMBZ n n = 0,1,2,3

If the specified bit in the memory location addressed by the B register is "0", skip the next program instruction.

A,C,RAM(B): Not affected

	Hex Code	7	6	5	4	3	2	1	0
SKMBZ 0	01	0	0	0	0	0	0	0	1
SKMBZ 1	11	0	0	0	1	0	0	0	1
SKMBZ 2	03	0	0	0	0	0	0	1	1
SKMBZ 3	13	0	0	0	1	0	0	1	1

Execution Time: 1 Instruction Cycle

Skip Conditions: Skip if RAM(B)n = 0

Restrictions: None

Availability: All COPS microcontrollers

SKSZ

If the four-bit serial input/output register is "0", skip the next program instruction.

A,C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
1C	0	0	0	1	1	1	0	0

Execution Time: 2 Instruction Cycles

Skip Conditions: Skip if SIO = 0

Restrictions: None

Availability: Group 3 devices

SKT

If T counter carry (overflow) has occurred since the last test (last SKT), skip the next program instruction. Reset the SKT latch. (Timer carry/overflow sets SKT latch. SKT instruction tests and resets this latch).

SKTL <– 0 A,C,T: Not affected

Hex Code	7	6	5	4	3	2	1	0
41	0	1	0	0	0	0	0	1

Execution Time: 1 Instruction Cycle

Skip Conditions: Skip if SKTL = 1

Restrictions: None

Availability: Not available on Group 1 devices

3.2.6 Input/Output Instructions

CAMR

Copy the contents of the accumulator and the memory location addressed by the B register to the eight-bit R port. This is the inverse of the INR instruction in function and with respect to the four bits of R which are accessed by A and RAM(B).

$R_{7:4}$ <– A A: Not affected

$R_{3:0}$ <– RAM(B) C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
3D	0	0	1	1	1	1	0	1

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: Group 3 devices

NOTE: On COP2441, COP2442, COP441, and COP442, R as I/O port is not present, but R as eight-bit internal register is available.

ING

Copy the status of the G I/O port into the accumulator.

A <– G

C: Not affected
G: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
2A	0	0	1	0	1	0	1	0

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	All COPS microcontrollers

INH

Copy the status of the H I/O port to the accumulator.

A <– H

C: Not affected
H: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
2B	0	0	1	0	1	0	1	1

Execution Time:	2 Instruction Cycles
Skip Conditions:	None
Restrictions:	None
Availability:	Group 3 devices

NOTE: On COP2441, COP2442, COP441, and COP442, H as I/O port is not present, but H as four-bit internal register is available.

ININ

Copy the status of the four IN lines to the accumulator.

A <– IN   C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
28	0	0	1	0	1	0	0	0

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: COP420, COP420L, COP444L, COP484, COP2440, COP2441, COP440, COP441, COP424C, COP444C

INIL

Copy the status of the IL latches and CKO input and zero cross input (COP440, 441, COP2440, 2441) to the accumulator. Reset the IL latches.

1a) A 3:0 <– IL 3, CKO, IN 1Z, IL 0   COP440, COP441, COP2440, COP2441

or

1b) A 3:0 <– IL 3, CKO, "0", IL 0   COP420, COP420L, COP444L, COP484, COP424C, COP444C

or

1c) A 3:0 <– "0", CKO, "0", "0"   COP442, COP2442, COP421, COP422, COP421L, COP422L, COP445C, COP485, COP425C, COP426C

2) IL_3 <– 0, IL_0 <– 0

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
29	0	0	1	0	1	0	0	1

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: If CKO is not selected as general input, "1" is loaded into A_2. IL latches are reset at power on in Group 3 and Group 4 devices only. On other devices, the latches are undefined until first INIL

Availability: Not available on Group 1 devices

INL

Copy the status of the eight-bit L port to the memory location addressed by the B register and the accumulator.

RAM(B) $<-$ $L_{7:4}$ C: Not affected

A $<-$ $L_{3:0}$

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
2E	0	0	1	0	1	1	1	0

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: All COPS microcontrollers

INR

Copy the status of the eight-bit R port to the memory location addressed by the B register and the accumulator. This is the inverse of the CAMR instruction in function and with respect to the four bits of R which are accessed by A and RAM(B).

RAM(B) $<-$ $R_{7:4}$ C: Not affected

A $<-$ $R_{3:0}$

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
2D	0	0	1	0	1	1	0	1

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: Group 3 devices

NOTE: On COP2441, COP2442, COP441, and COP442, R as an I/O port is not present but R as eight-bit internal register is available.

OBD

Copy the contents of the lower four bits of the B register (Bd) to the D output port.

D <– Bd

A: Not affected
B: Not affected
C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
3E	0	0	1	1	1	1	1	0

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: All COPS microcontrollers

OGI y

Output the immediate value y to the four-bit G port.

G <– y

A: Not affected
C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
5y	0	1	0	1	y_3	y_2	y_1	y_0

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: Not available on Group 1 devices

OMG

Copy the contents of the memory location addressed by the B register to the four-bit G port.

G <– RAM(B)

A: Not affected
C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
3A	0	0	1	1	1	0	1	0

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: All COPS microcontrollers

OMH

Copy the contents of the memory location addressed by the B register to the four bit H port.

H <– RAM(B) A: Not affected

C: Not affected

Hex Code	7	6	5	4	3	2	1	0
33	0	0	1	1	0	0	1	1
3B	0	0	1	1	1	0	1	1

Execution Time: 2 Instruction Cycles

Skip Conditions: None

Restrictions: None

Availability: Group 3 devices

XAS

Exchange the contents of the accumulator with the contents of the SIO register. Copy the contents of the one-bit C register to the SK latch. This is the basic MICROWIRE interface instruction and is the primary control over the serial port.

A <—> SIO C: Not affected

SKL <– C

Hex Code	7	6	5	4	3	2	1	0
4F	0	1	0	0	1	1	1	1

Execution Time: 1 Instruction Cycle

Skip Conditions: None

Restrictions: On COP2440, COP2441, COP2442, processor X only may use XAS. Processor Y treats XAS as NOP.

Availability: All COPS microcontrollers.

3.3 NOTES ON ADDRESSING MODES

COPS microcontrollers do not have addressing modes in the sense of most popular microprocessors. To be sure, every instruction can be said to have some form of addressing mode associated with it. For example, a jump can be direct (JMP or JMPL), indirect (JID), or "modified relative" (JP); and adds can be immediate (AISC) or inherent/implied (ASC,ADD). A classification of this kind can be made, but it is awkward and forced; it is an attempt to impose the structure of one type of microcomputer on another type of microcomputer. Because of the difference in kind between these microcomputers, a comparison on the basis of number of addressing modes between COPS and some other microcomputer is not valid. One may be able to find six or seven kinds of addressing modes in the COPS instruction set, but such an effort is more an exercise of the imagination than a meaningful evaluation of the instruction set. Comparisons should be made on what the instruction set really requires, in terms of the relevant parameters, memory usage, and speed, to perform a given function.

Chapter 4

PROGRAMMING COPS MICROCONTROLLERS

4.1 INTRODUCTION

This section deals with all aspects of programming COPS devices. The concepts, structures, rules, suggestions, and tricks for COPS programming are discussed. The detailed effects of various instructions are also discussed.

4.2 BOUNDARY CONDITIONS

Although the program counter in COPS microcontrollers will increment linearly throughout the address space, three types of boundaries exist in the program space that the user should remember

- Page boundaries
- Block boundaries
- Chapter boundaries

Even though these boundaries exist, their impact on the actual programming is minimal. This is true because these boundaries are important in only a few instructions and even there the primary effect, in most cases, is to allow the user to use a more code efficient instruction.

4.2.1 Page Boundaries

A page is composed of 64 contiguous ROM words. Page 0 is the group of ROM words located at hex addresses 000 through 03F; Page 1 is the group of ROM words located at hex addresses 040 through 07F; etc. (See Table 2-1.) The page boundary saves code by allowing the use of the single-byte jump (JP) and the single-byte subroutine call (JSRP).

Furthermore, Pages 2 and 3 are the special subroutine pages. Page 2 is the destination page for subroutines called by JSRP instruction.

The JP Instruction

The JP instruction is the single-byte jump. It loads the lower six bits of the program counter only; therefore, it causes a jump within a page only. There is an exception to this, however. A JP instruction located at the last word of a page (hex addresses 03F, 07F, 0BF, 0FF, etc.) will cause a jump into the next page. In all COPS microcontrollers, the program counter is incremented before the execution of the instruction. Thus, the program counter will increment from hex address 13F, the last word of a page, to hex address 140, the first word of the next page; then the JP will load the lower six bits of the PC. The effect is to cause a jump from one page to the next page with the single-byte JP.

The JP instruction cannot be used to jump to the last word of a page. The reason for this is evident from an examination of the instruction OP codes. The two most significant bits of the JP instruction are 11. The lower six bits of the address of the last word of a page are all ones. Thus, the OP code of a JP to the last word of a page would be hex FF. This, however, is the opcode for the JID instruction. Therefore, JP cannot be used to jump to the last word of a page because the opcode that would otherwise implement that jump has been used to create the JID instruction.

The JP instruction has an expanded range within the subroutine pages – Pages 2 and 3. In these two pages only, the JP instruction loads the lower 7 bits of the program counter. Thus, a JP within Pages 2 and 3 may jump anywhere, except the last word of Page 2 or last word of Page 3, within Pages 2 and 3.

The JSRP Instruction

The JSRP instruction is the single-byte subroutine call. Page 2 is the destination page for the subroutine jump. The instruction indicates the address within Page 2 where the subroutine begins. The two restrictions on the use of JSRP are as follows:

1. JSRP to the last word of Page 2 is not allowed
2. JSRP may not be used within Page 2 or 3

The reason for both restrictions is evident from the opcodes. The most significant two bits of JSRP are 10. The lower six bits are the address within Page 2. Thus, JSRP to the last word of Page 2 would have the opcode hex BF. This opcode, however, has been used to implement the LQID instruction. Thus, a JSRP to hex address OBF, the last word of Page 2, is not allowed. JSRP may not be used within Pages 2 and 3 simply because the opcodes have been used to expand the range of the JP instruction as explained in Section 4.2. The sacrifice of the JSRP to expand JP in the subroutine pages helps to create more entry points in Page 2 which tends to increase program efficiency.

4.2.2 Block Boundaries

A block is composed of four contiguous pages or 256 contiguous ROM words. Block 0 consists of Pages 0 through 3; Block 1 consists of Pages 4 through 7; etc. (See Table 2-1.) The block boundary is significant only with respect to the indirect instructions: JID, LQID, and LID. These instructions operate within a block and do not normally cross block boundaries.

LQID and LID

These are the table look-up instructions. LQID looks up data identified by A and RAM(B) and puts the value in Q. LID does the same but returns the value to A and RAM(B). Hence, the look up is based on an eight-bit value. The lower eight bits of the program counter are temporarily replaced by the contents of A and RAM(B). The remaining bits of the PC are not affected by the instruction. Thus, these instructions work within a block.

Just as with the JP instruction, a special situation exists if the LQID is at the last word of a block of LID is at the last two words of a block. In this situation, the look up is performed in the next block. The reason is, as explained before, the program counter is incremented before the instruction is executed. Thus, the program counter will be in the next block before the look-up operation is performed.

The JID Instruction

The JID instruction looks up an address on the basis of A and RAM(B), then loads the lower eight bits of the program counter with that address. Again, since eight-bit values are being used, block boundaries are respected.

Since the program counter is incremented prior to instruction execution, a JID at the last word of a block will look up its address in the next block and execute the jump in that block. An additional related special case exists with the JID instruction. If the look-up address for the JID is at the last word of a block (*i.e.*, A = 15_{10} and RAM(B) = 15_{10}), then the jump will be in the next block. A final combination case exists: If JID is at the last word of a block and A = 15_{10} and B = 15_{10}, then the jump will be in the second block from the present block (see Table 4-1).

4.2.3 Chapter Boundaries

The Chapter is the largest memory division in COPS microcontrollers. A Chapter is composed of eight contiguous blocks (32 contiguous pages, 2048 contiguous ROM words). Obviously, the Chapter boundary has no relevance, in fact does not exist, if the microcontroller has fewer than 2048 words of program memory. Only the two-byte JMP and two-byte JSR are affected by the Chapter boundary. These instructions will jump anywhere within a Chapter or call a subroutine anywhere within a Chapter and will not normally cross a Chapter boundary. The exception is basically the same as seen before: a JMP at the last two words of a Chapter will jump to the next Chapter; a JSR at the last two words of a Chapter will call a subroutine in the next Chapter. The reason is the same: the program counter is incremented before the instruction is executed.

TABLE 4-1. EFFECTS OF BLOCK BOUNDARIES ON JID DESTINATION

JID LOCATION	A	RAM(B)	DESTINATION
Block N, anywhere except last word	$\neq 15$	$\neq 15$	Block N
Block N, anywhere except last word	15	$\neq 15$	Block N
Block N, anywhere except last word	$\neq 15$	15	Block N
Block N, anywhere except last word	15	15	Block N+1
Block N, last word	$\neq 15$	$\neq 15$	Block N+1
Block N, last word	15	$\neq 15$	Block N+1
Block N, last word	$\neq 15$	15	Block N+1
Block N, last word	15	15	Block N+2

4.3 SKIP CONDITIONS

In COPS microcontrollers, program address information is contained only in the jump and subroutine call instructions. Thus, decision instructions, or tests, do not contain a branch address. There is no single instruction equivalent of "If condition X is true (false) branch to address A." Instead, in COPS devices, if the test condition is met a skip is generated. This skip prohibits the execution of the following instruction, *i.e.*, "skipping" that instruction. The number of program bytes in the instruction has no bearing on the skip operation. Thus, following a test instruction with jumps or subroutine calls produces the desired branching. However, the skip feature allows much greater flexibility than merely branching. In many cases, the skip feature eliminates the need for branching since almost any register or variable parameters in COPS microcontrollers can be modified, in line, on the basis of a skip (see Section 4.7).

4.3.1 Effect of Skips on Timing Loops

Software timing loops are commonly part of a microcontroller program. In such a case, it is usually necessary that various paths through the loop take the same amount of time. The skip feature actually helps to achieve this goal rather than, as might be expected, conflicting with it. The reason is in the operation of the skip. If an instruction is to be skipped, the internal logic forces a NOP equal in length to the number of program bytes in the skipped instruction in place of that instruction. Then the NOP is executed. Thus, whether or not an instruction is skipped has no effect on the time to execute a given sequence of instructions. Note: this "hardware NOP" is temporary; it exists for the duration of the skipped instruction only and in no way alters the ROM contents. It therefore becomes a simple matter to compute execution time through a given sequence. Merely count the number of bytes, not instructions, in the path without regard to tests or skips and multiply by the instruction cycle time.

The Indirect Instructions - An Exception

The indirect instructions JID, LQID, and LID constitute a exception to this general rule. These are the only COPS instructions that require more instruction cycle times than the number of bytes in the instruction to execute. They require one more instruction cycle time than the number of bytes to execute: JID and LQID are one-byte instructions and require two instruction cycles to execute; LID is a two-byte instruction that requires three instruction cycles to execute. The result is that these instructions use one more instruction cycle when executed than when skipped because the hardware forced NOP is related to the number of bytes in the instruction rather than the execution time of the instruction. This distinction is significant only for these three instructions.

4.3.2 Instructions That Generate a Skip

As would be expected, all test instructions can generate a skip. If the test condition is met, a skip is generated. However, certain other instructions can also generate skips. The following arithmetic instructions generate a skip if the result of a four-bit binary addition is greater than 15_{10}: ASC, CASC, and AISC. The advantage here is that the common test after such instructions (testing carry or overflow) is built directly into the instruction thereby eliminating the need for a separate test instruction.

The LBI (load B register immediate) can also generate a skip. This instruction forces a skip until an instruction is reached that is not an LBI. This permits multiple entry points to a common routine without affecting the code. The code savings of this feature are more subtle, but this allows the user a degree of flexibility not found in other devices. Section 4.7 will explain this feature in more detail.

The XIS and XDS instructions can also generate skips. These generate a skip when one increments or decrements "off the end" of a register (Bd incrementing from 15 to 0 or decrementing from 0 to 15). This becomes very useful in loop operations as the need for testing for completion of the loop is often eliminated - another test is eliminated. Section 4.7 will illustrate the use of these instructions.

The final instruction that generates a skip is the RETSK. When executed, this instruction always forces a skip of the instruction located at the return address. This instruction becomes very valuable in implementing complex tests in a subroutine, or in reversing the direction of a frequently used test by means of a special subroutine. It is, of course, useful whenever the user wishes to force a skip of a subroutine return address.

4.4 CARRY

The ALU in COPS microcontrollers is a four-bit parallel binary adder. The user does not have access to the bit-to-bit carry within the ALU, but does have varying degrees of access to the carry as a result of a four bit operation. Within this category the user should be aware of several distinctions: the carry register, the carry out of the ALU, and simple arithmetic overflow. These are not always the same thing and the difference can be important. The carry register, C, may be set or reset directly by the program. Those instructions that do an "add with carry", ASC and CASC, use the C register in the addition. These same two instructions are the **only** instructions that load the carry out of the ALU, the carry as a result of the four-bit addition, into the C register. The SKC instruction test the status of the C register, not the carry from the ALU.

The carry from the ALU is the controlling factor in those arithmetic instructions that can generate a skip: ASC, CASC, AISC. If the carry from the ALU is a one as a result of any of these instructions, a skip is generated. The C register is not used for this form of skip generation. In fact, the AISC instruction neither uses nor affects the C register.

The ADD and ADT instructions cause an add to be performed. This add may well cause an arithmetic overflow. This overflow, however, is not quite the same as the carry from the ALU since no skip condition occurs. Furthermore, the C register is neither used nor affected.

This can be viewed as a hierarchy of overflows:

1. Simple arithmetic overflow; no skip; C neither used nor affected. ADD, ADT
2. Carry from the ALU (= arithmetic overflow which generates a skip); C neither used nor affected. AISC
3. Carry from the ALU that loads C (= arithmetic overflow which generates skip, C loaded with status of carry from ALU); C both used and affected. ASC, CASC

4.5 INPUT/OUTPUT

All input/output operations are handled by unique instructions. The instructions may be executed at any point in the program.

4.5.1 Unidirectional Ports

Two unidirectional ports are found in COPS microcontrollers: the IN input port and the D output port. The IN port is read by the ININ instruction. The IL latches, associated with the IN port, are read by the INIL instruction. Pin CKO may be configured as an input, via a mask option, on some devices. The INIL instruction also reads the state of the CKO input in those devices that have that option. See the descriptions of the ININ and INIL instructions for further details.

The D output is loaded from the lower four bits of the B register (Bd) by means of the OBD instruction. There is no path from the accumulator or RAM to the D port.

4.5.2 Bidirectional Ports

Non TRI-STATE Ports

There are two bidirectional, non TRI-STATE ports available: The G port, available, at least partially, on all COPS microcontrollers; and the H port, available on the COP440 and COP2440. The output function is simple; merely write the data to the port with the appropriate instruction: OGI, OMG, or OMH. Data is read via the ING or INH instruction. In addition, the G lines may be directly tested individually or as a four bit group. When using any of the G or H lines as inputs the user must write a "1" to the lines used as inputs. This is a requirement imposed by hardware rather than software considerations. The external circuitry will pull the line to logic "0".

On 20-pin COPS devices, only two of the four G lines are brought out. The other two lines, however, are available for internal use as flags or storage. The same is true of the H port on the COP441, COP442, COP2441, and COP2442.

Any G or H line, or any combination, may be used as inputs while the others are used as outputs. There is no conflict and the user has complete flexibility.

TRI-STATE Ports

Two eight-bit bidirectional TRI-STATE ports are available: The Q register-L drivers available on all COPS microcontrollers and the R port available on the COP440 and COP2440. The L port is written by loading Q with CAMQ or LQID and enabling L, via LEI or CAME. The application will determine if L should be enabled before or after loading Q or enabled all the time. The decision is not significant in terms of software. Remember, the L outputs are drivers **only**. They are not latched. When enabled, L outputs the contents of Q. L **must** be enabled in order to output data. The R port is a latched output port. The user writes to the R register by means of the CAMR instruction. The R drivers are enabled by means of the CAME instruction. In terms of software alone, it is not significant when the R drivers are enabled, but the drivers must be enabled to output the contents of the R register.

There are two ways to use these lines as inputs. The first method requires that the drivers be disabled. In this case, the lines are truly floating and in an undefined state. The external circuitry must provide good logic levels, both high and low, to the input pins. The inputs are then read by the INL or INR instructions. The second method is very similar to the technique used for G and H. The drivers are enabled. A "1" must be written to the Q or R register in the positions of the input lines. The external circuitry will then be required only to pull the line down to a logic "0". The line will pull itself up to a logic "1". The INL and INR instructions are used as before to read the lines.

Any L or R line, or any combination, may be used as inputs while the others are used as outputs. However, the L drivers are enabled or disabled as a group. The same is true of the R drivers. The L drivers are enabled or disabled by means of the LEI or CAME instructions. The R drivers are controlled by means of the CAME instruction only. On most devices, the Q register can be read without affecting L. The R register can be read only through the R lines. The data on the L lines does not affect the contents of the Q register except on devices with the MICROBUS option selected. The data on the R lines does not affect the contents of the R register.

The R lines are available only on the COP440 and COP2440. The R register, however, is available and can be used in the COP441, COP442, COP2441, and COP2442.

4.5.3 The Serial I/O Port - MICROWIRE

As explained in Section 2.4.5, the serial I/O port may be configured as a serial shift register or a four-bit binary down counter. In the shift register mode, the serial port is the MICROWIRE interface (see Section 2.4.5). The operating mode of the serial port is controlled by the Enable register (see Section 2.5 and Table 2-2).

In the binary counter mode, SO and SK are logic controlled outputs. The state of SO is directly controlled by the LEI instruction. SK outputs the status of SKL, the SK latch. In the shift register mode, SO is either "0" or serial out, and SK is either "0" or a clock output as indicated in Table 2-2. Regardless of mode, SKL is loaded with the status of the C register whenever an XAS instruction is executed. Thus, SK is controlled by setting or resetting C and then executing an XAS. The XAS instruction, however, is also the means of reading the SIO register. Therefore, every time the user reads SIO, C is copied to SKL. Therefore, the user should insure the status of C before executing an XAS instruction if the status of SK is important. Also note that if SIO is in counter mode and SKL is "1" (SK = 1), and SIO changed to shift register mode, SK will become a clock immediately. The converse is also true: If SIO is shift register and SKL = 1, and SIO is changed to a counter, SK will go to a high state immediately.

Regardless of mode, SI can be used as a general purpose input. In the shift register mode, data will shift in at the SI pin. The user can read the status of SI with the XAS instruction. In the counter mode, SIO will, in effect, capture a low-going pulse. The user can preload the counter by setting the accumulator to some value, typically 0 or 15, and loading that value into SIO with an XAS instruction. The user would then periodically read SIO to see if the value had been decremented. If it had, the pulse had occurred.

With the SIO register in the shift register mode, continuous data streams can be sent or received. In this mode, data is normally in multiples of four bits. To preserve proper timing, an XAS must appear every fourth instruction cycle. As will be seen, this is simple to implement. The reason for this requirement should be obvious. SIO is a four-bit shift register which shifts at the instruction cycle rate. Thus data must be read, or new data loaded, every fourth instruction cycle.

4.6 INTERRUPT

The interrupt input on COPS microcontrollers is IN_1. In the COP440 series and COP2440 series, the CKO input may also be an interrupt input. Thus, except for the COP442 and COP2442, interrupt is not available on any device that does not have the IN_1 input.

4.6.1 Conditions for Interrupt Recognition

An interrupt will be recognized or acknowledged if and only if the following conditions are met:

1. Interrupt has been enabled by setting bit EN_1 of the enable register.
2. A low-going pulse ("1" to "0") at least two instruction cycles wide (one instruction cycle in COP2440 series) occurs at the IN_1 (or CKO in COP440/COP2440 series) input. The high to low transition must occur while EN_1 is set.
3. A currently executing instruction is completed.
4. All successive transfers of control instructions and successive LBI instructions are completed (*e.g.*, if the main program is executing a jump or subroutine call which transfers control to another jump or subroutine call, the interrupt will not be acknowledged until the second jump or subroutine call has been executed).

4.6.2 Effects of Interrupt Acknowledge

When an interrupt has been acknowledged as explained in Section 4.6.1, the following occurs:

1. The next sequential program counter address (PC+1) is pushed onto the program stack.
2. On COP440, COP2440, and COP484 series devices, an interrupt status bit is stored with the address in the subroutine stack.
3. On all other COPS microcontrollers, the interrupt status bit, which remembers the status of the skip logic, is saved separately. This bit is not carried with the address in these devices.
4. The program counter is set to address OFF hex. On all devices except the COP440/COP2440 series, the next executable address is hex 100. In the COP440/COP2440 series, hex 100 is the next executable address if EN_4 is reset. If EN_4 is set in these devices, the program counter branches from hex address OFF to hex address 300.
5. EN_1 is reset thereby disabling further interrupts.

4.6.3 Interrupt Handling

Due to hardware considerations, the instruction at hex address OFF **must** be a NOP.

The interrupt status bit remembers if a skip was generated as a result of the completed instruction. In the COP420/COP424C/420L/444L devices, this bit is stored separate from the return address. If set, this bit forces a skip on the first "stack pop" following the interrupt. This means that the use of subroutines, nested interrupts, or the LQID is limited in these devices. An unexpected skip may occur and the original skip status is lost. The user may, of course, defeat this skip by means of an artificial subroutine call, *e.g.*, a JSRP to a RET instruction, followed by a NOP. This will clear the status bit, and subroutines, etc. may be used without restriction. Remember, however, that this procedure destroys the original skip status. No such situation exists in the COP440/2440 series and COP484 devices. The status bit is saved with the address. Subroutines may be freely used in the interrupt service routines and nested interrupts are permitted.

Subject to the restraints mentioned above, interrupts may be re-enable at any time by means of the LEI or CAME instructions. Typically, this re-enabling would occur immediately before the return instruction at the end of the interrupt service routines.

4.6.4 Interrupt Disable

Interrupts are disabled by resetting EN_1 by any valid instruction (LEI or CAME) and by interrupt acknowledge. While EN_1 is low, no interrupt processing of any kind goes on. Thus, a high to low transition at IN_1 which is otherwise valid is not recognized when EN_1 is reset. Furthermore, when EN_1 is set, there is no memory of the event that occurred while EN_1 was reset. The software interrupt disable will prohibit recognition of all interrupt signals which occur subsequent to the disable. Obviously, the interrupt disable instruction cannot disable interrupts which occur before the instruction is executed. More significantly, the interrupt disable instruction also does not disable interrupts which occur during the execution of the instruction. Thus, a valid interrupt signal may occur, and interrupt acknowledge is pending completion of the current instruction. That current instruction may well be an interrupt disable; nonetheless, the interrupt will be acknowledged and the interrupt service routine entered.

Note that in branching to the interrupt routine, the microcontroller saves only the program counter and the skip status. If it is necessary to save other items, the user must do so himself in software. Similarly, the user must restore those values at the end of the interrupt service routine.

4.6.5 Interrupt in the COP440/COP2440 Series

The COP440 and COP2440 series devices are the only COPS microcontrollers with more than one possible interrupt source. The choice of interrupt is governed by bits EN_4 and EN_5 of the enable register as indicated in Table 2-3. The four possible interrupt sources are as follows:

1. IN_1 negative edge - This is the standard COPS interrupt (EN_5 , EN_4 = 00).
2. CKO input - If the CKO input mask option is selected, that input can be selected as an interrupt input. Operation is the same as the IN_1 interrupt (EN_5, EN_4 = 0). If CKO is not selected as an input, selection of CKO as interrupt source has no effect. No interrupt will occur.
3. Zero Crossing on IN_1 - IN_1 may be mask programmed to be a zero crossing detect input. Interrupt can be selected to occur at each zero crossing. If the zero cross detect option is not selected, this interrupt source selection will result in an interrupt at every transition of IN_1 (EN_5, EN_4 = 10).
4. T counter overflows - This is an internal interrupt which can be selected. Interrupt will occur whenever the T counter overflows. All the conditions required for interrupt to be acknowledged, with the obvious exception of input pulse width, are still valid and must be met (EN_5, EN_4 = 11).

The interrupt source should not be changed while the interrupt is enabled (EN_1 = 1). A false interrupt may occur if the interrupt source is changed while EN_1 is a 1. To avoid this problem, the interrupt must be disabled prior to, or at the same time as, the change of the interrupt source. Do not enable the interrupt at the same time as changing the interrupt source. A proper sequence for altering the interrupt source, then, is as follows:

1. Disable interrupt.
2. Change interrupt source (Steps 1 and 2 may be combined.)
3. Enable interrupt.

4.7 PROGRAM EFFICIENCY

Three factors are normally involved in determining program efficiency:

1. Program memory (ROM) efficiency, using the least amount of ROM.
2. Data memory (RAM) efficiency, using the least amount of RAM.
3. Execution time efficiency, executing the function in the shortest amount of time.

These three factors, unfortunately, conflict with one another. The most memory efficient implementation of a function is not usually the most execution time efficient implementation. The most RAM efficient implementation is frequently not the most ROM efficient implementation.

Like all single-chip microcontrollers, COPS microcontrollers are memory limited. A premium is therefore placed on general memory efficiency — getting the maximum function in the smallest memory. The reason is simple economics: devices with greater memory capacity are generally more expensive than devices with lesser capacity. Despite the premium on memory efficiency, the application can easily require compromises – sacrifice a little ROM or RAM or both in order to achieve faster execution speed.

Since these conflicting requirements exist, several versions of the standard programs in Chapter 5 are provided. These should help the user to understand the conflict and to make intelligent, informed decisions on any compromises.

4.8 RULES AND TECHNIQUES

4.8.1 Absolute Requirements

There are very few absolute requirements for COPS programming. The restrictions on the instructions are described in Chapter 3. The remaining absolute rules are as follows:

1. The instruction at address 000 must be a CLRA.
2. If interrupts are used, the instruction at hex address 0FF must be a NOP.
3. At least the first instruction of subroutines called with the single byte JSRP must be in Page 2. Note there is no requirement that any other instructions of such subroutines be located in Page 2.

4.8.2 General Guidelines

This section will provide general guidelines to help the programmer write an efficient COPS program. Examples are provided here and in Chapter 5. Most of these guidelines will reduce memory usage at the expense of execution speed. The programmer may have to make the compromises described in Section 4.7.

Maximize the Use of Subroutines

If a single operation is frequently performed, make that operation a subroutine. If possible, make it a subroutine that can be called with the single-byte JSRP. Try to combine similar operations into a common subroutine, even if it means that an unnecessary operation is performed in some cases. This is "wrong" only if this unnecessary operation interferes in some significant way with achieving the end result. The programmer may use pieces of existing subroutines as new subroutines: multiple entry points are a good thing if code is saved. Consider the following short routine:

```
ENTRY1: LBI 0,15
ENTRY2: LD
ENTRY3: CAB
ENTRY4: OBD
ENTRY5: RET
```

It is entirely conceivable that every instruction in this routine is a subroutine entry point. We shall assume that this routine is in Page 2 for maximum savings. A JSRP to ENTRY1 will output the value in RAM (0,15) to D. A JSRP to ENTRY2 will output the RAM digit addressed by B to the D port. A JSRP to ENTRY3 will output the accumulator to the D port. A JSRP to ENTRY4 simply does an OBD. A JSRP to ENTRY5 is, effectively, a NOP but finds usefulness in creating software delays. This is an example of maximizing subroutine usage and sharing commonality or finding commonality where it is not obvious. Page 2 should be filled with subroutine entry points. This will increase the memory efficiency of the program.

Any multibyte instruction can be converted into a single-byte instruction by means of a subroutine. Entry point ENTRY4 in the preceding example illustrates this. If a given multibyte instruction is frequently used in a program, it will probably be beneficial to make a subroutine out of it. Remember, this includes any multibyte instruction. It is common that various branches of a program will jump back to some central location in the program. These jumps can be implemented with a JSRP; the subroutine will consist totally of JMP CENTER, a jump to the central location. This is completely acceptable and will save code. A subroutine does not have to have a return instruction associated with it.

Use the skip feature of successive LBI instructions. This is a very powerful feature that permits code sharing and promotes commonality. It easily lends itself to multiple entry point routines. Consider the following digit right shift routine:

```
RSHO:  LBI   0,15
RSH1:  LBI   1,15
RSH2:  LBI   2,15
RSH3:  LBI   3,15
       CLRA
LOOP:  XDS
       JP    LOOP
       RET
```

Depending on the entry point, this routine will right shift register 0, 1, 2, or 3 one digit. The successive LBI feature finds use in this kind of routine, so the same routine can be used regardless of data location in tests and in "non-obvious" ways. Consider the following:

```
G10:  LBI   0,10
G9:   LBI   0,9
G1:   LBI   0,1
G0:   LBI   0,0
      CBA
      X
      OMG
      X                 ;Restore original RAM value
      RET
```

Here, the LBI instruction is being used to establish the G output value. The LBI instruction can be used in many similar ways. The interesting thing about this usage is that the LBI is, in itself, being used to create a value and not to point to a given RAM digit, even though the B register is modified by the instruction.

Careful RAM allocation is essential. Careful placement of data in RAM can have significant impact on the amount of program memory required. The use of a RAM map, a visualization of the data placement in RAM, is an invaluable aid. It is nearly impossible to write an efficient program without the use of a RAM map. Figure 4-1 is a sample RAM map for a

COP420. The basic guidelines for data placement in RAM are as follows:

1. Flags should be placed in memory locations addressable by a single-byte LBI.
2. A commonality of bit position within a digit for flags is desirable. This permits the creation of flag testing subroutines like the following:

```
FLAG1:  LBI     3,15
FLAG2:  LBI     3,14
        SKMBZ   1
        RET
        RETSK
```

3. Data should be placed at the "ends" of registers to take advantage of the skip features of the XIS and XDS instructions. It takes far less code to exit by "falling of the end" of a register than to test Bd, or some other loop counter, for completion.

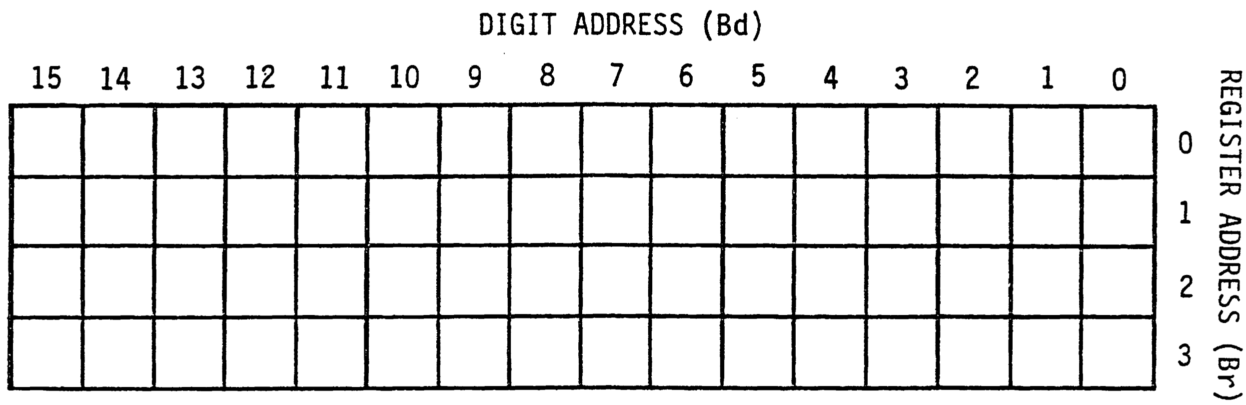
The LD, X, XIS, and XDS are associated with the exclusive OR feature whereby Br can be modified. If RAM data and flags are intelligently placed, data manipulation and B register modification can be accomplished in a single instruction thereby saving code. Obviously, the effective use of these instructions goes hand in hand with an effective RAM layout. The basic integer BCD addition below illustrates this feature. The routine is a four-digit BCD addition, adding register 0 to register 3; result to register 0.

```
BCDADD:  LBI    3,12
LOOP:    LD     3       ;fetch data and point to RO
         AISC   6       ;decimal adjust to force carry if A=9
         ASC            ;add
         ADT            ;decimal correct
:        XIS    3       ;place digit in RO, increment Bd, point to R3
         JP     LOOP    ;XIS skip indicates finish
         RET
```

In the above routine, the B register is being continuously modified but there is no LBI instruction other than the one required at the start of the routine.

