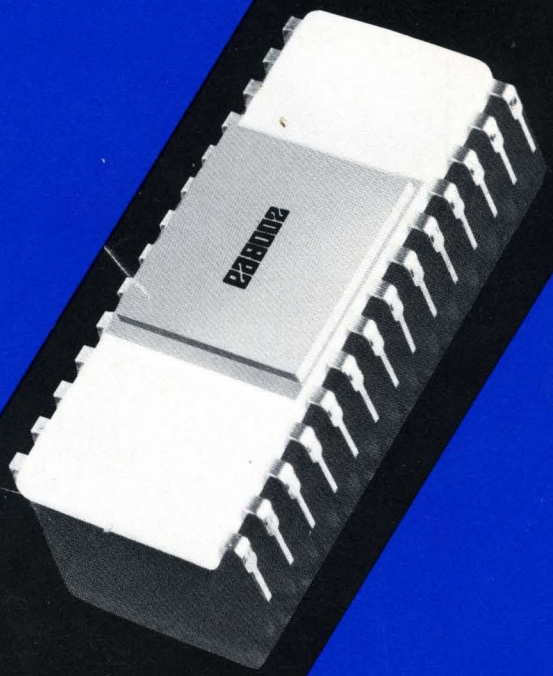
EA9002

USERS HANDBOOK



electronic arrays, inc.

EA9002 USERS HANDBOOK

PRICE $7.50

USERS HANDBOOK
EA 9002 MICROPROCESSOR

Material Contributed By

Kern Moulton
Adam Osborne
Gary Prosenko
Richard Umstattd
W.E. Wickes

Published by

electronic
arrays, inc.

550 East Middlefield Road
Mountain View, California, 94043

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

QUICK INDEX

QUICK INDEX (Continued)

Chapter 1
INTRODUCTION

The EA9002 is a single chip microprocessor designed with the user in mind. That is, it is a fast 8-bit parallel microprocessor unit containing a powerful but easy to understand instruction set; its architecture is well suited to logic replacement applications.

The EA9002 is a data processor — all digital logic circuits can be defined as data processors — but it is not specifically designed for large computer-type memory dominant applications. The term "large" is used in the sense of large mainframe memory systems. Rather, it has been designed to service the newly emerging market of "smart" instruments, data terminals, process controllers, real time instrumentation, credit verification terminals, electronic scales, electronic games; i.e., those products requiring real time solutions to real time problems.

The EA9002 is implemented with NMOS silicon gate depletion mode technology. This is the most advanced semiconductor processing technology available. This MOS process technology has been tested and proven reliable, as demonstrated by the millions of dynamic and static memory components, delivered by numerous semiconductor manufacturers. Utilization of this technology in the design and development of the EA9002 has resulted in one of the highest speed MPU's available — containing more circuitry and data processing capability than was considered possible even a few years ago.

Most microprocessor design and development emphasis in the past has been to emulate "computer architects" — to provide complex addressing modes, confusing architecture and sophisticated control signals. This is not to say that addressing, architecture and control are not significant, rather, it is to point out that computer designs have not had any category of applications in mind; the designs attempted to be all things to all applications, incorporating features required by the many complex disciplines without finite applications objectives. The result has been a rash of microprocessor type circuits which are in reality a montage of all prior CPU theory, rather than coordinated efforts striving toward any identifiable objective.

EA recognized a problem: the lack of microprocessor products designed specifically with the digital logic controller user in mind; products which incorporate the power of parallel microprocessing architecture, fixed instruction set and advanced semiconductor processing technologies in such a way that they could be easily understood and readily applied. It is toward this objective that the EA9002 was directed.

WHAT THIS BOOK CONTAINS

The results of EA9002 development are embodied in this handbook. The handbook is organized much like the circuit itself; i.e., it is written for the engineers — allowing you to quickly understand the product and start writing code for your unique application.

Chapter 2 deals with overall features, internal structure and control signals. It highlights the register oriented architecture, internal bus, status controls and significant CPU features such as the internal push-pop stack and internal 64 byte RAM.

Chapter 3 describes instruction cycle timing and information that appears on the address, data and control bus.

Chapter 4 covers systems level configurations; i.e., how to connect with various types of RAM, ROM and I/O devices. This chapter also discusses memory bank select techniques.

Chapter 5 describes each instruction in the EA9002. The utility of the powerful instruction set is clearly explained, with examples. EA9002 instructions have been carefully chosen to provide complete data manipulation capability along with unambiguous problem definition. Note the simplicity of program development offered by packed BCD add and subtract.

Chapter 6 describes program development techniques, and instruction sequences for common problems. The user unfamiliar with microprocessors and programming skills will find this chapter particularly useful.

Chapter 7 specifically deals with input and output programming. The EA9002 has a uniquely fast I/O capability; it can readily interface with many programmable devices available from other semiconductor manufacturers. This generalized approach to I/O control and data transfer is an important feature of the EA9002.

Chapter 8 presents examples of many common subroutines required in simple types of control and instrumentation systems.

The Appendix summarizes the instruction set and provides useful conversion tables.

HOW THIS BOOK HAS BEEN PRINTED

Notice that text in this book has been printed in boldface type and lightface type. **This has been done to help you skip those parts of the book that cover subject matter with which you are familiar. You can be sure that lightface type only expands on information presented in the previous boldface type.** Therefore, only read boldface type until you reach a subject about which you want to know more, at which point start reading the lightface type.

Chapter 2
AN EA9002 MICROPROCESSOR OVERVIEW

LOGIC DISTRIBUTION

The distribution of logic among various chips within a microcomputer system is not at all well defined. Early microprocessor designers simply assumed that traditional minicomputer logic distributions would also serve microcomputers. This viewpoint has been rapidly eroded by the realities of the emerging microcomputer industry. Therefore, when looking at a new microprocessor, your first step must be to determine the logic which is included within the microprocessor chip itself — as against the logic which must be provided externally.

Figure 2-1 illustrates the logic blocks that typically make up an entire microcomputer system. Logic that is implemented on the EA9002 CPU chip is shaded. In order to provide an immediate contrast, Figure 2-2 illustrates the logic which is implemented on the 8080 CPU — a widely used product manufactured by a number of semiconductor companies.

MICROPROCESSOR ARCHITECTURE

Figure 2-3 is a schematic representation of the logic which is actually implemented on the EA9002 microprocessor chip.

THE ACCUMULATOR

ACCUMULATOR

As in most microcomputers, the center of all data operations is the accumulator. Within the EA9002, this is an 8-bit general purpose register which is the primary source and destination for most CPU operations.

SCRATCH MEMORY

There is also a 64-byte read/write memory implemented within the 9002 CPU chip; in many applications this is all the read/write memory that will be needed.

The 9002 microprocessor can also access external read/write memory and read only memory.

The scratch memory and external memory occupy separate and distinct address spaces, and are accessed by different instructions.

SCRATCH MEMORY ACCESS

Scratch memory can only be accessed as data memory; programs cannot be executed out of scratch memory. Data may be loaded into the Accumulator from a scratch memory byte, or the

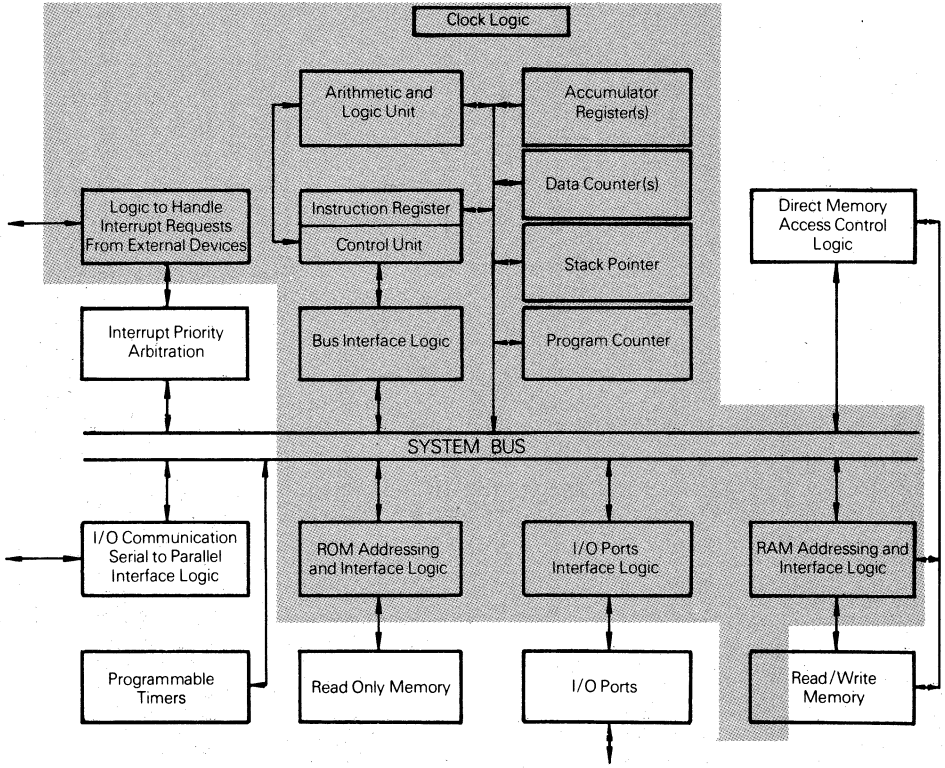


Figure 2-1 EA9002 Logic Functions

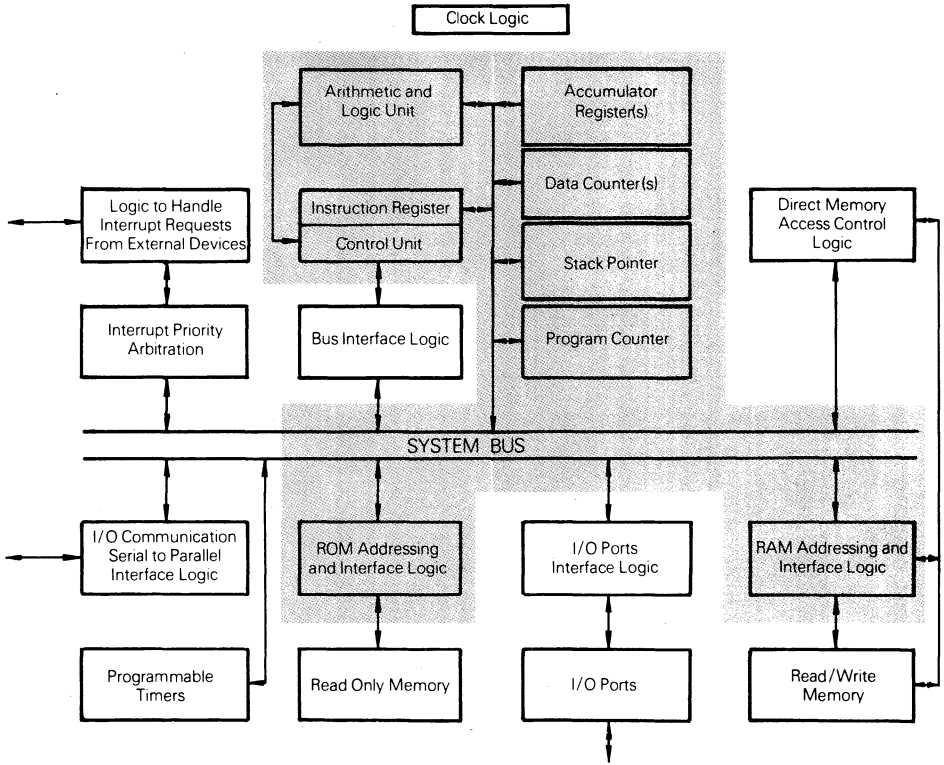


Figure 2-2 INTEL 8080 Logic Functions

Accumulator contents may be written into a scratch memory byte. The contents of a scratch memory byte may also be added to, or subtracted from the Accumulator contents.

SCRATCH MEMORY OPERATIONS

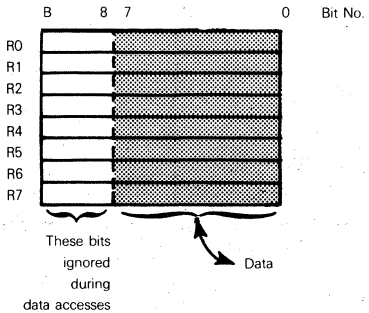
GENERAL PURPOSE REGISTERS AND DATA MEMORY ADDRESSING

GENERAL PURPOSE REGISTERS

The eight general purpose registers, identified in Figure 2-3 as R0 through R7, have no direct equivalent in any other microprocessor.

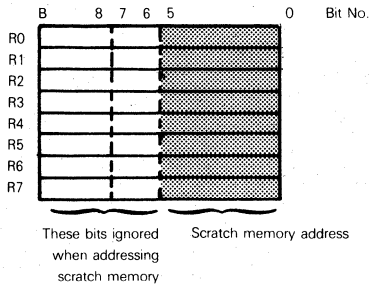
8-BIT DATA REGISTERS

Selected 9002 instructions access the eight low order bits of any general purpose register to perform a variety of data transfer or data manipulation operations:



SCRATCH MEMORY ADDRESSING

Instructions that access the scratch memory specify the low-order six bits of one general purpose register as providing the required scratch memory byte address:



EXTERNAL MEMORY ADDRESSING

External memory may be accessed indiscriminately as program memory, data memory or external logic (I/O devices).

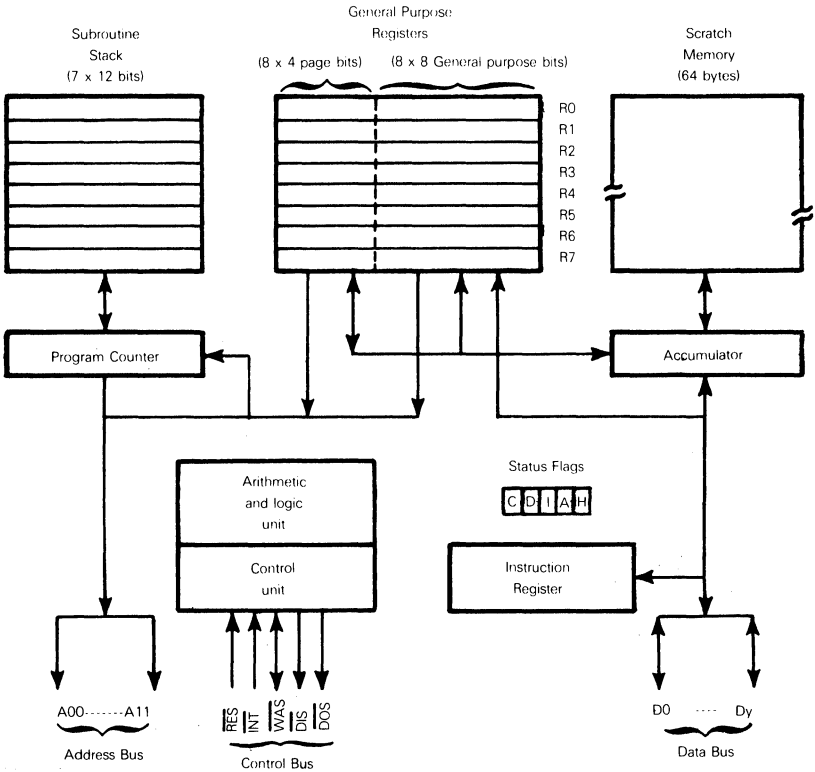


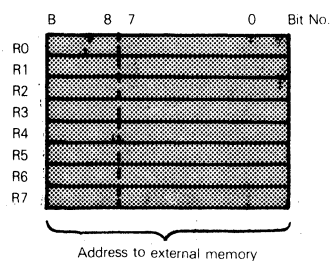
Figure 2-3 EA9002 Device Logic

INPUT/OUTPUT

Instructions make no distinction between data memory and I/O devices; that is to say there are no separate I/O instructions.

REGISTER INDIRECT ADDRESSING

The 9002 microprocessor can read a data byte from external memory (or I/O), or can write a data byte to external memory (or I/O). In either case **external memory or the selected I/O device is identified using register indirect, also called** implied memory addressing — with any one of the eight general purpose registers providing the 12-bit memory address:

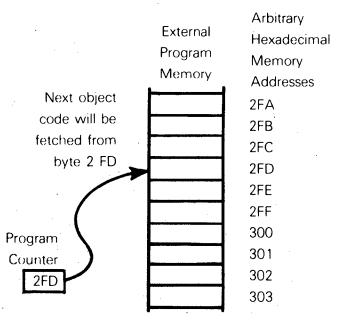


When the contents of a general purpose register is incremented or decremented, the register is treated as a 12-bit unit; this means that an entire memory address is either incremented or decremented.

12-BIT DATA REGISTERS

THE PROGRAM COUNTER AND PROGRAM MEMORY ADDRESSING

The Program Counter is a 12-bit register. At all times the Program Counter addresses the memory location out of which the next instruction code will be fetched. This must be a byte of external memory; instruction codes cannot be stored in the scratchpad memory:



Whenever an instruction is executed, the Program Counter content is incremented to address the next sequential program memory location. This sequential access of program memory may be altered using Jump instructions.

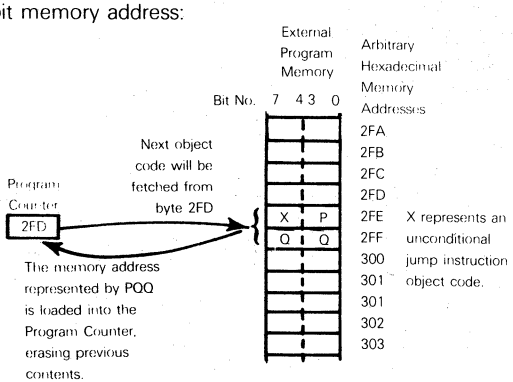
The 9002 has unconditional and conditional Jump instructions, as well as a Jump-to-subroutine instruction.

JUMP INSTRUCTIONS

Unconditional Jump instructions use direct addressing or register indirect (imp) addressing.

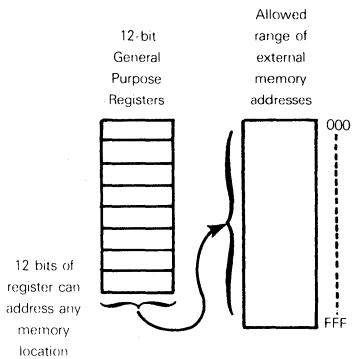
DIRECT ADDRESSING

Direct addressing means that the Jump instruction object code includes a 12-bit memory address:



REGISTER INDIRECT ADDRESSING

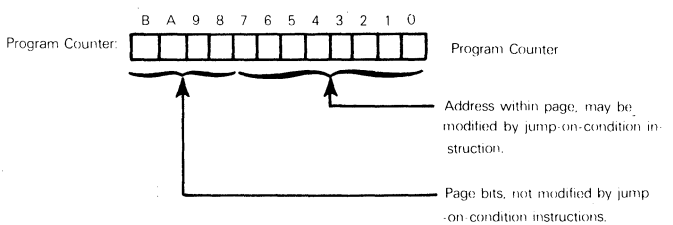
Register indirect (implied) addressing means that the Jump instruction specifies one of the general purpose registers as the source of the 12-bit memory address:



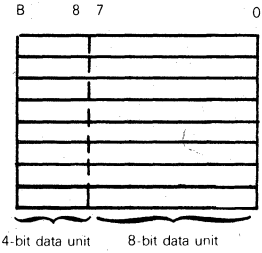
CONDITIONAL JUMP INSTRUCTIONS

PAGING OF PROGRAM MEMORY

Conditional Jump instructions modify the low order eight bits of the Program Counter only; this gives rise to a limited degree of paged program memory addressing. Visualize external program memory as divided into 256 byte pages; a conditional Jump instruction can only jump to an address within the current program memory page, as identified by the high order four Program Counter bits:



The concept of paged external memory is reinforced by the fact that the general purpose registers, which provide implied data memory addresses, are accessed as separate 4-bit and 8-bit units for data manipulation operations:



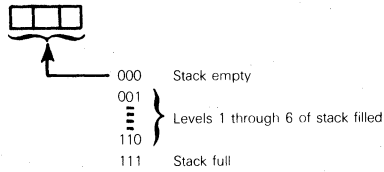
THE SUBROUTINE STACK

The EA9002 subroutine stack is a typical cascade stack implemented within the CPU. The stack consists of seven 12-bit registers. **There are no special stack instructions in the EA9002 instruction set.** Rather, whenever a jump to subroutine instruction is executed, the Program Counter contents are pushed onto the stack before the subroutine execution address gets loaded into the program counter.

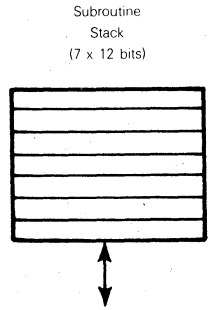
There is also a 3-bit Stack Pointer which at all times records the current level of stack access. The 3-bit Stack Pointer is treated by EA9002 instructions as part of an 8-bit Status Register. You can examine the current level of stack access by reading status register contents into the Accumulator.

STACK POINTER

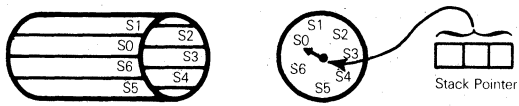
The 3-bit Stack Pointer should be interpreted as follows:



The stack is illustrated in Figure 2-3 as a simple concatenation of 12-bit registers:

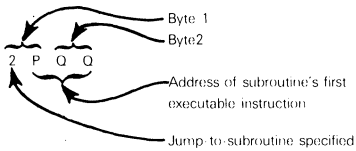


A conceptually more accurate representation would be as an "endless loop":

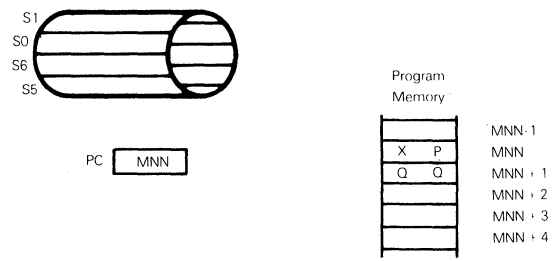


Assuming the stack is initially empty, let us examine how a Jump-to-subroutine instruction pushes the Program Counter contents onto the stack.

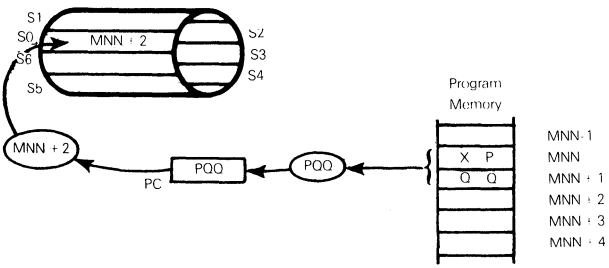
The Jump-to-subroutine object code consists of these four hexadecimal digits occupying two program memory bytes:



PQQ represent any three-hexadecimal-digit address. Assume the two Jump-to-subroutine object program bytes reside in some memory locations MNN and MNN + 1. Before the instruction is executed, this is the situation:

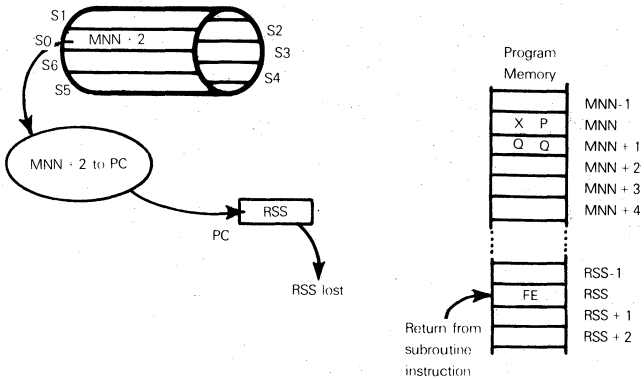


After the instruction is executed, this is the result:



When an interrupt is acknowledged, a Jump-to-subroutine instruction is forced with a subroutine execution address of 002_{16} is assumed. An interrupt acknowledge therefore automatically causes prior Program Counter contents to be pushed onto the stack.

When a return instruction is executed the top stack address is popped into the Program Counter:



Since there are seven registers in the subroutine stack, the EA9002 allows subroutines and interrupts to be nested to a depth of 7. If more than 7 interrupts and/or subroutine calls are executed in sequence, then the subroutine stack will overflow.

STACK OVERFLOW

STATUS FLAGS

The EA9002 status flags deserve special mention since they contribute significantly to the power of this microprocessor.

First we will discuss the interrupt, carry and half-carry statuses which are traditional.

The Carry status (C) identifies carries out of the high-order Accumulator bit; it is affected by arithmetic, compare and logical operations.

CARRY STATUS

The EA9002 carry status is unusual in subtract operations, when compared to common microcomputer practice. The EA9002 has direct subtract logic, it does not use twos complement addition. This means that a negative result following a subtract will set the Carry to 1; a positive result resets the Carry to 0.

CARRY STATUS AND SUBTRACT LOGIC

The Half-carry status (H) identifies carries out of Accumulator bit 3; it is used for binary-coded decimal arithmetic and logical operations.

HALF CARRY

The Interrupt status (I), when 1, indicates that external interrupts have been enabled. When this status contains 4, all interrupts are disabled.

INTERRUPT STATUS

The Accumulator and Decimal mode status are unusual.

ACCUMULATOR STATUS

The Accumulator status identifies the current Accumulator contents as being 0 (A = 0) or non-zero (A = 1). Following Compare instructions however, the Accumulator status identifies the result of the Compare operation, but only during execution of the single, next sequential instruction.

The Accumulator status differs from the traditional Zero status in these two respects:

1) The Accumulator status has opposite interpretation; traditionally, a "1" Zero status indicates a zero value, while a "0" Zero status indicates a non-zero value; this is the exact opposite of the Accumulator status's interpretation.

2) The Accumulator status represents current Accumulator contents, whereas a Zero status is set or reset by selected instructions, not necessarily reflecting Accumulator contents at all times.

DECIMAL STATUS

The Decimal status (D) allows arithmetic operations to occur in decimal mode or in binary mode. When set to one, this status causes data to be interpreted as packed binary coded decimal for all arithmetic operations. This status eliminates the need for separate decimal adjust or decimal arithmetic instructions.

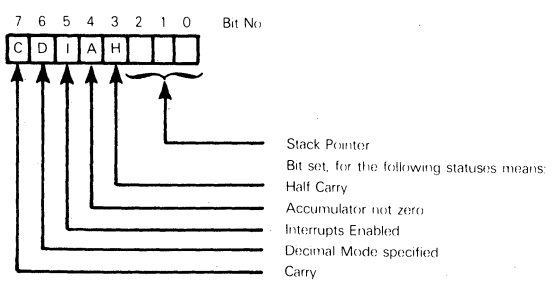
Decimal mode operations in the EA9002 are very complete; for example, binary values in excess of 9 are adjusted. Here are some examples of addition in decimal mode:

09	01 = 10	C = 0	H = 1
0A	00 = 10	C = 0	H = 1
C2	00 = 22	C = 1	H = 0
C2	20 = 42	C = 1	H = 1
0A	C2 = 32	C = 1	H = 1
C2	D5 = 57	C = 1	H = 1

So long as the result of an addition is 199 or less, an accurate decimal result will be generated — whatever the condition of addition inputs.

STATUS FLAG SUMMARY

The EA9002 status register is, in reality, an 8-bit register, whose contents can be transferred to the Accumulator; Accumulator contents are then interpreted as follows:



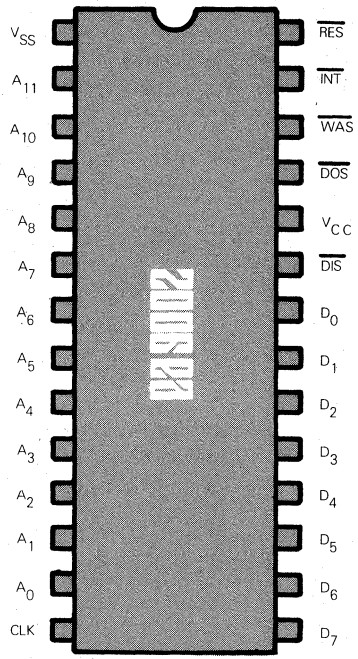


Figure 2-4 EA9002 Pins

PINS AND SIGNALS

EA9002 pins are identified in Figure 2-4.

ADDRESS BUS

The twelve address pins provide a 12-bit extended memory address. These pins are connected to the microcomputer system address bus.

DATA BUS

The eight data pins drive the bidirectional microcomputer system data bus.

CONTROL BUS

The control bus has five simple signals which make it very easy to interface external logic asynchronously to the EA9002. All control bus signals are active-low.

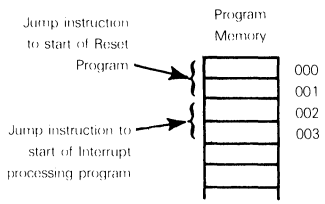
RESET

Reset ($\overline{RES}$) is a typical asynchronous Reset input signal. When this signal is input low (for a minimum of 3 external clock periods) the Program Counter is set to 0, and the stack is emptied. Program execution will now branch to memory location 000. The D and I status flags are set to 0, which means that interrupts are inhibited and arithmetic operations are set to binary mode.

When interrupt ($\overline{INT}$) is input low, external logic is requesting an interrupt. Providing the Interrupt status is set to 1, at the conclusion of the current instruction's execution (with some exceptions), the interrupt will be acknowledged. When an interrupt is acknowledged, a Jump-to-Subroutine instruction is executed, with the subroutine start address identified as memory location 002. The interrupt status is reset to 0, inhibiting further interrupts.

INTERRUPT

In order to handle reset and interrupt logic, the first bytes of memory are usually used as follows:



WAIT/SYNC ($\overline{WAS}$) is a bidirectional control line; it provides the logic necessary for the EA 9002 to communicate asynchronously with external logic. This signal is output low during the last clock period of an instruction cycle, indicating to external logic that an instruction fetch is to initiate a new instruction cycle during the next clock period.

WAIT /SYNC

External logic may at any time hold $\overline{WAS}$ low. So long as $\overline{WAS}$ is held low, the 9002 enters a Wait state during which pending operations internal to the CPU are completed, then no further operations occur.

Use of the WAS signal is described in Chapter 3.

When the EA9002 is outputting data, DATA OUT STROBE ($\overline{DOS}$) is output low while data and memory addresses are stable on their respective busses. This signal may be used as a write (R/W) input strobe to external read/write memory or peripheral devices.

DATA OUT STROBE

When the EA 9002 requires data input, it outputs DATA IN STROBE ($\overline{DIS}$) low in order to indicate that the appropriate memory address is on the address bus and external logic is to place data on the data bus. This signal may be used as an enable strobe for any external logic that has to transmit data to the EA9002.

DATA IN STROBE

CLK is the Clock signal input pin. A single phase clock with a cycle time of 250 nanoseconds or longer is required. See Chapter 3 for signal characteristics.

CLOCK

V_{SS} and V_{CC} provide power and ground connections. A single +5V power supply is all that is required.

POWER

Chapter 3
EA9002 MICROPROCESSOR CHARACTERISTICS

EA9002 characteristics are identified in terms of signals and timing, since these parameters may be used to explain instruction execution sequences.

Instructions' timing given in Figures 3-1 through 3-9 separately illustrate signal interactions and wave form timing.

INSTRUCTION TIMING

All EA9002 instructions are executed synchronously, via instruction cycles that are timed by clock signal CLK.

INSTRUCTION CYCLE

MACHINE CYCLE

STATE TIME

An instruction cycle defines the time required to fetch and execute an instruction. **Every instruction cycle consists of one or two machine cycles, referred to as M1 or M2. Each machine cycle consists of four state times; each state time is referred to as T1, T2, T3, or T4 and is defined as the time interval between the positive transitions of alternate input clock pulses (CLK). See Figure 3-1. The first clock input to occur during each state time is defined as Φ1 and the second clock input as Φ2.**

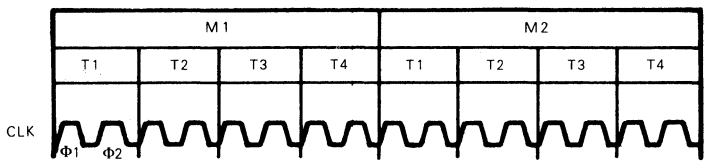


Figure 3-1. Machine Cycle, State Time And Clock Timing

If you need to initially synchronize CLK, include logic that suppresses clock inputs to the EA9002 during power up. Once power has been applied, initiate clock frequency inputs, and the first CLK output will be a Φ1 pulse.

In general, single byte instructions require one machine cycle to execute, whereas two byte instructions require two machine cycles to execute. Exceptions do exist. Those single byte instructions using decimal mode require two machine cycles, as do the single byte LRN and SRN instructions.

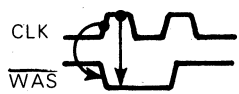
WAIT ($\overline{WAS}$) and RESET ($\overline{RES}$) operations are externally controlled, therefore, the duration of certain state times becomes indeterminate, although internal logic keeps these states synchronized with the clock input. The operations of $\overline{WAS}$ and $\overline{RES}$ are described later in this chapter.

For external logic synchronization, the EA9002 provides a SYNC pulse, via the WAIT AND SYNC pin ($\overline{WAS}$). SYNC identifies the beginning of a new instruction cycle.

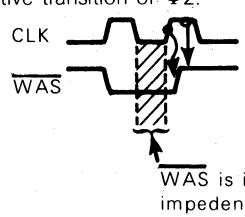
SYNC

The SYNC pulse is an active low output which occurs during the last T4 state time of an instruction cycle as shown in Figure 3-2A and 3-2B.

$\overline{WAS}$ is forced low by the positive transition of $\Phi 1$ and is held low during the $\Phi 1$ pulse input:



The positive transition of $\Phi 2$ forces $\overline{WAS}$ high; $\overline{WAS}$ is held high during the $\Phi 2$ pulse input. $\overline{WAS}$ is in a high impedence state after the negative transition of $\Phi 1$ and remains there until the positive transition of $\Phi 2$:



An internal pullup resistor on the $\overline{WAS}$ pin will cause the SYNC to return to a high level during the high impedence state, if for any reason the time interval between clocks is sufficiently long.

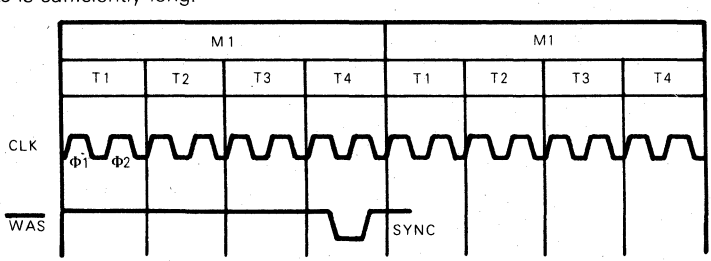


Figure 3-2A. Sync Timing For Single Byte Instructions

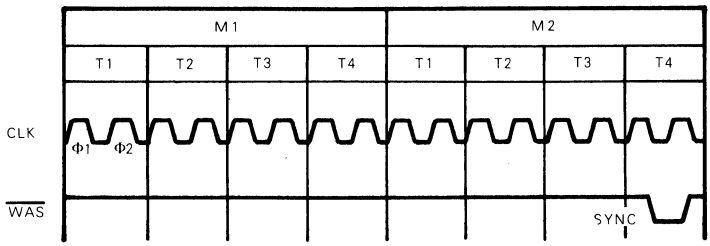


Figure 3-2B. Sync Timing For Two Byte, Decimal Mode, LRN, And SRN Instructions

INSTRUCTION FETCH AND EXECUTION

Every instruction cycle refers to memory at least once, during which an instruction is fetched. An instruction **FETCH** **cycle must always have a FETCH and the fetch occurs during T1 and T2 of the first machine cycle M1.**

Externally, the instruction fetch is a simple "read memory" operation. The EA9002 places the contents of the Program Counter onto the Address Bus and forces the Data In Strobe ($\overline{DIS}$) low. The addressed memory byte must return on the Data Bus while $\overline{DIS}$ is low, and is sampled during $\Phi 1$, T2 as shown in Figure 3-3.

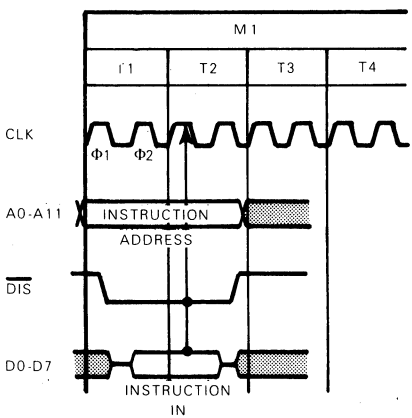


Figure 3-3. Instruction Fetch Timing

Following the instruction fetch, one of seven different instruction sequences may occur. These are best understood by examining the various instruction cycles.

SINGLE BYTE INSTRUCTIONS

Single byte instructions require only one machine cycle, MI, to perform the fetch and execution. The state times T1, T2 are used to fetch the instruction, and T3, T4 are used to perform the required internal operations. **As shown in Figure 3-4, only one memory reference is generated.**

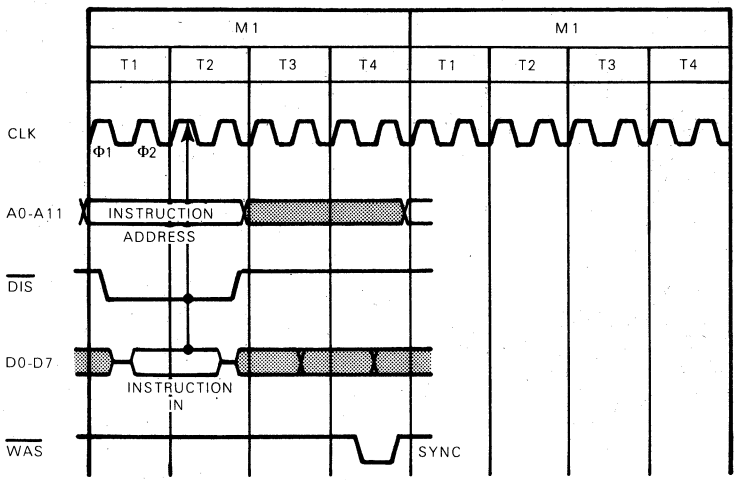


Figure 3-4. Single Byte Instruction Cycle

Exceptions to the timing shown in Fig. 3-4 exist for the single byte instructions INP, OUT, LRN, and SRN, as well as for those single byte instructions used in decimal mode.

Figure 3-5 illustrates the timing required for the Input instruction (INP). Only one machine cycle M1 is required to fetch and execute INP. The contents of the selected Page and General Register are placed on the Address Bus during T3, T4, which is used to select the external device from which data is to be read. The Data In Strobe ($\overline{DIS}$) is also forced low allowing the data on the Data Bus to be stored in the Accumulator during Φ1,T4.

Shown in Figure 3-6 is the cycle timing for the output (OUT) instruction. Again, only one machine cycle M1 is required. During T3, T4 the content of the Accumulator is placed onto the Data Bus; the selected Page and General Register contents are placed onto the Address Bus and are used to define the external device into which data is to be transferred. The Data Out Strobe ($\overline{DOS}$) is forced low after the Data Bus is stable.

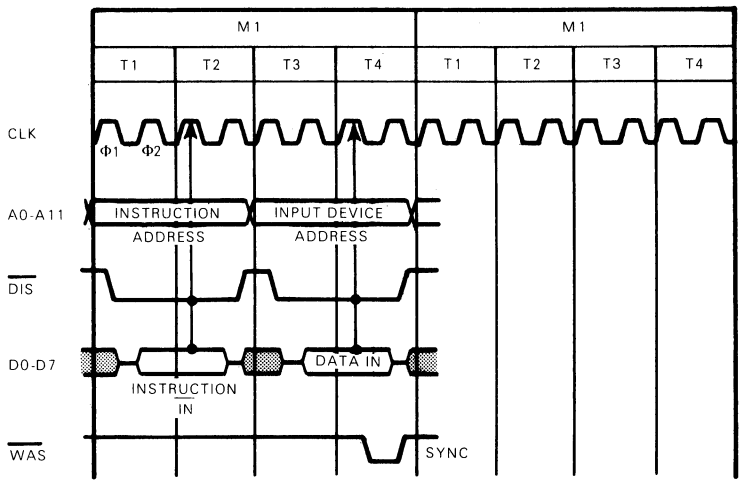


Figure 3-5. Input (INP) Instruction Cycle

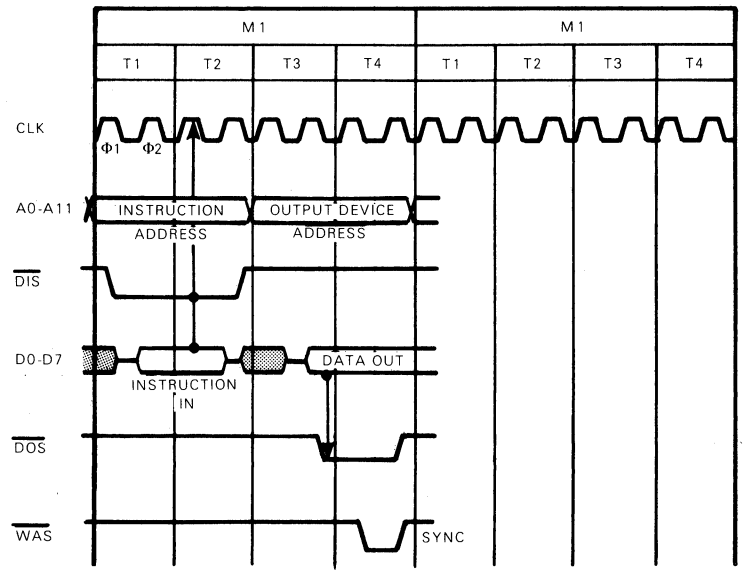


Figure 3-6. Output (OUT) Instruction Cycle

$\overline{DIS}$ low defines the time interval when the Address Bus is valid and must be decoded to enable an external device, such as a memory, to transfer data to the Accumulator.

$\overline{DOS}$ low defines the time interval when the value of the Accumulator is stable on the Data Bus and can be transferred to the external device identified by the Address Bus.

Hence, $\overline{DIS}$ and $\overline{DOS}$ control Read and Write operations performed by external devices, since $\overline{DIS}$ can be used as a Chip Select for Read operations and $\overline{DOS}$ can be used as a Chip Select and a Write pulse during Write operations.

Load Register Indirect (LRN), is a single byte instruction, but it requires two machine cycles, M1 and M2. Figure 3-7 shows the LRN instruction cycle. LRN may be thought of as an implied address instruction, since a General Purpose Register is implied as the source of the indirect address.

The content of the General Purpose Register is placed on the Address Bus during T2, T3, T4, of M2 and is used to select the external device from which data is to be read. The Data In Strobe ($\overline{DIS}$) is forced low, allowing the data to be stored in the selected General Register during $\Phi 1$, T3 as shown in Figure 3-7.

