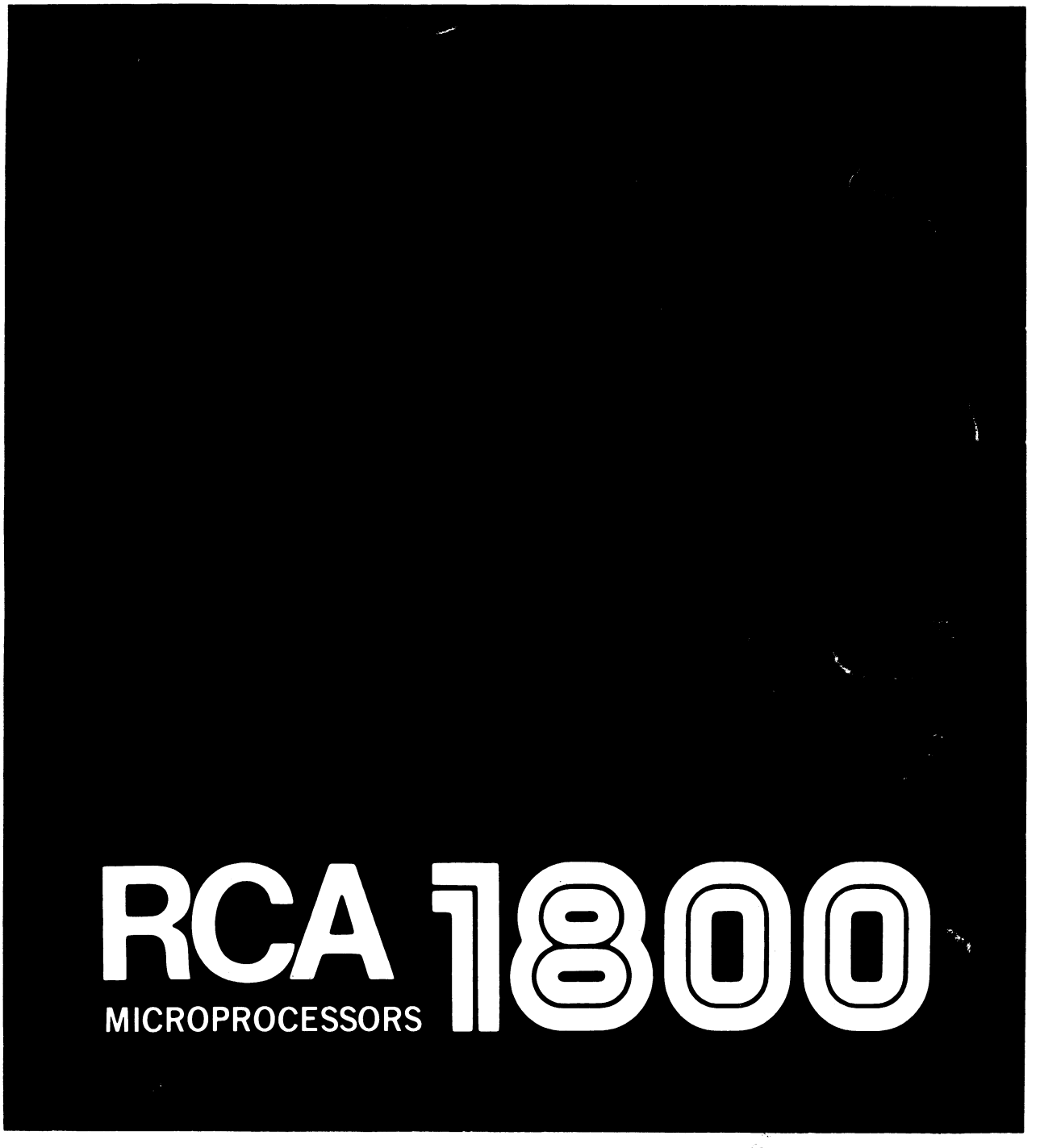


User Manual for the CDP1802 COSMAC Microprocessor

MPM-201B Suggested Price $5.00

User Manual
for the CDP1802
COSMAC Microprocessor

RCA Solid State

Buenos Aires • Hamburg • Liege • Madrid • Mexico City • Milan
Montreal • Paris • Sao Paulo • Somerville NJ • Stockholm
Sunbury-on-Thames • Taipei • Tehran • Tokyo

Printed in USA/11-77

Foreword

The **RCA CDP1802 COSMAC Microprocessor** is a one-chip CMOS 8-bit register-oriented central processing unit. It is suitable for use in a wide range of stored-program computer systems and products. These systems may be either special or general purpose in nature.

This **User Manual** provides a detailed guide to the COSMAC Microprocessor. It is written for electronics engineers and assumes only a limited familiarity with computers and computer programming. It describes the microprocessor architecture and provides a set of simple, easy-to-use programming instructions. Examples are given to illustrate the operation and usage of each instruction.

For systems designers, this Manual illustrates practical methods of adding external memory and control circuits. Because the processor is capable of supporting input/output (I/O) devices in polled, interrupt-driven, and direct-memory-access modes, detailed examples are provided for the use of the I/O instructions and the use of the I/O interface lines. The latter include direct-memory-access and interrupt inputs, external flag inputs, command lines, processor state indicators, and external timing pulses.

This Manual also discusses various programming techniques and gives examples. The material covers, in addition to basic guidelines, more advanced topics such as interrupt response and subroutine linkage and nesting.

This basic Manual is intended to help design engineers understand the COSMAC Microprocessor and to aid them in developing simpler and more powerful products utilizing the wide range of microprocessor capabilities. Users requiring information on available hardware and software support systems for the CDP1802 Microprocessor should also refer to the following publications:

MPM-202 **Timesharing Manual for the RCA CDP1802 COSMAC Microprocessor**

MPM-203 **Evaluation Kit Manual for the RCA CDP1802 COSMAC Microprocessor**

MPM-206 Binary Arithmetic Subroutines for RCA COSMAC Microprocessors

MPM-216 Operator Manual for RCA COSMAC Development System II CDP18S005

MPM-217 RCA COSMAC Floppy Disk System II CDP18S805 Instruction Manual

MPM-212 Instruction Manual for RCA COSMAC Microterminal CDP18S021

Table of Contents

Introduction

General

The RCA COSMAC Microprocessor architecture has been developed for a wide variety of applications. These applications range from replacement of SSI and MSI integrated circuits to new applications requiring the full flexibility of a computer-based approach.

The RCA-CDP1802 is a byte-oriented central processing unit (CPU) employing the COSMAC architecture and utilizing complementary-symmetry MOS technology (CMOS).

CDP1802 operations are specified by sequences of instruction codes stored in a memory. Sequences of instructions, called **programs**, determine the specific behavior or function of a COSMAC-based system. System functions are easily changed by modifying the program stored in memory. This ability to change function without extensive hardware modification is the basic advantage of a stored-program computer. Reduced cost results from using identical LSI components (memory and microprocessor) in a variety of different systems or products.

The CDP1802 Microprocessor includes all of the circuits required for fetching, interpreting, and executing instructions which have been stored in standard types of memories. Extensive **input/output (I/O)** control features are also provided to facilitate system design.

Although Microprocessor cost is only a small part of total system or product cost (memory, input, output, power-supply, system-control, and design costs are also major considerations), a unique set of COSMAC features combine to minimize the total system cost. For example, the low-power, single-voltage CMOS circuitry minimizes power-supply and packaging costs. A single-phase clock drives the system and an optional on-chip oscillator circuit works with an external crystal to provide this clock signal. High noise immunity and wide temperature tolerance facilitate use in hostile environments. In addition, compatibility with standard, high-volume memories assures minimum memory cost and maximum system flexibility for both current and future applications. Program storage requirements are reduced by means of an efficient one-byte operation code.

The 40-pin system interface of the CDP1802 is designed to minimize external I/O and memory control circuitry. Four directly testable input flags, an output flip-flop, an internal direct-memory-access (DMA) mode, flexible I/O instructions, program interrupt, program load mode, and static circuitry are other features explicitly aimed at total system cost reduction. The CDP1802 does not require an external bootstrap ROM.

Microprocessor programming and system design are facilitated by the availability of a variety of support programs and support hardware. Extensive support software and support hardware are available for use in developing COSMAC systems. Machine-language programming is sometimes indicated when only a few short programs need to be developed. A series of efficient, easy-to-learn instructions are provided for the CDP1802 which are simple to use in machine-language programs.

Specific Features

The advanced features and operating characteristics of the RCA Microprocessor CDP1802 include:

- static CMOS circuitry, no minimum clock frequency
- full military temperature range (−55 to +125°C)
- high noise immunity, wide operating-voltage range
- TTL compatibility
- single-phase clock; optional, on-chip, crystal-controlled oscillator
- simple control of reset, start, and pause

- 8-bit parallel organization with bidirectional data bus
- built-in program-load facility
- any combination of standard RAM's and ROM's via common interface
- direct memory addressing up to 65,536 bytes
- flexible programmed I/O mode
- program interrupt mode
- on-chip DMA facility
- four I/O flag inputs directly testable by branch instruction
- programmable output port
- one-to-three byte instruction format with two machine cycles for each instruction*
- 91 easy-to-use instructions
- 16 X 16 matrix of registers for use as multiple program counters, data pointers, or data registers

System Organization

Fig. 1 illustrates a typical computer system incorporating the RCA Microprocessor CDP1802. Operations that can be performed include:

a) control of input/output (I/O) devices,

b) transfer of data or control information between I/O and memory (M),

c) movement of data bytes between different memory locations,

d) interpretation or modification of bytes stored in memory.

In such a system, the CDP1802 can, for example, control the entry of binary-coded decimal numbers from an input keyboard and store them in predetermined memory locations. It can then perform specified arithmetic operations using the stored numbers and transfer the results to an output display or printing device.

System input devices may include switches, paper-tape/card readers, magnetic-tape/disc devices, relays, modems, analog-to-digital converters, photodetectors, and other computers. Output devices may include lights, relays, CRT/LED/liquid-crystal devices, digital-to-analog converters, modems, printers, and other computers.

Memory can comprise any combination of RAM and ROM up to a maximum of 65,536 bytes. ROM (Read-Only Memory) is used for permanent storage of programs, tables, and other types of fixed data. RAM (Random-Access Memory) is required for general-purpose computer systems which require frequent program changes. RAM is also required for temporary storage of variable data. The type of memory and required storage capacity is determined by the specific application of the system.

Bytes are transferred between I/O devices, memory, and COSMAC by means of a common, bidirectional eight-bit **data bus**.

Fifteen **I/O control signal lines** are provided. Systems can use some or all of these signals depending on required I/O sophistication. A three-bit **N code** is generated by the input/output instruction. It can be used to specify whether an I/O byte on the bus is meant to represent data, an I/O device selection code, an I/O status code, an I/O control code, etc. Use of the N code to specify an I/O device directly permits simple, inexpensive control of a small number of I/O devices or

*Except long-branch and long-skip instructions which require three machine cycles.

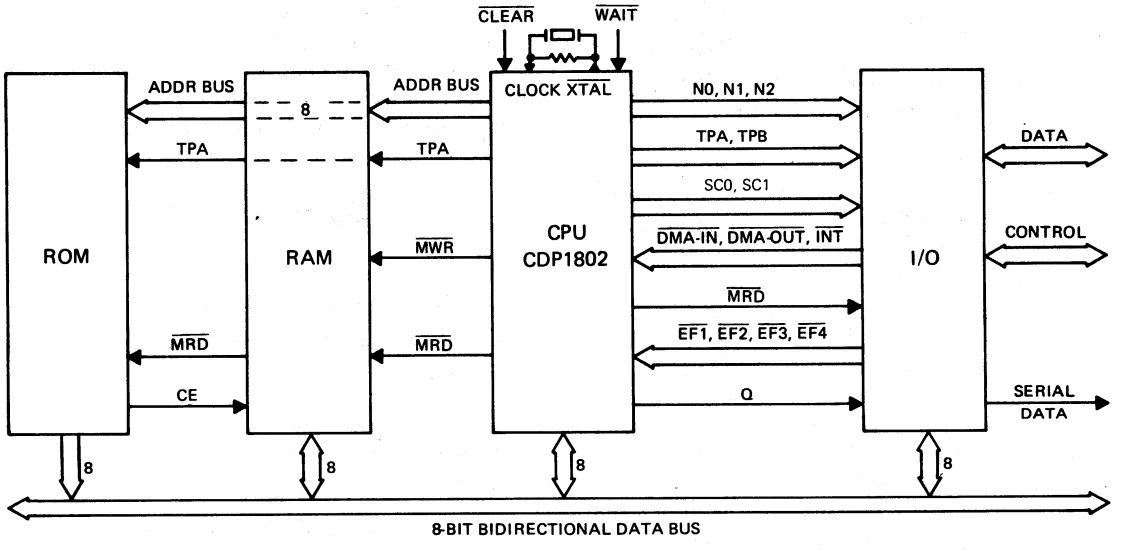


Fig. 1 – Block diagram of typical computer system using the RCA COSMAC Microprocessor CDP1802.

modes. Use of the N code to specify the meaning of the word on the data bus facilitates systems incorporating a large number of I/O devices or modes.

Four **I/O flag inputs** are provided. I/O devices can control these inputs at any time to signal the CDP1802 that a byte transfer is required, that an error condition has occurred, etc. These flags can also be used as binary input lines if desired. They can be tested by CDP1802 instructions to determine whether or not they are active. Use of the flag inputs must be coordinated with programs that test them.

An **output line** (Q) is also available which provides a level output whose value is controlled by COSMAC instructions. This Q line, under program control, can activate or signal I/O devices. It can also be used in connection with one of the flag inputs to form a serial I/O interface.

A program **interrupt line** can be activated at any time by I/O circuits to obtain an immediate microprocessor response. The interrupt causes the CDP1802 to suspend its current program sequence and execute a predetermined sequence of operations designed to respond to the interrupt condition. After servicing the interrupt, the CDP1802 resumes execution of the interrupted program. The CDP1802 can be made to ignore the interrupt line by resetting its **interrupt-enable flip-flop (IE)**.

Two additional I/O lines are provided for special types of byte transfer between memory and I/O devices. These lines are called **direct-memory-access (DMA) lines**. Activating the DMA-in line causes an input byte to be immediately stored in a memory location without intervention by the program being executed. The DMA-out line causes a byte to be immediately transferred from memory to the requesting output circuits. A built-in memory pointer register is used to indicate the memory location for the DMA cycles. The program initially sets this pointer to a beginning memory location. Each DMA byte transfer automatically increments the pointer to the next higher memory location. Repeated activation of a DMA line can cause the transfer of any number of consecutive bytes to and from memory independent of concurrent program execution.

I/O device circuits can cause data transfer by activating a flag line, the interrupt line, or a DMA line. The flag lines must be sampled by the program to determine when they become active and are used for relatively slow changing signals. Activating the interrupt line causes an immediate COSMAC response regardless of the program currently in progress, suspending operation of that program and allowing real-time response. Use of DMA provides the quickest response with least disturbance of the program.

A two-bit **state code** and two **timing lines** are provided for use by I/O device circuits. These four signals permit synchronization of I/O circuits with internal CDP1802 operating cycles. The state code indicates whether the CDP1802 is responding to a DMA request, responding to an interrupt request, fetching an instruction, or executing an instruction. The timing signals TPA and TPB are used by the memory and I/O systems to signal a new processor state code, to latch memory address bits, to take memory date from the bus, and to set and reset I/O controller flip-flops.

Bytes are transmitted to and from memory by means of the common data bus. The CDP1802 provides two lines to control memory read/write cycles. During a memory write cycle, the byte to be written appears on the data bus, either from the CPU or from an I/O device, and a **memory write pulse** is generated by the CPU at the appropriate time. During a memory-read cycle, a **memory read level** output is generated which is used by the system to gate the memory output byte onto the common data bus for use by the CPU or by an I/O device.

The CDP1802 provides eight **memory address lines**. These eight lines supply 16-bit memory addresses in the form of two successive 8-bit bytes. The more significant (high-order) address byte appears on the eight address lines first, followed by the less significant (low-order) address byte. The number of high-order bits required to select a unique memory byte location depends on the size of the memory. For example, a 4096-byte memory would require a 12-bit address. This 12-bit address is obtained by combining 4 bits from the high-order address byte with the 8 bits from the low-order address byte. One of the two CDP1802 timing pulses may be used to strobe the required high-order bits into an address latch (register) when they appear on the eight address lines. Latch circuits are not required at all if address registers are incorporated on the memory chips, as in the RCA 1800-series ROM's. An internal CPU register holds the eight low-order address bits on the address lines for the remainder of the memory cycle.

Four additional lines complete the microprocessor system interface. A single-phase **clock input** determines operating speed. The external clock may be stopped and started to synchronize the CDP1802 operation with system circuits if desired. Construction of the clock circuit is simplified by use of **$\overline{\text{XTAL}}$ input**. A crystal is connected between $\overline{\text{XTAL}}$ and clock input; no active components are needed. The **clear input** line initializes the microprocessor, and its release starts instruction execution. The **wait** line suspends the CPU operation cleanly. Simultaneous assertion of **clear** and **wait** puts the CPU in a program load mode.

COSMAC Architecture and Notation

Fig. 2 illustrates the internal structure of the COSMAC Microprocessor CDP1802. This simple, unique architecture results in a number of system advantages. The COSMAC architecture is based on a register array comprising sixteen general-purpose 16-bit **scratch-pad registers**. Each scratch-pad register, **R**, is designated by a 4-bit binary code. **Hexadecimal (hex) notation** will be used here to refer to 4-bit binary codes. The 16 hexadecimal digits (0,1,2,...E,F) and their binary equivalents

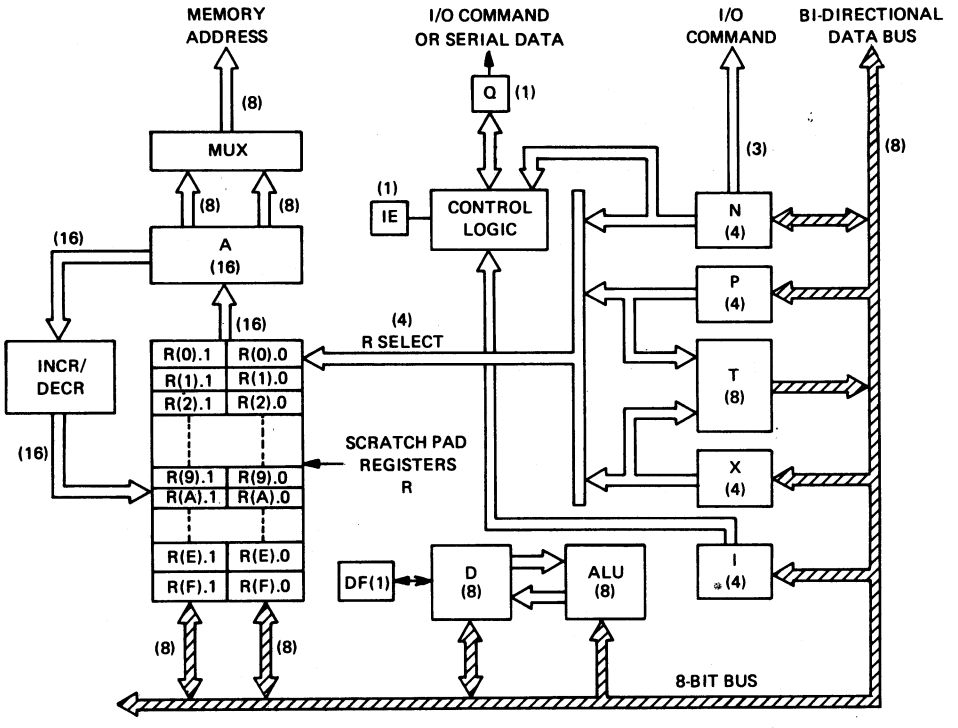


Fig. 2 — Internal structure of the CDP 1802 Microprocessor.

(0000,0001,0010,...,1110,1111) are listed in Appendix A.

Using hex notation, R(3) refers to the 16-bit scratch-pad register designated or selected by the binary code 0011. R(3).0 refers to the **low-order** (less significant) eight bits or byte of R(3). R(3).1 refers to the **high-order** (more significant) byte of R(3).

Three 4-bit registers labeled **N**, **P**, and **X** hold the 4-bit binary codes (hex digits) that are used to select individual 16-bit scratch-pad registers. The 16 bits contained in a selected scratch-pad can be used in several ways. Considered as two bytes, they may be sequentially placed on the eight external memory address lines for memory read/write operations. Either byte can also be gated to the 8-bit data bus for subsequent transfer to the **D register**. The 16-bit value in the A register can also be incremented or decremented by 1 and returned to the selected scratch-pad register to permit a scratch-pad register to be used as a counter.

The notation R(X), R(N), or R(P) is used to refer to a scratch-pad register selected by the 4-bit code in X, N, or P, respectively. Fig. 3 illustrates the transfer of a scratch-pad register byte, designated by N, to D. The

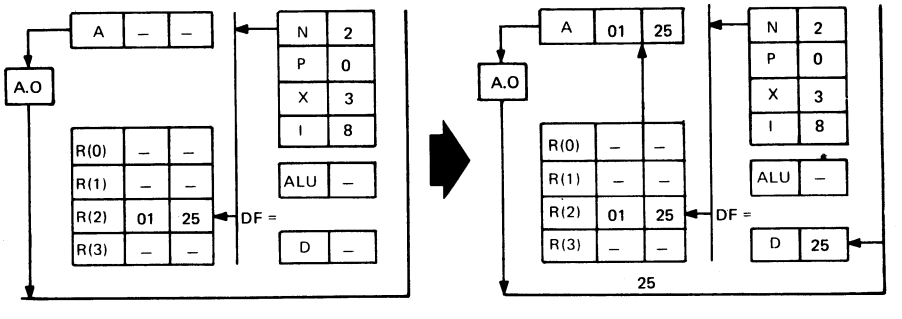


Fig. 3 — Use of N designator to transfer data from scratch-pad register R(2) to the D register.

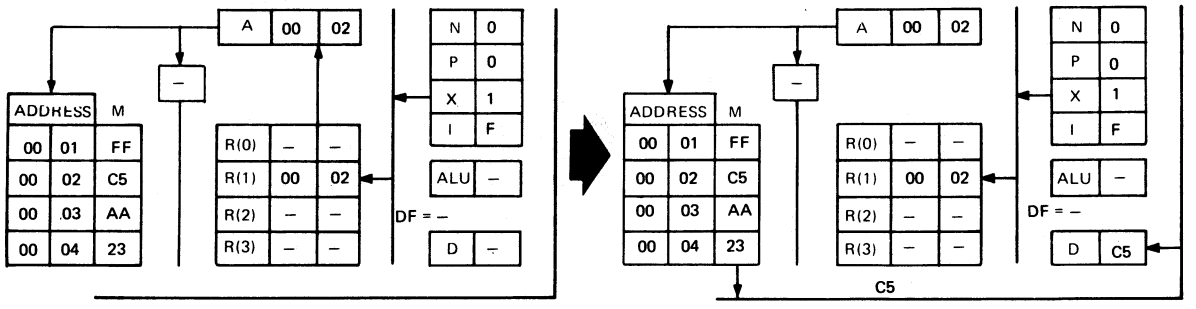


Fig. 4 – Transfer of data from memory to the D register.

left half of Fig. 3 illustrates the initial contents of various registers (hex notation). The operation performed can be written as

$$R(N).0 \rightarrow D$$

This expression indicates that the low-order 8 bits contained in the scratch-pad register designated by the hex digit in N are to be placed into the 8-bit D register. The designated scratch-pad register is left unchanged.

The right half of Fig. 3 illustrates the contents of the CDP1802 registers after this operation is completed. The following sequence of steps is required to perform this operation:

1) N is used to select R. (left half of Fig. 3)
2) R(N) is copied into A.
3) A.0 is gated to the bus.
4) The bus is gated to D.

(right half of Fig. 3)

Memory or I/O data used in various COSMAC operations are transferred by means of the common data bus. Memory cycles involve both an address and the data byte itself. Memory addresses are provided by the contents of scratch-pad registers. An example of a memory operation is

$$M(R(X)) \rightarrow D$$

This expression indicates that the memory byte addressed by R(X) is copied into the D register. Fig. 4 illustrates this operation. The following steps are required:

1) X is used to select R.
2) R(X) is copied into A.
3) A addresses a memory byte.

(left side of Fig. 4)

4) The addressed memory byte is gated to the bus.
5) The bus is gated to D.

(right side of Fig. 4)

Reading a byte from memory does not change the contents of memory.

The 8-bit **arithmetic-logic unit** (ALU in Fig. 2) performs arithmetic and logical operations. The byte stored in the D register is one operand, and the byte on the bus (obtained from memory) is the second operand. The resultant byte replaces the operand in D. A single-bit register **data flag** (DF) is set to "0" if no carry results from an add or shift operation. DF is set to "1" if a carry does occur. During subtraction, DF = 0 if the subtrahend is larger than the minuend, indicating that a borrow has occurred. The 8-bit D register is similar to the accumulator found in many computers.

The internal flip-flop Q can be set or reset by instructions, and can be sensed by conditional branch instructions. The state of Q is also available as a microprocessor output.

Instruction Format

COSMAC operations are specified by a sequence of instruction codes stored in external memory. A one-byte instruction format is applicable for most instructions. Two 4-bit hex digits contained in each instruction byte are designated as **I** and **N**, as shown in Fig. 5.

For most instructions, the execution requires two **machine cycles**. The first cycle fetches or reads the appropriate instruction byte from memory and stores the two hex instruction digits in registers I and N. The values in I and N specify the operation to be performed during the second machine cycle. I specifies the instruction type. Depending upon the instruction, N either designates a scratch-pad register, as illustrated in Fig. 3, or acts as a special code, as described in more detail below.

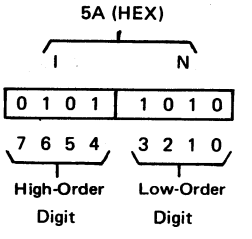


Fig. 5 – One-byte instruction format.

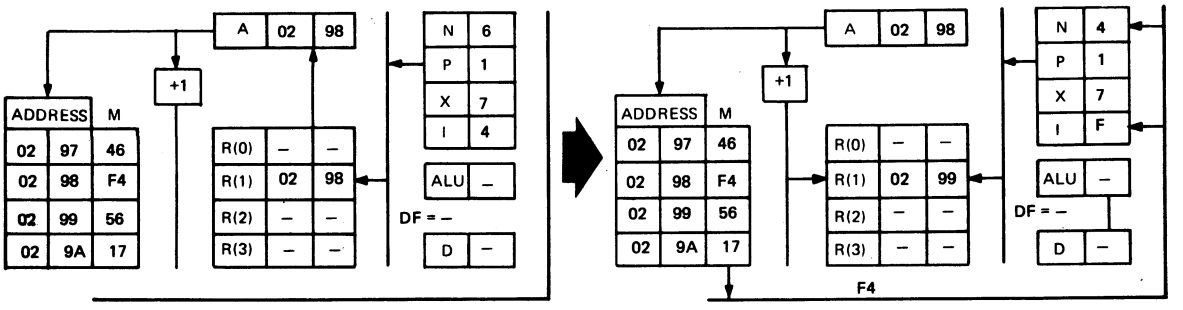


Fig. 6 – Typical instruction fetch cycle.

Instructions are normally executed in sequence. A **program counter** is used to address successively the memory bytes representing instructions. In the COSMAC architecture, any one of the 16-bit scratch-pad registers can be used as a program counter. The value of the hex digit contained in register P determines which scratch-pad register is currently being used as the program counter. The operations performed by the instruction fetch cycle are

$$M(R(P)) \rightarrow I,N;R(P)+1$$

Thus, the program counter is immediately incremented after the fetch cycle of the current instruction.

Fig. 6 illustrates a typical instruction fetch cycle. Register P has been previously set to 1, designating R(1) as the current program counter. During the instruction fetch cycle, the "0298" contained in R(P) is placed in A and used to address the memory. The F4 instruction byte at M (0298) is read onto the bus and then gated into I and N. The value in A is incremented by 1 and replaces the original value in R(P). The next machine cycle will perform the operation specified by the values in I and N. Following the execute cycle, another instruction fetch cycle will occur. R(P) designates the next instruction byte in sequence (56). Alternately repeating instruction fetch and execute cycles in this manner causes sequences of instructions that are stored in memory to be executed.

Although most of the program instructions have a one-byte format, some are two or three bytes in length.

The **immediate** and **short-branch** instructions have a two-byte format, as shown in Fig. 7. For example, the instruction "30" followed by "45" will execute an unconditional branch to the address 45 on the current page; the instruction "FC" followed by "22" will execute an immediate add operation in which the operand 22 is added to the second operand from the D register.

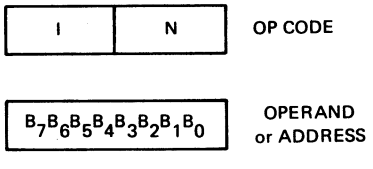


Fig. 7 – Two-byte instruction format.

The **long-branch** instructions have a three-byte format, as shown in Fig. 8. When the instruction "C32F9A" is encountered, a conditional long-branch operation is performed. In this case, if the DF flag is set, a long

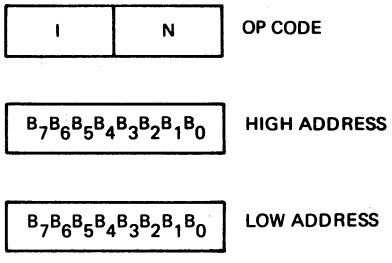


Fig. 8 – Three-byte format for long-branch instructions.

branch to the address 2F9A is executed. If DF is not set, the next instruction in sequence is executed (the one following 9A).