The table look-up instructions, LQID and LID, can save both code and execution time. Tables can be used in many ways: code conversions, arithmetic, data processing, key decoding, etc. If some set of values is to be derived from another set of values, a table will frequently be more efficient than a computation. The look up will also be invariably faster than a computation. Tables greatly facilitate the handling of inputs from non-linear sources, e.g., temperature sensors; they make creation of display a trivial task. The use of a table is not a panacea but is frequently a possible solution worth considering.

The indirect jump instruction, JID, should be used with some care. Because of its "two-tier" organization, this instruction does not always save code. JID permits a jump on the basis of data. As such, it is very useful in decode situations. It is not necessarily the most code efficient decoding scheme, but it is always the most time efficient and time uniform decoding scheme.



BA-05-0

Figure 4-1. COP420 RAM Map

For execution speed efficiency, do not put unnecessary instructions in loops. Look for ways to move instructions out of loops. It is frequently possible to move seemingly necessary instructions out of program loops. This is a speed improvement that usually costs little or no code.

4.9 STRUCTURED PROGRAMMING TECHNIQUES

The techniques of structured programming or top-down programming are excellent organizational tools and work well on large systems. However, these techniques have a basic implementation problem at the level of single-chip microcontrollers in general and COPS microcontrollers in particular. Systems based on COPS devices are generally seeking maximum function with minimum memory.

Efficient COPS programming requires the elimination or minimization of redundant or duplicated code. Maximum sharing of common or related code is necessary. Partial sharing of routines is also common. Most subroutines in an efficient COPS program will have multiple entry points. There are branches into and out of routines that exist solely to reduce memory usage. All of this is in direct conflict with the top-down modular approaches. An efficient COPS program is not written by assembling independent blocks. That technique will use excessive code and could require a user to use a larger device than necessary. It is difficult, in an efficient COPS program, to extract independent modules other than the most basic functions.

The concepts of structured programming are still useful in defining the functions that must be performed and their inter-relationships. When the time comes to write the code within the memory limits of the microcontroller, the concepts fail. At this point, the user should use the approaches and techniques in this manual. Remember, the objective is to write an efficient COPS program thereby obtaining maximum function in minimum memory. Rarely, if ever, is the objective to write an easily readable program with modular, transportable functional blocks that exceed the memory capacity of the device.

Chapter 5

STANDARD PROGRAMS

5.1 INTRODUCTION

This section contains a number of standard programs illustrating various techniques and the implementation of various functions. If the user wishes to use any of these programs, he or she should remember that maximum efficiency will be obtained by tailoring the program to the application. Copying the programs "as is" generally is not efficient.

5.2 MATH PACK

This section includes a variety of arithmetic routines, including the following:

- Increment routines
- Decrement routines
- Integer Addition
- Integer Subtraction
- Binary Multiply
- Basic Arithmetic Package: Add, Subtract, Multiply, Divide
- Square Root
- Binary to BCD Conversion
- BCD to Binary Conversion

Typically, more than one implementation of a function is given.

5.2.1 Basic Increment Routines

Binary Routines

The following three routines have the same function: They perform a binary addition of 1 to a 12-bit binary number. The number is located in register 0, digits 15 through 12.

	Bd 15	Bd 14	Bd 13
Br 0	M S B		L S B

I

```
INCR:   LBI     0,13
        SC
INCR1:  CLRA
        ASC
        NOP
        XIS
        JP      INCR1
        RET
```

II

```
INCR:   LBI     0,13
INCR1:  LD
        AISC    1
        JP      INCR2
        XIS
        JP      INCR1
        RET
INCR2:  X
        RET
```

II

```
INCR:   LBI     0,13
        LD
        AISC    1
        AISC    15
        XIS
        LD
        AISC    1
        AISC    15
        XIS
        LD
        AISC    1
        NOP
        X
        RET
```

	I	II	II
ROM Words Used:	8	9	14
Execution Time: (instruction cycles)	18	Data Dependent 6-16	14

The preceding three examples illustrate an important point: The most code efficient method of implementing this function takes more time to execute than either of the other two implementations. This is a fairly common characteristic. Implementation II is, on the average, the fastest executing routine. Its main drawback is that its execution time is data dependent. This may not be significant. Implementation I uses and modifies the C register; the other implementations do not. All three routines use the accumulator.

BCD Routines

The following routines have the same function: They increment a three-digit BCD number by one.

	Bd		
	15	14	13
Br 0	M S D		L S D

I			II			II		
INCR:	LBI	0,13	INCR:	LBI	0,13	INCR:	LBI	0,13
	SC		INCR1:	LD			LD	
INCR1:	CLRA			AISC	7		AISC	7
	AISC	6		JP	INCR2		AISC	9
	ASC			XIS			XIS	
	ADT			JP	INCR1		LD	
	XIS			RET			AISC	7
	JP	INCR1	INCR2:	ADT			AISC	9
	RET			X			XIS	
				RET			LD	
							AISC	7
							ADT	
							X	
							RET	
ROM Words Used: 9			10			14		
Execution Time: 21 (instruction cycles)			Data Dependent 7-17			14		

The same comments made for the binary routines are valid for the BCD routines.

5.2.2 Basic Decrement Routines

Binary Routines

The following routines take a 12-bit binary number and decrement it by one.

	Bd		
	15	14	13
Br 0	M S B		L S B

I			II			II		
DECR:	LBI	0,13	DECR:	LBI	0,13	DECR:	LBI	0,13
	RC		DECR1:	LD			LD	
DECR1:	CLRA			AISC	15		AISC	15
	CASC			JP	DECR2		XIS	
	NOP			X			LD	
	XIS			RET			AISC	15
	JP	DECR1	DECR2:	XIS			XIS	
	RET			JP	DECR1		LD	
				RET			AISC	15
							NOP	
							X	
							RET	

ROM Words Used: 8	9	12
Execution Time: 18 (instruction cycles)	Data Dependent 6-17	12

As with the increment routines, the routine requiring the least code takes the most time.

BCD Routines

The following routines take a three-digit decimal number and decrement it by one.

	Bd 15	Bd 14	Bd 13
Br 0	M S D		L S D

	I			II			II	
DECR:	LBI	0,13	DECR:	LBI	0,13	DECR:	LBI	0,13
	RC		DECR1:	LD			LD	
DECR1:	CLRA			AISC	15		AISC	15
	CASC			JP	DECR2		STII	9
	ADT			X			LD	
	XIS			RET			AISC	15
	JP	DECR1	DECR2:	ADT			STII	9
	RET			XIS			LD	
				JP	DECR1		AISC	15
				RET			ADT	
							X	
							RET	
ROM Words Used:	8			10			12	
Execution Time: (instruction cycles)	18			Data Dependent 6-20			12	

The same pattern is observed here as in the other similar routines.

5.2.3 Integer Addition

Binary Addition

The routine below is the basic addition routine. It illustrates the power of the exclusive OR argument of the LD, XIS, XDS, and X instructions. It also illustrates the conciseness that can come from intelligent data placement in RAM. As written, the routine is a 16-bit binary add, R1 + R0 --> R0.

		Bd 15	14	13	12
Br	0	MSB	Operand 1; sum		LSB
	1		Operand 2;		

```
BINADD:  LBI    1,12   ; set-up B register
         RC            ; initialize Carry to 0
LOOP:    LD        1   ; fetch data from R1 and point to R0
         ASC           ; add RAM(B) + A + C --> A
         NOP           ; defeat skip
         XIS       1   ; store result to R0, increment Bd, point to R1
         JP     LOOP   ; Loop control
         RET           ; all done, exit
```

ROM Words Used: 8
Execution Time: 23 instruction cycle times.

BCD Addition

This routine is essentially the same as the binary add routine. A four-digit BCD add is illustrated. Again, R1 + R0 --> R0.

	Bd 15	Bd 14	Bd 13	Bd 12
Br 0	MSD	Operand 1; sum		LSD
Br 1		Operand 2;		

```
BINADD:  LBI    1,12   ; initialize B to LSB
         RC            ; initialize Carry to 0
LOOP:    LD        1   ; fetch data from R1 and point to R0
         AISC      6   ; decimal adjust to force carry at 9 -> 10
         ASC           ; add
         ADT           ; decimal correct if no carry
         XIS       1   ; store result in R0
         JP     LOOP
         RET
```

ROM Words Used: 9
Execution Time: 23 instruction cycle times

Both of these addition routines can be expanded up to 64 bits or 16 digits merely by changing the starting address, the Bd value in particular. Also note that the data could be placed at the other end of the register and XDS used in place of XIS.

Since the routine is essentially independent of data length and the exclusive OR feature of the LD, XIS, XDS, and X instructions permits easy transportation across data registers, a very versatile and compact routine can be created. Consider the following variation on the BCD addition routine:

```
ADD1:   LBI     3,0   ; R3+R0->R0, 16-digit add
ADD2:   LBI     0,0   ; R3+R0->R3, 16-digit add
ADD3:   LBI    1,10   ; R1+R2->R2, 6-digit add
ADD4:   LBI    2,10   ; R1+R2->R1, 6-digit add
        RC
LOOP:   LD        3
        AISC      6
        ASC
        ADT
        XIS       3
        JP     LOOP
        RET
```

Here we have the same routine able to work on two different sets of registers with different data lengths. Furthermore, either register in a given set can be the destination for the result. The controlling factor in all of this is simply the value in the B register at the start of the routine. The repeated LBI skip feature proves very useful in creating a multiple entry subroutine such as this one.

Variations on these basic two register additions similar to the techniques shown in the basic increment and decrement routines can be created. This is left as an exercise for the programmer. The most code efficient techniques have been illustrated here.

5.2.4 A Doubling Routine

A routine to double the value in a register is a simple outgrowth from the basic addition routine. This routine is illustrated below for a binary double. Data placement is the same as shown earlier.

```
        2 x R0-->R0, 16-bit binary

DOUBLE: LBI     0,12
        RC
LOOP:   LD              ; RAM(B) ->A, Br not changed
        ASC
        NOP
        XIS             ; A ->RAM(B), increment Bd, Br unchanged
        JP      LOOP
        RET
```

The routine for a decimal double is derived from the BCD add routine in the same manner: the exclusive OR argument on the LD and XIS instructions is changed to 0 so that Br is not altered by those instructions.

An Example of the Effect of Data Placement in RAM

If assumed, in either of the addition routines presented earlier, that the the data is not optimally placed at the end of registers, the following routine could be the result:

	Bd 15	14	13	12
Br 0	M S B	Operand 1; sum		L S B
1		Operand 2		

R1+R0->R0, 16-bit binary

```
BINADD: LBI     1,10    ; initialize B
        RC              ; initialize Carry
LOOP:   LD      1
        ASC
        NOP
        XIS     1
        CBA             ; test for Bd > 13-if yes, done
        AISC    2
        JP      LOOP
        RET
```

ROM Words Used: 10
Execution Time: 31 instruction cycle times

In this example, inefficient data placement resulted in a 25 per cent code increase and a nearly 50 per cent increase in execution time. The message should be clear from this: Placement of data in RAM can have dramatic effects on the program.

5.2.5 Integer Subtract

These routines are the counterparts of the integer addition routines in Section 5.2.3. The RAM maps are the same as in that section.

Binary Subtraction

The routine as written below is a 16-bit binary subtraction (R1-R0->R1).

```
BINSUB:   LBI     0,12   ; initialize B register
          SC             ; set Carry for subtract
LOOP:     LD         1   ; fetch R value and point to R1
          CASC           ; subtract
          NOP
          XIS        1   ; save result in R1, increment Bd, point to R1
          JP      LOOP
          RET
```

BCD Subtraction

The BCD counterpart of the preceding is the following four-digit subtract routine (R1-R0->R1).

```
BCDSUB:   LBI     0,12   ; initialize B
          SC             ; set C for subtract
LOOP:     LD         1   ; fetch R0 value, point to R1
          CASC           ; subtract
          ADT            ; decimal correct (15 -> 9)
          XIS        1   ; save in R1, increment Bd, point to R0
          JP      LOOP
          RET
```

These routines are direct counterparts to the addition routines. The comments in Section 5.2.3 are equally valid for these subtract routines.

5.2.6 Up-Down Counters

The up-down counter routine is an extension or combination of the basic increment or decrement routines. Both an increment and a decrement have, effectively, been combined. The C register is used to distinguish between counting up or counting down. The basic flow for the routine and the RAM map is shown below in Figure 5-1.

	Bd 15	14	13	12
Br 2	Counter			
3	Not Used			TEMP STORE

The flow chart and RAM map are valid for both the binary and BCD versions of the routine. Two implementations of each are given: The first is a simple combination of the increment

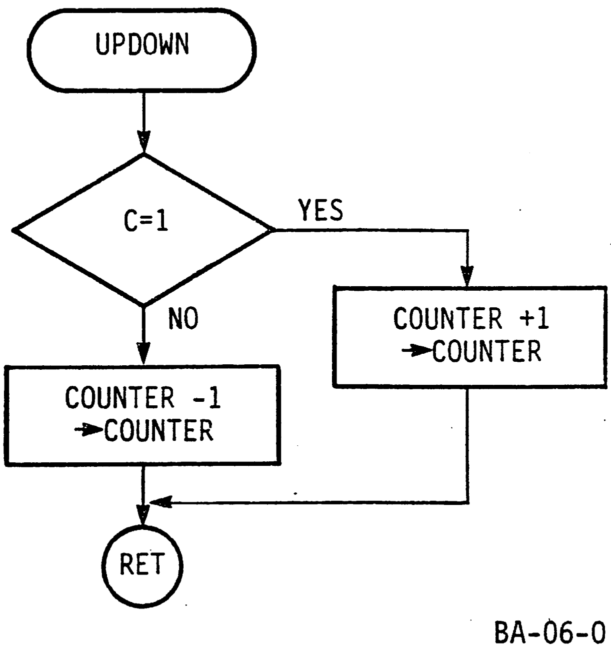


Figure 5-1. Basic Flow for Up-Down Counter Routine

decrement routines. The second is a somewhat more sophisticated implementation which saves a little code but uses more RAM (one extra digit which is in TEMP STORE in the RAM map).

Binary Up-Down Counter

I

```
UPDOWN:   LBI     2,12
          SKC
          JP      DOWN
UP:       CLRA
          ASC
          NOP
          XIS
          JP      UP
          RET
DOWN:     CLRA
          CASC
          NOP
          XIS
          JP      DOWN
          RET
```

II

```
UPDOWN:   LBI     3,12
          CLRA
          SKC
          COMP
COUNT:    X       1 ; point to 2,12
COUNT1:   LDD     3,12
          ASC
          NOP
          XIS
          JP      COUNT1
          RET
```

Version II of this routine loads 0 or 15 into a RAM location. Then the state of the carry controls addition or subtraction. Note that the location of the temporary data storage digit was chosen to use the exclusive OR capability of the X instruction to eliminate an instruction.

BCD Up-Down Counter

I

```
UPDOWN:  LBI   2,12
         SKC
         JP    DOWN
UP:      CLRA
         AISC  6
         ASC
         ADT
         XIS
         JP    UP
         RET
DOWN:    CLRA
         CASC
         ADT
         XIS
         JP    DOWN
         RET
```

II

```
UPDOWN:  LBI   3,12
         CLRA
         SKC
         AISC  9
COUNT:   X     1
COUNT1:  LDD   3,12
         AISC  6
         ASC
         ADT
         XIS
         JP    COUNT1
         RET
```

The comparison is the same as the binary routines. Version II here also illustrates another point. As written, Version II will execute (increment or decrement the four-digit counter) in 34 instruction cycle times. By merely moving the AISC 6 instruction from its present location to after the CLRA, the execution time is improved without any penalty.

IIa

```
UPDOWN:  LBI   3,12
         CLRA
         AISC  6          ; build decimal correct into the stored constant
         SKC
         AISC  9
COUNT:   X     1
COUNT1:  LDD   3,12
         ASC
         ADT
         XIS
         JP    COUNT1
         RET
```

This routine is completely equivalent in function, approach, and amount of code as Version II. It executes faster, however, 31 instruction cycles rather than 34. The reason for the speed improvement is that an instruction, the AISC 6, was moved out of the loop and into the "main body" of the routine.

5.2.7 Binary Multiply

A routine for a 16 by 16 bit binary multiply is given below. A 32-bit product is generated. A RAM map for this routine is given below. A flow chart is in Figure 5-2.

<table>
<tr><td></td><td></td><td colspan="9" align="center">Bd</td></tr>
<tr><td></td><td></td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td></tr>
<tr><td rowspan="2">Br</td><td>0</td><td colspan="4">X (multiplicand)</td><td colspan="4">Z</td><td>BIT COUN-TER</td></tr>
<tr><td>1</td><td colspan="4">“0”</td><td colspan="4">Y (multiplier)</td><td>NOT USED</td></tr>
</table>

The routine does the following:

X x Y –> XZ, previous X lost; Y unchanged

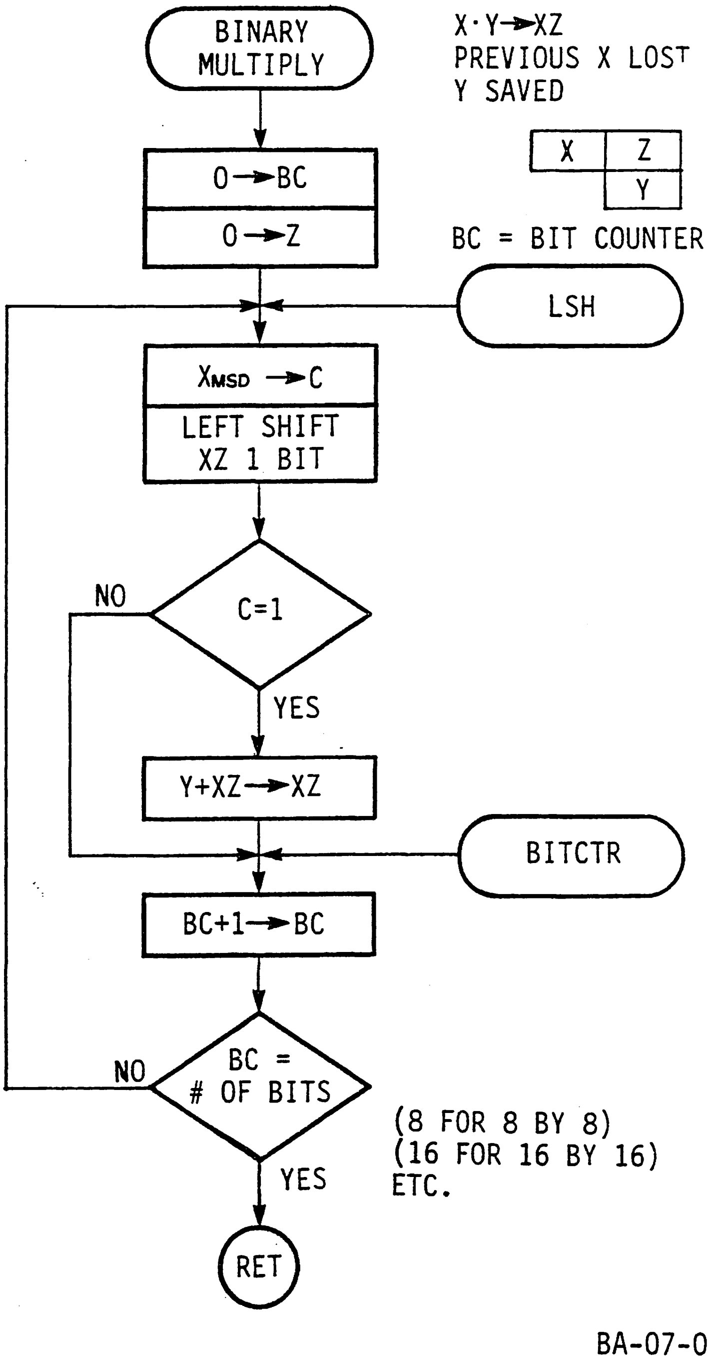


Figure 5-2. Binary Multiply

```
BINMULT:  LBI   0,7
          STII  0       ; clear bit counter and Z
          STII  0
          STII  0
          STII  0
          STII  0
          LBI   1,12
          STII  0
          STII  0
          STII  0
          STII  0
          LBI   0,8
          RC
LSH:      LD            ; left shift XZ 1 bit, putting XMSB into C
          ASC
          NOP
EX:       XIS
          JP    LSH
          SKC
          JP    BITCTR
BINADD:   RC
          LBI   1,8     ; Y + XZ -> XZ
ADD:      LD    1
          ASC
          NOP
          XIS   1
          JP    ADD
BITCTR:   LBI   0,7     ; increment bit counter and test if done
          LD
          AISC  1
          JP    EX
          RET
```

5.2.8 Basic Arithmetic Package

This section includes the basic arithmetic functions, add, subtract, multiply, and divide. The routines are written as a cohesive unit. They are for eight-digit floating-point fully algebraic arithmetic. Figures 5-4 through 5-7 are the RAM map and flow chart for these routines.

Both decimal and binary (hexadecimal) versions of these routines are provided. The flow charts and RAM map are valid for both these versions.

The routines listed in Figures 5-3 and 5-8 have an arbitrary error handling routine; the error is merely flagged by setting the decimal point and sign position to 15. The user can modify this to a perhaps more useful arrangement.

```
            ;BASIC BCD FLOATING POINT ARITHMETIC ROUTINES
            ;
            ;REGISTER 0 = X,REGISTER 1 = Y, REGISTER 2 = Z
            ;
            ;THE ROUTINES ARE FOR 8 DIGIT,BCD,FULLY ALGEBRAIC ADD,SUBTRACT
            ;MULTIPLY AND DIVIDE. ALL ROUTINES ARE FULLY FLOATING POINT.
            ;THE ROUTINES ASSUME AN 8 DIGIT MANTISSA, A SIGN DIGIT, AND A
            ;DECIMAL POINT DIGIT.THE DECIMAL POINT DIGIT IS A DECIMAL POINT
            ;POSITION INDICATOR,I.E.,A DEC. PT.POSITION OF 0 INDICATES
            ;THAT THE DECIMAL POINT IS PLACED AFTER THE LSD OF THE NUMBER;
            ;DEC.PT. POSITION OF 7 INDICATES THAT THE DECIMAL POINT IS
            ;PLACED AFTER THE MSD OF THE NUMBER. OTHER NUMBERS CORRESPOND
            ;IN THE SAME MANNER TO INTERMEDIATE DIGITS.
            ;
            ;THE ROUTINES ALSO ASSUME THAT THERE IS A GUARD OR OVERFLOW
            ;DIGIT FOR THE NUMBERS.THE MANTISSA IS 8 DIGITS PLUS THE GUARD
            ;DIGIT FOR A TOTAL OF 9 DIGITS.THE GUARD DIGIT IS FOR INTERNAL
            ;USE ONLY AND IS NOT AVAILABLE ON INPUT OR OUTPUT.
            ;
            ;THE ROUTINES CAN BE  MODIFIED FOR HEX OR BINARY ARITHMETIC.
            ;AS THE ALGORITHMS ARE NOT NUMBER BASE DEPENDENT(EXCEPT FOR
            ;OBVIOUS THINGS LIKE OVERFLOW TESTS, ETC. WHICH WOULD HAVE TO
            ;BE MODIFIED TO ACCOMODATE THE NUMBER BASE USED).
            ;
            ;THE CODE AS WRITTEN SHOULD WORK IN COP420 AND LARGER DEVICES.
            ;THE ROUTINES ARE WRITTEN AS SUBROUTINES CALLED BY A MAIN
            ;PROGRAM.  ONE LEVEL OF SUBROUTINE IS USED BY THE ARITHMETIC
            ;ROUTINES.  COMPARABLE ROUTINES CAN BE WRITTEN FOR THE COP410
            ;BUT SOME CHANGES ARE REQUIRED. THE ALGORITHM IS STILL VALID
            ;ALTHOUGH THE IMPLEMENTATION IS SOMEWHAT DIFFERENT.
            ;
     0022   SAVE1   =       2,2
     000F   XGUARD  =       0,15
     000E   XMSD    =       0,14
     0007   XLSD    =       0,7
     0006   ROUND   =       0,6
     0001   XSIGN   =       0,1
     0000   XDP     =       0,0
     001F   YGUARD  =       1,15
     001E   YMSD    =       1,14
     0017   YLSD    =       1,7
     0011   YSIGN   =       1,1
     0010   YDP     =       1,0
     002F   ZGUARD  =       2,15
     002E   ZMSD    =       2,14
     0027   ZLSD    =       2,7
     0021   ZSIGN   =       2,1
     0020   ZDP     =       2,0
     003F   FLAGS   =       3,15
     0030   OFLOW   =       3,0
            ;
```

Figure 5-3. BCD Arithmetic Package (Sheet 1 of 9)

```
     0000             .PAGE    0
000 00                CLRA
001 0F                LBI      0,0       ;CLEAR ALL THE RAM
002 53                AISC     3
003 12       RAMCLR:  XABR
004 81                JSRP     CLEAR
005 12                XABR
006 5F                AISC     15
007 C9                JP       TESTG
008 C3                JP       RAMCLR
             ;
             ;*************************************************************
             ;FOLLOWING CODE--TO NEXT LINE OF **--IS FOR CONTROL ONLY
             ;
009 335F     TESTG:   OGI      15        ;PUT G LINES HIGH FOR READING G
             ;
                   ;USING G LINES FOR PRIMITIVE CONTROL TO SELECT ADD,SUB
                   ;MULTIPLY OR DIVIDE--WILL ENTER NUMBERS IN BREAKPOINT
                   ;MODE USING MODIFY COMMAND
                   ;
00B 3301              SKGBZ    0
00D D3                JP       TESTG1
00E 3E                LBI      FLAGS
00F 70                STII     0         ;RESET BIT 2 FOR ADD
010 6840     JSRALN:  JSR      ALIGN
012 C9                JP       TESTG
013 3311     TESTG1:  SKGBZ    1
015 D9                JP       TESTG2
016 3E                LBI      FLAGS     ;SET SUBTRACT BIT
017 74                STII     4         ;SET BIT 2 FOR SUBTRACT
018 D0                JP       JSRALN
019 3303     TESTG2:  SKGBZ    2
01B DF                JP       TESTG3
01C 3E                LBI      FLAGS
01D 70                STII     0         ;RESET BIT 3 FOR DIVIDE
01E E4                JP       JSMD
01F 3313     TESTG3:  SKGBZ    3
021 C9                JP       TESTG
022 3E                LBI      FLAGS
023 78                STII     8         ;SET BIT 3 FOR MULTIPLY
024 6940     JSMD:    JSR      MULDIV
026 C9                JP       TESTG
             ;
             ;PRECEEDING CODE FOR CONTROL ONLY,HAS NOTHING TO DO WITH THE
             ;ARITHMETIC ALGORITHMS
             ;**************************************************************
             ;

     0040             .PAGE    1
             ; THIS IS THE ALIGN ROUTINE FOR ADD/SUBTRACT.  IT MAKES THE
             ; DECIMAL POSITIONS OF THE TWO NUMBERS EQUAL BEFORE ADD OR
             ; SUBTRACT TAKES PLACE.  THE ROUTINE ASSUMES THAT THE NUMBERS
```

Figure 5-3. BCD Arithmetic Package (Sheet 2 of 9)

```
                  ; ARE RIGHT JUSTIFIED ON ENTRY.  DECIMAL POINT POSITION VALUES
                  ; ARE RESTRICTED TO NUMBERS BETWEEN 0 - 8 (SINCE WE ARE ONLY
                  ; DOING 8 DIGIT ROUTINES).  ROUTINE ONLY REQUIRED FOR FLOATING
                  ; POINT ADD/SUBTRACT ALGORITHMS
                  ;
040 0F            ALIGN:   LBI     XDP
041 15                     LD      1
042 21                     SKE             ;TEST DP0=DP1(DPX=DPY)
043 C6                     JP      ALIGN2
044 6100                   JMP     ADDSUB  ;IF EQUAL,PROCEED TO ADD/SUBTRACT
046 10            ALIGN2:  CASC            ;TEST DP0 > DP1
047 D6                     JP      DP0GT1  ;YES
048 0D            DP0LT1:  LBI     XMSD    ;DP0<DP1.IF XMSD NOT 0,RIGHT SHIFT
049 00                     CLRA            ;M1,ELSE LEFT SHIFT M0
04A 21                     SKE
04B D1                     JP      R1RSFT
04C 87            R0LSFT:  JSRP    LSFTR0
04D 0F                     LBI     XDP
04E 1F            DPPL1:   LBI     YDP
04F B5                     JSRP    PLUS1   ;MODIFY DP AFTER SHIFT
050 C0                     JP      ALIGN
051 8E            R1RSFT:  JSRP    RSFTR1
052 1F                     LBI     YDP
053 0F            DPMIN1:  LBI     XDP
054 AD                     JSRP    MINUS1
055 C0                     JP      ALIGN
056 1D            DP0GT1:  LBI     YMSD    ;TESTING MSD OF M1 NOT 0
057 00                     CLRA
058 21                     SKE
059 DC                     JP      R0RSFT
05A 85            R1LSFT:  JSRP    LSFTR1
05B CE                     JP      DPPL1
05C 8D            R0RSFT:  JSRP    RSFTR0
05D D3                     JP      DPMIN1
                  ;

    0080                   .PAGE   2
                  ;THESE ARE THE BASIC REQUIRED SUBROUTINES FOR THE ARITHMETIC
                  ;ROUTINES--COP420 AND LARGER CODE
                  ;
080 0F            CLEAR0:  LBI     0,0
081 00            CLEAR:   CLRA
082 04                     XIS
083 81                     JP      CLEAR
084 48                     RET
085 3397          LSFTR1:  LBI     YLSD
087 3387          LSFTR0:  LBI     XLSD
089 00            LSFTX:   CLRA
08A 04            LSFT:    XIS
08B 8A                     JP      LSFT
08C 48                     RET
08D 0E            RSFTR0:  LBI     0,15
08E 1E            RSFTR1:  LBI     1,15
```

Figure 5-3. BCD Arithmetic Package (Sheet 3 of 9)

```
COP CROSS ASSEMBLER     PAGE:     4
MAINPR


08F 00       RSFTRX: CLRA
090 07       RSFT:   XDS
091 23A2             XAD     SAVE1    ;SAVE VALUE TEMPORARILY
093 4E               CBA              ;ONLY WANT 8 DIGIT SHIFT
094 59               AISC    9
095 99               JP      DONE
096 2322             LDD     SAVE1    ;FETCH SAVED VALUE
098 90               JP      RSFT
099 2322     DONE:   LDD     SAVE1
09B 48               RET
09C 32       BCDADD: RC
09D 15       BCD1:   LD      1        ;TWO REGISTER BCD ADDITION
09E 56               AISC    6
09F 30               ASC
0A0 4A               ADT
0A1 14               XIS     1
0A2 9D               JP      BCD1
0A3 48               RET
0A4 22       BCDSUB: SC               ;TWO REGISTER BCD SUBTRACTION
0A5 15       BCDS1:  LD      1
0A6 10               CASC
0A7 4A               ADT
0A8 14               XIS     1
0A9 A5               JP      BCDS1
0AA 48               RET
0AB 2F       ZDPMN1: LBI     ZDP
0AC 3F       OFLMN1: LBI     OFLOW
0AD 05       MINUS1: LD               ;SUBTRACT 1 FROM MEMORY
0AE 5F               AISC    15
0AF 44               NOP
0B0 06       PLUS1A: X
0B1 48               RET
0B2 0F       XDPPL1: LBI     XDP
0B3 3F       OFLPL1: LBI     OFLOW
0B4 2F       ZDPPL1: LBI     ZDP
0B5 05       PLUS1:  LD               ;ADD 1 TO MEMORY
0B6 51               AISC    1
0B7 B0               JP      PLUS1A   ;WILL SKIP IF GREATER THAN 15
0B8 06               X
0B9 49               RETSK
0BA 25       XFER2:  LD      2
0BB 24               XIS     2
0BC BA               JP      XFER2
0BD 48               RET
             ;

    0100             .PAGE   4
             ;THIS IS THE ADD/SUBTRACT ROUTINE.  ROUTINE IS FOR 8 DIGITS.
             ;FLOATING POINT, FULLY ALGEBRAIC.
             ;
100 3E       ADDSUB: LBI     FLAGS
101 03               SKMBZ   2        ;TEST IF SHOULD SUBTRACT
102 D0               JP      CHNGMO   ;CHANGE SIGN RO(X) IF SUBTRACT
```

Figure 5-3. BCD Arithmetic Package (Sheet 4 of 9)

```
103 3381   ADSB1:  LBI     XSIGN    ;NOW TEST FOR SIGNS EQUAL
105 15             LD      1
106 21             SKE
107 D7             JP      SUB      ;NOT EQUAL,HENCE SUBTRACT
108 3387   ADD:    LBI     XLSD
10A 9C             JSRP    BCDADD   ;R1+R0-->R1,(Y+X-->Y)
10B 1E     ERRCHK: LBI     YGUARD   ;TEST FOR OVERFLOW
10C 00             CLRA             ;IF 1,15(YGUARD) NOT 0,UNDERFLOW
10D 21             SKE
10E ED             JP      UNDRFL
10F 48             RET
110 3381   CHNGMO: LBI     XSIGN    ;CHANGE SIGN OF R0(X)
112 05             LD
113 58             AISC    8
114 44             NOP
115 06             X
116 C3             JP      ADSB1
117 3387   SUB:    LBI     XLSD
119 A4             JSRP    BCDSUB   ;R1-R0-->R1,(Y-X-->Y)
11A 20             SKC              ;SEE IF MUST COMPLEMENT
11B DD             JP      COMPL
11C CB             JP      ERRCHK
11D 3397   COMPL:  LBI     YLSD     ;NEGATIVE RESULT,COMPLEMENT
11F 22             SC
120 00     COMPL1: CLRA
121 06             X
122 10             CASC
123 4A             ADT
124 04             XIS
125 E0             JP      COMPL1
126 3391           LBI     YSIGN    ;NOW CHANGE SIGN OF R1(Y)
128 05             LD
129 58             AISC    8
12A 44             NOP
12B 06             X
12C CB             JP      ERRCHK
12D 8E     UNDRFL: JSRP    RSFTR1   ;DO AN UNDERFLOW
12E 1F             LBI     YDP      ;ERROR IF YDP IS 0 WHEN UNDERFLOW
12F AD             JSRP    MINUS1
130 5F             AISC    15
131 F3             JP      ERROR
132 48             RET
133 1F     ERROR:  LBI     YDP
134 7F             STII    15       ;15-->YDP & YSIGN FOR ERROR
135 7F             STII    15
136 48             RET

    0140           .PAGE   5
           ;MULTIPLY,DIVIDE ROUTINES. FLOATING POINT,8 DIGIT
           ;
140 3387   MULDIV: LBI     XLSD
142 BA             JSRP    XFER2    ;M0-->M2,X-->Z, THEN CLEAR X
143 25             LD      2        ;TRANSFER DP AND SIGN ALSO
```