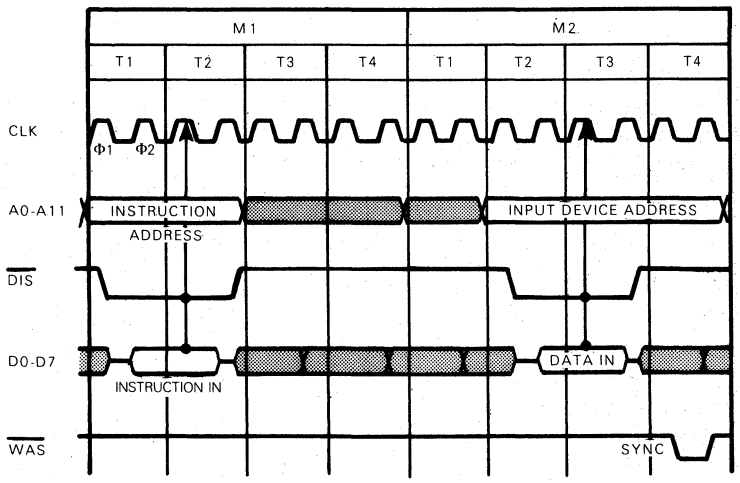


Figure 3-7. Load Register Indirect (LRN) Instruction Cycle

Store Register Indirect (SRN), shown in Figure 3-8, is also a single byte instruction which requires two machine cycles, M1 and M2. A General Purpose Register is implied as the source of the indirect address which is placed on the Address Bus during T2, T3, T4 of M2. This address is used to select the external device into which data is to be transferred. The content of the selected General Purpose Register is placed onto the Data Bus during T2, T3 of M2, and the Data Out Strobe ($\overline{DOS}$) is forced low after the Data Bus is stable.

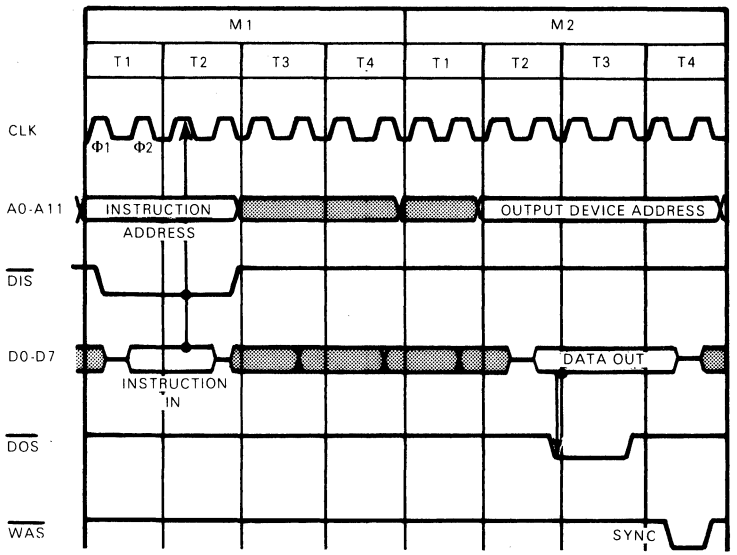


Figure 3-8. Store Register Indirect (SRN) Instruction Cycle

Figure 3-9 illustrates the instruction cycle timing of single byte instructions used in decimal mode. Decimal Mode instructions require two machine cycles, M1 and M2, for internal logic to have time needed to perform necessary BCD operations. With this exception, the instruction cycle is the same as the single byte instruction cycle illustrated in Figure 3-4.

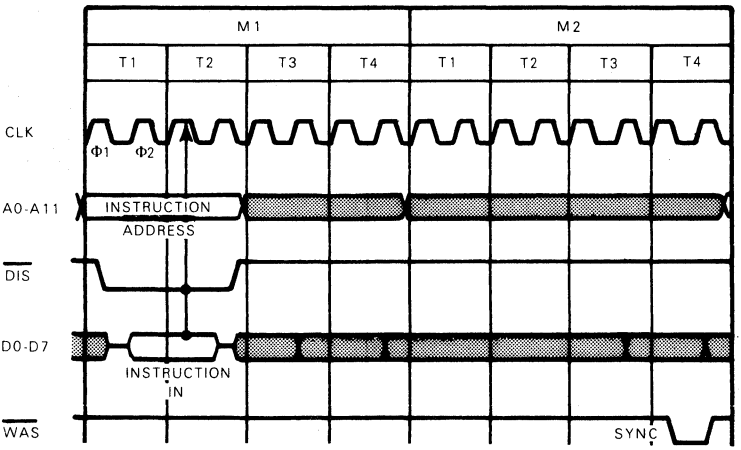


Figure 3-9. Decimal Mode Instruction Cycle

TWO BYTE INSTRUCTIONS

Immediate and Jump instructions (except JIN) require two bytes of object code, and hence, two machine cycles, M1 and M2, to fetch and execute the instruction. Figure 3-10 shows the instruction cycle for two byte instructions. During the second machine cycle (M2), the content of the Program Counter is placed on the Address Bus to FETCH the second byte of the instruction. The Data In Strobe ($\overline{DIS}$) is forced low during T1, T2 of M2 and allows the data fetched to be loaded into the EA9002 during $\Phi 1$, T2.

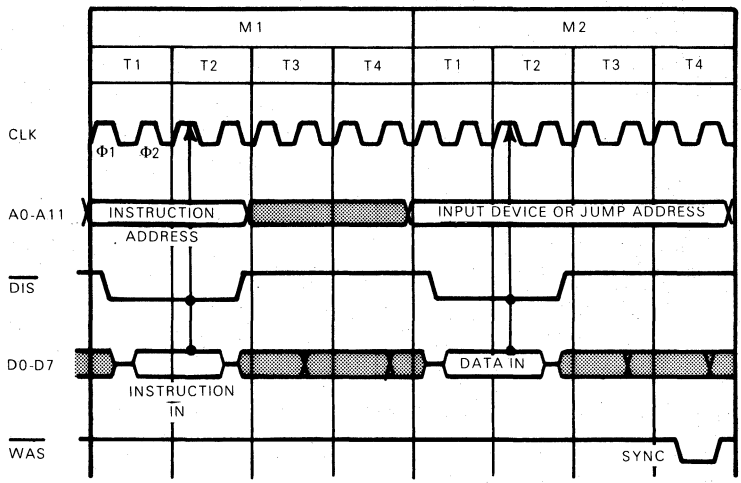
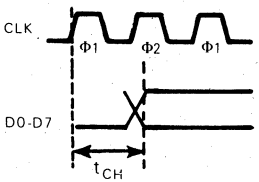


Figure 3-10. Two Byte Instruction Cycle

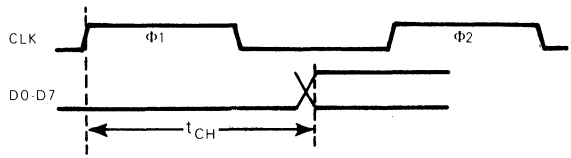
INTERMEDIATE STATE TIME BUS UTILIZATION

The Address Bus and the Data Bus, when not transferring data required by an instruction, output intermediate data. This data can be useful during functional testing of the microprocessor, and in program de-bugging. Referring to Figures 3-4 through 3-10, the intermediate data appear as shaded areas on the Address Bus and Data Bus.

Intermediate data timing should be computed by adding the propagation delays defined in Figure 3-15 to the active CLK signal edge. On the data bus, this is the result:



When CLK has maximum frequency, t_{CH} may cause intermediate data to appear during the next time period, that is, during the next $\Phi 2$. If the clock frequency slows down, intermediate data will appear during the current $\Phi 1$:



Intermediate address bus outputs will appear similarly, originating in the next $\Phi 2$ clock pulse for maximum frequency operations, or in the current $\Phi 1$ clock pulse for slow clocks.

Timing of intermediate data will generally be the same as timing for corresponding data as required by an instruction cycle; but intermediate data timing is not guaranteed.

Table 3-1 summarizes the intermediate Address Bus and Data Bus information during each state time of all instructions.

Table 3-1 assumes that you understand the EA9002 instruction set—which will not be done until you have read Chapter 5. Therefore, bypass Table 3-1 when reading this book to gain a general understanding of the EA9002.

Improvements embodied in future EA9002 type products may occur from time to time which will eliminate some of the intermediate data and address bus utilization which appear in this version of the EA9002 microprocessor. You are cautioned to refer to the detail specifications for the particular version being employed. Any product improvements, referred to above, are intended to maintain complete electrical, software and hardware compatibility.

The following abbreviations are used in Table 3-1:

AC	Accumulator contents just before being modified by the instruction.
AD	Accumulator contents just before decimal correction for decimal mode instruction.
GDX	Low order eight bits of selected general purpose register.
GPX	High order four bits of selected general purpose register, output on low order four data bus lines.
GRX	All twelve bits of selected general purpose register in GPX, GDX, or GRX, if x is a digit between 0 and 7, it specifies one general purpose register. Thus GR4 specifies the 12-bit contents of general purpose register 4.
I1	Low order four bits of the first byte in a two byte instruction.
I2	Second byte of a two byte instruction.
KK	Correction constant used in decimal arithmetic.
PC	Program counter contents.
SM	Selected scratchpad memory byte content.
SW	Status word.

Table 3-1 Intermediate Data and Address Bus Utilization

Instruction Mnemonic	Bus	M1				M2			
		T1	T2	T3	T4	T1	T2	T3	T4
ADD (Binary) Figure 3-4	Addr Data	PC	PC	GRX GDX	GRX AC				
ADD (Decimal) Figure 3-9	Addr Data	PC	PC	GRX GDX	GRX AC	PC	PC	PC KK	PC AD
ADS (Binary) Figure 3-4	Addr Data	PC	PC	GRX SM	GRX AC				
ADS (Decimal) Figure 3-9	Addr Data	PC	PC	GRX SM	GRX AC	PC	PC	PC KK	PC AD
AND Figure 3-4	Addr Data	PC	PC	GRX GDX	GRX AC				
CAP Figure 3-4	Addr Data	PC	PC	GRX AC	GRX AC				
CAR Figure 3-4	Addr Data	PC	PC	GRX AC	GRX AC				
CLA Figure 3-4	Addr Data	PC	PC	GR6 SW	GR6 AC				
CLB Figure 3-4	Addr Data	PC	PC	GR2 SW	GR2 AC				
CLC Figure 3-4	Addr Data	PC	PC	GR0 SW	GR0 AC				
CMA Figure 3-4	Addr Data	PC	PC	GR7 SW	GR7 AC				
CMC Figure 3-4	Addr Data	PC	PC	GR3 SW	GR3 AC				
CMP Figure 3-4	Addr Data	PC	PC	GRX GDX	GRX AC				

Table 3-1 Intermediate Data and Address Bus Utilization (Continued)

Instruction Mnemonic	Bus	M1				M2			
		T1	T2	T3	T4	T1	T2	T3	T4
CPA Figure 3-4	Addr Data	PC	PC	GRX GPX	GRX 0				
CRA Figure 3-4	Addr Data	PC	PC	GRX GDX	GRX 0				
CSA Figure 3-4	Addr Data	PC	PC	GR4 SW	GR4 0				
DAC (Binary) Figure 3-4	Addr Data	PC	PC	GR4 SW	GR4 AC				
DAC (Decimal) Figure 3-4	Addr Data	PC	PC	GR4 SW	GR4 AC	PC	PC	PC SW	PC AD
DCR Figure 3-4	Addr Data	PC	PC	GRX GDX-1	GRX AC				
DLY Figure 3-4	Addr Data	PC	PC	GR0 SW	GR0 AC	PC	PC	PC SW	PC AC
DRJ Figure 3-10	Addr Data	PC	PC	GR0 SW	GR0 AC	PC	PC	PC SW	PC AC
DSI Figure 3-4	Addr Data	PC	PC	GR2 SW	GR2 AC				
ENI Figure 3-4	Addr Data	PC	PC	GR3 SW	GR3 AC				
IAC (Binary) Figure 3-4	Addr Data	PC	PC	GR4 SW	GR4 AC				
IAC (Decimal) Figure 3-9	Addr Data	PC	PC	GR4 SW	GR4 AC	PC	PC	PC SW	PC AD
INP Figure 3-5	Addr Data	PC	PC	GRX	GRX				

Table 3-1 Intermediate Data and Address Bus Utilization (Continued)

Instruction Mnemonic	Bus	M1				M2			
		T1	T2	T3	T4	T1	T2	T3	T4
INR Figure 3-4	Addr Data	PC	PC	GRX GDX-1	GRX 0				
IOR Figure 3-4	Addr Data	PC	PC	GRX GDX	GRX AC				
IRJ Figure 3-10	Addr Data	PC	PC	GRX GDX + 1	GRX AC	PC	PC	PC SW	PC AC
JIN Figure 3-4	Addr Data	PC	PC	GRX SW	GRX GDX				
JSR, JUN Figure 3-10	Addr Data	PC	PC	GRX SW	GRX AC	PC	PC	PC SW	PC I1
All Jump Cond. Figure 3-10	Addr Data	PC	PC	GRX SW	GRX AC	PC	PC	PC SW	PC AC
LAI Figure 3-10	Addr Data	PC	PC	GR1 SW	GR1 AC	PC	PC	PC SW	PC I2
LRI Figure 3-10	Addr Data	PC	PC	GRX SW	GRX AC	PC	PC	PC SW	PC AC
LRN Figure 3-7	Addr Data	PC	PC	GRX SW	GRX AC	PC	GR0	GR0	GR0 AC
NOP Figure 3-4	Addr Data	PC	PC	GR7 SW	GR7 AC				
OUT Figure 3-6	Addr Data	PC	PC	GRX AC	GRX AC				
RAL Figure 3-4	Addr Data	PC	PC	GR0 AC	GR0 AC				
RAR Figure 3-4	Addr Data	PC	PC	GR1 AC	GR1 AC				

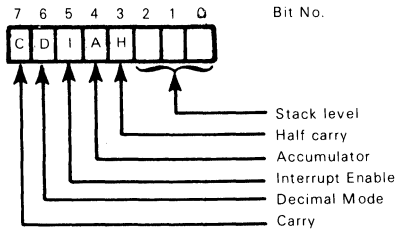
Table 3-1 Intermediate Data and Address Bus Utilization (Continued)

Instruction Mnemonic	Bus	M1				M2			
		T1	T2	T3	T4	T1	T2	T3	T4
RDS Figure 3-4	Addr Data	PC	PC	GRX SM	GRX 0				
RET Figure 3-4	Addr Data	PC	PC	GR6 SW	GR6 AC				
RLC Figure 3-4	Addr Data	PC	PC	GR2 AC	GR2 AC				
RRC Figure 3-4	Addr Data	PC	PC	GR3 AC	GR3 AC				
SEB Figure 3-4	Addr Data	PC	PC	GR5 SW	GR5 AC				
SEC Figure 3-4	Addr Data	PC	PC	GR1 SW	GR1 AC				
SED Figure 3-4	Addr Data	PC	PC	GR4 SW	GR4 AC				
SRN Figure 3-8	Addr Data	PC	PC	GRX GDX	GRX AC	PC	GR0	GR0	GR0 AC
SUB (binary) Figure 3-4	Addr Data	PC	PC	GRX GDX	GRX AC				
SUB (decimal) Figure 3-9	Addr Data	PC	PC	GRX GDX	GRX AC	PC	PC	PC KK	PC AD
SUS (binary) Figure 3-4	Addr Data	PC	PC	GRX SM	GRX AC				
SUS (decimal) Figure 3-9	Addr Data	PC	PC	GRX SM	GRX AC	PC	PC	PC KK	PC AD
WRS Figure 3-4	Addr Data	PC	PC	GRX AC	GRX AC				

Table 3-1 Intermediate Data and Address Bus Utilization (Continued)

Instruction Mnemonic	Bus	M1				M2			
		T1	T2	T3	T4	T1	T2	T3	T4
XCH Figure 3-4	Addr Data	PC	PC	GRX AC	AC GDX				
XOR Figure 3-4	Addr Data	PC	PC	GRX GDX	GRX AC				

The status word is output as follows:



THE WAIT STATE

The EA9002 will suspend any instruction cycle when the WAIT input is externally forced low. Functionally, the WAIT input is internally sampled every Φ1 clock time during the state times indicated in Table 3-2. If $\overline{WAS}$ is low and meets the timing shown in Figure 3-11, the instruction cycle is suspended for the duration of the WAIT signal—in the state time during which $\overline{WAS}$ was sampled. The machine cycle cannot advance to the next state time until the Φ1 clock following the return of $\overline{WAS}$ to a high level. **Data within the EA9002 is unaffected by WAIT. The Address Bus, Data Bus (for Data out only), $\overline{DIS}$, and $\overline{DOS}$ will remain stable.**

WAIT

TABLE 3-2. WAIT STATES

INSTRUCTION TYPES	WAIT OCCURS	
	M1	M2
SINGLE BYTE	T1	
INPUT	T1,T3	
OUTPUT	T1,T3	
LRN	T1	T2
SRN	T1	T2
DECIMAL	T1	
TWO BYTE	T1	T1

WAIT can be used to force a temporary CPU halt such as required by single step operations, or to extend the response time of the EA 9002 during data read and write operations with slower peripheral devices.

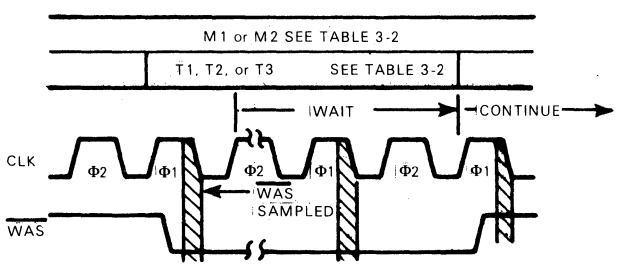


Figure 3-11. Wait Timing

A conflict can exist between SYNC generated by the EA 9002 and external logic attempting to control WAIT. If the external logic attempts to force $\overline{WAS}$ to a high level while SYNC is driving low ($\Phi1$, T4 of the last state time of an instruction cycle), an indeterminate logic level may exist during $\Phi1$. Likewise if external logic attempts to force $\overline{WAS}$ low while SYNC is driving high ($\Phi2$, T4 of the last state time of an instruction cycle), an indeterminate level may exist during $\Phi2$. No damage should occur to either the 9002 or external logic under these conditions, provided rated currents are not exceeded. Care should be taken in normal implementation such that this conflict does not occur.

Ways of implementing a WAIT sequence are discussed in Chapter 4.

INTERRUPT SEQUENCE

INTERRUPT

The Interrupt sequence starts with the EA9002 sampling $\overline{INT}$ during $\Phi2$, T3 of the machine cycle during which a SYNC will be generated. Timing is illustrated in Figure 3-12. If $\overline{INT}$ was low during the sample, the EA9002 acknowledges the interrupt at the beginning of the next M1 machine cycle, providing two prior conditions exist:

1) The interrupt enable status flag must be set to "enable interrupt"
2) the compare instruction (CMP) is not being executed.

If neither of these two conditions exists, or if one, but not both of these conditions exist, the Interrupt is ignored.

When the EA9002 acknowledges an interrupt, a Jump-to-Subroutine instruction (JSR) is executed, forcing the subroutines starting address to location 002. The interrupt enable status flag is set to "disable interrupt". No other status or register contents are modified in any way.

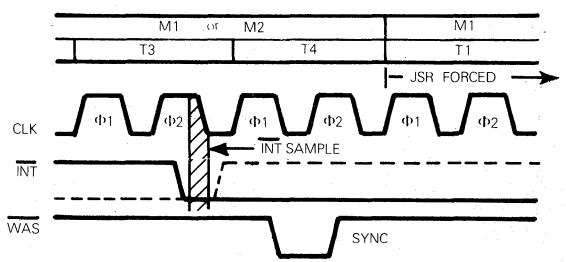


Figure 3-12A. $\overline{INT}$ Sample Time

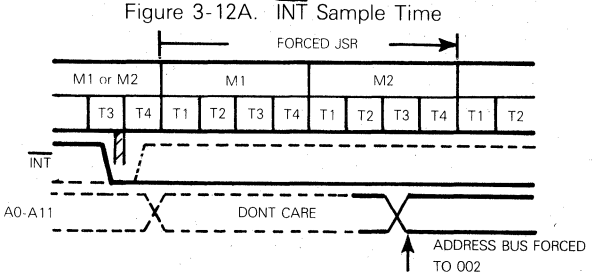


Figure 3-12B. Interrupt Cycle

Since $\overline{INT}$ is an asynchronous input it must be held low sufficiently long to guarantee that $\overline{INT}$ has a valid sample. For example, consider the instruction sequence INP, CMP, JCN. The $\overline{INT}$ input must be held low for slightly longer than twelve state times to guarantee an interrupt acknowledge by the EA9002. This time interval will be extended if a WAIT sequence is initiated prior to the $\overline{INT}$ sample.

Note that $\overline{INT}$ is not sampled during CMP.

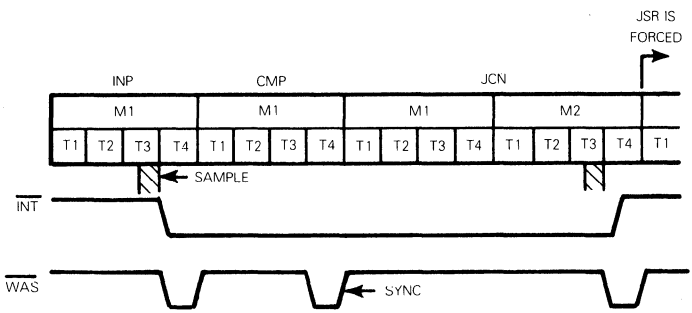


Figure 3-13. Interrupt Hold Time

It is good design practice to hold $\overline{INT}$ low until this interrupt request has been acknowleged, by whatever interrupt acknowlege logic is in effect. Some possiblities are described in Chapter 4.

RESET

When power is applied to the EA9002, all logic in the processor begins operating. The contents of its registers, program counter, etc. start at a random state; therefore, a RESET ($\overline{RES}$) is provided to force the processor to a known starting condition. $\overline{RES}$ is sampled during every Φ2 clock input. When this signal is low, the EA9002 responds by setting the program counter and the stack pointer to 000; the interrupt enable and decimal mode flags are reset to 0. The machine cycle reverts to T1, M1 until $\overline{RES}$ is returned high.

The EA9002 will begin program execution starting at memory location 000. Binary mode is set and Interrupts are disabled.

The $\overline{RES}$ input must be held low for four clock periods in order to complete the Reset. Figure 3-14 shows the Reset timing.

The control output signals are held at a high level while Reset is active.

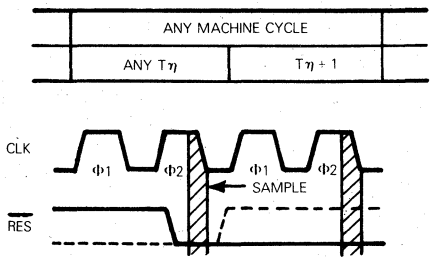


Figure 3-14A. Reset Sample Time

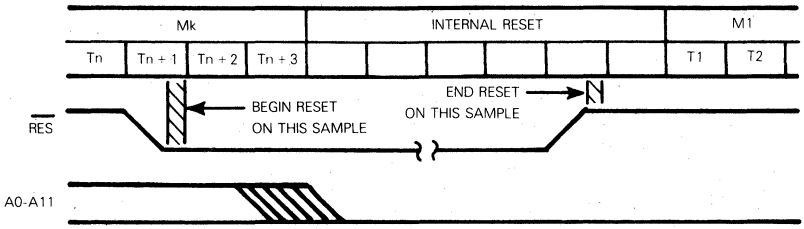
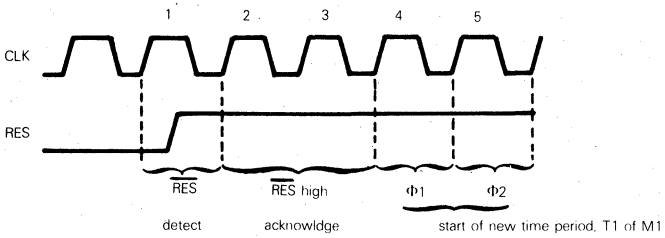


Figure 3-14B. Reset Cycle

As illustrated in Figure 3-14B, a new machine cycle will begin on the fourth CLK pulse following $\overline{\text{RES}}$ high:



DETAILED SIGNAL TIMING

An overall system timing diagram is presented in Figure 3-15 using symbols and data from Table 3-3. Table 3-3 is subject to change as improvements in operational characteristics are implemented in the EA9002, therefore, the reader is cautioned to refer to the latest detailed specification sheet for most current information.

Table 3-3. Timing Characteristics (Vcc=5.0v±5%, Vss=0v, Ta=0°C to 70°C Unless Otherwise Noted.)

SYMBOL	PARAMETER	MIN	MAX	UNIT	CONDITION
t_{CY}	CLOCK PERIOD	250	2500	ns	
t_{τ}	CLOCK TRANSITION TIME	5	25	ns	
$t_{\Phi W}$	CLOCK PULSE WIDTH	140		ns	
$t_{\Phi S}$	CLOCK PULSE SPACE	60		ns	
t_{DA}	ADDRESS OUTPUT DELAY TIME		195	ns	F.O.=TEST LOAD
t_{AH}	ADDRESS HOLD TIME	40		ns	F.O.=TEST LOAD
t_{CD}	$\overline{DIS}$ DELAY TIME		195	ns	F.O.=TEST LOAD
t_{CH}	$\overline{DIS}$ HOLD TIME		165	ns	F.O.=TEST LOAD
t_{DD}	$\overline{DIS}$ TO DATA IN DELAY TIME	0		ns	F.O. ON $\overline{DIS}$=TEST LOAD
t_{ACC}	ACCESS TIME FROM ADDRESS STABLE		475[1]	ns	F.O. ON A0-A11=TEST LOAD $T_{ACC}=3T_{CY}+T_{\Phi W}-T_{DA}-T_{DS}$
t_{DS}	DATA IN SETUP TIME	220		ns	
t_{DH}	DATA IN HOLD TIME	0		ns	
t_{DO}	DATA OUT DELAY TIME		315	ns	F.O.=TEST LOAD
t_{D1}	$\overline{DOS}$ DELAY TIME		165	ns	F.O.=TEST LOAD
t_{D2}	$\overline{DOS}$ HOLD TIME		125	ns	F.O.=TEST LOAD
t_{S1}	SYNC ($\overline{WAS}$) DELAY TIME		165	ns	F.O.=TEST LOAD
t_{S2}	SYNC ($\overline{WAS}$) HOLD TIME		125	ns	F.O.=TEST LOAD
t_{WS}	WAIT ($\overline{WAS}$) SETUP TIME	20		ns	
t_{RW}	$\overline{RES}$ PULSE WIDTH	1		μS	$t_{RW}=4t_{CY}$
t_{RS}	$\overline{RES}$ SETUP TIME	40		ns	
t_{IS}	$\overline{INT}$ SETUP TIME	40		ns	

[1] t_{ACC}=475nsec for t_{CY}=250nsec

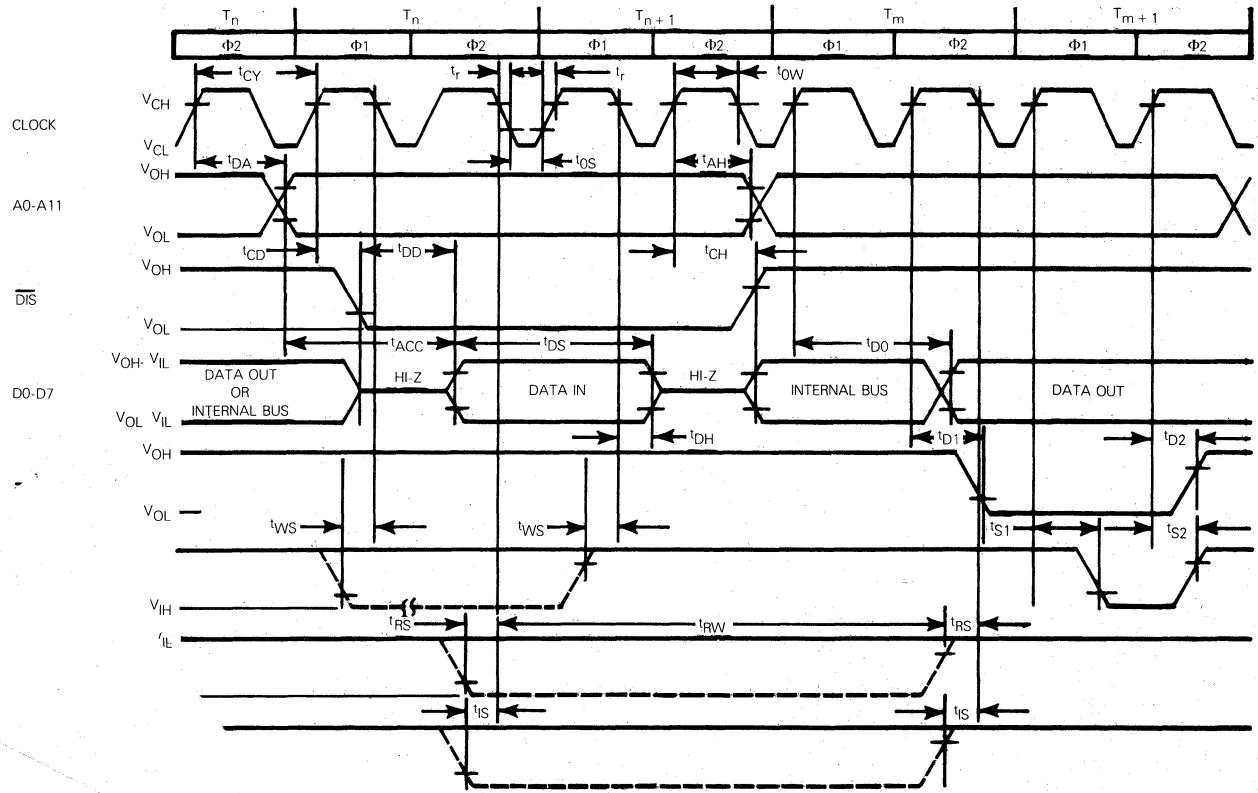


Figure 3-15 General Timing Diagram

Chapter 4
SYSTEMS CONFIGURATIONS

This chapter explains how to configure total microprocessor systems around the EA9002, we illustrate various means of interfacing the EA9002 to ROM, RAM and I/O.

Any microcomputer system must be able to input data or control signals, execute a predetermined sequence of program instructions (usually in ROM), temporarily storing the data (usually in RAM), then output data or control signals. Thus, the elements referred to as CPU (Central Process Unit), ROM (Read Only Memory), RAM (Random Access Memory), and I/O (input/output) must exist in one form or another within a total microprocessor system.

The EA9002 CPU chip contains a 64 x 8-bit RAM. Many real time data controllers will not need additional external RAM devices. External ROM devices, however, are always part of an EA9002 system, since there are no provisions for storing programs within EA9002 CPU. This allows you the freedom to select ROM devices which exactly fit your application. You may choose a few hundred bytes of ROM, or many thousand bytes of ROM. **EA, a major supplier of ROM devices, offers many varieties of ROM for every type of application and will be happy to help you select that device most suited to your needs. This section describes some of the available EA ROM's and how they are controlled by the EA9002.**

EA is also a major supplier of static and dynamic RAM devices of various sizes and speeds. Static RAM's like the EA2111 (256x4) are most often used in an EA9002 system. Therefore interface logic for this device is shown in this chapter.

New ROM's and RAM's of larger size and faster speed are a continuous development activity with EA, so it is advisable to stay in touch with EA sales for information and availability of new memory products.

I/O flexibility for microcomputer systems is another area where a great deal of effort is being applied by EA and other semiconductor manufacturers. **The EA9002, with its TTL bus structure and versatile control signals can interface with standard TTL devices, as well as speciality programmable I/O controllers.**

When implementing a microprocessor system with the EA9002, you may use industry standard devices such as A/D and D/A converters, UARTS, SDLC and other specialty communication controllers.

EA is developing programmable I/O devices specially for the EA9002 systems. Included are the EA9255 GP I/O and the EA9250 Keyboard/Display Controller.

The EA9255 GP I/O is a replacement part for the popular 8255 and provides up to 24 input/output pins definable under program control. See the product specifications on the EA9255 for complete details.

The EA9250 Keyboard/Display Programmable Controller is a new part which interfaces both keyboards (static and momentary) and displays. This device contains four basic modes of operation, which under program control interface with a broad variety of switches, as well as numeric and alpha-numeric displays. The device performs the functions of automatic keyboard scanning and display refresh, allowing the CPU to perform calculations and other tasks at real time speeds. See the 9250 product specifications and application notes for complete datails. Figure 4-11 shows how the EA9250 is interconnected to the 9002 system bus.

EXTERNAL MEMORY

INTERFACING ROM

The most elementary EA9002 system consists of just two devices: the CPU and external program storage memory. Program storage memory usually consists of a PROM or ROM device.

PROM

ROM

Figure 4-1 illustrates a very simple, two device configuration. The EA4600 is a 2048 x 8-bit ROM. Being the only memory device present, its eleven address lines are connected directly to the A0 through A10 outputs of the EA9002; the remaining EA9002 address line is ignored. Also, in an elementary, two device configuration, we can simply tie the output lines of the EA4600 directly to the data pins of the EA9002.

The EA4600 ROM requires two output enables: OE1 and OE2. Each signal enables four of the eight data output lines. Since this device is being treated as an 8-bit memory, both enables are tied directly to the inverse of $\overline{DIS}$, $\overline{DOS}$ and $\overline{WAS}$ outputs of the EA9002 may be ignored, since this simple configuration makes no allowance for data output and the EA4600 ROM is fast enough that it does not need to input $\overline{WAS}$ low.

The EA4600 has an additional address enable input AR. This signal can be permanently enabled by tying to power, since all address outputs from the EA9002 that occur when $\overline{DIS}$ is low will be true.

Since interrupt logic is not being used, the $\overline{INT}$ input is permanently disabled by tying to power. However, even in such a simple configuration as illustrated in Figure 4-1, reset logic is enabled. In this case, it is shown with a switch to ground. Closing the switch will automatically reset the system.

Staying within the normal address space of the EA9002, that is 4096-bits of external memory, very little additional logic is needed upon increasing the size of external memory. Figure 4-2 illustrates two EA4600 ROM devices connected to an EA9002 CPU. As compared to Figure 4-1, the only innovation is the need for chip-select logic based on the condition of address line A11.

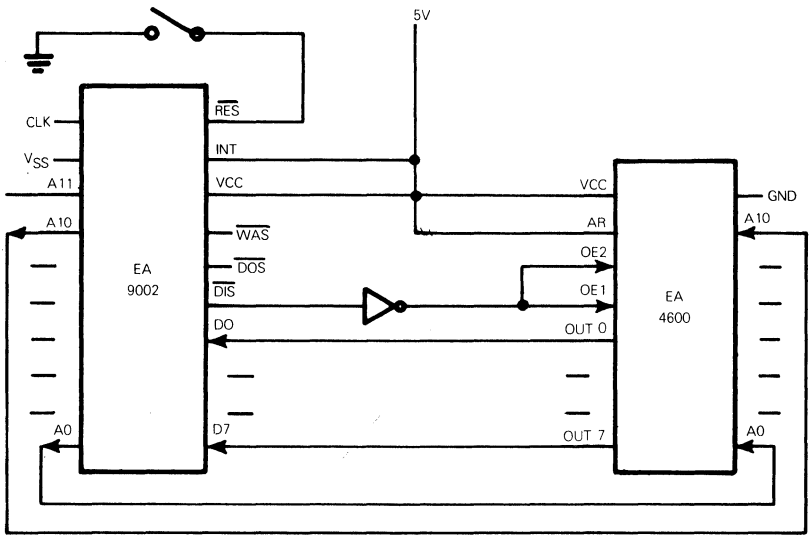
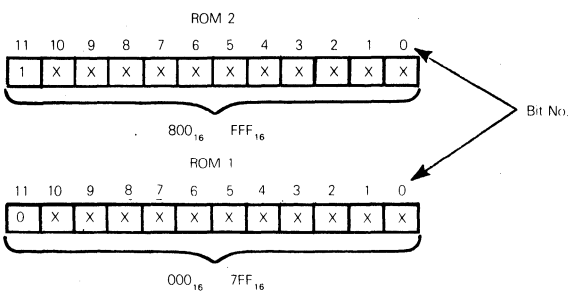
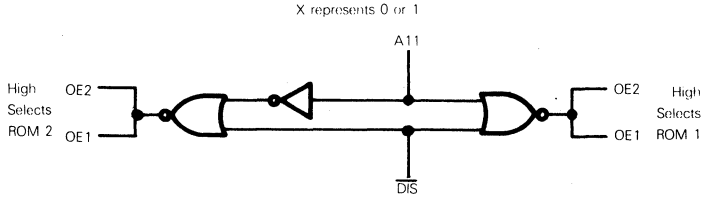


Figure 4-1 A Simple, Two Device, CPU - ROM System

Since each EA4600 ROM provides 2048-bits of memory, they cover the EA9002 address space as follows:





Thus, address line A11 may be used as a chip-select. The NOR of A11 and $\overline{DIS}$ will generate OE1 and OE2 true for the EA4600 ROM to the right of CPU if A11 is 0. If A11 is 1, A11 NOR $\overline{DIS}$ will generate OE1 and OE2 true for the EA4600 ROM to the left of the CPU.

Note the simplicity of chip-select logic.

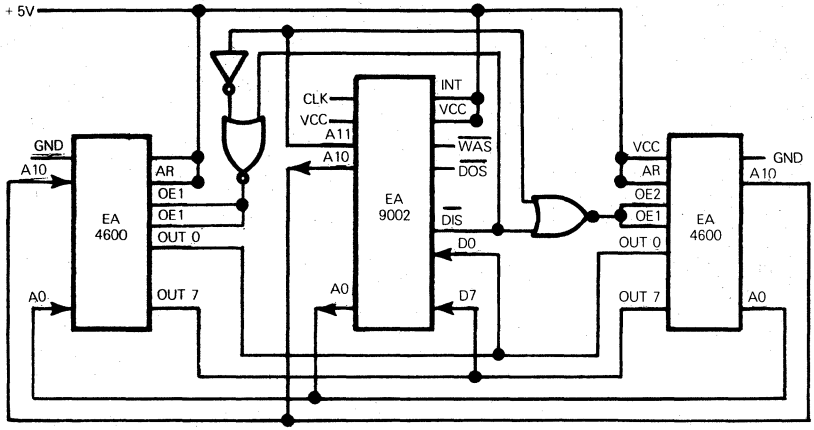
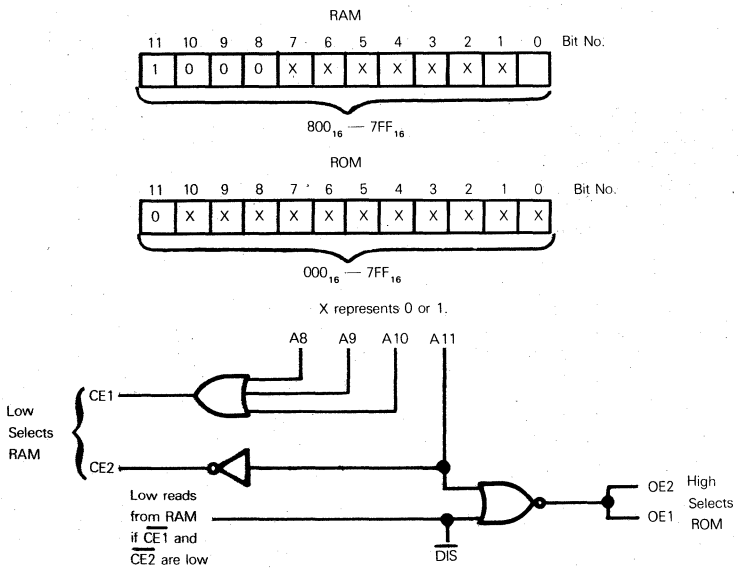


Figure 4-2 4096 Bytes Of ROM Connected To An EA9002

INTERFACING RAM

Suppose the 64 bytes of RAM provided within the EA9002 CPU is insufficient. External RAM may be added to the system. Figure 4-3 shows how one of the EA4600 ROM devices may be removed from the Figure 4-2 configuration, and two EA2111 RAM devices may be added, to provide 256 bytes of external RAM.

Select logic in Figure 4-3, defines address spaces as follows:



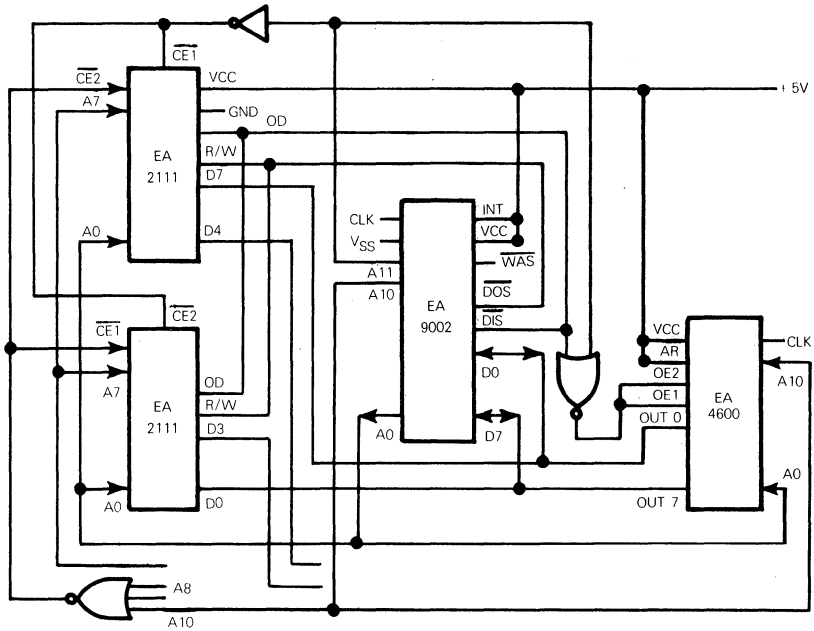


Figure 4-3 2048 Bytes Of ROM And 256 Bytes Of RAM Connected To A EA9002 CPU

The EA2111 RAM devices each provide 4 bits of 256 bytes. Each device is therefore connected to the low order or high order four bits of the data bus. Providing the two devices receive the same read, write and select logic, they will communicate 8-bit parallel data.

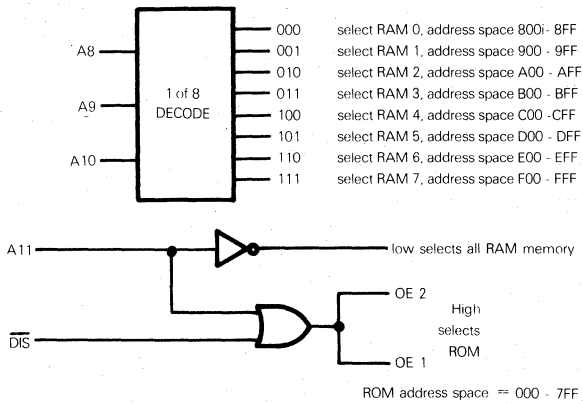
Each EA2111 device has two chip enable signals $\overline{CE1}$ and $\overline{CE2}$. These signals have been enabled as illustrated above.