The **long-skip** instructions are one byte and require no address bytes (as the long-branch instructions do).

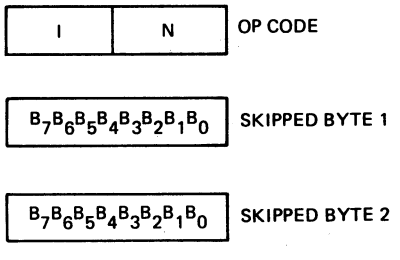


Fig. 9 – Three-byte format for long-skip instructions.

However, the unconditional long-skip and long-skips with test conditions met will, in effect, have the instruction format shown in Fig. 9.

If the test conditions are met, the two bytes are skipped. If the test condition is not satisfied, execution continues at the first byte following the operation code. For a summary of instructions and formats, see Appendix A.

Timing

The CPU machine cycle during which an instruction byte is fetched from memory is called **state 0** (S0). The cycle during which the instruction is executed is called **state 1** (S1). During execution of a program, the CDP1802 generally alternates between S0 and S1, as shown in Fig. 10. Each machine cycle (S0 or S1) is internally divided into eight equal time intervals, as illustrated in the section on **Timing Diagrams**. Each time interval is equivalent to one cycle (T). The rate at which machine cycles occur is, therefore, one eighth of the clock frequency. The **instruction time** is 16T for two machine cycles, and 24T for three machine cycles.

The majority of instructions require the same fetch/execute time. The only exceptions are the long-branch and long-skip instructions. These instructions require two machine cycles for execution. The instruction cycle in these cases contains three machine cycles (one fetch and two execute). The state sequencing will then be as shown in Fig. 11.

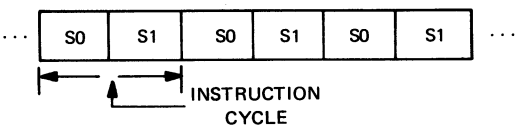


Fig. 10 – Sequence of machine states for normal instruction cycles.

R(P) is incremented after its use. Immediate addressing allows the user to extract data from the program stream without setting up special constant areas in memory and pointers to them. Operations ADD IMMEDIATE (FC) and LOAD IMMEDIATE (F8) are examples of immediate instructions.

In **stack** addressing, one specific CPU register is **implied** as the pointer to memory. Often, R(X) is used, and in one case R(2) is used. A "stack" is a last-in first-out working area in memory used to store intermediate calculations and to keep track of transfers of control between parts of a program.

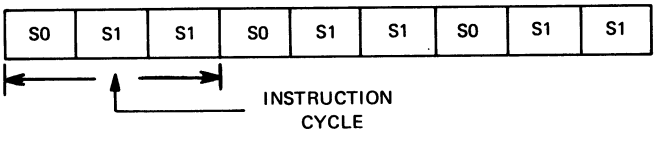


Fig. 11 – Sequence of machine states for long-branch and long-skip instruction cycles.

Addressing Modes

There are four basic modes of addressing in the COSMAC architecture: **register**, **register-indirect**, **immediate**, and **stack**.

In **register** addressing, the address of the operand is contained in the four lower-order bits, the N-field, of the instruction byte. This addressing mode allows the user to directly address any of the 16 scratch-pad registers for the purpose of counting or moving data in or out of registers. Typical instructions in this category are DECREMENT (2N) and GET LOW (8N).

Register-indirect addressing is a variant of indirect addressing utilizing CPU registers as pointers to memory. In this mode, the selected register contains not data, but the address of data. A four-bit address in register N will specify one of the sixteen scratch-pad registers whose contents are the address of data in memory.

Indirect addressing is the dominant mode in COSMAC. It allows the user to address up to 65 kilobytes of memory with a single one-byte instruction.

In **immediate** addressing, R(P) addresses memory so that the operand is the byte following the instruction.

The strength of the COSMAC architecture, and its ability to optimize program size and efficiency as compared with more conventional minicomputer architectures, lies in these four addressing modes and the liberal number of CPU registers. By using stacks for working space, immediate addressing for all constants, register pointers for tabular and vector arrays, and the registers themselves for miscellaneous counters and switches, optimal use of program space is made.

Multiple Program Counters

A **program counter** is a register which points to the next instruction to be fetched and executed. COSMAC provides the unique capability to specify, in a single instruction, any one of the 16 registers as program counter. This feature makes it possible to maintain pointers to several different programs simultaneously and to transfer control quickly from one to another. A pointer to a program which services an interrupt request is a special and important example of this feature.

Instruction Repertoire

This section defines each instruction and describes it in terms of internal machine operation. Examples are given throughout for illustration. The following section, **Instruction Utilization**, covers many of the same instructions but from a why, when, and how point of view. For example, this section defines operation of ADD and ADD WITH CARRY instructions. The **Instruction Utilization** section discusses how these instructions apply and interact in multiple precision arithmetic.

Each CPU instruction is fetched during the S0 machine cycle and executed during the S1 state except for long-branch and long-skip instructions which require two S1 states for execution. The operations performed during the execute cycle S1 are determined by the two hex digits contained in I and N. These operations are divided into eight general classes:

Register Operations – This group includes seven instructions used to count and to move data between internal COSMAC registers.

Memory Reference – Seven instructions are provided to load or store a memory byte.

Logic Operations – This group contains ten instructions for performing logic operations.

Arithmetic Operations – This group contains twelve instructions for performing arithmetic operations.

Branching – Twenty different conditional and unconditional branch instructions are provided. These instructions can be subdivided into sixteen short-branch instructions for in-page operation and eight long-branch instructions to any location in memory space.

Skip Operations – Nine conditional and unconditional skip instructions are provided covering both short- and long-skip instructions.

Control – Ten control instructions facilitate program interrupt, operand selection, branch and link operations, and control of an output flip-flop.

I/O Byte Transfer – Seven instructions are provided to load memory and CPU from I/O control circuits, and seven instructions to transfer data from memory to I/O control circuits.

Each instruction is designated by its two-digit hex code and by a name. A description of the operation is provided using the symbolic notation described earlier. A two-to-four-letter abbreviated name is also given and is used as a mnemonic for assembly language programming. Examples are shown in this section for most instructions. **Note that all the examples illustrate action only during the instruction execute cycle, S1.** R(P) has been incremented during the fetch cycle automatically. A summary of the instruction repertoire is given in Appendix A. It should be noted that "68", which is unused, is reserved for future use by RCA. It is considered an "illegal" code and should not be used.

Register Operations

(Examples illustrate execute cycle only)

INC	INCREMENT REG N	R(N)+1	1N

When I=1, the scratchpad register specified by the hex digit in N is incremented by 1. Note that FFFF+1=0000.

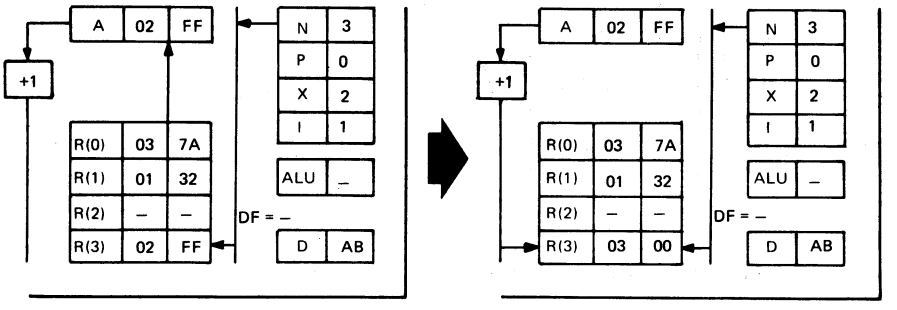


Fig. 12 – Example of instruction 1N – INCREMENT R(N).

DEC	DECREMENT REG N	R(N)–1	2N

When I=2, the register specified by N is decremented by 1. Note that 0000–1=FFFF.

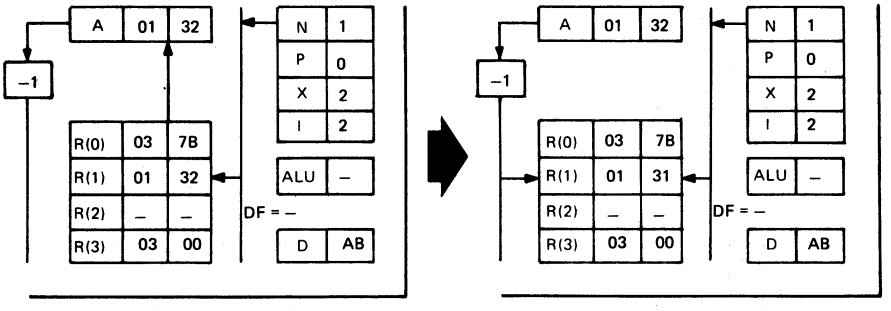


Fig. 13 – Example of instruction 2N – DECREMENT R(N).

IRX	INCREMENT REG X	R(X)+1	60

When I=6 and N=0, the scratch-pad register specified by the hex digit in X is incremented by 1.

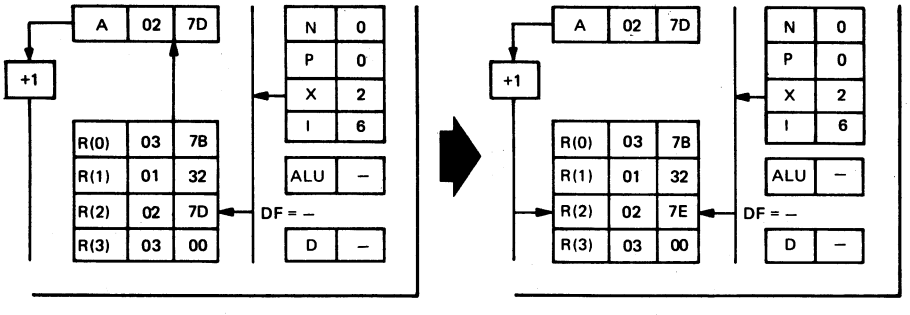


Fig. 14 – Example of instruction 60 – INCREMENT R(X).

GLO	GET LOW REG N	R(N).0 → D	8N

When I=8, the low-order byte of the register specified by N replaces the byte in the D register.

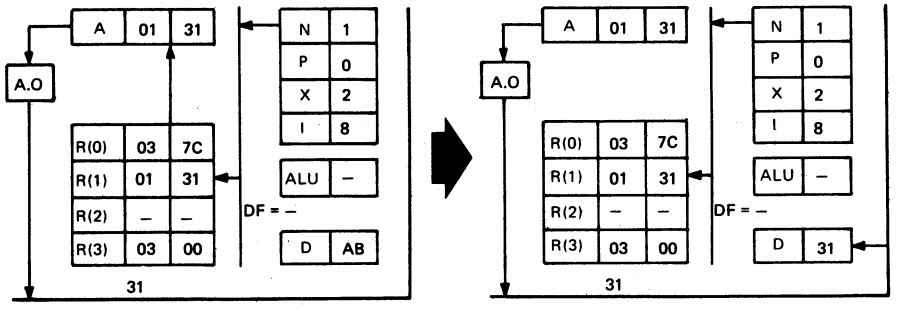


Fig. 15 – Example of instruction 8N – GET LOW.

PLO	PUT LOW REG N	D → R(N).0	AN

When I=A, the byte contained in the D register replaces the low-order byte of the register specified by N. The contents of D are not changed.

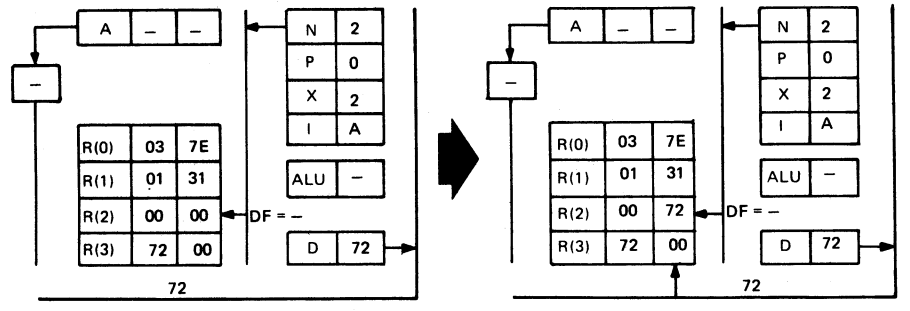


Fig. 16 – Example of instruction AN – PUT LOW.

GHI	GET HIGH REG N	R(N).1 → D	9N

When I=9, the high-order byte of the register specified by N replaces the byte in the D register.

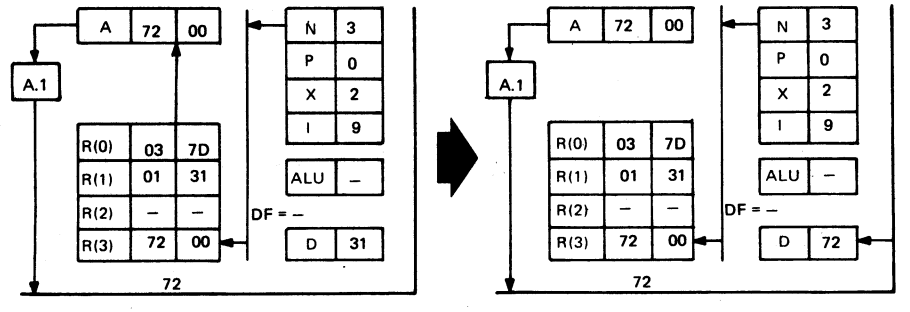


Fig. 17 – Example of instruction 9N – GET HIGH.

PHI	PUT HIGH REG N	D → R(N).1	BN

When I=B, the byte contained in the D register replaces the high-order byte of the register specified by N. The contents of D are not changed.

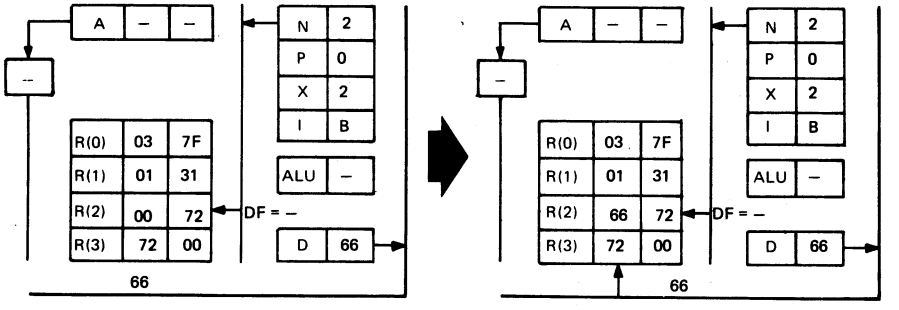


Fig. 18 – Example of instruction BN – PUT HIGH.

Memory Reference

(Examples illustrate execute cycle only)

LDN	LOAD VIA N	M(R(N)) → D; N≠0	0N

When I=0 and N is different from 0, the external memory byte addressed by the contents of the register specified by N replaces the byte in the D register. The contents of memory are not changed.

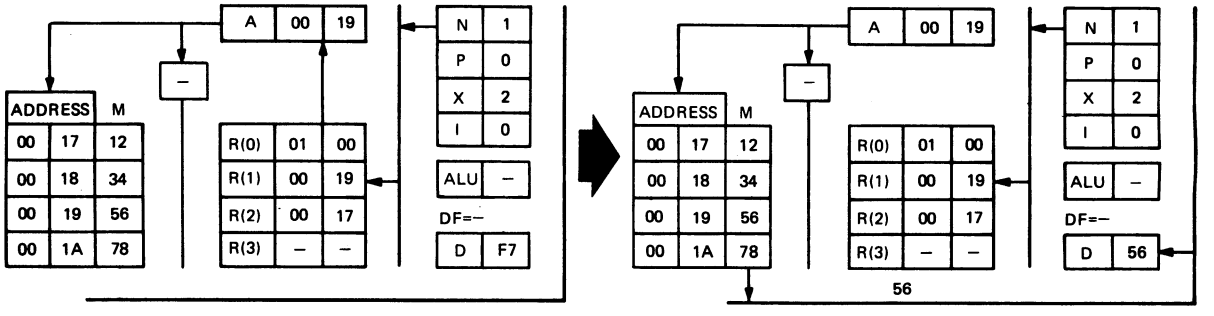


Fig. 19 – Example of instruction 0N – LOAD VIA N.

LDA	LOAD ADVANCE	M(R(N)) → D; R(N)+1	4N

When I=4, the external memory byte addressed by the contents of the register specified by N replaces the byte in the D register. The original memory address contained in R(N) is incremented by 1. The contents of memory are not changed.

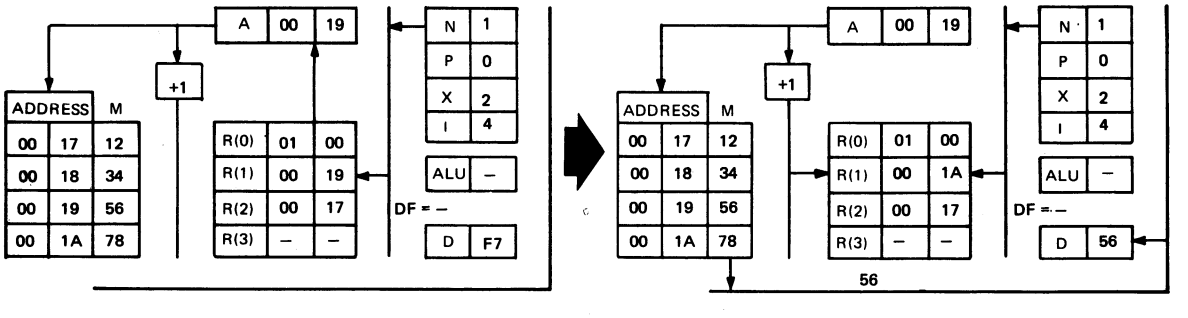


Fig. 20 – Example of instruction 4N – LOAD ADVANCE.

LDX	LOAD VIA X	M(R(X)) → D	F0

When I=F and N=0, the memory byte addressed by the contents of the register specified by X replaces the byte in the D register. (This instruction does not increment the address as LOAD ADVANCE does.) The contents of memory are not changed.

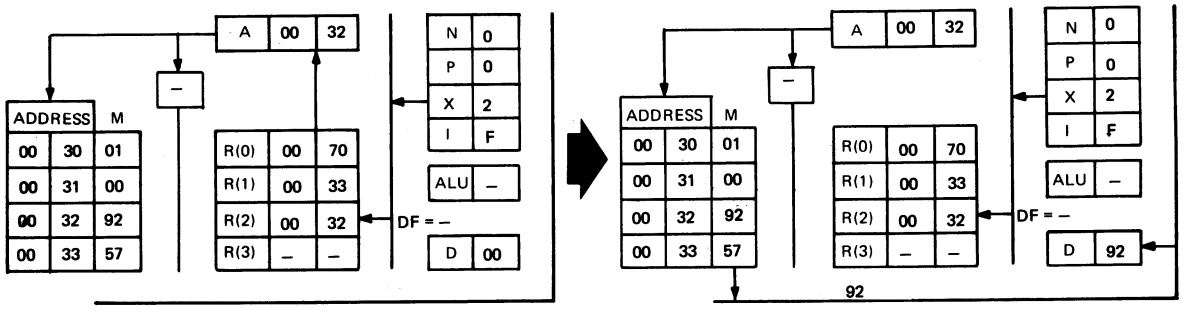


Fig. 21 – Example of instruction F0 – LOAD VIA X.

LDXA	LOAD VIA X AND ADVANCE	M(R(X)) → D; R(X)+1	72

When I=7 and N=2, the external memory byte addressed by the contents of the register specified by X replaces the byte in the D register. The original memory address contained in R(X) is incremented by 1. The contents of memory are not changed.

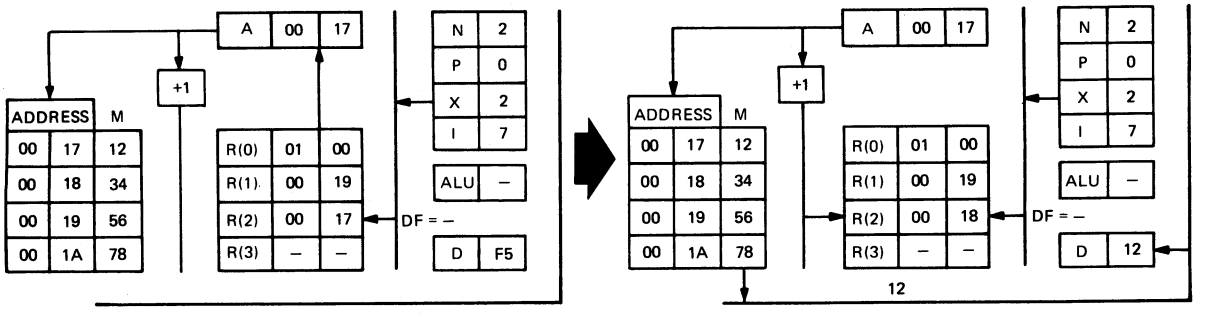


Fig. 22 – Example of instruction 72 – LOAD VIA X AND ADVANCE.

LDI	LOAD IMMEDIATE	M(R(P)) → D; R(P)+1	F8

When I=F and N=8, the memory byte immediately following the current instruction byte replaces the byte in D. Because the current program counter represented by R(P) is incremented again by 1 during the execution of this instruction, the instruction byte following the immediate byte placed in D will be fetched next.

The use of immediate data is a useful way to avoid setting up special constant areas in memory and pointers to them.

This instruction is one of five which load D from memory. It uses R(P) as a pointer, while LDA and LDN use R(N) and LDX and LDXA use R(X). LDI, as well as LDA and LDXA, increments the pointer after use, but LDX and LDN do not.

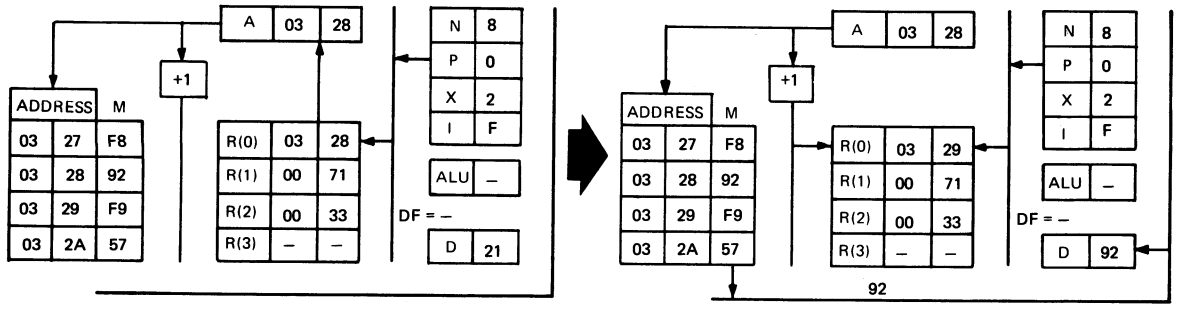


Fig. 23 – Example of instruction F8 – LOAD IMMEDIATE.

STR	STORE VIA N	D → M(R(N))	5N

When I=5, the byte in D replaces the memory byte addressed by the contents of the register specified by N. The contents of D are not changed.

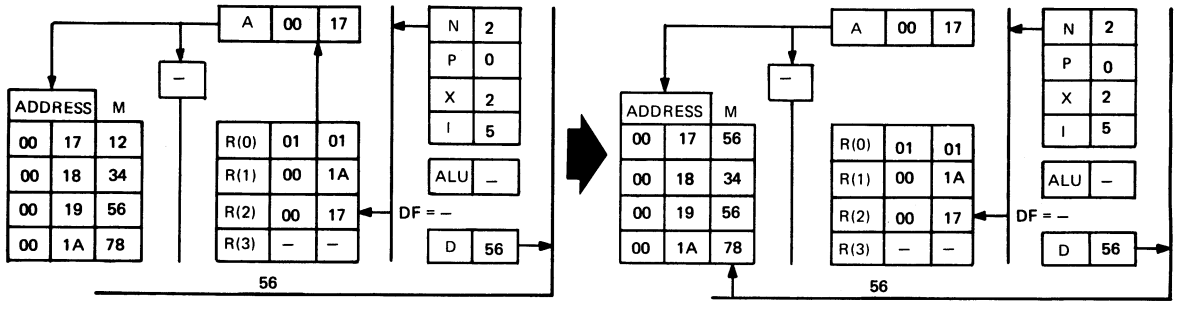


Fig. 24 – Example of instruction 5N – STORE.

STXD	STORE VIA X AND DECREMENT	D → M(R(X)); R(X)–1	73

When I=7 and N=3, the byte in D replaces the memory byte addressed by the contents of the register specified by X. The original memory address contained in R(X) is decremented by 1. The contents of D are not changed.

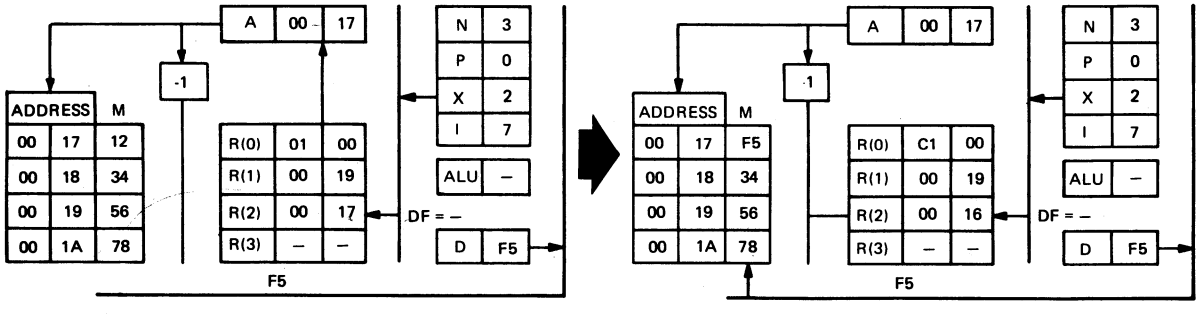


Fig. 25 – Example of instruction 73 – STORE VIA X AND DECREMENT.

Logic Operations

In general, R(X) or R(P) points to one operand, D is the other, and the result replaces the latter in the D register. When R(X) is used as the pointer, the X register must have been previously loaded (by an instruction SET X described among the control instructions). If R(P) is used as the pointer to the operand, it points to the byte in memory after the instruction, called the **immediate byte**. The use of immediate data is a simple way of extracting data directly from the instruction sequence.

(Examples illustrate execute cycle only)

OR	OR	M(R(X)) OR D → D	F1

When I=F and N=1, the individual bits of the two 8-bit operands are combined according to the rules for logical OR as shown to the right. The byte in D is one operand. The memory byte addressed by R(X) is the second operand. The result byte replaces the D operand. This instruction can be used to set individual bits.

M(R(X))	D	OR
0	0	0
0	1	1
1	0	1
1	1	1

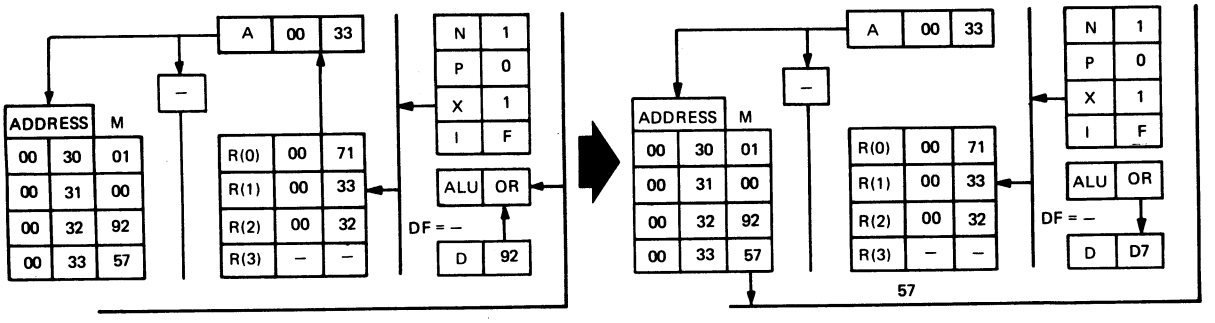


Fig. 26 – Example of instruction F1 – OR.

ORI	OR IMMEDIATE	M(R(P)) OR D → D; R(P)+1	F9

When I=F and N=9, a logical OR operation is performed similar to F1. The D byte is one operand, and the memory byte immediately following the F9 instruction is the second operand. The result goes to D.

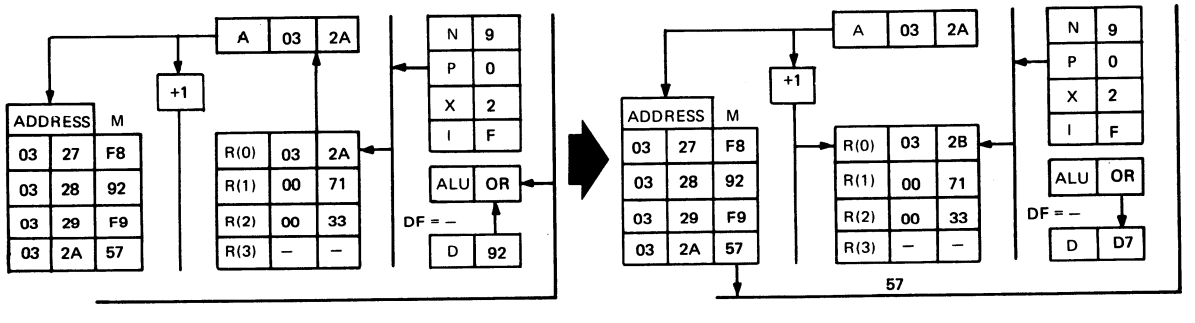


Fig. 27 – Example of instruction F9 – OR IMMEDIATE.