Figure 5-3. BCD Arithmetic Package (Sheet 5 of 9)

```
144 24             XIS     2
145 25             LD      2
146 26             X       2
147 80             JSRP    CLEAR0   ;CLEAR M0
148 3E             LBI     FLAGS    ;NOW TEST IF MULTIPLY OR DIVIDE
149 13             SKMBZ   3
14A 61C0           JMP     MULPLY
           ;
14C 22    DIVIDE:  SC               ;M0/M1 --> M0,(X/Y-->X)
14D 1F             LBI     YDP      ;DP2-DP0-->DP2,(DPZ-DPX-->DPZ)
14E 35             LD      3
14F 10             CASC
150 44             NOP
151 06             X
152 3F             LBI     OFLOW    ;15 TO OFLOW DIGIT IF BORROW,ELSE 0
153 00             CLRA
154 20             SKC
155 40             COMP
156 06             X
157 3397  DIV1A:   LBI     YLSD
159 A4             JSRP    BCDSUB   ;M0 - M1 TO M0,M1 SAVED
15A 20             SKC              ;PART OF THE REPEATED SUBTRACT FEATURE
15B E2             JP      DIV3A
15C 33A7  DIV3:    LBI     ZLSD     ;DIVIDE BY 0 CHECK
15E B5             JSRP    PLUS1
15F D7             JP      DIV1A    ;ALL OK,CONTINUE
160 6189           JMP     DIVBY0
162 3397  DIV3A:   LBI     YLSD
164 9C             JSRP    BCDADD   ;RESTORE VALUE
165 0F             LBI     XDP
166 05             LD
167 57             AISC    7
168 617F           JMP     DIV1B    ;TESTING DP FOR FINISHED
16A 2E    DIV4:    LBI     ZGUARD
16B 00             CLRA
16C 21             SKE
16D 61E5           JMP     MDEND1
16F 3F             LBI     OFLOW
170 21             SKE              ;TEST OVERFLOW DIGIT
171 F8             JP      DIV4A
172 2F             LBI     ZDP      ;TEST DP2(ZDP) >= 9
173 05             LD
174 57             AISC    7
175 F8             JP      DIV4A
176 61E5           JMP     MDEND1
178 B4    DIV4A:   JSRP    ZDPPL1   ;DP2+1-->DP2,(ZDP+1-->ZDP)
179 6181           JMP     DIV1B2
17B B3             JSRP    OFLPL1 ;INCREMENT OVERFLOW DIGIT
17C 44             NOP              ;DEFEAT SKIP
17D 6181           JMP     DIV1B2
17F B2    DIV1B:   JSRP    XDPPL1   ;DP0 + 1 --> DP0

180 44             NOP
```

Figure 5-3. BCD Arithmetic Package (Sheet 6 of 9)

```
181 33A7    DIV1B2: LBI     ZLSD
183 89              JSRP    LSFTX
184 3387            LBI     XLSD
186 8A              JSRP    LSFT
187 6157            JMP     DIV1A
189 621B    DIVBYO: JMP     MDERR


    01C0            .PAGE   7
1C0 32      MULPLY: RC              ;DP1+DP2-->DP2,(DPY+DPZ-->DPZ)
1C1 1F              LBI     YDP
1C2 35              LD      3
1C3 30              ASC
1C4 44              NOP
1C5 06              X
1C6 00              CLRA            ;1 TO OFLOW IF CARRY,ELSE 0
1C7 20              SKC
1C8 CA              JP      MUL1A
1C9 51              AISC    1
1CA 3F      MUL1A:  LBI     OFLOW
1CB 06              X
1CC 33A7    MUL1:   LBI     ZLSD
1CE 05              LD
1CF 5F              AISC    15      ;LSD CONTROLLING REPEATED ADDS
1D0 D6              JP      MUL2
1D1 06              X
1D2 3397            LBI     YLSD    ;M0 + M1 --> M0,(X+Y-->X)
1D4 9C              JSRP    BCDADD
1D5 CC              JP      MUL1
1D6 8D      MUL2:   JSRP    RSFTR0
1D7 2E              LBI     ZGUARD
1D8 90              JSRP    RSFT
1D9 B2              JSRP    XDPPL1
1DA 58              AISC    8
1DB CC              JP      MUL1    ;PRECEEDING IS DP ADJUST
1DC 78      MUL3:   STII    8
1DD 3387            LBI     XLSD
1DF 00      MUL3X:  CLRA            ;TEST M0=0(X=0)
1E0 21              SKE
1E1 6212            JMP     MUL5
1E3 04              XIS
1E4 DF              JP      MUL3X
1E5 2E      MDEND1: LBI     ZGUARD
1E6 00              CLRA
1E7 21              SKE
1E8 ED              JP      MDX
1E9 2F              LBI     ZDP
1EA 05              LD
1EB 57              AISC    7       ;TEST >= 9
1EC F7              JP      MDEND2
1ED 2E      MDX:    LBI     ZGUARD
1EE 8F              JSRP    RSFTRX
1EF 3386            LBI     ROUND   ;SAVE VALUE FOR ROUNDING
1F1 06              X
```

Figure 5-3. BCD Arithmetic Package (Sheet 7 of 9)

```
1F2 AB               JSRP    ZDPMN1
1F3 05               LD              ;TEST DP2(ZDP) = 15
1F4 51               AISC    1
1F5 F7               JP      MDEND2
1F6 AC               JSRP    OFLMN1  ;SUBTRACT 1 FROM OVERFLOW DIGIT
1F7 2F      MDEND2:  LBI     2,0     ;TRANSFER R2 TO R0
1F8 BA               JSRP    XFER2
1F9 3391             LBI     YSIGN   ;ADD SIGNS AND PUT TO M0(X)
1FB 15               LD      1
1FC 31               ADD
1FD 06               X
1FE 3F               LBI     OFLOW   ;TEST OVERFLOW DIGIT
1FF 05               LD

200 51               AISC    1
201 C3               JP      MDEND4  ;NOT 15
202 DB               JP      MDERR   ;IS 15,NUMBER TOO BIG
203 00      MDEND4:  CLRA            ;NOW TEST DIGIT > 0
204 21               SKE
205 80               JSRP    CLEAR0  ;IS NON ZERO,CLEAR M0
206 3387    MDRJ:    LBI     XLSD    ;RIGHT JUSTIFY THE RESULT
208 00               CLRA
209 21               SKE
20A 48               RET             ;IF LSD NON ZERO,STOP
20B 0F               LBI     XDP     ;IF DP PSN = 0,STOP
20C 05               LD
20D 5F               AISC    15
20E 48               RET
20F 06               X               ;ELSE,DECREMENT BY 1 AND CONTINUE
210 8D               JSRP    RSFTR0
211 C6               JP      MDRJ
212 2F      MUL5:    LBI     ZDP     ;TEST DP2(ZDP) = 0
213 00               CLRA
214 21               SKE
215 D8               JP      MUL3A
216 40               COMP            ;15 TO DP2(ZDP)
217 06               X
218 AD      MUL3A:   JSRP    MINUS1  ;DP2(ZDP) - 1 --> DP2(ZDP)
219 61D6             JMP     MUL2
21B 0E      MDERR:   LBI     0,15
21C 7F               STII    15
21D 7F               STII    15
21E 48               RET
                     .END
```

Figure 5-3. BCD Arithmetic Package (Sheet 8 of 9)

```
ADD     0108 *   ADDSUB 0100     ADSB1   0103     ALIGN   0040
ALIGN2  0046     BCD1    009D     BCDADD  009C     BCDS1   00A5
BCDSUB  00A4     CHNGMO  0110     CLEAR   0081     CLEARO  0080
COMPL   011D     COMPL1  0120     DIV1A   0157     DIV1B   017F
DIV1B2  0181     DIV3    015C *   DIV3A   0162     DIV4    016A *
DIV4A   0178     DIVBYO  0189     DIVIDE  014C *   DONE    0099
DPOGT1  0056     DPOLT1  0048 *   DPMIN1  0053     DPPL1   004E
ERRCHK  010B     ERROR   0133     FLAGS   003F     JSMD    0024
JSRALN  0010     LSFT    008A     LSFTRO  0087     LSFTR1  0085
LSFTX   0089     MDEND1  01E5     MDEND2  01F7     MDEND4  0203
MDERR   021B     MDRJ    0206     MDX     01ED     MINUS1  00AD
MUL1    01CC     MUL1A   01CA     MUL2    01D6     MUL3    01DC *
MUL3A   0218     MUL3X   01DF     MUL5    0212     MULDIV  0140
MULPLY  01C0     OFLMN1  00AC     OFLOW   0030     OFLPL1  00B3
PLUS1   00B5     PLUS1A  00B0     ROLSFT  004C *   RORSFT  005C
R1LSFT  005A *   R1RSFT  0051     RAMCLR  0003     ROUND   0006
RSFT    0090     RSFTRO  008D     RSFTR1  008E     RSFTRX  008F
SAVE1   0022     SUB     0117     TESTG   0009     TESTG1  0013
TESTG2  0019     TESTG3  001F     UNDRFL  012D     XDP     0000
XDPPL1  00B2     XFER2   00BA     XGUARD  000F *   XLSD    0007
XMSD    000E     XSIGN   0001     YDP     0010     YGUARD  001F
YLSD    0017     YMSD    001E     YSIGN   0011     ZDP     0020
ZDPMN1  00AB     ZDPPL1  00B4     ZGUARD  002F     ZLSD    0027
ZMSD    002E *   ZSIGN   0021 *

NO ERROR LINES

  356  ROM WORDS USED

COP   420  ASSEMBLY

SOURCE CHECKSUM = FBE9

INPUT FILE    ABDUL10:ARITH.SRC  VN:   20
```

Figure 5-3. BCD Arithmetic Package (Sheet 9 of 9)

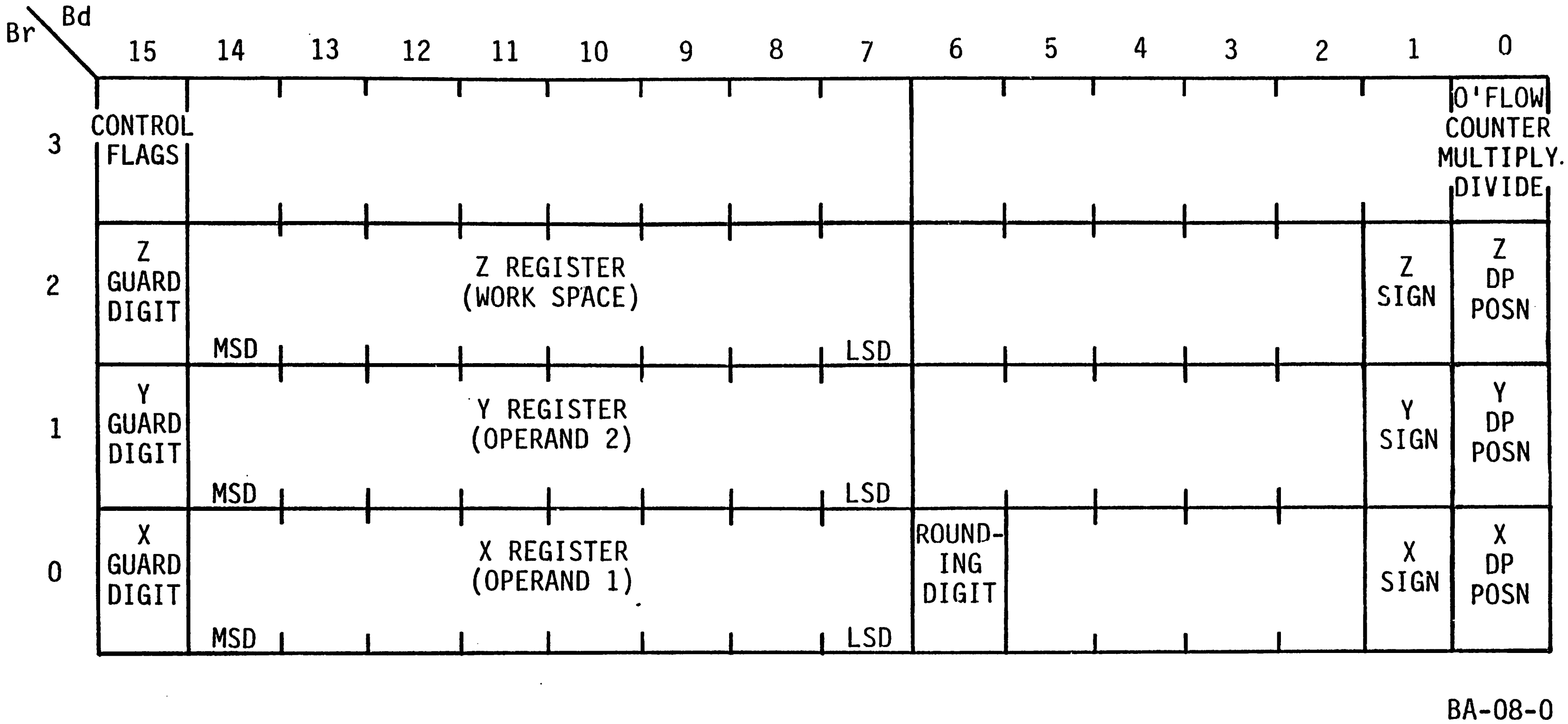


Figure 5-4. RAM Map - Basic Arithmetic Routines

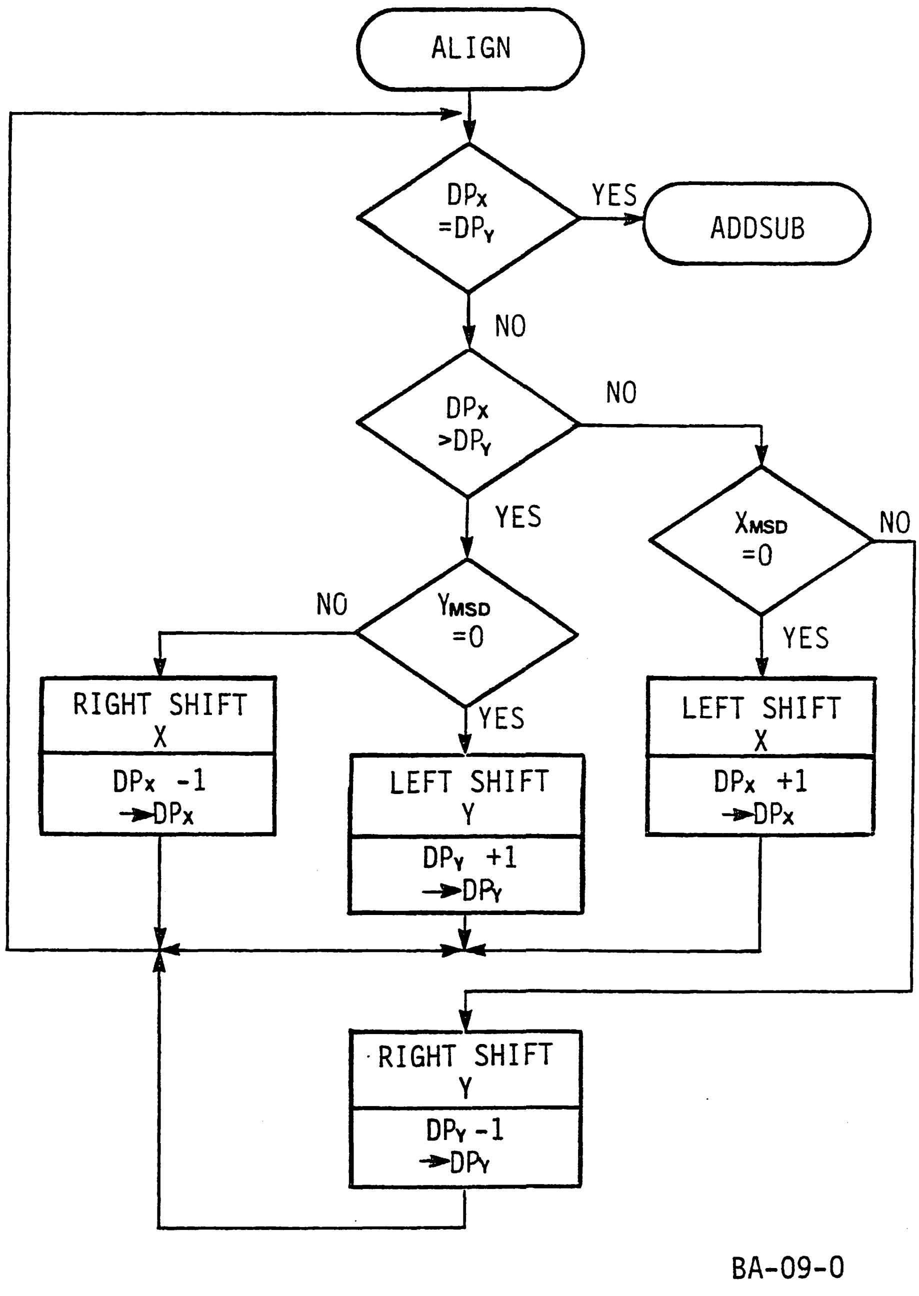
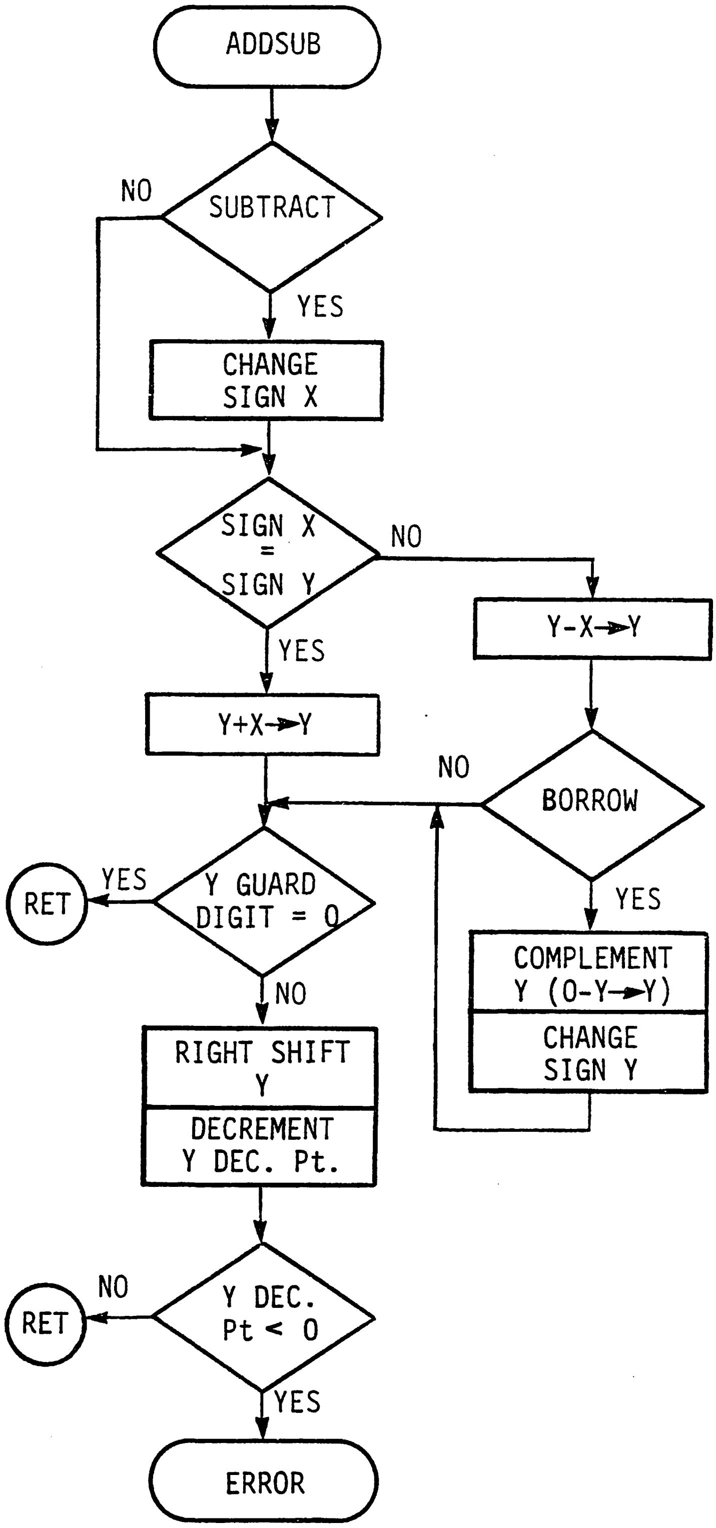


Figure 5-5. Align Routine for Add/Subtract



BA-10-0

Figure 5-6. Fully Algebraic Add/Subtract

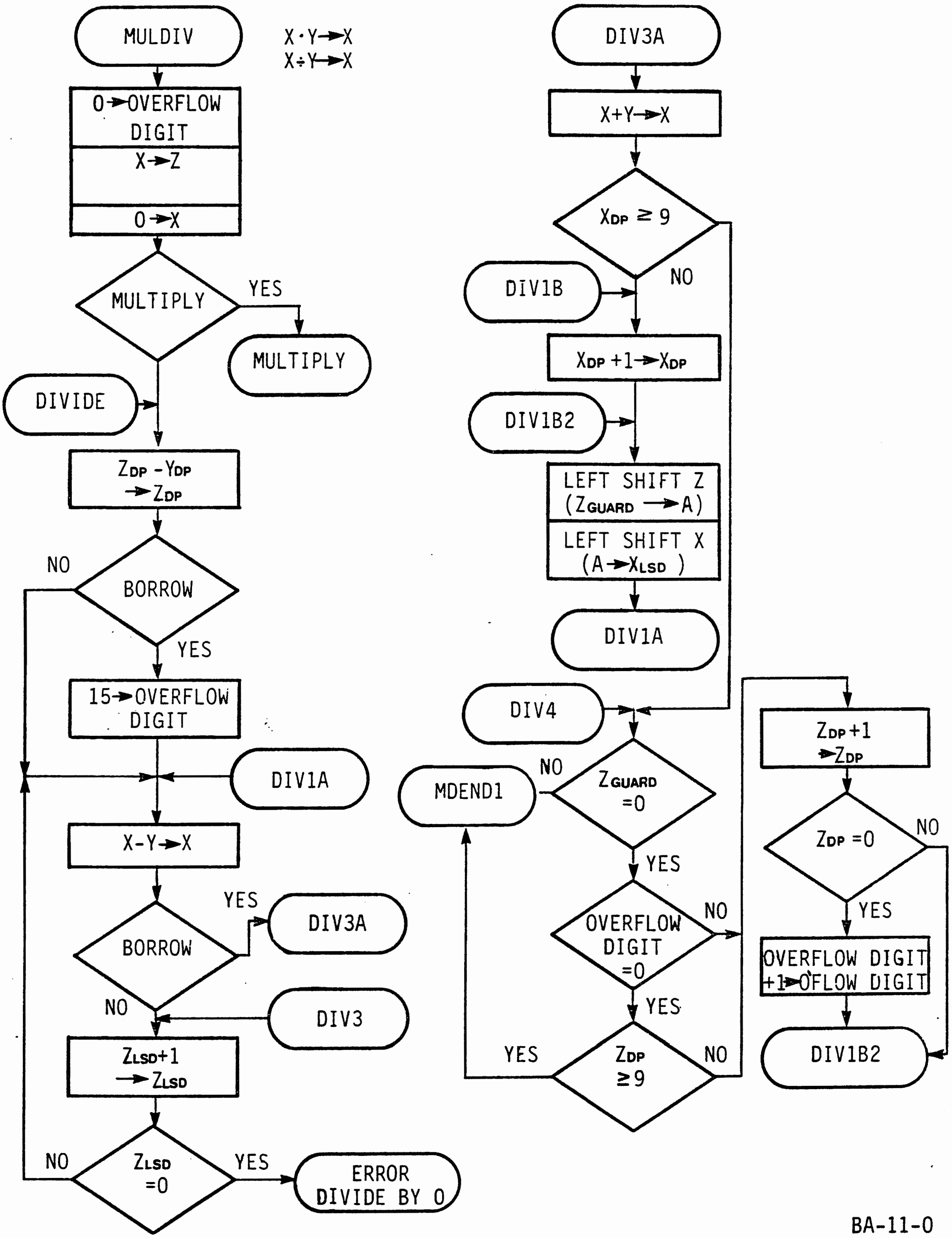


Figure 5-7. Multiply/Divide (Sheet 1 of 3)

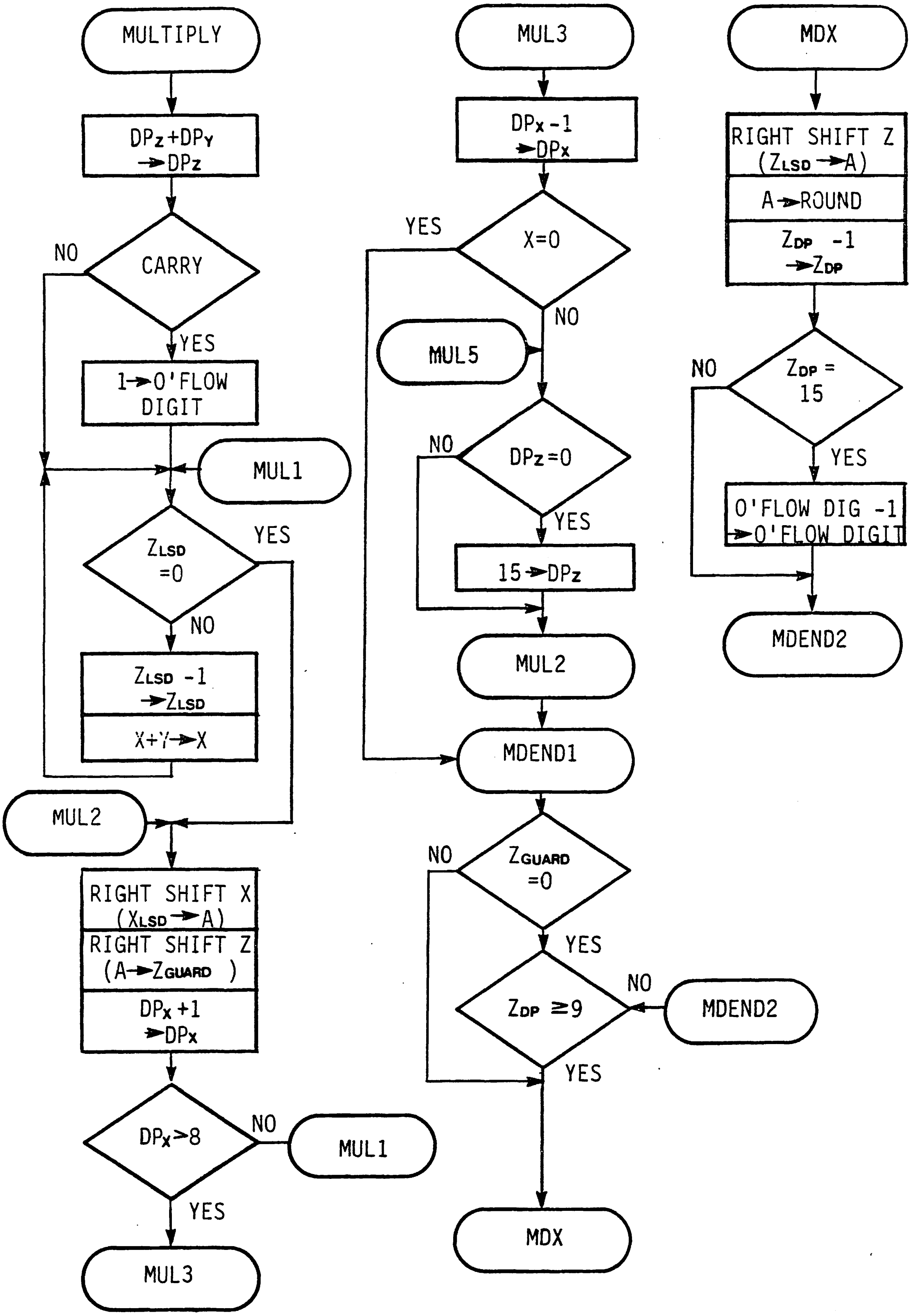


Figure 5-7. Multiply/Divide (Sheet 2 of 3)

BA-12-0

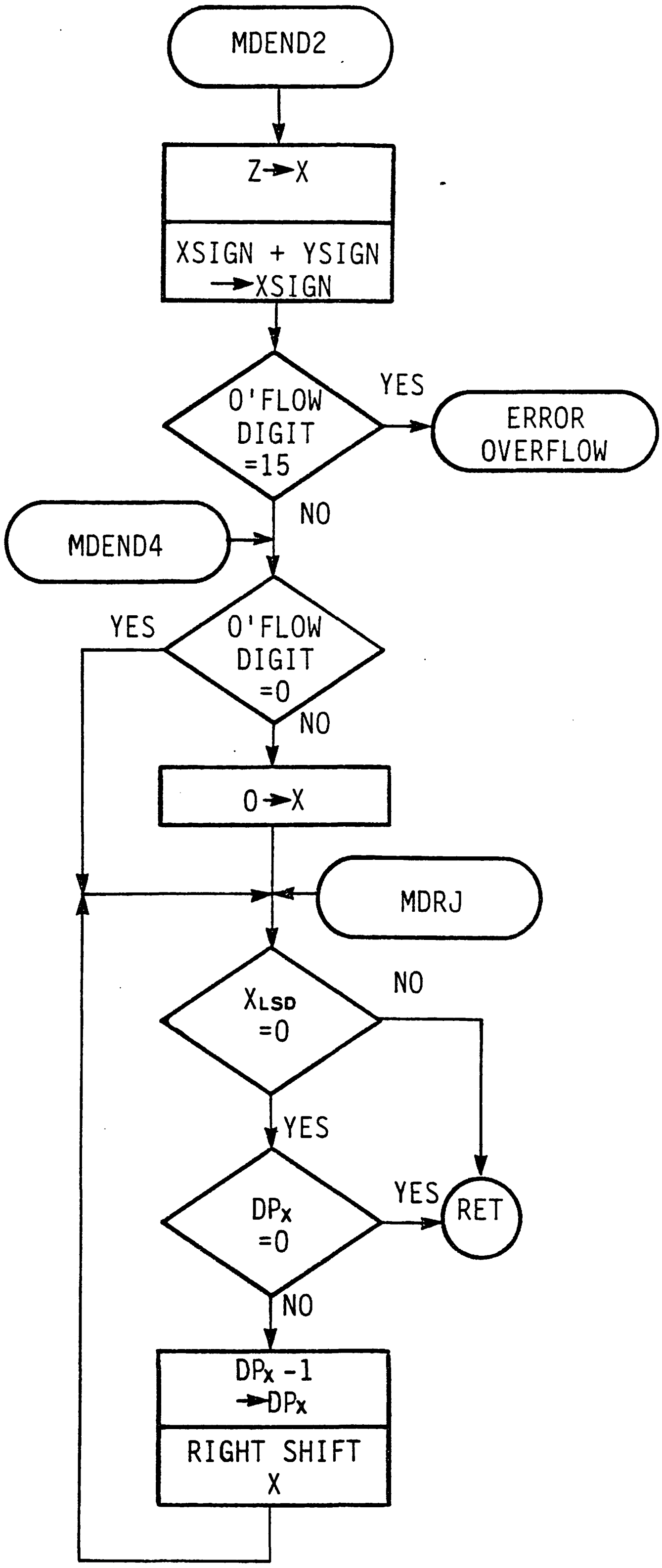


Figure 5-7. Multiply/Divide (Sheet 3 of 3)

```
COP CROSS ASSEMBLER    PAGE:    1
HXMATH  HEXADECIMAL ARITHMETIC
```

```
                ;BASIC HEXADECIMAL(BINARY) FLOATING POINT ARITHMETIC ROUTINES
                ;
                ;REGISTER 0 = X,REGISTER 1 = Y, REGISTER 2 = Z
                ;
                ;THE ROUTINES ARE FOR 8 DIGIT,HEX,FULLY ALGEBRAIC ADD,SUBTRACT
                ;MULTIPLY AND DIVIDE. ALL ROUTINES ARE FULLY FLOATING POINT.
                ;THE ROUTINES ASSUME AN 8 DIGIT MANTISSA, A SIGN DIGIT, AND A
                ;HEX POINT DIGIT. THE HEX POINT DIGIT IS A HEX POINT
                ;POSITION INDICATOR,I.E.,A HEX PT.POSITION OF 0 INDICATES
                ;THAT THE HEX POINT IS PLACED AFTER THE LSD OF THE NUMBER;
                ;HEX POINT POSITION OF 7 INDICATES THAT THE HEX POINT IS
                ;PLACED AFTER THE MSD OF THE NUMBER. OTHER NUMBERS CORRESPOND
                ;IN THE SAME MANNER TO INTERMEDIATE DIGITS.
                ;
                ;THE ROUTINES ALSO ASSUME THAT THERE IS A GUARD OR OVERFLOW
                ;DIGIT FOR THE NUMBERS.THE MANTISSA IS 8 DIGITS PLUS THE GUARD
                ;DIGIT FOR A TOTAL OF 9 DIGITS.THE GUARD DIGIT IS FOR INTERNAL
                ;USE ONLY AND IS NOT AVAILABLE ON INPUT OR OUTPUT.
                ;
                ;THE ROUTINES ARE USABLE AS IS FOR BINARY ARITHMETIC DUE TO
                ;THE OBVIOUS RELATIONSHIP BETWEEN HEX AND BINARY.  THE ONLY
                ;ADVERSE EFFECT IS THAT THE RAM IS NOT OPTIMALLY USED IF
                ;BINARY ARITHMETIC IS DESIRED.  NONETHELESS THE ROUTINES
                ;ARE FULLY FUNCTIONAL FOR BINARY ARITHMETIC.
                ;
                ;THE CODE AS WRITTEN SHOULD WORK IN COP420 AND LARGER DEVICES.
                ;THE ROUTINES ARE WRITTEN AS SUBROUTINES CALLED BY A MAIN
                ;PROGRAM.  ONE LEVEL OF SUBROUTINE IS USED BY THE ARITHMETIC
                ;ROUTINES.  COMPARABLE ROUTINES CAN BE WRITTEN FOR THE COP410
                ;BUT SOME CHANGES ARE REQUIRED. THE ALGORITHM IS STILL VALID
                ;ALTHOUGH THE IMPLEMENTATION IS SOMEWHAT DIFFERENT.
                ;
                ;IT WILL BE NOTED THAT THESE HEX ROUTINES DIFFER ONLY SLIGHTLY
                ;FROM THE EQUIVALENT DECIMAL ROUTINES.  WITH THIS INFORMATION
                ;IT WOULD BE POSSIBLE, IF IT WERE NECESSARY, TO WRITE A
                ;COMMON ROUTINE AND DO THE ARITHMETIC IN HEX OR DECIMAL WHICH-
                ;EVER WAS REQUIRED.  THE EXTRA CODE TO DO THIS WOULD NOT BE
                ;SIGNIFICANT IF THE APPLICATION GENUINELY REQUIRED THIS DUAL
                ;CAPABILITY.
                ;
                         .TITLE    HXMATH,'HEXADECIMAL ARITHMETIC'
        0022    SAVE1    =         2,2
        000F    XGUARD   =         0,15
        000E    XMSD     =         0,14
        0007    XLSD     =         0,7
        0006    ROUND    =         0,6
        0001    XSIGN    =         0,1
        0000    XDP      =         0,0
        001F    YGUARD   =         1,15
        001E    YMSD     =         1,14
        0017    YLSD     =         1,7
        0011    YSIGN    =         1,1
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 1 of 9)

```
    0010    YDP     =       1,0
    002F    ZGUARD  =       2,15
    002E    ZMSD    =       2,14
    0027    ZLSD    =       2,7
    0021    ZSIGN   =       2,1
    0020    ZDP     =       2,0
    003F    FLAGS   =       3,15
    0030    OFLOW   =       3,0
                    ;