Additional control signals required by the EA2111 are R/$\overline{W}$ and OD. R/$\overline{W}$ is a write enable; therefore it is connected to $\overline{DOS}$. OD is a read enable, therefore it is connected to $\overline{DIS}$.

Adding more read-write memory to the configuration illustrated in Figure 4-3 is very straightforward. Within the configuration illustrated in this figure, 16 EA2111 RAM devices may be included, providing 2048 bytes of external RAM. In order to support this number of RAM devices, two steps will be required:

1) the address, data and control lines will have to be buffered in order to support the current load.

2) Chip-select logic must decode the complete address space in order to select appropriate RAM devices. Here is one simple possibility:



SYSTEM CONFIGURATIONS WITH I/O

A minimum system will include the EA9002 CPU, a single ROM device and I/O.

I/O is very application oriented; it can be as simple as 1 to 8 flip-flop outputs, as illustrated in Figure 4-4.

The system in Figure 4-4 uses four 7474 dual D-type flip flops as output latches and one EA4700 1K x 8 ROM for program storage. The internal RAM within the 9002 is assumed sufficient for this simplified controller. This minumum system can be very useful for alarm or signalling systems which simply respond to an interrupt input by providing an output signal. Each interrupt response may terminate with a reset, to initialize the controller for the next occurance of an interrupt.

Another minimum system utilizing TTL components for I/O is shown in Figure 4-5 which uses two 74125 tri-state buffers to create a single 8-bit parallel input port, plus two 74175 quad D latches for a single 8-bit parallel output port. Assuming the internal RAM of the 9002 and a single 1K x 8 ROM (EA8308/2708 type) for program storage, the total system requires only 7 1/4 TTL packages. All required controls and strobes are obtained by gating $\overline{DIS}$ and $\overline{DOS}$ from the EA9002.

A more versatile system with TTL circuits is shown in Figure 4-6. Two 74175 flip-flops provide 8 latched outputs, while two 8233's are used to buffer two 8-bit parallel input ports. A single EA8308 PROM or EA2308 ROM is shown for program storage along with a single 74155 dual 2 of 4 decoder circuit for address decoding and chip-select function. Two 8 T 28's provide splitting and buffering of the data bus to interface with the ROM and I/O devices. Total TTL package count is seven. Additional 8233's and 74175's can be added to expand inputs and outputs. Also, there is another ROM chip select available to expand program memory to 2048 bytes (two 8308/2708). The separate $\overline{DIS}$ and $\overline{DOS}$ strobes as illustrated in this example show how four output ports can be configured with the 74155 device. Output ports C and

D, enabled by $\overline{DOS}$, occupy the same address space as do the ROM's; but the ROM only inputs information to the CPU. Thus, it only needs to be enabled by the $\overline{DIS}$ strobe.

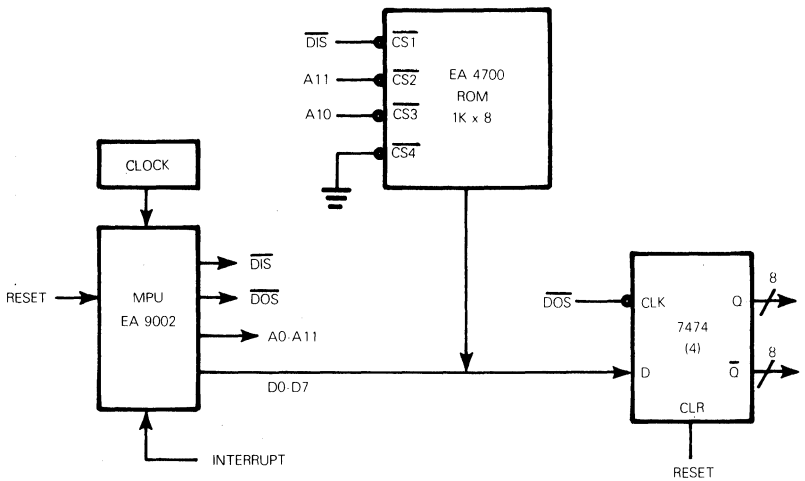


Figure 4-4 Minimum 9002 System

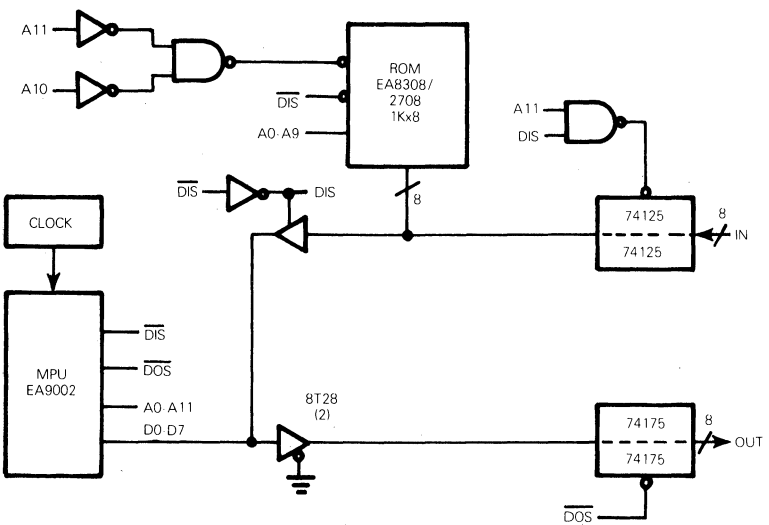


Figure 4-5 Minimum System With TTL

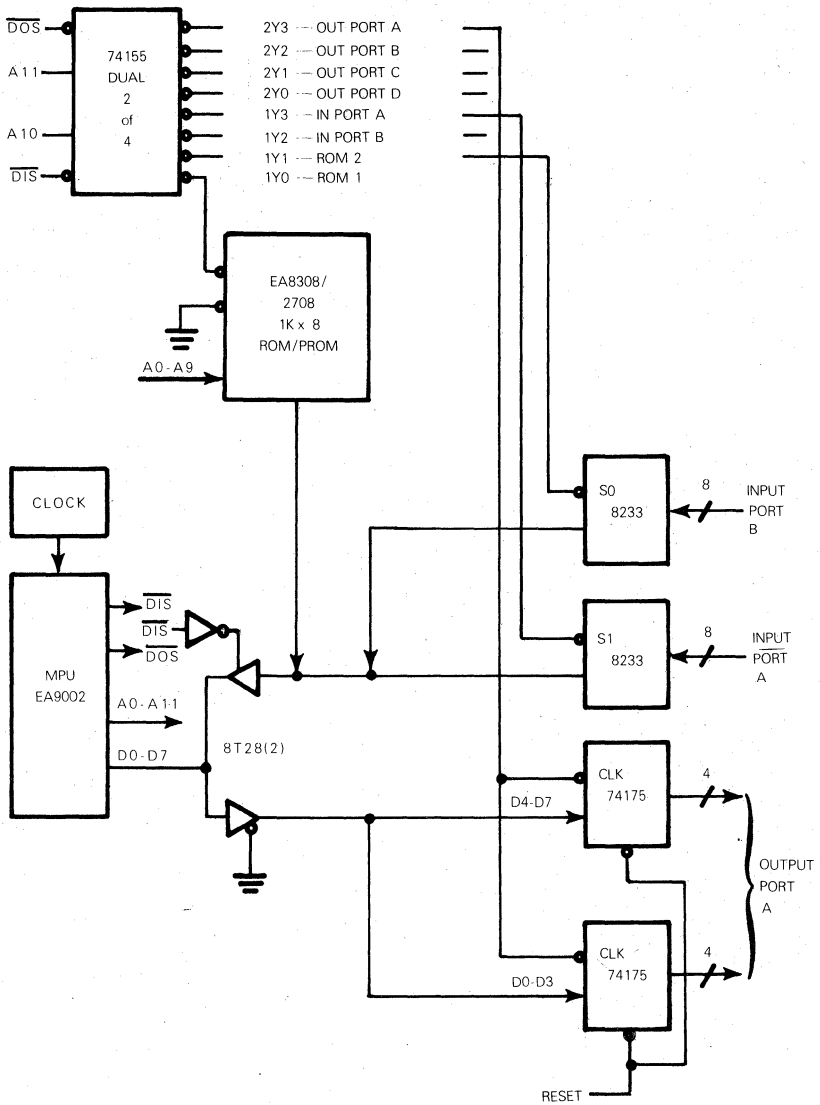


Figure 4-6 Minimum System Using TTL

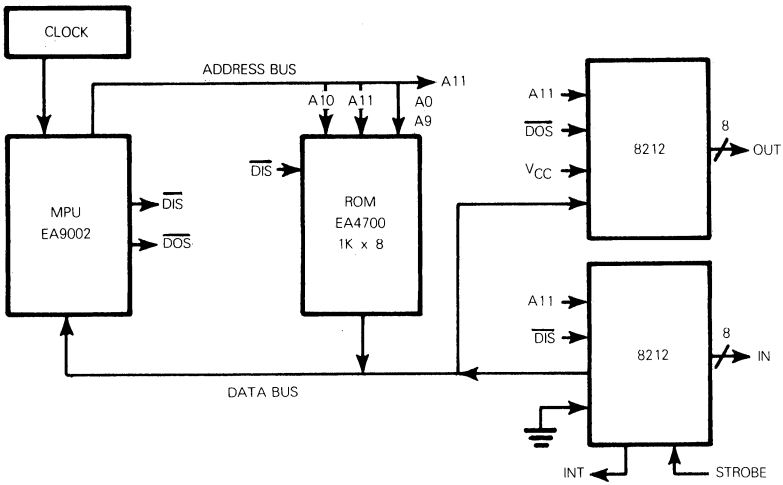


Figure 4-7 Minimum System With Intel 8212 I/O

It would be difficult to beat the minimum component parts count of the system shown in Figure 4-7. The 8212 is an 8-bit parallel buffered and latched Schottky TTL device. Again the versatility of the EA9002's $\overline{\text{DIS}}$ and $\overline{\text{DOS}}$ strobes are illustrated in that they allow both 8212's to occupy the same address space (A11 true). The EA4700 ROM, in this application, occupies memory space 000 to 300 (Hex) or the first 1 K of directly addressable memory.

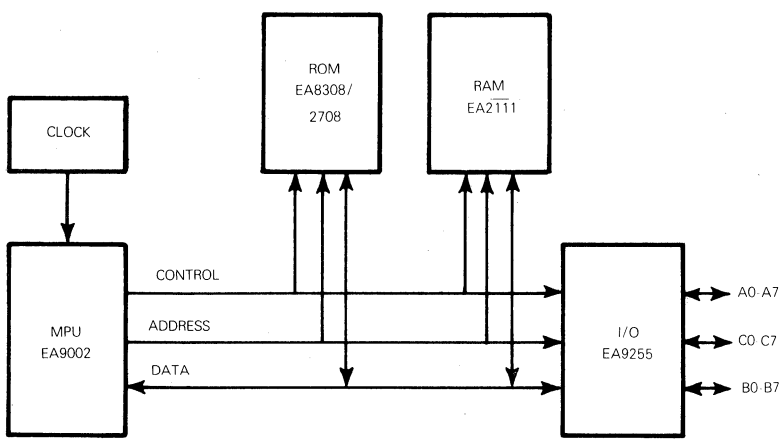


Figure 4-8 System With ROM/RAM And Programmable I/O

The system shown in Figure 4-8 makes use of the EA9255 GP I/O along with the EA8308/2708 PROM/ROM and EA2111 type 256 x 4 RAM. RAM, in addition to that provided within the 9002 along with flexibility in I/O definition is featured in this example. Component count is certainly not excessive as illustrated in Figure 4-9 which details required logic and address decoding. As is shown, only one 7406 hex inverter and one 7403 quad 2-input NAND gate is required.

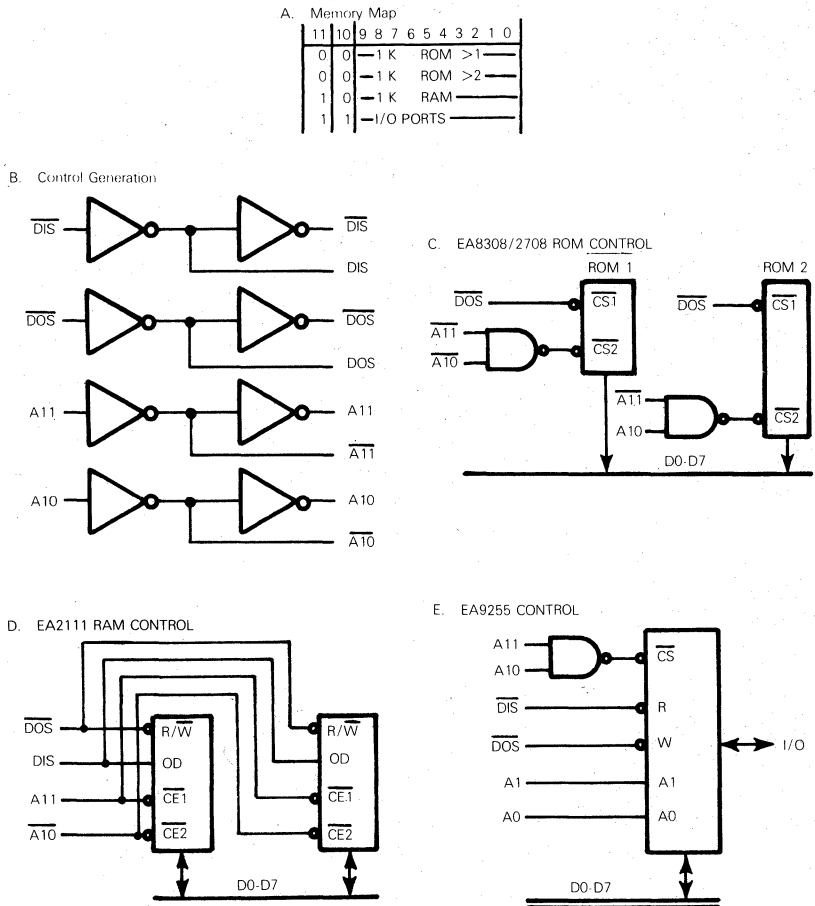


Figure 4-9 Details Of Peripheral Control With 3 TTL Package

Ability to use other manufacturers microprocessor I/O devices is shown in Figure 4-10. Here, the Motorola MC 6820 PIA device is interconnected into an EA9002 system, along with the EA4600 2K x 8 ROM. Minimum parts counts is again achieved due to the flexibility of the 9002 control signals.

A system making use of the EA9250 Keyboard/Display Controller and the EA9255 GP I/O controller is shown in Figure 4-11. The 9250 automatically scans, encodes and debounces a keyboard of up to 128 key or switch positions while, at the same time, helps a dynamic LED or gas-discharge display refreshed. This relieves the software programmer from the neccesity of writing keyboard scanning and encoding programs as well as having to periodically refresh multiplexed displays.

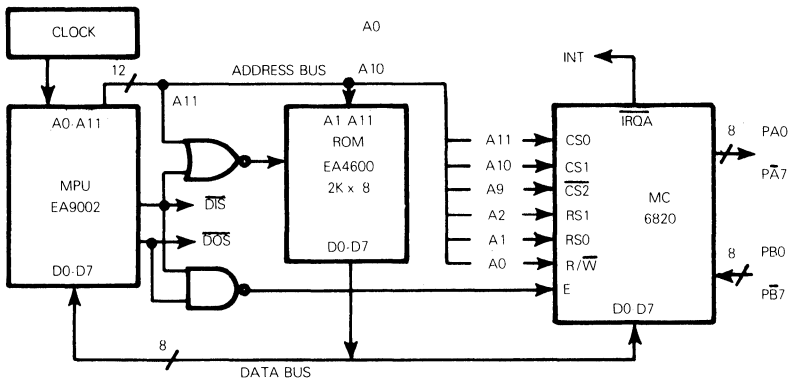


Figure 4-10 Minimum System With Motorola MC6820 PIA

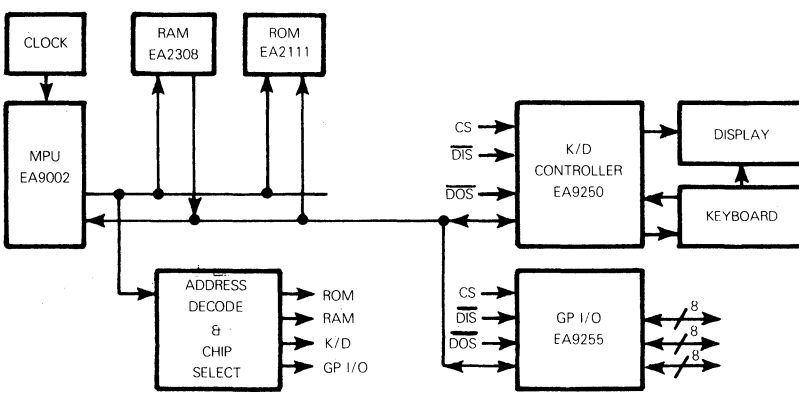


Figure 4-11 Using The EA9250 Keyboard Controller

MEMORY BANK SWITCHING

In the event that more than 4096 bytes of external memory are required, adding memory is quite straight-forward.

Now address space must be shared by memory banks and bank switching will be needed to insure that just one memory bank within any address space is selected at any time.

The simplest technique for bank switching is to use an I/O port (such as the 8212) to generate select lines which are ANDed with memory select logic. Figure 4-12 shows how an 8212 I/O port assigned addresse FFF_{16} is used to create eight select lines.

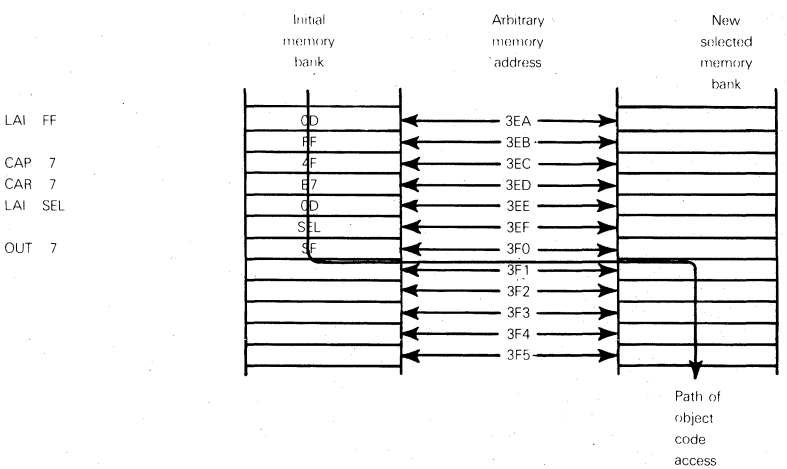
Assuming that each ouput line of the 8212 I/O port is an individual memory bank select line, eight memory banks may be present.

By combining the outputs of the 8212 I/O port, 256 memory banks sharing the address space may be implemented.

The following instruction sequence will select the appropriate memory bank:

```
LAI     FF      SET PAGE F IN R7
CAP     7
CAR     7       SET ADDRESS TO FF IN R7
LAI     SEL     LOAD MEMORY BANK SELECT INDICATOR
OUT     7       OUTPUT MEMORY BANK SELECT
```

Observe that the select lines illustrated in Figure 4-12, do not have to be used to select 4096 byte memory banks. If they are used in this fashion, then the memory bank select instructions illustrated above, when executed, will branch program logic out of one memory bank and into another memory bank at the next sequential memory location:



If considerable external data memory is needed, then the select lines illustrated in Figure 4-12, could be used to select banks of data memory only. For example, memory locations 000_{16} through $7FF_{16}$ could be reserved for program memory, while eight separate implementations of memory locations 800 through BFF_{16} could provide 8 K bytes of external data memory. Address space $C00_{16}$ through FFF_{16} remains unassigned.

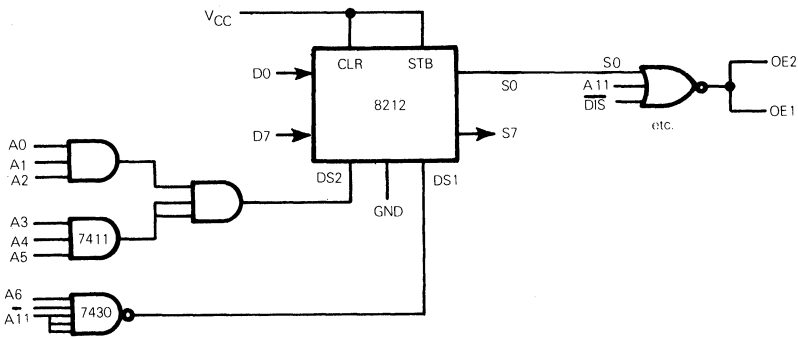


Figure 4-12 Using An 8212 I/O Port To Select Memory Banks

SEPARATING PROGRAM AND DATA MEMORY

A technique for expanding directly addressable memory of the EA9002 from 4K to 8K, with the simple addition of a few TTL components uses a 13th address line (A12) generated by control logic. This scheme is illustrated in Figure 4-13. When A12 output is false, the normal 12 address outputs are used to select an instruction related word from ROM. When A12 output is true, the 12 address outputs are used to select a data location in RAM or I/O. This allows the implementation of a system containing up to 4096 instruction codes and 4096 data RAM and I/O locations.

This technique takes advantage of the fact that the EA9002 CPU always addresses and fetches an instruction immediately following a sync output pulse from the $\overline{WAS}$ output pin. If the instruction fetched is one of the four data transfer instructions (INP, OUT, LRN, SRN) then and only then is a data transfer to memory or I/O to take place. Otherwise, the instruction performs operations internal to the CPU only — in which case there is no communication with data memory or I/O devices.

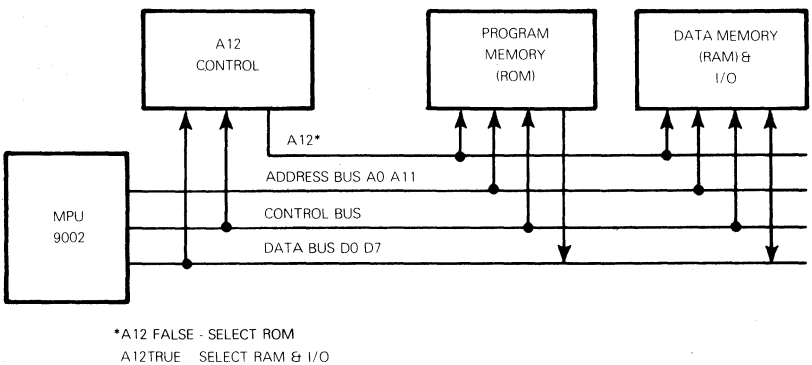


Figure 4-13 8K Memory System

The four I/O instructions binary op codes, as they appear on the data bus, are as follows:

Instruction	Op Code							
	MSB							LSB
	D7	D6	D5	D4	D3	D2	D1	D0
INP	0	1	0	1	0	x	x	x
OUT	0	1	0	1	1	x	x	x
LRN	1	1	1	0	0	x	x	x
SRN	1	1	1	0	1	x	x	x

Op codes are described in Chapter 5.

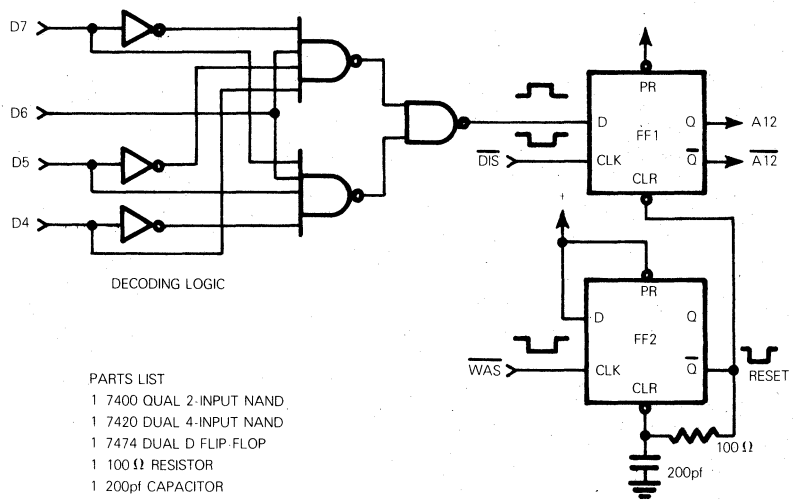


Figure 4-14 Control Generation

The three least significant bits (D2, D1, & D0) are used to designate one of the 8 GP registers. Bit D3 is also a "don't care" condition for decoding purposes since it selects between INP and OUT or LRN and SRN. Therefore, it is only necessary to decode the four most significant bits to ascertain that the instruction is one of the four possible data transfer codes. An implementation of the logic equation

$$\overline{D7} \cdot D6 \cdot \overline{D5} \cdot D4 + D7 \cdot D6 \cdot D5 \cdot \overline{D4}$$

will select the unique I/O instruction op codes from the total op code field. We must test for this condition immediately following a sync output pulse. Thus, the $\overline{WAS}$ sync pulse will initialize a flip-flop to the 0 (false) state, while the $\overline{DIS}$ (Data In) strobe will clock the flip-flop into the 1 (true) state, providing the test condition is true.

Figure 4-14 illustrates an implementation of logic to create address bit A12.

Figure 4-15 illustrates the timing associated with Figures 4-13 and 4-14. During T1 and T2 of an instruction cycle, an instruction is addressed via the address bus; the op code is transferred to the CPU via the data bus. If this instruction is not one of the four data transfer instructions, then no output occurs from the instruction decoder of Figure 4-13; the control flip-flop is not set, A12 remains low and all addresses remain directed to the program field of memory.

If the instruction is an INP, OUT, LRN or SRN, the $\overline{\text{DIS}}$ strobe sets the control flip-flop, and thus A12, to the 1 state. The control flip-flop will remain in the 1 state until reset by the output of the reset flip-flop shown in Figure 4-14. The reset flip-flop acts as a one-shot; it provides a delay beyond the end of the $\overline{\text{WAS}}$ sync pulse to ensure that the control flip-flop remains reset until the beginning of the next T1 period.

Implementation of the logic shown in Figure 4-14 is not necessarily the lowest cost approach. It is shown as a technique which has been tested and will work. The significant events to consider when implementing this scheme are that A12 should always be low at the beginning of a CPU cycle which is signaled by the $\overline{\text{WAS}}$ sync strobe. After T2 time, if A12 has been set, it must remain set until the required data transfer has taken place. $\overline{\text{WAS}}$ sync occurs before the ending of a $\overline{\text{DIS}}$ or $\overline{\text{DOS}}$ strobe, therefore, it is necessary to delay the resetting of A12 to avoid shortening the $\overline{\text{DIS}}$ or $\overline{\text{DOS}}$ cycle.

If data on the data bus coincides with an I/O instruction code and this occurs after A12 has been set, the most that can happen is that A12 toggles to the zero state — but it is supposed to go to the zero state at this time anyway. If A12 is in the zero state and gets toggled to the one state by spurious data, it will be immediately reset as a result of the sync pulse.

Additional considerations may be required in the use of this memory bank select technique if the $\overline{\text{WAS}}$ pin is to be pulled down by external networks to initiate a WAIT period. It may be necessary to isolate the clocking of the reset flip-flop from this condition.

INTERRUPT PROCESSING

Most microcomputer applications have their needs met by a single interrupt request line. Only in special circumstances will multiple interrupt, single CPU logic configurations be justifiable.

Within the minicomputer industry, interrupts are frequently used to share the expensive and underused capabilities of a Central Processing Unit for a variety of operations. Microcomputer Central Processing Units are so inexpensive that any attempt to share them, simply as a means of reducing the number of CPU's within a system, is almost certain to be economically unsound.

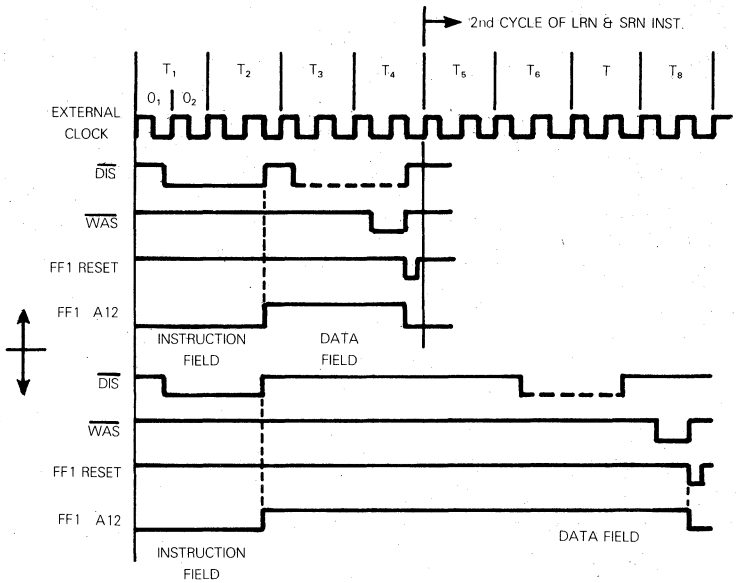


Figure 4-15 One Cycle And Two Cycle Timing

Nevertheless, there will be circumstances in which multiple interrupt configurations are both economical and viable, since to a microcomputer, interrupts signal asychronous events. A multiple interrupt, single CPU microcomputer configuration would economically support slow asychronous event sequences.

A multiple interrupt microcomputer system should handle interrupts serially; once one interrupt has been acknowledged, all other interrupts should wait until the acknowledged interrupt is serviced. In other words, interrupt priorities should only be arbitrated if the actual interrupt requests occur simultaneously from two or more external sources.

When an EA9002 configuration includes a single external interrupt (in addition to the Reset), then implementing interrupt logic becomes trivially simple. Providing external logic creates interrupts or reset request signals with the appropriate timing, as described in Chapter 3, **no other external logic is required specifically to support the interrupt.** Of course, subsequent I/O operations may be necessary following the interrupt acknowledge, but these are normal I/O operations.

In a microcomputer configuration where multiple interrupts are justified, an effective way of handling multiple interrupts is using a daisy chain, as illustrated in Figure 4-16. Every external logic source capable of requesting an interrupt has its own interrupt request line, illustrated in Figure 4-16 by INT0, INT1, INT2, etc.. These interrupt requests are active low. By ANDing all interrupt requests together, a master interrupt request $\overline{INT}$, is created for transmittal to the CPU.

When acknowledging the interrupt, in order to support the logic illustrated in Figure 4-16, the CPU must output a master acknowledge, shown as IACK. IACK is negative true. If IACK arrives at the first NAND gate and is matched by INT0 low, then the first NAND gate will generate a high output, which will cause all subsequent NAND gates to generate low outputs. Thus, IACK0 will become a high true interrupt acknowledge to the first device in the daisy chain.

If INT0 is high, then the first NAND gate will propagate a low output to the second NAND gate. This propagation will continue down the daisy chain until the first low output from one NAND gate is matched by a low interrupt request INTX. That NAND gate will generate a high output, which becomes a high interrupt acknowledge IACKX — and simultaneously forces all subsequent NAND gates to output low.

In summary, therefore, IACKX will be output high by the NAND gate which receives INTN low when the master interrupt acknowledge IACK is output low by the CPU.

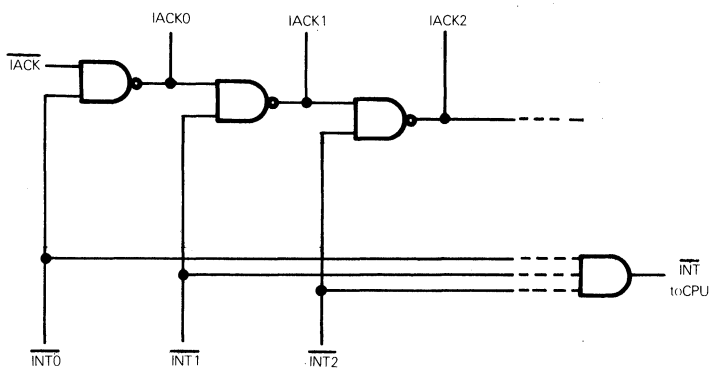


Figure 4-16 An Interrupt Request Daisy Chain

INTERRUPT ACKNOWLEDGE

There are numerous ways in which the CPU can output IACK low. Figure 4-17 illustrates how data bus line D0, in conjunction with memory address FF0, can create IACK. The logic in Figure 4-17 requires the EA9002 CPU to respond to an interrupt by executing the following instruction sequence:

```
*INTERRUPT RESPONSE MUST BE ORIGINED AT MEMORY LOCATION 2
   ORG  2
   LAI  0F
   CAP  7
   LRI  7,F0      ESTABLISH ADDRESS FF0 IN R7
   CLA            OUTPUT 0 ON D0 LINE OF DATA BUSS AS
   OUT  7         INTERRUPT ACKNOWLEDGE
```

Observe that so long as the low order accumulator bit is 0 when data is output to memory location FF0, it does not matter what values the other Accumulator bits have.

An alternate method of generating an interrupt acknowledge is by decoding the Address bus. Recall that when an interrupt is acknowledged, pro-

gram execution branches to memory location 002_{16}. You may therfore decode this single value off the address bus to generate an interrupt acknowledge signal. This scheme relies entirely on external logic to generate IACK. No instructions are executed for this purpose.

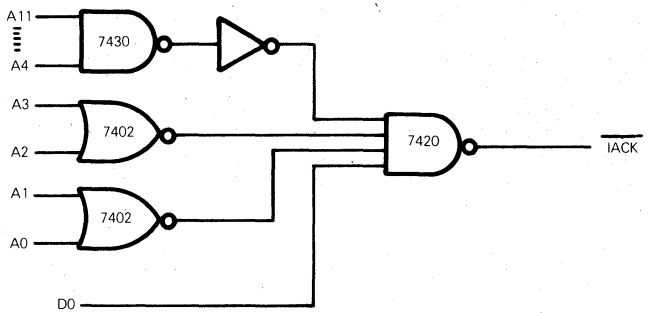


Figure 4-17 Logic To Transmit IACK Low If A11-A0 Is FF0 And D0 Is 1

The external device whose interrupt has been acknowledged is informed of this circumstance by its interrupt acknowledge signal IACKX being high. There are many ways in which this high signal can be used by the CPU to determine which interrupting external logic must be serviced. Here are two possibilities:

1) The individual IACKX signals can be input to an 8212 I/O port, or to an I/O port of an 8255 type parallel interface device. By inputing the contents of this I/O port, the interrupt service program can determine which device is requesting an interrupt.

2) The interrupt service program associated with each external logic source can be implemented on a separate ROM device sharing the same memory space.

   Suppose each external source has a IK byte ROM in which its interrupt service program has been implemented. Let us suppose all of these IK byte ROMs share the address space 800_{16} through BFF_{16}. If there are eight external logic sources which can request interrupt service, there will be eight ROM devices, all of which are selected by the address range 800_{16} through BFF_{16}. However, all of the ROM selects will be tied to the interrupt acknowledge signals, thus only the ROM for acknowledged logic will indeed be selected. Of course, the last instruction executed within the interrupt service program, must be a jump back to some instruction implemented in the memory range 000_{16} through $7FF_{16}$, since this address range is not duplicated.

   The logic needed to select alternate ROMs has been described earlier in this Chapter in conjunction with memory bank switching.

Chapter 5
THE INSTRUCTION SET

In this chapter we are going to identify the operations performed by individual instructions of the EA9002 instruction set. This is an instruction look-up chapter, which means that operations performed by instructions are described individually, isolated from program flow considerations. Programming techniques and efficient use of the 9002 instruction set are treated separately in Chapters 6, 7 and 8.

Because this is an instruction look-up chapter, instructions are listed in alphabetic order of instruction mnemonic.

Once you are familiar with the EA9002 instruction set, use the instruction set summaries provided in Appendix A.

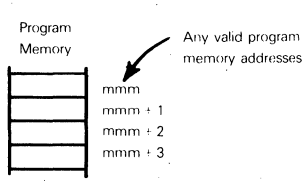
A generalized format has been adopted in this chapter to describe all instructions.

Each instruction begins with a graphic representation of active accumulators, registers, memory locations and statuses.

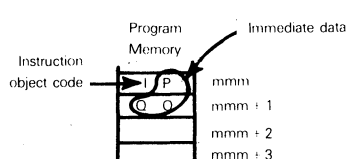
ABBREVIATIONS

A number of abbreviations have been adopted in this chapter, and they are summarized next.

Lower case 'm' is used to represent hexadecimal digits of program memory addresses:

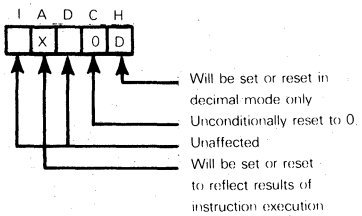


Upper case 'P' and 'Q' are used to represent hexadecimal digits of immediate data:

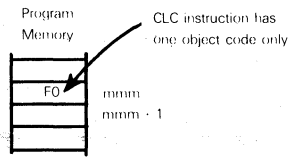


Upper case 'J' and 'K' are used to represent hexadecimal digits stored in the accumulator, or the low-order eight bits of general purpose registers:

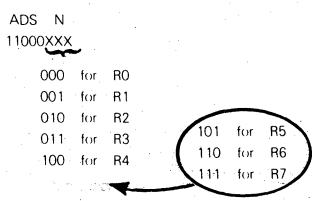
If a status flag may either be set or reset by an instruction, an X is placed in the appropriate status box. A D is placed in a status box if the status is modified in decimal mode only. An 0 is placed in the status box if the status flag is unconditionally reset. A blank means that the status is in no way modified by execution of the instruction:



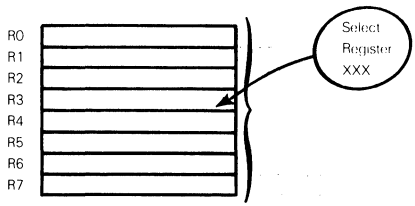
Instruction object codes are represented by two hexadecimal digits wherever an instruction has only one version:



Many 9002 instructions are register-dependent; the low-order three object code bits specify one of eight registers. Object codes for these instructions are shown in binary format. Here is a general example:

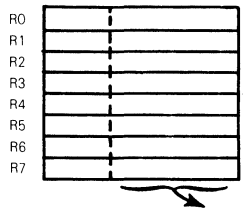


Recall that general-purpose registers are sometimes treated as single 12-bit units. A 12-bit unit being selected by an instruction object code is illustrated as follows:

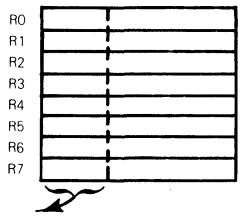


At other times general-purpose registers are treated as separate 4-bit and 8-bit units.

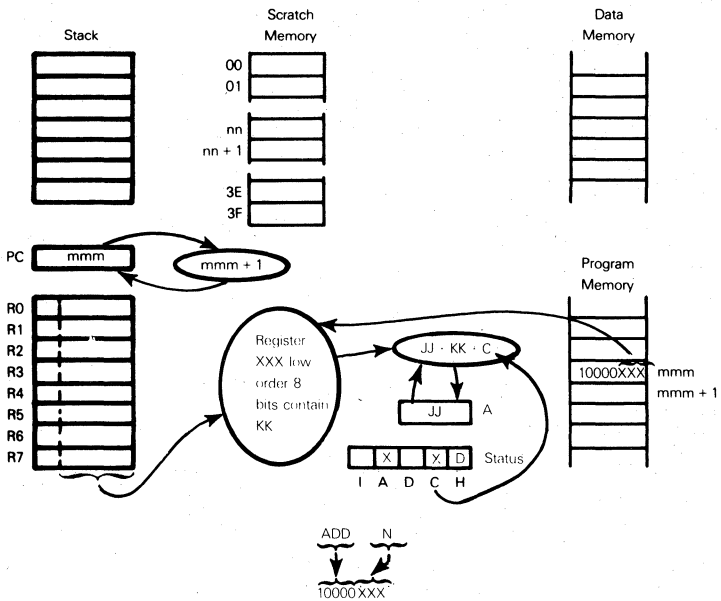
Selection of a register's 8 low order bits is identified as follows:



Selection of a register's high-order 4 bits is represented as follows:



ADD— ADD REGISTER TO ACCUMULATOR WITH CARRY



To the Accumulator add the Carry status and the low order eight bits of register N. If the D status is 0, the Accumulator and Register contents are treated as 8-bit binary data. If the D status is 1, the Accumulator and Register contents are each treated as a pair of BCD digits.

Suppose the Accumulator contains 73_{16}, Register R5 contains 347_{16} and the Carry status is 1; after execution of the instruction sequence:

```
SEB     Set binary mode

ADD     5
```

The Accumulator will contain BB_{16}:

```
73 = 01110011
47 = 01000111
C =         1
    1011 1011  <-- Non zero result so A = 1

No carry so C = 0

No carry so H = 0 (Decimal mode only)
```

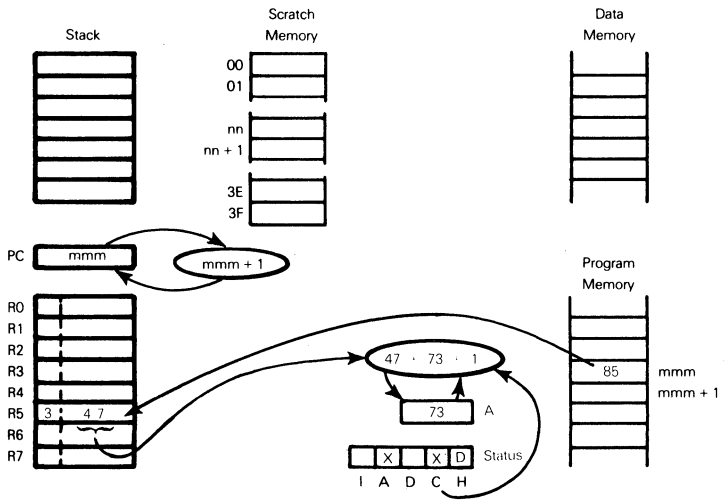
Now look at decimal addition:

```
 7  3
 4  7
    1
 2  1  <-- Non zero result, so A = 1

Carry sets C to 1

Carry sets H to 1
```

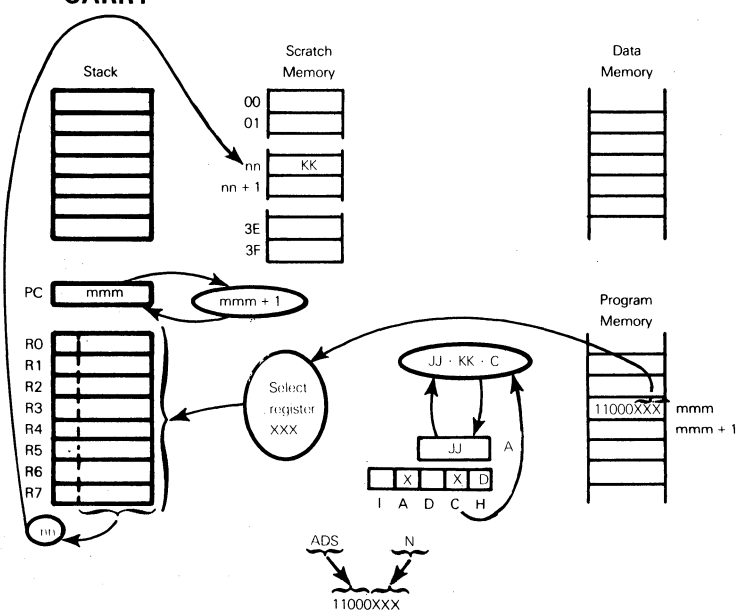
Decimal addition will result from execution of the sequence:

```
SED        Set decimal mode
ADD 5
```

This is what happens when the ADD 5 instruction is executed:



ADS — ADD SCRATCH MEMORY TO ACCUMULATOR WITH CARRY

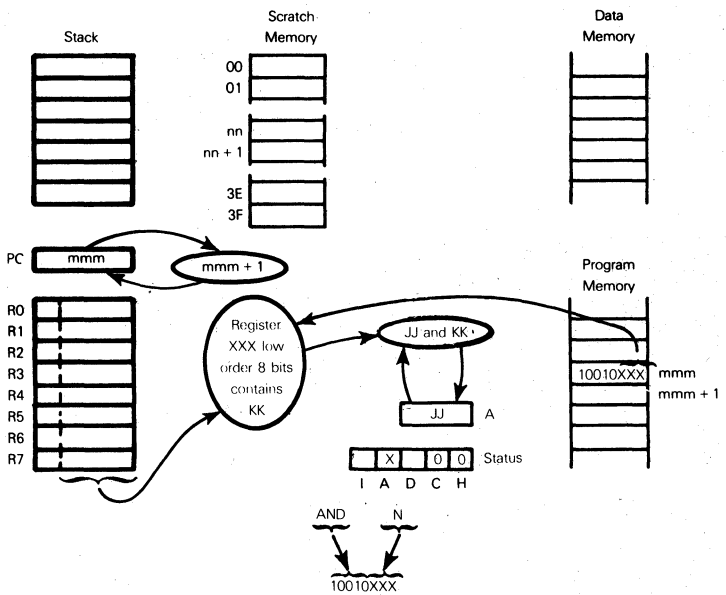


To the Accumulator add the Carry status and contents of the scratch memory byte addressed by the low order six bits of Register N.