XOR	EXCLUSIVE-OR	M(R(X)) XOR D → D	F3

When I=F and N=3, the individual bits of the two 8-bit operands are combined according to the rules for logical EXCLUSIVE-OR as shown to the right. The D byte and M(R(X)) are the two operands. The result byte replaces the D operand. This instruction can be used to compare two bytes for equality since identical values will result in all zeros in D.

M(R(X))	D	XOR
0	0	0
0	1	1
1	0	1
1	1	0

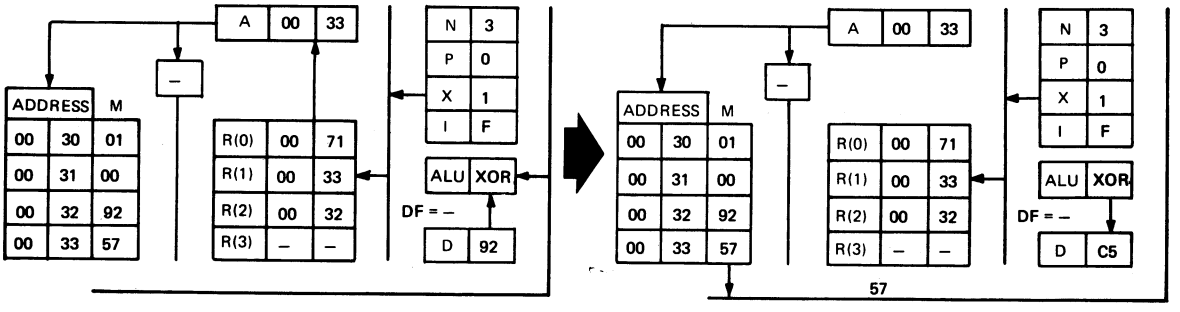


Fig. 28 – Example of instruction F3 – EXCLUSIVE-OR.

XRI	EXCLUSIVE-OR IMMEDIATE	M(R(P)) XOR D → D; R(P)+1	FB

When I=F and N=B, an EXCLUSIVE-OR operation similar to F3 is performed. The D byte is one operand, and the memory byte immediately following the FB instruction is the second operand. This instruction can be used to complement the D register when the immediate byte is "FF".

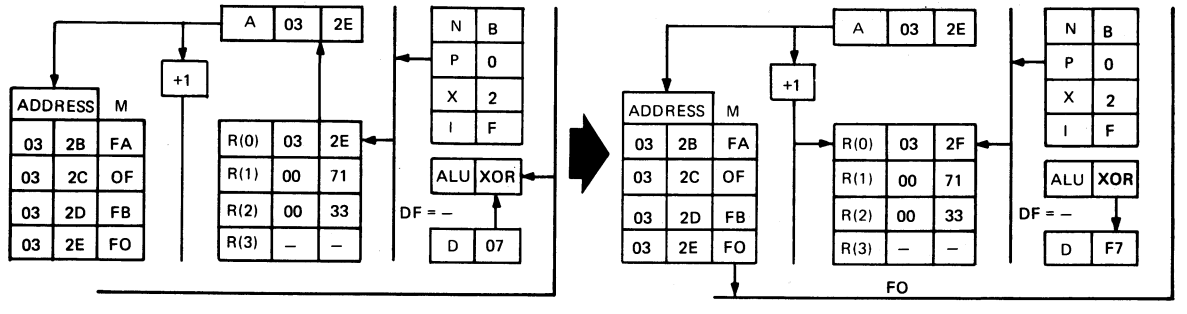


Fig. 29 – Example of instruction FB – EXCLUSIVE-OR IMMEDIATE.

AND	AND	M(R(X)) AND D → D	F2

When I=F and N=2, the individual bits of the two 8-bit operands are combined according to the rules for logical AND as shown to the right. The byte in D is one operand. The memory byte addressed by R(X) is the second operand. The result byte replaces the D operand. This instruction can be used to test or mask individual bits.

M(R(X))	D	AND
0	0	0
0	1	0
1	0	0
1	1	1

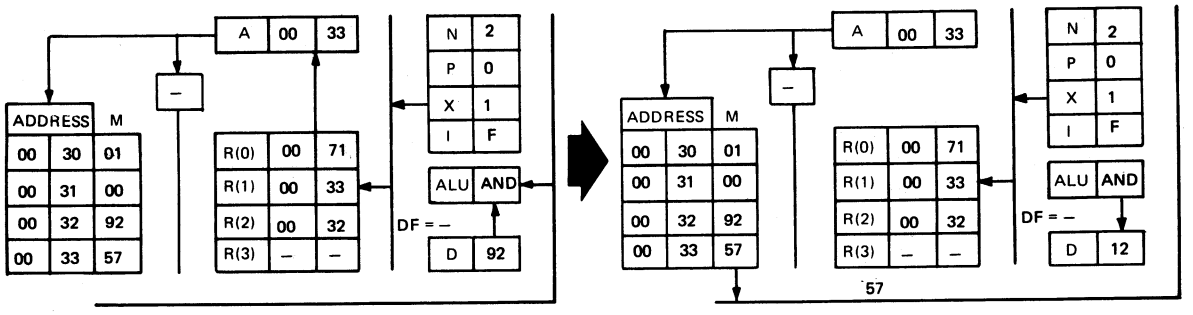


Fig. 30 – Example of instruction F2 – AND.

ANI	AND IMMEDIATE	M(R(P)) AND D → D; R(P)+1	FA

When I=F and N=A, a logical AND operation is performed similar to F2. The D byte is one operand, and the memory byte immediately following the FA instruction is the second operand.

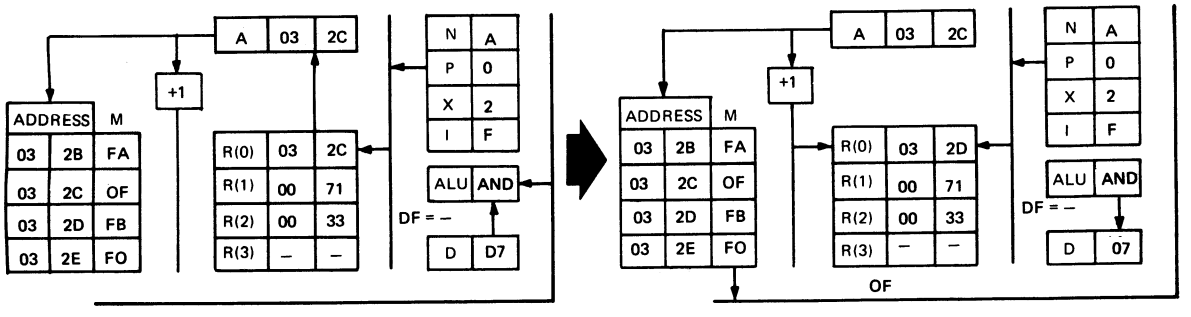


Fig. 31 – Example of instruction FA – AND IMMEDIATE.

SHR	SHIFT RIGHT	SHIFT D RIGHT; LSB(D) → DF, 0 → MSB(D)	F6

When I=F and N=6, the 8 bits in D are shifted right one bit position. The original value of the low-order D bit is placed in DF. The final value of the high-order D bit is always "0". This instruction can be used to test successive bits of the operand or to divide by 2.

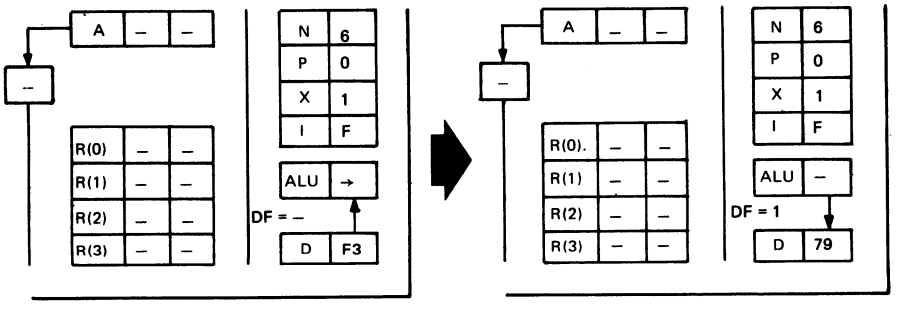


Fig. 32 – Example of instruction F6 – SHIFT RIGHT.

SHRC RSHR	SHIFT RIGHT WITH CARRY RING SHIFT RIGHT	SHIFT D RIGHT; LSB(D) → DF, DF → MSB(D)	76

When I=7 and N=6, the contents of the D register are shifted one bit position to the right. The low-order bit of the D register becomes the carry bit (DF), while the carry bit becomes the high-order bit of the D register.

Either mnemonic may be used for this instruction.

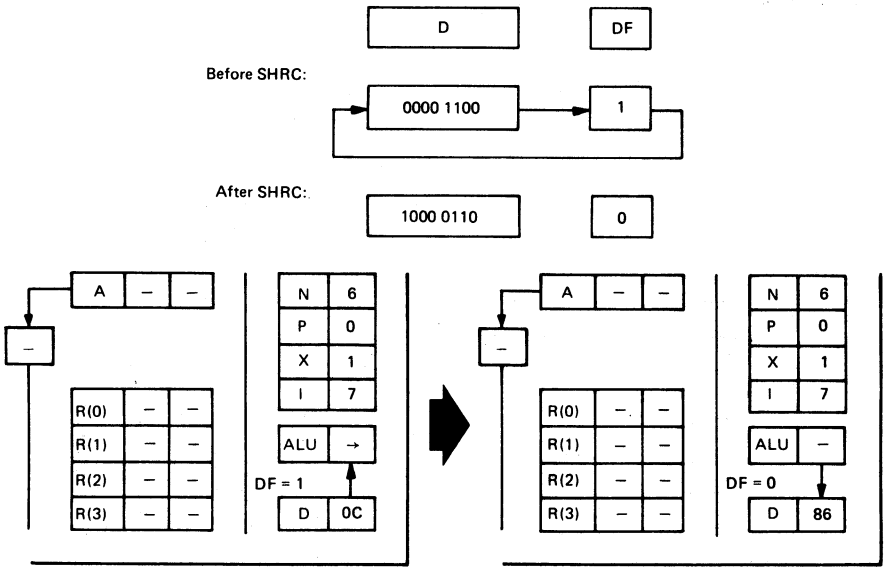


Fig. 33 – Example of instruction 76 – SHIFT RIGHT WITH CARRY.

SHL	SHIFT LEFT	SHIFT D LEFT; MSB(D) → DF, 0 → LSB(D)	FE

When I=F and N=E, the 8 bits in D are shifted left one bit position. The original value of the high-order D bit is placed in DF. The final value of the low-order D bit is always "0". This instruction can be used to test successively bits of the operand or to multiply by 2.

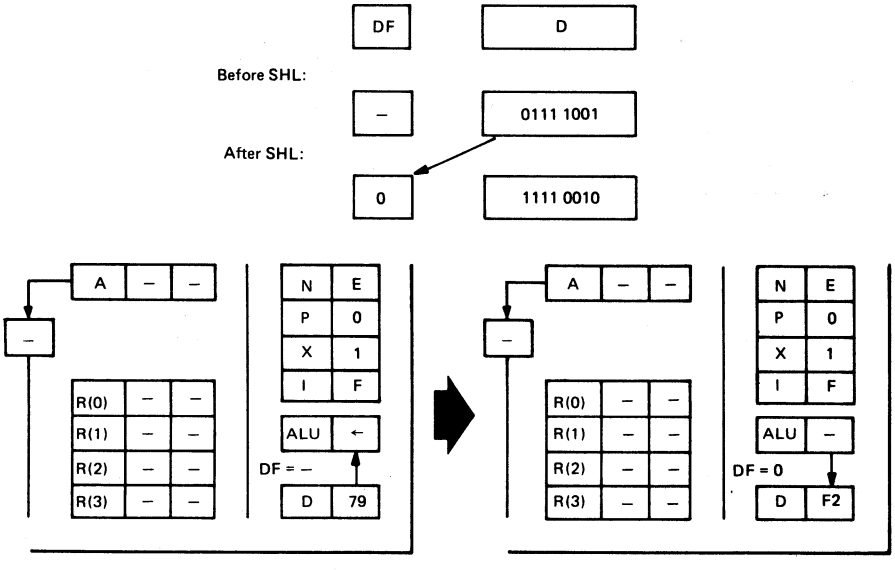


Fig. 34 – Example of instruction FE – SHIFT LEFT.

SHLC RSHL	SHIFT LEFT WITH CARRY RING SHIFT LEFT	SHIFT D LEFT; MSB(D) → DF, DF → LSB(D)	7E

When I=7 and N=E, the contents of the D register are shifted one bit position to the left. The high-order bit of the D register becomes the carry bit (DF), while the carry bit becomes the low-order bit of the D register. Either mnemonic may be used for this instruction.

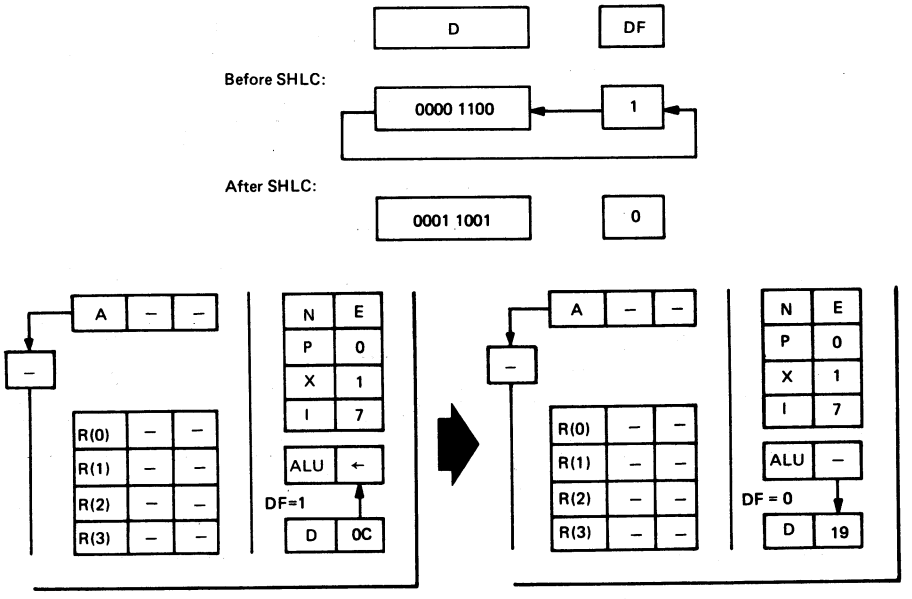


Fig. 35 – Example of instruction 7E – SHIFT LEFT WITH CARRY.

Arithmetic Operations

This group provides the operations ADD, SUBTRACT, and REVERSE SUBTRACT. The three basic instructions are augmented with instructions to handle immediate data, data with carry or borrow, and immediate data with carry or borrow.

In general, R(X) is the pointer to one operand in memory. The other operand is found in D. For immediate data, R(P) is used as the pointer and addresses the byte in memory after the instruction, called the **immediate byte.**

(Examples illustrate execute cycle only)

ADD	ADD	M(R(X))+D → DF, D	F4

When I=F and N=4, two 8-bit operands are added together. The D byte and M(R(X)) are the two single-byte operands. The 8-bit result of the binary addition replaces the D operand. The final state of DF indicates whether or not a carry occurred. It is independent of the original content in DF.

Example 1: 3A + 4B = 85

D register contains 85, DF contains 0

Example 2: 3A + F0 = 12A

D register contains 2A, DF contains 1

The latter example demonstrates overflow. The result is too big for the 8-bit register, and a carry is generated. DF can be subsequently tested with a branch instruction.

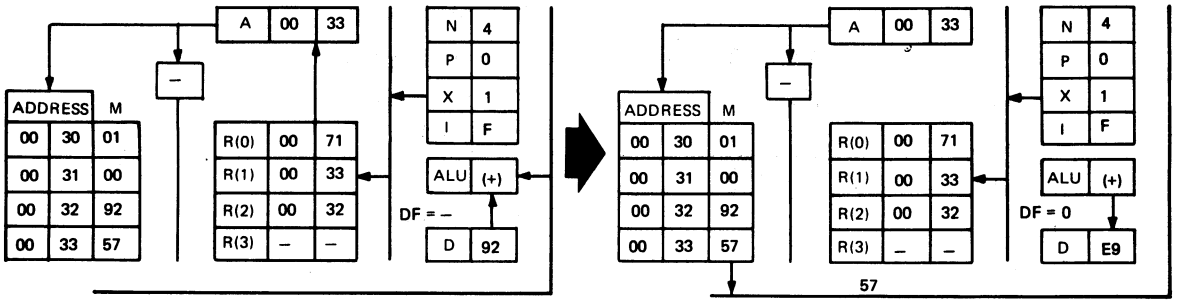


Fig. 36 – Example of instruction F4 – ADD.

ADI	ADD IMMEDIATE	M(R(P))+D → DF, D; R(P)+1	FC

When I=F and N=C, the two operands are added as in F4. The D byte is one operand, and the memory byte immediately following the FC instruction is the other operand.

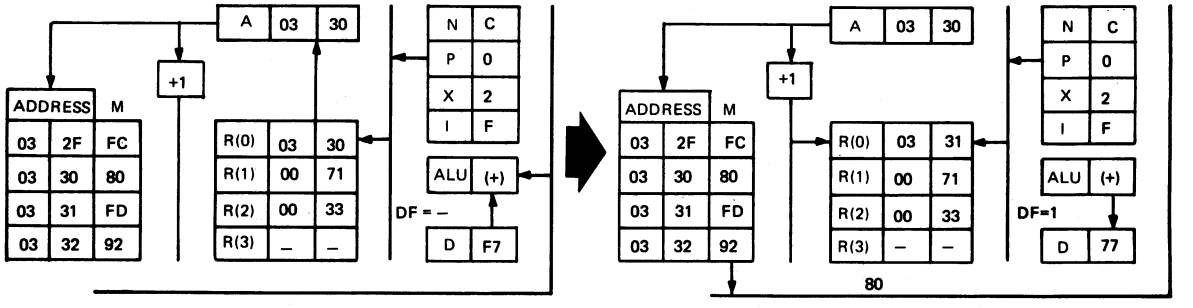


Fig. 37 – Example of instruction FC – ADD IMMEDIATE.

ADC	ADD WITH CARRY	M(R(X))+D+DF → DF, D	74

When I=7 and N=4, the specified byte plus the content of DF are added to the contents of the D register. The 8-bit result of the binary addition replaces the D operand. DF will indicate if the addition generated a carry.

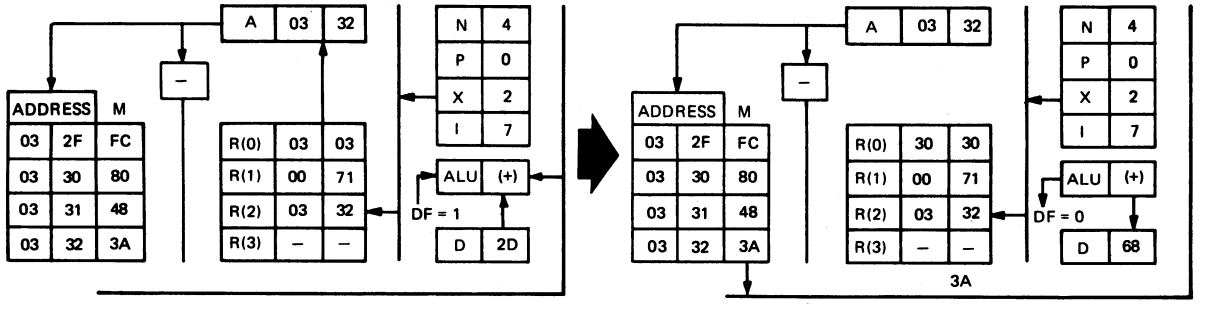


Fig. 38 – Example of instruction 74 – ADD WITH CARRY.

Example 1:

Byte in memory:	3A = 00111010
D register contains:	2D = 00101101
DF contains:	1
Result:	68 = 01101000
After addition:	
D contains:	01101000
DF contains:	0

Example 2:

Byte in memory:	C2 = 11000010
D register contains:	3D = 00111101
DF contains:	1
Result:	100 = 100000000
After addition:	
D contains:	00000000
DF contains:	1

The ADD WITH CARRY instruction is useful when multibyte words are to be added. In the sample above, two 8-bit words were first added (not shown) and generated a carry which must be included in the next higher-order byte addition as shown below. For instance add:

```
  3AF0
+ 2D20

        1
   3A       F0
+ 2D     + 20
   68       10
DF = 0   DF = 1

Final result: DF = 0   6810
```

Similarly to Example 1, the following operations were performed:

```
  C2D1
+ 3D33

        1
   C2       D1
+ 3D     + 33
   00       04
DF = 1   DF = 1

Final result: DF = 1   0004
```

ADCI	ADD WITH CARRY, IMMEDIATE	M(R(P))+D+DF → DF, D;R(P)+1	7C

When I=7 and N=C, the specified byte in memory plus the content of the carry bit is added to the contents of the D register. The final state of DF indicates whether or not a carry occurred.

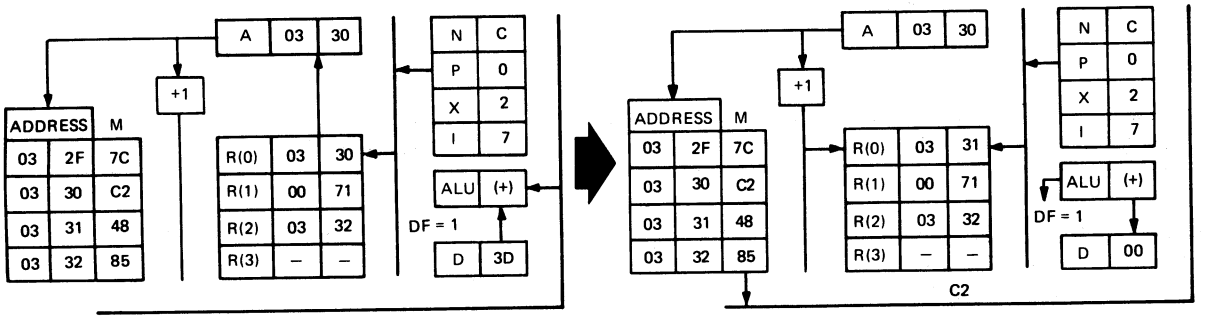


Fig. 39 – Example of instruction 7C – ADD WITH CARRY, IMMEDIATE.

SD	SUBTRACT D	M(R(X))–D → DF,D	F5

When I=F and N=5, the byte in D is subtracted from the memory byte addressed by R(X). The 8-bit result replaces the subtrahend in the D register. Subtraction is 2's complement: each bit of the subtrahend is complemented and the resultant byte added to the minuend plus 1. The final carry of this operation is stored in DF:

DF=0 indicates a borrow
DF=1 indicates no borrow

Example 1: 42 – 0E = 42 + F1 + 1 = 134
D register contains 34, DF contains 1. (No borrow)

Example 2: 42 – 42 = 42 + BD + 1 = 100
D register contains 00, DF contains 1. (No borrow)

Example 3: 42 – 77 = 42 + 88 + 1 = CB
D register contains CB, DF contains 0. (Borrow)

A final value of "0" in DF indicates a borrow and that the subtrahend was larger than the minuend. The answer is negative, but in 2's complement form: taking the 2's complement of CB and assigning a minus sign provides the correct answer (42 - 77 = -35).

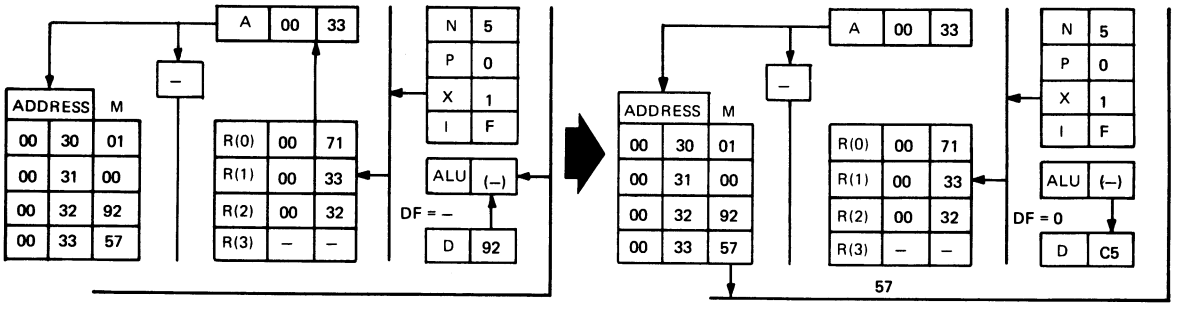


Fig. 40 – Example of instruction F5 – SUBTRACT D.

SDI	SUBTRACT D IMMEDIATE	M(R(P))–D → DF,D; R(P)+1	FD

When I=F and N=D, the two operands are subtracted as in F5. The D byte is the subtrahend, and the memory byte immediately following the FD instruction is the minuend. The final value in DF indicates whether or not a borrow occurred.

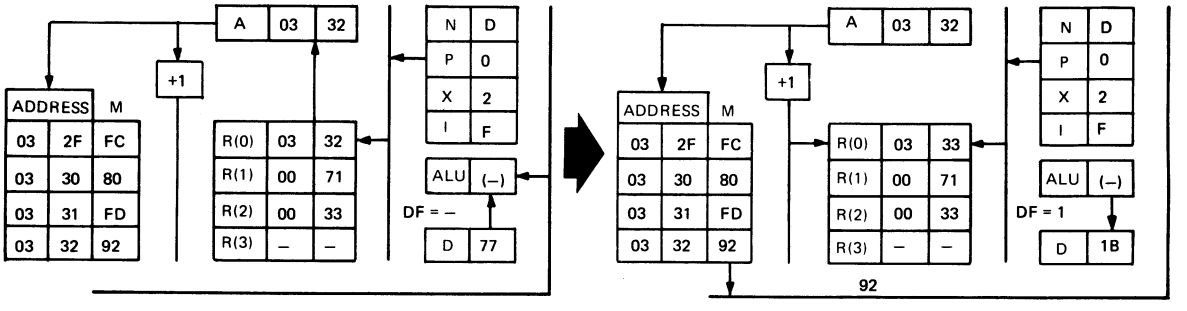


Fig. 41 – Example of instruction FD – SUBTRACT D IMMEDIATE.

SDB	SUBTRACT D WITH BORROW	M(R(X))–D–(NOT DF) → DF, D	75

When I=7 and N=5, the byte in D with a borrow-in from a previous operation is subtracted from the memory byte addressed by R(X). The 8-bit result replaces the subtrahend in the D register. A final borrow is complemented and stored in DF. Subtraction is performed by complementing each bit of the D register and adding it, with the carry-in from a previous operation, to the minuend.

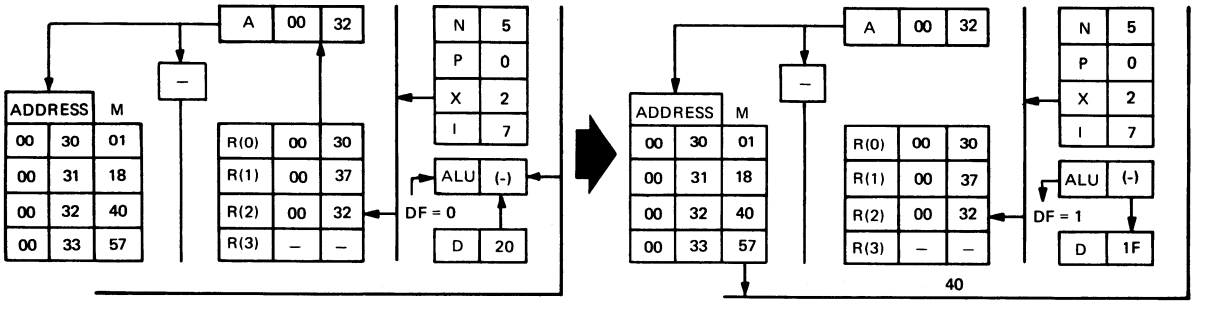


Fig. 42 – Example of instruction 75 – SUBTRACT D WITH BORROW.

The SUBTRACT D WITH BORROW instruction is applicable when multibyte words are subtracted. The following examples assume that two bytes have been subtracted generating a borrow which must be included in the next higher-order byte subtraction. Four alternatives are possible in the subtraction of two words:

CONDITION I: DF = 0, i.e. Borrow = 1
Borrow is present from a preceding carry

Case 1 M(R(X)) > D

Example:

M(R(X)) = 40
D = 20

40 – 20 – 1 = 40 + DF + 0 = 11F
After addition:
D register contains 1F
DF contains 1 (Borrow = 0)

Case 2 M(R(X)) < D

Example:

M(R(X)) = 4A
D = C1

4A – C1 – 1 = 4A + 3E + 0 = 88
After addition:
D register contains 88
DF contains 0 (Borrow = 1)

CONDITION II: DF = 1, i.e. Borrow = 0
No borrow is present from a preceding carry

Case 3 M(R(X)) > D

Example:

M(R(X)) = 64
D = 32

64 – 32 – 0 = 64 + CD + 1 = 132
After addition:
D register contains 32
DF contains 1 (Borrow = 0)

Case 4 M(R(X)) < D

Example:

M(R(X)) = 71
D = F2

71 – F2 – 0 = 71 + 0D + 1 = 7F
After addition:
D register contains 7F
DF contains 0 (Borrow = 1)

In Cases 2 and 4, the answer is a negative number and in 2's complement notation.