    0000                    .PAGE   0
000 00                      CLRA
001 0F                      LBI     0,0      ;CLEAR ALL THE RAM
002 53                      AISC    3
003 12              RAMCLR: XABR
004 81                      JSRP    CLEAR
005 12                      XABR
006 5F                      AISC    15
007 C9                      JP      TESTG
008 C3                      JP      RAMCLR
                    ;
                    ;*************************************************************
                    ;FOLLOWING CODE--TO NEXT LINE OF **--IS FOR CONTROL ONLY
                    ;
009 335F            TESTG:  OGI     15       ;PUT G LINES HIGH FOR READING G
                    ;
                            ;USING G LINES FOR PRIMITIVE CONTROL TO SELECT ADD,SUB
                            ;MULTIPLY OR DIVIDE--WILL ENTER NUMBERS IN BREAKPOINT
                            ;MODE USING MODIFY COMMAND
                            ;
00B 3301                    SKGBZ   0
00D D3                      JP      TESTG1
00E 3E                      LBI     FLAGS
00F 70                      STII    0        ;RESET BIT 2 FOR ADD
010 6840            JSRALN: JSR     ALIGN
012 C9                      JP      TESTG
013 3311            TESTG1: SKGBZ   1
015 D9                      JP      TESTG2
016 3E                      LBI     FLAGS    ;SET SUBTRACT BIT
017 74                      STII    4        ;SET BIT 2 FOR SUBTRACT
018 D0                      JP      JSRALN
019 3303            TESTG2: SKGBZ   2
01B DF                      JP      TESTG3
01C 3E                      LBI     FLAGS
01D 70                      STII    0        ;RESET BIT 3 FOR DIVIDE
01E E4                      JP      JSMD
01F 3313            TESTG3: SKGBZ   3
021 C9                      JP      TESTG
022 3E                      LBI     FLAGS
023 78                      STII    8        ;SET BIT 3 FOR MULTIPLY
024 6940            JSMD:   JSR     MULDIV
026 C9                      JP      TESTG
                    ;
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 2 of 9)

```
                ;PRECEEDING CODE IS FOR CONTROL ONLY,HAS NOTHING TO DO WITH THE
                ;ARITHMETIC ALGORITHMS
                ;**************************************************************
                ;

      0040              .PAGE    1
                ; THIS IS THE ALIGN ROUTINE FOR ADD/SUBTRACT.  IT MAKES THE
                ; HEX POSITIONS OF THE TWO NUMBERS EQUAL BEFORE ADD OR
                ; SUBTRACT TAKES PLACE.  THE ROUTINE ASSUMES THAT THE NUMBERS
                ; ARE RIGHT JUSTIFIED ON ENTRY.  HEX POINT POSITION VALUES
                ; ARE RESTRICTED TO NUMBERS BETWEEN 0 - 8 (SINCE WE ARE ONLY
                ; DOING 8 DIGIT ROUTINES).  ROUTINE ONLY REQUIRED FOR FLOATING
                ; POINT ADD/SUBTRACT ALGORITHMS
                ;
040 0F          ALIGN:  LBI     XDP
041 15                  LD      1
042 21                  SKE             ;TEST DP0=DP1(DPX=DPY)
043 C6                  JP      ALIGN2
044 6100                JMP     ADDSUB  ;IF EQUAL,PROCEED TO ADD/SUBTRACT
046 10          ALIGN2: CASC            ;TEST DP0 > DP1
047 D6                  JP      DP0GT1  ;YES
048 0D          DP0LT1: LBI     XMSD    ;DP0<DP1.IF XMSD NOT 0,RIGHT SHIFT
049 00                  CLRA            ;M1,ELSE LEFT SHIFT M0
04A 21                  SKE
04B D1                  JP      R1RSFT
04C 87          R0LSFT: JSRP    LSFTR0
04D 0F                  LBI     XDP
04E 1F          DPPL1:  LBI     YDP
04F B4                  JSRP    PLUS1   ;MODIFY DP AFTER SHIFT
050 C0                  JP      ALIGN
051 8E          R1RSFT: JSRP    RSFTR1
052 1F                  LBI     YDP
053 0F          DPMIN1: LBI     XDP
054 AC                  JSRP    MINUS1
055 C0                  JP      ALIGN
056 1D          DP0GT1: LBI     YMSD    ;TESTING MSD OF M1 NOT 0
057 00                  CLRA
058 21                  SKE
059 DC                  JP      R0RSFT
05A 85          R1LSFT: JSRP    LSFTR1
05B CE                  JP      DPPL1
05C 8D          R0RSFT: JSRP    RSFTR0
05D D3                  JP      DPMIN1
                ;

      0080              .PAGE   2
                ;THESE ARE THE BASIC REQUIRED SUBROUTINES FOR THE ARITHMETIC
                ;ROUTINES--COP420 AND LARGER CODE
                ;
080 0F          CLEAR0: LBI     0,0
081 00          CLEAR:  CLRA
082 04                  XIS
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 3 of 9)

```
083 81                JP      CLEAR
084 48                RET
085 3397    LSFTR1:   LBI     YLSD
087 3387    LSFTRO:   LBI     XLSD
089 00      LSFTX:    CLRA
08A 04      LSFT:     XIS
08B 8A                JP      LSFT
08C 48                RET
08D 0E      RSFTRO:   LBI     0,15
08E 1E      RSFTR1:   LBI     1,15
08F 00      RSFTRX:   CLRA
090 07      RSFT:     XDS
091 23A2              XAD     SAVE1     ;SAVE VALUE TEMPORARILY
093 4E                CBA               ;ONLY WANT 8 DIGIT SHIFT
094 59                AISC    9
095 99                JP      DONE
096 2322              LDD     SAVE1     ;FETCH SAVED VALUE
098 90                JP      RSFT
099 2322    DONE:     LDD     SAVE1
09B 48                RET
09C 32      BINADD:   RC
09D 15      BIN1:     LD      1         ;TWO REGISTER BINARY ADDITION
09E 30                ASC
09F 44                NOP
0A0 14                XIS     1
0A1 9D                JP      BIN1
0A2 48                RET
0A3 22      BINSUB:   SC                ;TWO REGISTER BINARY SUBTRACTION
0A4 15      BINS1:    LD      1
0A5 10                CASC
0A6 44                NOP
0A7 14                XIS     1
0A8 A4                JP      BINS1
0A9 48                RET
0AA 2F      ZDPMN1:   LBI     ZDP
0AB 3F      OFLMN1:   LBI     OFLOW
0AC 05      MINUS1:   LD                ;SUBTRACT 1 FROM MEMORY
0AD 5F                AISC    15
0AE 44                NOP
0AF 06      PLUS1A:   X
0B0 48                RET
0B1 0F      XDPPL1:   LBI     XDP
0B2 3F      OFLPL1:   LBI     OFLOW
0B3 2F      ZDPPL1:   LBI     ZDP
0B4 05      PLUS1:    LD                ;ADD 1 TO MEMORY
0B5 51                AISC    1
0B6 AF                JP      PLUS1A    ;WILL SKIP IF GREATER THAN 15
0B7 06                X
0B8 49                RETSK
0B9 25      XFER2:    LD      2
0BA 24                XIS     2
0BB B9                JP      XFER2
0BC 48                RET
            ;
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 4 of 9)

```
COP CROSS ASSEMBLER    PAGE:    5
HXMATH  HEXADECIMAL ARITHMETIC


        0100             .PAGE    4
                ;THIS IS THE ADD/SUBTRACT ROUTINE.  ROUTINE IS FOR 8 DIGITS.
                ;HEXADECIMAL,FLOATING POINT, FULLY ALGEBRAIC.
                ;
    100 3E      ADDSUB: LBI      FLAGS
    101 03              SKMBZ    2        ;TEST IF SHOULD SUBTRACT
    102 D0              JP       CHNGMO   ;CHANGE SIGN RO(X) IF SUBTRACT
    103 3381    ADSB1:  LBI      XSIGN    ;NOW TEST FOR SIGNS EQUAL
    105 15              LD       1
    106 21              SKE
    107 D7              JP       SUB      ;NOT EQUAL,HENCE SUBTRACT
    108 3387    ADD:    LBI      XLSD
    10A 9C              JSRP     BINADD   ;R1+RO-->R1,(Y+X-->Y)
    10B 1E      ERRCHK: LBI      YGUARD   ;TEST FOR OVERFLOW
    10C 00              CLRA              ;IF 1,15(YGUARD) NOT 0,UNDERFLOW
    10D 21              SKE
    10E ED              JP       UNDRFL
    10F 48              RET
    110 3381    CHNGMO: LBI      XSIGN    ;CHANGE SIGN OF RO(X)
    112 05              LD
    113 58              AISC     8
    114 44              NOP
    115 06              X
    116 C3              JP       ADSB1
    117 3387    SUB:    LBI      XLSD
    119 A3              JSRP     BINSUB   ;R1-RO-->R1,(Y-X-->Y)
    11A 20              SKC               ;SEE IF MUST COMPLEMENT
    11B DD              JP       COMPL
    11C CB              JP       ERRCHK
    11D 3397    COMPL:  LBI      YLSD     ;NEGATIVE RESULT,COMPLEMENT
    11F 22              SC
    120 00      COMPL1: CLRA
    121 06              X
    122 10              CASC
    123 4A              ADT
    124 04              XIS
    125 E0              JP       COMPL1
    126 3391            LBI      YSIGN    ;NOW CHANGE SIGN OF R1(Y)
    128 05              LD
    129 58              AISC     8
    12A 44              NOP
    12B 06              X
    12C CB              JP       ERRCHK
    12D 8E      UNDRFL: JSRP     RSFTR1   ;DO AN UNDERFLOW
    12E 1F              LBI      YDP      ;ERROR IF YDP IS 0 WHEN UNDERFLOW
    12F AC              JSRP     MINUS1
    130 5F              AISC     15
    131 F3              JP       ERROR
    132 48              RET
    133 1F      ERROR:  LBI      YDP
    134 7F              STII     15       ;15-->YDP & YSIGN FOR ERROR
    135 7F              STII     15
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 5 of 9)

```
136 48               RET

    0140             .PAGE   5
            ;MULTIPLY,DIVIDE ROUTINES. FLOATING POINT,8 DIGIT
            ;
140 3387    MULDIV: LBI     XLSD
142 B9              JSRP    XFER2     ;M0-->M2,X-->Z, THEN CLEAR X
143 25              LD      2         ;TRANSFER DP AND SIGN ALSO
144 24              XIS     2
145 25              LD      2
146 26              X       2
147 80              JSRP    CLEAR0    ;CLEAR M0
148 3E              LBI     FLAGS     ;NOW TEST IF MULTIPLY OR DIVIDE
149 13              SKMBZ   3
14A 61C0            JMP     MULPLY
            ;
14C 22      DIVIDE: SC                ;M0/M1 --> M0,(X/Y-->X)
14D 1F              LBI     YDP       ;DP2-DP0-->DP2,(DPZ-DPX-->DPZ)
14E 35              LD      3
14F 10              CASC
150 44              NOP
151 06              X
152 3F              LBI     OFLOW     ;15 TO OFLOW DIGIT IF BORROW,ELSE 0
153 00              CLRA
154 20              SKC
155 40              COMP
156 06              X
157 3397    DIV1A:  LBI     YLSD
159 A3              JSRP    BINSUB    ;M0 - M1 TO M0,M1 SAVED
15A 20              SKC               ;PART OF THE REPEATED SUBTRACT FEATURE
15B E2              JP      DIV3A
15C 33A7    DIV3:   LBI     ZLSD      ;DIVIDE BY 0 CHECK
15E B4              JSRP    PLUS1
15F D7              JP      DIV1A     ;ALL OK,CONTINUE
160 6189            JMP     DIVBY0
162 3397    DIV3A:  LBI     YLSD
164 9C              JSRP    BINADD    ;RESTORE VALUE
165 0F              LBI     XDP
166 05              LD
167 57              AISC    7
168 617F            JMP     DIV1B     ;TESTING DP FOR FINISHED
16A 2E      DIV4:   LBI     ZGUARD
16B 00              CLRA
16C 21              SKE
16D 61E5            JMP     MDEND1
16F 3F              LBI     OFLOW
170 21              SKE               ;TEST OVERFLOW DIGIT
171 F8              JP      DIV4A
172 2F              LBI     ZDP       ;TEST DP2(ZDP) >= 9
173 05              LD
174 57              AISC    7
175 F8              JP      DIV4A
176 61E5            JMP     MDEND1
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 6 of 9)

```
COP CROSS ASSEMBLER    PAGE:    7
HXMATH  HEXADECIMAL ARITHMETIC

178 B3      DIV4A:   JSRP    ZDPPL1   ;DP2+1-->DP2,(ZDP+1-->ZDP)
179 6181             JMP     DIV1B2
17B B2               JSRP    OFLPL1  ;INCREMENT OVERFLOW DIGIT
17C 44               NOP              ;DEFEAT SKIP
17D 6181             JMP     DIV1B2
17F B1      DIV1B:   JSRP    XDPPL1   ;DP0 + 1 --> DP0

180 44               NOP
181 33A7    DIV1B2:  LBI     ZLSD
183 89               JSRP    LSFTX
184 3387             LBI     XLSD
186 8A               JSRP    LSFT
187 6157             JMP     DIV1A
189 621B    DIVBY0:  JMP     MDERR

    01C0             .PAGE   7
1C0 32      MULPLY:  RC               ;DP1+DP2-->DP2,(DPY+DPZ-->DPZ)
1C1 1F               LBI     YDP
1C2 35               LD      3
1C3 30               ASC
1C4 44               NOP
1C5 06               X
1C6 00               CLRA             ;1 TO OFLOW IF CARRY,ELSE 0
1C7 20               SKC
1C8 CA               JP      MUL1A
1C9 51               AISC    1
1CA 3F      MUL1A:   LBI     OFLOW
1CB 06               X
1CC 33A7    MUL1:    LBI     ZLSD
1CE 05               LD
1CF 5F               AISC    15       ;LSD CONTROLLING REPEATED ADDS
1D0 D6               JP      MUL2
1D1 06               X
1D2 3397             LBI     YLSD     ;M0 + M1 --> M0,(X+Y-->X)
1D4 9C               JSRP    BINADD
1D5 CC               JP      MUL1
1D6 8D      MUL2:    JSRP    RSFTR0
1D7 2E               LBI     ZGUARD
1D8 90               JSRP    RSFT
1D9 B1               JSRP    XDPPL1
1DA 58               AISC    8
1DB CC               JP      MUL1     ;PRECEEDING IS DP ADJUST
1DC 78      MUL3:    STII    8
1DD 3387             LBI     XLSD
1DF 00      MUL3X:   CLRA             ;TEST M0=0(X=0)
1E0 21               SKE
1E1 6212             JMP     MUL5
1E3 04               XIS
1E4 DF               JP      MUL3X
1E5 2E      MDEND1:  LBI     ZGUARD
1E6 00               CLRA
1E7 21               SKE
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 7 of 9)

```
1E8 ED                JP      MDX
1E9 2F                LBI     ZDP
1EA 05                LD
1EB 57                AISC    7       ;TEST >= 9
1EC F7                JP      MDEND2
1ED 2E       MDX:     LBI     ZGUARD
1EE 8F                JSRP    RSFTRX
1EF 3386              LBI     ROUND   ;SAVE VALUE FOR ROUNDING
1F1 06                X
1F2 AA                JSRP    ZDPMN1
1F3 05                LD              ;TEST DP2(ZDP) = 15
1F4 51                AISC    1
1F5 F7                JP      MDEND2
1F6 AB                JSRP    OFLMN1  ;SUBTRACT 1 FROM OVERFLOW DIGIT
1F7 2F       MDEND2:  LBI     2,0     ;TRANSFER R2 TO R0
1F8 B9                JSRP    XFER2
1F9 3391              LBI     YSIGN   ;ADD SIGNS AND PUT TO M0(X)
1FB 15                LD      1
1FC 31                ADD
1FD 06                X
1FE 3F                LBI     OFLOW   ;TEST OVERFLOW DIGIT
1FF 05                LD

200 51                AISC    1
201 C3                JP      MDEND4  ;NOT 15
202 DB                JP      MDERR   ;IS 15,NUMBER TOO BIG
203 00       MDEND4:  CLRA            ;NOW TEST DIGIT > 0
204 21                SKE
205 80                JSRP    CLEAR0  ;IS NON ZERO,CLEAR M0
206 3387     MDRJ:    LBI     XLSD    ;RIGHT JUSTIFY THE RESULT
208 00                CLRA
209 21                SKE
20A 48                RET             ;IF LSD NON ZERO,STOP
20B 0F                LBI     XDP     ;IF DP PSN = 0,STOP
20C 05                LD
20D 5F                AISC    15
20E 48                RET
20F 06                X               ;ELSE,DECREMENT BY 1 AND CONTINUE
210 8D                JSRP    RSFTR0
211 C6                JP      MDRJ
212 2F       MUL5:    LBI     ZDP     ;TEST DP2(ZDP) = 0
213 00                CLRA
214 21                SKE
215 D8                JP      MUL3A
216 40                COMP            ;15 TO DP2(ZDP)
217 06                X
218 AC       MUL3A:   JSRP    MINUS1  ;DP2(ZDP) - 1 --> DP2(ZDP)
219 61D6              JMP     MUL2
21B 0E       MDERR:   LBI     0,15
21C 7F                STII    15
21D 7F                STII    15
21E 48                RET
                      .END
```

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 8 of 9)

```
ADD     0108 *   ADDSUB 0100      ADSB1  0103      ALIGN  0040
ALIGN2 0046      BIN1   009D      BINADD 009C      BINS1  00A4
BINSUB 00A3      CHNGMO 0110      CLEAR  0081      CLEAR0 0080
COMPL  011D      COMPL1 0120      DIV1A  0157      DIV1B  017F
DIV1B2 0181      DIV3   015C *    DIV3A  0162      DIV4   016A *
DIV4A  0178      DIVBY0 0189      DIVIDE 014C *    DONE   0099
DPOGT1 0056      DPOLT1 0048 *    DPMIN1 0053      DPPL1  004E
ERRCHK 010B      ERROR  0133      FLAGS  003F      JSMD   0024
JSRALN 0010      LSFT   008A      LSFTR0 0087      LSFTR1 0085
LSFTX  0089      MDEND1 01E5      MDEND2 01F7      MDEND4 0203
MDERR  021B      MDRJ   0206      MDX    01ED      MINUS1 00AC
MUL1   01CC      MUL1A  01CA      MUL2   01D6      MUL3   01DC *
MUL3A  0218      MUL3X  01DF      MUL5   0212      MULDIV 0140
MULPLY 01C0      OFLMN1 00AB      OFLOW  0030      OFLPL1 00B2
PLUS1  00B4      PLUS1A 00AF      ROLSFT 004C *    RORSFT 005C
R1LSFT 005A *    R1RSFT 0051      RAMCLR 0003      ROUND  0006
RSFT   0090      RSFTR0 008D      RSFTR1 008E      RSFTRX 008F
SAVE1  0022      SUB    0117      TESTG  0009      TESTG1 0013
TESTG2 0019      TESTG3 001F      UNDRFL 012D      XDP    0000
XDPPL1 00B1      XFER2  00B9      XGUARD 000F *    XLSD   0007
XMSD   000E      XSIGN  0001      YDP    0010      YGUARD 001F
YLSD   0017      YMSD   001E      YSIGN  0011      ZDP    0020
ZDPMN1 00AA      ZDPPL1 00B3      ZGUARD 002F      ZLSD   0027
ZMSD   002E *    ZSIGN  0021 *
```

NO ERROR LINES

355 ROM WORDS USED

COP 420 ASSEMBLY

SOURCE CHECKSUM = 7921

INPUT FILE   ABDUL10:HEXARITH.SRC VN: 5

Figure 5-8. Binary (Hexadecimal) Arithmetic Package (Sheet 9 of 9)

5.2.9 Square Root

Two square root routines are provided: an integer square root and full floating-point square root. Both routines are based on the mathematical relationship:

$$\sum_{i=1}^{n} \left(2i-1\right) = n^2$$

Therefore, if sequential odd numbers were subtracted from a value, the square root of that value is given by the number of odd numbers that must be subtracted from the original value to reduce that original value to "0" (or at least to reduce the integer part to 0).

Integer Square Root - BCD

A simple routine is provided that computes the integer portion of the square root of an integer. The technique is the simple subtraction of odd numbers as described above. The flow chart for this routine is given in Figure 5-9. The code and RAM map for a four-digit routine is given below.

Br \ Bd	15	14	13	12
0	MSD	X		LSD
1		Y		
2		Z		

The subroutines are assumed to be located in Page 2.

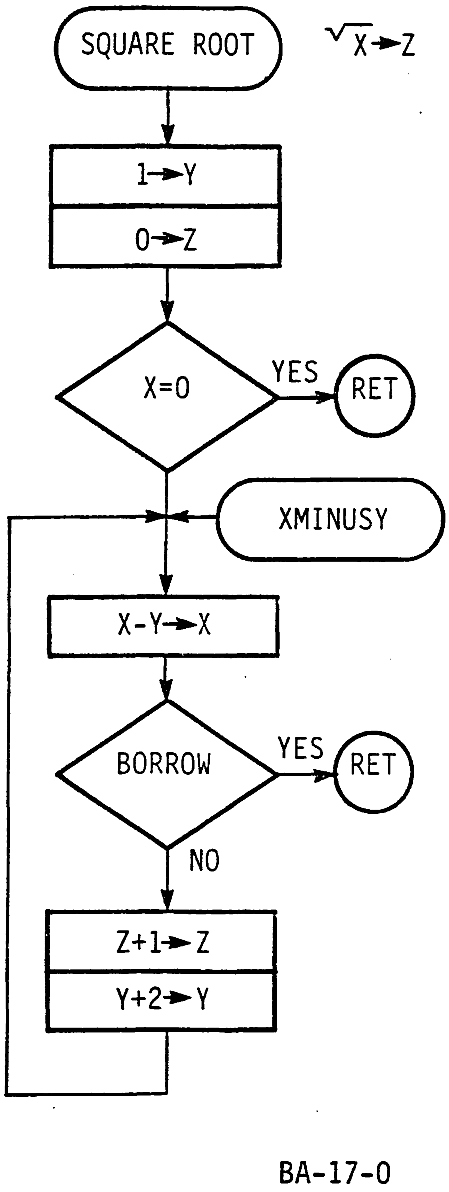


Figure 5-9. Integer Square Root

```
SQROOT:     LBI     1,12        ; -> Z
            STII    1           ; 1 -> Y, Bd +1 -> Bd
            JSRP    CLEAR
            LBI     2,12        ; 0 -> Z
            JSRP    CLEAR
            LBI     0,12        ; test X = 0, if so exit
TSTZRO:     CLRA
            SKE
            JP      XMINUSY     ; X = 0
            XIS
            JP      TSTZRO
            RET
XMINUSY:    JSRP    SUB         ; X-Y -> X
            SKC                 ; test borrow, C = 0 if borrow
            RET                 ; if borrow, exit-finished
            JSRP    ZPLUS1      ; Z+1 -> Z
            JSRP    YPLUS1      ; Y+2 -> Y
            JSRP    YPLUS1      ;
            JP      XMINUSY
```

The following subroutines, assumed to be in Page 2, are used by the square root routine above:

```
CLEAR:  CLRA            ; simple register clear
        XIS
        JP      CLEAR
        RET
SUB:    LBI     1,12
        LD      1       ; this is the basic BCD subtract routine as given
                        ; earlier
SUB1:   CASC
        ADT
        XIS     1
        JP      SUB1
        RET

ZPLUS:  LBI     2,12    ; this is the basic BCD increment routine as
                        ; given earlier, with the repeated LBI skip
                        ; feature
YPLUS:  LBI     1,12
        SC
PLUS1:  CLRA
        AISC    6
        ASC
        ADT
        XIS
        JP      PLUS1
        RET
```

Floating-Point Square Root - BCD

A full floating-point BCD square root routine is provided. As written, the routine works on a 12-digit number with a two-digit signed exponent. Although substantially more complex than the integer square root routine seen earlier, this routine has the same conceptual basis – the subtraction of odd numbers.

The first part of the routine creates the exponent of the result of dividing the original exponent by two. Note that this is accomplished by first multiplying the exponent by 5, via repeated additions, and then dividing it by 10 by means of a right-digit shift.

Two flow charts are provided, a generalized flow chart (Figure 5-10) and a detailed flow chart (Figure 5-10a), to help clarify the routine. The RAM map for the routine is indicated below.

Br \ Bd	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	X EXPONENT		X SIGN	GUARD	MSD				X MANTISSA							LSD
1	Y EXPONENT		Y SIGN						Y MANTISSA							
2	TEMP STORE	DIGIT COUNTER	NOT USED													

The routine performs $\sqrt{x} \rightarrow x$.

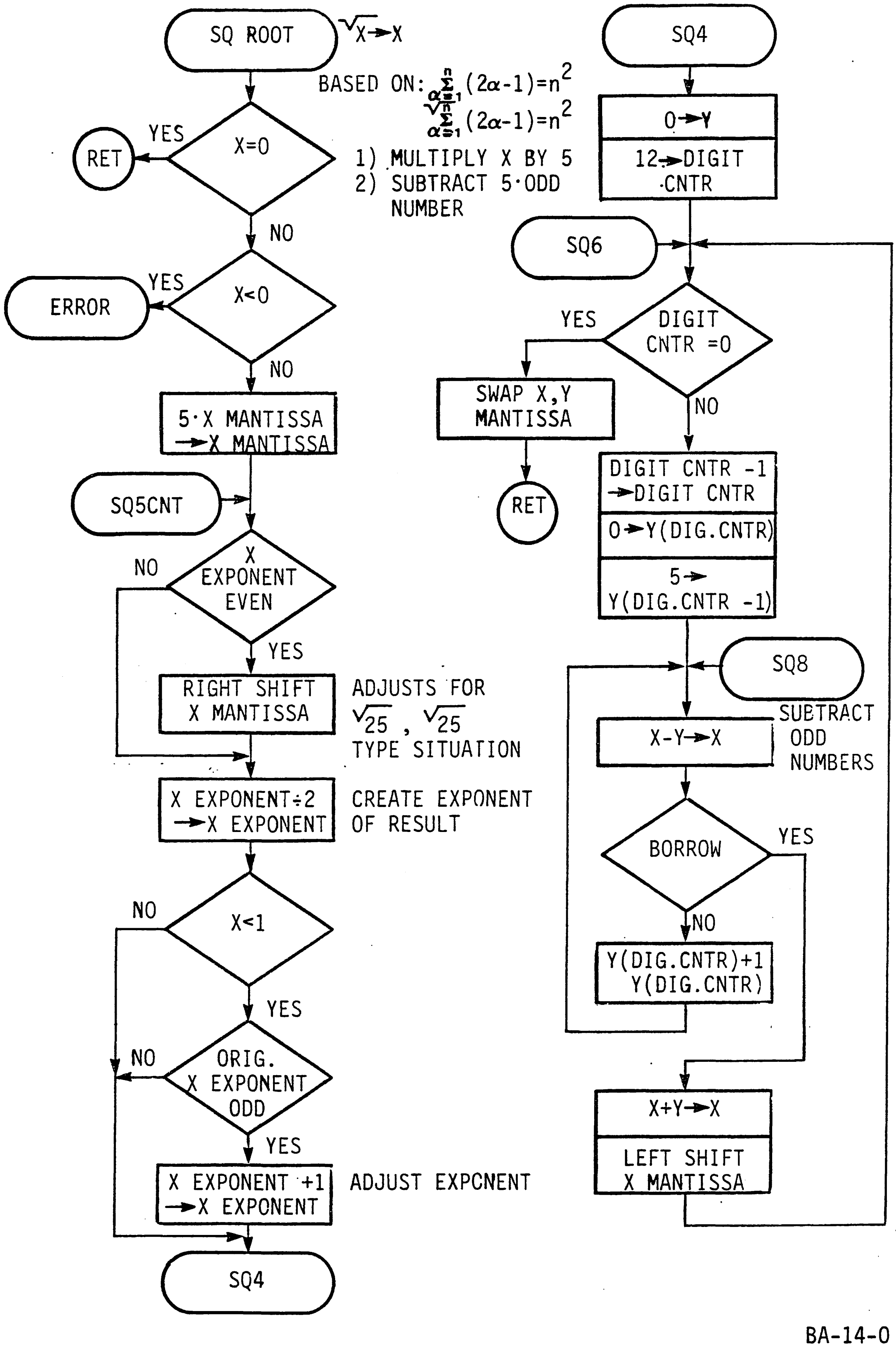


Figure 5-10. Square Root - General Flow Chart

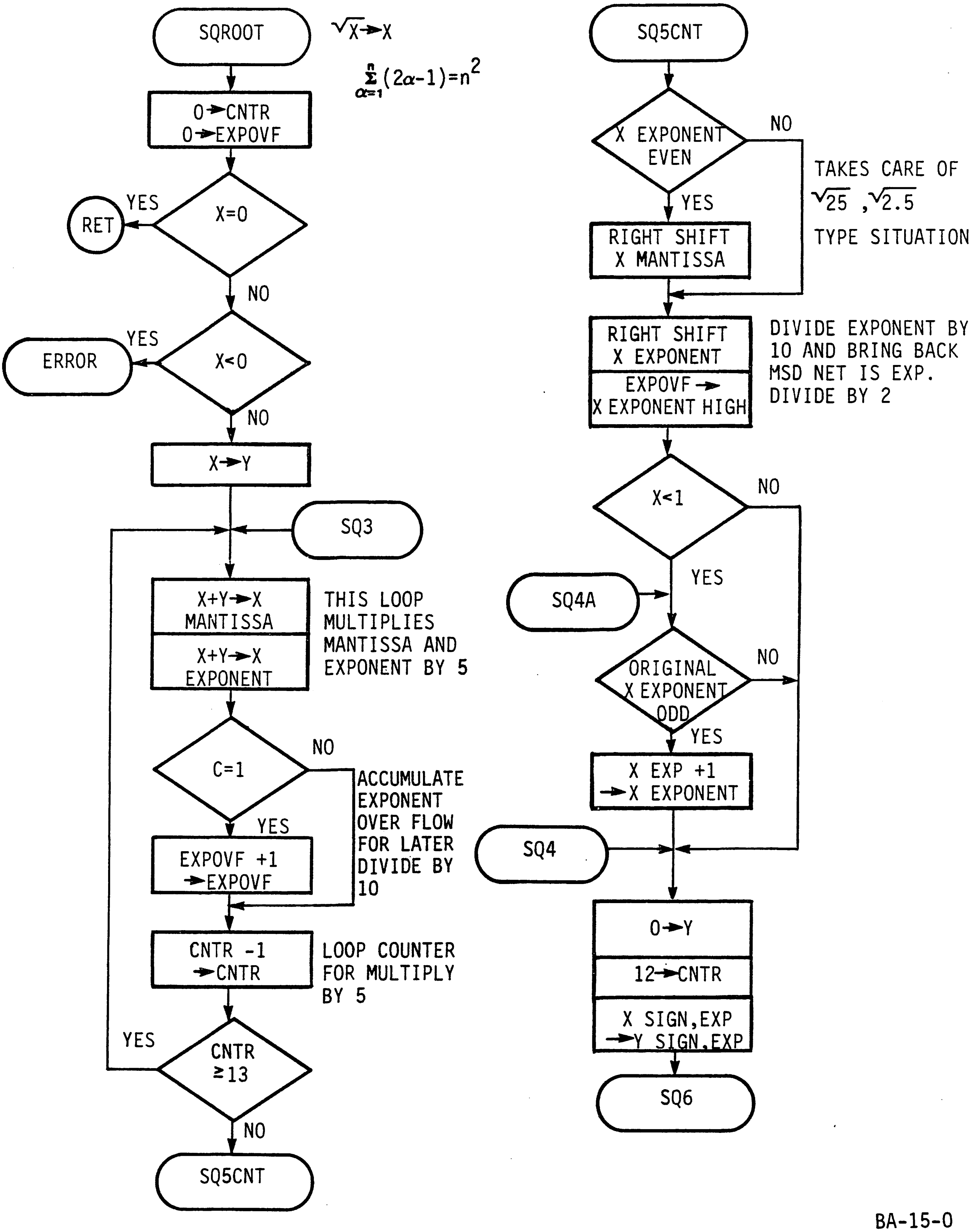


Figure 5-10a. Square Root - Detailed Flow Chart (Sheet 1 of 2)

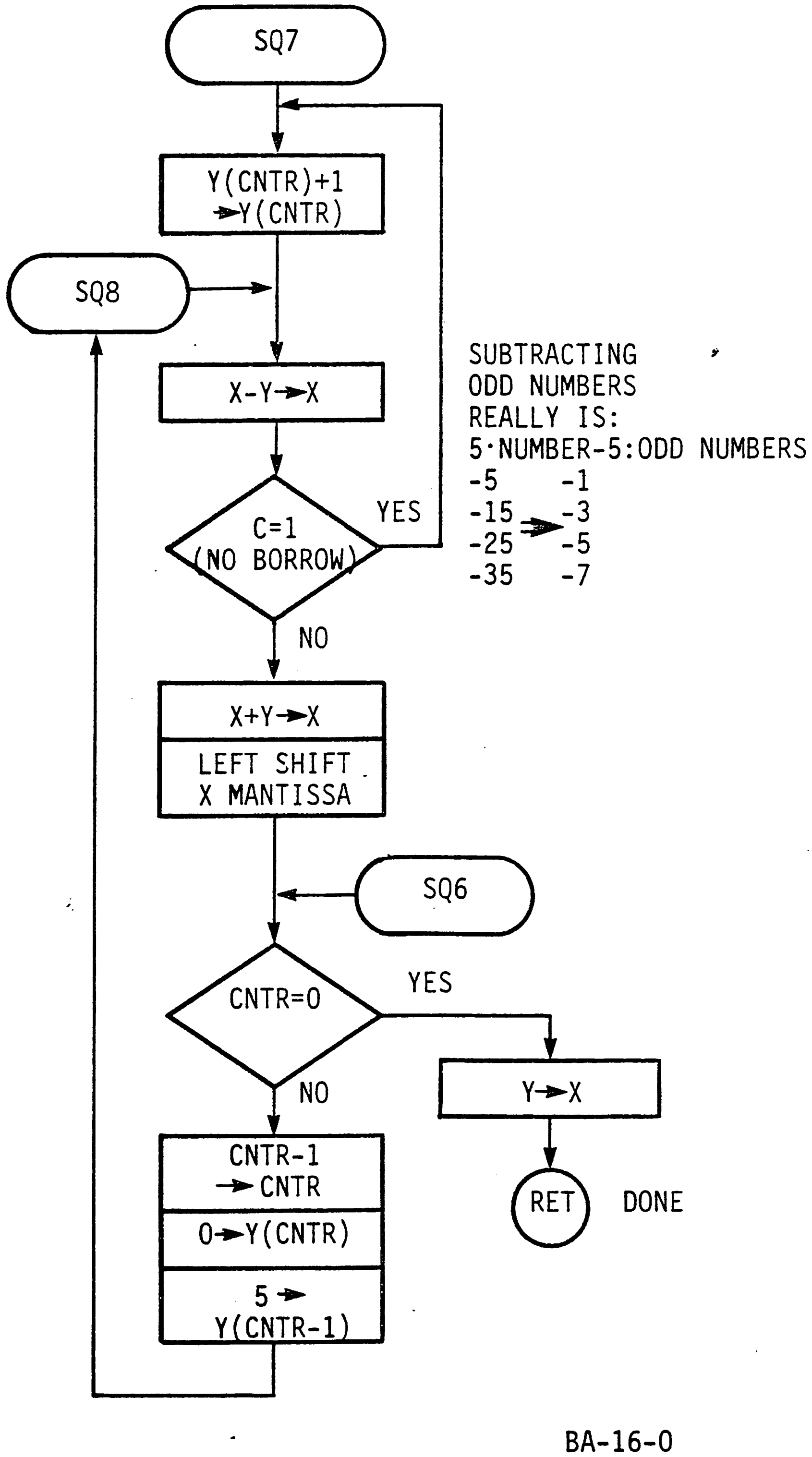


Figure 5-10a. Square Root - Detailed Flow Chart (Sheet 2 of 2)

```
COP CROSS ASSEMBLER     PAGE:     1
SQROOT  SQUARE ROOT ROUTINE


              ;THIS PROGRAM IS A FLOATING POINT SQUARE ROOT ROUTINE. THE
              ;ROUTINE ASSUMES THAT THE NUMBER X(REGISTER 0) IS IN
              ;SCIENTIFIC NOTATION FORMAT, I.E., SIGNED EXPONENT AND
              ;MANTISSA.  AS WRITTEN THE ROUTINE ASSUMES A 12 DIGIT BCD
              ;MANTISSA AND GENERATES A 12 DIGIT BCD RESULT.  THE
              ;EXPONENT IS APPROPRIATELY HANDLED.  BY CHANGING ONLY THE
              ;START VALUE IN THE DIGIT COUNTER, SMALLER MANTISSAS CAN
              ;BE EASILY HANDLED. THE STRUCTURE OF THE ROUTINE DOES NOT
              ;CHANGE.
              ;
              ;THE ROUTINE ASSUMES THAT THE LSD OF THE NUMBER IS LOCATION
              ;0. THE MSD OF THE NUMBER IS IN LOCATION 11.  LOCATION 13
              ;CONTAINS THE SIGN INFORMATION FOR BOTH THE EXPONENT AND THE
              ;MANTISSA.  BIT 0 OF LOCATION 13 IS THE EXPONENT SIGN; BIT
              ;3 IS THE MANTISSA SIGN; BITS 1 AND 2 ARE NOT USED.  A TWO
              ;DIGIT EXPONENT IS CONTAINED IN LOCATIONS 14 AND 15 WITH
              ;LOCATION 15 BEING THE MOST SIGNIFICANT DIGIT OF THE EXPONENT.
              ;LOCATION 12 IS THE MANTISSA GUARD DIGIT AND IS USED IN THE
              ;COMPUTATION BUT CONTAINS NO INFORMATION ON ENTRY TO
              ;OR EXIT FROM THE ROUTINE.
              ;
              ;THE ROUTINE FURTHER ASSUMES THAT THE DECIMAL POINT IS LOCATED
              ;TO THE RIGHT OF THE MSD OF THE MANTISSA,I.E.,ALL NUMBERS ARE
              ;OF THE FORM 1.2345 x 10**EXPONENT.
              ;
       0000   XLSD    =       0,0
       000B   XMSD    =       0,11
       000C   XGUARD  =       0,12
       000D   XSIGN   =       0,13
       000E   XEXPLO  =       0,14
       000F   XEXPHI  =       0,15
       0010   YLSD    =       1,0
       001C   YGUARD  =       1,12
       001E   YEXPLO  =       1,14
       001F   YEXPHI  =       1,15
       002E   CNTR    =       2,14     ;THIS IS MANTISSA DIGIT COUNTER
       002F   EXPOVF  =       2,15     ;EXPONENT OVERFLOW DIGIT
              ;
                      .TITLE  SQROOT,'SQUARE ROOT ROUTINE'
              ;