The ADS instruction is identical to the ADD instruction except for the source of the addend byte.

For a discussion of scratch memory addressing see the RDS instruction.

AND — AND REGISTER WITH ACCUMULATOR



Logically AND the Accumulator with the low order eight bits of Register N.

Suppose the Accumulator contains $3E_{16}$ and Register 3 contains $2F3_{16}$; after execution of the instruction

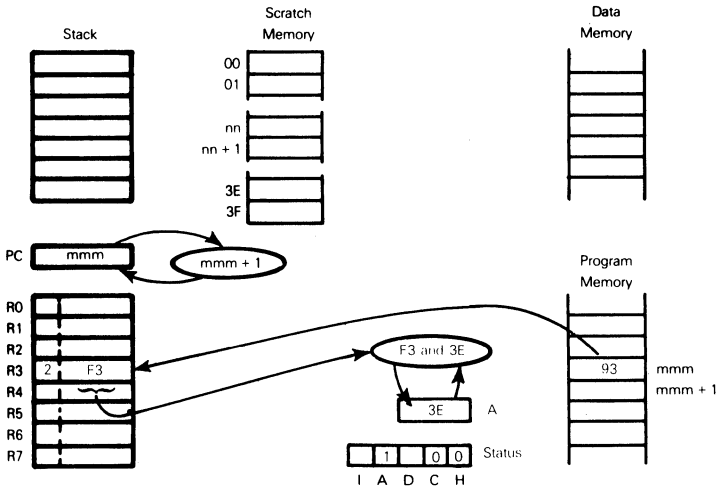
AND 3

the Accumulator will contain 32_{16}:

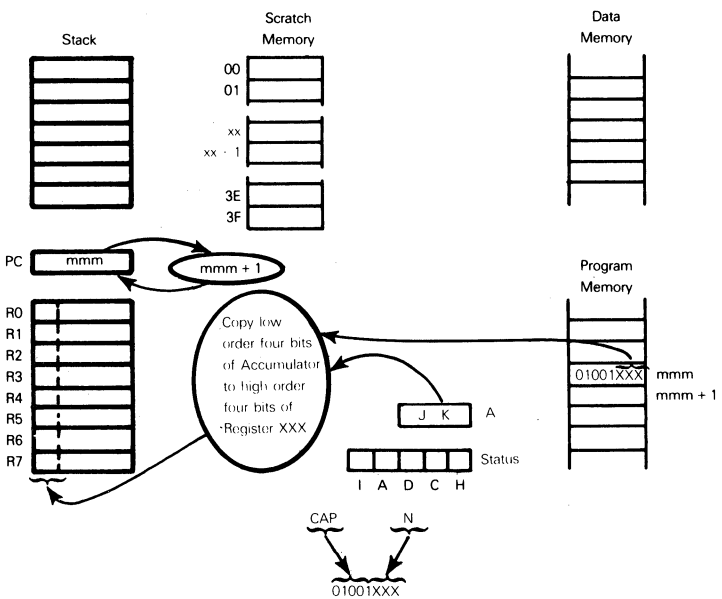
```
3E  =  00111110
F3  =  11110011
AND =  00110010
```

The C and H statuses are unconditionally reset to 0. The A status is set to 1 since the result in the Accumulator is not zero.

This is what happens:



CAP — COPY ACCUMULATOR TO PAGE

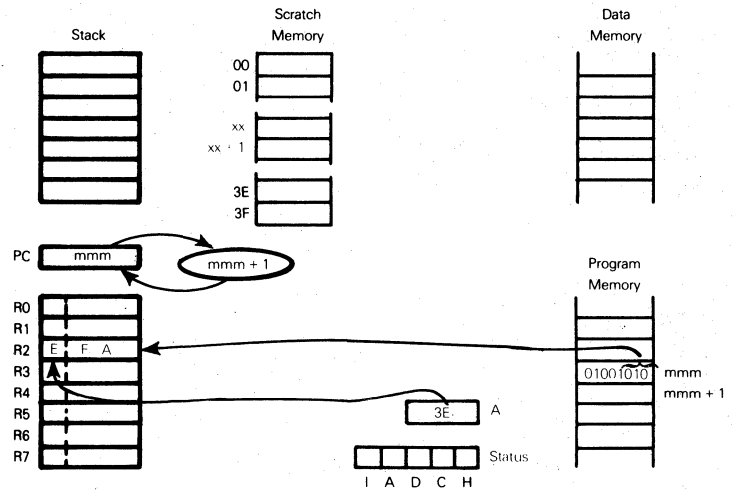


Copy the low order four Accumulator bits into the high order four (Page) bits of Register N.

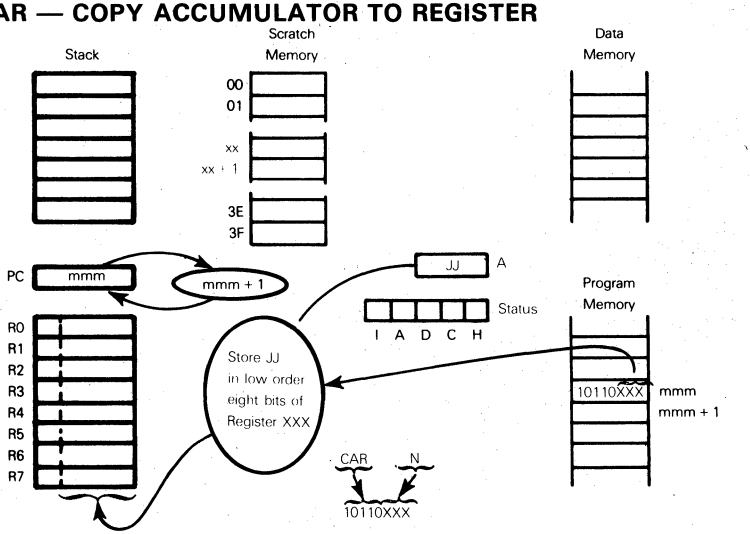
Suppose the Accumulator contains $3E_{16}$ and Register R2 contains $4FA_{16}$; after the instruction:

CAP 2

is executed, the Accumulator will contain $3E_{16}$, while Register R2 contains EFA_{16}:



CAR — COPY ACCUMULATOR TO REGISTER

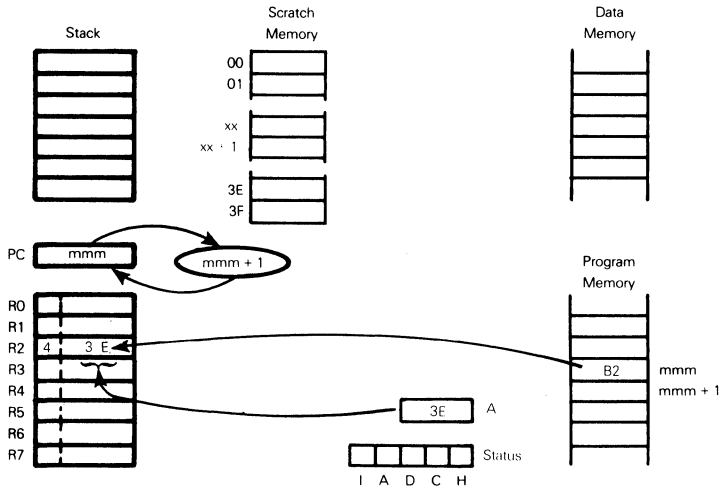


Store Accumulator contents in the low order eight bits of Register N.

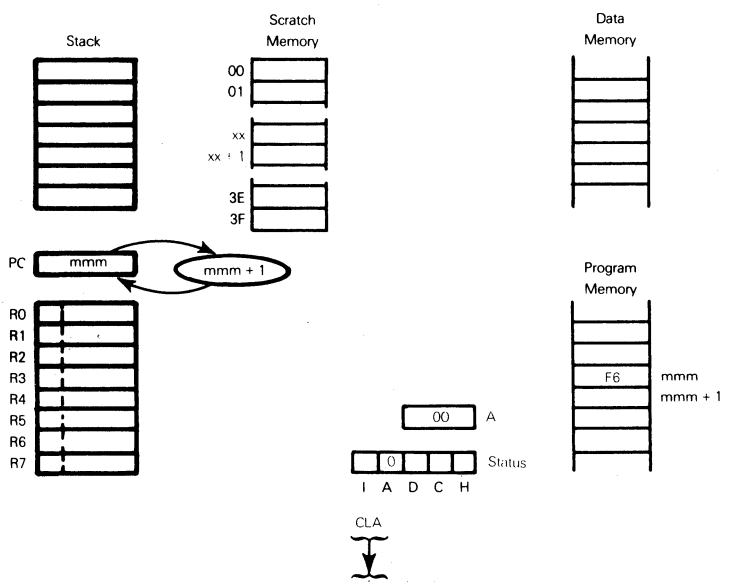
Suppose the Accumulator contains $3E_{16}$ and Register R2 contains $4FA_{16}$; after the instruction:

CAR 2

is executed, Register R2 will contain $43E_{16}$:

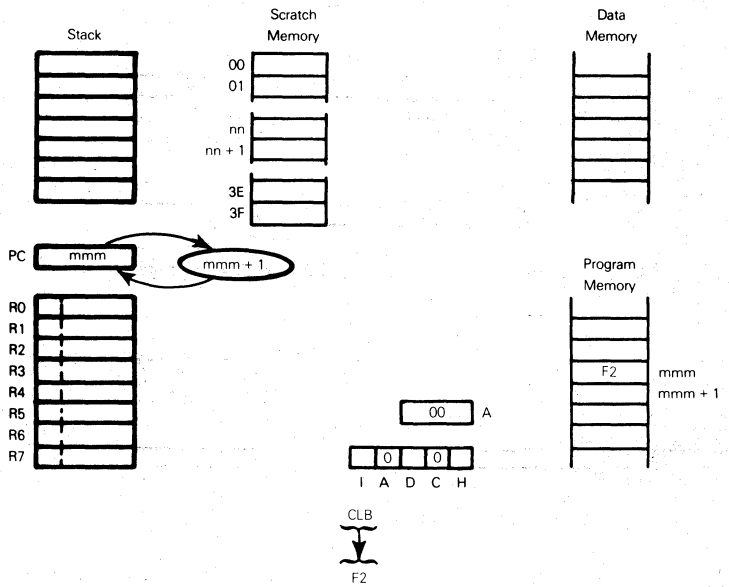


CLA — CLEAR THE ACCUMULATOR



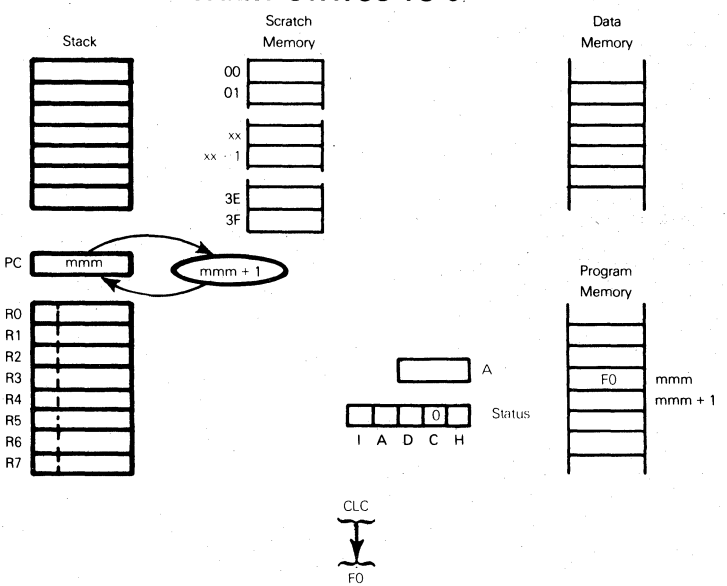
Unconditionally reset the Accumulator contents to 0.

CLB — CLEAR THE ACCUMULATOR AND CARRY STATUS



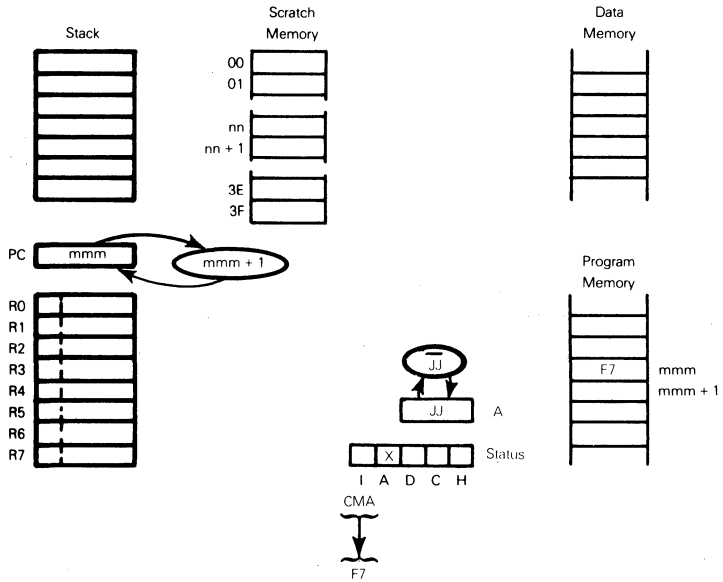
Unconditionally reset the Accumulator contents and the Carry status to 0.

CLC — RESET THE CARRY STATUS TO 0



Unconditionally clear the Carry status.

CMA — COMPLEMENT ACCUMULATOR



Ones complement the Accumulator contents.

Suppose the Accumulator contains $3E_{16}$. After execution of the instruction:

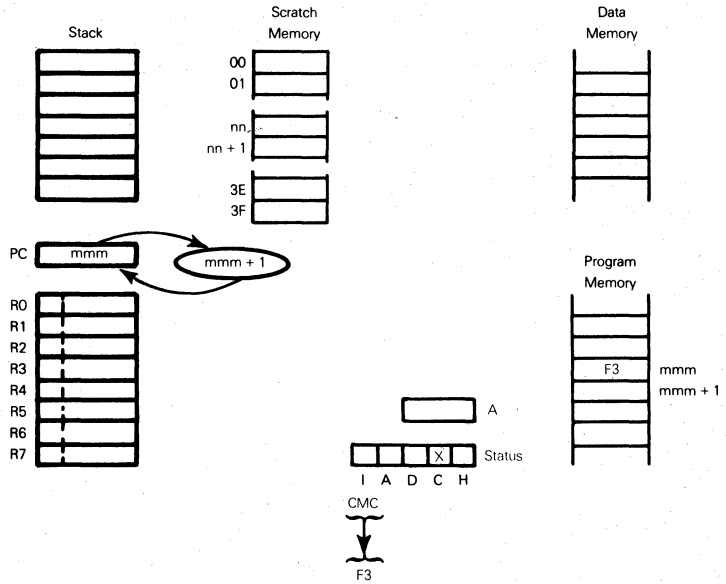
CMA

the Accumulator will contain $C1_{16}$:

3E = 00111110
ones complement = 11000001

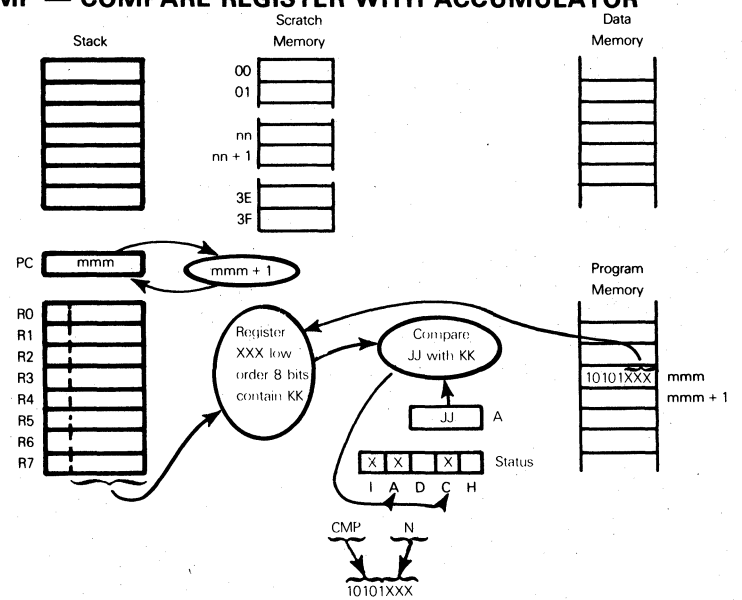
The A status will be set to 1 since the result is not zero.

CMC — COMPLEMENT THE CARRY STATUS



Unconditionally complement the Carry status.

CMP — COMPARE REGISTER WITH ACCUMULATOR

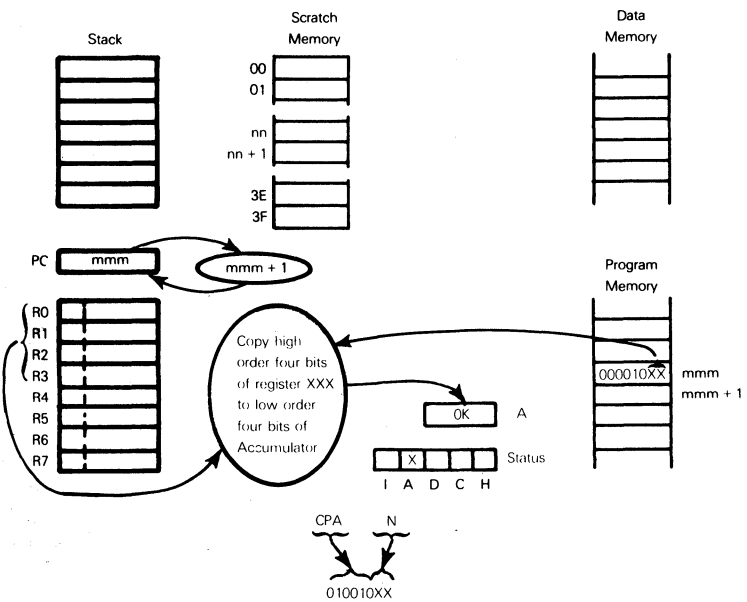


Compare the Accumulator contents with the contents of Register N's low order eight bits. Neither the Accumulator nor Register N contents are modified, but the C and A statuses reflect the result of the compare as follows:

	C	A	Test Condition
A=R	0	0	A=0
A<R	1	1	C=1
A>R	0	1	C=0 and A=1
A≥R	0	0 or 1	C=0
A≤R	0 or 1	0 or 1	C=1 or A=0
A≠R	0 or 1	1	A=1

Interrupts are disabled during execution of the CMP instruction and the next sequential instruction. However the interrupt status flag itself is not changed. This prevents an interrupt from dividing a CMP instruction's execution from execution of the Jump on condition instruction which usually follows directly.

CPA — COPY PAGE TO ACCUMULATOR

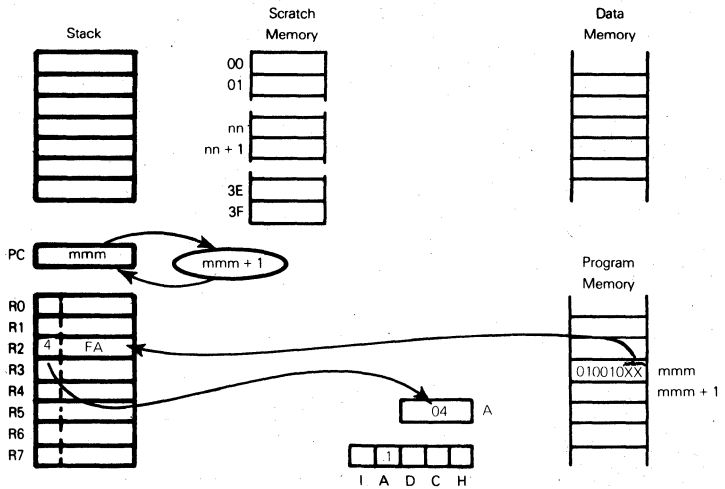


Copy the high order four (page) bits of Register R0, R1, R2 or R3 to low order four Accumulator bits. Clear four high order Accumulator bits.

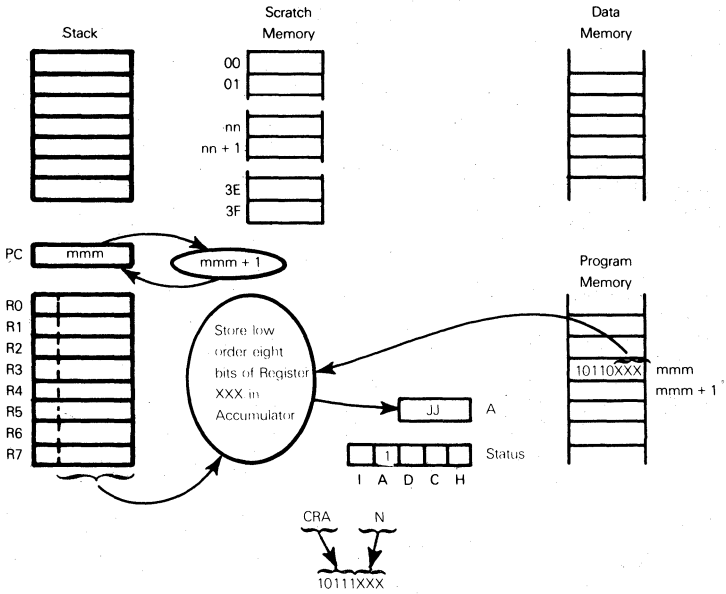
Suppose the Accumulator contains $3E_{16}$ and Register R2 contains $4FA_{16}$; after the instruction:

CPA 2

is executed, the Accumulator will contain 04_{16}, while Register R2 contains $4FA_{16}$:



CRA — COPY REGISTER TO ACCUMULATOR

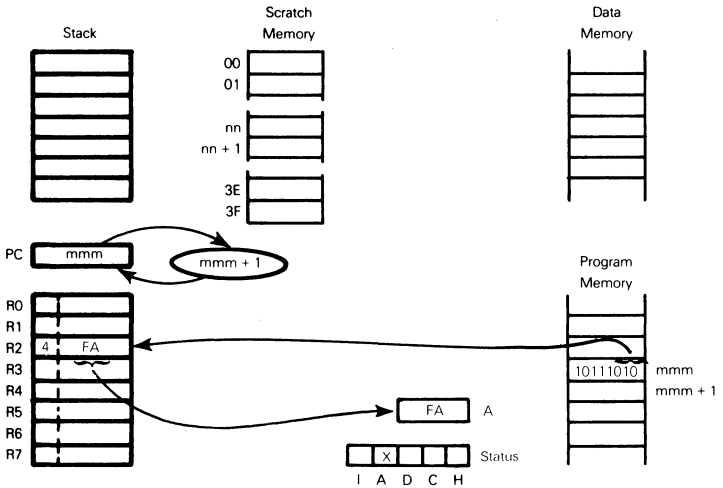


Store contents of low order eight bits of Register N in Accumulator

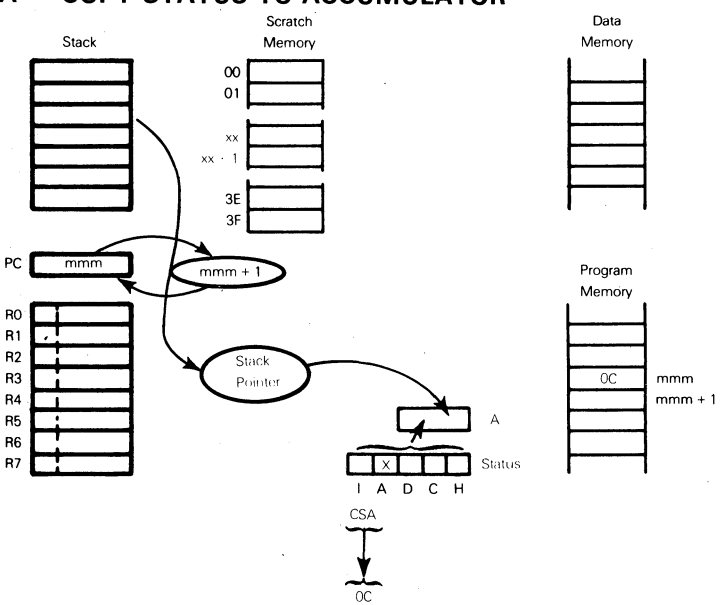
Suppose the Accumulator contains $3E_{16}$ and Register R2 contains $4FA_{16}$; after the instruction:

CRA 2

is executed, the Accumulator will contain FA_{16}:

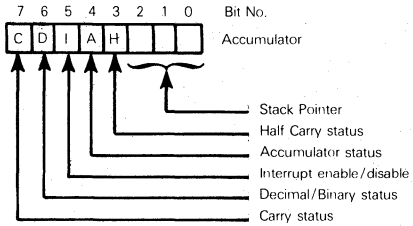


CSA — COPY STATUS TO ACCUMULATOR



Copy status flags and Stack Pointer to the Accumulator.

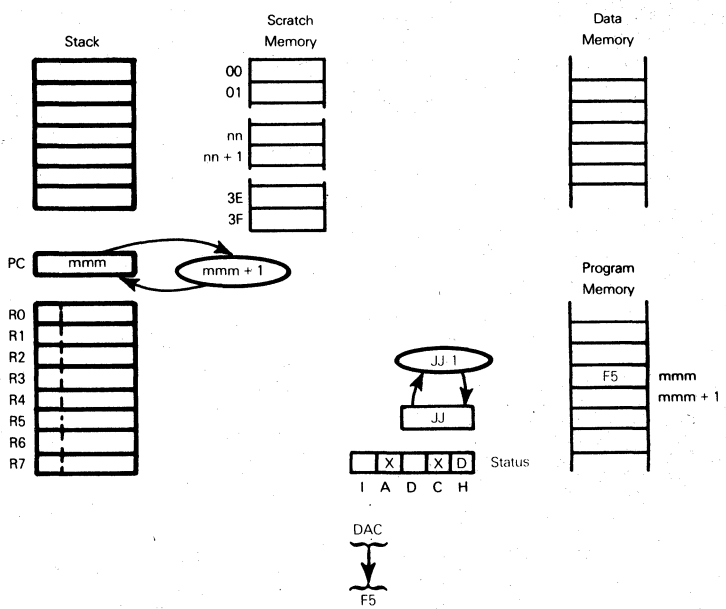
This is how Accumulator bits are interpreted following execution of the CSA instruction:



The statuses are straightforward and easy to understand.

The Stack Pointer identifies the level to which the stack has been pushed. For example, a value of 101 in the Stack Pointer stipulates that five more subroutine calls than returns have been executed; that means subroutines are currently nested to a level of 5.

DAC — DECREMENT THE ACCUMULATOR



Decrement the Accumulator contents. If the D status is 0, the Accumulator contents are treated as an 8-bit binary number. If the D status is 1, the Accumulator contents are treated as two BCD digits.

Suppose the Accumulator contains 73_{16}. Execution of the instruction:

DAC

will decrement the Accumulator to 72_{16} in either binary or decimal mode.

Now suppose the Accumulator contains 80_{16}. The instruction sequence:

```
SEB     Set binary mode
DAC     Decrement Accumulator
```

leaves $7F_{16}$ in the Accumulator; the C status is 0. However, the instruction sequence:

```
SED     Set decimal mode
DAC     Decrement Accumulator
```

leaves 79_{16} in the Accumulator, 1 in the H status and 0 in the C status.

Now suppose the Accumulator initially contains 00_{16}. The instruction sequence:

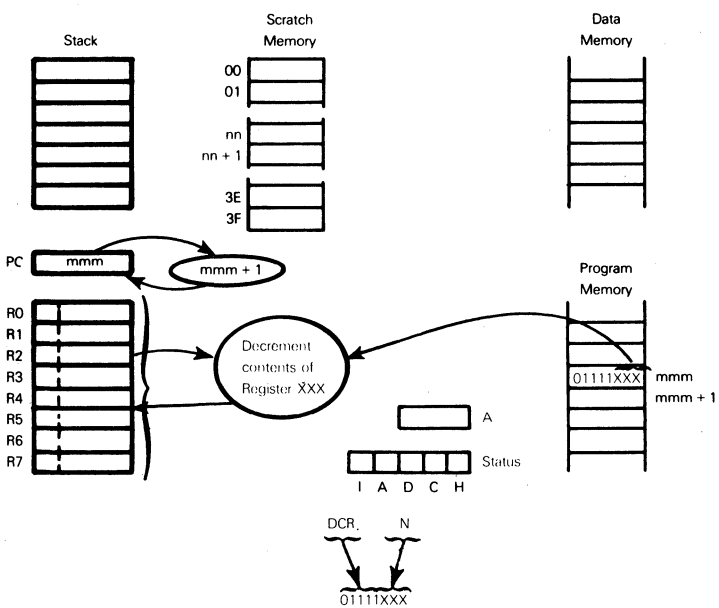
```
SEB     Set binary mode
DAC     Decrement Accumulator
```

leaves FF_{16} in the Accumulator and 1 in the C status. The instruction sequence:

```
SED     Set decimal mode
DAC     Decrement Accumulator
```

leaves 99_{16} in the Accumulator and 1 in both C and H statuses.

DCR — DECREMENT REGISTER



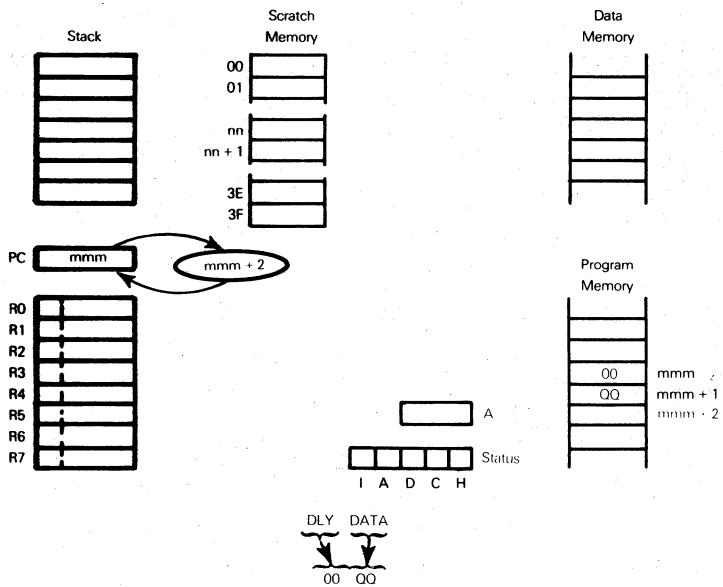
Decrement the twelve bit contents of Register N.

Suppose Register R3 contains $23A_{16}$; after execution of the instruction:

```
DCR 3
```

Register 3 will contain 239_{16}. Had Register R3 contained 300_{16}, it would now contain $2FF_{16}$; had the register contained 000_{16}, it would now contain FFF_{16}.

DLY — DELAY TWO CYCLES



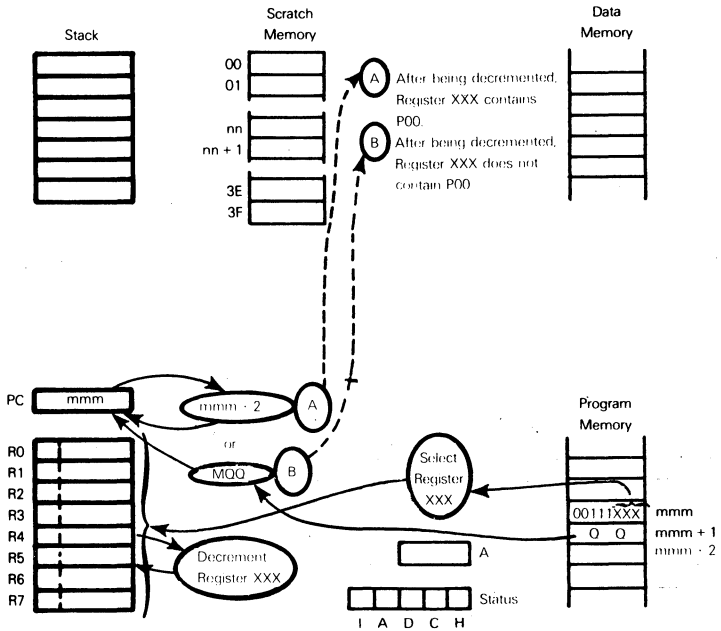
Nothing happens when this instruction is executed, but the Program Counter is incremented by 2, skipping the byte that follows the DLY object code.

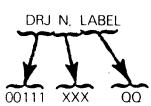
The DLY instruction may be used to implement skip logic: For example, all addition and subtraction instructions include the Carry status. You can clear the Carry status optionally as follows:

```
LABEL   DLY     F0
        ADD     3
```

F0 is the CLC instruction object code; normally this object code will be skipped by the DLY instruction. By jumping to LABEL + 1, you can clear the Carry status before executing the ADD instruction.

DRJ — DECREMENT REGISTER AND JUMP

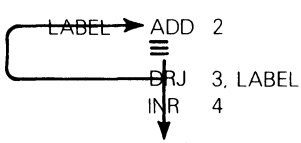




Decrement the 12 bit contents of Register N. After decrementing, test the low order eight bits; if they are all zero, program execution continues with the next sequential instruction; if they are not all zero, program execution branches to the instruction on the same page identified by the label LABEL.

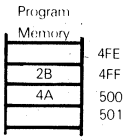
For a discussion of how the label LABEL is handled when the DRJ instruction object code resides at, or across a page boundary, see the JCY instruction.

Consider the instruction sequence:



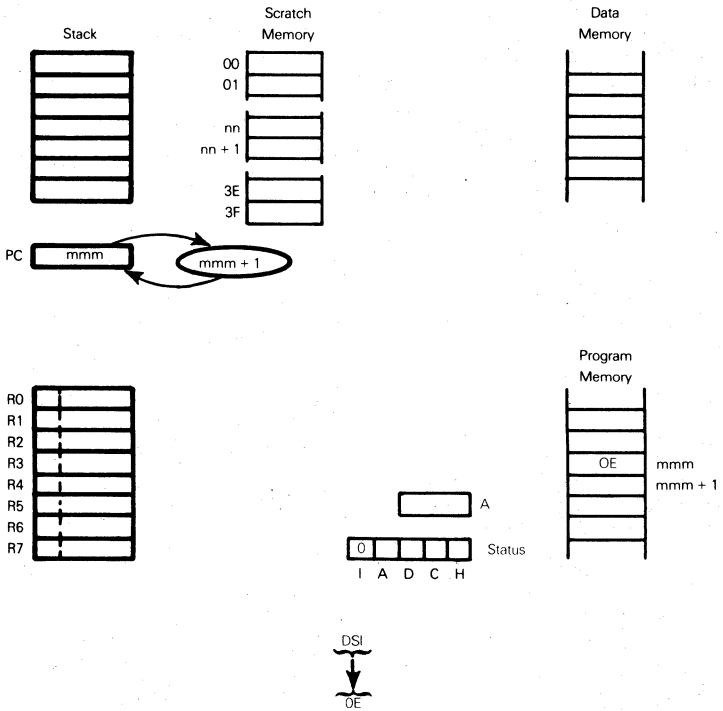
The DRJ instruction specifies Register 3 as the register to be decremented. After decrementing, if the low order eight bits of Register 3 are all 0, then the INR instruction will be executed; otherwise the ADD instruction will be executed.

With reference to case B in the instruction illustration, note that if the DRJ instruction's object code ends at, or lies across a page boundary, then the branch will occur into the next page. For example, suppose object code resides in these program memory bytes:



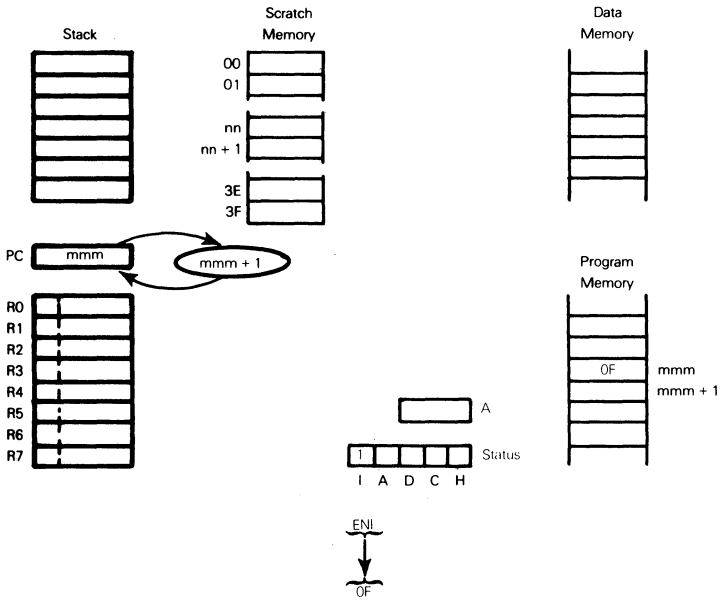
Register 3 is to be decremented; if it decrements to P00, then a branch to memory location $54A_{16}$ will occur.

DSI — DISABLE INTERRUPTS



Reset the Interrupt status to 0; this disables all interrupts. Interrupts remain disabled until an ENI instruction is executed.

ENI — ENABLE INTERRUPTS



Set the Interrupt status to 0; this enables interrupts after execution of the next sequential instruction.

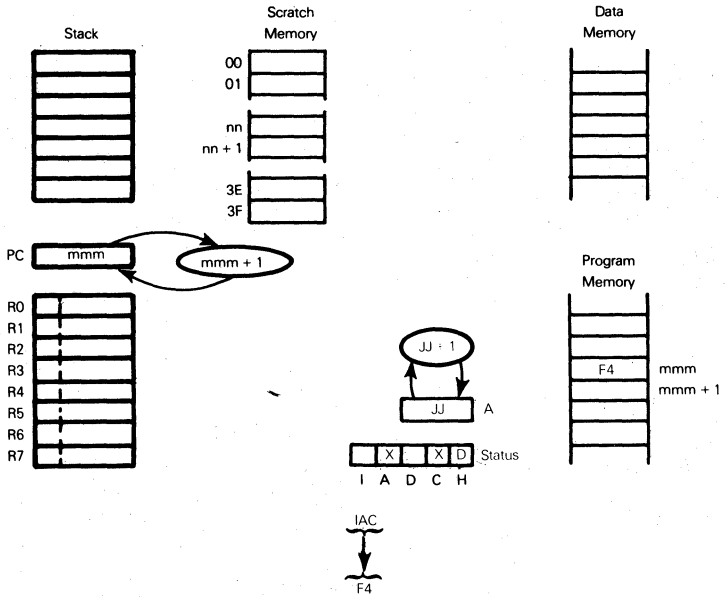
This is how an interrupt service subroutine frequently ends:

```
ENI     Enable interrupts
RET     Return from subroutine
```

Interrupts remain disabled until the RET instruction has completed execution; thus a new interrupt cannot be acknowledged until program execution has returned to the calling program.

Once enabled, interrupts remain enabled until a DSI instruction is executed, the system is reset, or an interrupt occurs.

IAC — INCREMENT THE ACCUMULATOR



Increment the Accumulator contents. If the D status is 0, the Accumulator contents are treated as an 8-bit binary number. If the D status is 1, the Accumulator contents are treated as two BCD digits.

Suppose the Accumulator contains 73_{16}. Execution of the instruction:

IAC

will increment the Accumulator to 74_{16} in either binary or decimal mode.

Now suppose the Accumulator contains 79_{16}. The instruction sequence:

```
SEB   Set binary mode
IAC   Increment Accumulator
```

leaves $7A_{16}$ in the Accumulator; the C status is 0. However, the instruction sequence:

```
SED   Set decimal mode
IAC   Increment Accumulator
```

leaves 80_{16} in the Accumulator, 1 in the H status and 0 in the C status.

Now suppose the Accumulator initially contains 99_{16}. The instruction sequence:

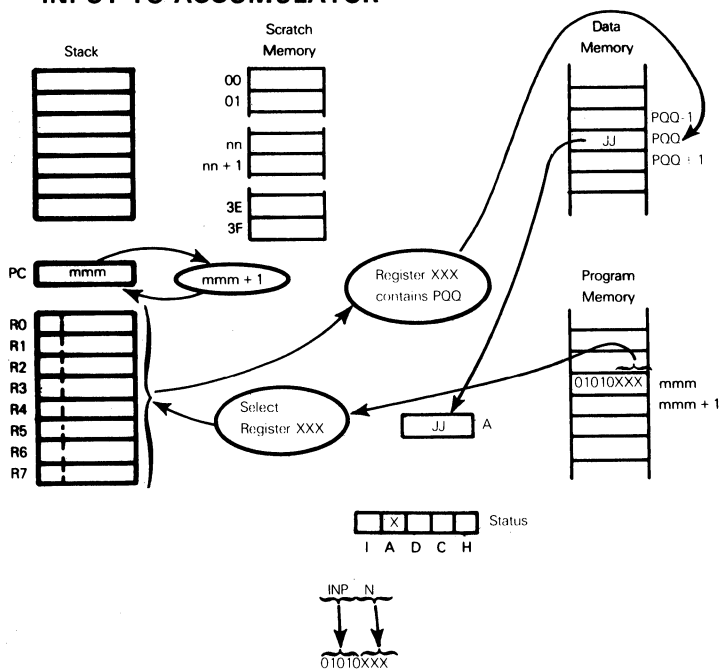
```
SEB   Set binary mode
IAC   Increment Accumulator
```

leaves $9A_{16}$ in the Accumulator and 0 in the C status. The instruction sequence:

```
SED   Set decimal mode
IAC   Increment Accumulator
```

leaves 00_{16} in the Accumulator and 1 in both C and H statuses.