SDBI	SUBTRACT D WITH BORROW, IMMEDIATE	M(R(P))–D–(NOT DF) → DF, D; R(P)+1	7D

When I=7 and N=D, the two operands and borrow are subtracted as in instruction 75. The memory byte immediately following the 7D instruction is the minuend. To the minuend is added the complement of the contents in D plus the carry-in in DF from a previous operation. The 8-bit result replaces the contents of the D register and a final borrow is complemented and stored in DF. The program counter is also incremented by 1.

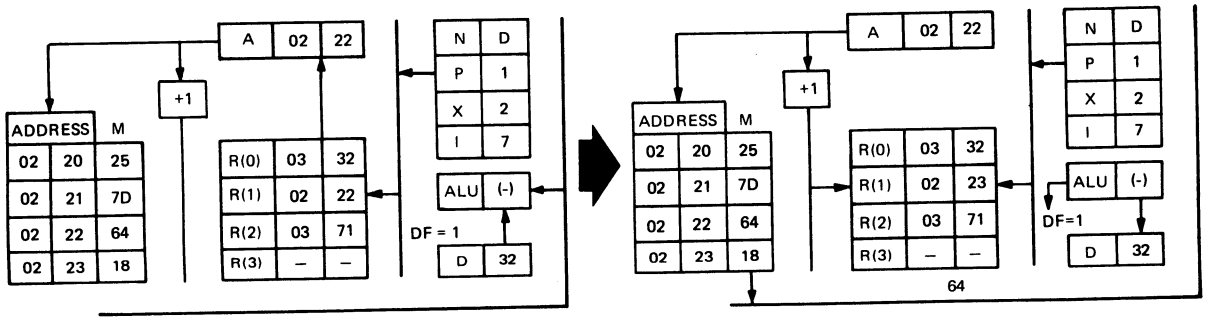


Fig. 43 – Example of instruction 7D – SUBTRACT D WITH BORROW, IMMEDIATE.

SM	SUBTRACT MEMORY	D–M(R(X)) → DF, D	F7

When I=F and N=7, the memory byte addressed by R(X) is subtracted from the byte in D. The result byte replaces the minuend in D. This operation is identical to F5 with the operands reversed. A final borrow is complemented and stored in DF.

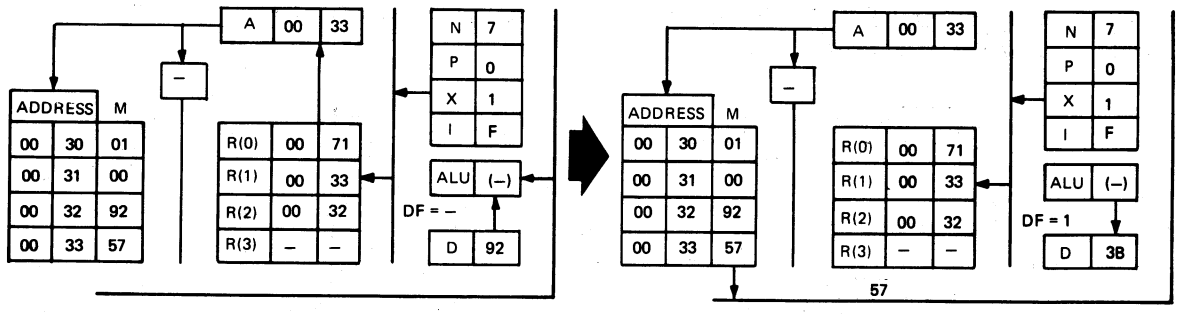


Fig. 44 – Example of instruction F7 – SUBTRACT MEMORY.

SMI	SUBTRACT MEMORY IMMEDIATE	D–M(R(P)) → DF, D; R(P)+1	FF

When I=F and N=F, the two operands are subtracted as in F7. The D byte represents the minuend, and the memory byte immediately following the FF instruction represents the subtrahend. (This instruction is equivalent to FD with the operands reversed.) A final borrow is complemented and stored in DF.

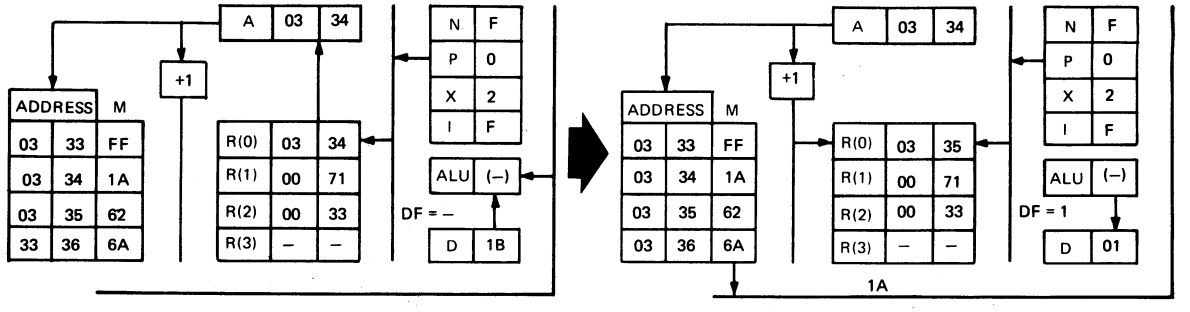


Fig. 45 – Example of instruction FF – SUBTRACT MEMORY IMMEDIATE.

SMB	SUBTRACT MEMORY WITH BORROW	D–M(R(X))–(NOT DF) → DF, D	77

When I=7 and N=7, the byte in memory addressed by R(X) plus the borrow (indicated by DF=0) is subtracted from the byte in the D register. This operation is similar to the instruction 75 but with the operands reversed. The 8-bit result replaces the minuend in D, and DF=0 will indicate if a final borrow occurred. Subtraction takes place by complementing the memory byte addressed by R(X) and adding it with the contents of DF to the minuend in D.

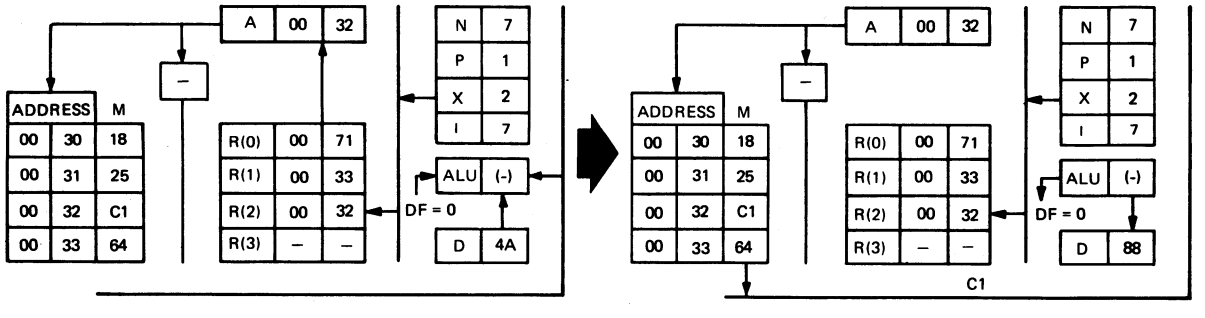


Fig. 46 – Example of instruction 77 – SUBTRACT MEMORY WITH BORROW.

SMBI	SUBTRACT MEMORY WITH BORROW, IMMEDIATE	D–M(R(P))–(NOT DF) → DF, D;R(P)+1	7F

When I=7 and N=F, the two operands and borrow are subtracted as in instruction 77. The immediate byte in memory following the instruction 7F plus the borrow is the subtrahend, and the contents of D is the minuend. The 8-bit result replaces the contents of D, and again DF=0 indicates that a final borrow was generated. The program counter is also incremented by 1.

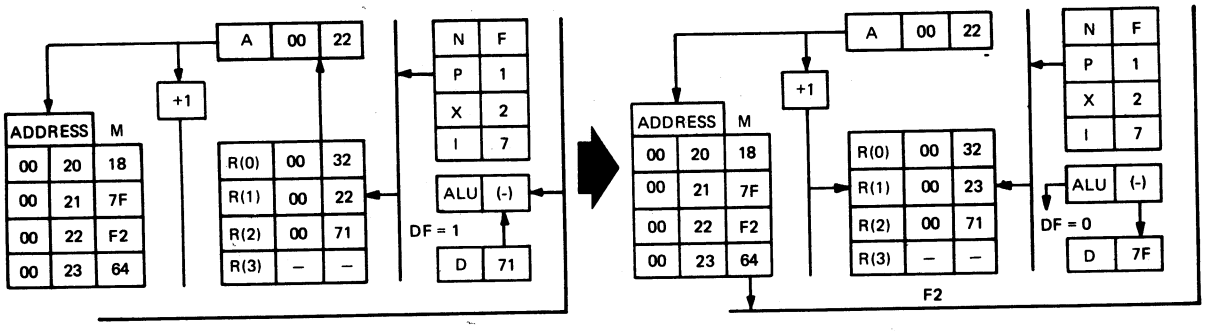


Fig. 47 – Example of instruction 7F – SUBTRACT MEMORY WITH BORROW, IMMEDIATE.

Branching

Short-Branch Operations

The current program counter, R(P), normally steps sequentially through a list of instructions, skipping over immediate data bytes. When I=3, a short branch instruction is executed. The N code specifies which condition is tested. If the test is satisfied, a branch is effected by changing R(P).

When a branch condition is satisfied, the byte immediately following the branch instruction replaces the low-order byte of R(P). The next instruction byte will be fetched from the memory location specified by the byte following the branch instruction. If the test condition is not satisfied, then execution continues with the instruction following the immediate byte. This ability to branch to a new instruction sequence (or back to the beginning of the same sequence to form a loop) is fundamental to stored-program computer usefulness.

Because with this instruction only the low-order byte of R(P) can be modified, the range of memory locations that can be branched to is limited. Since only the low-order 8 bits can be modified, short branching is limited to 2^8 or 256 bytes. Each 256-byte memory segment is called a page. Instructions for branching to any location in memory are described in the next subsection headed "Long-Branch Operations".

The special case of a short branch instruction and its immediate byte occupying the last two bytes in a page is treated as follows: If a branch takes place, R(P).1 is not changed–the branch stays on the same page. If a branch does not take place, execution continues at the first (0th) byte of the next page. A branch instruction on the last byte of a page always leads into the next page, either by branch or by increment. In other words, the address of the immediate byte determines the page to which a branch takes place.

(Examples illustrate execute cycle only)

BR	UNCONDITIONAL SHORT BRANCH	M(R(P)) → R(P).0	30

When I=3 and N=0, an unconditional short branch operation is performed. The byte immediately following the "30" replaces R(P).0.

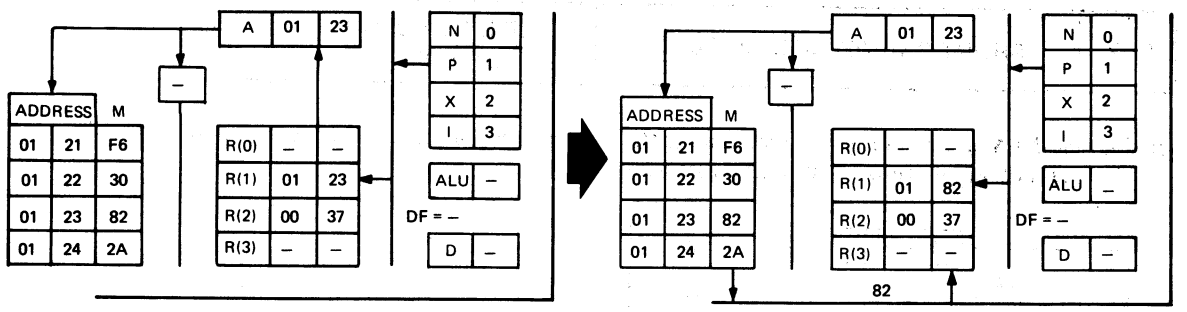


Fig. 48 – Example of instruction 30 – UNCONDITIONAL SHORT BRANCH.

NBR SKP	NO SHORT BRANCH SHORT SKIP	R(P)+1	38

When I=3 and N=8, the name NO SHORT BRANCH implies that the byte following the "38" instruction is an address which will be skipped. This instruction may also be considered to be a SHORT SKIP and is so described in the section on SKIP instructions.

BZ	SHORT BRANCH IF D=0	IF D=0, M(R(P)) → R(P).0 ELSE R(P)+1	32

When I=3 and N=2, a conditional short branch operation dependent on the value of D is performed. The byte in D is examined and if it is equal to zero a branch operation is performed. If the value of D is not zero, R(P) is incremented by 1. This increment causes the branch address byte following the "32" instruction to be skipped so that the next instruction in sequence is fetched and executed.

This instruction can be used following one of the ALU operations described earlier. For example, an EXCLUSIVE-OR operation (F3 or FB) might be used to compare an input byte with a byte representing a constant. A zero result byte in D would represent equality. The "32" instruction could then be used to branch to a location in the program for handling this value of the input byte when D=00, or to proceed to the next instruction in sequence if D≠00, possibly to look for equality with other constants.

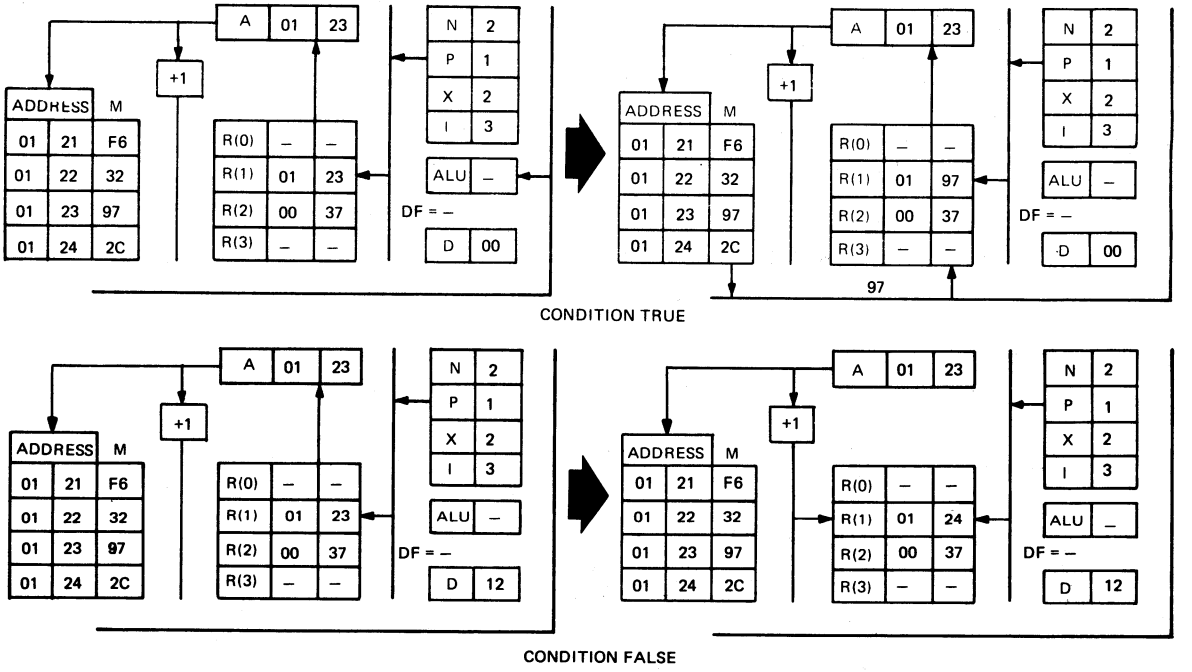


Fig. 49 – Example of instruction 32 – SHORT BRANCH IF D = 0 for both false and true conditions.

BNZ	SHORT BRANCH IF D NOT 0	IF D NOT 0, M(R(P)) → R(P).0 ELSE R(P)+1	3A

When I=3 and N=A, a branch is performed only if the byte in D does not equal zero; if it does, the next instruction in sequence is executed.

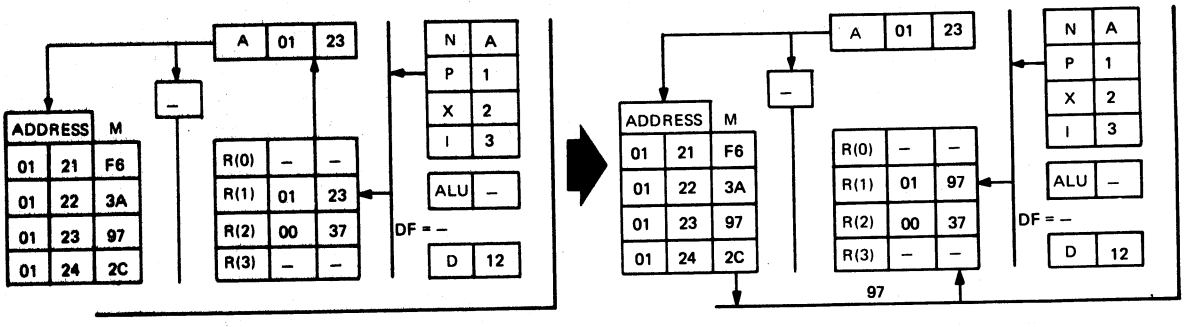


Fig. 50 – Example of instruction 3A – SHORT BRANCH IF D NOT 0.

BDF BPZ BGE	SHORT BRANCH IF DF=1 SHORT BRANCH IF POS OR ZERO SHORT BRANCH IF EQUAL OR GREATER	IF DF=1, M(R(P)) → R(P)0 ELSE R(P)+1	33

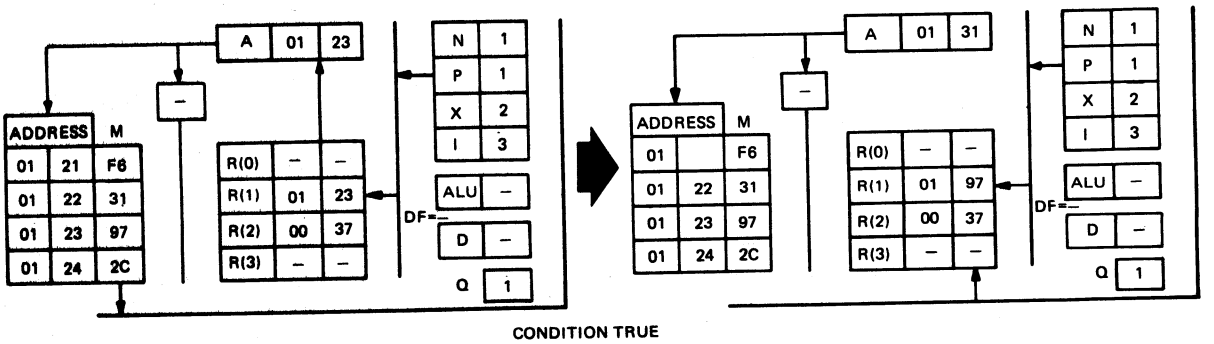
When I=3 and N=3, branching occurs if DF=1. Otherwise, the next instruction in sequence is performed. Examples are not shown for all of the remaining branch instructions because they differ only in the condition tested. The instruction has three mnemonics useful following a shift, subtraction, or comparison (by subtraction), respectively.

BNF BM BL	SHORT BRANCH IF DF=0 SHORT BRANCH IF MINUS SHORT BRANCH IF LESS	IF DF=0, M(R(P)) → R(P).0 ELSE R(P)+1	3B

When I=3 and N=B, a short branch occurs only if DF=0. Otherwise, the next instruction in sequence is fetched and executed. Again, three mnemonics may be useful, all resulting in the same machine action.

BQ	SHORT BRANCH IF Q=1	IF Q=1, M(R(P)) → R(P).0 ELSE R(P)+1	31

When I=3 and N=1, a short branch occurs only if Q=1. Otherwise, the next instruction in sequence is fetched and executed.



CONDITION TRUE

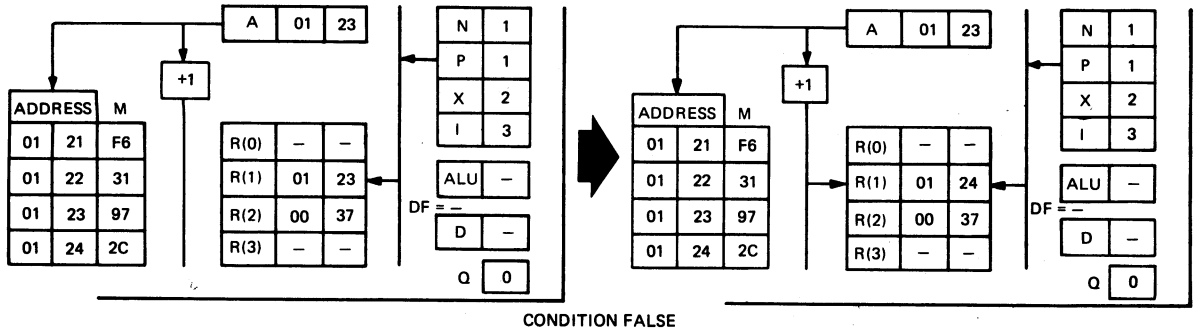


Fig. 51 — Example of instruction 31 — SHORT BRANCH IF Q = 1 for both true and false conditions.

BNQ	SHORT BRANCH IF Q=0	IF Q=0, M(R(P)) → R(P).0 ELSE R(P)+1	39

When I=3 and N=9, a short branch occurs only if Q=0. Otherwise, the next instruction in sequence is fetched and executed.

B1	SHORT BRANCH IF EF1=1	IF EF1=1, M(R(P)) → R(P).0 ELSE R(P)+1	34
BN1	SHORT BRANCH IF EF1=0	IF EF1=0, M(R(P)) → R(P).0 ELSE R(P)+1	3C
B2	SHORT BRANCH IF EF2=1	IF EF2=1, M(R(P)) → R(P).0 ELSE R(P)+1	35
BN2	SHORT BRANCH IF EF2=0	IF EF2=0, M(R(P)) → R(P).0 ELSE R(P)+1	3D
B3	SHORT BRANCH IF EF3=1	IF EF3=1, M(R(P)) → R(P).0 ELSE R(P)+1	36
BN3	SHORT BRANCH IF EF3=0	IF EF3=0, M(R(P)) → R(P).0 ELSE R(P)+1	3E
B4	SHORT BRANCH IF EF4=1	IF EF4=1, M(R(P)) → R(P).0 ELSE R(P)+1	37
BN4	SHORT BRANCH IF EF4=0	IF EF4=0, M(R(P)) → R(P).0 ELSE R(P)+1	3F

When I=3 and N=4,5,6, or 7, short branching occurs only when the corresponding external flag input (EF1, EF2, EF3, or EF4) is held in its "true" state by external circuits (i.e., $\overline{EF1}$, $\overline{EF2}$, $\overline{EF3}$, or $\overline{EF4}$ = 0 or Low).

When I=3 and N=C,D,E, or F, short branching occurs only when the corresponding external flag input (EF1, EF2, EF3, or EF4) is held in its "false" state by external circuits (i.e., $\overline{EF1}$, $\overline{EF2}$, $\overline{EF3}$, or $\overline{EF4}$ = 1 or High).

Long-Branch Operations

The long-branch instructions have a two-byte address and allow branching to any location within the full memory space during three machine cycles (one fetch plus two execute).

LBR	LONG BRANCH	M(R(P)) → R(P).1 M(R(P+1)) → R(P).0	C0

When I=C and N=0, an unconditional long branch is performed. The two bytes in memory following the operation code replace the full 16-bit contents of R(P).

If, for instance, the instruction C0253A has been executed, the next instruction will be found at memory location 253A.

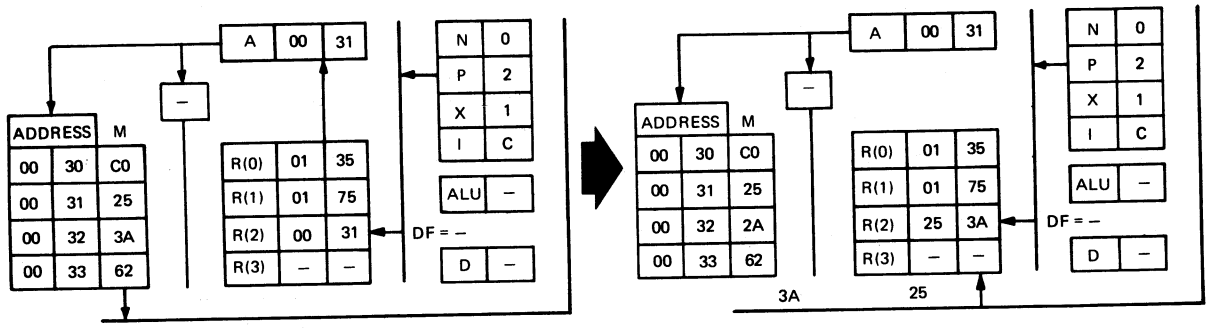


Fig. 52 – Example of instruction C0 – LONG BRANCH.

NLBR LSKP	NO LONG BRANCH LONG SKIP	R(P)+2	C8

When I=C and N=8, the program counter will be incremented twice. For instance, in the instruction sequence C85A2B23, the instruction to be executed following C8 is 23. The name LONG SKIP, LSKP, may also be used (see SKIP instructions). NO LONG BRANCH, NLBR, tells the assembler to expect a two-byte branch address, while LSKP has no restrictions on the next two bytes.

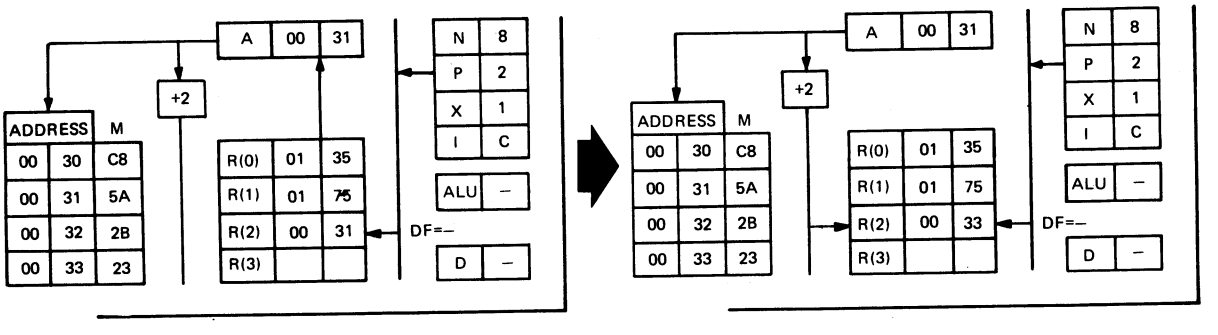


Fig. 53 – Example of instruction C8 – NO LONG BRANCH or LONG SKIP.

LBZ	LONG BRANCH IF D=0	IF D=0, M(R(P)) → R(P).1 M(R(P)+1) → R(P).0 ELSE R(P)+2	C2

When I=C and N=2, a conditional long branch is performed. If D=0, the contents of the program counter R(P) will be replaced with a specified two-byte address. If D≠0, the program counter is incremented twice.

Example: When C2 is fetched from the instruction sequence C21A3343, the next instruction to be fetched is at memory address 1A33 if D=0. If D≠0, execution continues with 43.

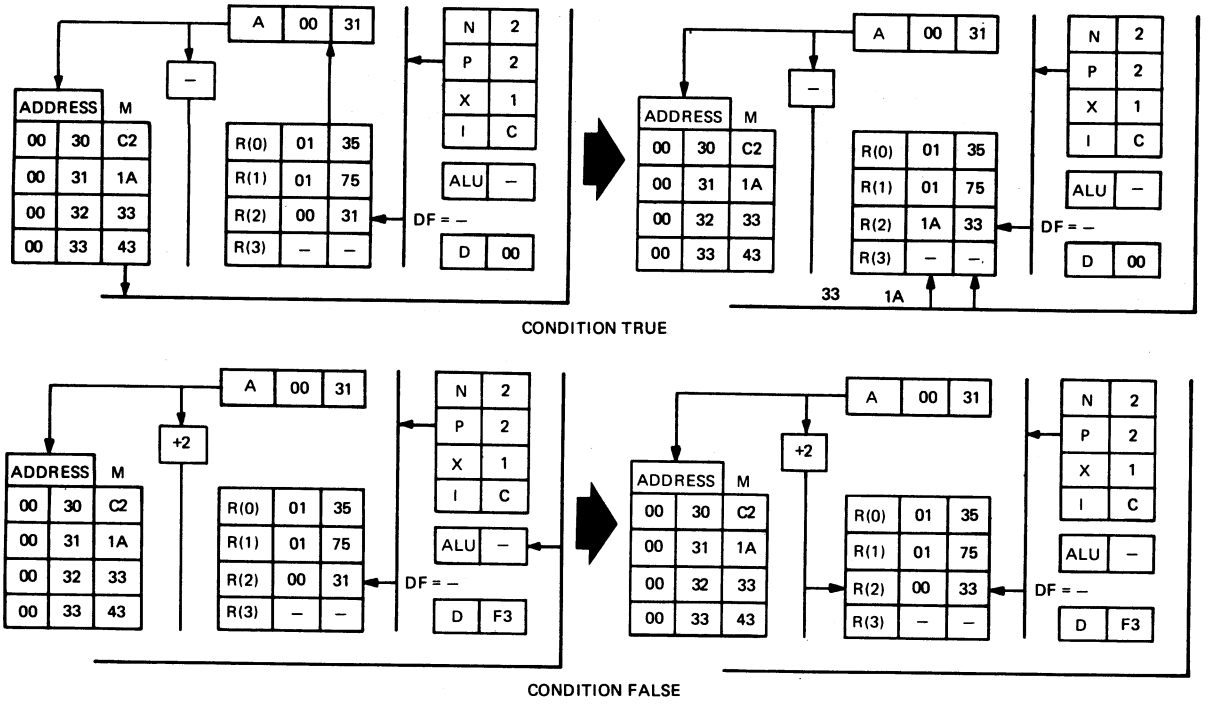


Fig. 54 — Example of instruction C2 — LONG BRANCH IF D = 0 for both true and false conditions.