       0000           .PAGE   0
              ;**************************************************************
              ;CODE FROM HERE TO NEXT LINE OF *'S IS NOT PART OF SQUARE ROOT
              ;ROUTINE.  IT IS FOR TEST ONLY.
              ;
   000 00     START:  CLRA
   001 12     STRT1:  XABR
   002 00     STRT2:  CLRA
   003 04             XIS              ;CLEAR ALL THE RAM FOR CONTROL
   004 C2             JP      STRT2
```

Figure 5-11. Square Root Routine (Sheet 1 of 6)

```
COP CROSS ASSEMBLER    PAGE:    2
SQROOT  SQUARE ROOT ROUTINE

 005 12               XABR
 006 5D               AISC     13
 007 C1               JP       STRT1
              TESTSQROOT:
 008 44               NOP
 009 44               NOP
 00A 335F             OGI      15       ;THIS JUST FOR TEST,DETECTING ERROR
 00C 6900             JSR      SQROOT   ;RETURN AND SKIP IF ERROR,SO G WILL BE
 00E 3350             OGI      0        ;SET TO 0 IF NO ERROR
 010 44               NOP
 011 44               NOP
 012 C8               JP       TESTSQROOT
              ;
              ;NOTE THAT THE PRECEEDING CODE IS NOT PART OF THE SQUARE ROOT
              ;ROUTINE.  IT IS FOR CONTROL AND TEST ONLY
              ;*************************************************************
              ;
              ;THE FOLLOWING CODE IS PART OF SQUARE ROOT ROUTINE.THE SUB-
              ;ROUTINES ARE INCLUDED. IN A SYSTEM REQUIRING SQUARE ROOT, IT
              ;IS HIGHLY PROBABLE THAT AT LEAST SOME OF THE OTHER BASIC
              ;ARITHMETIC FUNCTIONS WOULD ALSO BE REQUIRED. THE SUBROUTINES
              ;BELOW WOULD ALSO BE USABLE IN THOSE ROUTINES.
              ;
              ;*************************************************************
              ;

     0080             .PAGE    2
 080 00       CLEAR:  CLRA              ;CLEAR A REGISTER
 081 04               XIS
 082 80               JP       CLEAR
 083 48               RET
 084 1F       ADDXY:  LBI      YLSD     ;X + Y --> X,13 DIGITS(MANTISSA AND
 085 32       ADDXYE: RC                ;GUARD DIGIT)
 086 15       ADLOOP: LD       1
 087 56               AISC     6        ;DECIMAL ADJUST
 088 30               ASC
 089 4A               ADT               ;DECMAL CORRECT
 08A 14               XIS      1
 08B 8D               JP       ADLP2
 08C 48               RET
 08D 4E       ADLP2:  CBA               ;NOW TEST IF DONE
 08E 53               AISC     3        ;IF BD >= 13,DONE
 08F 86               JP       ADLOOP
 090 48               RET
              ;
 091 05       PLUS1:  LD                ;MEMORY LOCATION PLUS 1
 092 51               AISC     1
 093 44               NOP
 094 06       XRET:   X
 095 48               RET
 096 2D       CTRMN1: LBI      CNTR     ;DIGIT COUNTER MINUS 1
 097 05       MINUS1: LD                ;MEMORY LOCATION MINUS 1
 098 5F               AISC     15
```

Figure 5-11. Square Root Routine (Sheet 2 of 6)

```
099 94              JP      XRET
09A 94              JP      XRET
                ;
09B 22      SUBXY:  SC
09C 1F              LBI     YLSD
09D 15      SUBLOOP: LD     1         ;X - Y --> X,13 DIGITS (MANTISSA AND
09E 10              CASC              ;GUARD DIGIT)
09F 4A              ADT
0A0 14              XIS     1
0A1 4E              CBA
0A2 53              AISC    3
0A3 9D              JP      SUBLOOP
0A4 48              RET
                ;
0A5 0B      RSHX:   LBI     XGUARD    ;RIGHT SHIFT X MANTISSA
0A6 00              CLRA
0A7 07      RSHX1:  XDS
0A8 A7              JP      RSHX1
0A9 48              RET
                ;
0AA 0F      LSHX:   LBI     XLSD      ;LEFT SHIFT X
0AB 00              CLRA
0AC 04      LSHX1:  XIS
0AD AC              JP      LSHX1
0AE 48              RET
                ;
0AF 15      XFER1:  LD      1         ;REGISTER TRANSFER
0B0 14              XIS     1
0B1 AF              JP      XFER1
0B2 48              RET
                ;

    0100            .PAGE   4
                ;
                ;*****************************************************************
                ;
                ;THE FOLLOWING IS THE BODY OF THE SQUARE ROOT ROUTINE
                ;X MUST BE NORMALIZED ON ENTRY, I.E.,NO LEADING ZEROES
                ;THUS IF XMSD IS 0,THE MANTISSA IS 0
                ;
100 2D      SQROOT: LBI     CNTR
101 80              JSRP    CLEAR     ;CLEAR DIGIT COUNTER AND EXPONENT
                                      ;OVERFLOW DIGIT--CNTR & EXPOVF
102 0A      SQ1:    LBI     XMSD      ;TEST FOR X = 0,IF YES,RETURN
103 05              LD                ;IF XMSD 0,X IS 0
104 5F              AISC    15
105 48              RET
106 0C      SQ2:    LBI     XSIGN     ;ERROR IF X IS NEGATIVE
107 13              SKMBZ   3
108 49      ERROR:  RETSK             ;RETURN AND SKIP FOR ERROR
109 0F              LBI     XLSD
10A AF              JSRP    XFER1     ;X --> Y FOR SUBSEQUENT ADDS
                ;
```

Figure 5-11. Square Root Routine (Sheet 3 of 6)

```
10B 84      SQ3:    JSRP    ADDXY   ;THIS LOOP MULTIPLIES X MANTISSA BY
10C 1D              LBI     YEXPLO  ;5 AND THE X EXPONENT BY 5
10D 85              JSRP    ADDXYE
10E 86              JSRP    ADLOOP
10F 20              SKC
110 D3              JP      TSTCTR
111 2E              LBI     EXPOVF  ;EXTRA DIGIT FOR THE EXPONENT MULTIPLY
112 91              JSRP    PLUS1
113 96      TSTCTR: JSRP    CTRMN1  ;CNTR IS USED AS LOOP COUNTER HERE
114 05              LD
115 53              AISC    3       ;IF CNTR IS < 13,MULTIPLY IS COMPLETE
116 D8              JP      SQ5CNT
117 CB              JP      SQ3

            ;
118 0D      SQ5CNT: LBI     XEXPLO  ;TEST X EXPONENT EVEN(IF ORIGINAL X
119 05              LD              ;EXPONENT EVEN,5 TIMES IT WILL RESULT
11A 5F              AISC    15      ;IN XEXPLO BEING 0)
11B A5              JSRP    RSHX    ;FOR SQRT 25,SQRT 2.5 TYPE CASE

            ;
11C 2E              LBI     EXPOVF  ;RIGHT SHIFT X EXP WITH O'FLOW DIGIT
11D 25              LD      2       ;RESULTS IN NET EXPONENT DIVIDE BY 2
11E 07              XDS             ;WHICH IS DESIRED RESULT FOR SQUARE
11F 07              XDS             ;ROOT
120 01              SKMBZ   0       ;SEE IF X < 1
121 E9              JP      SQ4A    ;YES
122 1F      SQ4:    LBI     YLSD    ;CLEAR Y,WILL CREATE ANSWER IN Y
123 80              JSRP    CLEAR
124 0C              LBI     XSIGN   ;MOVE SIGN,EXPONENT TO Y
125 AF              JSRP    XFER1
126 2D              LBI     CNTR    ;LOAD DIGIT COUNTER WITH NUMBER
127 7C              STII    12      ;OF DIGITS
128 FC              JP      SQ6
129 0D      SQ4A:   LBI     XEXPLO  ;TEST ORIGINAL EXPONENT ODD,5 TIMES IT
12A 5B              AISC    11      ;RESULTS IN A, AT THIS POINT,=5 IF
12B E2              JP      SQ4     ;ORIGINAL X EXPONENT ODD
12C 91              JSRP    PLUS1   ;ORIGINAL EXPONENT WAS ODD & X<1,
12D 57              AISC    7       ;CORRECT EXPONENT BY ADDING 1
12E E2              JP      SQ4     ;IF LSD EXPONENT WAS < 9,STOP
12F 70              STII    0       ;WAS = 9,SO SET TO 0 AND INCREMENT
130 91              JSRP    PLUS1   ;MSD OF EXPONENT
131 E2              JP      SQ4

            ;
            ;EXPONENT COMPUTATION COMPLETE AT THIS POINT.THE REST OF THE
            ;CODE COMPUTES THE MANTISSA OF THE RESULT BY THE TECHNIQUE
            ;OF SUBTRACTION OF ODD NUMBERS. SINCE THE MANTISSA HAS BEEN
            ;MULTIPLIED BY 5, 5 TIMES THE VARIOUS ODD NUMBERS WILL BE SUB-
            ;TRACTED. THUS TO SUBTRACT 1,3,5,7,... FROM THE ORIGINAL WE
            ;SUBTRACT 5,15,25,35,...FROM 5 TIMES THE ORIGINAL VALUE.
            ;
132 2D      SQ7:    LBI     CNTR
133 35              LD      3       ;INCREMENT Y(CNTR)
134 50              CAB
135 91              JSRP    PLUS1
136 9B      SQ8:    JSRP    SUBXY   ;THIS IS THE REPEATED SUBTRACT
```

Figure 5-11. Square Root Routine (Sheet 4 of 6)

```
COP CROSS ASSEMBLER    PAGE:    5
SQROOT  SQUARE ROOT ROUTINE

137 20                  SKC               ;IF WE BORROW,NEED TO SHIFT
138 FA                  JP      SQ8A
139 F2                  JP      SQ7
13A 84        SQ8A:     JSRP    ADDXY     ;RESTORE VALUE
13B AA                  JSRP    LSHX
13C 96        SQ6:      JSRP    CTRMN1    ;DECREMENT DIGIT COUNTER
13D 05                  LD
13E 51                  AISC    1         ;SEE IF IT WAS 0 FOR EXIT
13F C3                  JP      SQ6A

140 1F        DONE:     LBI     YLSD      ;DONE,TRANSFER RESULT IN Y TO X
141 AF                  JSRP    XFER1
142 48                  RET
143 35        SQ6A:     LD      3         ;0 --> Y(CNTR)
144 50                  CAB
145 00                  CLRA
146 07                  XDS
147 75                  STII    5         ;5 --> Y(CNTR - 1)
148 6136                JMP     SQ8
              ;
                        .END
```

Figure 5-11. Square Root Routine (Sheet 5 of 6)

```
ADDXY   0084      ADDXYE 0085      ADLOOP 0086      ADLP2   008D
CLEAR   0080      CNTR    002E      CTRMN1 0096      DONE    0140 *
ERROR   0108 *    EXPOVF 002F      LSHX    00AA      LSHX1   00AC
MINUS1 0097 *    PLUS1   0091      RSHX    00A5      RSHX1   00A7
SQ1     0102 *    SQ2     0106 *   SQ3     010B      SQ4     0122
SQ4A    0129      SQ5CNT 0118      SQ6     013C      SQ6A    0143
SQ7     0132      SQ8     0136      SQ8A    013A      SQROOT 0100
START   0000 *    STRT1   0001      STRT2   0002      SUBLOO 009D
SUBXY   009B      TESTSQ 0008      TSTCTR 0113      XEXPHI 000F *
XEXPLO 000E      XFER1   00AF      XGUARD 000C      XLSD    0000
XMSD    000B      XRET    0094      XSIGN   000D      YEXPHI 001F *
YEXPLO 001E      YGUARD 001C *    YLSD    0010

NO ERROR LINES

  144  ROM WORDS USED

COP   420  ASSEMBLY

SOURCE CHECKSUM = 46ED

INPUT FILE    ABDUL10:SQROOT.SRC  VN:     9
```

Figure 5-11. Square Root Routine (Sheet 6 of 6)

5.2.10 Binary to BCD Conversion

Several methods of performing a binary to BCD conversion are illustrated. These different approaches illustrate different algorithms and different programming techniques.

A Simple 8-bit Binary to BCD Routine

This is a simple routine for converting an eight-bit binary number to its three-digit BCD equivalent. The conversion is the straight-forward scheme of adding the respective powers of two. However, this is reduced if the eight-bit number is treated as a two-digit hex number: then we merely expand the number by the powers of 16. Thus we have:

$1110\ 0111_2 = E7_{16} = 14_{10} * 16^1_{10} + 7_{10} * 16^0_{10} = 331_{10}$
$0101\ 1111_2 = 5F_{16} = 5_{10} * 16^1_{10} + 15_{10} * 16^1_{10} = 95_{10}$

The flow chart for this routine is given in Figure 5-12. The RAM map is given below.

Br \ Bd	15	14	13
2		Work Space - Z	
1	MSD	BCD Result	
0	Binary Number		Not Used

The routine converts the binary number in R0 to the BCD number in R1. R2 is used as work space. The binary value is destroyed. Four implementations of this routine are presented.

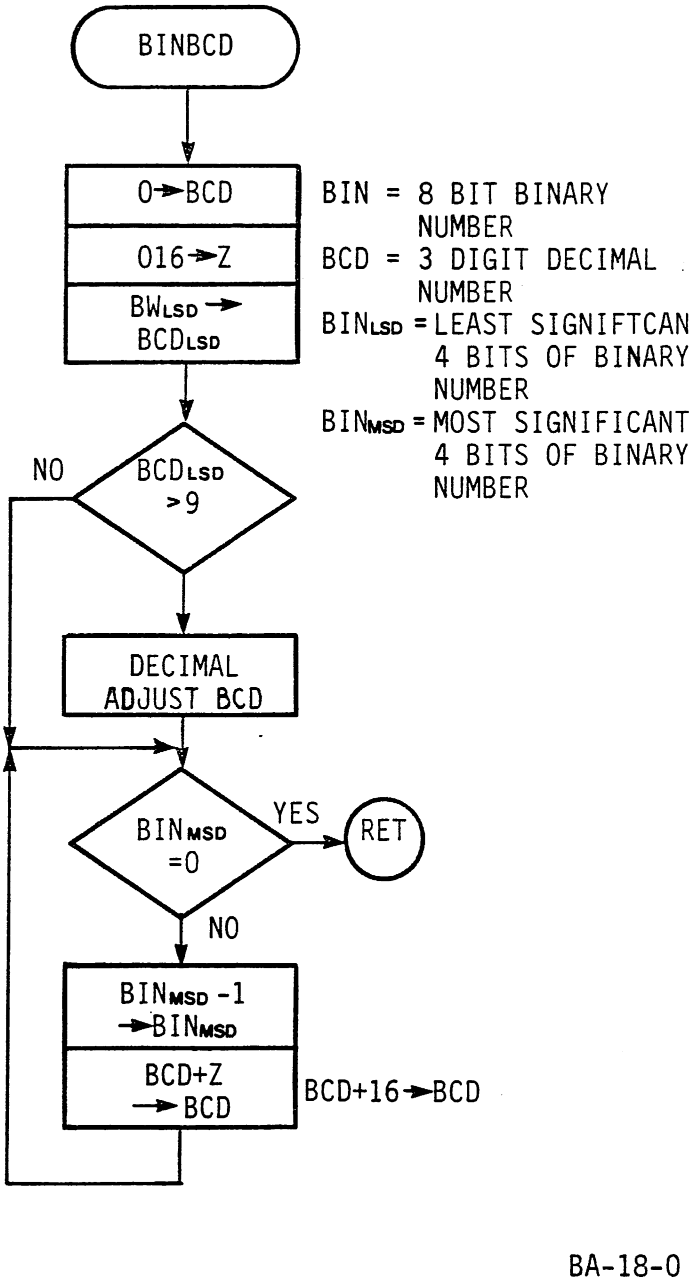


Figure 5-12. Eight-Bit Binary to BCD Conversion

Version I

```
RINBCD: LBI   0,14
        XDS   1       ; BIN_LSD -> BCD_LSD, 0 to other digits in BCD
        XIS
        STII  0
        STII  0
        LBI   2,13    ; 016 -> Z
        STII  6
        STII  1
        STII  0
        LBI   0,13    ; test BCD_LSD > 9, if so, decimal adjust
        LD
        AISC  6
        JP    TEST
        XIS
        STII  1
TEST:   LBI   0,15    ; test BIN_MSD = 0, if yes exit
        LD            ; conversion complete
        AISC  15
        RET           ; decrement BIN_MSD by 1
        X
        LBI   2,13
        RC
LOOP:   LD    3       ; add BCD+16 (BCD+Z) -> BCD
        AISC  6
        ASC
        ADT
        XIS   3
        JP    LOOP
        JP    TEST
```

Version II

```
RIBCD:  LBI    0,14
        XDS    1        ; BINLSD -> BCDLSD, 0 to other digits in BCD
        XIS
        STII   0
        STII   0
        LBI    2,13     ; equivalent of 016 -> Z
        STII   12       ; have incorporated AISC 6 into constant for subsequent BCD addition
        STII   7
        STII   6
        LBI    0,13     ; test BCDLSD -> 9, if so decimal adjust
        LD
        AISC   6
        JP     TEST
        XIS
        STII   1
TEST:   LBI    0,15     ; test BINMSD = 0, if yes exit - conversion complete
        LD
        AISC   15
        RET
        X               ; decrement BINMSD by 1
        LBI    2,13
        RC
LOOP:   LD     3        ; BCD+16 -> BCD
        ASC
        ADT
        XIS    3
        JP     LOOP
        JP     TEST
```

Version III

```
BINBCD: LBI    0,14   ; BIN_LSD --> BCD_LSD, 0 to other digits in BCD
        XDS    1
        XIS
        STII   0
        STII   0
        LBI    0,13   ; test BCD_LSD > 9, if so decimal adjust the number
        LD
        AISC   6
        JP     TEST
        XIS
        STII   1
TEST:   LBI    0,15   ; test BIN_MSD = 0, if yes exit
        LD            ; conversion complete
        AISC   15     ; else decrement BIN_MSD by 1
        RET
        X
        LBI    2,13   ; straight line BCD+16 -> BCD using no additional RAM
        RC
        CLRA
        AISC   12
        ASC
        ADT
        XIS
        CLRA
        AISC   7
        ASC
        ADT
        XIS
        CLRA
        AISC   6
        ASC
        ADT
        X
        JP     TEST   ; loop back to TEST
```

Version IV

```
BINBCD:   LBI    0,14
          XDS    1          ; BIN_LSD --> BCD_LSD, 0 to other digits in BCD
          XIS
          STII   0
          STII   0
          LBI    2,13
          STII   0          ; clear Z
          STII   0
          STII   0
LOOP:     JSRP   BCDADD     ; decimal adjust first time, add 16 in all subsequent times
          LBI    2,13
          STII   6
          STII   1
TEST:     LBI    0,15
          LD
          AISC   15
          RET
          X
          JP     LOOP
```

The routine uses the following subroutine, assumed to be located in Page 2.

```
BCDADD:   LBI    2,13
          RC
ADLOOP:   LD     3
          AISC   6
          ASC
          ADT
          XIS    3
          JP     ADLOOP
          RET
```

NOTE: By using the same kind of "trick" as was illustrated in Version II, the total ROM count can be reduced by one word and the execution speed improved.

Let us now consider these four programs. They all do precisely the same thing: convert an eight-bit binary number to a three-digit BCD number using the same algorithm. The differences are in implementation only. Version 1 takes 29 ROM words, uses 8 RAM digits (two for input binary number, three for BCD result, and three for scratch pad), has a worst case execution time of 409 instruction cycles, and uses no subroutines. Version II takes 28 ROM words, also uses 8 RAM digits, has a worst case execution time of 364 instruction cycles, and also uses no subroutines. Version III takes 34 ROM words, uses only 5 RAM digits, has a worst case execution time of 360 instruction cycles, and uses no subroutines. Version IV uses 28 ROM words, including 9 words in a subroutine; uses 8 RAM digits; and has a worst case execution time of 474 instruction cycles. Other variations on these routines are possible which will affect ROM, RAM, and execution time.

Version I is the straight-forward implementation of the flow chart with few tricks. It is fairly representative of the amount of code required for the task; uses the maximum RAM for the function and is about midrange in execution speed. Version II makes a very slight change to Version I. It sets up a constant with a decimal adjust factor built in. The result is that Version II uses one less ROM word and the same amount of RAM as Version 1; however, Version II executes considerably faster, about a 10 per cent speed improvement. Version III uses the minimum RAM for the function, uses the most ROM, and has the fastest execution time. This has been achieved by straight line coding the BCD add 16. This both maximizes speed and reduces RAM usage but the penalty is ROM code. RAM usage is reduced by storing the constant "16" in ROM rather than RAM. Version IV is preferable in cases where a BCD addition subroutine already exists in the program. Not counting the subroutine Version IV uses only 19 ROM words. However, Version IV has the slowest execution time. By the addition of two ROM words, as shown in Version IVA, the speed of Version IV can be significantly improved. Version IVA is the same as Version IV but achieves faster speed, by moving some code out of the main loop, with a small ROM penalty.

Version IVA

```
BINBCD:  LBI   0,14     ; BIN_LSD -> BCD_LSD, 0 to other
         XDS   1        ; digits in BCD
         XIS
         STII  0
         STII  0
         LBI   2,13     ; clear Z
         STII  0
         STII  0
         STII  0
         JSRP  BCDADD   ; decimal adjust BCD
         LBI   2,13     ; 16 -> Z
         STII  6
         STII  1
         JP    TEST
LOOP:    JSRP  BCDADD   ; basic loop
TEST:    LBI   0,15
         LD
         AISC  15
         RET
         X
         JP    LOOP
```

Version IVA uses 21 ROM words (not counting the subroutine), uses 8 RAM digits, and executes in 454 instruction cycle times.

Binary to BCD Conversion - Doubling Methods

If we have a binary number expressed as $b_n b_{n-1} ... b_2 b_1 b_0$ where bx is either 1 or 0, the standard expansion to produce the decimal (BCD) number is as follows:

$$b_n*2^n + b_{n-1}*2^{n-1} + ... + b_2*2^2 + b_1*2 + b_0 = \sum_{i=0}^{n} b_i*2^i$$

For simplicity, a six-bit binary number is used.

$b_5 b_4 b_3 b_2 b_1 b_0$

The expansion for this number for its decimal equivalent, is then

$b_5*2^5 + b_4* 2^4 + b_3*2^3 + b_2*2^2 + b_1*2 + b_0$

This can be rewritten as

$$2\left\{2\,[\,2 < 2\,(\,2\,(\,b_5\,(+\,b_4\,)+b_3 > +\,b_2\,]+b_1\right\}+b_0$$

This expression, although apparently more complex, points out one means of conversion that is easy to implement because it is iterative. The first step is to set the BCD number equal to the most significant bit, here it is b_5. Then the value is doubled and one is added if the next bit is one. The value is then doubled again and one is added if the next bit is one. The cycle continues until the LSB is added to the result. Figure 5-13 is the flow chart for this general approach.

The Straight-Forward Implementation

This implementation is the straight-forward implementation of the flow chart of Figure 5-13. As written, it converts a 16-bit binary number to its five-digit BCD counterpart. The routine expands by merely changing the pertinent LBI instructions. Figure 5-14 is the RAM map for this routine. The routine uses one subroutine level.

The routine uses the following subroutines assumed to be located in Page 2.

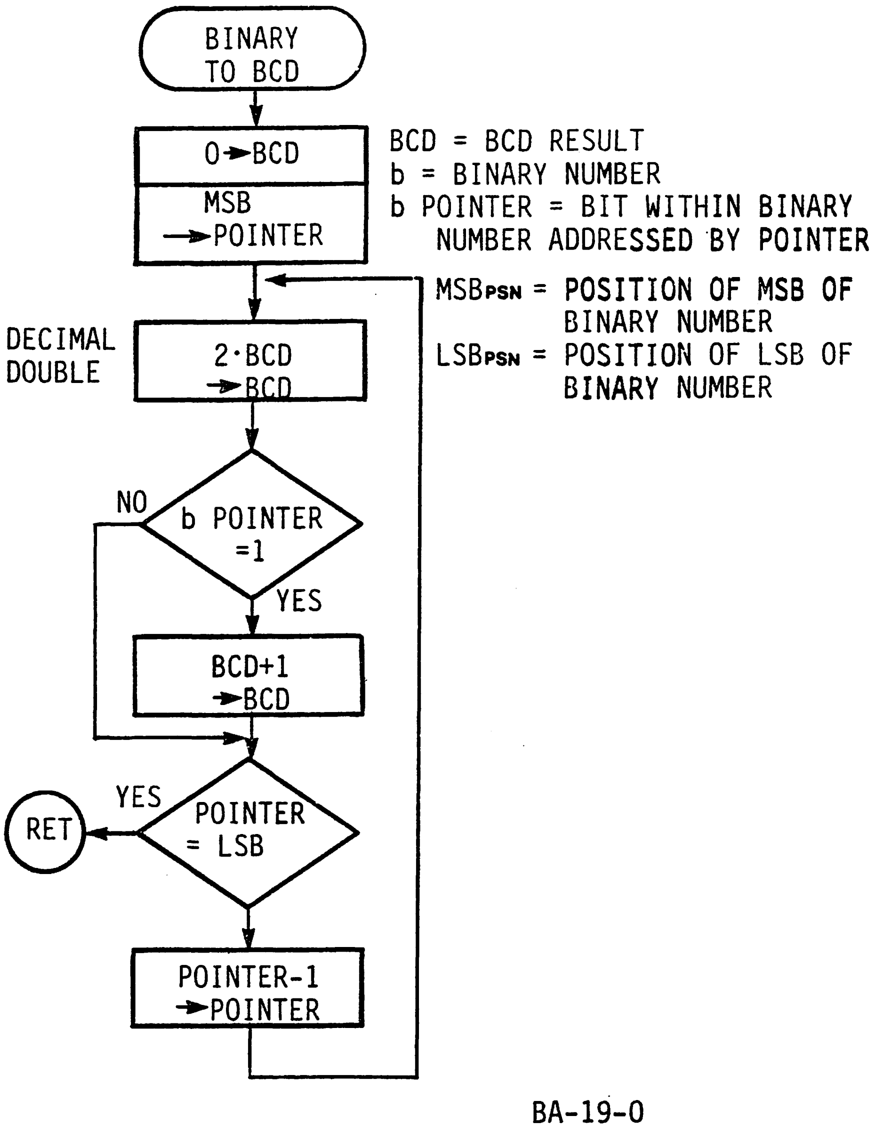


Figure 5-13. Binary to BCD Conversion – Basic Doubling Algorithm

Br \ Bd	15	14	13	12	11	10
1	MSB	BINARY NUMBER			NOT USED	
2	MSD	BCD NUMBER			LSD	TEMP STORE

Figure 5-14. RAM Map for Doubling Algorithm Straight-Forward Implementation

```
DOUBLE:  LBI     2,11      ; decimal double of BCD number
DBLA:    LD                ; modify the LBI for larger numbers
         AISC    6
         ASC
         ADT
         XIS               ; sips on Bd increment past 15
         JP      DBLA
SETUP:   RC                ; falls into SETUP routine-which
         LBI     2,10      ; is also called independently
         LD      3         ; fetch digit position of binary number
         CAB               ; and load Bd with it. Br adjusted by
         RET               ; LD 3 instruction
```

The basic routine is as follows:

```
BINBCD:  LBI     2,10      ; load pointer with position of binary MSB
         STII    15
CLEAR:   CLRA              ; 0 -> BCD
         XIS
         JP      CLEAR
BINDEC:  JSRP    SETUP     ; point to proper digit in binary number
         SKMBZ   3         ;
         SC                ; march down the digit; set C
         JSRP    DOUBLE    ; if addressed bit is 1. Then do
         SKMBZ   2         ; double to get 2* BCD + bx -> BCD
         SC                ; (C is reset on exit from SETUP and DOUBLE)
         JSRP    DOUBLE
         SKMBZ   1
         SC
         JSRP    DOUBLE
         SKMBZ   0
         SC
         JSRP    DOUBLE
         LBI     2,10      ; test for done, here finished
         LD
         AISC    3         ; if Bd ≤ 12
         RET
         AISC    12        ; else, decrement Bd by 1 and
         X
         JP      BINDEC    ; continue
```

This routine uses 25 words plus 12 words in the subroutine page for a total of 37 words. The routine executes in 690 instruction cycles. The execution time is data independent. The routine preserves the binary number. The routine uses 10 RAM digits; 4 for the original number, 5 for the BCD result, and 1 scratch pad.

Variation I - The Doubling Algorithm - "Shift 1"

The straight-forward implementation can be modified in a simple way by using some left bit shifting on the binary number. The basic flow chart is the same but a detailed modified flow chart is shown in Figure 5-15. Figure 5-16 is the RAM map for this variation.

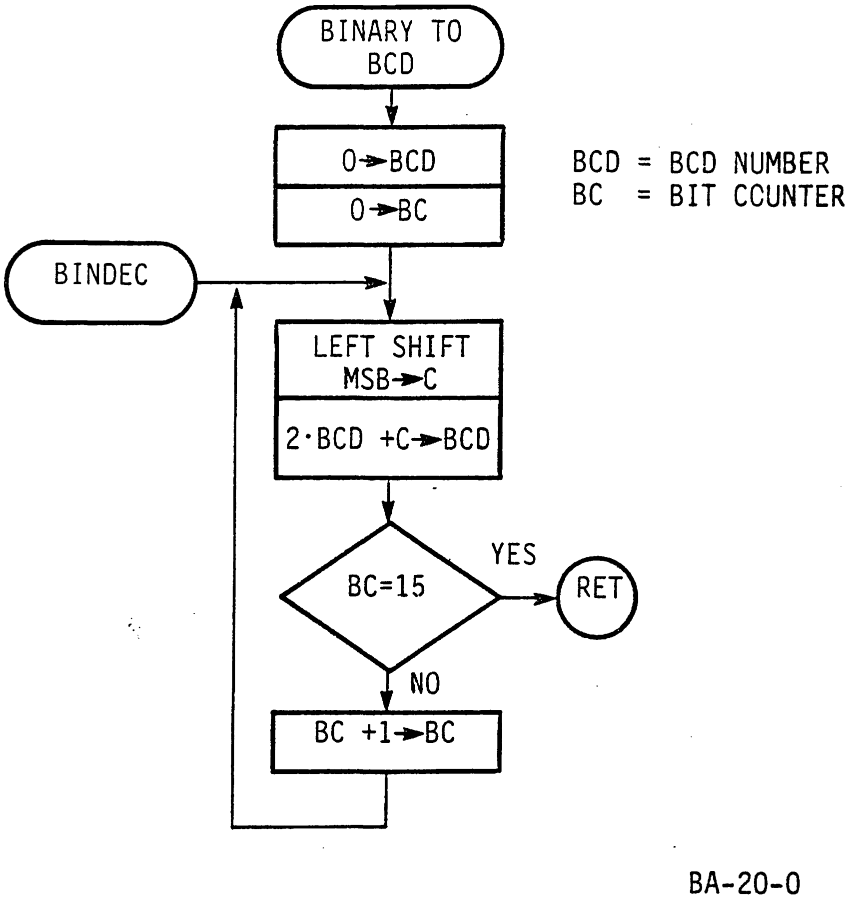


Figure 5-15. Flow Chart for Variation 1

Bd

Br	15	14	13	12	11	10
1	MSB	BINARY NUMBERS		LSB	NOT USED	
2	MSD	BCD NUMBER			LSD	BIT COUNTER

Figure 5-16. RAM Map for Variation 1 on the Doubling Algorithm

The code for this implementation is as follows:

```
BINBCD:  LBI    2,10
CLEAR:   CLRA              ; 0 -> bit counter, BCD number
         XIS
         JP     CLEAR
BINDEC:  LBI    1,12       ; left-shift binary number one
         RC                ; bit with MSB going into C
LOOP1:   LD
         ASC               ; left-shift by means of binary double
         NOP
         XIS
         JP     LOOP1
DCDBL:   LBI    2,11       ; double BCD number
LOOP2:   LD
         AISC   6
         ASC
         ADT
         XIS
         JP     LOOP2
TEST:    LBI    2,10       ; test if finished
         LD
         AISC   1          ; done if bit counter = 15
         JP     X1
         RET
X1:      X
         JP     BINDEC
```

This routine uses no subroutines, takes 25 ROM words, and uses 10 digits of RAM – just as the first method. The existence of a CLEAR subroutine and/or a decimal double and/or a binary double routine in the program would further reduce the code required for this routine. As written, the routine does not preserve the binary number. The routine executes in 954 instruction cycles. Thus, it uses less code, overall, than the previous routine but executes substantially slower.

Variation 2 - The Shifting Algorithm

The left bit shift scheme shown in Section 5.2.9 is simply a binary double. The primary algorithm still requires the add one to the doubled BCD number when the binary bit is a one. This routine does the doubling a little differently: A binary double is performed on the BCD number with a subsequent decimal correct. The RAM map for this scheme is given in Figure 5-17. The flow chart is in Figure 5-18. The flow chart is general for the algorithm. Judicious placement of data in RAM eliminates the need for the digit counter, DC, shown in the flow chart.

		Bd								
		15	14	13	12	11	10	9	8	7
Br	0	M S D	BCD NUMBER				M S D	M S B	BINARY NUMBER	L S B
	1	BIT COUN- TER								

Figure 5-17. RAM Map for Binary to BCD Conversion

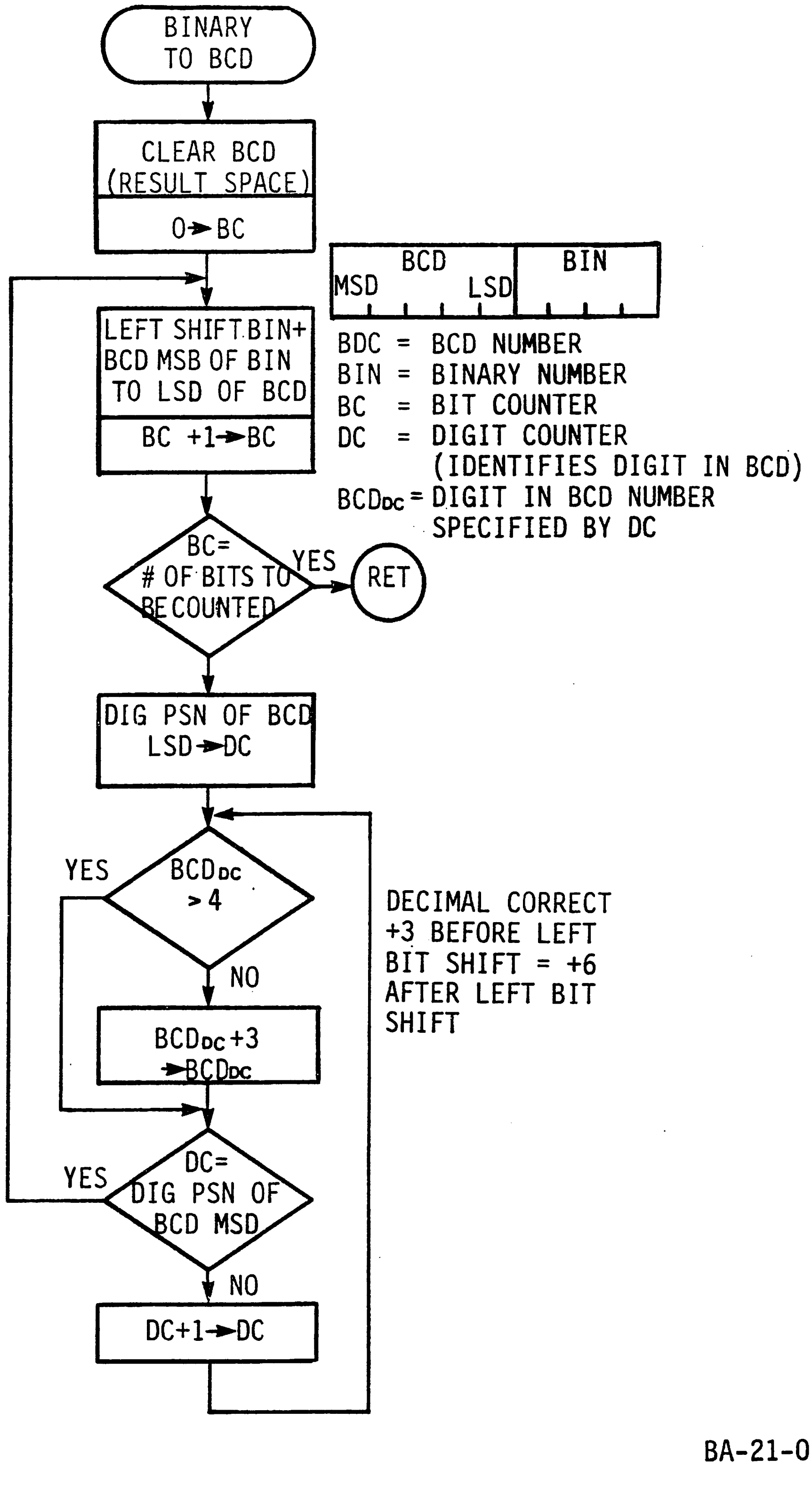


Figure 5-18. Binary to BCD Conversion – Shifting Algorithm

The routine as written uses no subroutines. However, the presence of a binary double routine in the program would reduce the code required in this routine.

The code is as follows:

```
BINBCD:   LBI    0,11      ; 0 -> BCD
CLEAR:    CLRA
          XIS
          JP     CLEAR
          LBI    1,15      ; 0 -> bit counter
          STII   0
SHIFT:    RC               ; left bit shift binary number
          LBI    0,7
LOOP1:    LD               ; and BCD number
          ASC
          NOP
          XIS
          JP     LOOP1
FINISH:   LBI    1,15      ; test if done
          LD
          AISC   1
          JP     RDCADJ    ; no done, BCD adjust
          RET
BCDADJ:   X
          LBI    0,11
DECADJ:   LD               ; decimal adjust before left shift
          AISC   11        ; number > 9 after shift is > 4 before shift
          JP     LESS5
          AISC   8         ; number > 4; do net +3 (=+6 after shift)
          NOP
          X
LESS5:    LD
          XIS              ; go through the whole number
          JP     DECADJ
          JP     SHIFT
```

The routine takes 31 ROM words, uses 10 RAM digits, and has a worst case execution time of 1525 instruction cycle times. Unlike the other routines, this routine is data dependent.

5.2.11 BCD to Binary Conversion

Several methods of performing a BCD to binary conversion are presented. The methods, like the binary to BCD routines in Section 5.2.9, illustrate different algorithms and different programming techniques.

An Efficient Two-Digit BCD to Eight-Bit Binary Routine

Figure 5-19 is the flow chart for a very efficient routine for converting a two-digit BCD number to its binary equivalent. The routine uses only two digits of RAM, replacing the BCD number with its binary representation. The simple RAM map is indicated below:

		Bd	
		15	14
Br	0	BCD NUMBER BINARY RESULT	

The routine is as follows:

```
BCDBIN:   LBI    0,15       ; clear MSD of BCD number, save value in A
          CLRA
NOCARY:   X
          LBI    0,14       ; point to LSD of BCD number
          AISC   15         ; if MSB (saved in A) = 0, done
          RET
          X                 ; else subtract 1 from it
          AISC   10         ; add 10 to the "binary" number
          JP     NOCARY
          XIS               ; if carry, must add 1 to binary "MSB"
          X
          AISC   1
          JP     NOCARY     ; loop until done. This word never
                            ; can be skipped since max BCD number = 99
```

This is a simple routine implemented in an "obscure" manner. The routine takes only 13 words, uses 2 RAM digits only, and has a worst case execution time of 104 instruction cycles. The execution time is data dependent. The minimum execution time is 6 instruction cycles.

Some attention should be given to this routine. It is a good example of code sharing, efficient use of memory, and clever use of the instructions. The routine uses the accumulator both as temporary storage and as work space for the arithmetic. The two RAM digits also serve multifunctions such as accumulate the result, temporary storage, and the input number. To be sure, a great deal of this routine's efficiency comes from the fact that we are working with only two digits. However, the techniques with the accumulator illustrated in this routine have much broader applicability.

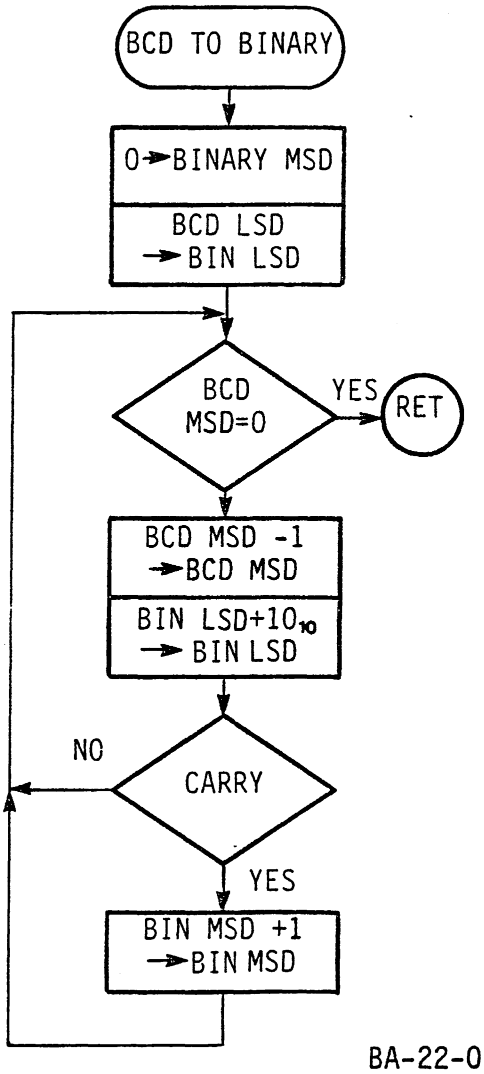


Figure 5-19. Two-Digit BCD to Binary Conversion

BCD to Binary Conversion—Multiply By 10

This routine is the counterpart to the binary to BCD conversion by the doubling technique. As written, the routine will convert a five-digit decimal number ($\leqslant 65535$) to its 16-bit binary equivalent. The routine again comes from the standard expansion

$$d_4d_3d_2d_1d_0 = d_4 10^4 + d_3 10^3 + d_2 10^2 + d_1 10^1 + d_0$$

d_n = digit within a decimal number.

The preceding expression can be rewritten in the following form:

$$d_4d_3d_2d_1d_0 = 10\left\{10[10\langle 10(d_4) + d_3\rangle + d_2] + d_1\right\} + d_0$$

By merely evaluating the right-hand expression above (and adding and multiplying by 10 in binary), the conversion is accomplished. For general information, the scheme is number base independent and can, therefore, be used to convert a decimal number to any desired number base: merely carry out the adds and multiplies in the desired base.

Analysis of the expression above yields the following iterative procedure: Multiply MSD of decimal number by 10, add the next MSD to that quantity, multiply the result by 10, add the next MSD to that result, multiply the result by 10, etc. until the LSB of the decimal number is added. The conversion is complete after this final addition.

The Straight-Forward Implementation

The flow chart of Figure 5-20 is the direct expression of the preceding math. The first example will be a straight-forward implementation of that flow chart. The routine is written to convert a five-digit BCD number (< 65536) to a 16-bit binary number. The RAM map is shown below.

		Bd 15	14	13	12	11
	0	WORK SPACE - SCRATCH PAD				Not Used
Br	1	BINARY NUMBER				Digit Counter
	2	MSD	BCD NUMBER			LSD

The routine uses the following subroutines, assumed to be located in Page 2.

```
CLEAR:   CLRA
CLEAR2:  XIS
         JP     CLEAR
         RET
BINADD:  RC                 ; R1 + R0 -> R1, 16 bit
         LBI    0,12
BINAD1:  LD     1
         ASC
         NOP
         XIS    1
         JP     BINAD1
         RET
BINDBL:  LBI    1,12        ; 2 x R1 -> R1, binary
         RC
DBL:     LD
         ASC
         NOP
         XIS
         JP     DBL
         RET
```

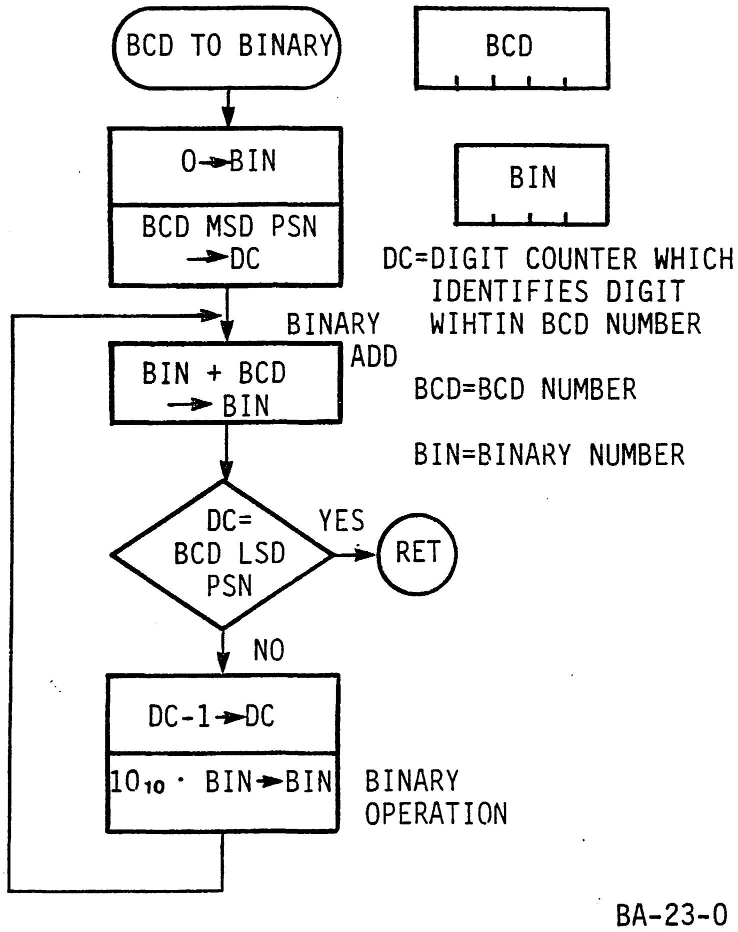


Figure 5-20. BCD to Binary Conversion - Multiply by 10

The main body of the routine is as follows:

```
BCDBIN:  LBI   1,11     ; load digit counter
         STII  15       ; position of BCD MSD
         JSRP  CLEAR    ; 0 -> binary number
DECBIN:  LBI   1,11
         LD    3
         CAB            ; point to digit in BCD number
         LD
         LBI   0,12     ; and put it into R0
         JSRP  CLEAR2   ; note call into middle of subroutine
         JSRP  BINADD   ; add the digit to the rest of the number
         LBI   1,11     ; now test if finished
         LD
         AISC  4        ; if digit counter <12 done
         RET
         AISC  11       ; this results in net subtract 1
         X              ; save new value of digit counter
         JSRP  BINDBL   ; now multiply by 10, first do 2 x R1 -> R1
         LBI   1,12
XFER:    LD    1        ; transfer R1 -> R0, could be a subroutine
         XIS   1
         JP    XFER1
         JSRP  BINDBL   ; 2 x R1 -> R1
         JSRP  BINDBL   ; 2 x R1 -> R1
         JSRP  BINADD   ; result after all this is 10 x R1 -> R1, binary
         JP    DECBIN
```

The routine uses 25 ROM words, not counting the 20 words of subroutine. The routine uses 14 RAM digits and executes in 709 instruction cycles. The original number is preserved and the routine uses one subroutine level.

The Shifting Approach

Consider a slightly modified version of the preceding routine. Figure 5-21 shows this modification. The important differences are 1) there is no digit counter or pointer and 2) the original number is lost. The RAM map follows Figure 5-21.

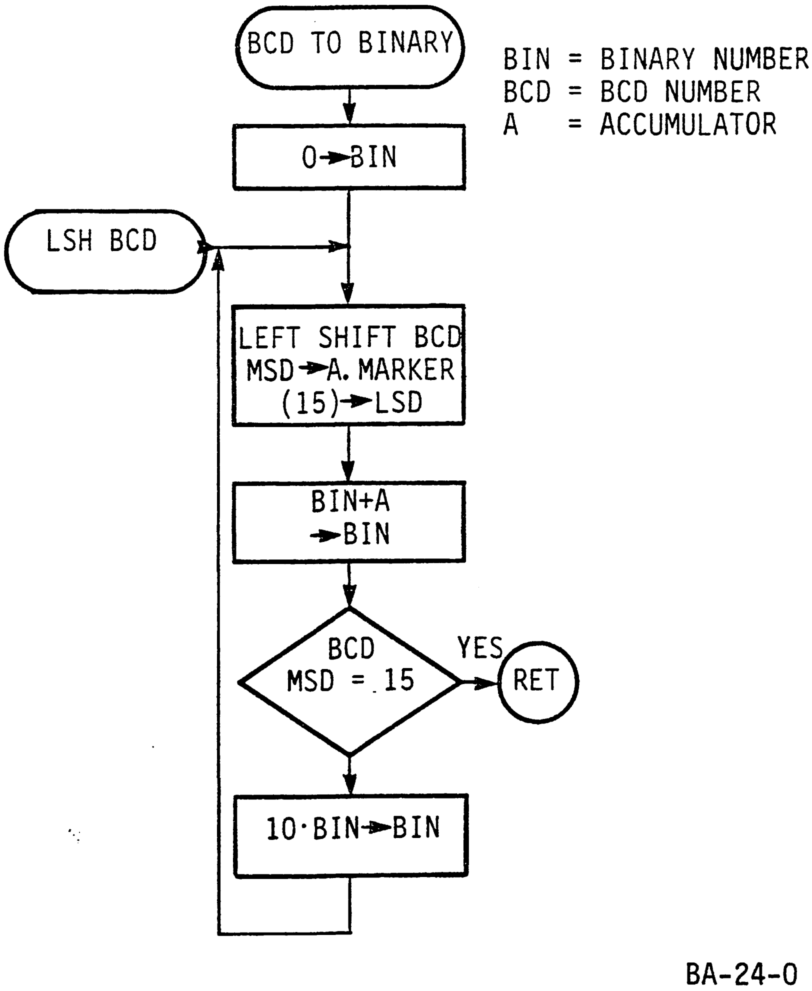


Figure 5-21. BCD to Binary Conversion – Multiply by 10 – the Shifting Approach

		Bd				
		15	14	13	12	11
	0	WORK SPACE - SCRATCH PAD				Not Used
Br	1	BINARY NUMBER				Not Used
	2	MSD		BCD NUMBER		LSD

This version of the routine uses the same subroutines as given in the preceding section.

```
BCDBIN:   LBI    1,12      ; clear binary number
          JSRP   CLEAR
LSHBCD:   CLRA             ; left shift BCD number, MSD -> A;
          COMP             ; marker to LSD
          LBI    2,11
LSH:      XIS
          JP     LSH       ; A -> R0, digit 12, 0 to rest of R0
          LBI    0,12
          JSRP   CLEAR2
          JSRP   BINADD    ; add the digit to converted value
          LBI    2,15
          LD               ; test for done, if BCD MSD=15, finished
          AISC   1
          JP     TIMES10
          RET
TIMES10:  JSRP   BINDBL    ; multiply by 10
          LBI    1,12
XFER:     LD     1
          XIS    1
          JP     XFER1
          JSRP   BINDBL
          JSRP   BINDBL
          JSRP   BINADD
          JP     LSHBCD
```

This outline uses 24 ROM words, not counting the 20 subroutine words and 13 RAM digits. It executes in 731 instruction cycles.

This second approach, a slightly different implementation of the same basic algorithm, uses slightly less memory and executes slightly slower than the straight-forward approach.

A "Paper and Pencil" Method and a Common Mistake

One of the standard methods for base conversion, at least on paper, is the technique of successively dividing the original number by the destination base. The remainders constitute the digits of the converted number. Thus to convert from BCD to binary, simply divide the BCD number by two repeatedly. See the simple example below:

	NUMBER	REMAINDER	
2	17		
	8	1	LSB
	4	0	
	2	0	
	1	0	
	0	1	MSB

Thus, $17_{10} = 1000_{12}$. The technique is well established and useful in instruction.

A conversion scheme using this algorithm is presented. This scheme is presented for comparison and for illustration of certain techniques, *e.g.*, a decimal divide by two without a divide routine. This particular scheme for BCD to binary conversion is not recommended since it is neither more memory efficient nor faster than techniques previously shown. This approach is simply the implementation of a well known conversion technique and serves to illustrate the effect of the algorithm itself on the code.

The flow chart for this algorithm is given in Figure 5-22. The RAM map is given below.

	Bd: 15	14	13	12	11	10
Br 0	MSD	BCD NUMBER			LSD	Scratch Pad
Br 1	SCRATCH PAD - WORK SPACE					
Br 2	BINARY NUMBER				Scratch Pad	Bit Counter

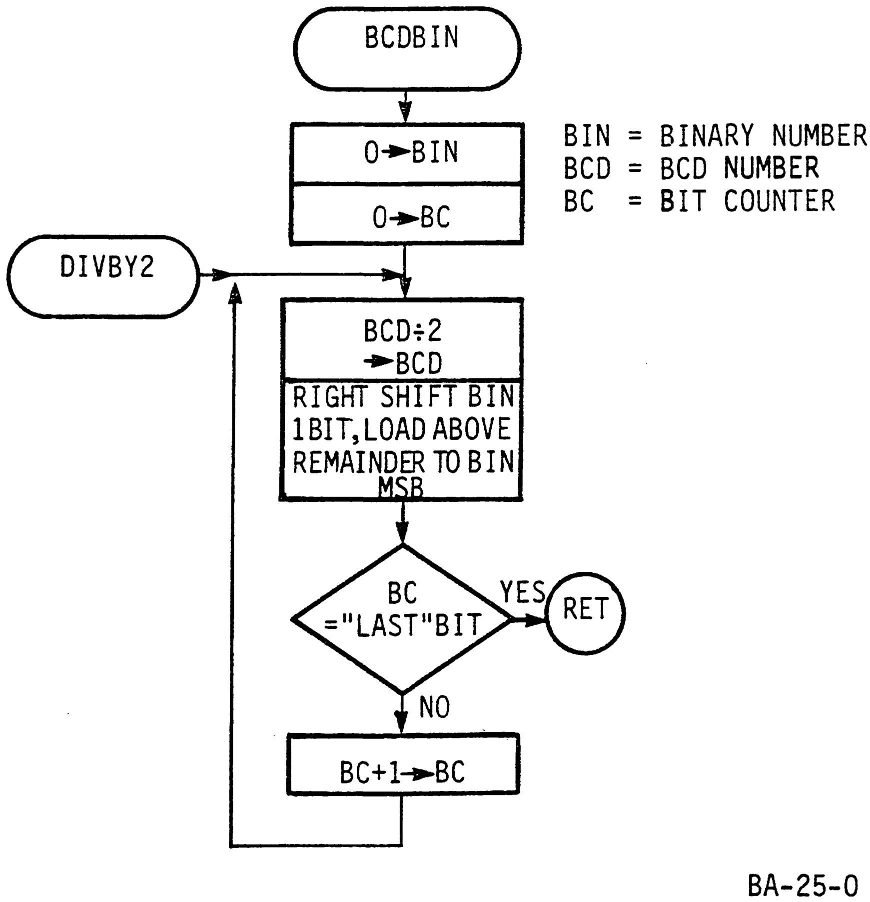


Figure 5-22. BCD to Binary Conversion by Successive Divide by Two

The routine uses the following subroutines, assumed to be located in Page 2.

```
RSH:      XDS              ; a simple 5- or 6-digit right shift
RSH2:     XDS
          XDS
          XDS
          XDS
          X
          RET
BINDBL:   RC               ; 2 x R2 -> R2; binary
          LBI    2,11
BDLOOP:   LD
          ASC
          NOP
          XIS
          JP     BDLOOP
          RET
ADD:      RC               ; R1+R0-> R0, decimal
          LBI    1,10
ADLOOP:   LD     1
          AISC   6
          ASC
          ADT
          XIS    1
          JP     ADLOOP
          RET
```

The main body of the routine follows:

```
BCDBIN:   LBI    2,10      ; 0 -> binary number and initialize
CLEAR:    CLRA
          XIS              ; bit counter
          JP     CLEAR
DIVBY2:   LBI    0,15      ; divide BCD number by 2, by first
          CLRA             ; divide by 10 (right digit shift) and
          JSRP   RSH       ; then multiply by 5
          LBI    0,10
XFER1:    LD     1         ; R0 -> R1, for subsequent adds
          XIS    1
          JP     XFER1
          JSRP   ADD       ; 2 x R0
          JSRP   ADD       ; 3 x R0
          JSRP   ADD       ; 4 x R0
          JSRP   ADD       ; 5 x R0, therefore have net divide by 2
          CLRA
          LBI    0,10      ; fetch the remainder
          RMB    2         ; make sure it is only 0 or 1
          X
          LBI    2,15      ; load remainder to binary number and
          JSRP   RSH2      ; shift right 4 bits
          JSRP   BINDBL    ; now shift left 3 bits
          JSRP   BINDBL
          JSRP   BINDBL    ; net effect is 1-bit right shift of
                           ; binary number with divide remainder going into MSB
TSTFIN:   LBI    2,10
          LD               ; test bit counter for done, if not
          AISC   1         ; increment bit counter
          JP     TFIN2
          RET
TFIN2:    X
          JP     DIVBY2
```

This routine takes 31 ROM words, plus 24 words in the subroutine page; uses 18 RAM digits; and executes in approximately 5500 instruction cycles.

There is little reason to recommend this routine over the others presented. It takes significantly more memory and executes significantly slower. The only benefit, if it can be so termed, is that it implements a commonly known procedure. The lesson is obvious: Do not assume that the "standard" procedure will yield the most efficient implementation. The programmer should be prepared to investigate various algorithms and approaches in the interest of efficiency.

5.3 TIMEKEEPING ROUTINES

Several routines for keeping time are presented. These include routines for a 12- or 24-hour clock based on internal or external timing references.

5.3.1 Basic Clock Routines - External Input

The following two routines implement a basic clock. The two routines do the same thing. One is written as a 12-hour clock, the other as a 24-hour clock. This, however, is not the significant difference between them. Both routines use the RAM map below, and it is assumed that both the routines are called once per second on the basis of a 1-Hz input signal.

	Bd					
	15	14	13	12	11	10
Br 3	Hours MSD	Hours LSD	Minutes MSD	Minutes LSD	Seconds MSD	Seconds LSD

The flow chart of Figure 5-23 applies to both routines. The flow chart indicates the minor differences when implementing a 12- or 24-hour clock. Note that both routines have implemented the same flow chart in different ways.

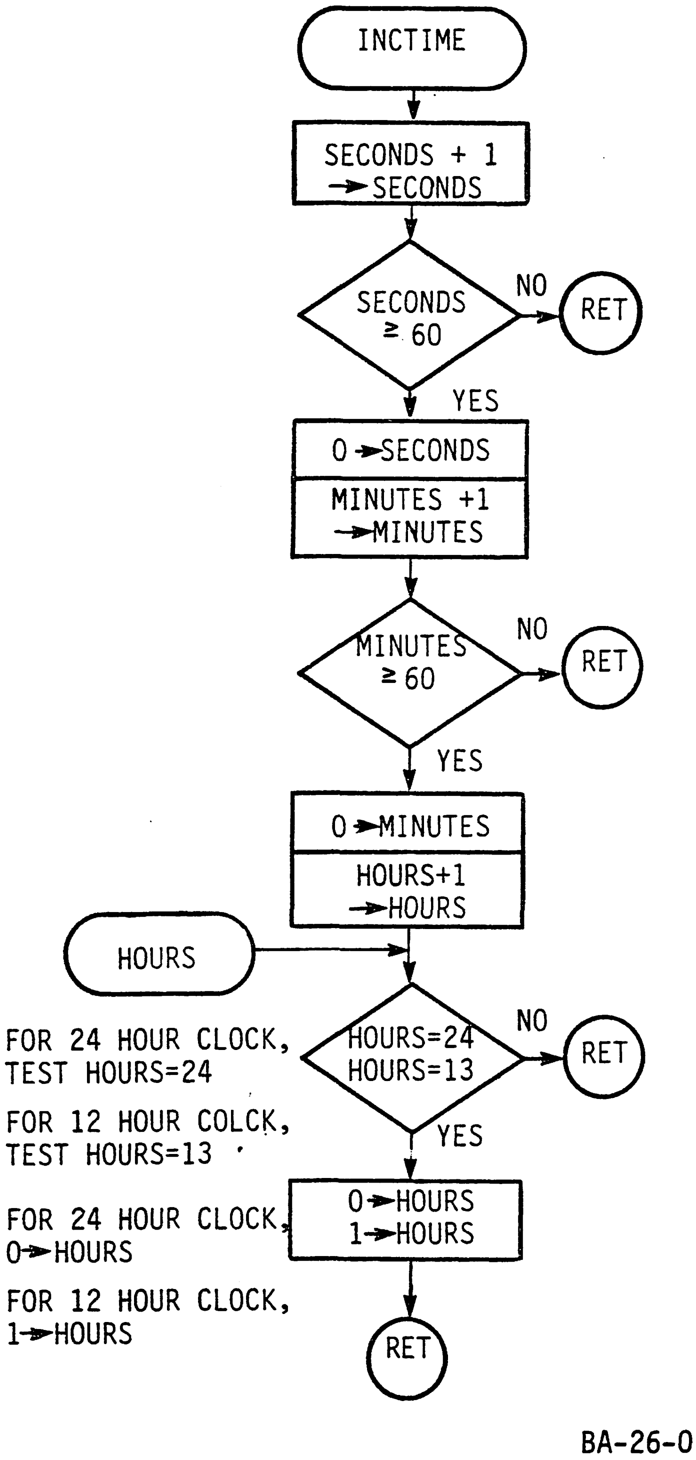


Figure 5-23. Basic Block Flow Chart

The first implementation, Version I, uses a master increment loop which increments seconds, minutes, and hours as required. The loop handles the overflow from 60 to 00 in the seconds and minutes. Version I is written as a 24-hour clock.

```
INCTIME:  LBI    3,10     ; point to seconds LSD
PLUS1:    SC              ; add 1
          CLRA
          AISC   6
          ASC
          ADT
          XIS             ; LSD incremented, point to MSD
          CLRA
          AISC   6        ; increment saved in C
          ASC             ; increment MSD of seconds, minute or hours
          ADT
          X               ;
                          ; test = 6, if so correct to 0 and move
          LD              ; to next digit. If not, exit
          AISC   10
          JP     HOURS    ; will always escape loop here if get to hours
          STII   0
          JP     PLUS1
HOURS:    LBI    3,15     ; test if hours need to be corrected
          LD              ; here testing for hours ≥24
          AISC   14
          RET             ; hours ≤20
          LBI    3,14
          LD
          AISC   12
          RET             ; hours <24
          STII   0        ; hours ≥, therefore set to 0
          STII   0
          RET
```

This routine takes 28 ROM words, 6 RAM digits, and has a worst case execution time of 58 instruction cycle times. The routine uses no subroutines and execution time is data dependent. Minimum execution time is 19 instruction cycle times.

The second implementation, Version II, is a more direct implementation of the flow chart shown in Figure 5-23. It moves sequentially through the clock data, incrementing and adjusting as required.

```
INCTIME:  LBI    3,10    ; point to seconds LSD
          JSRP   PLUS1   ; 2 digit BCD increment
          LD
          AISC   10      ; mod 6 correct
          RET            ; seconds MSD < 6, exit
          XIS
          JSRP   PLUS1   ; increment minutes
          LD             ; mod 6 correct
          AISC   10
          RET            ; minutes MSD < 6, exit
          XIS
          JSRP   PLUS1   ; increment hours
          LD             ; now do hours adjust
          AISC   15
          RET            ; exit if hours MSD = 0
          LBI    3,14    ; hours MSD = 1, test hours LSD < 3
          LD
          AISC   13
          RET            ; hours ≤ 12 - exit
          STII   1       ; hours = 13, set to 01 and exit
          STII   0
          RET
```

The routine uses the following subroutines, assumed to be located in Page 2.

```
PLUS1:  SC           ; 2 digit BCD increment
        CLRA
        AISC   6
        ASC
        ADT
        XIS
        CLRA
        AISC   6
        ASC
        ADT
        X
        RET
```

The routine takes a total of 34 ROM words, 22 in the main routine and 12 in the subroutine; uses 6 RAM digits; and has a worst case execution time of 58 instruction cycle times. Execution time is data dependent with the minimum execution time being 17 instruction cycles. The routine uses one subroutine level.

1-Hz Input and 50- or 60-Hz Input

The two routines provided are written assuming they are called as a result of a 1-Hz signal. It is a simple task to modify the routines for a 50- or 60-Hz input signal. As Version I is the more code efficient routine, the necessary modifications will only be illustrated for that implementation.

60-Hz Only Input

If the signal source is a 60-Hz signal, the modification is trivial. By simply changing the first LBI from LBI 3,10 to LBI 3,8, the routine becomes a clock increment based on a 60-Hz input. The rest of the routine is completely unchanged. Of course, two extra RAM digits are used, digits 3,9 and 3,8, to count the 60-Hz signal. Also, as should be expected, worst case execution time increases.

A General 50- or 60-Hz Input

It is fairly simple to modify the routine to operate with either a 50-Hz or 60-Hz reference input. The modification will use the characteristic described in the preceding paragraph. For a 50-Hz input, the frequency counter is set to 10 rather than 00. Otherwise, the routine remains the same. The routine arbitrarily selects G2 as the input line to define whether the input is 50-Hz or 60-Hz. Figure 5-24 is the flow chart shown in Figure 5-23 modified to indicate the specific implementation and the 50- or 60-Hz feature.

	Bd 15	14	13	12	11	10	9	8
Br 3	Hours MSD	Hours LSD	Minutes MSD	Minutes LSD	Seconds MSD	Seconds LSD	Counter MSD	Counter LSD

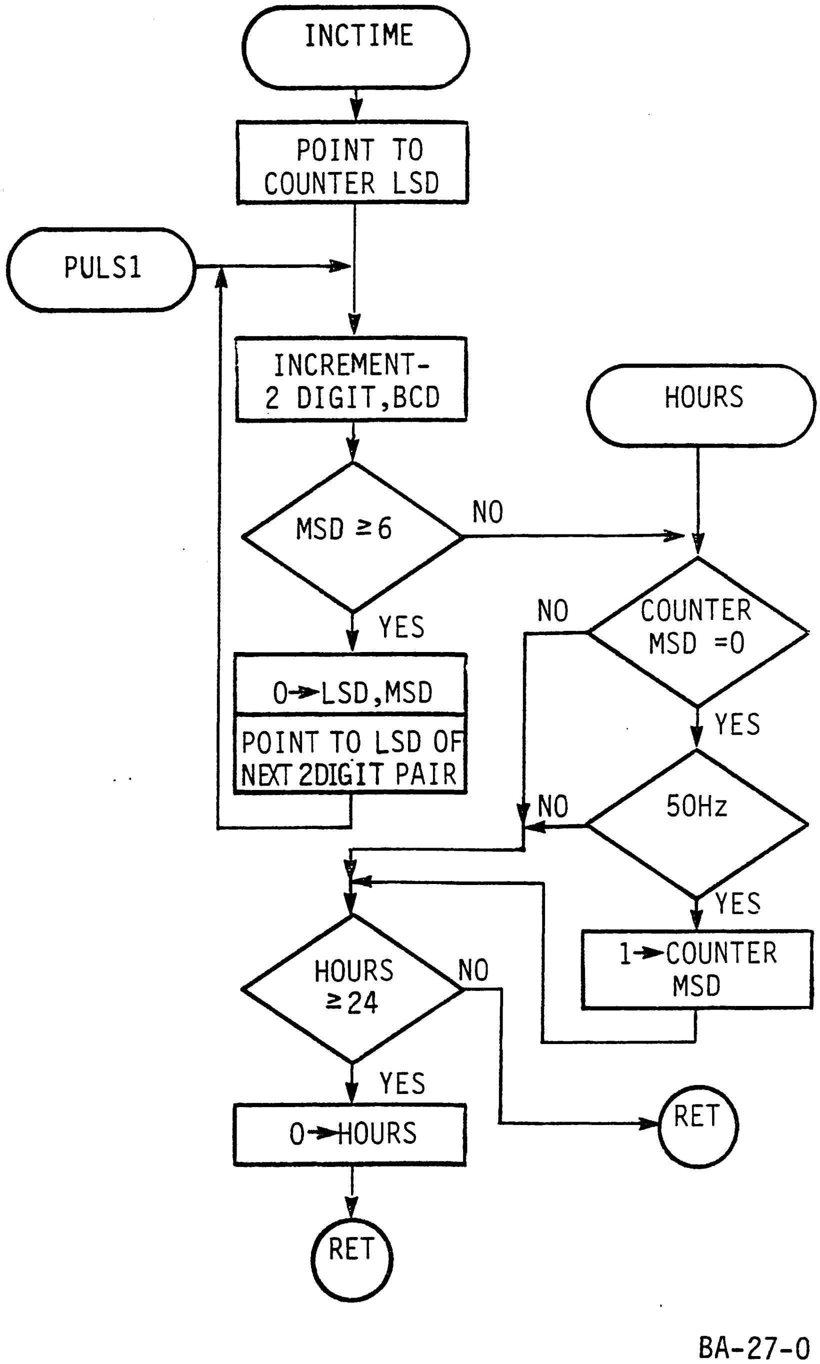


Figure 5-24. Clock Based on 50- or 60-Hz Input