INP — INPUT TO ACCUMULATOR

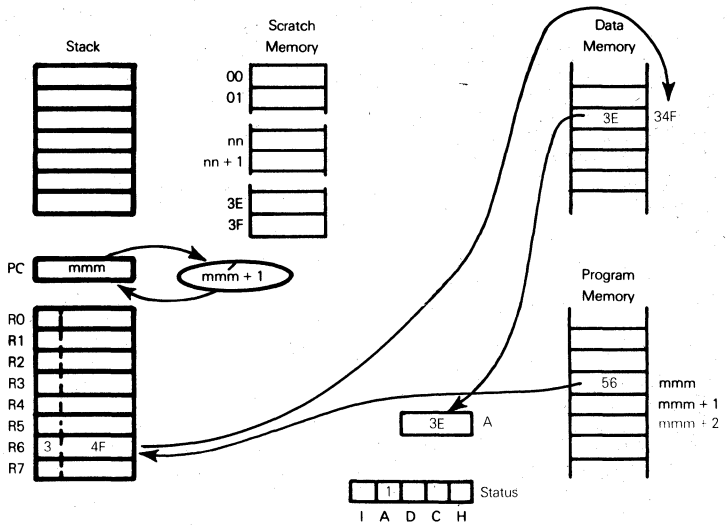


Load into the Accumulator the contents of the memory byte addressed by Register N.

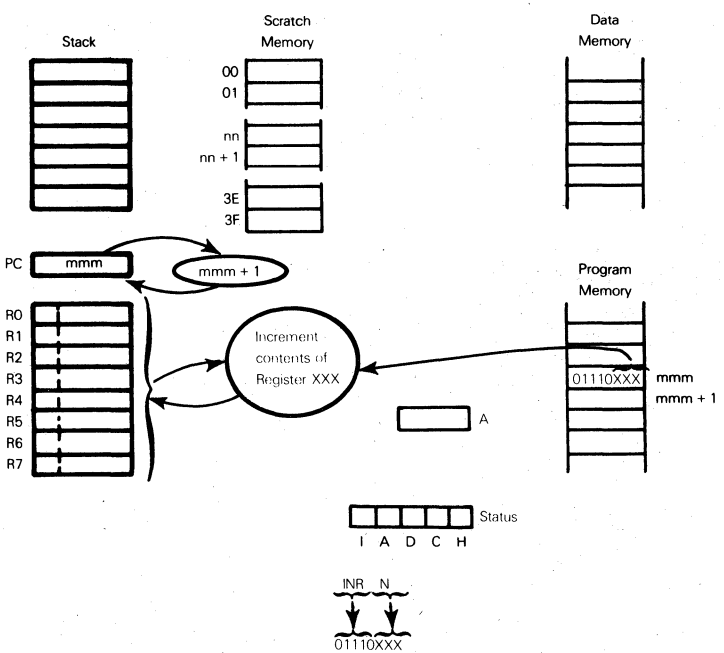
Register N may also address an external device; in other words, the contents of Register N are output on the Address bus. Any logic external to the 9002 may decode the Address bus, select itself and return data on the data bus.

Suppose the data memory byte with address $34F_{16}$ contains the value $3E_{16}$; when the instruction:

is executed, assuming the Register R6 contains $34F_{16}$, this is what happens:



INR — INCREMENT REGISTER



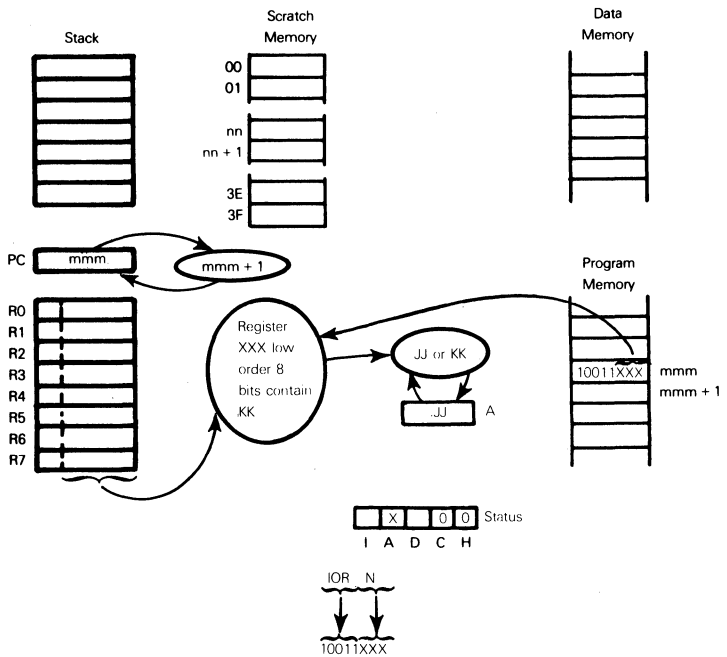
Increment the twelve bit contents of Register N.

Suppose Register R3 contains $23A_{16}$; after execution of the instruction:

INR 3

Register 3 will contain $23B_{16}$. Had Register R3 contained $2FF_{16}$, it would now contain 300_{16}; had the register contained FFF_{16}, it would now contain 000_{16}.

IOR — INCLUSIVE OR REGISTER WITH ACCUMULATOR



Logically OR the Accumulator with the low order eight bits of Register N.

Suppose the Accumulator contains $3E_{16}$ and Register 3 contains $2F3_{16}$; after execution of the instruction

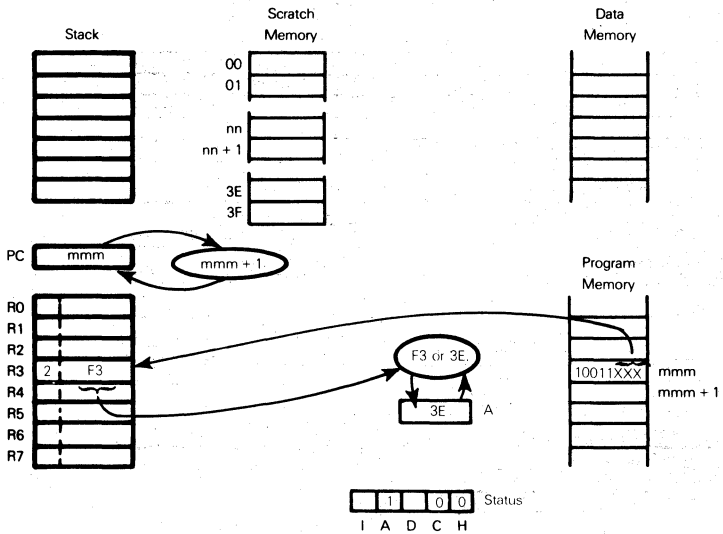
IOR 3

the Accumulator will contain FF_{16}:

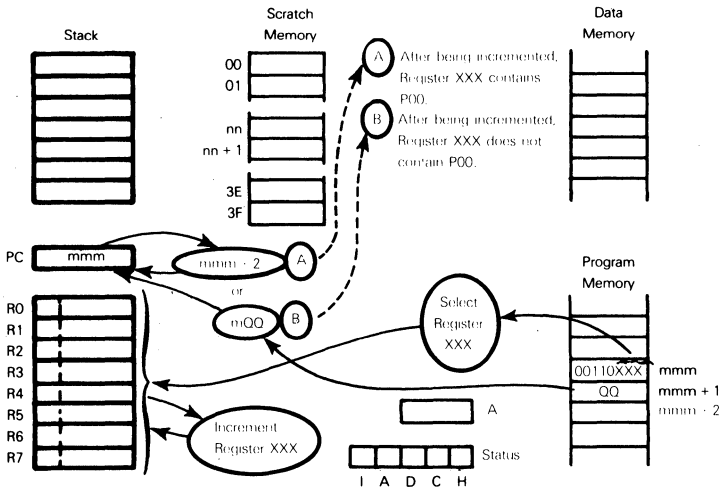
```
3E=   00111110
F3=   11110011
OR=   11111111
```

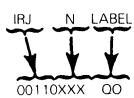
The C and H statuses are unconditionally reset to 0. The A status is set to 1 since the result in the Accumulator is not zero.

This is what happens:



IRJ — INCREMENT REGISTER AND JUMP

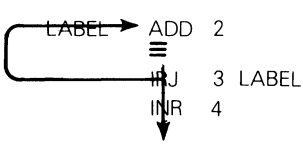




Increment the 12 bit contents of Register N. After incrementing, test the low order eight bits; if they are all zero, program execution continues with the next sequential instruction; if they are not all zero, program execution branches to the instruction on the same page identified by the label LABEL.

For a discussion of how the label LABEL is handled when the IRJ instruction object code resides at, or across a page boundary, see the JCY instruction.

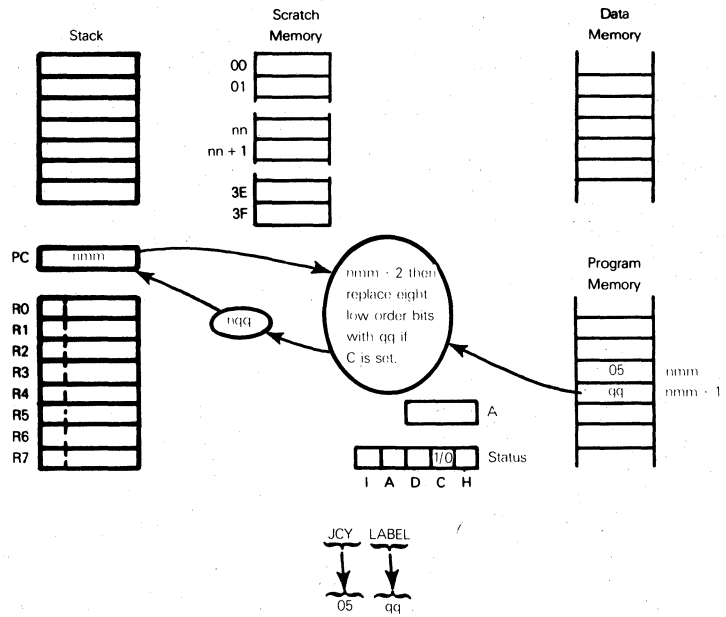
Consider the instruction sequence:



The IRJ instruction specifies Register 3 as the register to be incremented. After incrementing, if the low order eight bits of Register 3 are all 0, then the INR instruction will be executed; otherwise the ADD instruction will be executed.

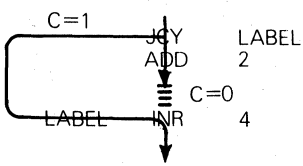
As described for the DRJ instruction, note that a jump into the next page will occur if the IRJ instruction object code terminates at, or lies across a page boundary.

JCY — JUMP IF CARRY EQUALS 1



If Carry status equals 1, jump to instruction with label LABEL in current page; otherwise execute the next sequential instruction.

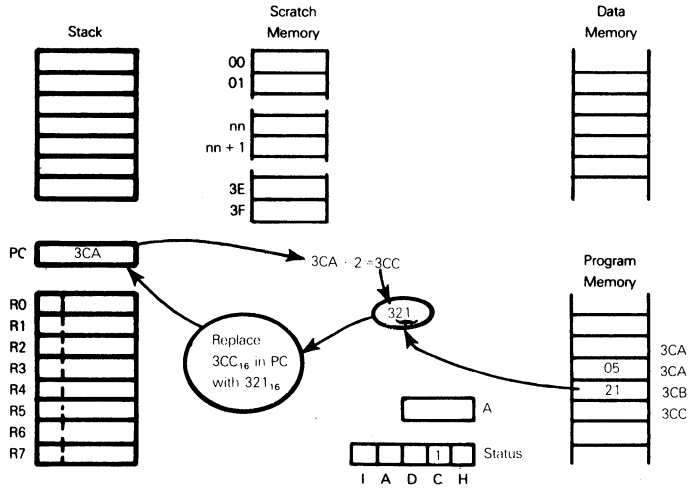
Consider the instruction sequence:



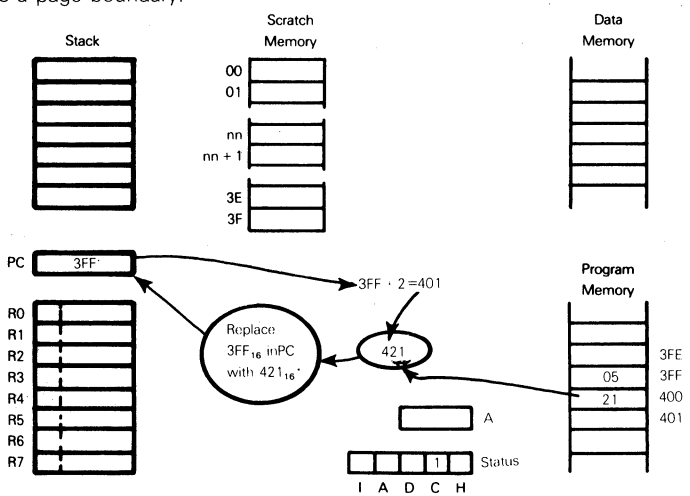
If the Carry status equals 1, then the INR 4 instruction will be executed following the JCY instruction, providing the INR 4 instruction is on the same program memory page as the JCY instruction.

If the Carry status equals 0, then the ADD 2 instruction will be executed following the JCY instruction.

Suppose the JCY instruction object code is stored in program memory bytes $3CA_{16}$ and $3CB_{16}$; that is to say nmm = $3CA_{16}$. Suppose also that the INR 4 instruction object code is stored in program memory byte 321_{16}. If the Carry status equals 1, this is how the JCY instruction will be executed:



Now consider what happens when the JCY instruction object code is located on, or across a page boundary:



As illustrated above, the branch occurs into the page occupied by the instruction following the JCY instructions.

JEQ — JUMP IF ACCUMULATOR AND REGISTER ARE EQUAL

This instruction is identical to the JZE instruction, but it is phrased to represent conditions after execution of a CMP, SUB or SUS instruction.

JGE — JUMP IF ACCUMULATOR IS GREATER THAN, OR EQUAL TO REGISTER

This instruction is identical to the JNC instruction, but it is phrased to represent conditions after execution of a CMP, SUB or SUS instruction.

JGT — JUMP IF ACCUMULATOR IS GREATER THAN REGISTER

```
JGT          LABEL

03           QQ
```

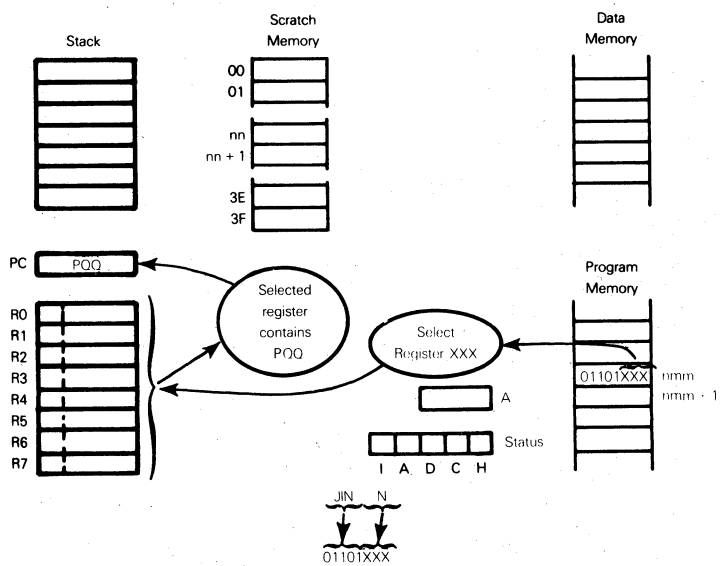
This instruction is similar to the JCY instruction, except that the jump only occurs if the Carry status equals 0 and the Accumulator status equals 1.

This instruction can be executed at any time, but it is phrased to represent conditions after execution of a CMP, SUB or SUS instruction.

JHC — JUMP IF HALF CARRY

This instruction is similar to the JCY instruction, except that the jump only occurs if the Half Carry status equals 1.

JIN — JUMP INDIRECT (IMPLIED)



Jump to the memory location specified by the identified general purpose register.

Suppose register R3 contains $3CA_{16}$; the instruction:

```
JIN  3
```

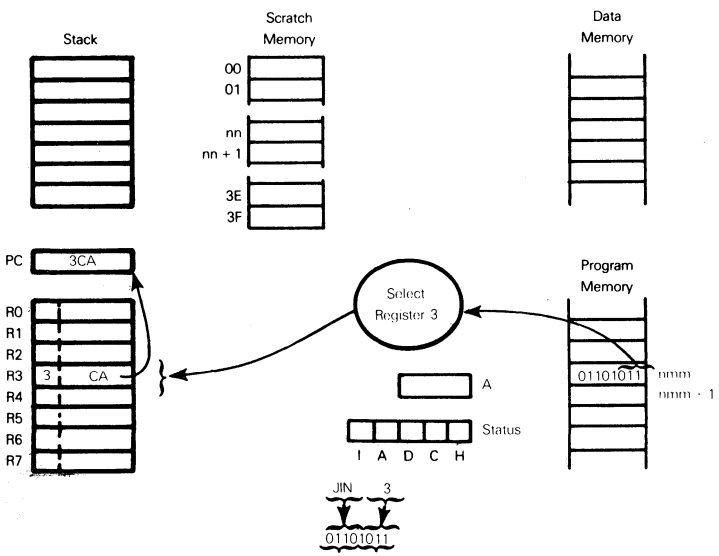
will cause program execution to continue with the instruction whose object code is stored in program memory byte $3CA_{16}$.

Now suppose the label LABEL is equivalent to the value $3CA_{16}$ in the following instruction sequence:

```
      JIN  3
      ADD  2
       ≡
LABEL INR  4
```

This means the INR 4 instruction code is stored in program memory byte $3CA_{16}$. Again assuming that Register R3 contains $3CA_{16}$, the instruction sequence above becomes identical to the instruction sequence illustrated for the JUN instruction.

The JIN 3 instruction specifically may be illustrated as follows:



JLE — JUMP IF ACCUMULATOR IS LESS THAN OR EQUAL TO REGISTER

This instruction is similar to the JCY instruction, except that the jump only occurs if the Carry status equals 1 or the Accumulator status equals 0.

This instruction can be executed at any time, but it is phrased to represent conditions after execution of a CMP, SUB or SUS instruction.

JLT — JUMP IF ACCUMULATOR IS LESS THAN REGISTER

This instruction is identical to the JCY instruction, but it is phrased to represent conditions after execution of a CMP, SUB or SUS instruction.

JNC — JUMP IF NO CARRY

This instruction is similar to the JCY instruction, except that the jump only occurs if the Carry status equals 0.

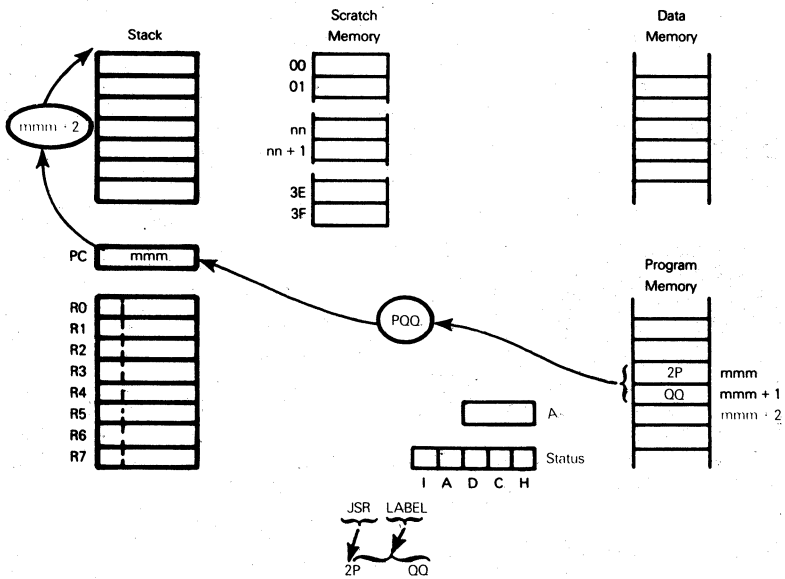
JNE — JUMP IF ACCUMULATOR IS NOT EQUAL TO REGISTER

This instruction is identical to the JNZ instruction, but it is phrased to represent conditions after execution of a CMP, SUB or SUS instruction.

JNZ — JUMP IF ACCUMULATOR IS NOT ZERO

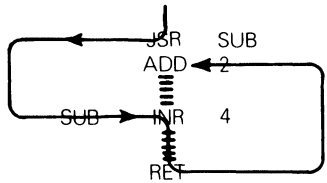
This instruction is similar to the JCY instruction, except that the jump only occurs if the Accumulator status equals 1.

JSR — JUMP TO SUBROUTINE



Save the program memory address of the next instruction on the stack; then load the twelve bit address provided by the JSR instruction into the Program Counter.

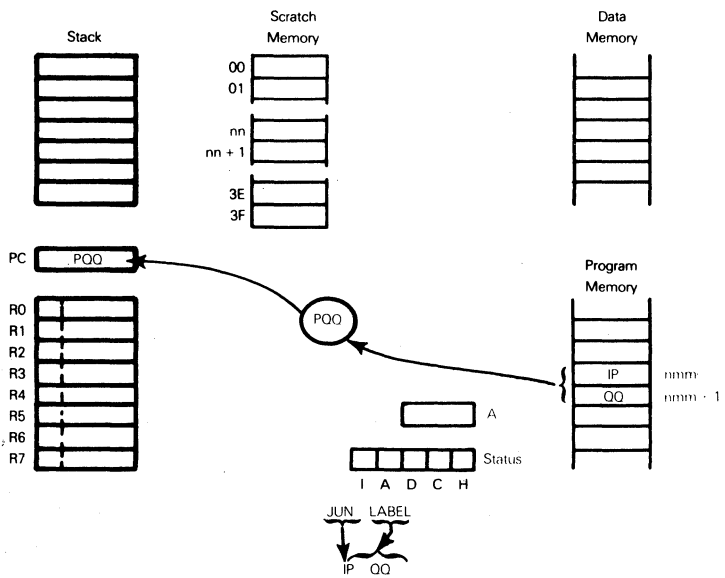
Consider the following instruction sequence:



The JSR instruction causes program execution to branch unconditionally to the instruction labeled SUB; the JSR instruction also saves the address of the ADD instruction on the stack.

The first RET instruction executed following the INR instruction will branch program execution back to the ADD instruction.

JUN — JUMP UNCONDITIONAL



Jump to the instruction identified by the label, LABEL.

Consider this instruction sequence:

```
        JUN   LABEL
        ADD   2
        ≡
LABEL   INR   4
```

After JUN LABEL, the INR 4 instruction will be executed. Unless some other instruction jumps to ADD 2, this instruction will never be executed.

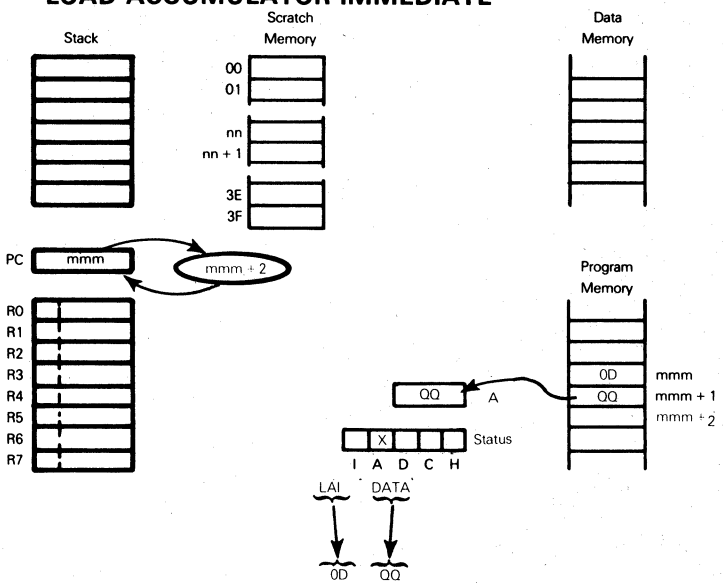
Suppose the object code for the INR 4 instruction happens to be stored in memory location $3FA_{16}$; the JUN LABEL instruction's object code will be:

13 FA

JZE — JUMP IF ACCUMULATOR IS ZERO

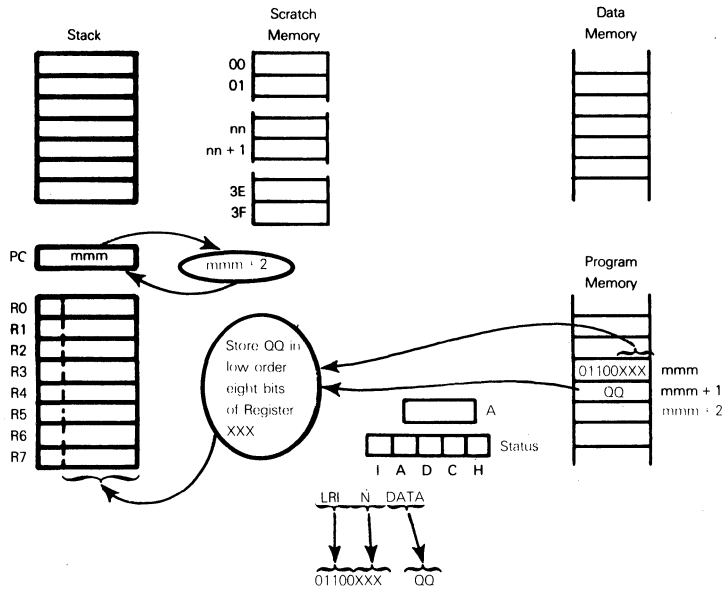
This instruction is similar to the JCY instruction, except that the jump only occurs if the Accumulator status equals 0.

LAI — LOAD ACCUMULATOR IMMEDIATE



Load the contents of the second instruction object code byte into the Accumulator. Set the A status to 1 if the Accumulator now contains a non zero value; reset the A status to 0 if the Accumulator now contains a zero value.

LRI — LOAD REGISTER IMMEDIATE

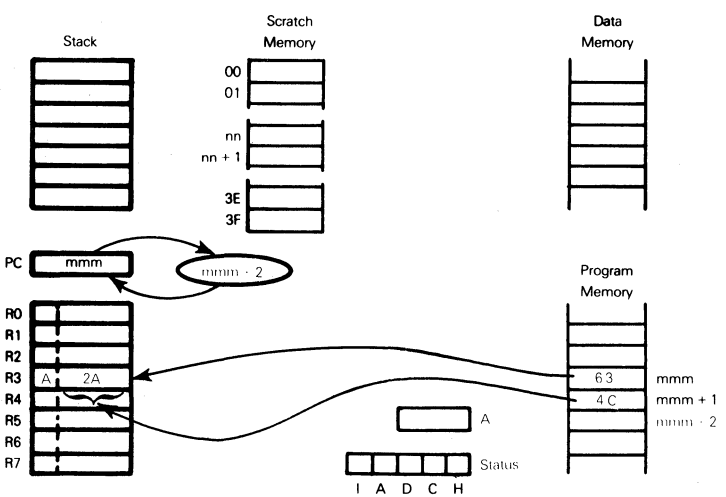


Load the contents of the second instruction object code byte into the low order eight bits of Register N.

Consider the instruction:

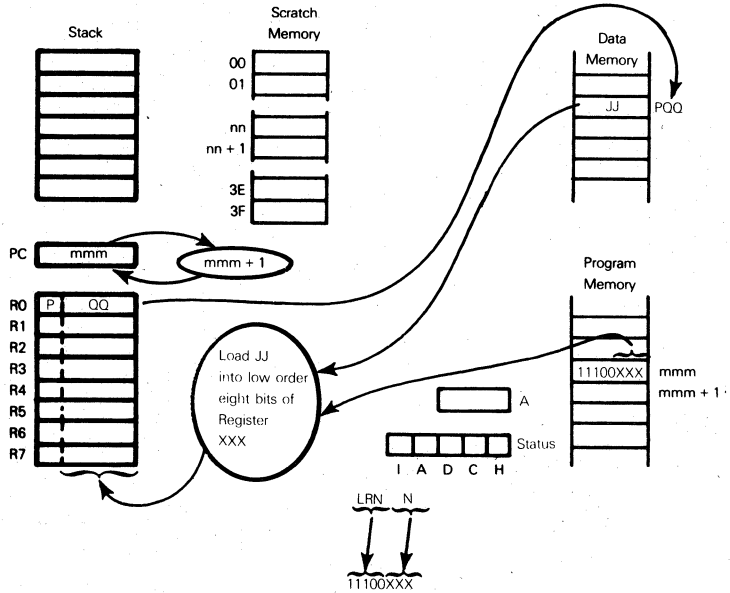
LRI 3,4C

When executed, this is what happens:



Now Register R3 will contain $A4C_{16}$.

LRN — LOAD REGISTER INDIRECT



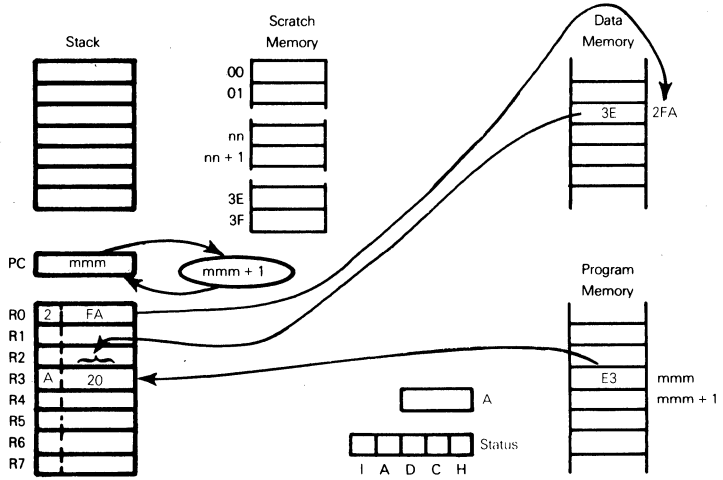
Load into the low order eight bits of Register N the contents of the memory byte or external device addressed by register R0.

The contents of Register R0 are output on the Address bus. Either external memory or any external logic may decode the Address bus, select itself and place data on the data bus.

Suppose R0 contains $2FA_{16}$ and memory byte $2FA_{16}$ contains $3E_{16}$. After the instruction:

LRN 3

is executed, if Register 3 originally contained $A20_{16}$, it will now contain $A3E_{16}$:

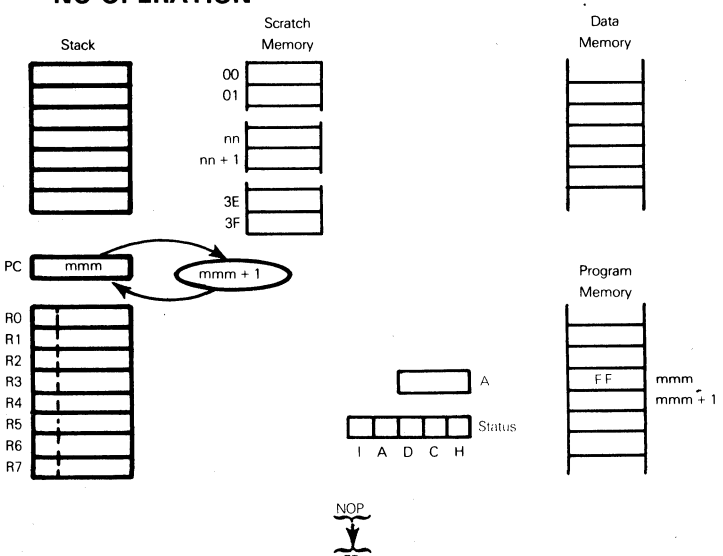


The instruction:

LRN 0

is allowed; in the above illustration it would leave $23E_{16}$ in Register 0.

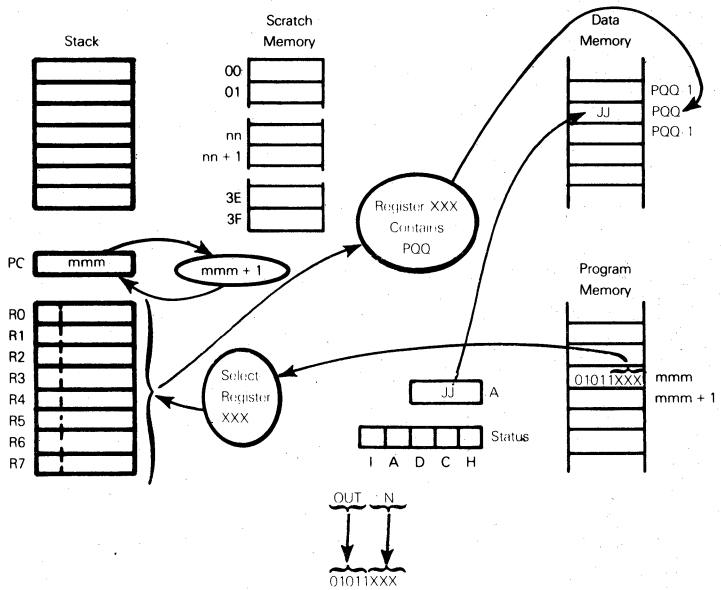
NOP — NO OPERATION



Nothing happens when this instruction is executed, but the Program Counter is incremented to address the next sequential instruction code.

The NOP instruction is frequently used to create time delays for signal synchronization. There are also circumstances in which it is used to create an address to which jumps can be specified.

OUT — OUTPUT FROM ACCUMULATOR



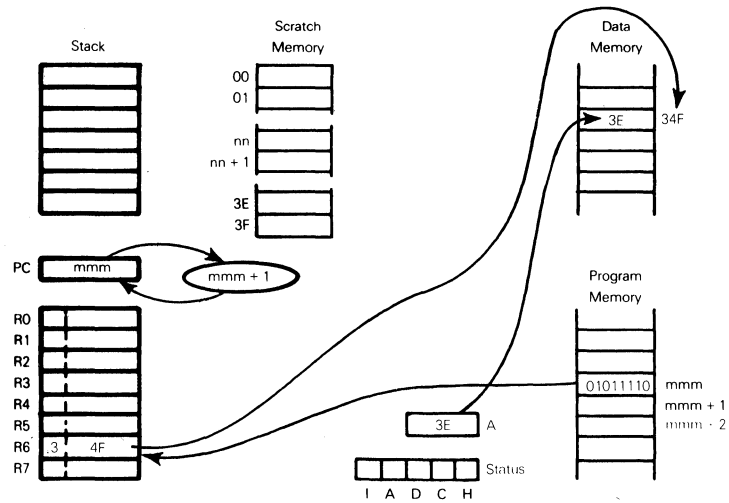
Store the Accumulator contents in the memory byte addressed by Register N.

Register N may also address an external device; in other words, the contents of Register N are output on the Address bus. Any logic external to the 9002 may decode the Address bus, select itself and accept the data on the data bus.

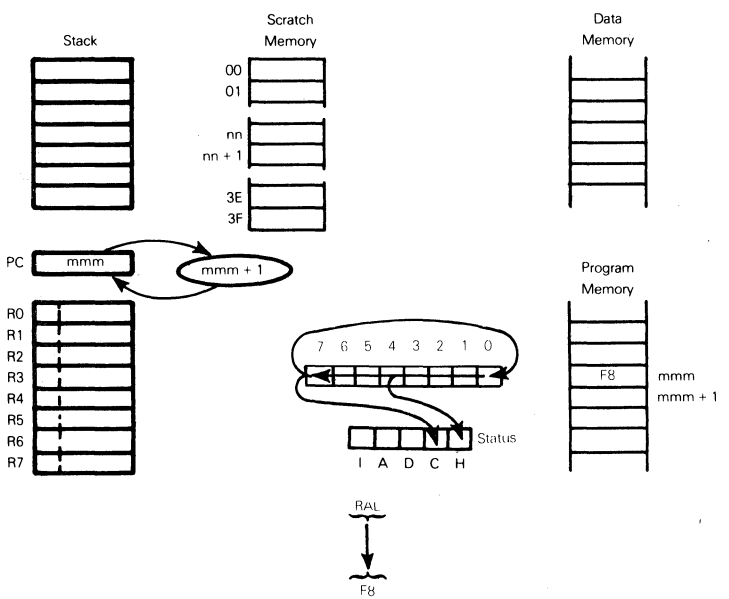
Suppose the Accumulator contains $3E_{16}$ and Register R6 contains $34F_{16}$; after the instruction:

OUT 6

is executed, the memory byte (or external logic device) with address $34F_{16}$ will contain $3E_{16}$:



RAL — ROTATE ACCUMULATOR LEFT

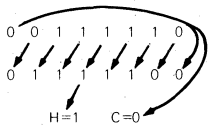


Rotate the Accumulator left one bit position.

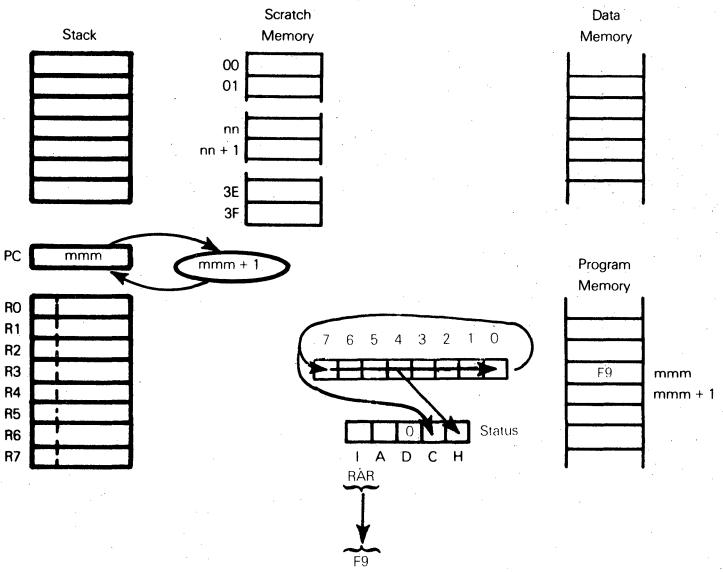
Suppose the Accumulator initially contains $3E_{16}$. After execution of the

RAL

instruction, the Accumulator will contain $7C_{16}$:



RAR — ROTATE ACCUMULATOR RIGHT

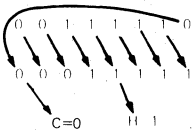


Rotate the Accumulator right one bit position.

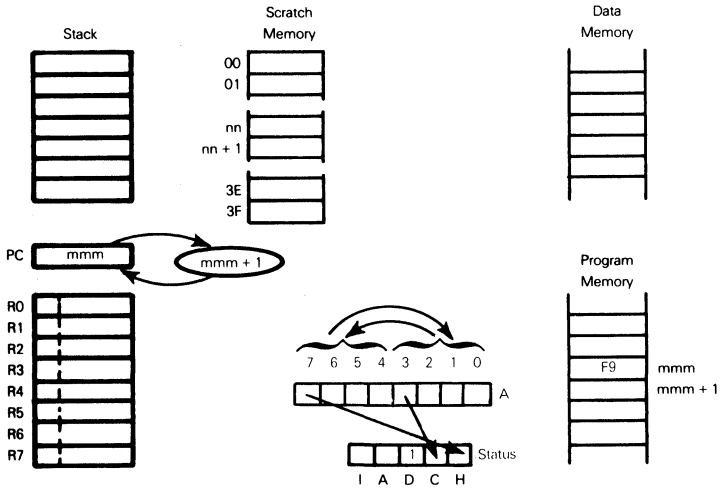
Suppose the Accumulator initially contains $3E_{16}$. After execution of the

SEB
RAR

instruction, the Accumulator will contain $1F_{16}$:



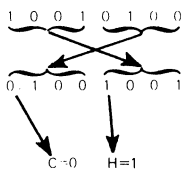
In decimal mode, the right rotate is executed as follows:



Suppose the Accumulator contains 94_{16}; after execution of the instruction sequence:

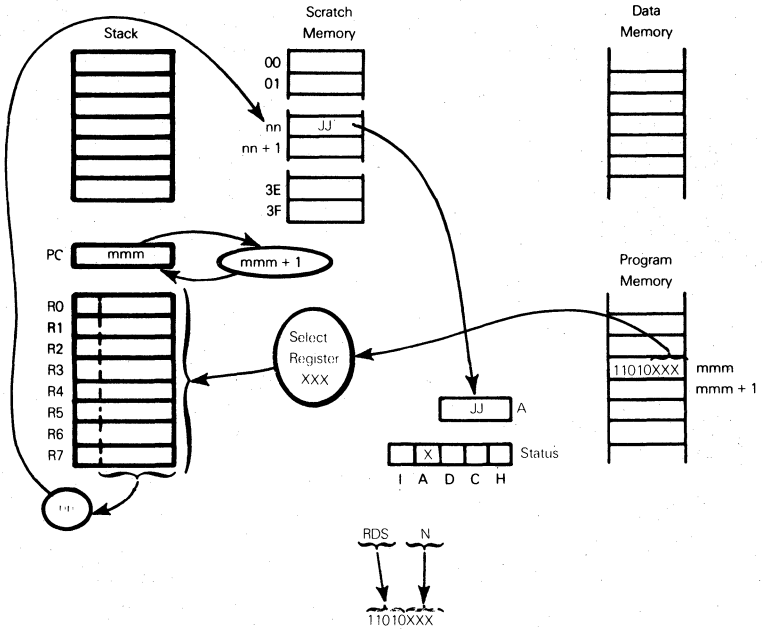
```
SED        Set decimal mode
 ⋮
RAR
```

the Accumulator will contain 49_{16}:



You can use decimal mode right rotate even if the Accumulator contains non BCD data (e.g. FA_{16}).

RDS — READ SCRATCH MEMORY TO ACCUMULATOR



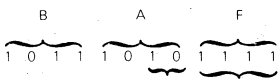
Input to the Accumulator the contents of the scratch memory byte addressed by the low order six bits of Register N.