LBNZ	LONG BRANCH IF D NOT 0	IF D NOT 0, M(R(P)) → R(P).1 M(R(P)+1) → R(P).0 ELSE R(P)+2	CA
LBDF	LONG BRANCH IF DF=1	IF DF=1, M(R(P)) → R(P).1 M(R(P)+1) → R(P).0 ELSE R(P)+2	C3
LBNF	LONG BRANCH IF DF=0	IF DF=0, M(R(P)) → R(P).1 M(R(P)+1) → R(P).0 ELSE R(P)+2	CB
LBQ	LONG BRANCH IF Q=1	IF Q=1, M(R(P)) → R(P).1 M(R(P)+1) → R(P).0 ELSE R(P)+2	C1
LBNQ	LONG BRANCH IF Q=0	IF Q=0, M(R(P)) → R(P).1 M(R(P)+1) → R(P).0 ELSE R(P)+2	C9

Skip Instructions

The SHORT SKIP is unconditional and skips the byte following the operation code. The LONG SKIP is also unconditional but skips two bytes following the operation code. The other instructions are long skips if test conditions for D, DF, or Q are satisfied. The long-skip instructions require three machine cycles, one fetch and two execute, as do the long-branch instructions.

(Examples illustrate execute cycle only)

SKP NBR	SHORT SKIP NO SHORT BRANCH	R(P)+1	38

When I=3 and N=8, the byte following the "38" instruction is skipped. The name SHORT SKIP implies nothing about the following byte, but the alternative name NO SHORT BRANCH implies a branch address.

LSKP NLBR	LONG SKIP NO LONG BRANCH	R(P)+2	C8

When I=C and N=8, the two bytes following the "C8" instruction are skipped. The alternative name NO LONG BRANCH implies that these two bytes represent an unused branch address.

LSZ	LONG SKIP IF D=0	IF D=0, R(P)+2 ELSE CONTINUE	CE

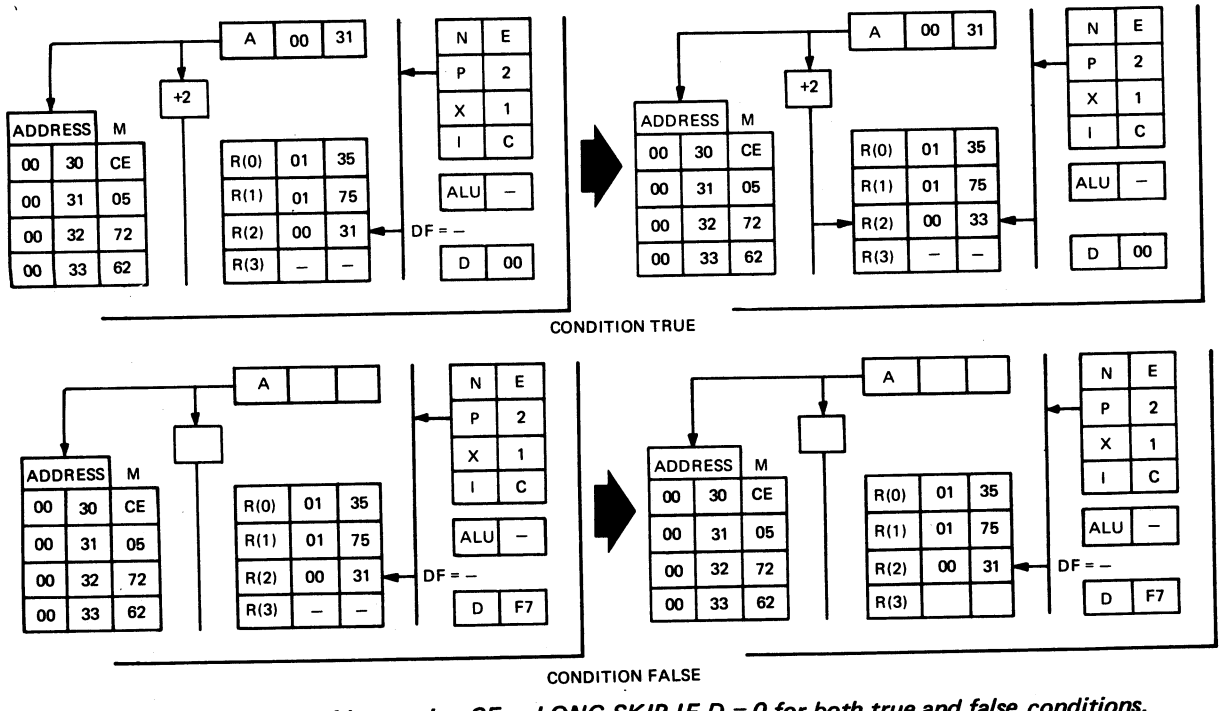


Fig. 55 – Example of instruction CE – LONG SKIP IF D = 0 for both true and false conditions.

LSNZ	LONG SKIP IF D NOT 0	IF D NOT 0, R(P)+2 ELSE CONTINUE	C6

LSDF	LONG SKIP IF DF=1	IF DF=1, R(P)+2 ELSE CONTINUE	CF

LSNF	LONG SKIP IF DF=0	IF DF=0, R(P)+2 ELSE CONTINUE	C7
LSQ	LONG SKIP IF Q=1	IF Q=1, R(P)+2 ELSE CONTINUE	CD
LSNQ	LONG SKIP IF Q=0	IF Q=0, R(P)+2 ELSE CONTINUE	C5
LSIE	LONG SKIP IF IE=1	IF IE=1, R(P)+2 ELSE CONTINUE	CC

When I=C and N=E, 6, F, 7, D, 5, or C, a conditional long skip is performed. If the test conditions for D, DF, Q, or IE are satisfied, the two bytes following the operation code are skipped. If the test condition is not met, normal program execution continues. For instance, if instruction "CD" is fetched from the sequence CD5525F2, the Q bit is examined and if Q=1, the next instruction to be executed is F2. If Q=0, execution continues with instruction "55".

Control Instructions

(Examples illustrate execute cycle only)

IDL	IDLE	WAIT FOR DMA OR INTERRUPT M(R(0)) → BUS	00

When I=0 and N=0, the microprocessor repeats execute (S1) cycles until an I/O request ($\overline{\text{INTERRUPT}}$, $\overline{\text{DMA-IN}}$, or $\overline{\text{DMA-OUT}}$) is asserted. When the request is acknowledged, the IDLE cycle is terminated and the I/O request is serviced, whereupon normal operation is resumed.

NOP	NO OPERATION	CONTINUE	C4

When I=C and N=4, no operation occurs. Execution proceeds with the next sequential instruction (A3 in the example below). This instruction requires three machine cycles, as do the other I=C instructions.

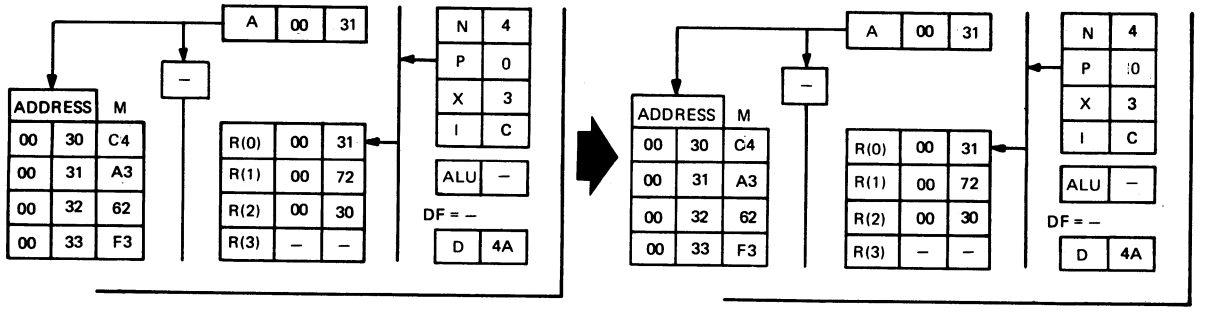


Fig. 56 – Example of instruction C4 – NO OPERATION.

SEP	SET P	N → P	DN

When I=D, the digit contained in N replaces the digit in P. This operation is used to specify which scratch-pad register is to be used as the program counter. This instruction causes a jump to the instruction sequence beginning at M(R(N)). It facilitates "branch and link" functions and subroutine nesting. (These topics are discussed in the section on **Instruction Utilization** and in the section on **Programming Techniques** under the heading "Subroutine Techniques".)

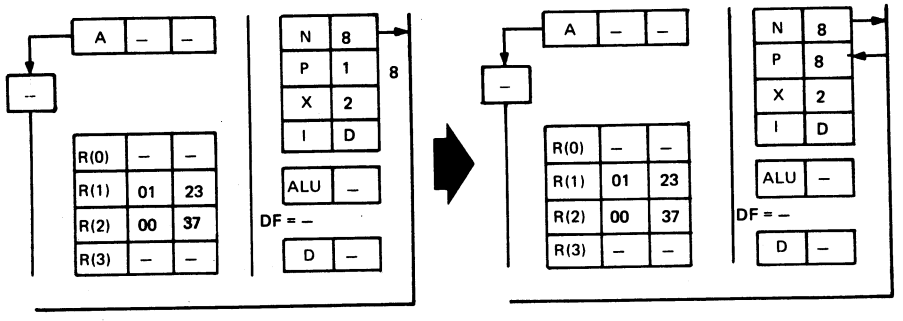


Fig. 57 – Example of instruction DN – SET P.

SEX	SET X	N → X	EN

When I=E, the N digit replaces the digit in X. This instruction is used to designate R(X) for ALU and I/O byte transfer operations.

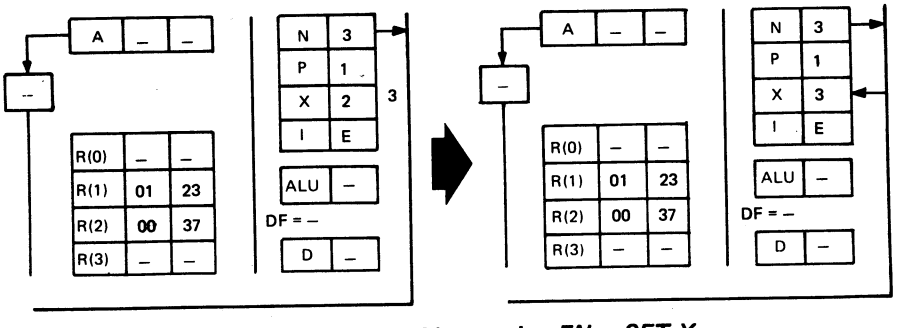


Fig. 58 – Example of instruction EN – SET X.

SEQ	SET Q	1 → Q	7B

When I=7 and N=B, the Q output flip-flop is set. Q was initially reset to "0" in the RESET mode and can later be tested by the branch instructions BQ and BNQ.

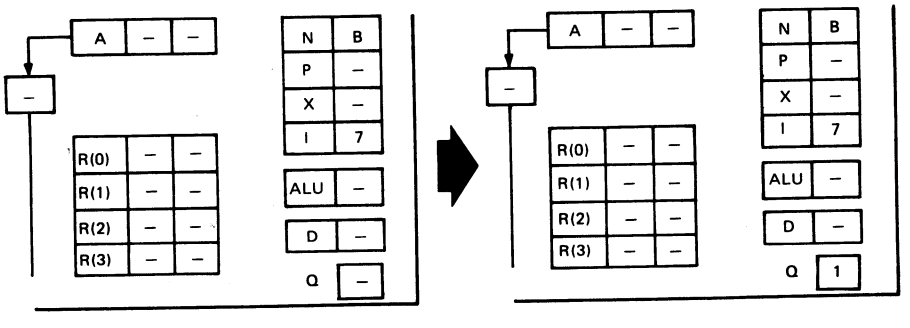


Fig. 59 – Example of instruction 7B – SET Q.

REQ	RESET Q	0→Q	7A

When I=7 and N=A, the output flip-flop Q is reset. Q is initially reset to "0" in the RESET mode and can later be tested by the two branch instructions BQ and BNQ.

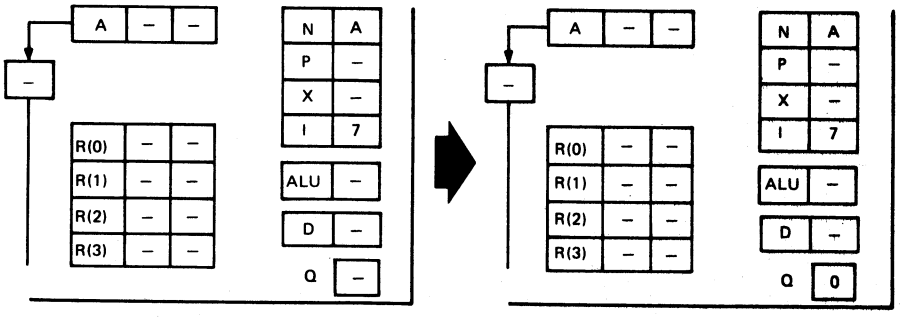


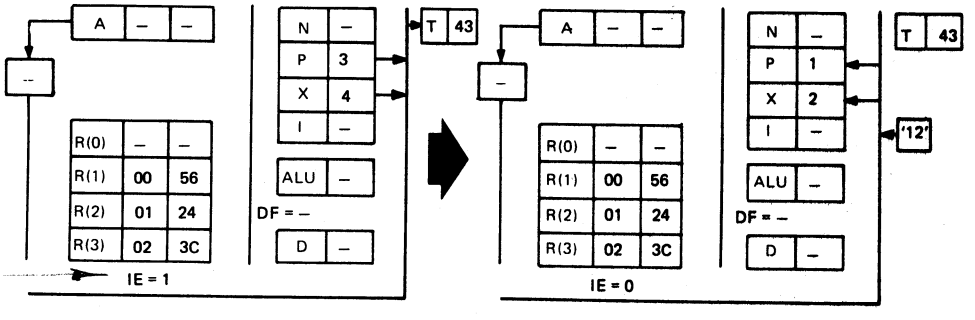
Fig. 60 – Example of instruction 7A – RESET Q.

Interrupt and Subroutine Handling

The special interrupt servicing instructions can best be understood by examining COSMAC's response to an interrupt. When an interrupt occurs, it is necessary to save the current configuration of the machine by storing the values of X and P, and to set X and P to new values for the interrupt service program. The interrupt forces X and P to be automatically transferred into a temporary register T (P goes into the lower 4 bits, while X goes into the higher 4 bits), and forces a value of "1" into P and "2" into X. In addition, further interrupts are disabled by resetting the interrupt enable flip-flop (E) to "0". Also, a specific code is provided on the COSMAC state code line. Details of the interrupt servicing are discussed in the section on **Interfacing and System Operations** under the heading "I/O Interface".

(Examples illustrate execute cycle only)

–	INTERRUPT ACTION	X,P→T; 1→P; 2→X; 0→IE	–



INTERRUPT ENABLE

Fig. 61 – Example of ––– INTERRUPT ACTION.

SAV	SAVE	T→M(R(X))	78

When I=7 and N=8, a SAVE operation is performed. This operation stores the byte contained in the T register at the memory location addressed by R(X). Subsequent execution of a RETURN or DISABLE instruction can then replace the original X and P values to resume (or return to) normal program execution.

MARK	PUSH X, P TO STACK	(X,P) → T; (X,P) → M(R(2)) THEN P → X; R(2)–1	79

When I=7 and N=9, another save operation is performed. The current contents of the X and P registers are stored through the temporary register T and into the byte in memory addressed by R(2). The contents of P are set into X and R(2) is decremented by 1.

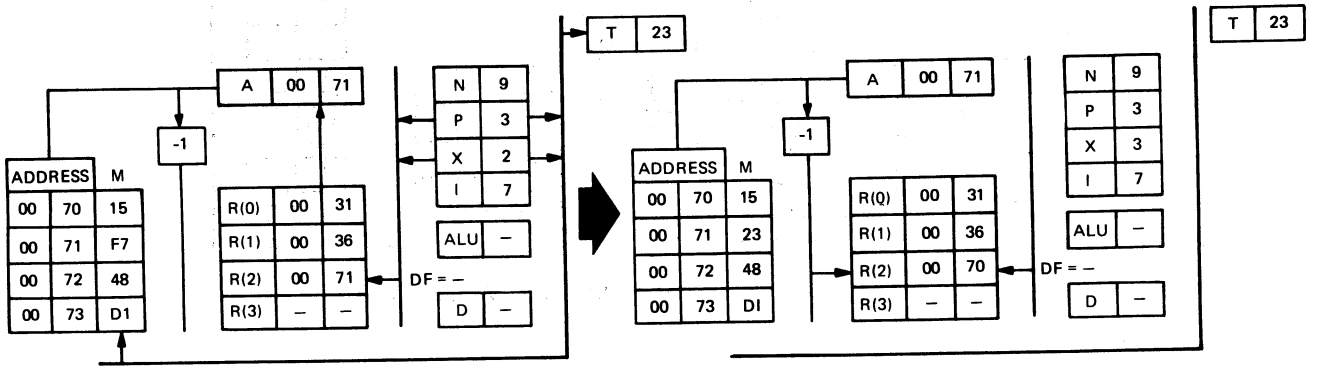


Fig. 62 – Example of instruction 79 – PUSH X, P TO STACK.

RET	RETURN	M(R(X)) → (X,P); R(X)+1; 1 → IE	70

When I=7 and N=0, a RETURN operation is performed. The digits in X and P are replaced by the memory byte addressed by R(X), and R(X) is incremented by 1. The 1-bit Interrupt Enable (IE) flip-flop is set.

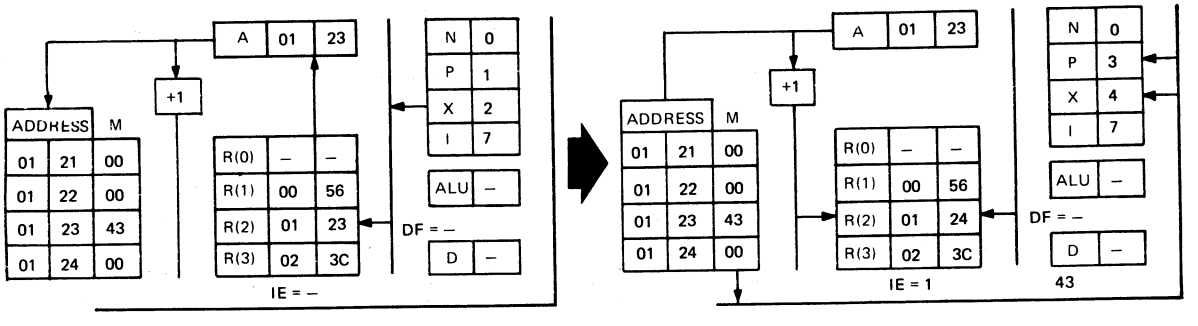


Fig. 63 – Example of instruction 70 – RETURN.

DIS	DISABLE	M(R(X)) → (X,P), R(X)+1; 0 → IE	71

When I=7 and N=1, an instruction similar to RETURN is executed, except that in this case IE is reset. While IE=0, the interrupt line is ignored by the processor.

Either the RETURN or DISABLE instruction can be used to set or reset IE, respectively, as explained in the section on **Programming Techniques** under the heading "Interrupt Service".

Input/Output Byte Transfer

(Examples illustrate execute cycle only)

OUT	OUTPUT	M(R(X)) → BUS; R(X)+1	6N N=1-7

When I=6 and N=1,2,3,4,5,6, or 7, the memory byte addressed by R(X) is placed on the data bus. The three lower-order bits of N are simultaneously sent from the CPU to the I/O system. These three N lines are low at all times except when an Input/Output instruction is being executed (I=6). The I/O system recognizes these conditions and reads the output byte from the lines.

The N lines may be decoded with $\overline{MRD}$ to select or control 7 output devices. For more complex systems, see "I/O Interface" in the section **Interfacing and System Operations.**

R(X) is incremented by 1 so that successively executed output instructions can transfer bytes from successive memory locations. If X is set to the same value as P, then the byte immediately following the output instruction is read out as immediate data.

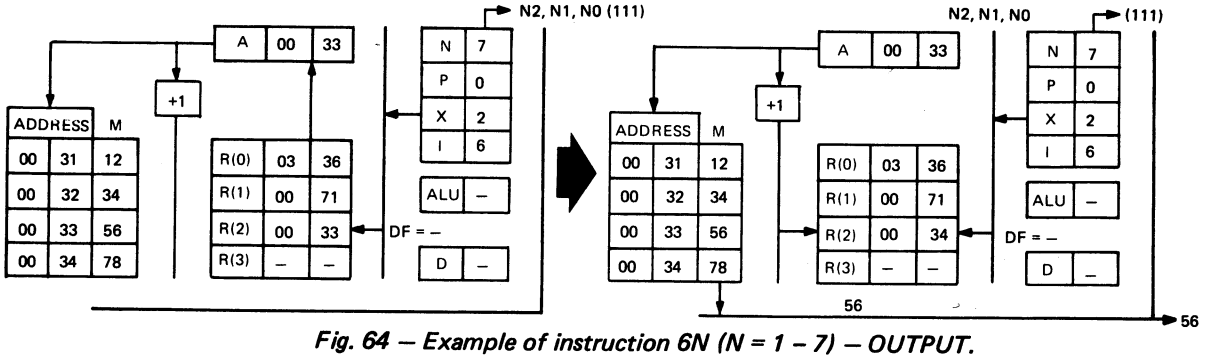


Fig. 64 – Example of instruction 6N (N = 1 – 7) – OUTPUT.

INP	INPUT	BUS → M(R(X)); BUS → D	6N N=9-F

When I=6 and N=9,A,B,C,D,E, or F, an input byte replaces the memory byte addressed by R(X). The input byte is also placed in the D register. R(X) is not modified. The three bits of N are simultaneously sent from the CPU to the I/O system during execution of the instruction. The I/O circuits should gate an input byte onto the data lines during the execute cycle. The N lines may be decoded with $\overline{MRD}$ to select or control 7 input devices. For more complex systems, see "I/O Interface" in the section **Interfacing and System Operations.**

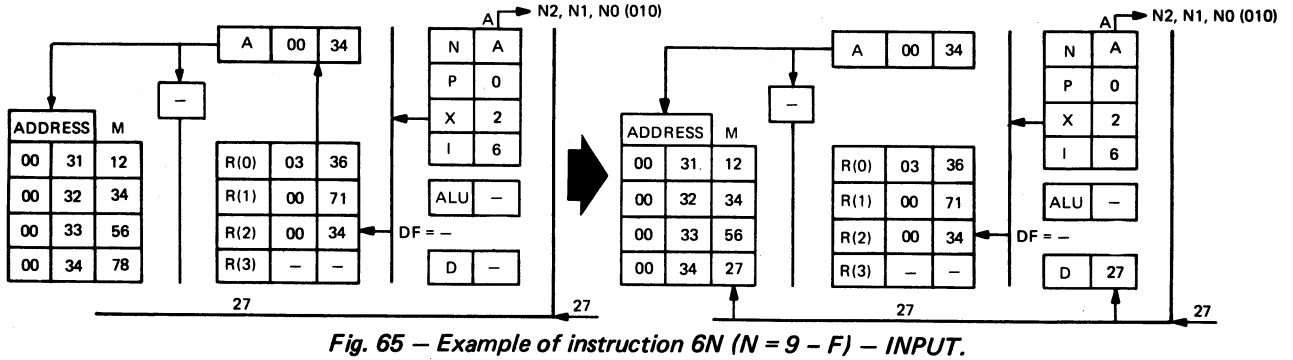


Fig. 65 – Example of instruction 6N (N = 9 – F) – INPUT.

DMA Servicing

–	DMA-IN ACTION	BUS → M(R(0)); R(0)+1	–
–	DMA–OUT ACTION	M(R(0)) → BUS; R(0)+1	–

During DMA operation, R(0) points to a memory location for data transfer. After each byte transfer, R(0) is incremented by 1. For concurrent DMA and Interrupt requests, DMA-IN has priority, then DMA-OUT, and then Interrupt. For further details, refer to "I/O Interface" in the section **Interfacing and System Operations.**

Instruction Utilization

In this section, the basic usage of some of the instructions defined in the preceding section is described from the user's point of view. Additional information on instructions applicable to subroutines and interrupts is given in the subsections on "Subroutine Techniques" and "Interrupt Service" in the section on **Programming Techniques.**

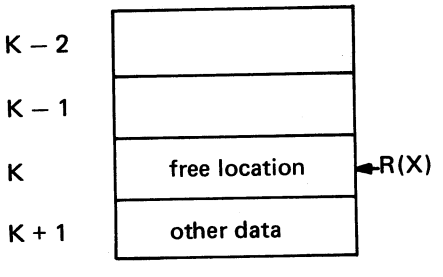
Stack Handling Instructions

These instructions are provided for data movement to and from memory, and are well suited for stack handling. The further use of these instructions in subroutine linkages is discussed in the subsection on "Subroutine Techniques" in the section **Programming Techniques.**

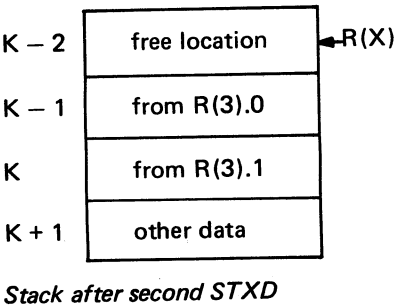
Mnemonic	Op Code
IRX	60
LDX	F0
LDXA	72
STXD	73

Example 1. *Pushing data onto a stack; saving a register*

```
GHI     R3          . . Load R(3).1 into D
STXD                . . Store it onto stack (push)
GLO R3              . . Load R(3).0 into D
STXD                . . Store it onto stack (push)
```



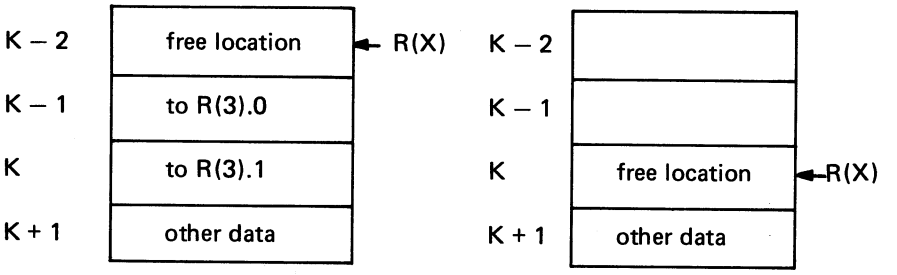
Stack at start up



Stack after second STXD

Example 2. *Retrieving data from a stack; restoring a register*

```
IRX          . . Advance pointer to data
LDXA         . . Load data and advance pointer (pop)
PLO R3       . . Move D to R(3).0
LDX          . . Load data, no advance of pointer (pop)
PHI R3       . . Move D to R(3).1
```



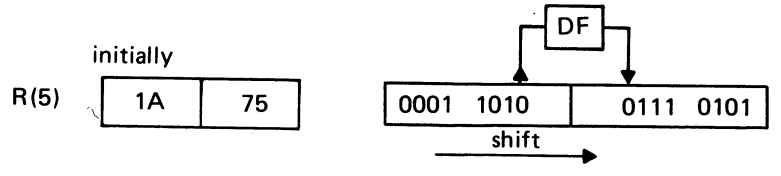
Stack at start up *Stack after using LDXA and LDX*

Shift Instructions

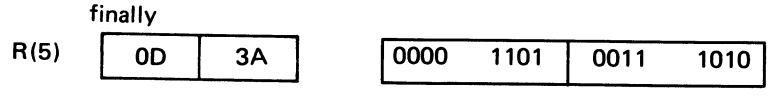
Shift instructions are used for division, multiplication, bit and byte manipulation, and testing. A multiplication by 2 is accomplished by instruction SHL, whereas a division by 2 is done by SHR. Bit shifting in either direction can be performed either by using the basic shift instructions SHR and SHL or by using the SHRC and SHLC instructions. For the basic shift instructions, zeros are shifted appropriately into the D register; bits shifted out of the DF are lost. The shift with carry or ring shift instructions retain all bits by shifting them through the DF and back into the D register.

Mnemonic	Op Code
SHR	F6
SHRC	76
SHL	FE
SHLC	7E

Example. *Shifting the contents of the 16-bit register R(5) one bit to the right*




```
SUB1:   GHI R5      . . Load #1A into D
        SHR         . . Shift it one bit to the right
        PHI R5      . . Store the result (#OD) in R(5).1
        GLO R5      . . Load #75 into D
        SHRC        . . Shift it one bit to the right
                    . . Shifting in carry from SHR
        PLO R5      . . Store the result (#3A) in R(5).0
```



Arithmetic Instructions

Multiple Precision Addition

Multiple precision addition is used to add two operands of multiple byte length. The multiple precision addition is performed by adding the two least significant bytes and then adding the next two bytes to the carry created by the preceding addition. This operation is repeated for each subsequent byte in an operand. Finally, the two most significant bytes are added together with the carry from the preceding addition. The DF will be set to "1" if there is a carry from the two most significant bytes.