```
INCTIME:  LBI     3,8       ; point to counter LSD
PLUS1:    SC
          CLRA              ; 2 digit BCD increment by 1
          AISC    6
          ASC
          ADT
          XIS
          CLRA
          AISC    6
          ASC
          ADT
          X
          CLRA
          AISC    6         ; now test MSD ≥6
          SKE
          JP      HOURS
          STII    0
          JP      PLUS1     ; is ≥6, correct to 0 and continue
HOURS:    LBI     3,9       ; test counter MSD = 0
          CLRA
          SKE
          JP      HOURS2
          OGI     15        ; now test 50- or 60-Hz, set G2 high
          SKGBZ   2
          STII    1         ; if 50-Hz, 1-> counter MSD
HOURS2:   LBI     3,15      ; G2 = 1 indicates 50-Hz input
          LD
          AISC    14
          RET               ; hours MSD <2
          LBI     3,14      ; hours MSD = 2, test hours LSD <4
          LD
          AISC    12
          RET               ; hours <24
          STII    0
          STII    0         ; hour ≥24, set to 0
          RET
```

The routine uses 39 ROM words, the extra words being used to read the input and adjust the counter accordingly, and 8 RAM digits. Input $G_2 = 1$ indicates a 50-Hz input signal.

12- or 24-Hour Capability

It is a trivial matter to expand the routine further to give it the option of 12- or 24-hour capability. Figure 5-23 indicates the differences, which are minor. One need only test another input and alter the hours digits accordingly.

5.3.2 Clock Routines Based on Internal Timer

The internal timer of COPS microcontrollers can be used as the time reference for a clock. Routines using this feature must count timer overflows. These overflows are dependent, of course, on the operating frequency of the microcontroller. This points out a major restriction on this type of clock routine: It is impossible for the clock to be more accurate than the oscillator frequency. Another difficulty is that the selection of operating frequency may give a fractional SKT, timer overflow, frequency. This complicates the routine by requiring compensation for this fractional frequency.

An SKT-Based Timekeeping Routine

The following routine is representative of the worst case conditions when using the internal timer as a clock time base: A common, inexpensive crystal is used for the oscillator and creates a fractional SKT frequency. The following information is essentially a duplication of Section 4.9 of the *COPS Family User's Guide*. It is presented here for completeness.

The routine presented here is a 12-hour clock using the SKT overflow as the time base. The oscillator used will be based on a 3.579545 MHz-crystal, the inexpensive, readily available TV crystal. Therefore, a high-speed part (*e.g.*, COP420) with the divide by 16 option must be used. The SKT overflow frequency is the instruction cycle frequency (here 3.579545 MHz divided by 16) divided by 1024 or, in this case, 218.478 Hz. Therefore, the timekeeping calling routine must execute an SKT instruction at an approximate 218-Hz rate to guarantee detection of every SKT overflow. The routine must compensate for the non-integer SKT overflow frequency to provide timing accuracy.

Compensation is achieved by establishing a counter for the SKT overflows. Seconds are incremented when this counter reaches 0. This counter is preset to various values, from which it is counted down, at various points in the routine. The details of the compensation are as follows:

- Every odd second in the range of 0-59 seconds, the counter is set to 218.
- Every even second in the range of 0-59 seconds, the counter is set to 219.
- Every minute in the range of 0-59 minutes, the counter is set to 218.
- Every hour the counter is set to 199.

Regardless of the preset, the counter is decremented every time the SKT instruction skips, *i.e.*, an SKT overflow is detected. The technique previously described will provide accuracy at the end of each hour. The short term inaccuracies during the hour are small. The *COPS Family User's Guide* explains why this particular compensation scheme works and the reader is referred to the manual for explanation.

Figure 5-25 is the flow chart and RAM map for this routine. Note that the counter for SKT overflows is binary. Also note that the hours portion of the clock is binary, to save RAM, and that the minutes and seconds portions of the clock are BCD. The routine is located outside Page 2 and uses a subroutine located in Page 2.

	Bd 15	14	13	12	11	10	9
Br 2	COUNTER FOR SKT OVERFLOWS MSD	LSD	HOURS (BIN-ARY)	MINS MSD (BCD)	MINS LSD (BCD)	SECS MSD (BCD)	SECS LSD (BCD)

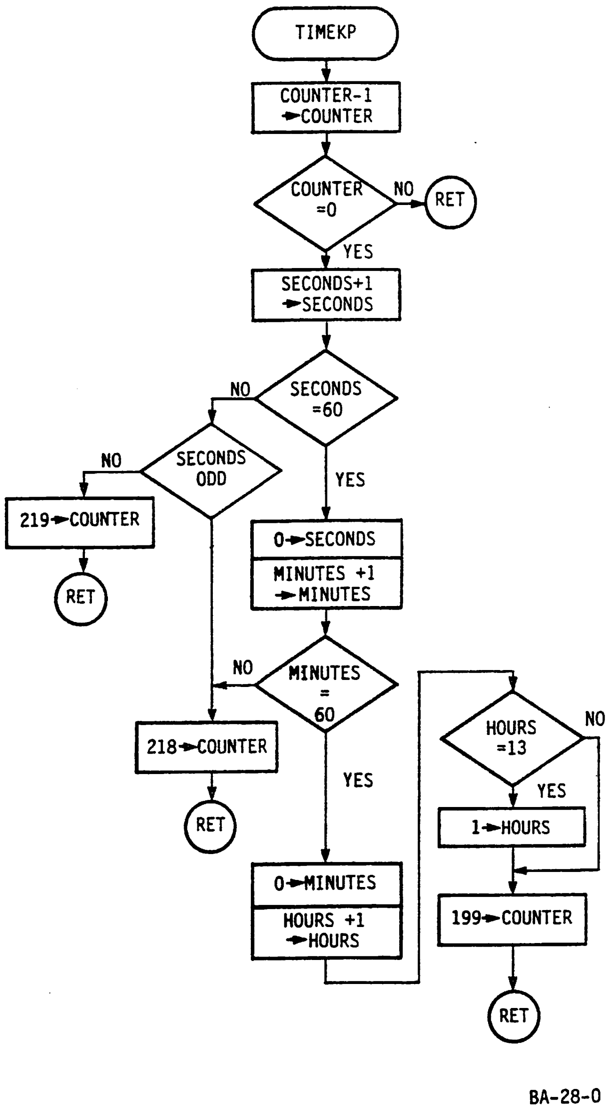


Figure 5-25. Flow Chart for Internal Time Base Clock (Oscillator Frequency = 3.579545 MHz)

```
TIMEKP:   LBI    2,14      ; point to low-order digit of counter
DECR:     LD               ; decrement the counter by 1
          AISC   15
          JP     NEXTDIG
          X                ; counter = 0 return to main routine
          RET
NEXTDIG:  XIS              ; if skip executed, counter is 0
          JP     DECR
SECONDS:  LBI    2,9       ; points to seconds LSD
          JSRP   INC2      ; 2 digit BCD increment with MOD6 adjust
          JP     TSEC      ; seconds <60, test ODD or EVEN
          STII   0         ; seconds = 60, 0 -> seconds, increment mins.
          JSRP   INC2
          JP     C218      ; minutes <60, set counter = 218
          STII   0         ; 0 -> minutes, increment hours
          LD
          AISC   1
          X
          AISC   4         ; test hours >12
          JP     C199      ; no, set counter to 199
          STII   1         ; yes, set hours to 1 and counter to 199
C199:     LBI    2,14
          STII   7         ; set counter = 199 (binary 12,7)
          STII   12
          RET
TSEC:     LBI    2,9       ; point to seconds LSD to test ODD/EVEN
          SKMBZ  0
          JP     C218      ; seconds ODD, set counter to 218
C219:     LBI    2,14      ; seconds EVEN, set counter to 219
          STII   11        ; 219 = binary 13,11
C21X:     STII   13
          RET
C218:     LBI    2,14      ; 218 = binary 13,10
          STII   10
          JP     C21X
```

This routine uses the following subroutine:

```
INC2:   SC              ; 2-digit BCD increment
        CLRA
        ASC
        ADT
        XIS
        CLRA
        AISC    6
        ASC
        ADT
        X               ; now test if reached 60
        LD
        AISC    10
        RET             ; 2 digits <60
        RETSK           ; 2 digits = 60
```

It should be clear that a more convenient choice of oscillator frequency would significantly reduce the code in this routine. An integer SKT overflow frequency would reduce the routine to, essentially, one of the routines shown initially.

5.4 DATA MANIPULATION AND STRING OPERATIONS

5.4.1 Register Transfers

Several routines are provided for transferring data between registers. Some more or less specialized routines are presented along with a completely general routine.

Four Register Blocks

The LD, XIS, XDS, and X instructions have an exclusive OR argument which permits easy data transfer among the registers within a four register block, registers 0-3, 4-7, etc. Moving data across a register block boundary is less efficient and the general purpose routines have to be used. Within the register block, the following routines can be used:

```
XFER1:  LD   1          XFER2:  LD   2          XFER3:  LD   3
        XIS  1                  XIS  2                  XIS  3
        JP   XFER1              JP   XFER2              JP   XFER3
        RET                     RET                     RET
```

NOTE: XDS can be used in place of XIS in any of these routines.

Routine XFER1 will transfer data from R0 to R1, R1 to R0, R2 to R3, or R3 to R2. Routine XFER2 will transfer data from R0 to R2, R2 to R0, R1 to R3, R3 to R1. Routine XFER3 will transfer data from R0 to R3, R3 to R0, R1 to R2, or R2 to R1. The direction of the transfer depends only on the status of the B register when the routine is executed. In fact, the routines are commonly preceded by one or more LBI instructions. The successive skip feature of the LBI instruction is very powerful when used in conjunction with these routines.

Register exchanges within the four register blocks are written in much the same way as the following routine indicates.

```
SWAP1:  LD    1
        X     1
        XIS   1
        JP    SWAP1
        RET
```

This routine will exchange the contents of the R0 and R1 or R2 and R3. Similar routines for the other registers can also be written in the same manner as the data transfers. Again, XDS may be used in place of XIS.

Completely General Transfers

A completely general register transfer routine is indicated below. The routine uses a RAM digit for temporary storage. The routine is called by setting up the source register with an LBI and establishing the destination register number in the accumulator. RAM digit TEMP is any convenient digit.

```
LOOP:   XAD     TEMP
        XABR
XFER:   XAD     TEMP    ; XFER is the entry point for the routine
        LD
        XAD     TEMP
        XABR
        XAD     TEMP
        XIS
        JP      LOOP
        RET
```

The calling sequence for the routine is as follows:

```
LBI     SOURCE
CLRA
AISC    N               ; N defines destination register
JSRP    XFER
```

Obviously, if a transfer from RN to RK is common, the setup can be included in the subroutine.

The routine can be rewritten in the following form and the calling sequence modified as follows:

CALLING SEQUENCE:

```
LBI     TEMP
STII    N           ; destination register
LBI     SOURCE
JSRP    XFER
```

The subroutine is as follows:

```
LOOP:  JSR    EXCH     EXCH:  XAD    TEMP
XFER:  LD                     XABR
       JSR    EXCH            XAD    TEMP
       XIS                    RET
       JP     LOOP
       RET
```

There is no particular benefit in doing this for the simple register transfer but it will result in code savings where register swaps, general purpose swaps, are also required.

The routine for a general purpose register swap and the calling sequence are given below.

CALLING SEQUENCE:

```
LBI     TEMP
STII    N           ; one register number
LBI     SOURCE
JSRP    SWAP
```

The SWAP subroutine is:

```
SWAP2:  JSR    EXCH
SWAP:   LD                ; entry point for the routine
        JSR    EXCH
        X
        JSR    EXCH
        XIS
        JP     SWAP2
        RET
```

Subroutine EXCH is the same routine as indicated in the general purpose transfer.

5.4.2 Shift Routines

Right Digit Shift

The following routines will perform right digit shifts. The first routine shifts right one digit from the starting B address to the end of the register. The second routine shifts an arbitrary four-digit group right one digit. Both routines place a "0" in the starting digit and leave the previous contents of the last digit in the accumulator.

I

```
RSHIFT:  CLRA            ; to put 0 to first digit
RSH:     XDS
         JP      RSH     ; simple right shift loop, exit on XDS skip
         RET
```

RSHIFT:

```
         XDS
         XDS             ; shift 4-digit block right one digit
         XDS
         X               ; save value of last digit in A
         RET
```

Left Digit Shift

The following routines will perform left digit shifts. The first routine shifts left one digit from the starting B address to the end of the register. The second routine shifts an arbitrary four-digit group left one digit. Both routines place a "0" in the starting digit and leave the previous contents of the last digit in the accumulator.

I

```
LSHIFT:  CLRA            ; to put 0 to first digit
LSH:     XIS
         JP      LSH     ; simple left shift loop, exit on XIS skip
         RET
```

II

```
LSHIFT:  CLRA            ; to put 0 to first digit
         XIS
         XIS             ; shift 4-digit block left on digit
         XIS
         X               ; save value of last digit in A
         RET
```

NOTE: The left and right digit shift routines are written in the sense that the direction of increasing Bd value is "left". The direction of decreasing Bd value is "right". It is entirely possible that the user may, for his or her application, wish to reverse this directional sense. This causes no problem and the routines above are merely reversed (*i.e.*, the left shifts become right shifts and vice-versa).

Right Bit Shift

A right bit shift is one of those very few things that COPS microcontrollers do not do well. If the algorithm or approach chosen involves right bit shifting, it is strongly recommended that an alternative approach be used or developed. An alternative nearly always exists and will commonly be COPS code efficient. Rarely, if ever, does the failure to find an alternative to right bit shifting mean that no alternative exists. The programmer should think in broader terms than the specific function of right bit shifting; if an algorithm requires right bit shifting, consider other algorithms for the same function.

However, if there is no choice and right bit shifting must be performed, some routines to perform the shift are presented. Note, right shift has the same directional sense here as in digit right shift; data movement is in the direction of decreasing Bd.

Right Shift Memory Digit 1 Bit

This routine is a simple, straight-forward approach to shift a memory digit right one bit. The shifted data is formed in the accumulator and then exchanged into memory. The routine can be written for a simple shift or a right circular shift. Both versions are indicated. The routines take advantage of the bit testing capability of COPS microcontrollers.

I - Simple Shift

```
RBSHIFT: CLRA
         SKMBZ  3
         AISC   4
         SKMBZ  2
         AISC   2
         SKMBZ  1
         AISC   1
         X
         RET
```

II - Circular Bit Shift

```
RBSHIFT: CLRA
         SKMBZ  3
         AISC   4
         SKMBZ  2
         AISC   2
         SKMBZ  1
         AISC   1
         SKMBZ  0
         AISC   8
         X
         RET
```

These routines are not particularly long nor complex and work well. They form the most efficient basis for general right bit shifting in COPS microcontrollers.

Right Shift Using SIO

If the SIO register is not otherwise being used, it can be used to perform a right circular shift of the data in the accumulator. This technique requires that pins SO and SI of the microcontroller be tied together externally. The routine is then reduced.

```
RSHIFT:  XAS   ; SIO must be in shift register mode
         NOP
         NOP
         XAS
         RET
```

The SIO register shifts left one bit each instruction cycle when it is enabled as a shift register. Thus, a right bit shift is achieved by three left bit shifts.

Left Bit Shifts

Left bit shifts are easy to perform even though there is no bit shift instruction. Bit left shift has the same directional sense as digit left shift; data movement is in the direction of increasing Bd.

Left Bit Shift by Means of Binary Double

Left shifting a value by one bit is equivalent to a binary doubling of that value. Thus, a binary doubling routine can be used for left bit shifting. Two routines are provided; one simply left shifts a single memory digit 1 bit; the other shifts several digits left 1 bit.

```
I - Single Digit          II - Multidigit

LBSHIFT: LD               LBSHIFT: RC
         ADD              LSHFT:   LD
         X                         ASC
         RET                       NOP
                                   XIS
                                   JP    LSHFT
                                   RET
```

These two routines perform the left shift in the same manner. The number is added to itself to do a binary double. The second version remembers the state of the MSB of a given digit in C so shifting can be performed across the digits.

Use of SIO for Left Bit Shifting

The SIO register can be used to shift the data in the accumulator left one bit. In the shift register mode, SIO is always shifting left. This normal operational feature can be used to advantage. The routine is simplicity itself:

```
LBSHIFT:   XAS    ; SIO must be in shift register mode
           XAS
           RET
```

A and SIO are simply swapped twice. Since SIO is always shifting (in shift register mode), this results in a net one bit left shift. This routine does not require that SI and SO be tied together and is therefore more or less unrestricted in its use. The user must remember that the state of SI, whatever it may be, is shifted into SIO and that the LSB of the accumulator after this routine will be controlled by the state of SI during the shift. Tying SI to SO will result in a left circular shift of one bit, the MSB of the accumulator will be moved to the LSB as the left bit shift occurs.

5.4.3 Data/String Compare

A routine to compare two strings of data or characters is provided. It is the same routine that would be used to compare two registers (within the four register blocks). The RAM map for this routine is indicated below:

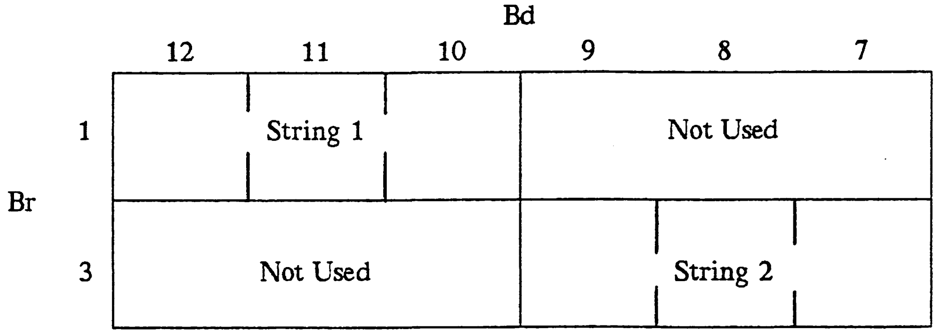
Br \ Bd	15	14	13	12	11
1			String 1		
2			String 2		

The routine is setup as a subroutine. It will simply return if the strings are not equal and return and skip if the two strings are identical. By changing the starting LBI, larger strings can be tested.

```
COMPARE:  LBI     1,11     ; initialize B
CMPR:     LD      3        ; load value to A, point to other register
          SKE              ; test equal
          RET              ; not equal, return
          XIS     3
          JP      CMPR
          RETSK            ; all digits equal, return and skip
```

The preceding routine is excellent if the data is placed so that it can be used. The programmer should strive to place data in RAM so that routines such as the one previously illustrated can be used. However, data is not always located in the most efficient places. Therefore, a general purpose compare routine is provided. This routine will compare a three-digit string located in

1,10, 1,11, and 1,12 to another three-digit string located in 3,7, 3,8, and 3,9.




```
COMPARE:  LBI     1,10   ; initialize B register
          LDD     3,7    ; fetch first digit to compare
          SKE
          RET            ; not equal
          XIS            ; point to next digit
          LDD     3,8    ; fetch second digit
          SKE
          RET            ; not equal
          XIS
          LDD     3,9    ; fetch third digit
          SKE
          RET            ; not equal
          RETSK          ; strings equal
```

This routine is general and the two strings could be located anywhere. By merely supplying the proper values in the LBI and LDD instructions, the routine is modified for data in locations other than those indicated here.

5.4.4 String Search

It is often necessary to search data memory for a string of characters. This routine will search register 0 for the three character string located in digits 2,15, 2,14, and 2,13. The routine simply returns if no match and returns and skips if the string is found.

```
SEARCH:  LBI     0,15    ; initialize B register
CHAR1:   LDD     2,15    ; fetch first character
         SKE
         JP      DECR    ; not equal, move B register
         XDS
         JP      CHAR2   ; matched first character, test second
         RET             ; string not found in register 0
CHAR2:   LDD     2,14    ; fetch second character
         SKE
         JP      CHAR1   ; no match
         XDS
         JP      CHAR3
         RET             ; string not found in register 0
CHAR3:   LDD     2,13    ; fetch third character
         SKE
         JP      INCR
         RETSK           ; string found
DECR:    LD              ; no match, move Bd down
         XDS
         JP      CHAR1   ; and start over
         RET             ; moved over the end, string not found
INCR     LD
         XIS
         JP      CHAR1
```

Remember, the routine is searching for the contiguous three-digit group and exists via RETSK when that group is found.

5.4.5 RAM Clear Routines

Routines that clear the data memory are commonly required in programs. Some of the more standard techniques are indicated here.

Single Register Clear

The following routines will clear all or part of a register. They are normally preceded by an LBI instruction.

```
              I                            II

CLEARX:  LBI    START   CLEARX:  LBI    START
CLR:     CLRA           CLR:     CLRA
         XIS                     XDS
         JP     CLR              JP     CLR
         RET                     RET
```

The routines are equivalent. Routine I clears the data in the register from the digit defined by START up to and including digit 15. Routine II clears the data in the register from the digit defined by START down to and including digit 0.

Clearing Entire RAM

It is a common requirement that the entire RAM be cleared at power up or on the basis of a master clear operation or both. This can be done by calling the register clear instructions provided previously. It will usually be more code efficient to use the routine provided here.

```
MCLEAR:  LBI    0,0
         CLRA
         AISC   N      ; N = highest number of register in device
                       ; N = 3 for COP420, N = 7 for COP444L, etc.
LOOP:    XABR
CLR:     CLRA          ; these three words could be replaced
         XIS           ; with a subroutine call to CLR
         JP     CLR    ; subroutine defined above
         XARR
         AISC   15     ; decrement BR
         JP     LOOP
```

The routine merely establishes the maximum value of BR allowed in the device - or desired to be cleared - and successively clears each register.

5.5 INPUT/OUTPUT

This section deals with the techniques for getting data in and out of COPS microcontrollers. Some of this is straight-forward since COPS devices have independent instructions for input and output.

5.5.1 Table Look Up

The LQID instruction makes outputting converted data very simple. It is powerful in its own right as a table look-up instruction but that power is increased if it is necessary to output the table values. A routine to output information is shown below. The table is not shown but is obviously required. Note that the table may be any kind of code conversion: BCD to Seven Segment, ASCII conversion, etc. The output is not affected by the table contents. By virtue of the successive LBI feature, the routine is set up to output either of two data streams.

		Bd 15	14	13	12	11	10	9	8
Br	0	Data Stream 1							
	1	Data Stream 2							

```
OUTDS1:  LBI   0,8      ; this entry point will output data stream 1
OUTDS2:  LBI   1,8      ; this entry point will output data stream 2
         CLRA
         AISC  N        ; setup accumulator for Table location
OUTPUT:  OBD            ; output digit position on D lines
         LQID
         X              ; this allows movement through
         XIS            ; the data without disturbing the data
         JP    OUTPUT   ; or the accumulator
         RET
```

The routine assumes that the L drivers have been enabled prior to calling the routine. Note that the LQID instruction loads the Q register. The L drivers must be enabled to output the data in Q. Remember also that the LQID instruction uses a subroutine level in some COPS microcontrollers.

5.5.2 Microbus I/O

Microbus I/O is, of course, relevant only to those COPS microcontrollers which have the Microbus option implemented. This option makes the code required for the interface simplicity itself. Only one caution is necessary: Do not enable the L drivers, *i.e.*, do not set EN_2, on Microbus parts. COPS Microbus devices are structured to be peripheral devices for some host processor. The host has control over the L drivers via the chip select, read strobe, and write strobe.

As stated earlier G0 is the handshake line for the Microbus interface. It is the responsibility of the COPS program to set G0 to a 1 level to indicate the COPS device is ready for access by the host. A write to the COPS Microbus peripheral by the host will set G0 low. A typical sequence for this is as follows:

```
         .
         .
         .
         OGI      1       ; G0 assumed low, 0 prior to this
                          ; set G0 high to indicate COPS ready
WAIT:    SKGBZ    0       ; wait for a write by host
         JP       WAIT
         CQMA             ; G0 was low, a write was performed
         .                ; read the data and continue with
         .                ; the program
         .
```

Note that when the host processor writes to a COPS Microbus device, the host writes directly into the Q register. The COPS microcontroller then merely reads the Q register.

A read by the host is equally simple. Upon seeing G0 high, the host will execute a read operation which takes the Q data out to the eight-bit bus. The only possible difficulty is that the COPS microcontroller does not know that a read has been performed. If it is necessary for the microcontroller to know a read has been performed, the following sequence is recommended.

```
         .
         .
         .
         CAMQ             ; load Q; could use LQID
         OGI      1       ; set G0 high to indicate data ready
WAIT:    SKGBZ    0
         JP       WAIT    ; host acknowledges ready by a dummy write
         JMP      MAIN
         .
         .
         .
```

This sequence outputs the data to Q and then sets G0 high to indicate ready. The host reads the data and then does a dummy write to indicate the data has been read. The microcontroller detects this and then returns to the main loop where G0 is set high and the device waits for the next write.

The procedure above is, of course, not necessary if there is no requirement that the COPS microcontroller know that a read operation by the host has taken place.

5.5.3 Serial I/O - MICROWIRE

Routines for handling serial I/O are provided. Two versions of output routines are provided: a destructive output and a nondestructive output. The routines are written for 16-bit transmissions but are trivially expandable up to 64-bit transmissions by merely changing the initial LBI instruction. The routines are written using the XIS instruction, but the XDS instruction could be used equally well.

The routines arbitrarily select register 0 as the I/O register. It is assumed that the external device requires a logic low chip select. It is further assumed that chip select is high, SK is low, and SO is low on entry to the routines. The routines exit with chip select high, SK low, and SO low. G0 is arbitrarily chosen as the chip select for the external device.

Destructive Data Output

This routine outputs the data under the conditions specified above. The output data is destroyed after it is transmitted.

```
OUT1:   LBI   0,12    ; point to start of data word
        SC            ; set C to enable SK clock
        OGI   14      ; select external device by 0 -> G0
        LEI   8       ; enable shift register output
SEND:   LD
        XAS           ; data transmission loop, first
        XIS           ; XAS turns on SK clock
        JP    SEND
        RC
        XAS           ; turn off SK clock, transmission done
        OGI   15      ; deselect external device
        LEI   0       ; set SO to 0
        RET
```

Note that this is a general purpose routine and handles all the overhead except loading the data into R0. The routine takes a total of 17 ROM words and can undoubtedly be reduced in specific applications.

Nondestructive Data Output

This routine is identical to the destructive data output routine except that the transmitted data is preserved in the microcontroller.

```
OUT2:   LBI     0,12      ; point to start of data word
        SC
        OGI     14        ; select the external device
        LEI     8         ; enable shift register mode
        JP      SEND2
SEND1:  XAS
SEND2:  LD                ; data output loop
        XIS
        JP      SEND1
        XAS               ; send last data
        RC                ; wait 4 cycles to data to get out
        CLRA
        NOP
        XAS               ; turn SK clock off
        OGI     15        ; deselect the device
        LEI     0         ; turn SO low
        RET
```

The nondestructive routine takes 21 ROM words, four more than the destructive routine. Again, this is a general purpose routine which can probably be reduced in specific applications.

Serial Data Input

The code for reading serial data is almost the same as the write code. This should be expected because of the nature of the SIO register and the XAS instruction.

The first routine enables shift register mode, selects the external device, and reads the data in. Register 0 is the input register and the routine, as written, is for a 16-bit data stream. As before, the routine is trivially expandable up to 64 bits. G0 is arbitrarily selected as the chip select for the external device. SK is 0, and G0 is high or entry to the routine.

```
READ:   LEI     0         ; enable shift register mode, SO is 0
        OGI     14
        SC
        XAS               ; turn on the clock
        LBI     0,13      ; initialize the B register
        NOP               ; NOPs to preserve the timing
LOOP:   NOP
        XAS
        XIS               ; read all but last four bits in this loop
        JP      LOOP
        RC
        XAS               ; turn off the clock and read last four bits
        OGI     15        ; deselect the device
        RET
```

The routine exits with the data in digits 0,13, 0,14, 0,15, and the accumulator.

A variation on this routine which places the input data in digits 0,12 through 0,15 is presented below. This routine uses one subroutine level.

```
READ:   LEI    0       ; enable shift register mode 0->SO
        OGI    14      ; select external device
        LBI    0,12    ; initialize B register
LOOP:   JSRP   SIO
        LD             ; data read loop
        XIS
        JP     LOOP
        OGI    15      ; deselect the devices
        RET
```

The following subroutine is used:

```
SIO:  SC     ; turn on SK clock
      XAS
      RC     ; wait 4 cycles for the data to full SIO and
      NOP    ; turn off the clock
      NOP
      XAS
      X      ; put data to memory
      RET
```

These are two implementations of the same basic routine. The first version reads the data in one continuous stream; the second version reads the data in four-bit groups. The second routine uses a little more code. The choice of routine is entirely governed by the application, the peripheral devices used, and not by the microcontroller.

It is fairly common that the peripheral device must be sent some command or instruction directing it to output some data to the MICROWIRE interface. A typical routine of this type is given below. G0 is again chosen as the chip select. It is assumed that the peripheral device requires a start bit followed by four bits of instruction information. Location 0,0 is arbitrarily selected for storage of the instruction data. The routine is again written for 16 data bits. The input portion of the routine is essentially the same routine as the first version above. There is a subtle difference: the data is all placed in RAM and four extra clocks are generated. This is not normally a problem, but if it is, use another form of the input routine. There is no requirement that the input routine must be in this form:

```
READ:   OGI     14      ; select the device
        LEI     8       ; enable shift register
        CLRA            ; setup start bit in A
        AISC    1
        SC              ; turn on clock and send start bit
        XAS
        LDD     0,0     ; fetch command/instruction
        LBI     0,12    ; initialize B register
        XAS             ; send command/instruction
        NOP             ; wait 4 cycles for data to get to
        CLRA            ; the peripheral
        XAS             ; just maintaining the timing, send 0s
        NOP             ; delay - typical required 0 to 3 instruction cycles
        NOP             ; now wait 4 cycles for data to fill SIO
        NOP
LOOP:   CLRA
        XAS             ; data read loop
        XIS
        JP      LOOP
        RC
        XAS             ; turn off the clock
        OGI     15
        LEI     0       ; deselect the device and turn SO off
        RET
```

5.5.4 SI as a General Purpose Input

When not used as part of the MICROWIRE interface, SI can be used as a general purpose input. There are two ways in which this can be done:

1. Leave SIO in shift register mode. SO may be enabled or disabled depending on system requirements. Then reading SI is simple:

```
        CLRA            ; this clear not absolutely necessary
        XAS
        AISC    15      ; test SIO for 0, if 0 SI=0, else SI=1
        JP      SIEQ0
SIEQ1   .
        .
        .
```

2. Put SIO in counter mode. Then SI will capture pulses that meet minimum width requirement. Load SIO with 0 and test for 15.

   Sample code for this is as follows:

```
        CLRA            ; CLRA required here
        XAS
```

```
            AISC    1
            JP      NOPULSE
PULSE:      .
            .
```

Remember that this mode captures and remembers the occurrences of a high to low transition at SI input. SIO is in binary counter mode for this method to work.

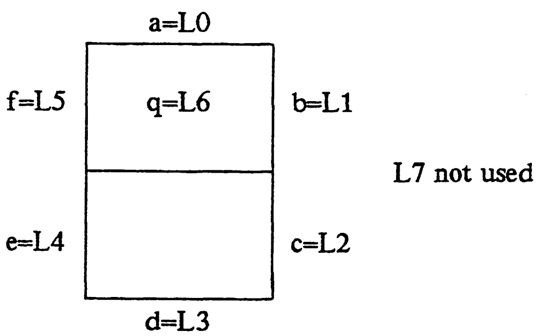
Some devices have the SKSZ instruction. This makes testing SI, or SIO, particularly easy. SKSZ tests the contents of SIO without affecting those contents and generates a skip if SIO is 0. This is essentially the same test as above except that it is a single instruction.

5.6 DISPLAY CONTROL

It is frequently required to control a display as part of an application using COPS microcontrollers. There are several approaches to this and this section will attempt to illustrate those approaches.

5.6.1 A Four-Digit Multiplexed Display

This routine will output a four-digit number to a standard seven segment display. The D lines will be the digit strobes, with D3 being the most significant display digit. The L lines will provide the segment data with the following format:



The interconnect and flow chart are shown in Figures 5-26 and 5-27. The code is written independently and simply displays the data. In a real application, the routine would have to be merged with the main code. The routine provides both segment and digit interdigit blanking. A simple delay routine is used to control display ON/OFF time.

	Bd 15	14	13	12	11
Br 0	DISPLAY MSD			DISPLAY LSD	WORK SPACE

The RAM map for this routine is shown above. The display data is in BCD.

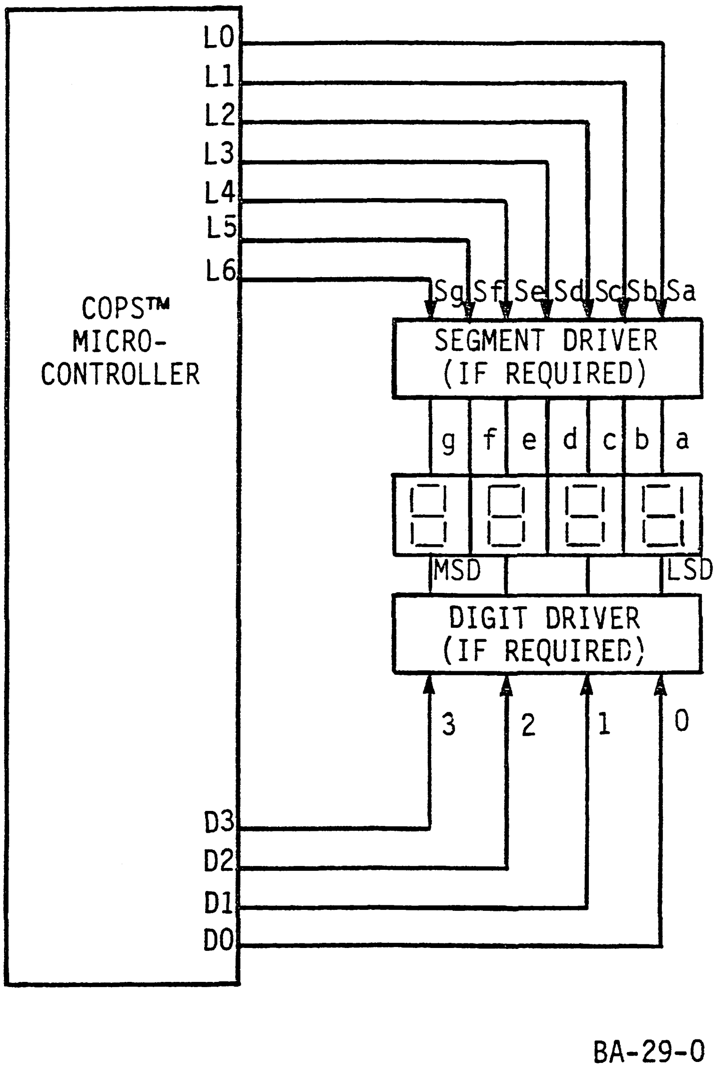
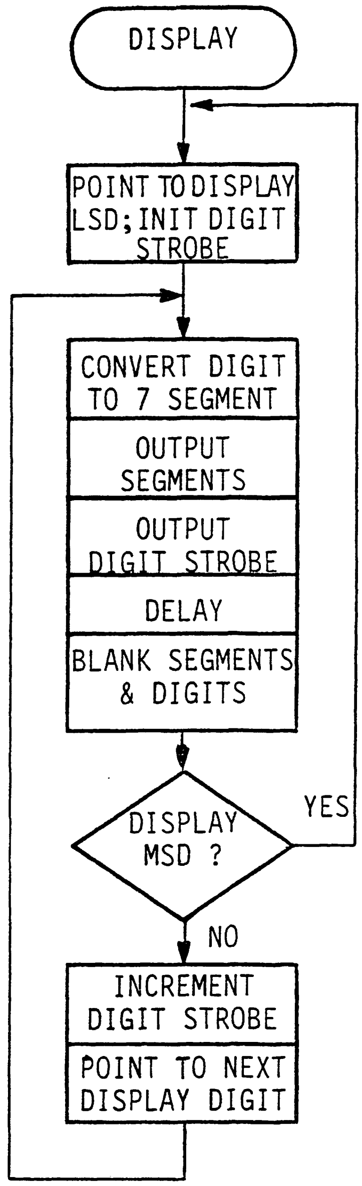


Figure 5-26. Interconnect for Sample and Multiplexed Display Code



BA-30-0

Figure 5-27. Multiplexed Display Flow Chart

```
DISPLAY:  LBI   0,11      ; initialize digit strobe
          STII  1
          JSR   OUT       ; output first digit - LSD
          LBI   0,13
          JSR   OUT       ; second digit
          LBI   0,14
          JSR   OUT       ; third digit
          LBI   0,15
          JSR   OUT       ; fourth digit - MSD
          JP    DISPLAY
```

The subroutine OUT does most of the work:

```
OUT:    CLRA            ; set up address for table
        AISC    4
        LQID
        LDD     0,11    ; output digit strobe
        CAB
        OBD
        LEI     4       ; enable segment outputs
        LBI     0,11
        LD
        ADD             ; shift the strobe to next digit
        X
WAIT:   CLRA            ; delay time arbitrary for display
        SKT             ; on time
        JP      WAIT
        LBI     0,15    ; turn off the digits; all high
        OBD
        LEI     0       ; turn off the segments; L drives off
        RET             ; return for the next digit
```

The preceding routine uses a subroutine level. A routine that performs the same function but does not use a subroutine level is indicated below. As the RAM map indicates, an extra RAM digit is used in this implementation of the multiplexed display routine.

	Bd					
	15	14	13	12	11	10
Br 0	DISPLAY MSD			DISPLAY LSD	DIGIT STROBE	DISPLAY POSI- TION

As before, the data is assumed to be in BCD.