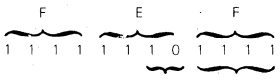
Suppose Register 4 contains $02F_{16}$ and scratch memory byte $02F_{16}$ contains $3E_{16}$. After the instruction:

RDS 4

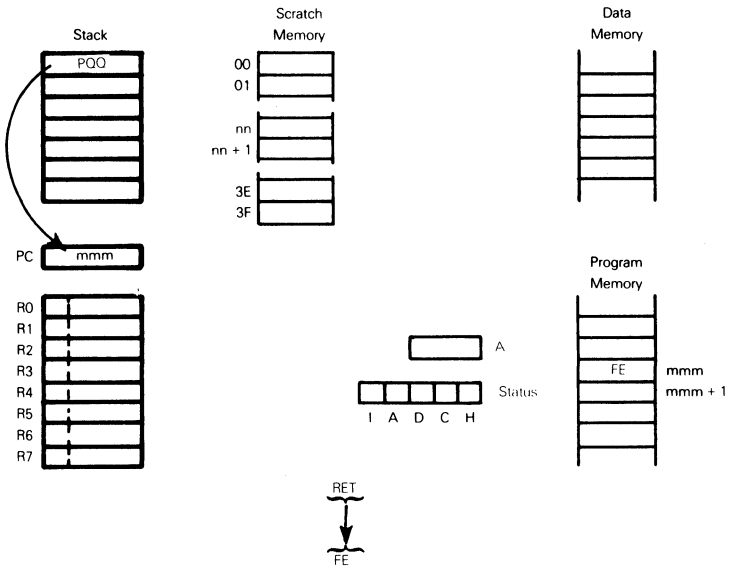
has executed, the Accumulator will contain $3E_{16}$ and the A status will be set to 1.

Note that only the low order six bits of the Register are significant; a scratch memory address of $02F_{16}$ would be derived from any of these register contents:



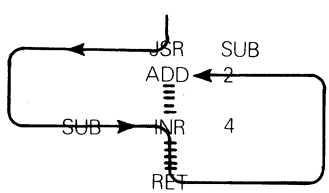


RET — RETURN FROM SUBROUTINE



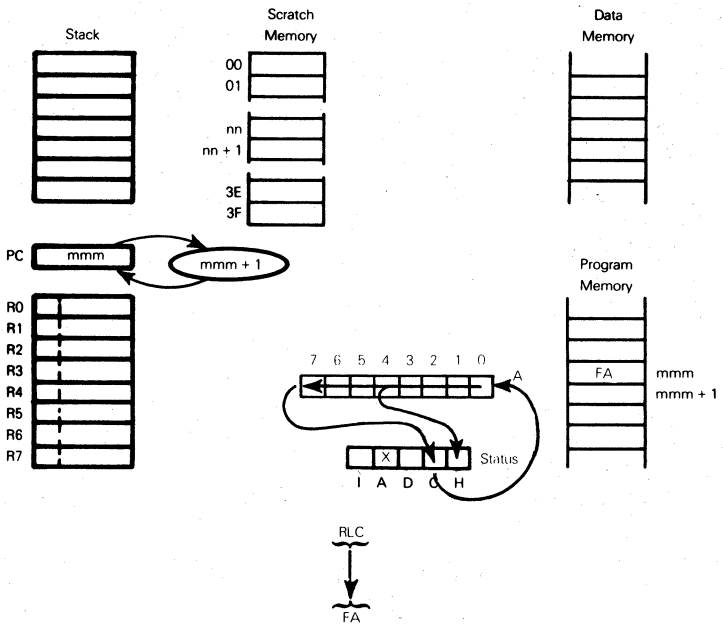
The address at the top of the stack is popped into the Program Counter, thus effecting a return from subroutine.

Consider the following instruction sequence:



When the JSR instruction is executed, the address of the ADD instruction is pushed onto the stack. This is the address popped off the stack and into the Program Counter by the RET instruction, thus causing program execution to continue with the ADD instruction.

RLC — ROTATE ACCUMULATOR LEFT THROUGH CARRY

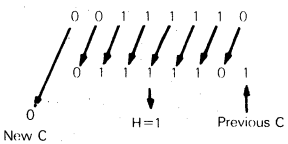


Rotate the Accumulator left one bit position, through carry.

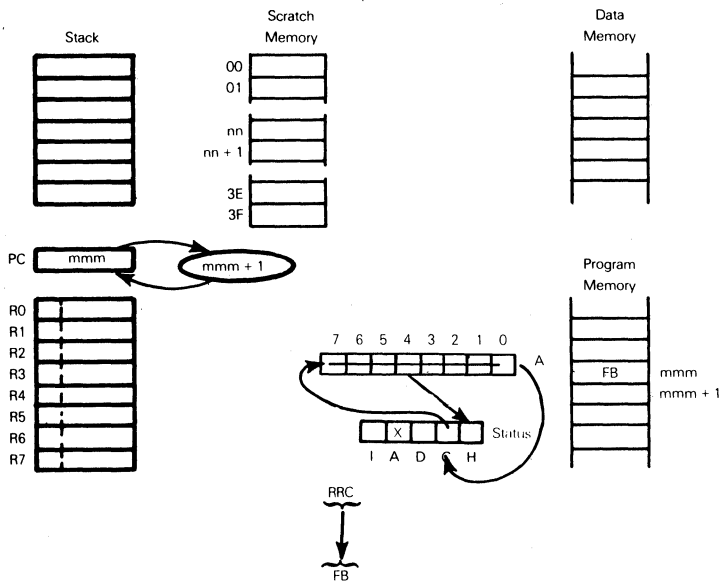
Suppose the Accumulator initially contains $3E_{16}$ and the Carry status is 1. After execution of the instruction:

RLC

the Accumulator will contain $7B_{16}$:



RRC — ROTATE ACCUMULATOR RIGHT THROUGH CARRY

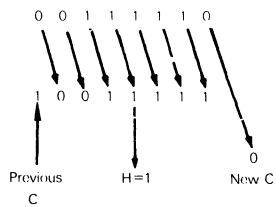


Rotate the Accumulator right one bit position, through carry.

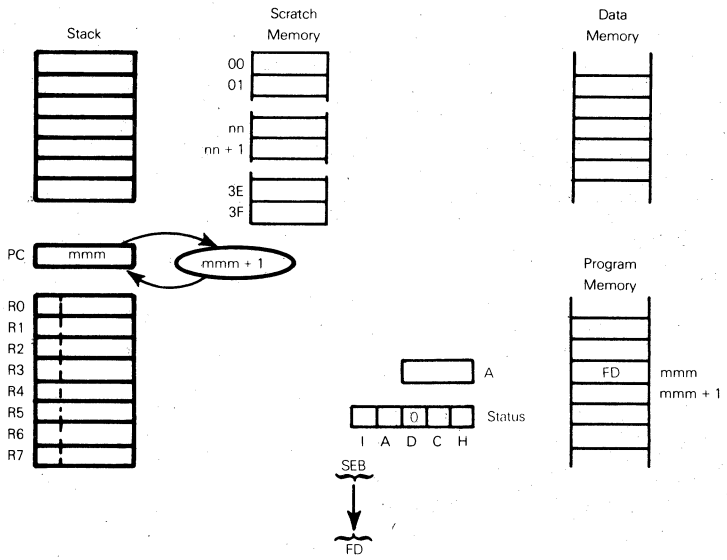
Suppose the Accumulator initially contains $3E_{16}$ and the Carry status is 1. After execution of the instruction:

RRC

the Accumulator will contain $9F_{16}$:



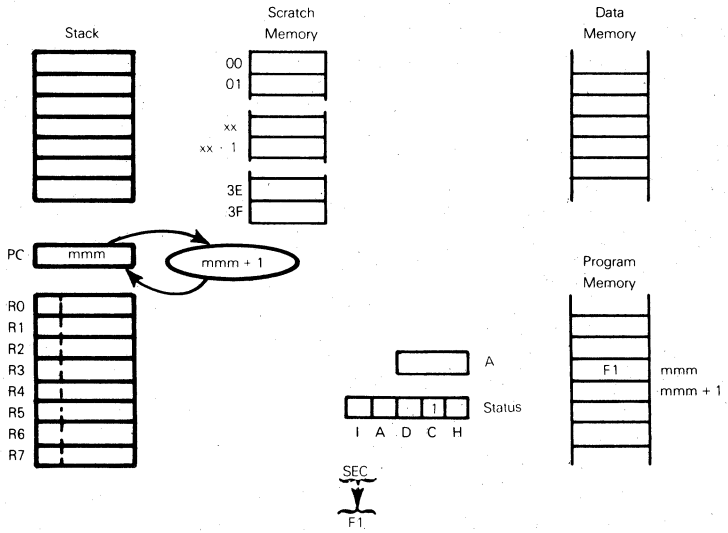
SEB — SET BINARY MODE



Unconditionally reset the Decimal status to 0. This selects binary mode when any of the following instructions are executed: ADD, ADS, DAC, IAC, RAR, SUB, SUS.

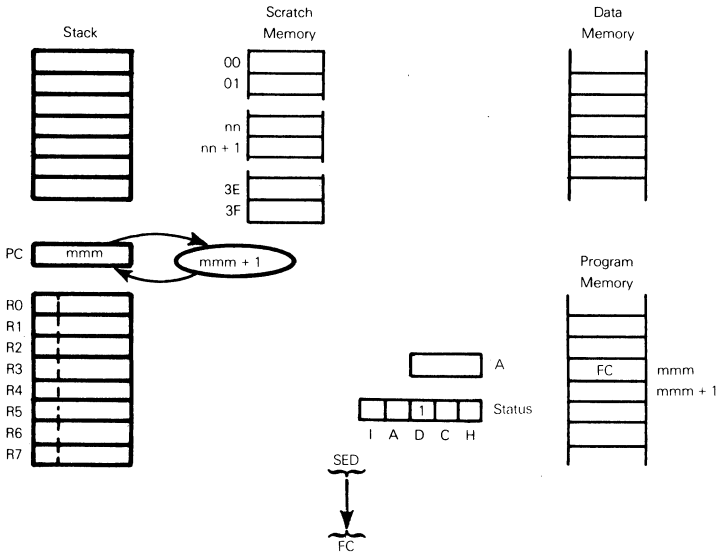
Binary mode remains in effect until an SED instruction is executed to set Decimal mode.

SEC — SET THE CARRY STATUS TO 1



Unconditionally set the Carry status to 1.

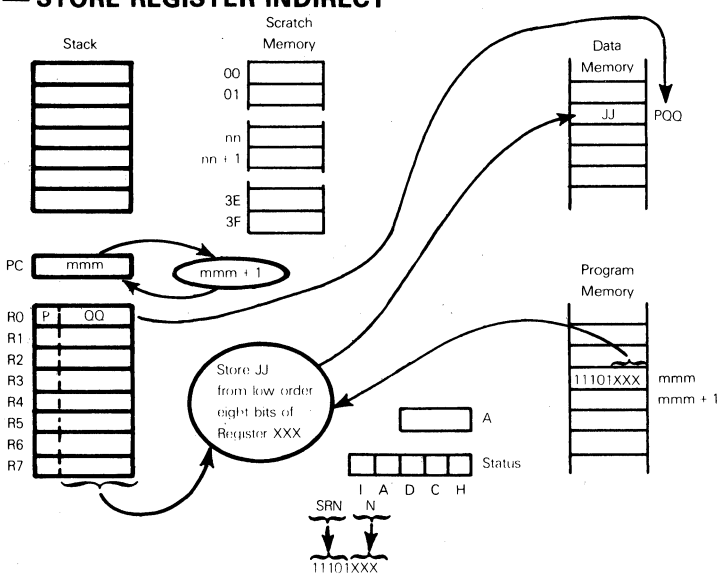
SED — SET DECIMAL MODE



Unconditionally set the Decimal status to 1. This selects decimal mode when any of the following instructions are executed: ADD, ADS, DAC, IAC, RAR, SUB, SUS.

Decimal mode remains in effect until an SEB instruction is executed to set binary mode, or until the system is reset.

SRN — STORE REGISTER INDIRECT



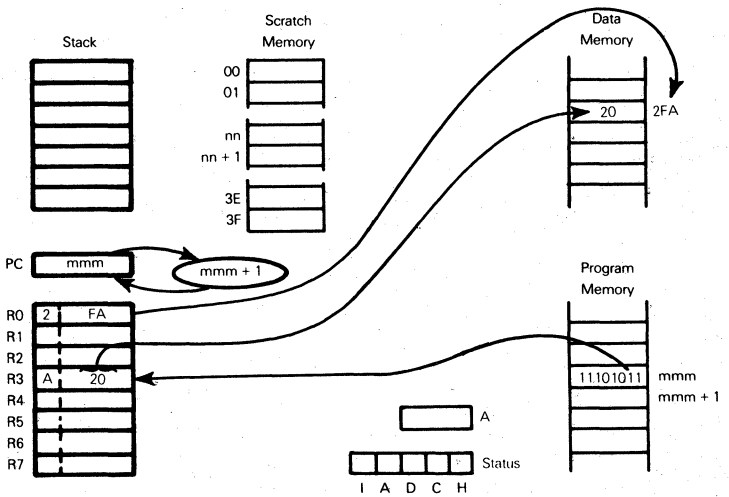
Store the low order eight bits of Register N into the memory byte or external device addressed by register R0.

The contents of Register R0 are output on the Address bus. Either external memory or any external logic may decode the Address bus, select itself and accept the data on the data bus.

Suppose R0 contains $2FA_{16}$ and Register R3 contains $A20_{16}$. After the instruction:

SRN 3

is executed, the data memory byte (or external logic) selected by address $2FA_{16}$ will contain 20_{16}:

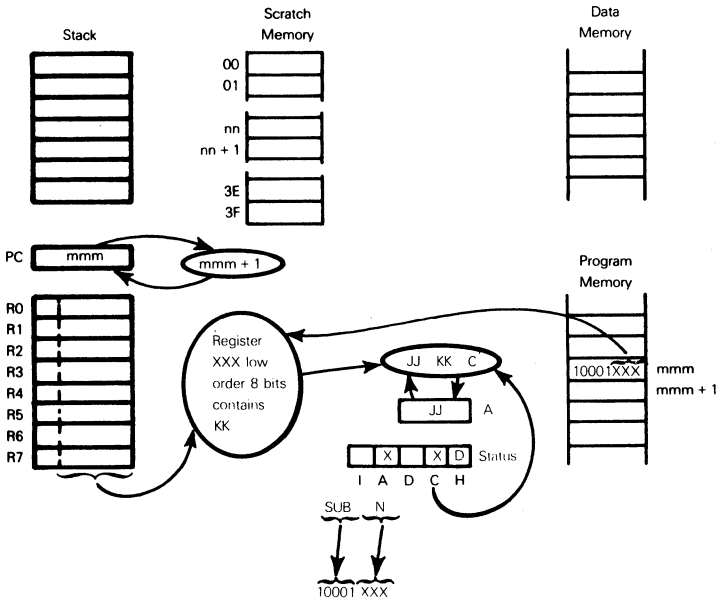


The instruction:

SRN 0

is allowed; in the above illustration it would leave FA_{16} in memory byte $2FA_{16}$.

SUB — SUBTRACT REGISTER FROM ACCUMULATOR WITH BORROW



From the Accumulator subtract the Carry status and the low order eight bits of register N. If the D status is 0, the Accumulator and Register contents are treated as 8-bit binary data. If the D status is 1, the Accumulator and Register contents are each treated as a pair of BCD digits.

Suppose the Accumulator contains 73_{16}, Register R5 contains 347_{16} and the Carry status is 1; after execution of the instruction sequence:

```
SEB      Set binary mode
 ≡
SUB
```

the Accumulator will contain $2B_{16}$:

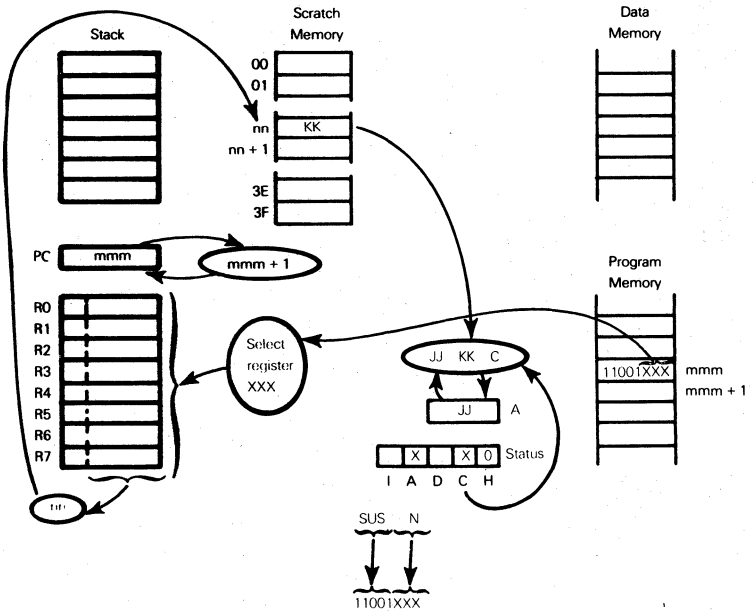
```
                73 =  01110011
-(47 + C) = (-48) =  01001000
                     --------
                     00101011

C=0
```

Now look at decimal subtraction:

```
 73
-48
---
=25

H=1
C=0
```

Note that as illustrated above, the EA9002 has subtract logic; it does not use twos complement addition. The principal effect this has is on the Carry status, which is set for a negative result, and is reset for a positive result. This is opposite from Carry status interpretation when using twos complement subtract logic.

SUS — SUBTRACT SCRATCH MEMORY FROM ACCUMULATOR WITH BORROW

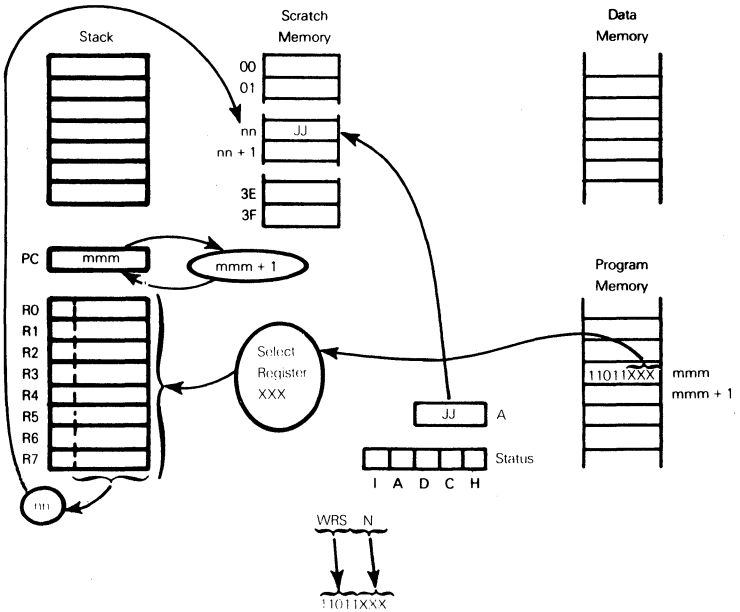


From the Accumulator subtract the Carry status and the contents of the scratch memory byte addressed by the low order six bits of Register N.

The SUS instruction is identical to the SUB instruction except that the subtrahend byte is fetched from scratch memory.

For a discussion of scratch memory addressing see the RDS instruction.

WRS — WRITE ACCUMULATOR TO SCRATCH MEMORY



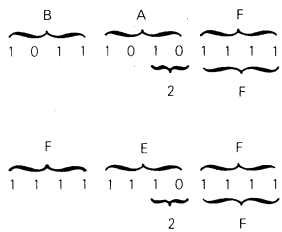
Output the Accumulator contents to the scratch memory byte addressed by the low order six bits of Register N.

Suppose Register 4 contains $02F_{16}$ and the Accumulator contains $3E_{16}$. After the instruction:

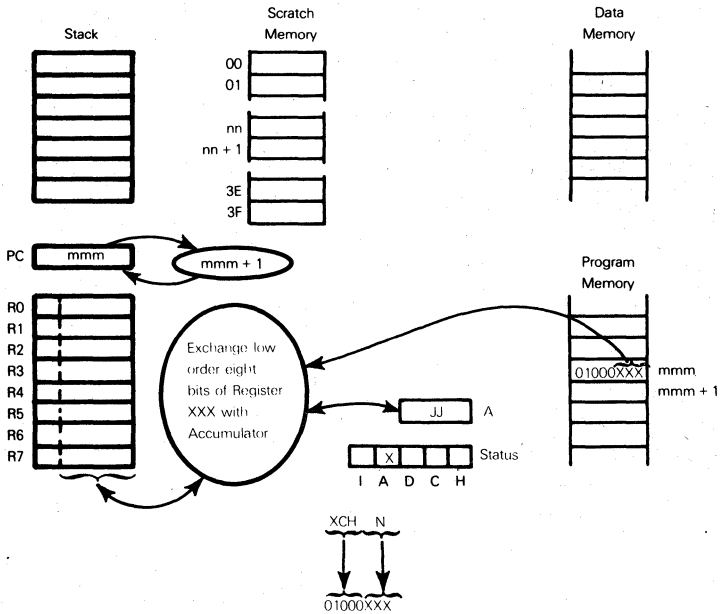
WRS 4

has executed scratch memory byte $02F_{16}$ will contain $3E_{16}$.

Note that only the low order six bits of the Register are significant; a scratch memory address of $02F_{16}$ would be derived from any of these other register contents:



XCH — EXCHANGE REGISTER AND ACCUMULATOR CONTNETS

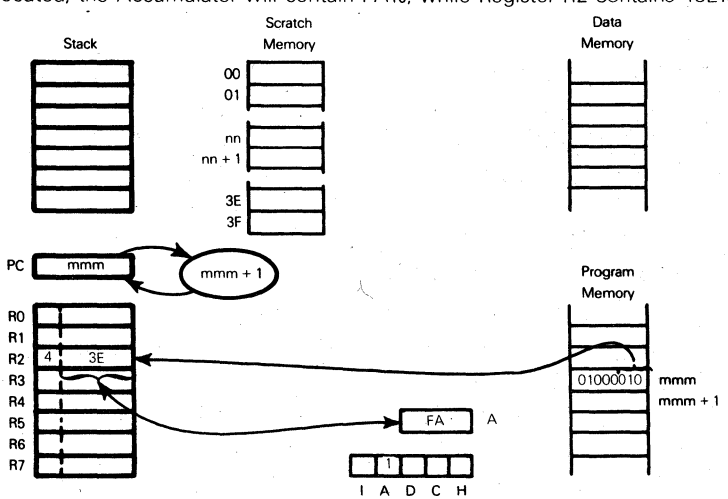


Exchange the contents of low order eight bits of Register N with the Accumulator.

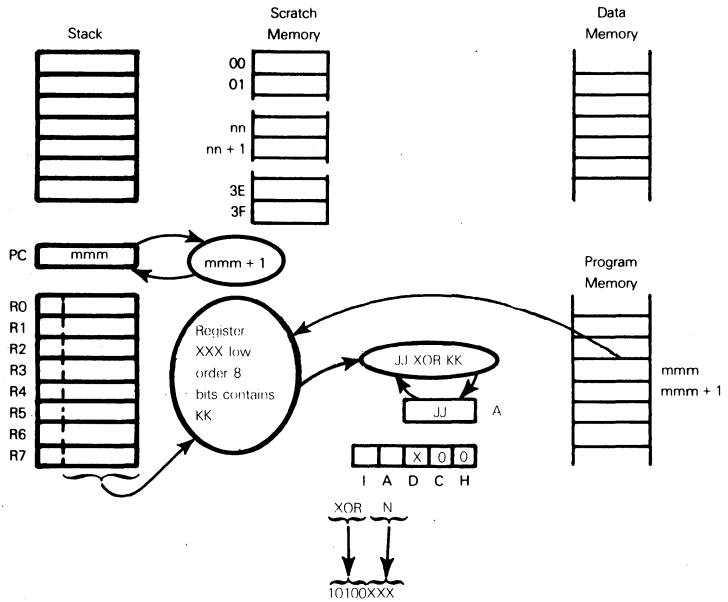
Suppose the Accumulator contains $3E_{16}$ and Register R2 contains $4FA_{16}$; after the instruction:

XCH 2

is executed, the Accumulator will contain FA_{16}, while Register R2 contains $43E_{16}$.



XOR — EXCLUSIVE OR REGISTER WITH ACCUMULATOR



Exclusive OR the Accumulator with the low order eight bits of Register N.

Suppose the Accumulator contains $3E_{16}$ and Register 3 contains $2F_{16}$; after execution of the instruction:

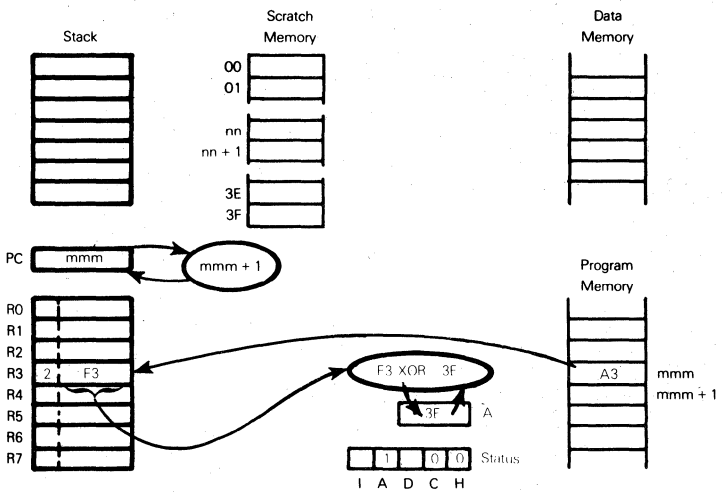
XOR 3

the Accumulator will contain CD_{16}:

```
3E =  00111110
F3    11110011
      --------
XOR   11001101
```

The C and H statuses are unconditionally reset to 0. The A status is set to 1 since the result in the Accumulator is not zero.

This is what happens:



Chapter 6
ELEMENTARY PROGRAMMING TECHNIQUE

A logic designer with no prior programming background may well have less trouble than a programmer, understanding the basic concepts of efficient microcomputer programming. When compared to many minicomputers, the average microcomputer may indeed appear to have both a primitive instruction set and limited processing capabilities; however, many microcomputer features which look like handicaps to the traditional programmer, are in fact advantages in typical microcomputer applications.

The key to understanding efficient microcomputer programming is to bear in mind that the typical microcomputer program is going to become a PROM or ROM chip.

The microcomputer is not a vehicle for the execution of an indeterminate sequence of undefined programs.

MICROCOMPUTER PROGRAMMING CONCEPTS

If we are to define "microcomputer programming" as a technique for the creation of ROM or PROM chips, then programming economy becomes very important. "Programming economy" means that you should attempt to create object programs that are as short as possible — without resorting to expensive or strange gimmicks.

The cost of program memory is not in itself the overwhelming reason for keeping programs short; but there is a snowball cost effect.

First consider read/write memory. If you can confine your data to using 64 bytes of read/write memory, then your 9002 microcomputer system will require no external RAM. This eliminates the cost of the external RAM, it's select and interface logic, plus design and fabrication expenses.

Similar reasoning applies to the number of PROM or ROM chips needed to implement your program. It is not just the cost of the extra chips that are important; there is also cost of support and interface logic — plus design and implementation costs associated with the extra real estate on PC cards, which additional chips will require. Also, remember that each additional ROM chip requires the additional expense of defining and creating a ROM mask.

The problem with allowing your chip count to proliferate is that what used to fit on one PC card may soon require two; that means additional edge connectors, perhaps a more costly back plane, or even a larger power supply.

Chip counts and system configurations can vary significantly when different logic designers implement the same system; this will come as no surprise to an experienced logic designer, but it has some important, non obvious ramifications: microcomputer programs have to consider every hardware configuration as possibly unique. Thus **definition of memory and I/O becomes an integral part of the system creation process; writing the microcomputer program is merely another step of the same process.** This being the case, you must use general purpose subroutines with extreme caution. The minicomputer programmer takes general purpose subroutines for granted — to the point where many such subroutines are packaged into standard program modules, referred to as "systems software", and included in every program, whether they will be needed or not. To the microcomputer programmer, **using system software will probably make no sense. Saving a little programming cost cannot be justified if the price paid is more memory and a significantly more complex eventual microcomputer systems.**

MICROCOMPUTER PROGRAM INPLEMENTATION SEQUENCE

In order to program a microcomputer efficiently, you must start by defining the hardware that your system is going to require. Some hardware definition is a necessary beginning, but this definition is likely to change frequently in the course of system implementation and debugging.

Beyond the microcomputer CPU itself, you must define the address spaces that have been allocated to program memory, to data memory and to external devices. You must also define the address spaces that are to be implemented in ROM or RAM. Program memory will invariably be implemented in ROM; data memory is frequently RAM, it can be ROM when used to store tables or other nonvarying data.

FLOW CHART

The next step is to flow chart functions which the microcomputer program must perform. Figure 6-1, illustrates a simple program flow chart. The flow chart illustrates a simple program which receives digital input signal in the range 0 through 255 from an instrument — then uses this input signal to access a table out of which a 5 decimal digit number will be read for display. This program could be used by any instrument whose sensor outputs a millivoltage which must be converted, using a nonlinear transfer function, into a panel digital display. Figure 6-2 illustrates the hardware configuration and address spaces assumed.

Having drawn your program flow chart, the next step is to break this flow chart into program modules. The extent of a single program module is not easily defined in advance; however, once you have written a few programs for any microcomputer, you will find it quite easy to estimate the extent of an individual program module. **A program module begins with instruction sequences that load data or parameters into general purpose registers and/or scratchpad memory; the program module operates on this data, then outputs results at the end of the module.** You should not have to reuse general purpose registers or scratchpad memory in unrelated ways for the duration of any single program module. Program modules may also pass data, one to the next, by leaving the data in preassigned registers or scratchpad memory.

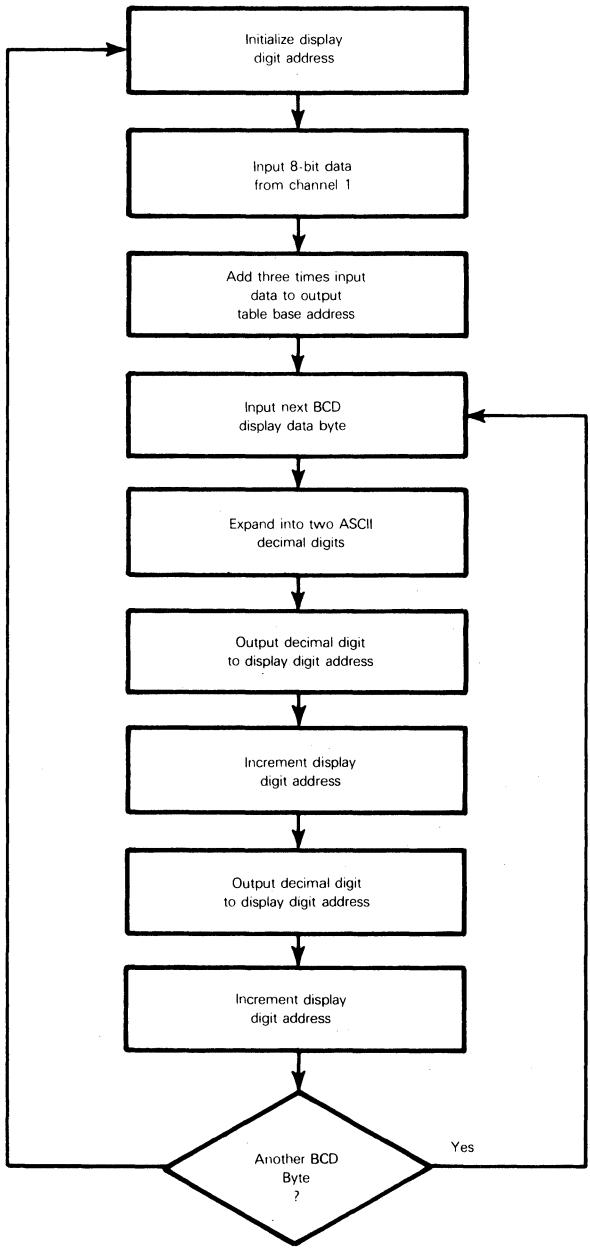


Figure 6-1 A Sample Program Flow Chart

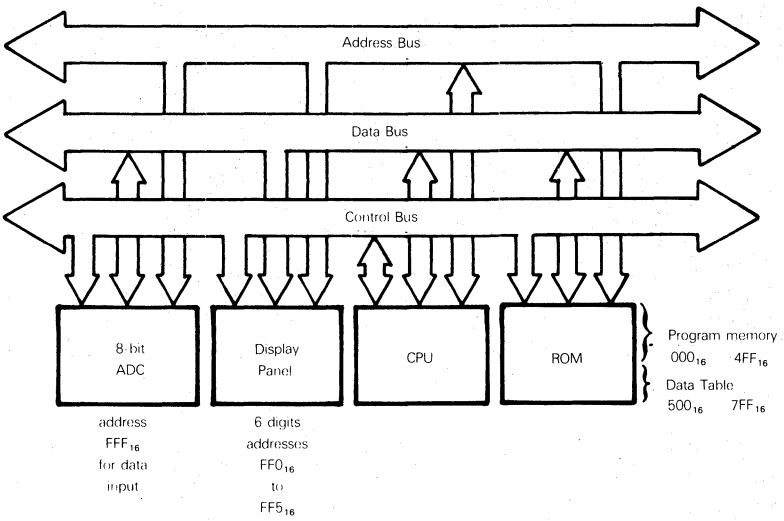


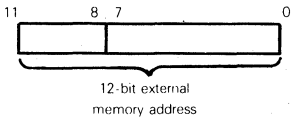
Figure 6-2 System Configuration For Sample Problem

As the above definition of a program module would imply, **you should begin by defining the use to which every single general purpose register will be put for the duration of the program module. You should also define scratchpad memory utilization.**

GENERAL PURPOSE REGISTERS ALLOCATION

First consider general purpose registers. You have eight such registers, which will support a program module of significant size. Recall that there are four ways in which a general purpose register may be used:

1) The register may identify a 12-bit external data or program memory address:



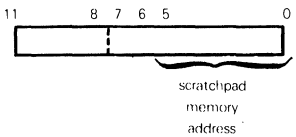
General purpose register R0 always holds the 12-bit external memory address for register indirect load and store instructions; therefore R0 will commonly be used as an external memory addressing register.

Also, the page bits of registers R0, R1, R2 and R3, only, can be copied to the Accumulator; therefore make these four registers your first choices for external memory addressing.

You may also use a general purpose register to store a program memory address. This is useful if your program has one or more instructions to which you will frequently want to jump from various other points in the program. Remember that the EA9002 provides register direct addressing jump instruction; this is a single

byte instruction that jumps to any memory location addressed by one of the eight general purpose registers.

2) **General purpose registers may be used to address scratchpad memory;** the low order six bits of any general purpose register provide a scratchpad memory address.



3) **Any general purpose register may be used as an 8-bit or a 12-bit counter.** Remember that when the contents of a general purpose register are either incremented or decremented, the entire 12-bit value of the register is affected.

4) **General purpose registers may also be used for data storage and manipulation.** Remember, only the low order eight bits of the general purpose register can be used for data access. In fact, there is no way of reading the high order four bits of registers R4, R5, R6, or R7; therefore do not attempt to store data for subsequent retrieval.

Consider the application illustrated in Figures 6-1 and 6-7.
This is how we might allocate registers:

Register R0 - Display Panel addresses
Register R1 - Data input ADC address
Register R2 - Data table base address
Register R3 - Data table computed address
Register R4 - ASCII digit to be output
Register R5 - ASCII digit to be output
Register R6 & R7 - Undefined scratch registers

Our sample program is so small that it does not use scratchpad memory. General purpose registers provide sufficient read/write memory space. What is interesting about this observation is the fact that a carefully organized program of considerable size can get by with the read/write memory provided by the scratchpad. If your program makes extensive use of scratchpad, you should map out the way in which different areas of scratchpad will be allocated, just as we have done above for the general registers.

SUBROUTINE CREATION

Let us now examine some frequently used instruction sequences. Examples will be presented in this chapter as subroutines, since frequently used program sequences are most effectively implemented as subroutines. Note, however, that in order to convert an instruction sequence to a subroutine, you must provide a label for the entry point instruction and to add a return instruction at the point of exit — that is all.

DATA MOVEMENT SUBROUTINES

Moving blocks of data from one memory location to another is a frequently needed operation.

For example, frequently one memory buffer may be set aside for entry of data from all external sources. Depending on the external data source, the data may subsequently have to be moved to one of many permanent buffers.

Below are three subroutines that load a block of data from external memory into the scratchpad, from scratchpad into external memory, and from one external memory buffer to another. Registers have been arbitrarily assigned in these three subroutines; if you change register assignments, associated program changes are self-evident.

```
* SUBROUTINE TO LOAD A BLOCK OF DATA INTO SCRATCHPAD,
* FROM EXTERNAL MEMORY
*    R1 ADDRESSES START OF EXTERNAL MEMORY BUFFER
*    R2 ADDRESSES START OF SCRATCHPAD BUFFER.
*    R3 HOLDS BUFFER LENGTH
FMTS    INP    1          INPUT NEXT EXTERNAL MEMORY BYTE
        WRS    2          WRITE TO SCRATCHPAD
        INR    1          INCREMENT SOURCE ADDRESS
        INR    2          INCREMENT DESTINATION ADDRESS
        DRJ    3, FMTS    DECREMENT BYTE COUNT, RETURN FOR MORE
        RET               AT END, RETURN FROM SUBROUTINE
```

Before calling subroutine FMTS, registers R1, R2 and R3 must hold appropriate data. Observe that providing registers are used on a modular basis, as suggested earlier in this chapter, you will find that every subroutine does not have to be preceded with register loading instruction — registers will normally be correctly loaded.

```
* SUBROUTINE TO STORE A BLOCK OF DATA FROM SCRATCHPAD,
* INTO EXTERNAL MEMORY
*    R1 ADDRESSES START OF EXTERNAL MEMORY BUFFER
*    R2 ADDRESSES START OF SCRATCHPAD BUFFER
*    R3 HOLDS BUFFER LENGTH
FSTM    RDS    2          INPUT NEXT BYTE FROM SCRATCHPAD
        OUT    1          WRITE TO EXTERNAL MEMORY
        INR    1          INCREMENT SOURCE ADDRESS
        INR    2          INCREMENT DESTINATION ADDRESS
        DRJ    3, FSTM    DECREMENT BYTE COUNT, RETURN FOR MORE
        RET               AT END, RETURN FROM SUBROUTINE
```

Subroutine FSTM simply moves data in the opposite direction to subroutine FMTS.

Subroutines FMTM and FMTL move data between two external memory data buffers. External memory data buffers can be more than 256 bytes long, in which case a slightly different end of buffer testing technique must be employed, as illustrated by subroutine FMTL.

```
*  SUBROUTINE TO MOVE DATA BETWEEN EXTERNAL MEMORY
*  BUFFERS THAT ARE 256 BYTES IN LENGTH, OR LESS
*     R3 HOLDS BUFFER LENGTH
*     R4 HOLDS STARTING ADDRESS FOR SOURCE BUFFER
*     R5 HOLDS STARTING ADDRESS FOR DESTINATION BUFFER

FMTM    INP   4          INPUT NEXT SOURCE BYTE
        OUT   5          OUTPUT TO DESTINATION
        INR   4          INCREMENT BUFFER ADDRESSES
        INR   5
        DRJ   3, FMTM    DECREMENT COUNTER, RETURN FOR MORE
        RET              END OF SUBROUTINE
```

```
*  SUBROUTINE TO MOVE DATA BETWEEN EXTERNAL MEMORY
*  BUFFERS THAT ARE MORE THAN 256 BYTES IN LENGTH
*     R3 HOLDS BUFFER LENGTH
*     R4 HOLDS STARTING ADDRESS FOR SOURCE BUFFER
*     R5 HOLDS STARTING ADDRESS FOR DESTINATION BUFFER

FMTL    INP   4          INPUT NEXT SOURCE BYTE
        OUT   5          OUTPUT TO DESTINATION
        INR   4          INCREMENT BUFFER ADDRESS
        INR   5
        DRJ   3, FMTL    DECREMENT COUNTER, RETURN FOR MORE
        CPA   3          IF LOW ORDER EIGHT BITS ARE 0,
        JNZ   FMTL       TEST PAGE. IF NOT ZERO CONTINUE
        RET              RETURN AT END
```

SELF DEFINING DATA TABLES

There are message switching applications that must handle many tables of various sizes. A common technique assigns scratchpad memory locations to tables by table type. For example, an application may process raw ASCII text strings, ASCII digit strings, packed BCD data and pure binary data; each table type may have its own, reserved area of scratchpad memory.

Consider reserving the first two bytes of every table in external memory to define the table as follows:

First byte: Starting address in scratchpad for this table type.
Second byte: Table length.

Subroutines DMTS and DSTM move self defining data from memory to scratchpad, and from scratchpad to memory.

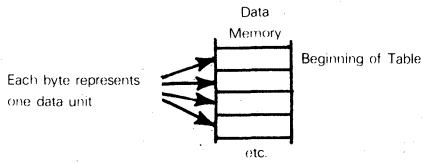
```
*  SUBROUTINE TO LOAD A BLOCK OF SELF DEFINING DATA INTO
*  SCRATCHPAD, FROM EXTERNAL MEMORY
*     R0 ADDRESSES THE FIRST PARAMETER BYTE IN EXTERNAL MEMORY
*     RI IS TO HOLD THE SCRATCHPAD STARTING ADDRESS
*     R2 IS TO HOLD THE BYTE COUNT
```

```
DMTS    LRN     1         LOAD SCRATCHPAD STARTING ADDRESS
        INR     0         INCREMENT EXTERNAL MEMORY ADDRESS
        LRN     2         LOAD BYTE COUNT
LOOP    INR     0         INCREMENT EXTERNAL MEMORY ADDRESS
        INP     0         INPUT NEXT EXTERNAL MEMORY BYTE
        WRS     1         WRITE TO SCRATCHPAD
        INR     1         INCREMENT SCRATCHPAD ADDRESS
        DRJ     2, LOOP   DECREMENT BYTE COUNT, RETURN FOR MORE
        RET               AT END, RETURN FROM SUBROUTINE
```

```
*  SUBROUTINE TO STORE A BLOCK OF SELF DEFINING DATA
*  FROM SCRATCHPAD, INTO EXTERNAL MEMORY
*     R0 ADDRESSES THE FIRST PARAMETER BYTE IN EXTERNAL MEMORY
*     R1 IS TO HOLD THE SCRATCHPAD STARTING ADDRESS
*     R2 IS TO HOLD THE BYTE COUNT
DSTM    LRN     1         LOAD SCRATCHPAD STARTING ADDRESS
        INR     0         INCREMENT EXTERNAL MEMORY ADDRESS
        LRN     2         LOAD BYTE COUNT
LOOP    INR     0         INCREMENT EXTERNAL MEMORY ADDRESS
        RDS     1         READ NEXT SCRATCHPAD MEMORY BYTE
        OUT     0         OUTPUT TO EXTERNAL MEMORY
        INR     1         INCREMENT SCRATCHPAD ADDRESS
        DRJ     2, LOOP   DECREMENT BYTE COUNT, RETURN FOR MORE
        RET               AT END, RETURN FROM SUBROUTINE
```

TABLE LOOKUPS

A data buffer may in reality be a data table. A data table may consist of a sequence of single byte data items:



Or individual units of a data table may be two or more bytes long:

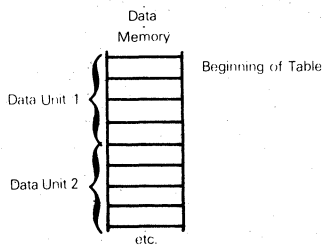


Table lookups are very easy to perform given the EA9002 instruction set. At the most general level, begin by loading the table base address into one general purpose register. Create the required table address by adding an index to the base address. In the following instruction sequence, the index is in the Accumulator, and the table consists of one byte data units:

TABLE INDEX

```
*LOAD TABLE BASE ADDRESS INTO R1
        LAI   TPAG    'TPAG IS A SYMBOL REPRESENTING BASE ADDRESS
                       PAGE
        CAP   1
        LRI   1,TADR  TADR IS A SYMBOL REPRESENTING BASE ADDRESS
                       WITHIN THE PAGE

(other instructions can follow here)

*ASSUMING TABLE INDEX IS IN THE ACCUMULATOR,
*CREATE TABLE ADDRESS IN R0
        ADD   1       ADD INDEX TO BASE ADDRESS
        CAR   0
        CPA   1       IF THERE IS A CARRY, INCREMENT
        JNC   NOCAR   THE PAGE
        IAC
NOCAR   CAP   0
*THE TABLE ADDRESS IS IN R0
```

Now suppose there are four bytes per data unit in the table. The index in the Accumulator must be multiplied by four before being added to the base address.