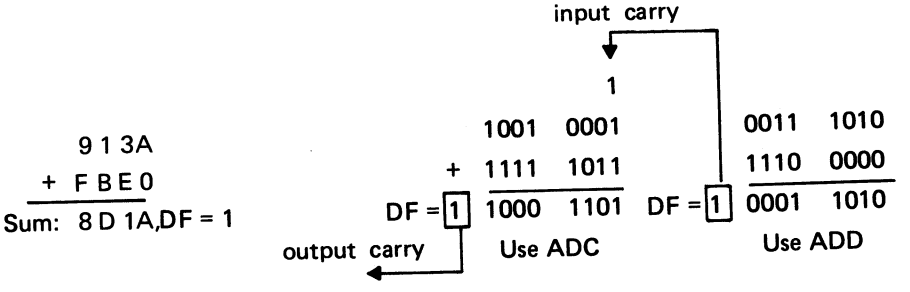
The ADC instruction together with ADD is used for adding two operands of multiple byte length. Consider the addition of two numbers each 2 bytes long.

1. The two least significant bytes are added first by using the ADD instruction. The 8-bit sum will be stored in the D register, and the 1-bit DF (which represents the output carry) will be set to "1" if there is a carry out from the most significant bit. If there is no carry, the DF is reset.

Mnemonic	Op Code
ADD	F4
ADC	74

2. Next, the two most significant bytes are added using the ADC instruction. The state of DF, which represents the output carry from step 1 and the input carry to step 2, will be taken into consideration. The 8-bit sum will be stored in the D register, and the 1-bit DF will be set to "1" if there is a carry from the most significant bit.

Example 1. *Adding two operands #913A and #FBE0 — arithmetic*



Example 2. *Adding two operands each 2 bytes long — assembly code*

Register MA contains the address of the first operand, while the second operand is found in register AC.

```
ADDX:   INC MA      . . Point to low 8-bit memory location
        SEX MA      . . Set X to MA
        GLO AC      . . Fetch AC low 8 bits
        ADD         . . Add the two low-order bytes
        PLO AC      . . Store the result in (AC).0
        DEC MA      . . Point to high 8-bit memory location
        GHI AC      . . Fetch AC high 8 bits
        ADC         . . Add the two high-order bits with carry
        PHI AC      . . Store the result in (AC).1
                    . . AC contains 16-bit sum
                    . . DF = 1 denotes overflow
```

Multiple Precision Subtraction

The concept of multiple precision for subtraction is analogous to that for addition. It is performed by successive subtractions starting with the two low-order bytes and ending with the two high-order bytes. The borrow from each step is included in the next higher-order subtraction.

The SDB instruction together with SD is used to subtract two operands of any byte length. Consider the subtraction of two numbers each 2 bytes long.

1. The least significant byte of the subtrahend in D is subtracted from the least significant byte of the minuend in M(R(X)) by using the SD instruction. The 8-bit result will be stored in the D register, and the 1-bit DF (which represents the borrow) will be set to 1 if there is no borrow out or to 0 if there is a borrow out from the most significant bit.
2. Next, the most significant byte of the subtrahend is subtracted from the most significant byte of the minuend using the SDB instruction. The state of DF, which represents the output borrow from step 1 and the input carry to step 2, will be taken into consideration. The 8-bit result will be stored in the D register and the 1-bit DF will be set to "1" if there is a carry from the most significant bit, i.e. no borrow.

Mnemonic	Op Code
SD	F5
SDB	75

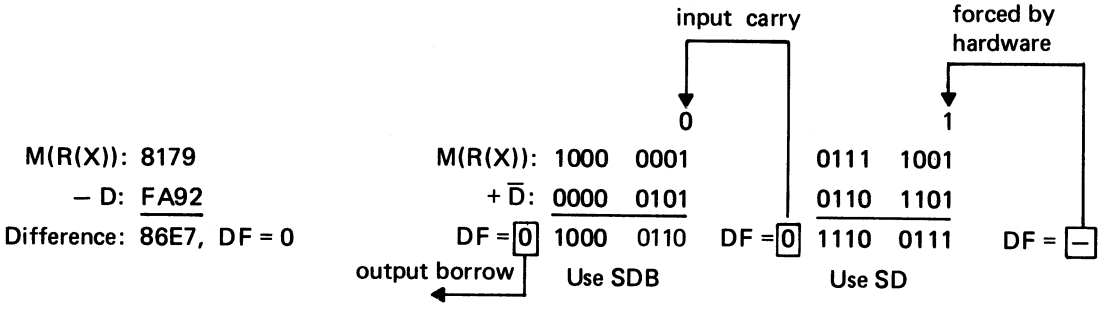
Note that upon completion of subtraction:

DF = 0 means a borrow (the minuend is less than the subtrahend).

DF = 1 means a non-negative result (the minuend is greater than or equal to the subtrahend).

In case of DF = 0, the result is negative; the corresponding value can be obtained by complementing each bit and adding "1" to the result.

Example 1. *Subtracting #FA92 from #8179 – arithmetic*



(D̄ denotes that the D values are complemented)

Example 2. *Subtracting one 2-byte operand from another 2-byte operand – assembly code*

The operand contained in register AC is subtracted from the operand the address of which is in register MA.

```
SDX:  INC MA      . . Point to the low 8 bits
      SEX MA      . . Set X to MA
      GLO AC      . . Fetch AC low 8 bits
      SD          . . Subtract D from M(R(MA))
      PLO AC      . . Store result in (AC).0
      DEC MA      . . Point to the high 8 bits
      GHI AC      . . Fetch AC high 8 bits
      SDB         . . Subtract D from M(R(MA)) with
                  . . Borrow
      PHI AC      . . Store result in (AC).1
                  . . AC contains 16-bit result
```

Interpretation of DF — A Summary

The four shift instructions and the twelve arithmetic instructions are the only ones that can alter the content of DF.

ADD. Executing 74 or 7C allows a carry-in from a previous addition, thus facilitating multibyte addition. Executing F4 or FC, on the other hand, ignores the original content of DF. After any of the four add instructions F4, FC, 74, or 7C, the content of DF will indicate if a carry occurred.

DF = 0 indicates a carry did not occur.

DF = 1 indicates a carry did occur. In unsigned binary representation, DF = 1 also signals an overflow condition.

SUBTRACT. Subtraction is done in 2's complement arithmetic. Each bit of the subtrahend is complemented, and the resultant byte plus 1 is added to the minuend.

Executing 75, 7D, 77, or 7F allows a borrow-in from a previous subtraction, thus facilitating multibyte subtraction. Again, execution of F5, FD, F7, or FF ignores the initial state of DF. After any of the eight subtract instructions above, the content of DF will indicate if a borrow occurred.

DF = 1 indicates no borrow occurred.

DF = 0 indicates a borrow did occur and that the magnitude of the subtrahend was larger than the minuend. The negative answer is then in 2's complement representation.

Branch and Skip Instructions

SKP: When the SKP instruction is used, the byte following it will be unconditionally skipped.

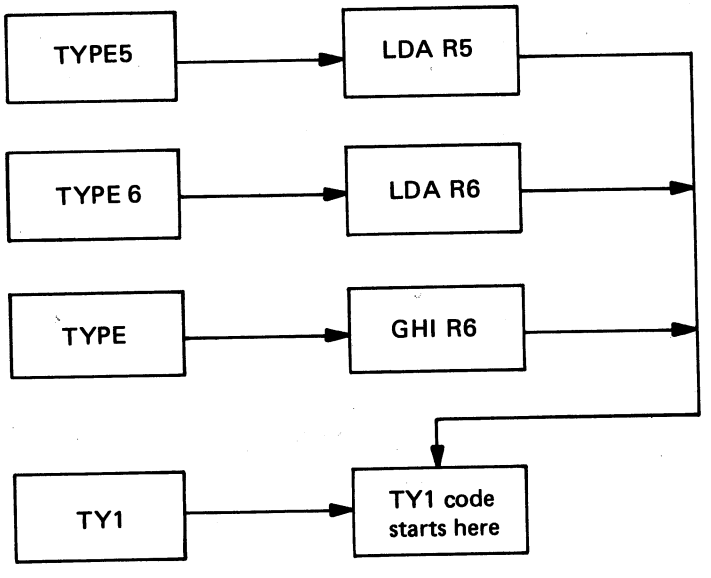
```
      SKP        . . SKIP the next instruction
UP:   INC R1     . . INC R1 if a BRANCH to UP
      GLO R1     . . Always do R(1).0 → D
```

In this case, the INC R1 instruction will be skipped and execution continues at the instruction following INC R1, which is GLO R1.

The SKP instruction can be used in a subroutine with more than one entry point. Depending on the selected entry, the code for the other entries to the subroutine will be skipped.

Mnemonic	Op Code
SKP NBR	38
LSKP NLBR	C8

Example:




```
TYPE5:  LDA R5   ; SKP     . . Skip to TY1
TYPE6:  LDA R6   ; SKP     . . Skip to TY1
TYPE:   GHI R6
TY1:                       . . Subroutine code starts here
```

In the above example the subroutine TY1 has multiple entries. When an entry is selected, the other entries will be skipped. For example, if TY1 is called via TYPE5, the LDA R5 will be executed first, and then execution continues at the first instruction in TY1 subroutine, skipping over the two entries TYPE6 and TYPE.

NBR: When the NBR instruction is used, the branch to the specified address following the NBR instruction will **not** be taken.

```
NBR LABEL
```

In this case, a short branch to the address LABEL will not occur, and execution continues at the instruction following the skipped byte. This instruction may be considered a conditional SHORT BRANCH to LABEL, the condition for which is never met.

The SKP instruction is a different syntactic form of the same machine operation code as for NBR. This form does not require an argument.

LSKP: When an unconditional long skip is executed, the **two** bytes following the instruction will be skipped.

```
LSKP; ADD; INC RA
```

In this case, the two one-byte instructions ADD; INC RA will be skipped, and execution continues at the instruction following INC RA.

The conditional long skip is used to skip on specific conditions of the two bytes following the instruction.

```
      LSDF     .. If DF = 1, RSHR
      ANI #01  .. Else AND #01
MA:   RSHR     .. Always ring shift
```

In this example, the two-byte instruction ANI #01 will be skipped if DF = 1 and execution continues at label MA.

NLBR: When the NLBR instruction is used, the long branch to the specified two-byte address following the instruction will **not** be taken.

```
NLBR LABEL
```

This instruction may be considered a conditional long branch to LABEL, the condition for which is never met.

The NLBR and LSKP, and NBR and SKP pairs, have the same machine operation code. The two assembler syntaxes for each pair exist for the convenience of the user and to aid program debugging.

Control Instructions

NOP: The NOP instruction causes only the program counter to be incremented; it has no additional effects. This instruction is useful in timing loops to provide a time delay or wait function until, perhaps, a certain operation has been completed.

```
      LDI 50     .. Load number of loops
      PLO R6     .. Into R6
MA:   DEC        .. Reduce count
      NOP        .. Delay one instruction
      GLO R6     .. Test for done
      BNZ MA     .. If not done, branch
                 .. Time expired; continue.
```

The NOP instruction can also be used to reserve space for other code which may be unknown at the time the program is prepared. Additionally, it can be used to replace an instruction in a program, thus removing its effect, a useful debugging technique.

SEP: The SEP instruction is used to specify which scratch-pad register is to be used as the program counter. This instruction causes an immediate jump to the instruction sequence beginning at M(R(N)) and R(N) becomes the program counter. The instruction facilitates branch and link functions and subroutine nesting (refer to subsection on "Subroutine Techniques" in the section **Programming Techniques**).

SEX: This instruction is used to designate R(X) used by some logic, arithmetic, register, or I/O byte transfer operations. Setting X to a new value assigns a register

Mnemonic	Op Code
NOP	C4
SEP	DN
SEX	EN
SEQ	7B
REQ	7A
SAV	78
MARK	79
RET	70
DIS	71

R(X) to be used as a pointer to the data byte.

Example: Designating an R(X) can be used to advantage when two bytes stored at different memory locations are compared. The first byte is stored at M(R(7)), and the second is stored at M(R(8)).

```
COMPAR:  SEX R7     .. R(X) points to byte one
         LDX        .. Load the first byte into D
         SEX R8     .. R(X) points to byte two
         SM         .. Compare the two bytes
         BNZ XYZ    .. Branch to XYZ if no match
                    .. Else continue here
```

WHEN X = P: There are three instructions which have particular usefulness when X is set equal to P: the OUTPUT instructions (61-67), the RETURN instruction (70), and the DISABLE instruction (71). Because each of these instructions increments the R(X) register, when X = P the R(P)/R(X) register will be incremented once

for the fetch cycle when it acts as a program counter and once for the execute cycle when it acts as R(X). As a result, the byte immediately following the instruction byte is the operand byte. For example, if P = 3, the sequence will output the byte AD by means of the data bus.

```
E3    SEX R3     . . Set X = 3
61    OUT 1      . . Output a byte from memory
AD      #AD      . . Immediate byte
        ---      . . Next instruction
```

This technique is also useful with the RETURN and DISABLE instructions, as discussed in the subsection on "Interrupt Service" in the section **Programming Techniques**.

SEQ and REQ: Q is a flip flop brought out of the CDP1802 as a single output line. Q can be set (SEQ) or reset (REQ) under program control and later tested in the program by the conditional branch instructions BQ and BNQ. Depending upon the outcome of the test, the program can decide upon a course of action. Note that at start up, the Q line is reset (Q = 0) by the RESET mode ($\overline{CLEAR}$ = L and $\overline{WAIT}$ = H).

For example, the SEQ and REQ instructions can be used to send serial bits of data to an output device (TTY for example). The length of each bit is determined by a time delay subroutine. See Fig 66.

Another application is the control of an external relay or lamp.

SAV: When an interrupt occurs, X and P are automatically transferred into the temporary register T. The SAV instruction is used in the interrupt subroutine to store the byte contained in the T register onto the stack. It is usually preceded by a DEC R2 instruction to make sure that R(2) is pointing to a free memory location. Subsequent execution of a RETURN or DISABLE instruction can replace the original X and P values to return to the interrupted program for normal execution (refer to subsection "Interrupt Service" in the section **Programming Techniques**).

MARK: A primary use of the MARK instruction is to facilitate nested subroutine linkage when multiple program counters are employed. This use is exemplified in the subsection on "Subroutine Techniques" in the section **Programming Techniques**.

A secondary use of the MARK instruction is simply to determine X and P by storing their values in memory for subsequent analysis. This capability is useful in the design of debugging aids.

RET and DIS: Because the interrupt mechanism stores X and P in the temporary register T and is typically followed by the execution of SAV instruction, M(R(2)) contains the value of X and P at the time of interrupt. The DIS and RET instructions are used to restore the machine status (X,P) from M(R(2)) and give control back to R(P). The DIS instruction also resets the interrupt enable flip-flop (IE=0), while RET sets the interrupt enable flip-flop (IE=1). Thus, a return from an interrupt program or subroutine may be made with either interrupt processing enabled or disabled. Note that because interrupt is not sampled during the first instruction, the programmer is able to place the 71 (DIS) instruction in the first memory location followed by the byte 00. The net effect is to inhibit interrupt until the service routine is set up.

The two instructions RET and DIS can also be used in nested subroutine calls in conjunction with the MARK instruction (See subsection "Subroutine Techniques" in the section **Programming Techniques**).

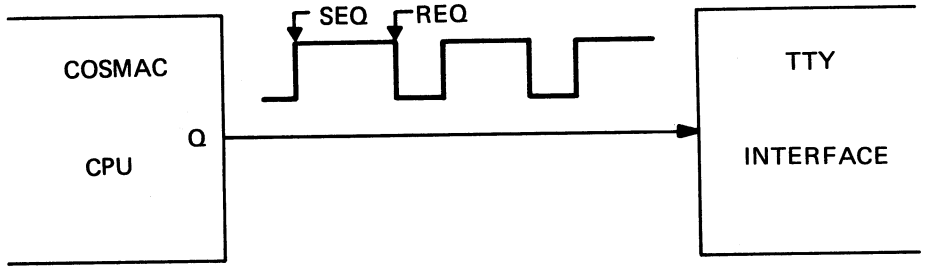


Fig. 66 – Sending serial data from microprocessor to TTY interface.

Programming Techniques

The purpose of this section is to discuss basic programming concepts especially as they relate to the writing of COSMAC programs. It is intended for engineers new to programming. Experienced programmers, however, are also encouraged to read this section to get a feeling for the differences between COSMAC and more conventional computer architectures.

Resource Allocation

Before detailed programming can begin, decisions must be made as to which functions are to be executed by software and which are to be implemented in the input/output hardware. The layout of data in memory must be planned and the utilization of registers worked out.

The **hardware/software tradeoff** is often the most difficult but rewarding phase of designing a microprocessor-based product. On the basis of previous familiarity, engineers may tend to favor incorporation of hardware timers, decoders, rate multipliers, etc. when these functions might be done more economically in software. Generally, the system designer should attempt initially to do everything in software (except jobs requiring submicrosecond response), pushing functions out to special I/O hardware only when the CPU cannot keep up. He may find, even then, that a second CPU/ROM subsystem is more cost-effective than special-purpose hardware.

Allocation of the various built-in I/O capabilities of COSMAC is difficult to discuss in general terms because applications are so varied. The DMA channel can clearly be used for CRT refresh from memory and for block transfers such as between a floppy disc and memory. The decision whether to use the DMA or the Interrupt channels for a slow communication line is more difficult and depends on what other I/O interfacing is required. More subtle uses of DMA include simply using R(0) as a counter (ignoring the data transfer), and using DMA to cycle through a sequence of A/D conversions, for example. The input flags are obviously appropriate for slow-varying binary real-world inputs, but can also be used by I/O circuits to signal status to the CPU. The Q level output may be used as the system output, to signal I/O circuits, or even to select banks of memory.

Often, the most basic system design issue is deciding what functions to carry out in response to one or more interrupt signals. Generally, the less done in servicing interrupts, the better. In this way the amount of bookkeeping overhead is minimized each time an interrupt comes in. It also minimizes the problems of contention among multiple interrupt signals. Furthermore, it makes the system easier to design, debug, and more likely to be error-free.

RAM and Register Allocation for Data

Registers must be allocated among program counter usage, data pointing, storage usage, and general utility usage. This allocation many vary dynamically as a program executes, but generally it is more efficient to assign fixed functions to most registers. The utility registers may be used differently by different parts of the program. Allocation as program counters will be discussed later.

Data may be considered as: 1) isolated variables, parameters, or switches which are referred to at many different parts of a program, 2) temporary, intermediate results obtained in the process of a computation and then

thrown away, 3) constants used as masks or for comparisons, for example, and 4) strings, blocks, or tables of data which relate possibly to I/O operations or are stored in ROM. These types of data are respectively best handled 1) by direct use of registers to hold the random isolated variables, 2) by use of a memory "stack" (defined below) for intermediate results, 3) by use of data intermediate instructions for constants, and 4) by use of register pointers into memory for strings, tables, etc.

The advantage of **using register storage directly** is that simple 1-byte instructions are used to bring data to and from the D register, saving time and program space compared with storing them in RAM. Furthermore, a parameter which needs to be incremented or decremented can be stored in the low half of a register and incremented or decremented in place without using the D register.

A stack is a last-in first-out storage mechanism. It is best implemented in COSMAC by dedicating a block of RAM and using one register R(2) to point at the "top" of the stack, i.e., the space where the next byte should be put. The programmer "pushes" intermediate results onto the stack for storing using R(2) as the pointer and then decrementing the pointer (so that the block of RAM used starts at the highest address). Later, he "pops" them off when he is ready to use them by a load or ALU instruction, incrementing the pointer before he does so. Users of certain Hewlett Packard calculators will be familiar with the idea that a stack can be used to organize a very complicated calculation.

(**A Programming Note**: Often, a programmer knows he will be using a piece of data pushed onto the stack soon with no intervening further use of the stack. In such cases he will omit the decrement of the pointer after pushing data and the increment before using data, thus saving one or two instructions. Such deviations from standard usage should be well marked by comments in the program to avoid problems in case the code is changed later.)

There are many good reasons to use a stack mechanism, some of which are discussed subsequently in the material on program structure and subroutines. For now, the main reason is efficiency of resource use. First, only one register pointer R(2) is required to work with a potentially large number of pieces of data. Second, RAM is used efficiently because the allocation of space required must match the maximum number of intermediate bytes stored at any given time, rather than the total required over the duration of the program (The maximum "depth" of a stack is generally very small). Third, the stack is efficiently addressed by 1-byte COSMAC instructions, thus saving program space.

The **data immediate mode** of addressing is the best COSMAC practice when constants are needed in a program. This mode provides best economy of resources in most cases (no need for a special pointer to memory), and a 1-byte load or ALU instruction suffices to address it. Furthermore, this mode makes code easiest to "read" because each constant used is found at the point in the program where it is needed and its value is immediately obvious. Constants can readily be located and changed during the programming process. With this approach, a constant which is used at several different places in a program will be stored several times. In extreme cases it may be better to set up a pointer to such a constant. In most cases, however, the data immediate mode is usually best.

Data which appears as a string of bytes is best **stored in RAM** and addressed by setting up a **register pointer** to it. Multiple strings usually should have multiple address pointers. The COSMAC instructions are designed to work efficiently with such data, allowing the pointer to be incremented and decremented as the bytes of data are accessed. Sometimes, the programmer will share a few pointers between several different strings of data not being simultaneously accessed. In this case, it is good practice to allocate all strings to one 256-byte page of memory so that a pointer can be moved from one data item to another simply by loading the lower byte of the pointer register.

ROM tables, when frequently used, may also justify a dedicated pointer.

Writing a Program

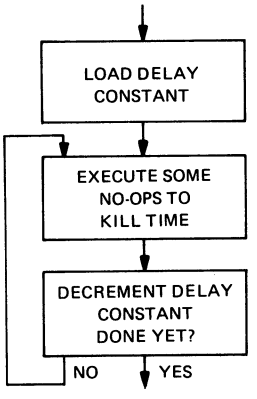
Structure is the essence of programming. The better a programmer organizes the program's structure, the quicker the design, the more efficient the result, and the more likely it is to be correct or "bug-free."

Loops

The most characteristic structure in programming is a simple loop. A loop consists of an initialization section, a main body of steps to be executed, and a test section to determine whether and how often to "loop" through the main body. As a simple example, consider a routine which implements a delay. Fig. 67 shows such a routine in three forms: flow chart, symbolic, and numeric. The programmer should become familiar with all three of these representations.

The **flow chart** (Fig. 67a) shows the program structure explicitly and says in words what happens at each point in the structure.

The **symbolic form**, shown in Fig. 67b, specifies the instructions to be executed and includes the movement of data among the various COSMAC registers. The delay constant, which is assumed to be stored in a memory location, is loaded into D and then moved to the lower half of a utility register UTIL. The expression "LOOP:" labels the next part of the program for future use. Three "NOP" instructions are specified. These instructions are the main body of the loop. Then, the utility register is decremented. Finally, the lower half of UTIL is loaded into the D register and a conventional branch instruction is executed. Control keeps going back to the "LOOP" until the count goes to zero.



a. Flow chart form

```
        LDX         . . Load delay constant      0103 F0
        PLO UTIL    . . into UTIL.0              0104 A8
LOOP:   NOP         . . Kill time                0105 C4
        NOP         . . Kill time                0106 C4
        NOP         . . Kill time                0107 C4
        DEC UTIL    . . Decrement                0108 28
        GLO UTIL    . . Examine current value    0109 88
        BNZ LOOP    . . Branch if not 0          010A 3A05
                    . . Else continue
```

b. Symbolic form

c. Numeric form

Fig. 67 – Simple loop example: delay function.

This same program is also shown in Fig. 67c in **numeric form**, which is as it might appear in hexadecimal code. Note that UTIL is assumed to be R(8).

Conditional Branches

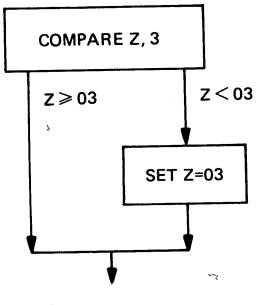
A second characteristic structure in programs employs the conditional branch. In the conditional branch, a comparison or test of some kind is made, and one of two different bodies of code is executed, depending on the outcome. Fig. 68 illustrates a simple example where the intention is to fix a lower bound to a variable Z and substitute a constant 03 if Z is less than 03. Note that once the appropriate action is carried out, the two branches of the program come back together.

By defining steps in a loop to be themselves loops or conditional executions, and by building up a hierarchy of nested loops and conditionals, any function can be programmed. Structures of this type are the most efficient to generate, the most efficient to check out, and the least likely to contain undetected bugs. More complicated structures are very common (they appear in RCA utility programs, for example), because programmers like to play "tricks" such as branching from one part of a program into another, sharing a common part for a while, and then branching back to the original part conditionally on some obscure characteristic that distinguishes the two program parts. These practices, however, lead to problems. A simple change in the one part of code may have unfortunate consequences for another part which is "borrowing" a piece of it in the above manner. The optimum flow chart is usually one without odd-looking branches from one part to another.

Subroutines

Very often, however, a piece of program is useful in many different place in the total program–a multiply routine, for example. To avoid the dangerous practices referred to above, but still to need have only one copy of such a routine present in memory, the programmer should use the concept of a subroutine. A subroutine is a generalized form of instruction, a subprogram which does something that might have been implemented in the original CPU as an instruction. It should be exactly defined so far as the function it performs, where it gets its data and puts its results, and what resources it uses–registers and RAM. Subroutines may have the structure of a loop or of a conditional branch, and in either case may themselves use other subroutines within the body of their code. The main design effort in a large program is in the building up of a set of subroutines suitable for a given application.

COSMAC offers many different ways to handle subroutine structure, representing different tradeoffs among efficiency in execution time, efficiency in program size, and efficiency in use of register resources. These ways are described in the next part of this section. COSMAC also offers more direct mechanisms for treating subrou-



a. Flow chart form.

```
                . . Assume R(X) points to Z
        LDX     . . Fetch Z                  0213 F0
        SMI #03 . . Compare with 03          0214 FF 03
        BGE EXIT                             0216 33 1B
        LDI #03 . . Load 03                  0218 F8 03
        STXD    . . and store at Z           021A 73
EXIT:                                        021B
```

b. Symbolic form.

c. Numeric form.

Fig. 68 – Simple conditional branch example: limiting a variable.

tines as extensions of the basic instruction set. These **interpretive techniques** are described in the last part of this section on **Programming Techniques**.

Subroutine Techniques

In large programs, a particular sequence of instructions is often used many times. For example, a code conversion from one data format to another might be required several places in a communications program. A straightforward approach to programming the code conversion is to insert the proper sequence of instructions each place in the program where the needed function is to be performed. This duplication of instructions, however, would consume much memory storage space, especially if the sequence is long. An alternative method is to write the sequence only once and reuse it each time it is needed. This shared usage of the same code is accomplished by writing the function as a **subroutine** which can be **called** each time it is needed. When the subroutine has completed its function, it **returns** to the program that called it. A subroutine may be called many times from different places in the main program. Most programs will contain several subroutines.

If subroutines are required frequently, the most efficient technique for entering and exiting from the subroutine should be used. The COSMAC architecture provides several techniques for calling and returning from subroutines. The particular technique or combination of techniques to be used is determined by the complexity and requirements of the function to be performed. In the following material, each of three techniques is described along with application examples. Although the techniques are described independently, they are not mutually exclusive and features of each can be combined.

SEP Register Technique

The SEP register technique is the fastest and yet the most basic subroutine call and return convention. It utilizes the COSMAC architecture to rapidly change program counter assignments from one register to another. The procedure is as follows:

STEP 1. Point one of the 16 registers to the subroutine that the program will call. This step is typically accomplished by executing the following code:

```
LDI A.0(sub)  . .replace "sub" with subroutine name
PLO Rn        . .replace "n" with a register number
LDI A.1(sub)  . ."sub" is the entry point to the subroutine
PHI Rn        . ."n" is the register to point to "sub"
```

This four-instruction sequence will load the address of the first instruction in the subroutine into a register. Thus, the register "n" will **point** to the **entry point** of the subroutine. If the programmer does not use this register for any other purpose than to point to the subroutine, the initialization procedure need be done only once. If, however, the same register is to be used variously in the program to point to another subroutine or to hold data, then, before additional calls, the register must be reinitialized.

(**Note**: In many of the examples to follow it will be necessary to initialize pointers using the four-instruction sequence given above. As a shorthand notation, this instruction sequence will be represented by a statement such as:

LOAD 'RN', 'SUB'.

This statement happens to conform to that required of an assembler **macro** call, but for present purposes its use is intended for saving space.)

STEP 2. A call to the subroutine is performed by making R(n) the program counter. This change is done by executing the instruction SEP to register "n". Execution of the subroutine will then begin with R(n) as the program counter. Because the initial value of P is 0 at program start up, this technique is also used to change program counters from R(0) to any other register as shown in Example 1 below.

STEP 3. A return from the subroutine is performed by making R(p) the program counter where p was the register used as the program counter by the calling program at the time of the call to the subroutine. This change is done by executing the instruction SEP to register "p". The execution of the calling program will resume with R(p) the program counter. Examples 2 and 3 illustrate this SEP Register Technique.

The procedure above in basic and expanded forms is illustrated in the following three examples.

Example 1. **Change the Program Counter from R(0) to R(N).**

Any COSMAC program always starts with R(0) as the program counter. Changing from R(0) to any arbitrary register (R(3) in this example) may be necessary to free R(0) for later use as a DMA pointer.

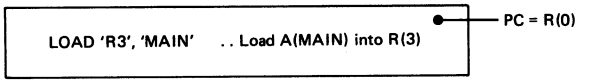
The address of the first instruction in the main program is loaded into R(3). Then, a SEP R3 instruction will cause the main program to use R(3) as the program counter thereby freeing R(0) for DMA use. The register assignment table is given in Fig. 69.