```
DISPLAY:   LBI    0,10       ; initialize display pointer and digit strobe
           STII   12
           STII   1
DSP1:      CLRA
           AISC   4          ; set up address for table
           LQID              ; look up segments
           LDD    0,11       ; output digit strobe
           CAB
           OBD
           LEI    4          ; enable L to output segment data
           LBI    0,11       ; increment digit strobe (left shift)
           LD
           ADD
           X
WAIT:      CLRA              ; delay arbitrary for display ON time
           SKT
           JP     WAIT
           LBI    0,10       ;increment display pointer
           LD
           AISC   1
           JP     DSP2
           JP     DISPLAY    ; have outputted MSD, start over
DSP2:      X
           LD
           CAB
           JP     DSP1
```

This routine is completely equivalent to the preceding routine but does not have a subroutine call. Both routines use the following BCD to seven-segment code conversion table:

```
.=0140            ; set up table location - address
                  ; starts at 140 hex
.WORD    03F      ; 0
.WORD    006      ; 1
.WORD    05B      ; 2
.WORD    04F      ; 3
.WORD    066      ; 4
.WORD    06D      ; 5
.WORD    07D      ; 6
.WORD    007      ; 7
.WORD    07F      ; 8
.WORD    067      ; 9
```

Both routines assume that the L drivers are off and that the digit strobes are high on entry to the routine. Some display types do not require both digit and segment blanking. If this is the case, the routines can be shortened by removing the unnecessary blanking code. Note that the routines do not alter the BCD data. Remember, also, that the LQID instruction uses a subroutine level on some COPS microcontrollers. Also note that the delay time included in the routine may not be necessary for some display types. In these cases, that code may be eliminated. The delay, if required at all, may be implemented in any convenient manner.

5.6.2 Peripheral Display Drivers

Several display drivers are available which are compatible with the COPS MICROWIRE and remove the burden of display control from the microcontroller to an inexpensive driver.

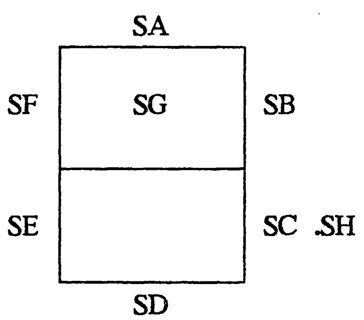
The COP470 and COP472

The COP470 is a four-digit multiplexed vacuum fluorescence display driver. The device is loaded with 32 bits of segment data and controls the display directly. Updating the display merely requires loading the new data. Note that any required code conversion must be performed by the microcontroller.

The COP472 is a similar device intended for use with a multiplexed (three backplane) liquid crystal display. The COP472 is a 4½ digit driver and can drive 36 segments of data. Again, any required code conversion must be done in the microcontroller.

Both the COP470 and COP472 may be cascaded to drive somewhat larger displays. The COP470 and COP472 are software compatible devices. Code can be written that works with either the COP470 or the COP472 either alone or cascaded. The four extra data bits in the COP472 correspond to brightness control in the COP470.

Both the COP470 and COP472 load data eight bits at a time. The format for the data is as follows:



SA	SB	SC	SD	SE	SF	SG	SH

SH for digit 1 is the first data bit shifted into the device. SA for digit four is the last data bit (*i.e.*, 32nd data bit) shifted into the device. The segments are mapped into a standard numeric seven-segment plus decimal point display. There is, of course, no requirement that the display be configured in this manner.

The fifth and final group of eight bits sent to the device(s) is as follows:

C4	C3	C2	C1	SP4	SP3	SP2	SP1

SP1 is the first data bit sent in this group, C4 is the last bit sent.

The COP470 and COP472 display drivers may be "cascaded" to provide more digits and "stacked" to provide more segments per digit. Both the COP472 and COP470 are code compatible devices even when they are used in expanded form.

Single COP470, COP472 Control Bits:

The control bits for the COP470 and COP472 are listed below in Table 5-1. These control bits were positioned to allow for common software operations.

The COP470 also contains four bits of intensity information which is in the same bit locations corresponding to the four special segments of the COP472. In code compatible routines, the four special segments of the LCD display will reflect the intensity information of the COP470. The control bits that enable code compatible operation with four-digit displays are given in Table 5-1.

TABLE 5-1. CONTROL BITS

	C4	C3	C2	C1	SP4	SP3	SP2	SP1
COP470 CONTROL:	SYNC	OSC	RT	LT	INTENSITY INFORMATION			
COP472 CONTROL:	SYNC	Q7	Q6	X	SPECIAL SEGMENTS			

X = DON'T CARE

COP472	COP470	CONTROL BIT
SYNC	SYNC	0
Q7	OSC	0
Q6	RT	1
X	LT	1

BA-36-0

Eight Digit

COP470 and COP472 devices are cascadable to obtain more digits of display. The control codes for a multiple device display driver configuration are listed in Table 5-2.

TABLE 5-2. CONTROL CODES

		CONTROL CODES		
COP472	COP470	INITIALIZE (BOTH DEVICES)	MASTER (LEFT DEVICE)	SLAVE (RIGHT DEVICE)
Sync	Sync	1	0	0
07	Osc	1	0	1
06	RT	1	0	1
X	LF	0	1	0
			X = Don't Care	

The sequence of operations to load a single COP470 or COP472 is as follows:

1. Turn $\overline{CS}$ low.
2. Clock in eight bits of data for digit 1.
3. Clock in eight bits of data for digit 2.
4. Clock in eight bits of data for digit 3.
5. Clock in eight bits of data for digit 4.
6. Clock in eight bits of data for special segments/brightness and the control function.
   0 0 1 1 SP4 SP3 SP2 SP1
7. Turn $\overline{CS}$ high.

$\overline{CS}$ may be turned high after any step. It is not necessary to continuously reload the control bits but they must be loaded at least once. If the special segments or brightness bits are changed, the control bits must be reloaded.

$\overline{CS}$ must toggle between writes. $\overline{CS}$ is the state that resets the internal counters in the device which controls data loading.

Typical code to write to a single COP470 or COP472 is shown below. The look-up table is not shown but is obviously required. The routine is written as in-line code. It does the code conversion and writes to the display driver. The original values are destroyed in the operation. DO is arbitrarily chosen as a chip select for the device. Note that chip select is an essential connection for these devices. Chip select must toggle between accesses for proper operation. The data to be displayed is in locations 0,12 through 0,15. The special segments or brightness bits are in location 0,0.

```
DISPLAY:  LBI    0,12   ; point to first display data
          OBD           ; turn CS low (DO) to select drive
LOOP:     CLRA
          LQID          ; look up segment data
          CQMA          ; copy data from Q to M & A
          SC            ; set C to turn on SK
          XAS           ; output lower four bits of data
          NOP           ; delay
          NOP           ; delay
          LD            ; load A with upper four bits
          XAS           ; output four bits of data
          NOP           ; delay
          NOP           ; delay
          RC            ; reset C
          XAS           ; turn off SK clock
          XIS           ; increment B for next data
          JP     LOOP   ; skip this jump after last digit
          SC            ; set C
          LBI    0,0    ; address special segments or brightness
          LD            ; load into A
          XAS           ; output special segments or brightness
          NOP
          CLRA
          AISC   12     ; 12 to A=code for single chip operation
          XAS           ; output control bits
          NOP
          LBI    0,15   ; 15 to B to deselect the device
          RC             reset C
          XAS           ; turn off SK
          OBD           ; turn CS high (DO)
          .
          .
          .
```

This code works with either the COP470 or COP472.

The sequence to drive two COP470s or COP472s in an eight-digit display is outlined below. There is an initialization procedure required in order to set up the two devices properly. The control bits are different during the initialization sequence than they are during subsequent data loads. For the COP472s, this sequence sets up the left chip as the master and the right chip as the slave. For the COP470s, the left chip provides the oscillator for the right chip. The sequence is as follows:

1. Turn $\overline{CS}$ low to both devices.
2. Shift in 32 bits of data - slave's four digits for COP472, right four digits for COP470.
3. Shift in four bits of special segment/brightness data, a zero and three ones.

1	1	1	0	SP4	SSP3	SP2	SP1

This synchronizes and stops both chips. Both chips are expecting an external oscillator.

4. Turn $\overline{CS}$ high to both chips.
5. Turn $\overline{CS}$ low to left device - (master COP472, left COP470).
6. Shift in 32 bits of data for that device.
7. Shift in four bits of special segment/brightness data, a one and three zeroes.

0	0	0	1	SP4	SSP3	SP2	SP1

This sets this device to internal oscillator and provides an oscillator output to the other device.

8. Turn $\overline{CS}$ high.

The chips are now synchronized and driving eight digits of display. New data is loaded in the normal manner. Care must be taken to keep the control bits in the proper state. For the master COP472 or left COP470, the control bits specified in Step 7 are the proper state. For the slave COP472 or right COP470, the following information must be sent in every case except the initialization sequence:

0	1	1	0	SP4	SSP3	SP2	SP1

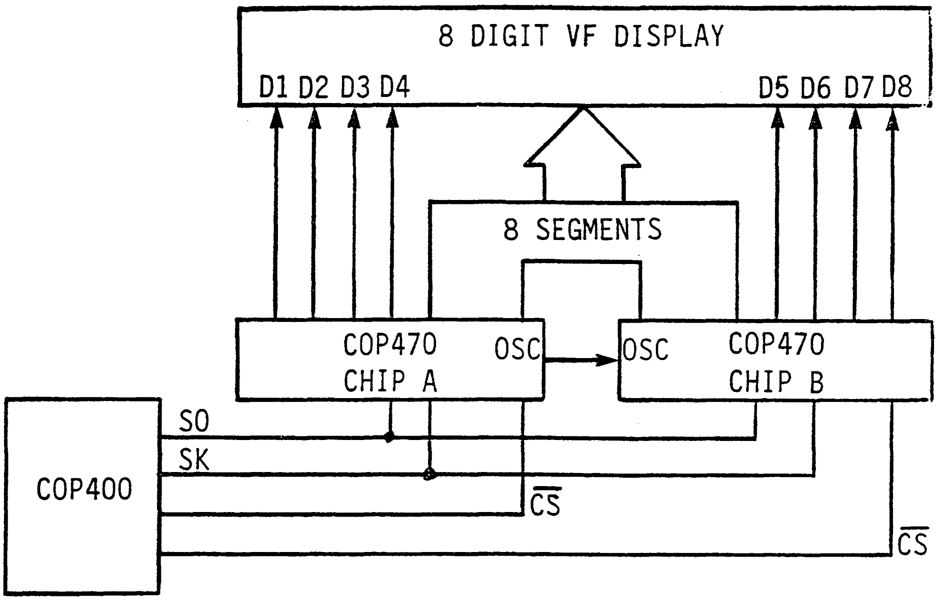
Figure 5-28 provides system diagrams for the dual COP470/COP472 systems.

Typical code to write to the devices in this way is shown below. The display data for the slave (right) device is in register 0, digits 12 through 15. The display data for the master (left) device is in register 1, digits 12 through 15. Digit 0,0 contains special segment/brightness data for the slave. Digit 1,0 contains special segment/brightness data for the master. DO is used as the chip select for the master; D1 is the chip select for the slave. The code is again shown as in-line code.

Display Initialization Sequence:

```
INIT:  LBI   0,15
       OBD          ; turn both CSs high
       LEI   8      ; enable SO out of S.R.
       RC
       XAS          ; turn off SK clock
       LBI   3,15   ; use M(3,15) for control bits
       STII  7      ; store 7 to sync both chips
       LBI   0,12   ; set B to turn both CSs low
       JSR   OUT    ; call output subroutine
```

Main Display Sequence:



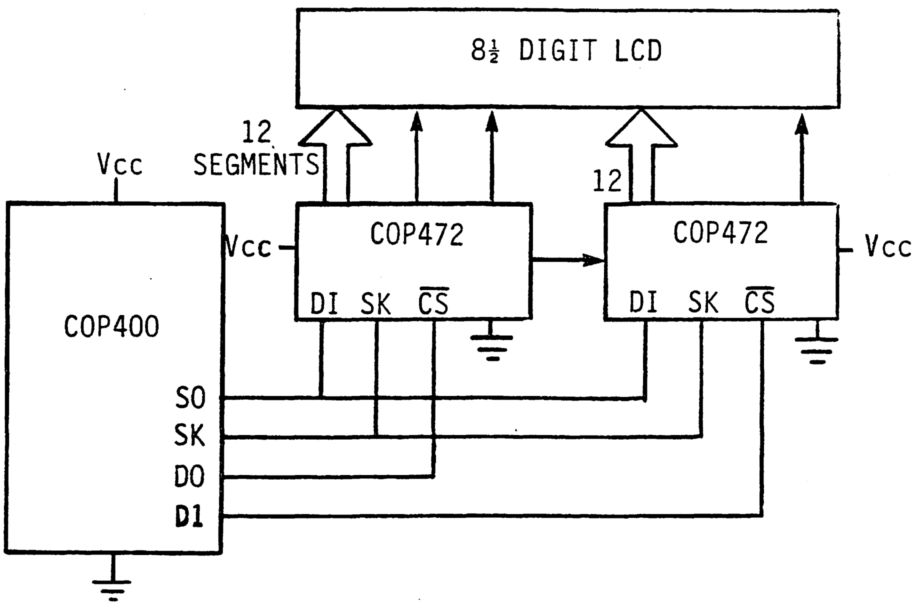


Figure 5-28. Dual COP470/472 Systems

```
DISPLAY:  LBI   3,15
          STII  8       ; set control bits for slave right devices
          LBI   0,13    ; set B to turn slave CS low
          JSR   OUT     ; output data from register 0
          LBI   3,15
          STII  6       ; set control bits for master left device
          LBI   1,14    ; set B to turn master CS low
          JSR   OUT     ; output data from register 1
```

Output Subroutine:

```
OUT:    OBD             ; output B to CSs
        CLRA
        AISC    12      ; 12 to A
        CAB             ; point to display digit (BD=12)
LOOP:   CLRA
        LQID            ; look up segment data
        CQMA            ; copy data from Q to M & A
        SC
        XAS             ; output lower four bits of data
        NOP             ; delay
        NOP             ; delay
        LD              ; load A with upper four bits
        XAS             ; output four bits of data
        NOP             ; delay
        NOP             ; delay
        RC              ; reset C
        XAS             ; turn off SK
        XIS             ; increment B for next display digit
        JP      LOOP    ; skip this jump after last digit
        SC              ; set C
        NOP
        LD              ; load special segments
        XAS             ; output special segments
        NOP
        LBI     3,15
        LD              ; load A
        XAS             ; output control bits
        NOP
        NOP
        RC
        XAS             ; turn off SK
        OBD             ; turn CSs high (BD=15)
        RET
```

The MM54XX Series Display Drivers

The MM54XX series drives are a family of status display drivers for vacuum fluorescent, liquid crystal, and LFD displays. All of these devices require a start bit and 35 data bits. All the devices are MICROWIRE compatible. Table 5-3 indicates the present devices that comprise the MM54XX series. The code here is applicable to all similar type devices. The MM54XX devices are static segment drivers and must be loaded with the appropriate segment information.

TABLE 5-3. MM54XX SERIES DEVICES

MM5445 - Static Vacuum Fluorescent
MM5446 - Static Vacuum Fluorescent
MM5447 - Static Vacuum Fluorescent
MM5448 - Static Vacuum Fluorescent
MM5450 - Static LED
MM5451 - Static LED
MM5452- Static Liquid Crystal
MM5453 - Static Liquid Crystal
MM5480 - Static LED (Smaller Package)
MM5481 - Static LED (Smaller Package)

Two basic output techniques can be used. The first approach is the same as that illustrated for the COP470 and COP472: turn the clock on and off and convert the number on the fly. This example will use G0 as the data enable control: G0 must go low to enable the device. The routine assumes G0 high, SO low, and SK low on entry. The look-up table is not shown.

```
DISPLAY:    CLRA                ; set up start bit
            AISC    1
            SC
            OGI     14          ; select the device
            XAS                 ; turn on clock and send start bit
            RC
            CLRA
            NOP
            XAS                 ; turn off the clock
            LBI     0,7         ; point to start of data
LOOP:       CLRA                ; set up table address
            LQID
            CQMA
            SC                  ; send eight data bits
            XAS
            NOP
            NOP
            LD
            XAS
            NOP
            CLRA
            RC
            XAS
            LD
            XIS
            JP      LOOP
            OGI     15          ; deselect the device
            LEI     0           ; turn SO low
            RET
```

The other approach is to load a display buffer with the segment data and then simply send all the information out in one burst of data. This technique can also be used with the COP470 and COP472. The following routine implements this procedure. Again, the table is not shown, and G0 is the data enable. The display output is the BCD number contained in locations 2,12 through 2,15. Register 0 will be used as the display output register. The segmented data will be placed in digits 0,7 through 0,15. Digit 0,15 will be loaded with 0s to fill out the required 35 data bits. The code is as follows:

```
DISPLAY:  LBI    2,12      ; convert data to segment information
          CLRA             ; set up table address
          LQID
          LBI    0,7       ; save segments in register 0
          JSRP   INQ
          LBI    2,13
          LQID
          LBI    0,9
          JSRP   INQ
          LBI    2,14
          LQID
          LBI    0,11
          JSRP   INQ
          LBI    2,15
          LQID
          LBI    0,13
          JSRP   INQ
          STII   0         ; load 0s to 0,15
          LBI    0,7       ; point to first segment data
          SC               ; set C to turn on clock
          AISC   1         ; set up start bit
          LEI    8         ; enable shift register output
          XAS              ; send start bit
          JSR    DATOUT
          RET
```

The following subroutines are used:

```
INQ:  CQMA     DATOUT:  LD
      XIS               XAS
      XIS               XIS
      CLRA              JP    DATOUT
      RET               RC              ; turn off clock
                        XAS
                        RET
```

Universal Display Loading Routine

Theory of Operation

The universal display driver loading routine both initializes and sends 32 data bits to the display drivers. In those devices with more than 32 data bits, the extra segments are not used. The routine is compatible with the COP470, COP472, and MM54XX series devices.

Associated with the COP470/COP472 and MM54XX series are two communication protocols. The COP470 and the COP472 accept data in blocks of eight bits and require an initialization procedure. The MM54XX series requires a start bit and a block of 35 bits before data is latched in the output buffers. There exists a common block of 32 data bits between all these devices (less are bonded out on the MM5480 and MM5481) and this similarity makes it possible to create universal display load routine. The control bits for the COP470 and the COP472 are sent once upon initialization, and the start bit for the MM54XX series is sent on the tail end of the data load routine every time it is called.

The COP470 and COP472 have a chip select which, upon a high to low transition, clears the input register and the internal counters which route the data and control bits to their ultimate positions. (See COP470, COP472 block diagrams.) Each of these devices accepts a serial data pattern and latches that serial stream in blocks of eight. For example, once initialized, the first digit may be changed, without affecting the other digits, by chip selecting and sending eight data bits. Data streams of less than eight bits, between chip selects or after a block of eight bits has been accepted, will be ignored. The initialization routine for the COP470 and COP472, which sends 44 bits, makes use of this type of operation; the last four bits are ignored.

The MM54XX series displays, unlike the COP470 and COP472, have a data enable. This input to the device does not reset any counter and functions only as a data enable. This is to say that information contained within the display buffers and the input counter are not affected by the data enable signal. It is for this reason that the start bit for MM54XX series devices is sent out at the tail end of each data output routine. Initially, the MM54XX devices must be cleared and this is accomplished by clocking in more than 35 zeroes. In normal operation, the MM54XX type devices are automatically cleared at power up due to SIO port power up state; SK as clock and SO as a logical zero, lasting much more than 36 cycles. In the universal display routine, the MM54XX series devices will contain the COP470 and COP472 control codes along with a start bit in the first position. This must be cleared out by sending 35 zeroes and a new start bit. This will clock in 32 zeroes to the COP470 and COP472, and again the last four bits will be ignored in the COPS display drivers.

Now both display device types are initialized and data may be sent out in 36 bit blocks, first 32 data, next three zeroes, and the last bit a start bit. The first 32 segment outputs of the COP472 and MM54XX series devices will correspond to the COP470's segment outputs.

5.7 KEYBOARD SCAN

Reading a keyboard is a common requirement. The following routine is representative of a keyboard scan routine. The four D lines provide the strobes for the keyboard. The IN lines are the keyboard return lines. Thus, this routine is structured to read a 16-key keyboard arranged in a 4 by 4 matrix. A key is detected when one of the IN lines goes low. The strobes, D lines, are normally high and go low to strobe the keyboard. Figure 5-29 is the flow chart for this routine. Figure 5-30 is the interconnect. This routine uses two RAM digits: digit 0,15 for a debounce counter and digit 0,14 for temporary storage. The routine debounces the keys up and down.

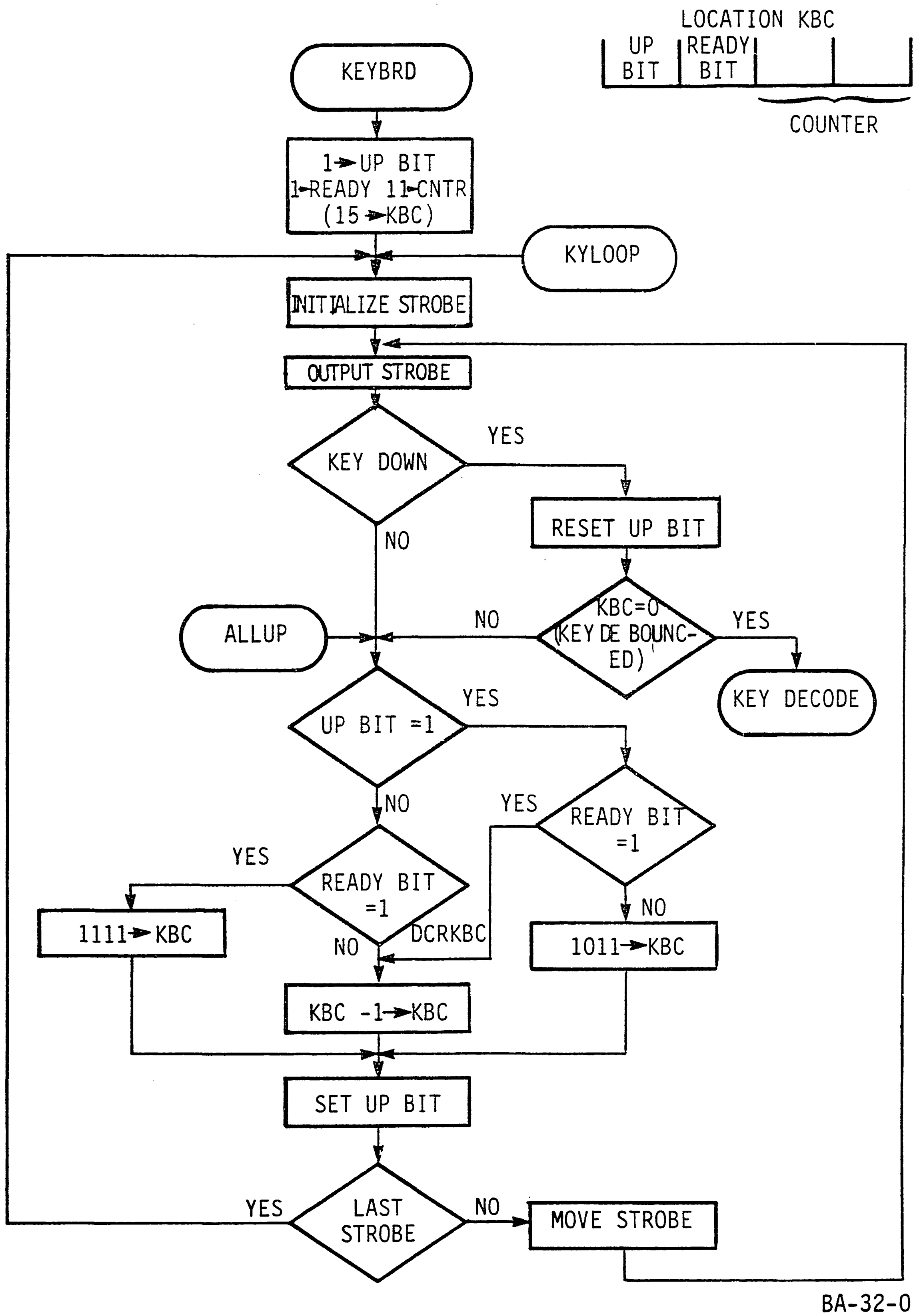


Figure 5-29. Keyboard Scan Flow Chart

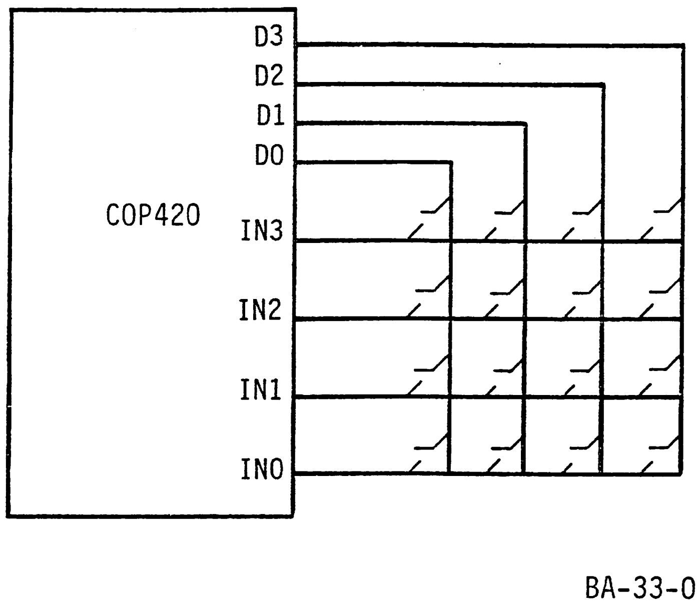


Figure 5-30. Interconnect for Key Scan Routine

```
          KBC   = 0,15
          KEYIN = 0,14

KEYBRD:   LBI       KBC       ; initialize debounce counter
          STII      15
KYLOOP:   LBI       1,14      ; set DO low, see if a key is down
          JSRP      SCAN
          JP        KEY0      ; key is down
          LBI       0,13      ; set D1 low and see if a key is down
          JSRP      SCAN
          JP        KEY1
          LBI       0,11      1 set D2 low and see if a key is down
          JSRP      SCAN
          JP        KEY2
          LBI       0,7       ; set D3 low and see if a key is down
          JSRP      SCAN
          JP        KEY3      ;  if the routine falls through to this point
                              ; there is no key down on this scan,
                              ; or key not fully debounced
NOKEY:    LBI       KBC
          CBA                 ; put 15 to A
DBNCE:    SKMBZ     3         ; test up bit = 1
          JP        ALLUP     ; yes
          SKMBZ     2         ; up bit = 0, test ready bit
          JP        STR       ; 15 -> KBC else decrement KBC
DCRKBC:   ADD                 ; remember A=15, so decrement KBC
STR:      X                   ; A -> KBC
          SMB       3         ; set up bit
          JP        KYLOOP
ALLUP:    SKMBZ     2         ; if ready bit=1, decrement KBC
          JP        DCRKBC
          STII      11        ; else, load KBC with 11
          JP        KYLOOP

KEY3:     These are the key decode positions; location KEYIN
KEY2:     contains IN line data; entry point defines strobe
KEY1:     line.  The key is fully debounced if reach any of
KEY0:     these points.
```

Appendix A

DATA RAM IN COP410L/411L/413L AND COP410C/411C DEVICES

A.1 DATA RAM DESCRIPTION

All COPS microcontrollers except the COP410L, COP411L, COP410C, and COP411C have the data RAM matrix organized as a number of registers by 16 digits. The COP410C series devices mentioned above have the data RAM organized as 4 registers by 8 digits. This is significant because the Bd portion of the RAM address register B is still four bits wide. The D output port is still a four-bit port and it is loaded by Bd as in all COPS microcontrollers.

Physically, only the lower three bits of Bd address the digit portion of RAM. The upper bit is not connected to the RAM in any way. However, the XIS and XDS instructions work on the entire Bd register. The skip conditions on these instructions is the same as always. Bd will increment from 0 to 15. Thus each RAM digit in a COP410 series device is addressed by two values of Bd. Because of this characteristic, the programmer must exercise some care in the implementation of any routine which increments or decrements through the register, *e.g.*, shift routines. The standard digit shift routines provided earlier could actually shift a COP410 register right or left two digits if the programmer started at one end of the register and relied on the XIS or XDS skip to exit the routine. The two shift routines provided below provide one method of circumventing the problem.

```
         LBI   0,9                LBI   0,9
         LD                       LD
         XDS                      XDS
         LD                       LD
LSHIFT:  XIS             RSHIFT   XDS
         JP    LSHIFT             JP    RSHIFT
         RET                      RET
```

As written, these routines will shift register 0 left or right one digit. Figure A-1 below illustrates the RAM mapping in COP410 series devices.

The following is the key scan subroutine:

```
SCAN:   OBD                     ; output key strobe
        ININ                    ; read the return lines
        LBI     KEYIN
        COMP
        X                       ; store key information
        LD                      ; test if a key is down
        AISC    15
        RETSK                   ; no key, return and skip
        CLRA                    ; a key is down
        LBI     KBC
        RMB     3               ; reset key up bit
        SKE                     ; if KBC is 0, key is fully debounced
        RETSK                   ; not debounced yet
        OBD                     ; key fully debounced, turn the strobes high
        LBI     KEYIN           ; set up pointing to KEYIN for key decode
        RET
```

This is a simple keyboard routine. It is a variation on the routine provided in Section 5.3 of the *COPS Family User's Guide.* The routine continues to scan until a key is detected and fully debounced.

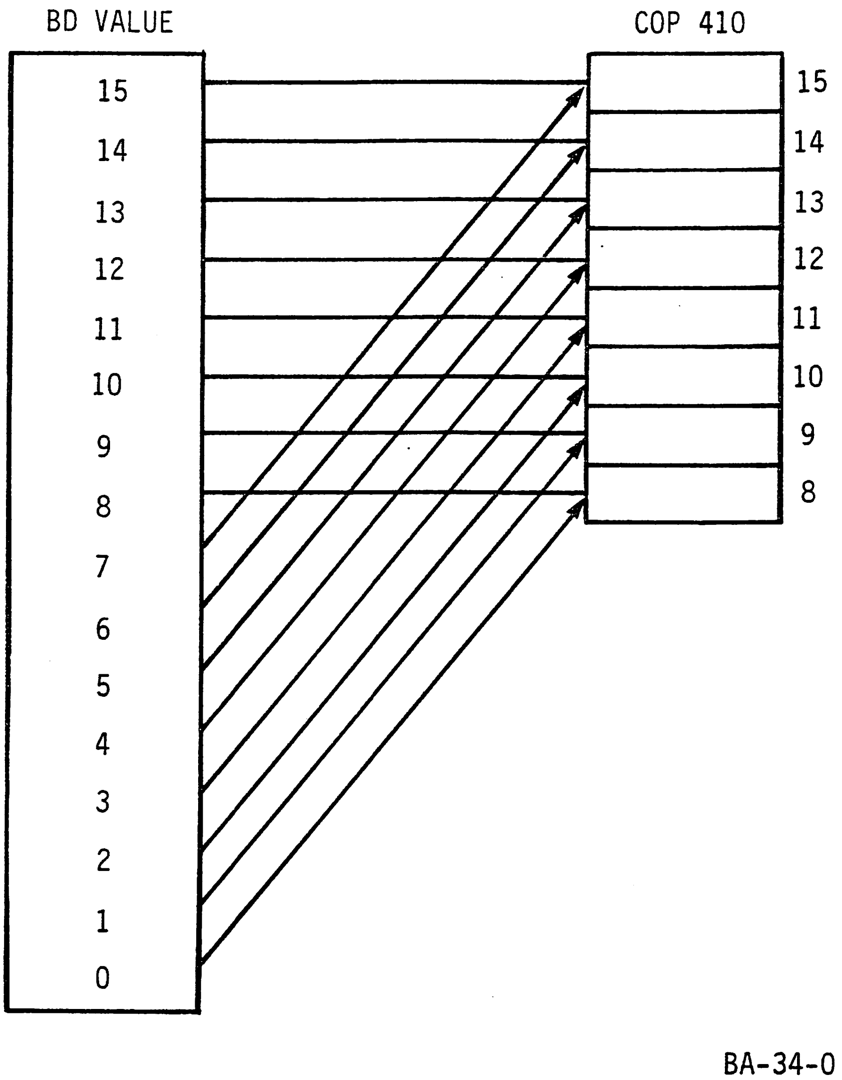


Figure A-1. RAM Mapping

Appendix B

DEVICES WITH SUBROUTINE STACK IN RAM

B.1 SUBROUTINE STACK IN RAM DESCRIPTION AND LOCATION

As mentioned earlier, a number of COPS microcontrollers have the subroutine stack in data RAM. In these devices the stack is assigned a specific location and does not, under any circumstances, go outside of the assigned area. It is not possible for the programmer to overflow the stack and destroy some data, although it is quite possible to overflow the stack. The only information lost if the stack overflows is some previous return address. The devices which have the stack in RAM and the location of the stack in the RAM is indicated below.

DEVICE	LOCATION
COP440/441/442 COP404	Stack in register 8
COP2440/2441/2442 COP2404	CPU X stack in register 8, CPU Y stack in register 9
COP484/485 COP408	Stack in register 15
COP409	Stack in registers 30 and 31

Note that the registers are numbered starting at 0. The register number is the Br address.

Figure B-1 is the structure for the stack in RAM. This organization is valid for all the devices with the subroutine stack in data RAM.

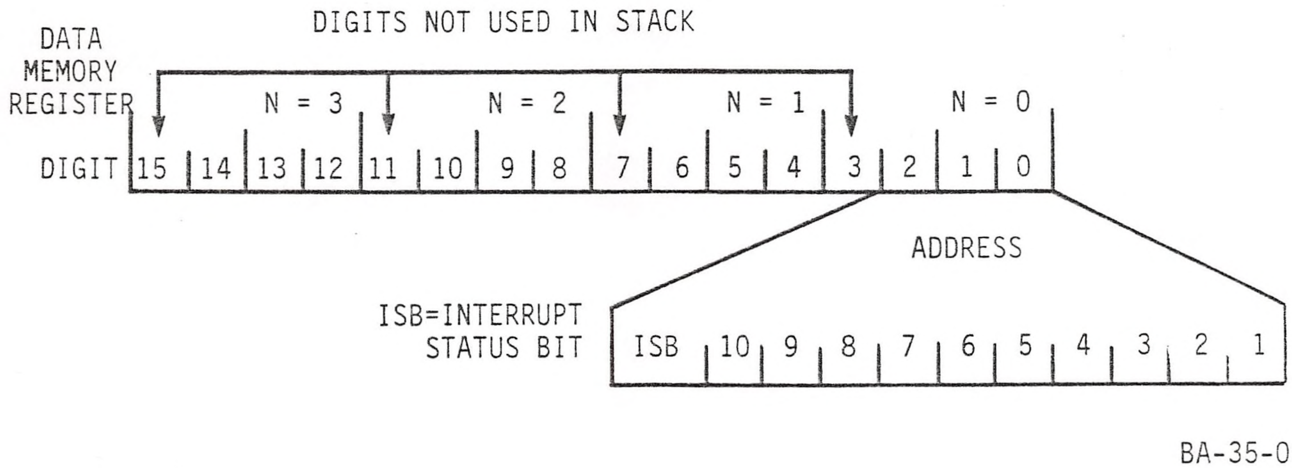


Figure B-1. Stack Structure in RAM

National
Semiconductor

MICROCOMPUTER
SYSTEMS DIVISION

READER'S COMMENT FORM

In the interest of improving our documentation, National Semiconductor invites your comments on this manual.

Please restrict your comments to the documentation. Technical Support may be contacted at:

(800) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(408) 733-2600

Please rate this document according to the following categories. Include your comments below.

	EXCELLENT	GOOD	ADEQUATE	FAIR	POOR
Readability (style)	☐	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐	☐
Fulfills Needs	☐	☐	☐	☐	☐
Organization	☐	☐	☐	☐	☐
Presentation (format)	☐	☐	☐	☐	☐
Depth of Coverage	☐	☐	☐	☐	☐
Overall Quality	☐	☐	☐	☐	☐

NAME ____________________ DATE __________

TITLE ____________________

COMPANY NAME/DEPARTMENT ____________________

ADDRESS ____________________

CITY ____________________ STATE __________ ZIP __________

Do you require a response? ☐ Yes ☐ No PHONE ____________________

Comments:

FOLD, STAPLE, AND MAIL

424410284-001A