To multiply Accumulator contents by four, shift the Accumulator contents left two bit positions; however, that may lose the two high order bits. Therefore, save the two high order bits of the Accumulator for subsequent addition to the Page bits:

```
R1 contents:    P P P P A A A A A A A A
Accumulator:    0 0 x x x x x x x x 0 0
                                    ---
Table address:                      Sum
```

P, A and X represent any binary digits.

Here is the appropriate instruction sequence, assuming the table base address is already in R1:

```
*  ASSUMING TABLE INDEX IS IN THE ACCUMULATOR,
*  CREATE TABLE ADDRESS IN R0
        RAL           ROTATE ACCUMULATOR CONTENTS LEFT
        RAL           TWO BIT POSITIONS
        CAR   0       SAVE IN R0
*  PAGE BITS OF INDEX ARE NOW IN BIT POSITIONS
*  0 AND 1 OF R0, FOLLOWING THE TWO ROTATES
*  ISOLATE THESE TWO BITS IN THE ACCUMULATOR
*  BY ANDING WITH THE APPROPRIATE MASK
        LAI   3       LOAD MASK INTO ACCUMULATOR
        AND   0       ISOLATE PAGE BITS
        CAP   0       SAVE IN PAGE BITS OF R0
```

```
*  ACCUMULATOR NOW CONTAINS THE SAME BITS 0 AND 1
*  AS R0. ACCUMULATOR CONTAINS 0 IN ALL OTHER
*  BITS. XOR ACCUMULATOR WITH R0. THIS
*  WILL CLEAR BITS 0 AND 1, WHILE COPYING
*  BITS 2 THROUGH 7, ISOLATING THE SHIFTED
*  ADDRESS BITS
        XOR   0
        ADD   1       ADD TABLE BASE ADDRESS
        CAR   0       SAVE IN R0
        CPA   1       LOAD TABLE BASE PAGE TO ACCUMULATOR
        CAR   2       SAVE IN R2
        CPA   0       ADD SAVED HIGH ORDER ACCUMULATOR
        ADD   2       INDEX BITS
        CAP   0       TABLE ADDRESS IS IN R0
```

While the illustrated table lookups are general purpose, they are frequently more complex than they need be. **Providing you can keep table length below 256 bytes, origin tables at page boundaries.** Now the data bits of a general purpose register directly become the table index:

```
General Purpose register:  P P P P 0 0 0 0 0 0 0 0  Table base address
  Table lookup address:    P P P P x x x x x x x x
```

P and X represent any binary digits.

Initializing a table base address now simply involves correctly setting the page bits:

```
        LAI   TPAG    TPAG IS A SYMBOL REPRESENTING BASE
                      ADDRESS PAGE
        CAP   1       R1 WILL BE USED TO ADDRESS THE TABLE
```

Assuming the index is in the Accumulator, one instruction creates the required table address:

```
        CAR   1
```

Since the low order eight bits of the table base address are all 0, they do not need to be saved; thus R1 serves double duty as the register which holds the table base address and the required address.

DELAYS AND ONE-SHOTS

You can compute a time delay by executing instructions within a loop some fixed number of times. Consider this instruction sequence:

```
Cycles
  1          LRI   0,TIME   LOAD TIME CONSTANT
  1    LOOP  NOP            NO OPERATION
  2          DRJ   0,LOOP   DECREMENT TIME CONSTANT, RETURN
                            IF NOT END OF TIME LOOP
```

The length of the delay equals 3 * TIME + 2 cycles. Assuming a 2μ sec cycle time, the above program computes delays of 10 through 1540 μ sec, in 6 μ sec increments. Remember, TIME is an 8 bit value, with a maximum of 256_{10} ; the maximum value is achieved by loading 0 initially, since on first decrement it will go to FF. **For longer delays you can include additional NOP instructions within the loop:**

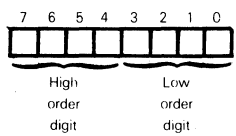
```
        LRI   0,TIME    LOAD TIME CONSTANT
LOOP    NOP
        NOP
        NOP
        DRJ   0,LOOP    DECREMENT TIME CONSTANT
```

For very long time constants you can loop within a loop:

```
        LRI   0,0       CLEAR R0
        LRI   1,TIME    SET TIME CONSTANT IN R1
LOOP    DRJ   0,LOOP    CREATE  1024μ SEC DELAY BY DECREMENTING R0
        DRJ   1,LOOP    CREATE TIME 1024μ SEC DELAYS
```

BCD DATA MANIPULATION

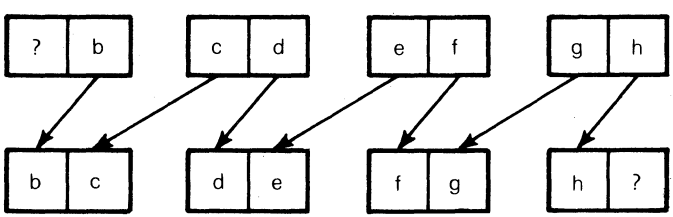
BCD data can be processed in four bit units by treating the low and high order four bits of every data byte as discrete data units:



BCD SHIFTS

Use of the RAR instruction in decimal mode allows multibyte BCD data to be shifted left or right in single digit increments.

A left shift may be illustrated as follows:

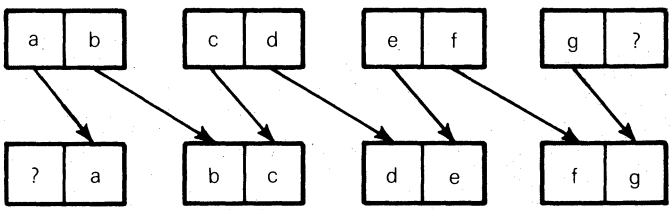


Assuming that R1 addresses the start of the BCD buffer and R2 holds the buffer byte count, here is the instruction sequence which performs the left shift:

```
        SED             SET DECIMAL MODE
        LRI   3 X 'F0'  LOAD HIGH ORDER DIGIT MASK
        LRI   4 Y '0F'  LOAD LOW ORDER DIGIT MASK
LOOP    INP     1       INPUT NEXT BYTE
```

```
        AND     4       SAVE LOW ORDER DIGIT IN R5
        CAR     5
        INR     1       INPUT NEXT BYTE
        INP     1
        AND     3       ISOLATE HIGH ORDER DIGIT
        IOR     5       ADD LOW ORDER DIGIT
        RAR             SWITCH DIGITS
        DCR     1       OUTPUT TO OVERSTORE PREVIOUS BYTE
        OUT     1
        INR     1
        DRJ     2,LOOP  RETURN FOR MORE BYTES
        SEB             SET BINARY MODE
```

A right shift may be illustrated as follows:

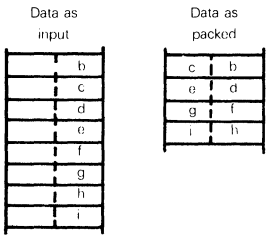


Assuming that R1 addresses the end of the BCD buffer and R2 holds the buffer byte count, here is the instruction sequence which performs the right shift:

```
        SED             SET DECIMAL MODE
        LRI     3 X 'F0' LOAD HIGH ORDER DIGIT MASK
        LRI     4 Y '0F' LOAD LOW ORDER DIGIT MASK
LOOP    INP     1       INPUT NEXT BYTE
        AND     3       SAVE HIGH ORDER DIGIT IN R5
        CAR     5
        DCR     1       INPUT NEXT BYTE
        INP     1
        AND     4       ISOLATE LOW ORDER DIGIT
        IOR     5       ADD HIGH ORDER DIGIT
        RAR             SWITCH DIGITS
        INR     1       OUTPUT TO OVERSTORE PREVIOUS BYTE
        OUT     1
        DCR     1
        DRJ     2,LOOP  RETURN FOR MORE BYTES
        SEB             SET BINARY MODE
```

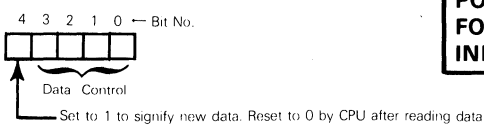
PACKING A BCD DATA STREAM

BCD digits, input as discrete 4-bit units, may be packed into bytes. This is what has to be done:



Assume that the four bit data is being input by an external device, via a 5-bit I/O port, as follows:

POLLING FOR DATA INPUT



This I/O port is addressed by R1. R0 addresses the beginning of the packed BCD buffer. Here is the appropriate instruction sequence:

```
        SED             SET DECIMAL MODE
        LRI     2,X '10' LOAD I/O CONTROL BIT MASK
BACK    JSR     IN      LOAD FIRST BCD DIGIT
        CAR     3       SAVE IN R3
        JSR     IN      LOAD SECOND BCD DIGIT
        RAR             MOVE TO HIGH ORDER FOUR BITS
        IOR     3       ADD LOW ORDER FOUR BITS
        OUT     0       OUTPUT PACKED BYTE
        INR     0       INCREMENT PACKED DATA BUFFER ADDRESS
        JUN     BACK

*SUBROUTINE TO INPUT A BCD DIGIT FROM I/O PORT
*ADDRESSED BY R1
IN      INP     1       INPUT PORT CONTENTS
        AND     2       ISOLATE CONTROL BIT
        JZE     IN      IF 0, RETURN AND TEST AGAIN
        INP     1       IF 1, LOAD DATA
        CAP     2       CLEAR CONTROL BIT BY MOVING
        CPA     2       LOW ORDER FOUR BITS TO AND FROM R2 PAGE
        OUT     1       OUTPUT TO I/O PORT TO CLEAR CONTROL
        RET             RETURN
```

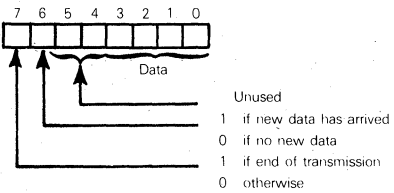
END TEST CONDITIONS

The program illustrated has a flow — it forms an endless loop with no logic to exit. Let us look at some ways in which we could test for the end of data transmission.

We could terminate the packed BCD data buffer on a page boundary. When the INR 0 instruction leaves zero in the data bits of R0, the end of transmission could be assumed. This is how our program changes:

```
        IRJ     0,BACK   INCREMENT PACKED DATA BUFFER ADDRESS

*Continue here if page boundary has been reached.
*This is the program exit.
```

The exit sequence illustrated above requires the program to call the end of data transmission. That is not always feasible. We can allow the external transmitting logic to identify the end of transmission by adding an extra control bit to the I/O port for this purpose:



Subroutine IN must be modified so that it tests for end of transmission. **The end of transmission bit will be returned in the carry status,** allowing the calling program to know whether or not transmission has ended. This is how subroutine IN must be modified:

```
IN      INP     1       INPUT PORT CONTENTS
        RAL             MOVE END OF TRANSMISSION BIT TO CARRY
        JCY     OUT     IF CARRY IS 1, TERMINATE
        RAL             MOVE NEW DATA BIT TO CARRY
        JNC     IN      IF 0, RETURN AND TEST AGAIN
        INP     1       IF 1, LOAD DATA
        CAP     2       CLEAR CONTROL BITS BY MOVING LOW
        CPA     2       ORDER FOUR BITS TO AND FROM R2 PAGE
        OUT     1       OUTPUT TO I/O PORT TO CLEAR CONTROL
        CLC             CLEAR CARRY FOR DATA RETURNED
OUT     RET
```

The calling program now tests for end of transmission via the carry status:

```
BACK    JSR     IN      LOAD FIRST BCD DIGIT
        JCY     NEXT    IF CARRY IS SET, EXIT
        CAR     3       SAVE IN R3
        JSR     IN      LOAD SECOND BCD DIGIT
        JCY     NEXT    IF CARRY IS SET, EXIT
        RAR             MOVE TO HIGH ORDER FOUR BITS
```

CONDITION TESTING

We have already tested a condition bit in the previous example, when determining if an I/O port has new data, or transmission has ended.

Being able to test conditions and status is a very important aspect of many microcomputer applications. These are the conditions that you may wish to test:

1. **A signal or status going from zero to one.**
2. **A signal or status going from one to zero.**
3. **A signal or status changing state.**

An effective way of handling signals and status is to assign dedicated scratchpad memory locations as status storage buffers. You can store the status of eight signals or condition indicators per byte.

BIT ISOLATION

To isolate a single bit out of any status buffer, AND the buffer contents with a mask that contains 1 bit in the bit location or locations that must be saved and 0 bits elsewhere. Assuming that scratchpad byte 5 holds status information, the following instruction sequence isolates within the Accumulator bit 3 of scratchpad byte 5:

```
LRI     1, 5    ADDRESS SCRATCHPAD VIA R1
LRI     2, 8    LOAD MASK INTO R2
RDS     1       MOVE FLAGS TO ACCUMULATOR
AND     2       MASK OUT ALL BAR BIT 3
```

This is the mask in R2:

```
0000    1000
  0       8
```

This is the result of the AND:

```
x x x x y x x x  in Accumulator
0 0 0 0 1 0 0 0  in R2
0 0 0 0 y 0 0 0  in Accumulator
```

BIT TESTING

Having isolated a single bit, you can use the JZE and JNZ instructions to determine subsequent program execution paths as a function of the isolated bit level.

RESETTING A BIT

In order to reset to 0 a single bit of any scratchpad byte, AND with a mask that contains 0 in all bit positions that must be reset and 1 elsewhere. For example, bit 2 of scratchpad byte 5 may be reset to 0 as follows:

```
LRI     1, 5     ADDRESS SCRATCHPAD VIA R1
LRI     2, X'FB' LOAD MASK INTO R2
RDS       1      MOVE FLAGS TO ACCUMULATOR
AND       2      CLEAR BIT 2
WRS       1      RETURN RESULT
```

This is the mask in R2:

This is the result of the AND

```
x x x x x x x x  in Accumulator
1 1 1 1 1 0 1 1  in R2
x x x x x 0 x x  in Accumulator
```

SETTING A BIT

In order to set to 1 a bit of any scratchpad byte, OR with a mask that contains 1 in all bit positions that must be unconditionally set to 1. The following instruction sequence unconditionally sets to 1 bits 5 and 1 of scratchpad byte 5:

```
LRI     1, 5     ADDRESS SCRATCHPAD VIA R1
LRI     2, X'22' LOAD MASK INTO R2
RDS       1      MOVE FLAGE INTO ACCUMULATOR
IOR       2      SET BITS 5 AND 1
WRS       1      RETURN RESULT
```

This is the mask in R2:

This is the result of the OR:

```
x x x x x x x x  in Accumulator
0 0 1 0 0 0 1 0  in R2
x x 1 x x x 1 x  in Accumulator
```

BIT CONDITION CHANGE TEST

If you want to test flags for change of state, then you must reserve two scratchpad bytes for each set of eight flags; one byte holds the old values while the other byte holds the new values. **Exclusive OR the old and new flag values** in order to determine flags which may have changed.

Suppose scratchpad bytes 5 and 6 contain the old and new flags values, respectively; the following instruction sequence leaves 1 in Accumulator bits for flags that have changed state:

```
LRI     1, 5    ADDRESS SCRATCHPAD VIA R1
RDS     1       LOAD OLD FLAGS INTO ACCUMULATOR
CAR     2       SAVE IN R2
INR     1       INCREMENT SCRATCHPAD ADDRESS
RDS     1       LOAD NEW FLAGS INTO ACCUMULATOR
XOR     2       EXCLUSIVE OR WITH OLD FLAGS
```

Suppose "new flags" are 01100101 and "old flags" are 11100001. At the conclusion of the instruction sequence illustrated above, R2 contains 11100001; the Accumulator contains 10000100:

```
new flags   0 1 1 0 0 1 0 1  in Accumulator
old flags   1 1 1 0 0 0 0 1  in R2
XOR         1 0 0 0 0 1 0 0  in Accumulator
```

To determine whether changed statuses went from 0 to 1 or from 1 to 0, AND the change indicator with the old status values:

```
                XOR   1 0 0 0 0 1 0 0  in Accumulator
           old flags  1 1 1 0 0 0 0 1  still in R2
                AND   1 0 0 0 0 0 0 0  in Accumulator
went from 1 to 0 ___|       |___ went from 0 to 1
```

All boolean instructions "clear" the CPU "C and H" status flags. This makes it very easy for instruction sequences that modify flags to be followed by jump on condition instructions.

Thus program logic can be controlled by signal or condition bit level changes.

Chapter 7
INPUT/OUTPUT PROGRAMMING

EA9002 Input/Output Programming is very straightforward; it is described in this chapter as Programmed I/O or Interrupt I/O.

PROGRAMMED I/O

The EA9002 Microcomputer has no special I/O instructions. As illustrated in Figure 6-2, external logic which communicates with the CPU must decode the address lines and respond to the general input and output instructions. The Input, Output, Load Register Indirect and Store Register Indirect instructions all serve double purpose as input/output instructions—providing the specified external addresses are decoded to select external logic.

To illustrate programmed I/O, refer to Figures 6-1 and 6-2. The following instruction sequence will input an 8-bit value from the ADC:

```
        LAI     X'FF'   SET I/O PAGE IN R0
        CAP     0
        CAR     0       SET ADC ADDRESS IN R0
        LRN     1       INPUT ADC VALUE TO R1
```

The following instruction sequence will output a series of five digits, assumed to reside in scratchpad bytes 20_{16} through 24_{16}, to the display panel:

```
        LAI     X'FF'   SET I/O PAGE IN R0
        CAP     0
        CAR     0       SET LOW PANEL ADDRESS IN R0
        LRI     I,X'20' SET SCRATCHPAD ADDRESS IN R1
        LRI     2,5     SET COUNTER IN R2
LOOP    RDS     1       INPUT NEXT DISPLAY DIGIT FROM SCRATCHPAD
        OUT     0       OUTPUT TO DISPLAY PANEL
        INR     0       INCREMENT ADDRESSES
        INR     1
        DRJ     2,LOOP  DECREMENT COUNTER, RETURN FOR MORE DIGITS
```

INTERRUPT I/O

As we stated at the beginning of Chapter 6, interrupts are used in microcomputer systems as a means of clocking slow, asynchronous events. In minicomputer and large computer applications, interrupts are also used as a means of sharing a central processing unit between a number of diverse applications with differing priorities. This use of interrupts within a microcomputer system

may be counterproductive; the overhead associated with such interrupt processing might cost more than implementing a multiple CPU system in which each potentially interrupting external source is assigned its own CPU.

A SINGLE INTERRUPT CONFIGURATION

Consider first a simple, one interrupt configuration. When this interrupt is acknowledged, current Program Counter contents are pushed onto the Stack and program execution branches to memory location 002.

If we are to assume that interrupts are constantly enabled, then we must also assume that the interrupt may occur at any time, with any register or status flag holding information which must be preserved. This being the case, **an instruction sequence beginning at memory location 002 must save the contents of all registers whose contents may be altered by the interrupt service routine; this will include Accumulator contents, status conditions and some, or all general purpose register contents.** This information may be saved in scratchpad memory, or in external RAM. In the program below, selected scratchpad memory bytes are reserved for this function; the first scratchpad byte is addressed by general purpose register R7. Here is the appropriate instruction sequence:

```
* INTERRUPT SERVICE ROUTINE
* INTERRUPT SERVICE PROGRAM INITIATION
* SAVE STATUS AND ALL REGISTERS IN SCRATCHPAD
* BYTES, BEGINNING WITH BYTE ADDRESSED BY R7
* R7 IS RESERVED FOR INTERRUPT PROCESSING
* IF R7 IS USED IN ANY OTHER WAY, INTERRUPTS MUST BE
* INHIBITED, AND R7 CONTENTS MUST BE SAVED, THEN RESTORED
        ORG     2
        WRS     7       SAVE ACCUMULATOR CONTENTS
        INR     7
        CSA             MOVE STATUS FLAGS TO ACCUMULATOR
        WRS     7       THEN SAVE IN SCRATCHPAD
        INR     7
        CRA     6       SAVE R6
        WRS     7
        INR     7
        CRA     5       SAVE R5
        WRS     7
        INR     7
        CRA     4       SAVE R4
        WRS     7
        INR     7
        CRA     3       SAVE R3, PAGE AND DATA
        WRS     7       BITS
        INR     7
        CPA     3
        WRS     7
        INR     7
        CRA     2       REPEAT FOR R2
        WRS     7
```

```
        INR     7
        CPA     2
        WRS     7
        INR     7
        CRA     1       REPEAT FOR R1
        WRS     7
        INR     7
        CPA     1
        WRS     7
        INR     7
        CRA     0       REPEAT FOR R0
        WRS     7
        INR     7
        CPA     0
        WRS     7
* INTERRUPT SERVICE PROGRAM FOLLOWS
```

Observe that since the page bits of registers R4, R5, R6 and R7 cannot be read into the Accumulator, they cannot be saved across an interrupt. Therefore, if the page bits of registers R4, R5, R6 or R7 contain meaningful data, do not increment or decrement these registers' contents following an interrupt; if you do, you may alter the page bits' contents.

Once the interrupt service program has completed execution, registers, status and Accumulator contents must be restored in the reverse order from which they were saved. Here is the completely general restoration instruction sequence:

```
* POST INTERRUPT REGISTERS AND STATUS RESTORATION
        RDS     7       RESTORE R0, PAGE AND DATA BITS
        DCR     7
        CAP     0
        RDS     7
        DCR     7
        CAR     0
        RDS     7       REPEAT FOR R1
        DCR     7
        CAP     1
        RDS     7
        DCR     7
        CAR     1
        RDS     7       REPEAT FOR R2
        DCR     7
        CAP     2
        RDS     7
        DCR     7
        CAR     2
        RDS     7       REPEAT FOR R3
        DCR     7
        CAP     3
        RDS     7
```

```
        DCR     7
        CAR     3
        RDS     7       RESTORE R4 DATA BITS
        DCR     7
        CAR     4
        RDS     7       REPEAT FOR R5
        DCR     7
        CAR     5
        RDS     7       REPEAT FOR R6
        DCR     7
        CAR     6
* READ STATUS. ASSUME CARRY AND DECIMAL
* MODES HAVE BEEN MODIFIED, THEREFORE MUST
* BE RESTORED.
        RDS     7       LOAD STATUSES
        DCR     7
        SED             SET DECIMAL MODE
        RLC             GET PREVIOUS DECIMAL MODE FLAG
        RLC
        JCY     DEC
        SEB             IF NOT SET, SET BINARY MODE
DEC     RRC             SHIFT PREVIOUS CARRY INTO CARRY
        RDS     7       RESTORE ACCUMULATOR
        ENI             ENABLE INTERRUPTS
        RET             RETURN
```

INTERRUPT WINDOWS

Another approach to interrupt handling is to keep interrupts disabled during the execution of program modules that use general purpose registers, status and the Accumulator. At the conclusion of every such module, and before executing the next module, interrupts are enabled for a short window which may consist of nothing more than two no operation instructions:

```
        —
        —
        —
        ENI             ENABLE INTERRUPTS
        NOP             INTERRUPT ACKNOWLEDGE WINDOW
        NOP
        DSI             DISABLE INTERRUPTS
        —
        —
        —
```

With just a little caution you can insure that interrupt enable windows are spaced not more than a few milliseconds apart — which identifies the maximum period for which an interrupt request could be denied and external logic could be kept waiting.

If the use of interrupt windows is acceptable, then the interrupt service and restoration routines disappear. Since there is nothing to save, there is no initial housekeeping instruction sequence; the actual interrupt service routine can begin at memory location

002. Since nothing has to be restored, the interrupt service routine can terminate with a simple RET instruction. The interrupt service routine now looks like this:

```
        ORG     002
* START INTERRUPT SERVICE PROGRAM
        —
* END INTERRUPT SERVICE PROGRAM
        ENI             ENABLE INTERRUPTS
        RET             RETURN FROM INTERRUPT
```

DELAY LOOPS

INTERRUPTING OUT OF DELAY LOOPS

Frequently in microcomputer applications, real time events occur too slowly to keep the microcomputer completely busy. When idle, such applications usually will execute a simple delay loop as follows:

```
* DELAY LOOP EXECUTED WHILE MICROCOMPUTER IS IDLE

DELAY   JUN     DELAY
```

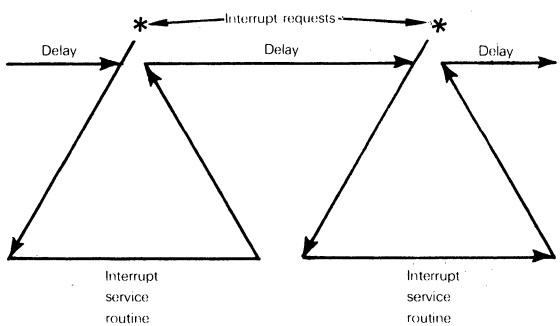
The period during which the delay loop is being executed is an ideal time for interrupts to be enabled; in fact, acknowledgment of an interrupt is the only way in which program execution can leave the delay loop.

A microcomputer application that uses the delay loop may be illustrated by an enhancement of the configuration illustrated in Figure 6-2.

Consider a dozen or more ADC's being monitored by a single EA9002 CPU.

It is conceivable that a number of seconds may elapse between any ADC receiving data to transmit to the CPU. Thus the CPU spends the bulk of its time executing the delay loop illustrated above.

When an ADC does have a new reading to transmit to the CPU, it will request an interrupt. The NAND gate daisy chain described in Chapter 4 can be used to interface a number of interrupt requesting ADC's to the CPU. Now all programs executing outside the delay loop are interrupt service routines. In other words, all non-trivial program execution is confined to interrupt service routines; and at the conclusion of every interrupt service routine, program execution will return to the delay loop. The program execution sequence which results may be illustrated as follows:



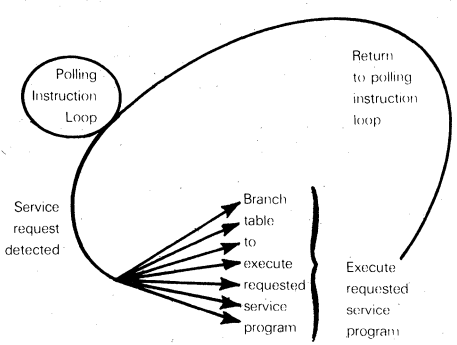
The actual program which services each interrupt will be structured as follows:

```
        ORG     2
*ALL INTERRUPT SERVICE ROUTINES START HERE
*LOCATE SOURCE OF INTERRUPT
        —
        —
        —
        —
*BRANCH TO PROGRAM THAT SERVICES IDENTIFIED
*INTERRUPTING SOURCE
        —
        —
        —
        —
*EXECUTE INTERRUPT SERVICE PROGRAM
        —
        —
        —
        —
*ALL INTERRUPT SERVICE PROGRAMS END WITH
*THE FOLLOWING TWO INSTRUCTIONS
        ENI             ENABLE INTERRUPTS
        RET             RETURN FROM INTERRUPT
```

I/O POLLING AS A SUBSTITUTE FOR INTERRUPTS

When a microcomputer program consists of a no operation loop with interrupt exits, as we have just described, an alternative is to execute a polling program instead of the no operation loop. Interrupts are eliminated.

Consider again numerous ADC's and panel displays being serviced by a single EA9002 CPU. Now the microprocessor will spend the bulk of its time executing a program which tests each ADC in turn, until it locates one that is ready to transmit data. At that time a branch will occur to the instruction sequence supporting the particular ADC requesting service. Once this ADC has been serviced, program execution returns to the polling routine. This sequence may be illustrated as follows:



Let us assume that there are 16 ADC's, with addresses FEO_{16} through FEF_{16}. Each ADC specifies that it has data to send by storing a nonzero value in its addressed buffer. The CPU clears this buffer after reading its contents.

Here is the necessary polling program which substitutes for a no operation sequence followed by an interrupt exit. Real memory addresses have been arbitrarily selected since they make the sample program easier to follow.

```
* PROGRAM INITIALIZATION
* R1 WILL ADDRESS ADC's
* R2 WILL ADDRESS BRANCH TABLE
* SET UP PAGE PORTION OF EACH ADDRESS
START   LAI     X'0F'     POLL PAGE IS F
        CAP     1
        LAI     04        BRANCH TABLE PAGE IS 4
        CAP     2
POLL    LRI     1,X'E0'   INITIALIZE POLL ADDRESS
        LRI     2,X'80'   INITIALIZE BRANCH TABLE BASE ADDRESS
LOOP    INP     1         INPUT CONTENTS OF NEXT ADC BUFFER
        JNZ     BRANCH    IF A NON ZERO VALUE IS INPUT, BRANCH OUT
        INR     1         INCREMENT POLL ADDRESS
        INR     2         DOUBLE INCREMENT BRANCH TABLE ADDRESS
        INR     2
        LAI     X'F0'     COMPARE POLL ADDRESS FOR END OF LOOP
        CMP     1
        JNE     LOOP      NOT END OF 16 DEVICES LOOP
        JUN     POLL      END OF 16 DEVICES LOOP
* A NON ZERO VALUE HAS BEEN INPUT. THE DEVICE
* CAN BE IDENTIFIED BY THE CURRENT BRANCH
* TABLE ADDRESS POINTER. LOAD THE
* APPROPRIATE BRANCH TABLE ADDRESS, PAGE AND
* ADDRESS WITHIN PAGE, INTO R0
BRANCH  JIN     2         JUMP INTO BRANCH TABLE
        ORG     X'480'
* BRANCH TABLE IS ORIGINED AT 480(16).
        JUN     ADDR1     16 UNCONDITIONAL JUMPS
        JUN     ADDR2     TO 16 SERVICE ROUTINES
        JUN     ADDR3     THESE JUMP INSTRUCTIONS
        JUN     ADDR4     CONSTITUTE A BRANCH TABLE
        JUN     ADDR5
        etc
        —
        —
        —
        ORG     ADDR1
* FIRST SERVICE PROGRAM
        —
        —
        —
        JUN     START     END OF FIRST SERVICE PROGRAM
        ORG     ADDR2
```

```
* SECOND SERVICE PROGRAM
        —
        —
        —
        JUN     START   END OF SECOND SERVICE PROGRAM
        —
        —
        —
        ETC
```

There are two parts to the polling program illustrated above; there is the detection of an external device requesting service and there is the branch table which follows.

BRANCH TABLE

The branch table requires special mention since it **is a very common microcomputer feature, with universal application. The purpose of a branch table is to allow logic to select one of many execution paths based on a sensed status,** which in this case happens to be an ADC device number. The logic of a branch table may be illustrated by the following completely general instruction sequence:

```
        ORG     BTBL
* THIS BRANCH TABLE CONSISTS OF A NUMBER OF
* ADDRESSES. THE SYMBOL BTBL REPRESENTS
* THE MEMORY ADDRESS OF THE FIRST BRANCH TABLE BYTE
* EACH ADDRESS OCCUPIES TWO BYTES. THE
* FIRST BYTE SPECIFIES A PAGE. THE SECOND
* BYTE SPECIFIES AN ADDRESS WITHIN THE PAGE.
        DC      ADDR1
        DC      ADDR2
        DC      ADDR3
        —
        —
        —
        —
        ETC
        ORG     PROG
* THE BRANCH TABLE PROGRAM ITSELF FOLLOWS.
* ASSUME THAT THE BRANCH TABLE PROGRAM IS ENTERED
* WITH A NUMBER IN THE ACCUMULATOR. THIS
* NUMBER SPECIFIES THE ADDRESS WITHIN THE
* BRANCH TABLE TO WHICH A JUMP MUST
* OCCUR. FOR EXAMPLE, 3 MEANS THAT
* A JUMP TO ADDR3 MUST OCCUR.
* SINCE EACH BRANCH TABLE ADDRESS OCCUPIES
* TWO BYTES, MULTIPLY THE ACCUMULATOR CONTENTS
* BY 2. ASSUME THAT THE ACCUMULATOR
* CONTAINS 7F OR LESS.
        RAL
        CLC
```

```
* CREATE THE BRANCH TABLE ADDRESS IN R1.
        CAR     R1      MOVE ADDRESS INDEX TO R1
        LAI     BTAD    LOAD ADDRESS PORTION OF BTBL
        ADD     1       ADD INDEX, CURRENTLY IN R1
        CAR     1       RETURN SUM TO R1
        LAI     BTPG    LOAD PAGE PORTION OF BTBL
        JNC     NEXT    INCREMENT IF CARRY IS SET
        IAC
NEXT    CAP     1       MOVE TO PAGE BITS OF R1
        INP     1       LOAD ADDRESS INTO R0
        CAP     0
        INR     1
        INP     1
        CAR     0
        JIN     0       JUMP TO SELECTED ADDRESS
```

Chapter 8
SOME USEFUL SUBROUTINES

This chapter provides a number of frequently used subroutines and instruction sequences.

MATHEMATICAL SUBROUTINES

ADDITION AND SUBTRACTION

Binary or decimal addition and subtraction can use the same subroutines, providing initial conditions are met.

For binary addition or subtraction, the input data must be in pure binary form, signed or unsigned. The binary/decimal flag must be set to binary mode.

For decimal addition or subtraction, the input data must be in BCD form. The binary/decimal flag must be set to decimal mode.

Below are multibyte addition and subtraction subroutines.

```
* BINARY OR DECIMAL MULTIBYTE SCRATCHPAD ADDITION.
*    R1 ADDRESSES LOW AUGEND BYTE
*    R2 ADDRESSES LOW ADDEND BYTE
*    R3 ADDRESSES LOW ANSWER BYTE
*    R4 HOLDS BYTE COUNT
ADD     CLC             CLEAR CARRY
LOOP    RDS     1       LOAD NEXT AUGEND BYTE
        ADS     2       ADD NEXT ADDEND BYTE
        WRS     3       STORE ANSWER
        INR     1       INCREMENT ADDRESSES
        INR     2
        INR     3
        DRJ     4,LOOP  DECREMENT BYTE COUNT, RETURN FOR MORE
        RET             AT END, RETURN FROM SUBROUTINE
* BINARY OR DECIMAL MULTIBYTE SCRATCHPAD SUBTRACTION
*    R1 ADDRESSES LOW SUBTRAHEND BYTE
*    R2 ADDRESSES LOW MINUEND BYTE
*    R3 ADDRESSES LOW ANSWER BYTE
*    R4 HOLDS BYTE COUNT
SUB     CLC             CLEAR CARRY
LOOP    RDS     1       LOAD NEXT SUBTRAHEND BYTE
        SUS     2       SUBTRACT NEXT MINUEND BYTE
        WRS     3       STORE ANSWER
        INR     1       INCREMENT ADDRESSES
```

```
INR     2
INR     3
DRJ     4,LOOP   DECREMENT BYTE COUNT, RETURN FOR MORE
RET              AT END, RETURN FROM SUBROUTINE
```

Note that subroutines ADD and SUB can return the answer to either of the source buffers. Simply eliminate R3, and modify the "WRS 3" instruction appropriately to "WRS 2" or "WRS 1" Of course, the "INR 3" instruction is also eliminated.

BINARY MULTIPLICATION

Any multiplication can be implemented by repeated addition; for example, 25 x 31 = 775 can be created by adding 25 thirty-one times to a buffer which initially holds zero. But you would never use this simple scheme to perform binary multiplication, it takes too long to execute.

Remember, binary digits can have a value of 0 or 1. **At the digit level,** therefore, **multiplication degenerates to addition, since "multiply by 1" is the same as "add". This makes a "shift-and-add" algorithm work for binary multiplication.**

Consider B5 * 6D, the binary representation may be illustrated thus:

Multiplier = 0 1 1 0 1 1 0 1
Multiplicand = 1 0 1 1 0 1 0 1

			RESULT HIGH ORDER BYTE	LOW ORDER BYTE
	Start:		0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 1 1 0 1 1 0 1	Step 1 (a)		1 0 1 1 0 1 0 1	0 0 0 0 0 0 0 0
	1 (b)		0 1 0 1 1 0 1 0	1 0 0 0 0 0 0 0
0 1 1 0 1 1 0 1	Step 2 (a,b)		0 0 1 0 1 1 0 1	0 1 0 0 0 0 0 0
0 1 1 0 1 1 0 1	Step 3 (a)		1 0 1 1 0 1 0 1	
			1 1 1 0 0 0 1 0	0 1 0 0 0 0 0 0
	3 (b)		0 1 1 1 0 0 0 1	0 0 1 0 0 0 0 0
0 1 1 0 1 1 0 1	Step 4 (a)		1 0 1 1 0 1 0 1	
		c→1	0 0 1 0 0 1 1 0	0 0 1 0 0 0 0 0
	4 (b)		1 0 0 1 0 0 1 1	0 0 0 1 0 0 0 0
0 1 1 0 1 1 0 1	Step 5 (a,b)		0 1 0 0 1 0 0 1	1 0 0 0 1 0 0 0
0 1 1 0 1 1 0 1	Step 6 (a)		1 0 1 1 0 1 0 1	
			1 1 1 1 1 1 1 0	1 0 0 0 1 0 0 0
	6 (b)		0 1 1 1 1 1 1 1	0 1 0 0 0 1 0 0
0 1 1 0 1 1 0 1	Step 7 (a)		1 0 1 1 0 1 0 1	
		c→1	0 0 1 1 0 1 0 0	0 1 0 0 0 1 0 0
	7 (b)		1 0 0 1 1 0 1 0	0 0 1 0 0 0 1 0
0 1 1 0 1 1 0 1	Step 8 (a,b)		0 1 0 0 1 1 0 1	0 0 0 1 0 0 0 1
			4 D	1 1

This algorithm has eight steps, one for each bit of the multiplier and multiplicand.

Each step has two parts. Part (a) tests the status of the next multiplier bit; part (b) adds the multiplicand to the result if part (a) locates a 1 bit.

If part (b) is needed, the multiplicand must be added to the correct eight bits of the 16 bit result. The high order eight bits of the result space is always assumed to constitute a "window" on the required eight bits; for this to be the case, the high order eight bits must initially contain the low order eight answer bits:

	H.O. Byte	L.O. Byte
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Answer bits:	7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8

After each step, Part (a), the answer 16 bits must be rotated right one bit.

For example, we enter Step 2 as follows:

	H.O. Byte	L.O. Byte
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Answer bits:	8 7 6 5 4 3 2 1	0 15 14 13 12 11 10 9

After eight shifts, we will have the correct final bit configuration for the answer:

	H.O. Byte	L.O. Byte
	7 6 4 4 3 2 1 0	7 6 5 4 3 2 1 0
Answer bits:	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

Now consider a program to implement this 8-bit multiplication algorithm.

Assume that Registers R4 and R5 hold the 8-bit multiplier and multiplicand, respectively. The 16-bit result will be stored in the data bits of registers R7 (high order) and R6 (low order).

The algorithm clears R7 and R6, and assumes that the 8 data bits of R7 act as the "window" on the 16-bit answer. Initially, therefore, R7 holds the low order 8 bits.

With each step of the multiplication, the next low order multiplier bit is tested, these are the shaded bits in the leftmost column of numbers illustrated above. If the tested bit is 1, then the multiplicand is added to R7.

Whether or not the multiplicand is added to R7, the contents of R7 and R6 data bits are rotated right one bit, as a 16 bit unit. Thus after 8 shifts for the eight binary digits of the source numbers, R7 becomes the high order 8 bits and R6 holds the low order 8 bits.

```
* 8 BIT MULTIPLICATION PROGRAM USING FIVE REGISTERS
MUL 4   LRI   7,0    CLEAR GENERAL PURPOSE REGISTERS.
        LRI   6,0    R7 AND R6 TO HOLD THE ANSWER.
        LAI   7      LOAD BIT COUNT INTO ACCUMULATOR.
LOOP    XCH   4      SHIFT LOW ORDER BIT OF R4 INTO CARRY
        RRC          TO TEST ITS STATUS.
        XCH   4
        XCH   7      IF BIT IS 1, ADD R5 TO R7
        JNC   NOAD   IF BIT IS 0, BYPASS ADDITION
        CLC
        ADD   5
        JNC   NOAD   IF THERE IS A CARRY, INCREMENT R6
        INR   6
NOAD    CLC          ROTATE ACCUMULATOR - R6 16-BIT UNIT
```

```
        RRC             RIGHT ONE BIT. FIRST CLEAR CARRY
        XCH     7       THEN ROTATE ACCUMULATOR RIGHT ONE BIT
        XCH     6       NOW MOVE R6 TO ACCUMULATOR AND
        RRC             ROTATE RIGHT WITH CARRY.
        XCH     6
        JNC     NOST    IF CARRY IS NOT SET, THIS STEP IS FINISHED
        XCH     7       IF CARRY IS SET, SET BIT 7 OF R7 TO 1
        RLC
        SEC
        RRC
        XCH     7
NOST    DAC             DECREMENT BIT COUNTER IN ACCUMULATOR
        JNZ     LOOP    CONTINUE IF NOT END
        RET             RETURN AT END.

* 8 BIT MULTIPLICATION PROGRAM USING FIVE REGISTERS
MUL 5   LRI     7, 0    CLEAR GENERAL PURPOSE REGISTERS
        LRI     6, 0    R7 AND R6 TO HOLD THE ANSWER
        LAI     1       LOAD BIT MASK INTO ACCUMULATOR
LOOP    CAR     3       SAVE IN R3
        AND     4       TEST NEXT R4 BIT
        JZE     NOAD    IF BIT IS 0, BYPASS ADDITION
        CRA     5       BIT IS 1, SO ADD R5 TO R7
        ADD     7
        CAR     7
        JNC     NOAD    IF THERE IS A CARRY, INCREMENT R6
        INR     6
NOAD    RRC             MOVE THE LOW ORDER ACCUMULATOR BIT INTO CARRY
        CRA     6       ROTATE R7-R6 RIGHT ONE BIT AS
        RRC             A 16 BIT UNIT
        CAR     6
        CRA     7
        RRC
        CAR     7
        CRA     3       SHIFT BIT MASK LEFT ONE BIT
        RAL
        JNC     LOOP    IF CARRY IS NOT SET, CONTINUE TO NEXT BIT
        RET             IF CARRY IS SET, END.
```

Two multiplication subroutines are illustrated.