Register	Function	Comment
R(0)	Initial program counter Later, DMA pointer	Defined by hardware
R(3)	Program counter	Register # is arbitrary

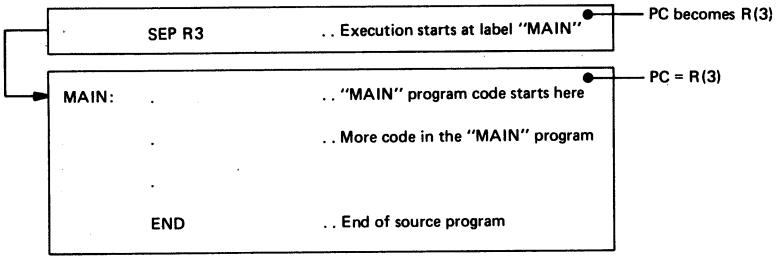
Fig. 69 – Register assignment table for Example 1.

Programming Technique:

. . Execution begins at location 0000 with R(0) as the program counter.
. . Load the address of "MAIN" program into R(3):



. . Change program counter to R(3) to effect an immediate call to the main program:



A typical assembly listing for Example 1 is given in Fig. 70.

Program counter	M Address	M Byte	Assembly program	Comment
R(0)	0000	F834	LDI A.0 (MAIN)	. . Point R3 to
	0002	A3	PLO R3	. . "MAIN" program
	0003	F812	LDI A.1 (MAIN)	. .
	0005	B3	PHI R3	. .
	0006	D3	SEP R3	. . Change PC to R3
R(3)	1234		MAIN: ORG *	. . "MAIN" program starts here

Fig. 70 — Assembly listing for Example 1.

Example 2. **Main Program Calling a Subroutine.**

In this example, a subroutine is called by loading its address into a register and using "SEP register" to do the call. The subroutine is called from two places in the main program. When it has performed its function, the subroutine does a return by doing a SEP back to the register of the main program. This returning SEP instruction is performed just in front of the entry point to the subroutine. This step leaves the subroutine's program counter as it was originally (i.e., pointing to the same location as at initialization). Thus, the initialization need be done only once in this example. The register assignment table is given in Fig. 71.

Register	Function	Comment
R(3)	The "MAIN" program pointer	Arbitrary
R(7)	"Subroutine" entry pointer	Arbitrary

Fig. 71 — Register assignment table for Example 2.

Programming Technique:

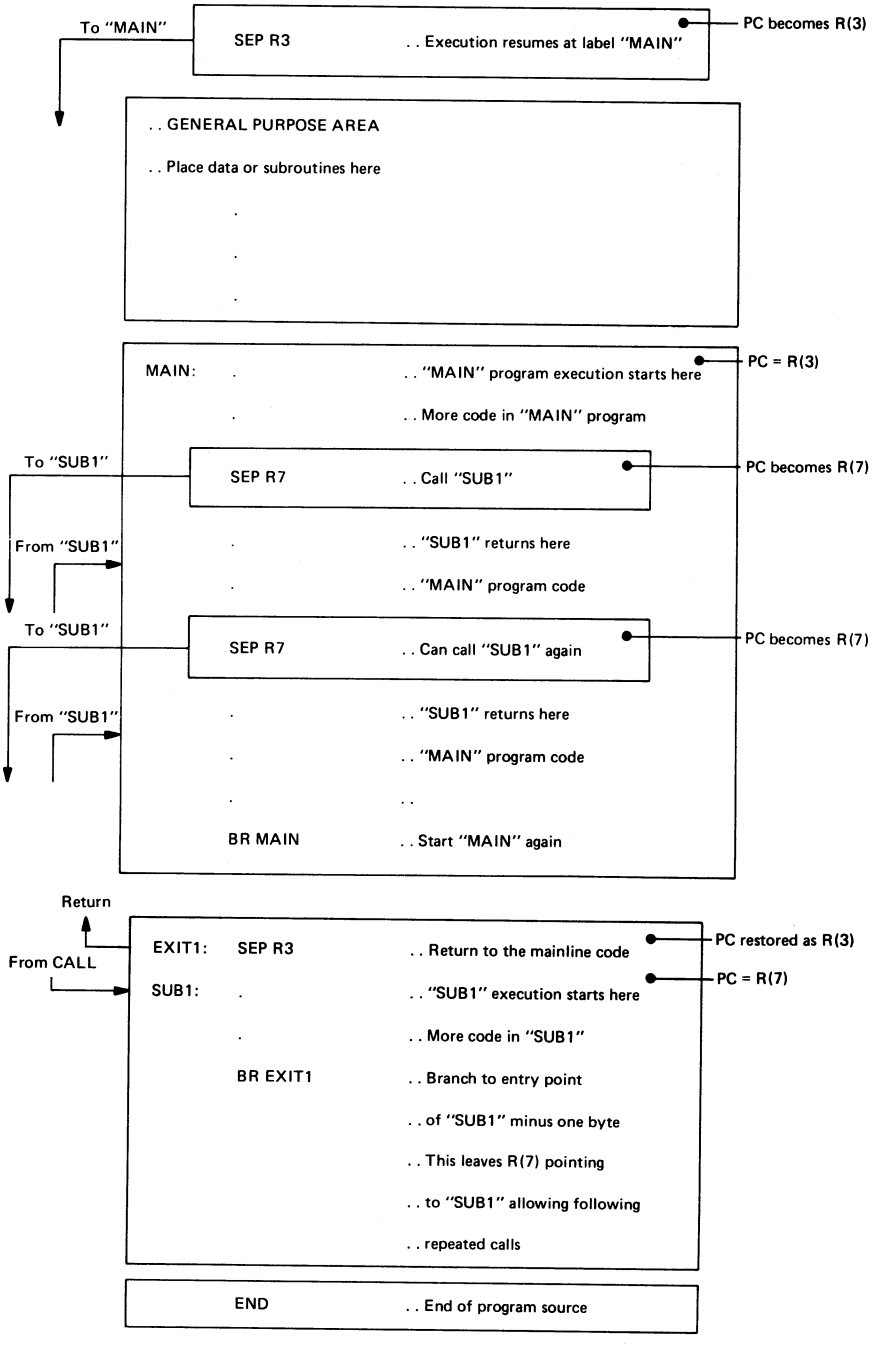
. . Execution starts with R(0) as the program counter.
. . Load the address of "MAIN" program into R(3):

```
LOAD 'R3', 'MAIN'    . . Load A(MAIN) into R(3)        ● PC = R(0)
```

. . Load the address of subroutine "SUB1" into R(7):

```
LOAD 'R7', 'SUB1'    . . Load A(SUB1) into R(7)        ● PC = R(0)
```

. . Change program counter to R(3) to give control to "MAIN" program:



Example 3. **Main Program Calling Two Subroutines.**

This is an example of a source program containing two subroutines (SUB1 and SUB2). These two subroutines are called from several places in the mainline. A separate register is assigned to each subroutine entry point permitting rapid subroutine calls and requiring no reinitialization of pointers before successive calls. The register assignment table is given in Fig. 72.

Register	Function	Comment
R(3)	"MAIN" program counter	Arbitrary
R(7)	"SUB1" pointer	Arbitrary
R(8)	"SUB2" pointer	Arbitrary

Fig. 72 – Register assignment table for Example 3.

Programming Technique:

. . Execution starts with R(0) as the program counter.

. . Load the address of "MAIN" program into R(3):

```
LOAD 'R3', 'MAIN'    . . Load A(MAIN) into R(3)        <-- PC = R(0)
```

. . Load the address of subroutine "SUB1" into R(7):

```
LOAD 'R7', 'SUB1'    . . Load A(SUB1) into R(7)        <-- PC = R(0)
```

. . Load the address of subroutine "SUB2" into R(8):

```
LOAD 'R8', 'SUB2'    . . Load A(SUB2) into R(8)        <-- PC = R(0)
```

. . Change program counter to R(3) to give control to "MAIN" program:

```
To "MAIN"
    SEP R3           . . Execution resumes at label "MAIN"   <-- PC becomes R(3)

    . . GENERAL PURPOSE AREA
    . . Place data or subroutines here
    .
    .
    .

MAIN:  .             . . "MAIN" program execution starts here   <-- PC = R(3)
       .             . . More code in "MAIN" program
To "SUB1"
    SEP R7           . . Call "SUB1"                         <-- PC becomes R(7)
From
"SUB1"
       .             . . "SUB1" returns here
       .             . . More code in "MAIN" program
To "SUB2"
    SEP R8           . . Call "SUB2"                         <-- PC becomes R(8)
From
"SUB2"
       .             . . "SUB2" returns here
       .             . . More code in "MAIN" program
To "SUB1"
    SEP R7           . . Can call "SUB1" again               <-- PC becomes R(7)
From
"SUB1"
       .             . . "SUB1" returns here
       .             . . More code in "MAIN" program
       .             . . Program
    BR MAIN          . . Start "MAIN" again
```

```
EXIT1:  SEP R3          . . Return to the mainline call      <-- PC restored as R(3)
                                                             <-- PC = R(7)
SUB1:   .               . . "SUB1" execution starts here
        .               . . More code in "SUB1"
        .               . .
        BR EXIT1        . . Branch up to before entry
                        . . to restore R(7)

EXIT2:  SEP R3          . . Return to the mainline call      <-- PC restored as R(3)
                                                             <-- PC = R(8)
SUB2:   .               . . "SUB2" execution starts here
        .               . . More code in "SUB2"
        BR EXIT2        . . Branch up to before entry
                        . . to restore R(8)

END                     . . End of program source
```

Fig. 73 is a pictorial representation of Example 3.
The "MAIN" program is running in R(3)
"SUB1" runs in R(7)
"SUB2" runs in R(8)
To call "SUB1", SEP R7
To call "SUB2", SEP R8
To return to the "MAIN" program, SEP R3

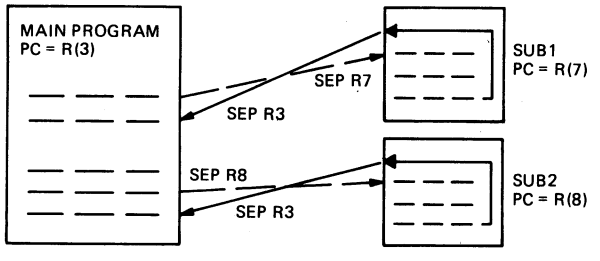


Fig. 73 — Pictorial representation of program counter assignments.

Note that the subroutines "SUB1" and "SUB2" are mutually exclusive in that "SUB1" cannot call "SUB2", or vice versa, because "SUB2" always returns to the caller with a SEP R3. In order for "SUB2" to return to "SUB1", if would have to do a SEP R7. Caution must be exercised to assure that a subroutine "knows" the previous program counter for the return.

MARK Subroutine Technique

In each of the preceding examples, every subroutine had to know the program counter of the calling program in order to do the proper SEP for the return. In large complicated programs where subroutines call other subroutines, a given subroutine may have to return to different program counters at different times. This problem may be overcome with the use of the "MARK Subroutine Technique," a technique which is required only when subroutines are **nested** (call one another), and the order of nesting varies dynamically.

The MARK instruction is used to save the current value of X and P in an area of memory called a **stack**. In a COSMAC based system, this stack consists of any number of locations in RAM pointed to by a 16-bit user-specified register. The pointer contains an address which enables the CPU to find the current "top" of the stack. The stack provides for temporary storage and retrieval of successive bytes of information in a last-in first-out manner. When a byte of information is stored in the stack, it is stored at the address which is specified by the contents of the stack pointer. The stack pointer is then decremented by one, enabling another byte to be "pushed" onto the stack "on top" of the last one stored. Hence, the stack pointer points to a free memory location. Conversely, to retrieve information from the stack, the pointer is incremented by one and the byte is "popped' from the stack by loading it into a register. The top of the stack is then free for other "push" operations. The programmer must make sure that the stack pointer is initialized to an appropriate high address memory location before an instruction that uses the stack is executed.

In the COSMAC architecture, R(2) is the most natural register to use as a stack pointer because of the way interrupts are handled (X is set to 2).

The use of the MARK instructions also provides another benefit. The calling program may pass data (parameters) to the subroutine via "inline" data lists. An inline data list is a block of immediate constant data

supplied by the calling program to the subroutine. This data is adjacent to (inline with) the call statement. In Example 4, SUB1 passes an inline #24 to SUB2.

The MARK Subroutine Technique works as follows:

1. The calling program executes a MARK instruction. This instruction pushes the values of X and P onto the stack pointed to by R(2) and then copies P into X. The current value of X is now the same as the old value of P. P remains the same.
2. The calling program calls the subroutine via a SEP instruction to a register that points to the subroutine. The execution of the subroutine then begins.
3. If the calling program has provided inline parameters to the subroutine, then the subroutine picks up these parameters by executing LDXA instructions. This procedure will load the next parameter to the D register and automatically advance the caller's program counter (R(X)) to the next byte. There should be as many LDXA instructions as there are inline data bytes. Thus, the passing of data is accomplished without the subroutine knowing the old program counter (X designator).
4. After the subroutine performs its task, it returns to the caller by setting X to 2, incrementing R(2) (preparing R(X) for a return), and executing a RET instruction. These instructions will load the contents of memory byte pointed to by R(2) to the X and P registers. Processing continues with the calling program running with its original X and P.
5. Upon return of control from the subroutine to the calling program, the calling program must decrement R(2) to compensate for the increment in the RET instruction.

Example 4. **Main Program Calling Nested Subroutines.**

This example illustrates a program that has two subroutines. Both subroutines are called by the main program. One of the subroutines is also called by the other subroutine. The register assignment table is given in Fig. 74.

Register	Function	Comment
R(2)	Stack pointer	
R(3)	"MAIN" program counter	Arbitrary
R(7)	"SUB1" pointer	Arbitrary
R(8)	"SUB2" pointer	Arbitrary

Fig. 74 — Register assignment table for Example 4.

Programming Technique:

. . Execution starts at location 0000 with R(0) as the program counter.
. . Load the address of "MAIN" program into R(3):

```
LOAD 'R3', 'MAIN'    . . Load A(MAIN) into R(3)        <-- PC = R(0)
```

. . Load the address of subroutine "SUB1" into R(7):

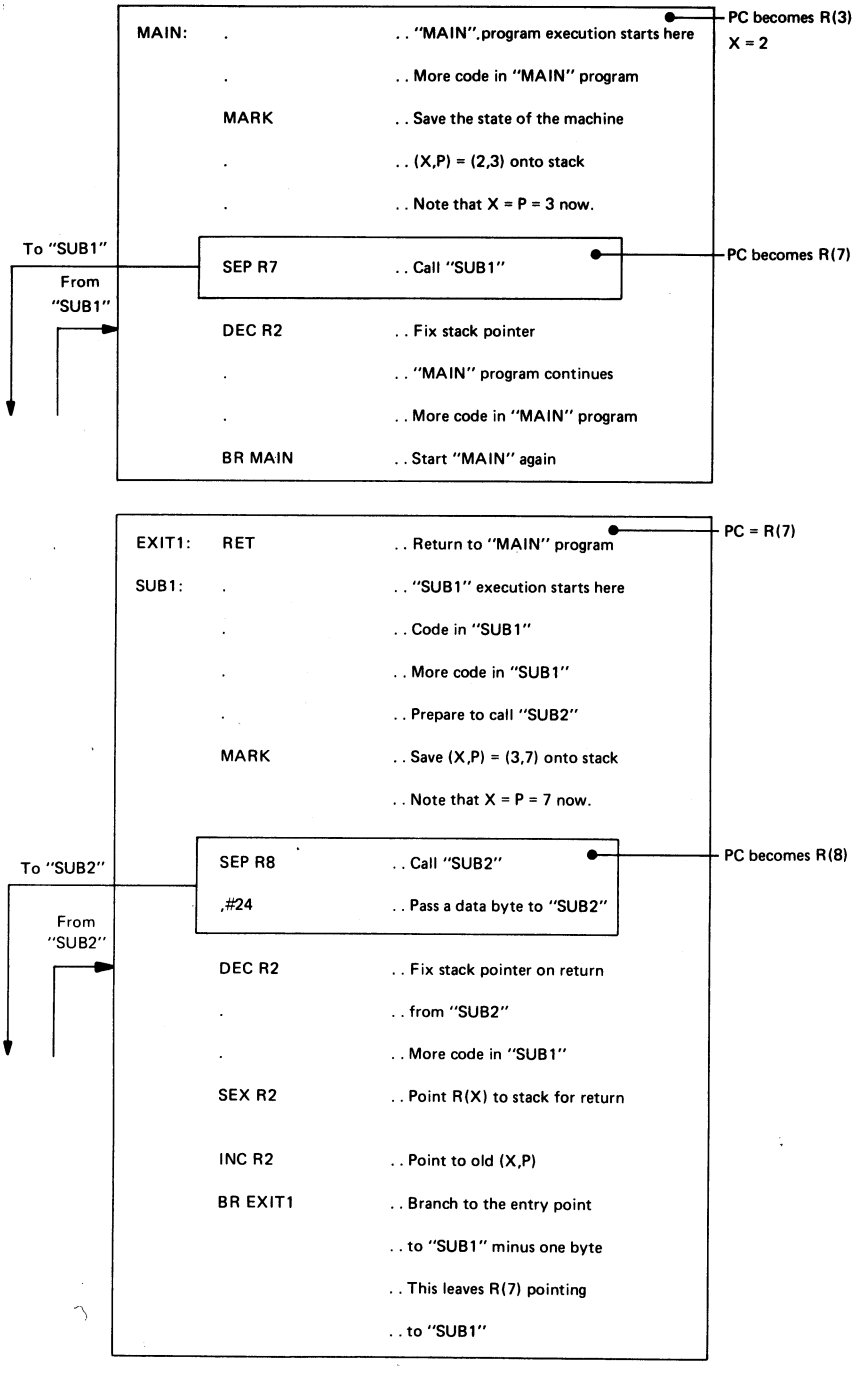
```
LOAD 'R7', 'SUB1'    . . Load A(SUB1) into R(7)        <-- PC = R(0)
```

. . Load the address of subroutine "SUB2" into R(8):

```
LOAD 'R8','SUB2'     . . Load A(SUB2) into R(8)        <-- PC = R(0)
```

. . Set stack pointer R(2) to a RAM area:

```
LOAD 'R2','STACK'    . . Load A(STACK) into R(2)       <-- PC = R(0)
SEP R3               . . Execution resumes at label "MAIN"   <-- PC becomes R(3)
To "MAIN"

. . GENERAL PURPOSE AREA
. . Place data or subroutines here
.
.
.
```


MAIN: . . "MAIN".program execution starts here
PC becomes R(3)
X = 2
. . More code in "MAIN" program
MARK . . Save the state of the machine
. . (X,P) = (2,3) onto stack
. . Note that X = P = 3 now.
To "SUB1"
From "SUB1"
SEP R7 . . Call "SUB1"
PC becomes R(7)
DEC R2 . . Fix stack pointer
. . "MAIN" program continues
. . More code in "MAIN" program
BR MAIN . . Start "MAIN" again
EXIT1: RET . . Return to "MAIN" program
PC = R(7)
SUB1: . . "SUB1" execution starts here
. . Code in "SUB1"
. . More code in "SUB1"
. . Prepare to call "SUB2"
MARK . . Save (X,P) = (3,7) onto stack
. . Note that X = P = 7 now.
To "SUB2"
From "SUB2"
SEP R8 . . Call "SUB2"
PC becomes R(8)
,#24 . . Pass a data byte to "SUB2"
DEC R2 . . Fix stack pointer on return
. . from "SUB2"
. . More code in "SUB1"
SEX R2 . . Point R(X) to stack for return
INC R2 . . Point to old (X,P)
BR EXIT1 . . Branch to the entry point
. . to "SUB1" minus one byte
. . This leaves R(7) pointing
. . to "SUB1"

```
EXIT2:  RET          . . Return to "SUB1"          <-- PC = R(8)
SUB2:                . . "SUB2" execution starts here
        LDXA         . . Pick up inline parameter
                     . . from caller
                     . . More code in "SUB2"
        SEX R2       . . Point R(X) to stack for return
        INC R2       . . Point to old (X,P)
        BR EXIT2     . . Branch to entry point
                     . . to "SUB2" minus one byte
                     . . This leaves R(8) pointing
                     . . to "SUB2"

        END          . . End of program source
```

Standard Call and Return Technique

The Standard Call and Return Technique (SCRT) is the most advanced technique described in this material. It has several advantages over the preceding techniques in that:

1. There is unlimited subroutine nesting capability.
2. There is no confusion over program counter assignments.
3. The passing of parameters to subroutines is well defined.
4. It has a maximum of flexibility in storing working registers.

SCRT is not without its disadvantages, however. They are:

1. It requires additional execution time in calling and returning.
2. It reserves three registers for linkage.

The specific implementation discussed here may also be tailored to suit the preferences of the programmer in that additional registers may be saved or restored.

The SCRT centers around the register assignments given in Fig. 75.

Register R(2) must point to a free memory area for use as a stack. This stack must be large enough to hold two bytes for each level of nesting that might occur plus any additional bytes that the programmer might choose to push onto the stack. Thus, for a program that contains five subroutines and the possibility exists that the main program might call a subroutine which calls another which in turn calls another, then the last subroutine is said to be three levels deep.

Register R(3) is the basic program counter for both the main program and the subroutine. So long as the proper SCRT call and return conventions are maintained, the programmer is assured that R(3) is the program counter.

The SCRT uses two linking subroutines: one when the call operation is to be performed and the other when the return from the subroutine is to be performed. Registers R(4) and R(5) must be initialized once in the program to point to the linking call subroutine and the linking return subroutine, respectively.

Calling a Subroutine

A call to a subroutine is performed by executing a SEP R4 instruction. The two bytes following this SEP instruction must contain the address of the subroutine

Register	Function
R(2)	Stack pointer
R(3)	Program counter
R(4)	Dedicated program counter for call routine
R(5)	Dedicated program counter for return routine
R(6)	Pointer to the return location and arguments passed by the calling program

Fig. 75 — Register assignment table for standard call and return technique (SCRT).

to be called. For example, to call the subroutine labeled MULT, the following assembly language statement could be employed:

```
SEP R4          . . Call multiply subroutine
,A(MULT)        . . Call multiply subroutine
```

For those subroutines that expect a constant to be passed to it as an inline parameter, the following call could be used:

```
SEP R4          . . Call type program
,A(TYPE)        . . Call type program
,T'Text for Typing'
,#00            . . Denote end of string with null
```

Other methods for passing parameters to a subroutine involve the use of a register, a memory area pointed to by a register, a reserved area of memory, DF, or Q. Data may not be passed via the D register, however, because the D register is used by the linking subroutines.

After a SEP R4 instruction is executed, the following events take place:

1. R(4) becomes the program counter.
2. The call subroutine starts running in R(4).
3. The current contents of R(6) is pushed onto the stack.
4. The contents of R(3) is copied into R(6).
5. The address of the subroutine being called is loaded into R(3).
6. A SEP R3 starts execution of the called subroutine.

As a result of the register manipulations, (a) the subroutine being called will run in R(3), (b) register R(6) will point back to the data list of inline parameters provided by the caller or to the return address for the return operation, and (c) because R(6) was saved, the stack will have "grown" by two bytes.

If the subroutine is expecting inline parameters to be passed to it, it should execute as many LDA R6 instructions as there are bytes in the data list. This execution will load the successive data bytes into the D register for use by the subroutine and increment R(6) up to the proper address for a return operation. If the subroutine needs to call other subroutines, it may call by executing a SEP R4, etc. As many subroutines in succession as are required may be called.

Returning from a Subroutine

Once a subroutine has been called and has completed its function, control should be returned to the caller by executing a SEP R5 instruction:

```
SEP R5          . . Return to caller
```

After the SEP R5 instruction is executed, the following events take place:

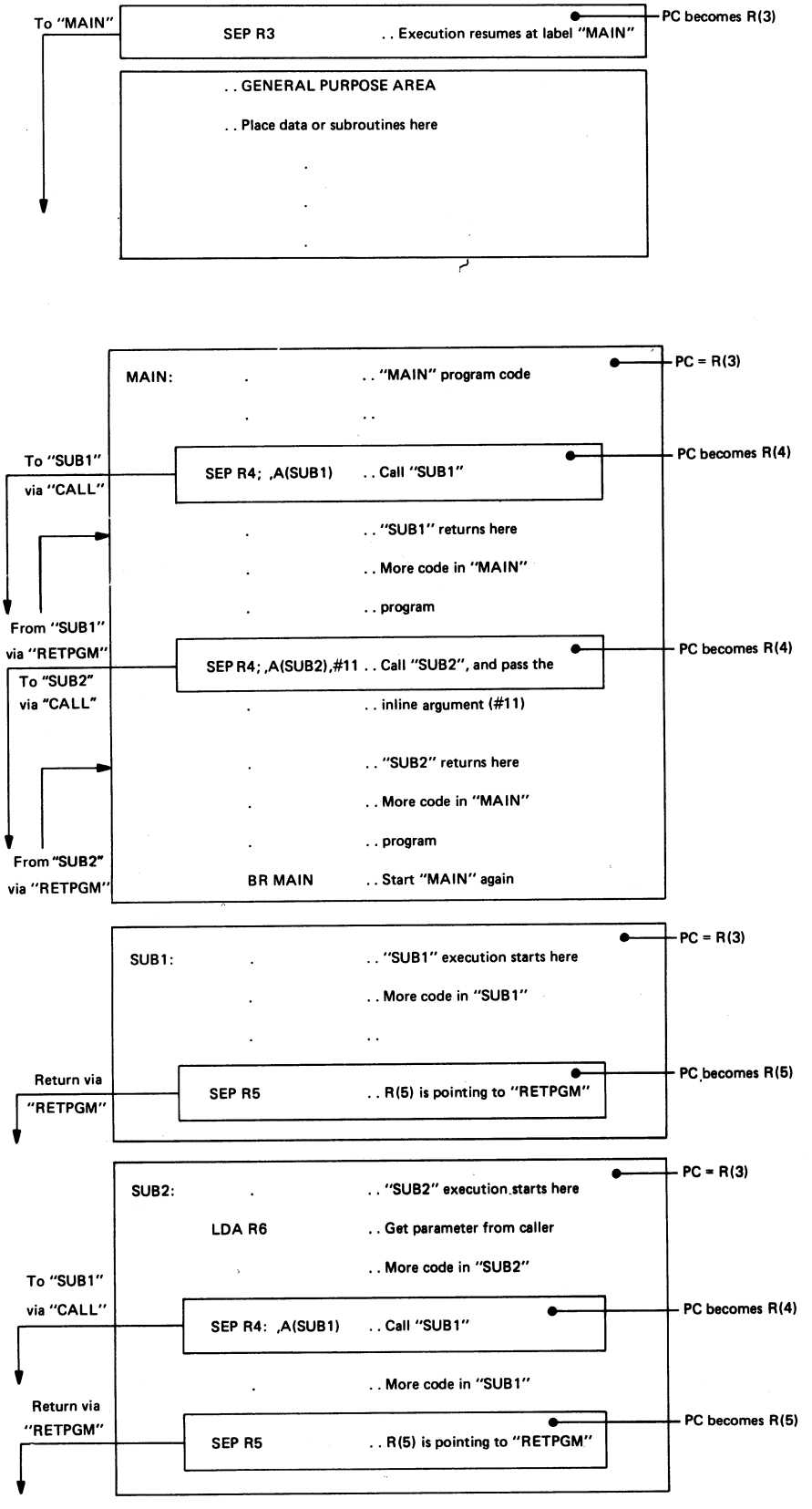
1. R(5) becomes the program counter.
2. The return subroutine starts running.
3. The contents of R(6) is moved to R(3).
4. The saved R(6) contents is reloaded from the stack into R(6).
5. The execution of the calling program is resumed with a SEP R3 from the linking program.