The first multiplication subroutine is longer, but it uses only four registers. To do this, the bit counter is held in the Accumulator; XCH instructions save the counter whenever the Accumulator must operate on data.

The second multiplication subroutine uses R3 to hold a bit mask, in lieu of a bit counter. The mask initially identifies the low order bit:

00000001

on the eighth rotate right, the 1 will shift into the carry, triggering an exit from the subroutine. **This second subroutine is five instructions shorter, but it uses an additional general purpose register.**

BINARY DIVISION

When dealing with small numbers, simple binary division is most effectively implemented using multiple subtractions. Continuously subtract the divisor from the dividend until a negative answer is detected.

Add the divisor to the negative answer in order to determine the remainder.

The answer is equal to the number of subtractions performed, less one, before a negative result was generated.

DATA HANDLING PROGRAMS

Programs in this category convert data from one coded form to another. Recall that data may be interpreted as binary, BCD, or ASCII.

Binary and BCD data represent numeric information; therefore, only the numeric subset of ASCII characters is relevant when making binary-ASCII or BCD-ASCII conversions.

HEXADECIMAL — ASCII CONVERSION

The following subroutine converts a hexadecimal digit into its ASCII equivalent, using a table lookup.

```
* ROUTINE TO CONVERT HEXADECIMAL TO ASCII
*     ENTER WITH HEXADECIMAL VALUE IN REGISTER 5
*     EXIT WITH ASCII CHARACTER IN ACCUMULATOR
HASC  LAI  TBPG  GET PAGE VALUE OF CONVERSION TABLE
      CAP  5     SET PAGE
      LAI  TBAD  GET WORD ADDRESS OF TABLE START
      ADD  5     ADD CHARACTER VALUE (TABLE DOES NOT OVERLAP PAGE
                 BOUNDARY)
      CAR  5     SET ADDRESS
      INP  5     GET ASCII CHARACTER
      RET        RETURN FROM SUBROUTINE
Table DC   '0'
      DC   '1'
      DC   '2'
      DC   '3'
      DC   '4'
      DC   '5'
      DC   '6'
      DC   '7'
      DC   '8'
      DC   '9'
      DC   'A'
      DC   'B'
      DC   'C'
      DC   'D'
      DC   'E'
      DC   'F'
      END
```

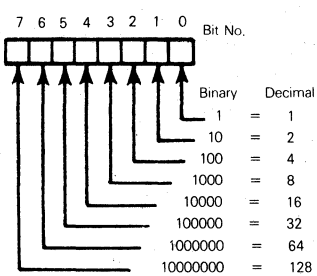
Next, a subroutine is given to convert ASCII digits 0-9 and A-F to hexadecimal equivalents.

```
*  ROUTINE TO CONVERT ASCII 0-9 AND A-F TO HEXADECIMAL EQUIVALENT
*       ENTER WITH CHARACTER IN ACCUMULATOR
*       EXIT WITH VALUE IN ACCUMULATOR
*       NON-HEXADECIMAL CHARACTER CAUSE JUMP TO ERR
AHEX    LRI     6,X'7F'     LOAD MASK FOR PARITY BIT
        AND     6           MASK OFF PARITY AND CLEAR CARRY
        LRI     6,X'30'     LOAD ASCII ZERO IN R6
        SUB     6           IS CHARACTER < 0
        JLT     ERR         YES, ERROR
        LRI     6,9         TEST FOR 9
        CMP     6           IS IT 0-9?
        JLE     GOOD        YES, GOOD NUMBER
        LRI     6,X'11'     NO, LOAD DIFFERENCE BETWEEN
        SUB     6           ASCII A AND 0 AND SUBTRACT
        JLT     ERR         ERROR IF BETWEEN 9 AND A
        LRI     6,5         TEST FOR A-F
        CMP     6           IS IT > F
        JGT     ERR         YES, ERROR
        CLC                 CLEAR CARRY
        ADD     6
        ADD     6           NO, ADD 10
GOOD    RET                 RETURN FROM SUBROUTINE
ERR     EQU     *
*
```

BINARY — BCD CONVERSIONS

Two programs are described for converting binary data to its BCD equivalent.

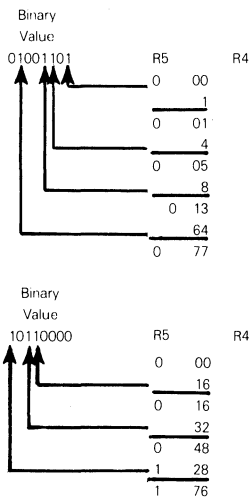
The first program takes eight data bits of a general purpose register, illustrated in the first program as R4 and creates three decimal digits in two general purpose registers, illustrated as R4 and R5. The conversion technique is based on the fact that each binary digit within an 8 bit unit has a decimal representation, as follows:



The first program tests each individual binary digit position of the initial binary value. Upon detecting a 1 in any binary bit position, the decimal equivalent is added to R4, the BCD low order decimal digits space — which must initially be cleared. Recall that BCD addition is easy to do, since the EA9002 has decimal addition logic. The most significant decimal digit is created by incrementing R5 whenever decimal addition to R4 creates a carry.

The eight decimal equivalents for each binary digit position are stored in eight contiguous scratch-pad bytes. Because the high order binary digit is equivalent to 128 decimal, $C8_{16}$ is stored. When this value is added to zero, the EA9002 automatically corrects the "C" to a "2" and generates a Carry. The result is 28 in the low order register, and a carry. The carry is detected and causes the high order register to be incremented by 1.

Here are two examples of how the first program's logic works:



Here is the program:

```
* CONVERT THE BINARY CONTENTS OF REGISTER 4 TO THREE BCD DIGITS.
* STORE THE LEAST SIGNIFICANT TWO DIGITS IN REGISTER 4 AND THE MOST
* SIGNIFICANT DIGIT IN BITS 0-3 OF REGISTER 5. SCRATCH MEMORY
* LOCATIONS 1-8 CONTAIN THE FOLLOWING HEX VALUES, RESPECTIVELY:
* 01, 02, 04, 08, 16, 32, 64, C8
BBCD    LRI   7,8     SET BIT COUNTER AND CONSTANT TABLE POINTER
        CLA           ZERO ACCUMULATOR
        CAR   5       ZERO REGISTER 5
        XCH   4       ZERO REGISTER 4, LOAD BINARY VALUE
        SED           SET DECIMAL MODE
CON1    RAL           TEST NEXT BIT
        JNC   CONB    NOT SET, GO DECREMENT BIT COUNTER
        XCH   4       SET, EXCHANGE FOR RUNNING BCD VALUE
        ADS   7       ADD CONSTANT CORRESPONDING TO THIS BIT
```

```
        JNC     CON2    IF NO CARRY, GO ON
        INR     5       CARRY, INCREMENT MOST SIGNIFICANT DIGIT
CON2    XCH     4       EXCHANGE BCD VALUE FOR SHIFTED BINARY
CON3    DRJ     7,CON1  DECREMENT BIT COUNTER, NOT DONE GO DO NEXT
                        BIT
        SEB             RESTORE BINARY MODE
        RET             RETURN FROM SUBROUTINE
```

The next program converts sixteen binary digits into five BCD digits.

```
* CONVERT THE BINARY CONTENTS OF TWO CONSECUTIVE MEMORY LOCA-
TIONS TO
* FIVE BCD DIGITS AND STORE IN THREE CONSECUTIVE MEMORY LOCATIONS.
* THE BINARY DATA IS STORED WITH THE LEAST SIGNIFICANT HALF AT THE
* LOWER ADDRESS. THE BCD DATA IS STORED WITH THE LEAST SIGNIFICANT
* DIGIT IN THE LOWER HALF OF THE BYTE AT THE LOWEST ADDRESS, AND
* THE MOST SIGNIFICANT DIGIT IN THE LOWER HALF OF THE BYTE AT THE
* HIGHEST ADDRESS. THE BCD RESULT MAY OVERLAP THE ORIGINAL BINARY
* DATA.
        LADR    7,BNRY  LOAD ADDRESS OF BINARY L.S. BYTE
        INP     7       READ IT
        CAR     2       SAVE IN R2
        INR     7       INCREMENT ADDRESS
        INP     7       READ M.S. BYTE
        CAR     7       SAVE IN R7
        CLA             ZERO ACCUMULATOR
        CAR     4       ZERO
        CAR     5       INITIAL
        CAR     6       BCD DIGITS
        LADR    3,CNVTB LOAD ADDRESS OF CONVERSION TABLE
        CRA     2       GET L.S. BYTE
        SED             SET DECIMAL MODE
CNVT4   LRI     1,8     SET BIT COUNTER
CNVT5   RRC             GET NEXT BIT
        JNC     CNVT7   IF NOT SET, GO ON
*                       SET, GET CORRESPONDING POWER OF 2 AND ADD
*                       TO RUNNING BCD VALUE
        CAR     2       SAVE BINARY
        INP     3       GET DIGITS 1 AND 2
        ADD     4       ADD RUNNING DIGITS 1 AND 2
        CAR     4       SAVE
        INR     3       NEXT BYTE
        INP     3       GET DIGITS 3 AND 4
        ADD     5       ADD RUNNING DIGITS  AND 4
        CAR     5       SAVE
        INR     3       NEXT BYTE
        INP     3       GET DIGIT 5
        ADD     6       ADD RUNNING DIGIT 5
        CAR     6       SAVE
```

```
         CRA   2          RESTORE BINARY
CNVT6    INR   3          NEXT BYTE
         DRJ   1,CNVT5    DECREMENT BIT COUNTER, NOT DONE GO DO NEXT
                          BIT
         CLA              DONE, TEST FOR COMPLETION OF BOTH HALVES
         XCH   7          ZERO R7 AND GET PREVIOUS VALUE
         JNZ   CNVT4      NOT DONE. DO SECOND HALF
         LADR  0,BCDVAL
                          LOAD ADDRESS WHERE RESULT IS TO BE STORED
         SRN   4          STORE DIGITS 1 AND 2
         INR   0          NEXT BYTE
         SRN   5          STORE DIGITS 3 AND 4
         INR   0          NEXT BYTE
         SRN   6          STORE DIGIT 6
*        (FALL THROUGH)
CNVT7    INR   3          SKIP 2 CONVERSION TABLE BYTES
         INR   3
         JUN   CNVT6      RE-ENTER MAIN LINE
BNRY     EQU   450
BCDVAL   EQU   460
CNVTB    EQU   470
*
*
```

Appendix A

AN INSTRUCTION SET SUMMARY

Appendix A

AN INSTRUCTION SET SUMMARY

EA9002 instructions are summarized by instruction type, for quick reference once you are familiar with the EA9002 instruction set.

Within this appendix, symbols and abbreviations are used as follows:

A - Accumulator stated
AC - Accumulator
ACL - Low order four Accumulator bits
C - Carry status
D - Decimal status
GDN - Data bits of general purpose register N
GPN - Page bits of general purpose register N
GRN - All twelve bits of general purpose register N
H - Half carry status
I1 - First, or only byte of instruction object code
I2 - Second byte of instruction object code
IL - Low order four bits of first, or only object code byte
MGRN - External memory byte addressed by all twelve bits of GRN
PC - Program counter
SW - Status word
SGDN - Scratchpad byte addressed by low order six bits of GDN
[ ] - Contents location enclosed by brackets
← - Move data in indicated direction
↔ - Exchange data
\+ Add
\- Subtract
∧ AND
∨ OR
⊻ XDR

Under status flag columns only:
X - specifies the status is set or reset to reflect the results of instruction execution
1 - the flag is unconditionally set to 1
0 - the flag is unconditionally set to 0

A blank space implies that the flag is not modified in any way.

For the operand field, these abbreviations are used:
DATA - An 8-bit, binary data value
LABEL - A 17-bit address
N - A digit in the range 0 through 7, specifying a register number

Object code is normally identified in hexadecimal digits; if binary digit options need to be defined at the binary digit level, then object code is identified using binary digits. These special symbols are used:

P - Variable hexadecimal digit representing the page of an address
QQ - Two variable hexadecimal digits representing the low order eight bits of as memory address
X - Variable binary digit
ZZ - Two variable hexadecimal data digits

Table A-1. An Instruction Set Summary

	MNEMONIC	OPERAND(S)	OBJECT CODE	MPU CYCLE (NOTE 1)	STATUSES C	D	A	H	OPERATIONS PERFORMED
I/O OR PRIMARY MEMORY REFERENCE	INP	N	01010XXX				X		[AC]←[MGRN] Input to Accumulator
	LRN	N	11100XXX						[GDN]←[MGRO] Load Register, Register O indirect
	OUT	N	01011XXX						[MGRN]←[AC] Output from Accumulator
	SRN	N	11101XXX						[MGRO]←[GDN] Store Register, Register O indirect
PRIMARY SCRATCHPAD REFERENCE	RDS	N	11010XXX				X		[AC]←[SGDN] Read scratch memory to Accumulator
	WRS	N	11011XXX						[SGDN]←[AC] Write scratch memory to Accumulator
SECONDARY SCRATCHPAD REFERENCE	ADS	N	11000XXX		X		X	X	[AC]←[AC] · [SGDN] · [C] Decimal or binary addition specified by D
	SUS	N	11001XXX	1(2)	X		X	X	[AC]←[AC]-[SGDN]-[C] Subtract scratchpad from Accumulator
IMMEDIATE	LAI	DATA	OD ZZ	2			X		[AC]←ZZ Load Accumulator immediate
	LRI	N,DATA	01100XXX ZZ	2					[GDN]←ZZ Load Register immediate
JUMP	JIN	N	01101XXX QQ	1					[PC]←[GRN] Jump unconditional, register indirect
	JUN	LABEL	1P QQ	2					[PC]←PQQ Jump unconditional
	JSR	LABEL	2P QQ	2					[STACK]←[PC], [PC]←PQQ Jump to subroutine

Table A-1. An Instruction Set Summary (Continued)

TYPE	MNEMONIC	OPERAND(S)	OBJECT CODE	MPU CYCLE (NOTE 1)	STATUSES C D A H	OPERATIONS PERFORMED
JUMP ON CONDITION	DRJ	N,LABEL	00111XXX QQ	2		[GRN]←[GRN] - FFF If [GDN]=0, [PC]←QQ else [PC]←[PC + 2] Decrement register and jump
	IRJ	N,LABEL	00110XXX QQ	2		[GRN]←[GRN] + 1 If [GDN]=0, [PC]←QQ else [PC]←[PC + 2] Increment register and jump
	JCY or JLT	LABEL	05 QQ	2		If [C]=1, [PC]←QQ else [PC]←[PC + 2] Jump if Carry
	JEQ or JZE	LABEL	06 QQ	2		If [A]=0, [PC]←QQ else [PC]←[PC + 2] Jump if equal
	JGE or JNC	LABEL	01 QQ	2		If [C]=0, [PC]←QQ else [PC]←[PC + 2] Jump if greater or equal
	JGT	LABEL	03 QQ	2		If [C]=0 and [A]=1, [PC]←QQ else [PC]←[PC + 2] Jump if Accumulator is greater than Register
	JHC	LABEL	04 QQ	2		If [H]=1, [PC]←QQ else [PC]←[PC + 2] Jump if half carry
	JLE	LABEL	07 QQ	2		If [C]=1 and [A]=0, [PC]←QQ else [PC]←[PC + 2] Jump if less than or equal
	JNE or JNZ	LABEL	02 QQ	2		If [A]=1, [PC]←QQ else [PC]←[PC + 2] Jump if not equal

Table A-1. An Instruction Set Summary (Continued)

TYPE	MNEMONIC	OPERAND(S)	OBJECT CODE	MPU CYCLE (NOTE 1)	STATUSES C	D	A	H	OPERATIONS PERFORMED
REGISTER-REGISTER MOVE	CAP	N	01001XXX	1					[GPN]←[ACL] Copy Accumulator to Page
	CAR	N	10110XXX	1					[GDN]←[AC] Copy Accumulator to Register
	CPA	N	010010XX	1			X		[ACL]←[GPN] Copy Page to Accumulator (N=0, 1, 2 or 3 only)
	CRA	N	10111XXX	1			X		[AC]←[GDN] Copy RXEGISTER TO Accumulator
	XCH	N	01000XXX	1			X		[AC]←→[GDN] Exchange Register with Accumulator
REGISTER-REGISTER OPERATE	ADD	N	10000XXX	1(2)	X		X	X	[AC]←[AC] + [GDN] + [C] Decimal or binary addition specified by D
	AND	N	10010XXX	1	0		X	0	[AC]←[AC] ∧ [GDN] AND Register with Accumulator
	CMP	N	10101XXX	1	X		X		Compare Accumulator with register
	IOR	N	10011XXX	1	0		X	0	[AC]←[AC]V[GDN] OR Register with Accumulator
	SUB	N	10001XXX	1	X		X	X	[AC]←[AC]-[GDN]-[C] Subtract Register from Accumulator
	XOR	N	10100XXX	1(2)	0		X	0	[AC]←[AC]V[GDN] XOR Register with Accumulator
REGISTER OPERATE	CLA		F6	1			0		[AC]←00 Clear Accumulator
	CLB		F2		0		0		[AC]←00 Clear Accumulator and Carry
	CMA		F7	1			X		[AC]←[AC] Complement Accumulator
	DAC		F5	1	X		X	X	[AC]←[AC]-1 Decrement the Accumulator
	DCR	N	01111XXX	1					[GRN]←[GRN]-FFF Decrement Register

Table A-1. An Instruction Set Summary (Continued)

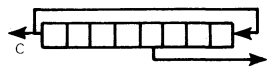
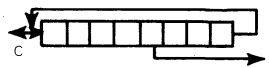
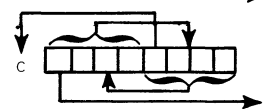
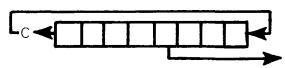
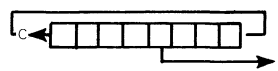
TYPE	MNEMONIC	OPERAND(S)	OBJECT CODE	MPU CYCLE (NOTE 1)	STATUSES C	D	A	H	OPERATIONS PERFORMED
REGISTER OPERATE	IAC		F4	1(2)	X		X	X	[AC]←[AC] + 1 Increment Accumulator
	INR	N	01110XXX	1(2)					[GRN]←[GRN] + 1 Increment Register
	RAL		F8	1	X			X	 Rotate Accumulator left
	RAR		F9	1(2)	X			X	 Rotate Accumulator right binary RAR decimal swaps HO and LO 4 bits  (decimal mode) Swap BCD digits, Bit 7 to H Bit 3 to C
	RLC		FA	1	X		X	X	 Rotate Accumulator left through Carry
	RRC		FB	1	X		X	X	 Rotate Accumulator right through Carry
STACK	JSR	LABEL	2P QQ	2 1					[STACK]←[PC] [PC]←PQQ Jump to subroutine
	RET		FE	1					[PC]←[STACK]

Table A-1. An Instruction Set Summary (Continued)

TYPE	MNEMONIC	OPERAND(S)	OBJECT CODE	MPU CYCLE (NOTE 1)	STATUSES	OPERATIONS PERFORMED
INTERRUPT	DSI		0E	1		[I]←0 Disable interrupts
	ENI		0F	1		[I]←1 Enable interrupts
	RET		FE	1		[PC]←Stack up
STATUS	CLB		F2	1	0 0	[AC]←00 Clear Accumulator and Carry
	CLC		F0	1	0	[C]←0 Clear Carry
	CMC		F3	1	X	[C]←[C] Complement Carry
	CSA		0C	1	X	[AC]←[SW] Copy status to Accumulator
	SEB		FD	1	0	[D]←0 Set binary mode
	SEC		F1	1	1	[C]←1 Set Carry status to 1
	SED		FC	1	1	[D]←1 Set decimal mode
	DLY	DATA	00 ZZ	2		Delay and skip I2
	NOP		FF	1		No operation

Table A-2 Instructions Affecting Status Flags

		A Flag		C Flag		H Flag	
Instr	**Mode**	**Arithmetic = 0**	**Arithmetic = 1**	**Carry = 0**	**Carry = 1**	**H. Carry = 0**	**H. Carry = 1**
ADD	B	$Accum_r = 0$	$Accum_r \neq 0$	**Sum** < FF	Sum > FF		
	D	$Accum_r = 0$	$Accum_r \neq 0$	Sum < 99	Sum > 99	Sum of lower digits $\leq$ 9	Sum of lower digits > 9
ADS	B	$Accum_r = 0$	$Accum_r \neq 0$	Sum < FF	Sum > FF		
	D	$Accum_r = 0$	$Accum_r \neq 0$	Sum < 99	Sum > 99	Sum of lower digits $\leq$ 9	Sum of lower digits > 9
AND		$Accum_r = 0$	$Accum_r \neq 0$	Always	Never	Always	Never
CLA		Always	Never				
CLB		Always	Never	Always	Never		
CLC				Always	Never		
CMA		$Accum_r = 0$	$Accum_r \neq 0$				
CMC				$Carry_l = 1$	$Carry_l = 0$		
CMP		Accum = Reg	Accum ≠ Reg	Accum > Reg	Accum < Reg	Always	Never
CPA		$Accum_r = 0$	$Accum_r \neq 0$				
CRA		$Accum_r = 0$	$Accum_r \neq 0$				
CSA		$Accum_r = 0$	$Accum_r \neq 0$				
DAC	B	$Accum_l = 1$	$Accum_l \neq 1$	$Accum_l \neq 0$	$Accum_l = 0$		
	D	$Accum_l = 1$	$Accum_l \neq 1$	$Accum_l \neq 0$	$Accum_l = 0$	$Accum_l \neq$ NO	$Accum_l =$ NO
IAC	B	$Accum_l =$ FF	$Accum_l \neq$ FF	$Accum_l \neq$ FF	$Accum_l =$ FF		
	D	$Accum_l =$ 99	$Accum_l \neq$ 99	$Accum_l \neq$ 99	$Accum_l =$ 99	$Accum_l \neq$ N9	$Accum_l =$ N9
INP		$Accum_r = 0$	$Accum_r \neq 0$				
IOR		$Accum_r = 0$	$Accum_r \neq 0$	Always	Never	Always	Never
LAI		$Accum_r = 0$	$Accum_r \neq 0$				
RAL				Bit $7_l = 0$	Bit $7_l = 1$	Bit $3_l = 0$	Bit $3_l = 1$

Table A-2 Instructions Affecting Status Flags (Continued)

Instr	Mode	A Flag		C Flag		H Flag	
		Arithmetic = 0	Arithmetic = 1	Carry = 0	Carry = 1	H. Carry = 0	H. Carry = 1
RAR	B			Bit $O_i = 0$	Bit $O_i = 1$	Bit $4_i = 0$	Bit $4_i = 1$
	D			Bit $3_i = 0$	Bit $3_i = 1$	Bit $7_i = 0$	Bit $7_i = 1$
RDS		$Accum_r = 0$	$Accum_r \neq 0$				
RLC		$Accum_r = 0$	$Accum_r \neq 0$	Bit $7_i = 0$	Bit $7_i = 1$	Bit $3_i = 0$	Bit $3_i = 1$
RRC		$Accum_r = 0$	$Accum_r \neq 0$	Bit $0_i = 0$	Bit $0_i = 1$	Bit $4_i = 0$	Bit $4_i = 1$
SEC				Never	Always		
SUB	B	$Accum_r = 0$	$Accum_r \neq 0$	Difference ≥ 0	Difference < 0		
	D	$Accum_r = 0$	$Accum_r \neq 0$	Difference ≥ 0	Difference < 0	Difference of lower digits ≥ 0	Difference of lower digits < 0
SUS	B	$Accum_r = 0$	$Accum_r \neq 0$	Difference ≥ 0	Difference < 0		
	D	$Accum_r = 0$	$Accum_r \neq 0$	Difference ≥ 0	Difference < 0	Difference of lower digits ≥ 0	Difference of lower digits < 0
XCH		$Accum_r = 0$	$Accum_r \neq 0$				
XOR		$Accum_r = 0$	$Accum_r \neq 0$	Always	Never	Always	Never

Notes:

B = Binary
D = Decimal
N = Any Digit
i = Initial Value
r = Resulting Value

All digits shown are hexadecimal.

Half carry is set to zero at the beginning of any instruction which affects it.

The following instructions do not affect status flags: SED, SEB, RET, NOP, JIN, LRN, ADR, CAR, OUT, WRS, LRI, JUN, JSR, JCN, INR, IRJ, DCR, DRJ, SRN, CAP, ENI, DSI.

Bit references are to the accumulator.

Table A-3 An Object Code Map

4 MSB \ 4 LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	DLY*	J	CONDITIONAL JUMPS*						CPA				CSA	LAI*	DSI	ENI
1	JUN* UNCONDITIONALLY															
2	JSR* TO SUBROUTINE															
3	IRJ* INCREMENT								DRJ* DECREMENT							
4	XCH EXCHANGE								CAP ACC TO PAGE							
5	INP INPUT								OUT OUT							
6	LRI* LOAD IMMEDIATE								JIN INDIRECT							
7	INR INCREMENT								DCR DECREMENT							
8	ADD ADD								SUB SUBTRACT							
9	AND AND								10R INC. OR							
A	X0R EX. OR								CMP COMPARE							
B	CAR ACC → REG.								CRA REG → ACC							
C	ADS ADD SCRATCH								SUS SUB SCRATCH							
D	RDS READ SCRATCH								WRS WRITE SCRATCH							
E	LRN LOAD REG.								SRN STORE REG.							
F	CLC	SEC	CLB	CMC	IAC	DAC	CLA	CMA	RAL	RAR	RLC	RRC	SED	SEB	RET	NOP

*Two Byte Instructions

Appendix B

HEXADECIMAL-DECIMAL INTEGER CONVERSION

Appendix B

HEXADECIMAL-DECIMAL INTEGER CONVERSION

The table below provides for direct conversions between hexadecimal integers in the range 0–FFF and decimal integers in the range 0–4095. For conversion of larger integers, the table values may be added to the following figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

Hexadecimal fractions may be converted to decimal fractions as follows:

1. Express the hexadecimal fraction as an integer times 16^{-n}, where n is the number of significant hexadecimal places to the right of the hexadecimal point.

$$0.\ CA9BF3_{16} = CA9\ BF3_{16} \times 16^{-6}$$

2. Find the decimal equivalent of the hexadecimal integer

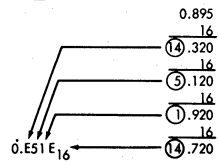
$$CA9\ BF3_{16} = 13\ 278\ 195_{10}$$

3. Multiply the decimal equivalent by 16^{-n}

$$\begin{array}{l} 13\ 278\ 195 \\ \times\ 596\ 046\ 448 \times 10^{-16} \\ \hline 0.791\ 442\ 096_{10} \end{array}$$

Decimal fractions may be converted to hexadecimal fractions by successively multiplying the decimal fraction by 16_{10}. After each multiplication, the integer portion is removed to form a hexadecimal fraction by building to the right of the hexadecimal point. However, since decimal arithmetic is used in this conversion, the integer portion of each product must be converted to hexadecimal numbers.

Example: Convert 0.895_{10} to its hexadecimal equivalent



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
200	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
400	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
510	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
590	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	4761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	3851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2866	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB0	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

TABLE OF POWERS OF SIXTEEN$_{10}$

16^n	n	16^{-n}				
1	0	0.10000	00000	00000	00000	x 10
16	1	0.62500	00000	00000	00000	x 10^{-1}
256	2	0.39062	50000	00000	00000	x 10^{-2}
4 096	3	0.24414	06250	00000	00000	x 10^{-3}
65 536	4	0.15258	78906	25000	00000	x 10^{-4}
1 048 576	5	0.95367	43164	06250	00000	x 10^{-6}
16 777 216	6	0.59604	64477	53906	25000	x 10^{-7}
268 435 456	7	0.37252	90298	46191	40625	x 10^{-8}
4 294 967 296	8	0.23283	06436	53869	62891	x 10^{-9}
68 719 476 736	9	0.14551	91522	83668	51807	x 10^{-10}
1 099 511 627 776	10	0.90949	47017	72928	23792	x 10^{-12}
17 592 186 044 416	11	0.56843	41886	08080	14870	x 10^{-13}
281 474 976 710 656	12	0.35527	13678	80050	09294	x 10^{-14}
4 503 599 627 370 496	13	0.22204	46049	25031	30808	x 10^{-15}
72 057 594 037 927 936	14	0.13877	78780	78144	56755	x 10^{-16}
1 152 921 504 606 846 976	15	0.86736	17379	88403	54721	x 10^{-18}

10^n	n	10^{-n}				
1	0	1.0000	0000	0000	0000	
A	1	0.1999	9999	9999	999A	
64	2	0.28F5	C28F	5C28	F5C3	x 16^{-1}
3E8	3	0.4189	374B	C6A7	EF9E	x 16^{-2}
2710	4	0.68DB	8BAC	710C	B296	x 16^{-3}
1 86A0	5	0.A7C5	AC47	1B47	8423	x 16^{-4}
F 4240	6	0.10C6	F7A0	B5ED	8D37	x 16^{-4}
98 9680	7	0.1AD7	F29A	BCAF	4858	x 16^{-5}
5F5 E100	8	0.2AF3	1DC4	6118	73BF	x 16^{-6}
3B9A CA00	9	0.44B8	2FA0	9B5A	52CC	x 16^{-7}
2 540B E400	10	0.6DF3	7F67	SEF6	EADF	x 16^{-8}
17 4876 E800	11	0.AFEB	FF0B	CB24	AAFF	x 16^{-9}
E8 D4A5 1000	12	0.1197	9981	2DEA	1119	x 16^{-9}
918 4E72 A000	13	0.1C25	C268	4976	81C2	x 16^{-10}
5AF3 107A 4000	14	0.2D09	370D	4257	3604	x 16^{-11}
3 8D7E A4C6 8000	15	0.480E	BE7B	9D58	566D	x 16^{-12}
23 8652 6FC1 0000	16	0.734A	CA5F	6226	F0AE	x 16^{-13}
163 4578 5D8A 0000	17	0.B877	AA32	36A4	B449	x 16^{-14}
DE0 B6B3 A764 0000	18	0.1272	5DD1	D243	ABA1	x 16^{-14}
8AC7 2304 89E8 0000	19	0.1D83	C94F	B6D2	AC35	x 16^{-15}

Appendix C

STANDARD CHARACTER CODES

Appendix C

STANDARD CHARACTER CODES

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
A		
B		
C		
D		
E		
F		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
1A		
1B		
1C		
1D		
1E		
1F		
20	blank	
21	!	
22	"	
23	#	
24	$	
25	%	
26	&	
27	'	
28	(	
29	)	
2A	*	
2B	+	
2C	,	
2D	–	
2E	.	
2F	/	
30	0	

Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
31	1	
32	2	
33	3	
34	4	
35	5	
36	6	
37	7	
38	8	
39	9	
3A	:	
3B	;	
3C	<	
3D	=	
3E	>	
3F	?	
40	@	blank
41	A	
42	B	
43	C	
44	D	
45	E	
46	F	
47	G	
48	H	
49	I	
4A	J	]
4B	K	.
4C	L	<
4D	M	(
4E	N	+
4F	O	!
50	P	&
51	Q	
52	R	
53	S	
54	T	
55	U	
56	V	
57	W	
58	X	
59	Y	
5A	Z	[
5B	[	$
5C	\	*
5D	]	)
5E		;
5F		^
60		
61	a	

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)
62	b	
63	c	
64	d	
65	e	
66	f	
67	g	
68	h	
69	i	
6A	j	
6B	k	,
6C	l	%
6D	m	-
6E	n	)
6F	o	?
70	p	
71	q	
72	r	
73	s	
74	t	
75	u	
76	v	
77	w	
78	x	
79	y	
7A	z	:
7B		#
7C		@
7D		'
7E		=
7F		"
80		
81		a
82		b
83		c
84		d
85		e
86		f
87		g
88		h
89		i
8A		
8B		
8C		
8D		
8E		
8F		
90		
91		j
92		k
93		l
94		m
95		n
96		o

Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
97		p
98		q
99		r
9A		
9B		
9C		
9D		
9E		
9F		
A0		
A1		
A2		s
A3		t
A4		u
A5		v
A6		w
A7		x
A8		y
A9		z
AA		
AB		
AC		
AD		
AE		
AF		
B0		
B1		
B2		
B3		
B4		
B5		
B6		
B7		
B8		
B9		
BA		
BB		
BC		
BD		
BE		
BF		
C0		
C1		A
C2		B
C3		C
C4		D
C5		E
C6		F
C7		G
C8		H
C9		I
CA		
CB		

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)
CC		
CD		
CE		
CF		
D0		
D1		J
D2		K
D3		L
D4		M
D5		N
D6		O
D7		P
D8		Q
D9		R
DA		
DB		
DC		
DD		
DE		
DF		
E0		
E1		
E2		S
E3		T
E4		U
E5		V

Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
E6		W
E7		X
E8		Y
E9		Z
EA		
EB		
EC		
ED		
EE		
EF		
F0		0
F1		1
F2		2
F3		3
F4		4
F5		5
F6		6
F7		7
F8		8
F9		9
FA		
FB		
FC		
FD		
FE		
FF		

No attempt has been made in this handbook to teach basic boolean algebra, binary arithmetic, logic design or software programming. Rather, the objective is to present the unique features of Electronic Array's EA9002 MPU and how it may be applied. For basic understanding of digital logic implemented with TTL the reader is directed to "Logic Design With Integrated Circuits" by W.E. Wickes published by Wiley & Sons, N.Y., N.Y. For better understanding of logic techniques and microprocessor architecture the reader is directed to "An Introduction to Microcomputers" written and published by Adam Osborne and Assoc. Inc., 2950 Seventh Street, Berkeley, CA. 94710.

Electronic Arrays Sales Offices

EA SALES OFFICES

U.S. SALES REPRESENTATIVES

INTERNATIONAL SALES REPRESENTATIVES AND DISTRIBUTORS

Electronic Arrays Sales Offices

EA SALES OFFICES

Vice President of Marketing

Ed Turney
Tel: (415) 964-4321

Headquarters Sales Manager

Ed Meagher
Tel: (415) 964-4321

Headquarters

550 E. Middlefield Road
Mountain View, CA 94043
Tel: (415) 964-4321
TWX: 910-379-6985
Toll Free WATS:
(800) 277-9962

Eastern Area

Jim Carr
Regional Sales Manager
2 Coleman Avenue
Cherry Hill, NJ 08034
Tel: (609) 795-5066
TWX: 710-896-1488

Mid-America Area

Shel Schumaker
Regional Sales Manager
11960 Westline Industrial Drive
Suite 322
St. Louis, MO 63141
Tel: (314) 878-6446
TWX: 910-764-0872

Western Area

Dick Goodman
Regional Sales Manager
9932 Voyager Circle
Huntington Beach, CA 92646
Tel: (714) 968-3775

Ed Meagher
Regional Sales Manager
San Francisco Bay Area
550 E. Middlefield Road
Mountain View, CA 94043
Tel: (415) 964-4321
TWX: 910-379-6985

Europe

Chris P. Carli
Sales Manager - Europe
Hobbema, Str. 26
Amsterdam/Zuid, Netherlands
Tel: (020)712560
TLX: 12674

Far East

Jim Shales
Sales Manager - Far East
550 E. Middlefield Road
Mountain View, CA 94043
Tel: (415) 964-4321
TWX: 910-379-6985

U.S. SALES REPRESENTATIVES

Arizona

Chaparral-Dorton
P.O. Box 16127
364 E. Virginia Avenue
Phoenix, AZ 85011
Tel: (602) 263-0414
TWX: 910-951-4225

Southern California

Bestronics
1728 S. La Cienega Boulevard
Los Angeles, CA 90035
Tel: (213) 870-9191
TWX: 910-340-6369

Bestronics
8369 Vickers Street
San Diego, CA 92111
Tel: (714) 278-2150
TWX: 910-335-1267

Northern California

Caltron/Pyle,Inc.
415 Clyde Avenue, Suite D
Mountain View, CA 94043
Tel: (415) 964-3244

Colorado

The Thorson Company
5290 Yale Circle
Denver, CO 80222
Tel: (303) 759-0809
TWX: 910-931-0429

Connecticut

Com-Sale, Inc.
633 Williams Road
Wallingford, CT 06492
Tel: (203) 269-7964

Crane & Egert Corp.
Greenwich, CT 06830
Tel: (203) 622-9191

Florida

Electrocomp Associates, Inc.
4101 N. Andrews Ave., Suite 101
Ft. Lauderdale, FL 33309
Tel: (305) 564-8571
TWX: 510-955-9735

Illinois

Gassner & Clark Company
1127 S. Mannheim Road
Westchester, IL 60153
Tel: (312) 345-4245
TWX: 910-234-2082

Indiana

Ihrig Associates, Inc.
6410 Woodwind Drive
Indianapolis, IN 46217
Tel: (317) 783-7630

Iowa

Engineering Services Co.
P.O. Box 2145
Cedar Rapids, IA 52406
Tel: (319) 362-0503

Kansas

Engineering Services Co.
7830 State Line Road
Prairie Village, KS 66208
Tel: (913) 649-4000

Massachusetts

Com-Sale, Inc.
235 Bear Hill Road
Waltham, MA 02154
Tel: (617) 890-0011

Michigan

A.P. Associates
39950 Grand River
Novi, MI 48050
Tel: (313) 476-2300
TWX: 810-223-6031

Missouri

Engineering Services Co.
8420 Delmar Boulevard
St. Louis, MO 63124
Tel: (314) 997-1515

New Hampshire

Com-Sale, Inc.
c/o Yankee Electronics
Grenier Industrial Village
Manchester, NH 03053
Tel: (603) 668-8500

New Mexico

The Thorson Company
2201 San Pedro Drive, NE
Building 2, Suite 107
Albuquerque, NM 87110
Tel: (505) 265-5655
TWX: 910-989-1174

New York

Crane & Egert Corp.
146 Meacham Avenue
Elmont, NY 11003
Tel: (516) 488-2100
TWX: 510-223-0417

Ohio

K.W. Electronic Sales, Inc.
8312 North Main Street
Dayton, OH 45415
Tel: (513) 890-2150
TWX: 810-450-2521

K.W. Electronic Sales, Inc.
3439 West Brainard Road
Cleveland, OH 44122
Tel: (216) 831-8292
TWX: 810-427-2902

Oregon

N.R. Schultz Company
P.O. Box 156
4195 S.W. Cedar Hills Boulevard
Beaverton, OR 97005
Tel: (503) 643-1644
TWX: 910-467-8707

Pennsylvania

K.W. Electronic Sales, Inc.
4024 William Flynn Highway
Route 8
Allison Park, PA 15101
Tel: (412) 487-4300
TWX: 710-795-3636

Harry Nash Associates
P.O. Box 188
1009 York Road
Willow Grove, PA 19090
Tel: (215) 657-2213
TWX: 510-665-6273

Texas

J. Clay Co.
2619 Electronic Lane, Suite 302
Dallas, TX 75220
Tel: (214) 350-1281
TWX: 910-861-9160

Virginia

Boyle Associates
12001 Whip Road
Reston, VA 22091
Tel: (703) 620-9558

Washington

N.R. Schultz Co.
P.O. Box 159
855-106th NE
Bellevue, WA 98009
Tel: (206) 454-0300
TWX: 910-443-2329

INTERNATIONAL SALES REPRESENTATIVES AND DISTRIBUTORS

Australia

A.J. Ferguson
44 Prospect Road
Prospect 5082, S. Australia
Tel: 269-1244
TLX: 82635

Belgium

Betea Sprl.
15 Av. Geo Bernier
Brussels 5, Belgium
Tel: 02-649-99-00
TLX: 23188

Electronic Arrays
Hobbema, Str.26
Amsterdam/Zuid, Netherlands
Tel: (020) 712560
TLX: 12674

Finland

S.W. Instruments
Karstulantie 4B
Helsinki 55, Finland 00550
Tel: 738265/713575
TLX: 122411

France

Technology Resources S.A.
27-29 Rue des Poissonniers
92200 Neuilly-Sur-Seine
Paris, France
Tel: 01 747 4717
TLX: 610657

Germany

Electronic Arrays, GmbH
8000 Munchen 70
Hofmannstrasse 20
West Germany
Tel: (089) 7853168
TLX: 5213066

Holland

Famatra Benelux
Ginnenkenweg 128
Breda, Holland
Tel: 01600-33457
TLX: 54521

Electronic Arrays
Hobbema, Str. 26
Amsterdam/Zuid, Netherlands
Tel: (020) 712560
TLX: 12674

Hong Kong*

Caduceus Ltd.
Rm. 1204 Hang Lung Center
Patterson Street
Causeway Bay, Hong Kong
Tel: 5-770504,505
TLX: 83640

India

Hinditron Services Pvt. Ltd.
69/A.L. Jagmohandas Marg
Bombay, 400 006, India
Tel: 36-5-3-44
TLX: 0112594 or 0112326
Cable: TEKHIND

Israel

Telsys Ltd.
54, Jabotinsky Road
Ramat-Gan 52-462, Israel
Tel: 739865, 722362
TLX: 032392

Italy

Memos Italiana S.R.L.
Via Boccaccio 2
20123 Milano, Italy
Tel: 02-871353 or 867589
TLX: 34343

Japan

Rikei Corporation
1-18-14 Nishi-Shimbashi
Minato-ku
Tokyo 105, Japan
Tel: 03-591-5241
TLX: 24208

Korea

Caduceus Ltd.
Rm. 301 Sung Won Bldg.
15, 2-GA Hyehyun-Dong
Chung-ku, Seoul, Korea
Tel: 28-6108

Norway

Kjell Bakke
NYGT 48 P.O. Box 143
Strommen 2011 Norway
Tel: 02-711872
TLX: 19407

South and Central America

Intectra
1950 Colony Street
Mountain View, CA 94043
Tel: (415) 967-8818
Cable: INTECTRA

Spain

Belport Electronica
Canillas 22
Madrid 2, Spain
Tel: 01 262 88 37/8
TLX: 22048

Sweden

Svensk Tele Industri
Box 502
162 05 Vallingby, Sweden
Tel: 08/89 04 35
TLX: 13033

Switzerland

Dimos AG
Badenerstrasse 701
8048 Zurich, Switzerland
Tel: 01-626-140
TLX: 52028

Taiwan*

Caduceus Ltd.
Lincoln Center
Flat "N" 7th Floor
121 Szu-Wei Road
Taipei, Taiwan
Tel: 721-0431,0483
TLX: 21071

United Kingdom

Analog Devices, Ltd.
Central Ave. Trading Estate
W. Molesey
Surrey, England
Tel: 01-941-0466
TLX: 929962

Yugoslavia

Ellyptic AG
Ptujska 14
62000 Maribor, Yugoslavia
Tel: 062 31041

**U.S. Contact: Centauri Trading Co.,* 1245 S. Winchester, Suite 101, San Jose, CA 95128, Tel: (408) 246-4071, TLX: 352020

Electronic Arrays, Inc., 550 east middlefield road, mountain view, california 94043
telephone (415) 964-4321 • TWX 910-379-6985