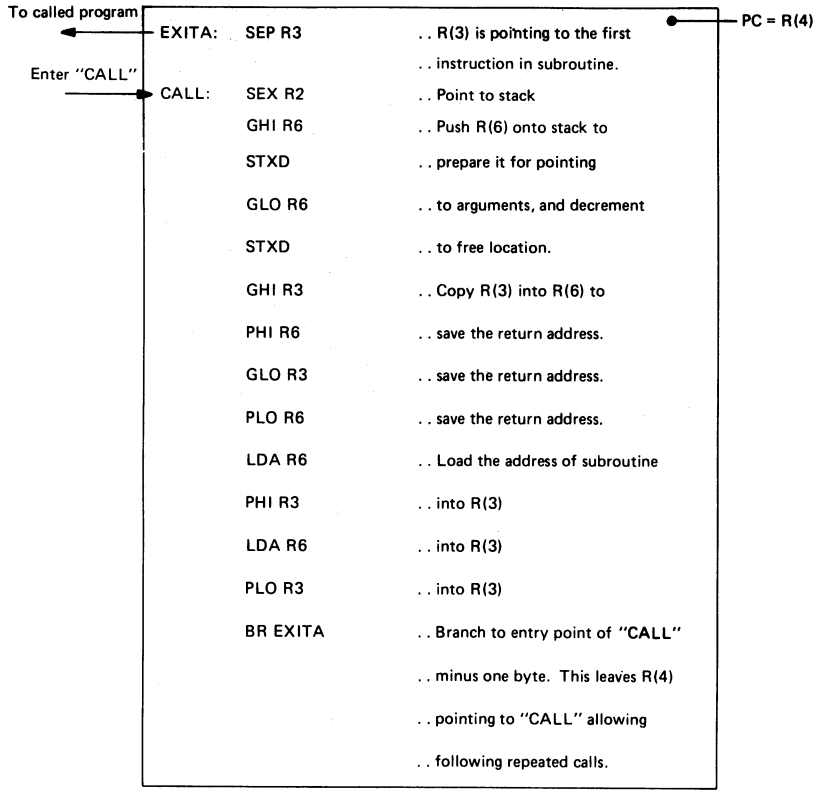
Example 5. **Main Program Calling for Unlimited Subroutine Nesting.**

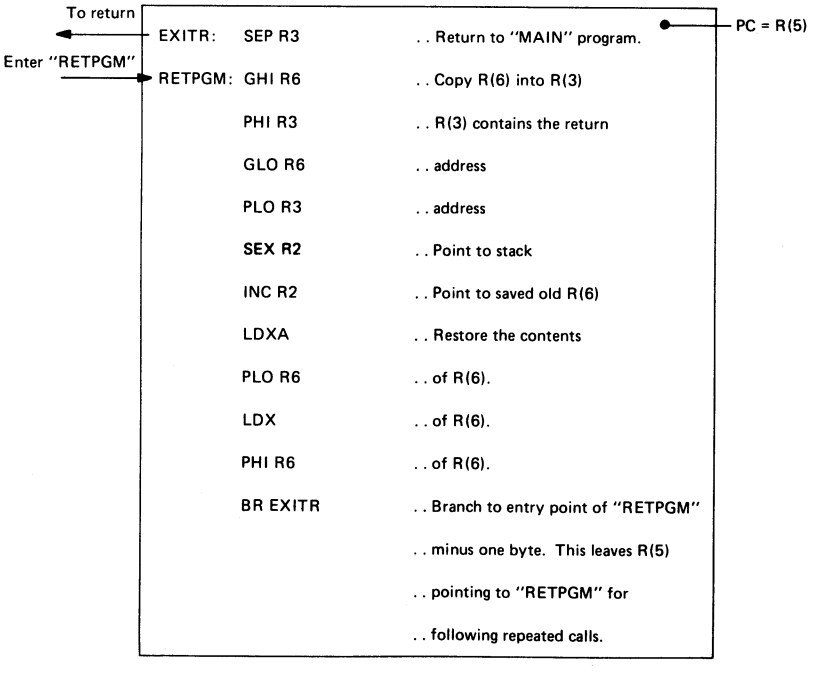
This example illustrates the Standard Call and Return Technique (SCRT) capable of handling an unlimited number of nested subroutines. The register assignments are given in Fig. 75.

Programming Technique:

```
. . Execution starts at location 0000 with R(0) as the program counter.
. . Load the address of "MAIN" program into R(3):

LOAD 'R3','MAIN'    . . Load A(MAIN) into R(3)        PC = R(0)

. . Load the address of the call routine into R(4):

LOAD 'R4','CALL'    . . Load A(CALL) into R(4)        PC = R(0)

. . Load the address of the return routine into R(5):

LOAD 'R5','RETPGM'  . . Load A(RETPGM) into R(5)      PC = R(0)

. . Set stack pointer R(2) to a RAM area:

LOAD 'R2','STACK'   . . Load A(STACK) into R(2)       PC = R(0)
```


To "MAIN"
SEP R3
. . Execution resumes at label "MAIN"
PC becomes R(3)
. . GENERAL PURPOSE AREA
. . Place data or subroutines here
MAIN:
. . "MAIN" program code
PC = R(3)
To "SUB1"
via "CALL"
SEP R4; ,A(SUB1)
. . Call "SUB1"
PC becomes R(4)
. . "SUB1" returns here
. . More code in "MAIN"
. . program
From "SUB1"
via "RETPGM"
To "SUB2"
via "CALL"
SEP R4; ,A(SUB2),#11
. . Call "SUB2", and pass the
. . inline argument (#11)
PC becomes R(4)
. . "SUB2" returns here
. . More code in "MAIN"
. . program
From "SUB2"
via "RETPGM"
BR MAIN
. . Start "MAIN" again
SUB1:
. . "SUB1" execution starts here
PC = R(3)
. . More code in "SUB1"
Return via
"RETPGM"
SEP R5
. . R(5) is pointing to "RETPGM"
PC becomes R(5)
SUB2:
. . "SUB2" execution starts here
PC = R(3)
LDA R6
. . Get parameter from caller
. . More code in "SUB2"
To "SUB1"
via "CALL"
SEP R4: ,A(SUB1)
. . Call "SUB1"
PC becomes R(4)
. . More code in "SUB1"
Return via
"RETPGM"
SEP R5
. . R(5) is pointing to "RETPGM"
PC becomes R(5)





Interrupt Service

The use of the COSMAC interrupt line involves special programming considerations. The user should be aware of the fact that an interrupt may occur between any two instructions in a program. Therefore, the sequence of instructions initiated by the interrupt routine must save the values of any machine registers it shares with the original program and restore these values before resuming execution of the interrupted program.

R(1) must always be initialized to the address of the interrupt service program before an interrupt is allowed. Fig. 76 illustrates a hypothetical interrupt service routine. R(1) is initialized to 0054 before permitting interrupt. R(2) is a **stack pointer**, i.e., it addressed the free topmost byte in a variable-size data storage area. This

stack area grows in size as the pointer moves upward (lower memory addresses), much like a stack of dishes on a table. Also like the dish stack, it shrinks as bytes are removed from the top. In the interrupt service example of Fig. 76, the stack grew by five bytes as X,P,D, DF, and R7 were stored on it, and then decreased to its original size when R7,DF,D, and X,P were restored. Such a stack is sometimes referred to as a "LIFO" (Last-In-First-Out) because the first item removed from the stack is the last one placed on it.

When bytes are to be stored onto the stack by the interrupt routine the pointer R(2) is first decremented to assure that it is pointing to a free space. In the example shown, location 00F0 may have been in use when the interrupt occurred, so the pointer decrements to 00EF to store X,P. When bytes are no longer needed, they are removed from the stack and the pointer is incremented.

The stack in Fig. 76 is used to store the values of X,P, saved on the stack are now restored. In the example of Fig. 76, program execution branches to location M(0053). The RETURN instruction (70) sets IE=1 and restores the original, interrupted X and P register values. The next instruction executed will be the one which would have been executed had no interrupt occurred (unless the interrupt is still present, in which case the whole process is repeated). Note that R(1) is left pointing at M(0054) and R(2) is pointing at M(00F0), as they were before the interrupt.

When IE is reset to 0 by the S3 interrupt response cycle, further interrupts are inhibited regardless of the $\overline{\text{INTERRUPT}}$ line state. This setting prevents a second interrupt response from occurring while an interrupt is being processed. The instruction (70) that restores original program execution at the end of the interrupt routine sets IE=1 so that subsequent interrupts are permitted.

Sometimes the programmer needs to control IE

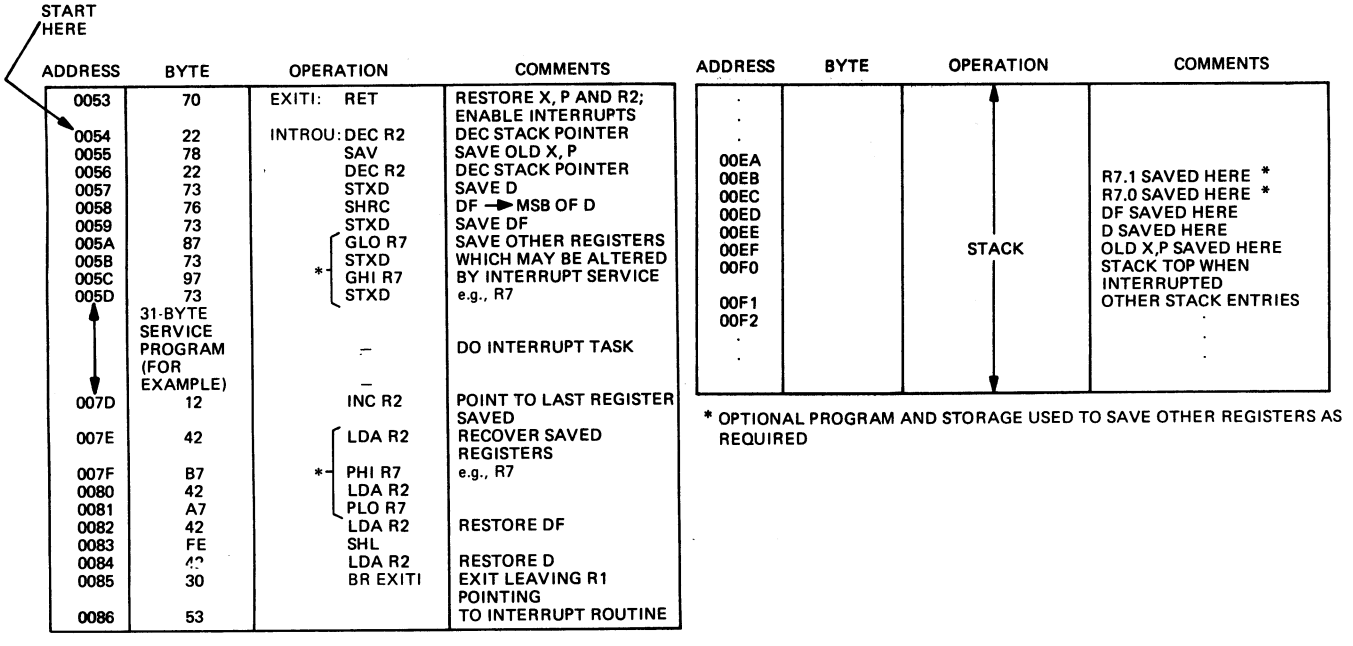


Fig. 76 — Interrupt service routine.

D, and DF associated with the interrupted program. If the interrupting program will modify any other registers, such as R7 in the example given in Fig. 76, their contents must also be saved.

After these "housekeeping" steps have been completed, the "real work" requested by the interrupt signal can be performed. This work may involve such tasks as transferring I/O bytes, initializing the DMA pointer R(0), checking the status of peripheral devices, incrementing or decrementing an internal timer/counter register, branching to an emergency power-shut-down sequence, etc.

Upon completion of the "real work", return housekeeping must be performed. The contents of registers directly. For example, he may want to permit new interrupts to interrupt the servicing of old interrupts. Or, he may want to shut off interrupts during a critical part of the main program.

The RETURN and DISABLE instructions can be used to set or reset IE without changing P and performing a branch. A convenient method is to set X equal to the current P value and then perform the RETURN (70) or DISABLE (71) instruction, using the desired X,P for the immediate byte. For example, if IE=0, X=5, and P=3, the sequence

```
E3   SEX R3    . . Set X = 3
70   RET       . . Return X to 5, P to 3, 1 → IE,
```

```
                    R(3) + 1.
53    ,#53          . . Immediate byte
```

would have no effect other than setting the interrupt enable IE. A similar sequence with a 71 instruction can be used to disable interrupts during a critical instruction sequence.

Interpretive Techniques

An interpretive system offers the advantages of a "higher-level" language without the disadvantages of complex translation programs. The idea is to define a set of **pseudo instructions** which are more powerful than basic CPU instructions and, consequently, easier to program with. Each pseudo instruction is implemented by a corresponding subroutine. In the simplest interpretive system, each subroutine ends with a mechanism which passes control on to the next subroutine (i.e., pseudo instruction) to be executed. The sequence of pseudo instruction is defined by a **pseudo program**, analogous to the way a program defines a sequence of instructions. A **pseudo program counter** is a register which is generally pointing at the next pseudo instruction to be executed. Just as with a real program counter, pseudo branch instructions may affect the normal sequencing of the pseudo program counter.

Specifically, let PPC be the pseudo program counter—one of the COSMAC registers. Let PC be the normal program counter. Suppose that all subroutines begin and end on the same page in memory. They may branch to other pages, but they eventually come back. Then, a pseudo program is nothing more than a series of addresses—the low-byte address of each successive subroutine to be executed. Each subroutine ends with the same two instructions:

```
LDA PPC       . . fetch next address
PLO PC        . . into PC low.
```

These instructions give control over to the next subroutine.

Just as with subroutine calls (which they closely resemble), pseudo instructions may be followed by arguments or argument addresses. For example, a (long) branch pseudo subroutine would be:

```
LBR:  LDA PPC       . . Put first address byte
      STR STACK     . . into the stack.
      LDA PPC       . . Put second byte
      PLO PPC       . . into PPC low.
      LDN STACK     . . Put first byte
      PHI PPC       . . into PPC high.
      LDA PPC       . . Go
      PLO PC        . . to next pseudo instruction.
```

A typical set of pseudo instructions might include multiple-precision or floating-point arithmetic functions, I/O handling instructions, multi-way branches on arithmetic comparisons, subroutine linkage routines, and a mechanism to drop into standard COSMAC instructions whenever necessary. The programmer should choose and program a set of instructions suitable to his specific application.

More details and a discussion of alternative interpretive systems may be found in COSMAC Application Notes to be provided.

Interfacing and System Operations

This section describes some circuits and suggested techniques for interfacing the COSMAC Microprocessor CDP1802 with external memories, control circuits, and input/output devices in various system configurations. Reference to the section on **Timing Diagrams** will be helpful in reading this material.

Memory Interface and Timing

The use of memory interface lines is best described by specific examples. Fig. 77 shows the direct interconnection of a static 32-byte RAM to the CPU. No external parts are required. The same simplicity in interfacing is evident in Fig. 78 for a 256-byte static RAM. For memories requiring more than eight addressing bits, Fig. 79 illustrates the interconnections of a static 4096-byte RAM to the CPU. The 4096-byte read-write memory comprises thirty-two 256-word by 4-bit CDP1822 RAM's. These static RAM's, requiring only a single power supply, are easy to use. Twelve memory address bits are required to select 1 out of 4096 memory byte locations. The high-order byte (A.1) of a 16-bit memory address appears on

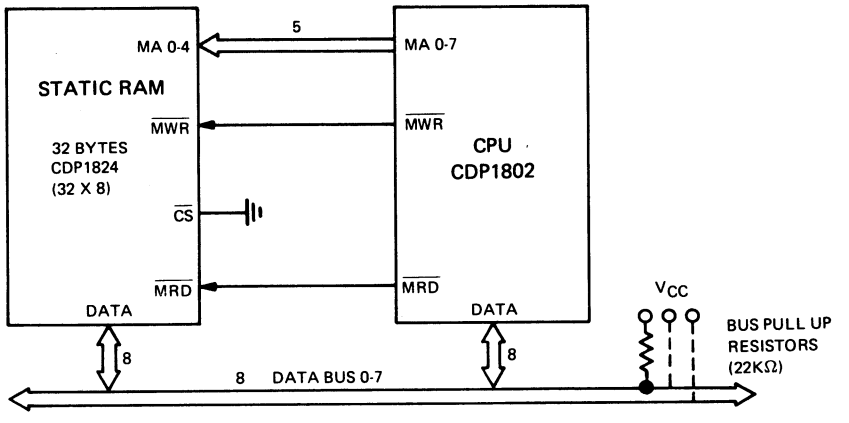


Fig. 77 – Interface for the CDP1824 static RAM to the CDP1802 microprocessor in a 32-byte RAM system.

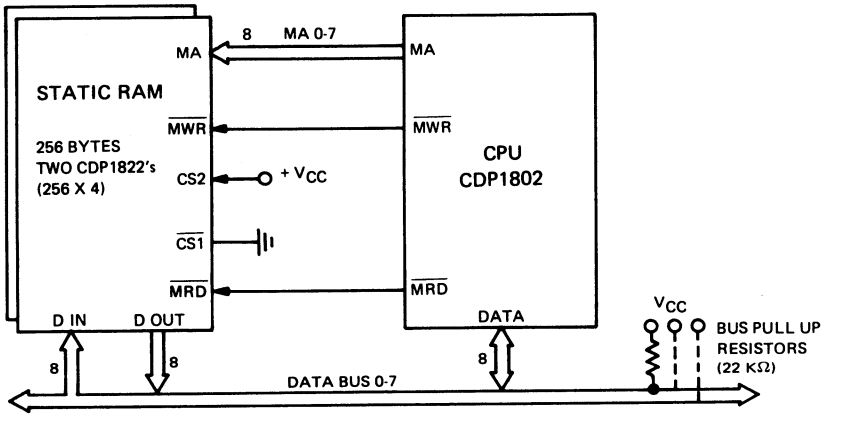


Fig. 78 – Interface for the CDP1822 static RAM to the CDP1802 microprocessor in a 256-byte RAM system.

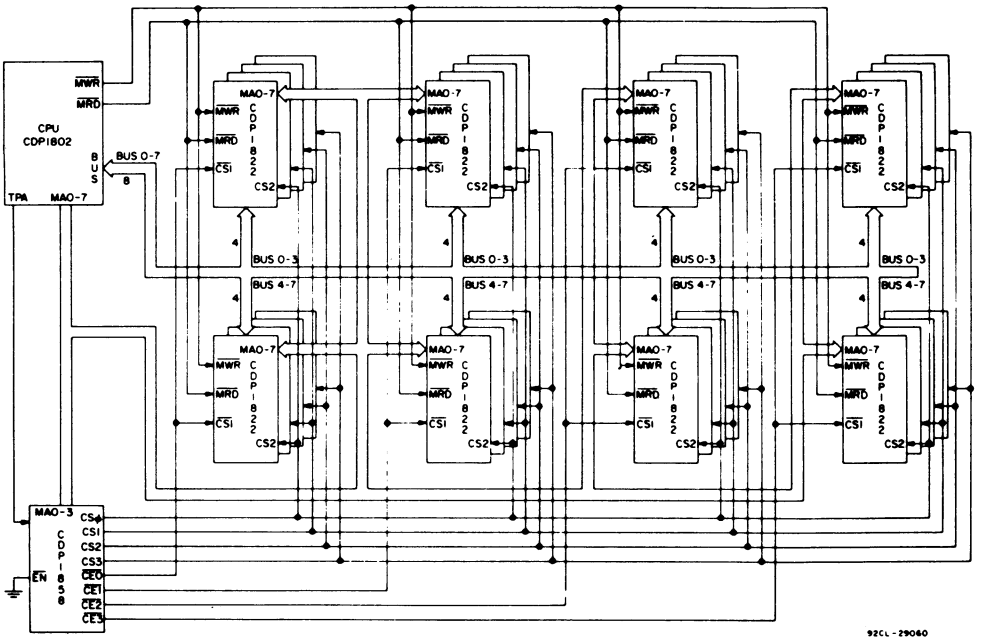


Fig. 79 — Interface for the CDP1822 static RAM to the CDP1802 microprocessor in a 4096-byte RAM system.

the memory address lines MA 0-7 first. The four least significant bits of A.1 are strobed into the latch and decoder circuit CDP1858 by timing pulse TPA. Fig. 80 shows the memory read and write timing. For a more detailed timing diagram including set-up and settling time delays, refer to the data bulletin for the CDP1802.

The low-order byte (A.0) of a 16-bit COSMAC memory address appears on the MA 0-7 lines after the high-order bits have been strobed into the address latch. Latching all eight A.1 bits would permit memory expansion to 65,536 bytes. The MA 0-7 lines may require buffer circuits in large systems to reduce the load on them to achieve high speed.

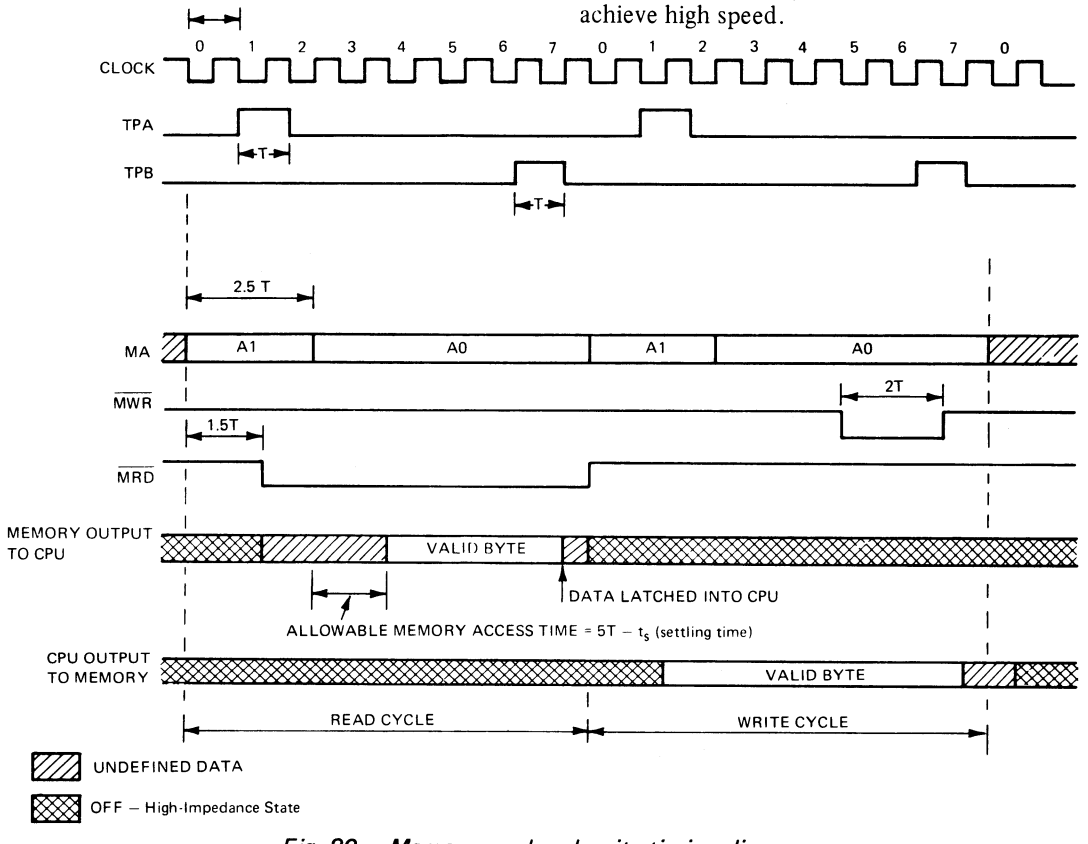


Fig. 80 — Memory read and write timing diagram.

The state of the $\overline{MWR}$ and $\overline{MRD}$ lines determines whether a byte is to be read from the addressed memory location, written into it, or neither operation performed. Fig. 81 tabulates the operation of the memory control lines. Note that the $\overline{MRD}$ and $\overline{MWR}$ lines are active low. The CPU controls the destination of the memory output

CPU LINES	MEMORY READ	MEMORY WRITE	NON-MEMORY OPERATION
$\overline{MRD}$	L	H	H
$\overline{MWR}$	H	L	H

Fig. 81 – Operation of the memory control lines.

byte when it appears on the data bus. The byte may be strobed into an internal CPU register or into an external I/O register. During a WRITE cycle, the memory output is in a high-impedance state. The CPU or I/O circuits can then place a byte to be stored in memory on the bus. A negative-going $\overline{MWR}$ pulse will cause the data byte to be written into the addressed memory location. Eight bus pull-up resistors should be provided to place the bus in a known state when it is not being driven.

Figs. 82 and 83 show interfacing for the CDP1802 with various other memories. Other industry standard RAM's and ROM's are readily accommodated by the CDP1802's general-purpose interface lines. **Access time**, however, must be consistent with clock frequency. The data bulletin for the CDP1802 gives curves relating access time, clock frequency, and instruction cycle. For example, a 4-MHz clock will require a memory having a maximum access time of 900 nanoseconds. The time required by the CPU and internal gating is also specified on the data sheet.

If a memory does not have a 3-state high-impedance output, $\overline{MRD}$ is useful for driving memory-bus separator gates; otherwise, it is used to control 3-state outputs from the addressed memory. A low on $\overline{MRD}$ indicates a read cycle; the low $\overline{MRD}$ line enables the memory-output-bus gates during the read cycle. For various memory systems, the $\overline{MRD}$ signal and the $\overline{MWR}$ pulse polarity and width may require modification by external circuitry.

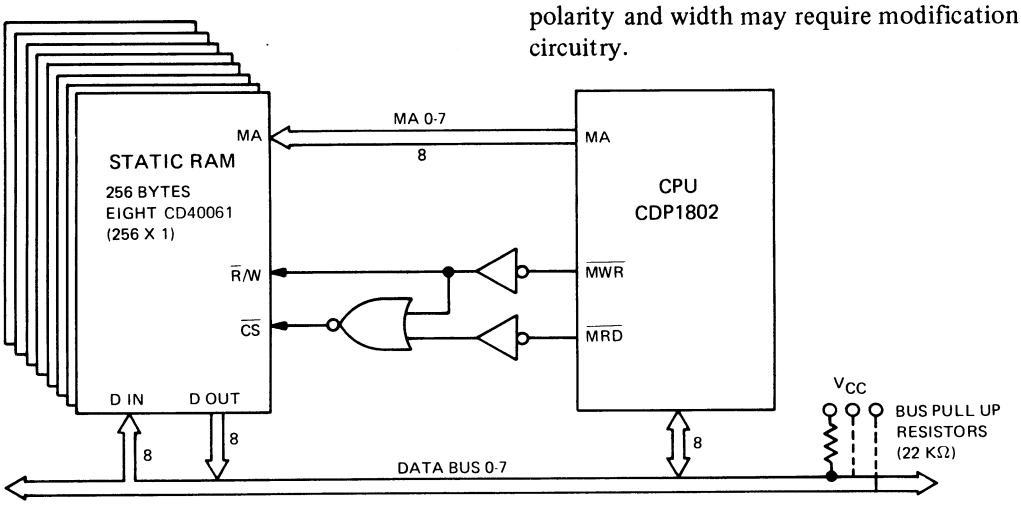


Fig. 82 – Interface for the CD40061 static RAM to the CDP1802 microprocessor in a 256-byte RAM system.

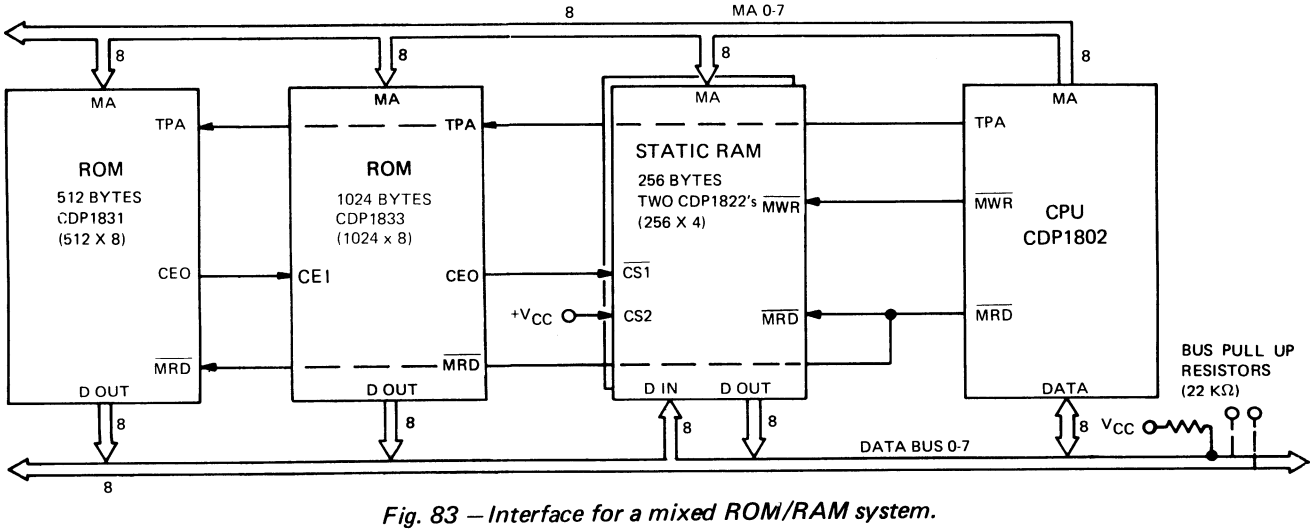


Fig. 83 – Interface for a mixed ROM/RAM system.

Segments of ROM's can be attached in the same manner as RAM's but with the write controls omitted. The CDP1831 and CDP1833 ROM's are especially easy to use because address latching is provided on chip to latch the 8 most significant bits of a 16-bit address. The on-chip decoder is mask programmable which enables placement of a 512-byte memory block anywhere within 65 kilobytes of memory space. Note that the chip enable output signal goes high when the device is selected. It is intended as a chip-select control for small (up to 256-byte) RAM systems.

Dynamic RAM's can be used with appropriate refresh circuits. Because the CDP1802 circuitry is static, the clock may be stopped and restarted for asynchronous memory operation if required, or the $\overline{WAIT}$ input may be used to signal a Data Ready condition. For additional information on this subject, refer to the following subsection on "Control Interface."

Control Interface

The COSMAC Microprocessor CDP1802 has an internal oscillator that works with a crystal connected between the CLOCK and $\overline{XTAL}$ terminals. If desired, however, an external oscillator may be used and fed into the CLOCK input. If an external oscillator is used, no connection is required at the $\overline{XTAL}$ terminal. (Note: care must be taken not to load the $\overline{XTAL}$.) Any type of single-phase clock may be used so long as the rise and fall times of the clock pulse are less than 15 microseconds. Each machine cycle consists of eight clock pulses, and each instruction requires two or three machine cycles. Thus, with a 6.4-MHz clock frequency, a machine cycle of 1.25 microseconds could be achieved, and instructions would be executed in 2.5 to 3.75 microseconds depending on the instruction.

During normal operation, the $\overline{CLEAR}$ and $\overline{WAIT}$ lines are both held high. A low level on the $\overline{CLEAR}$ line will put the machine into the reset mode with I, N, X, P, Q, Data Bus = 0, and IE = 1. Actually, X, P, and R(0) are reset during a special S1 cycle (not available to the programmer) immediately following transition from the reset mode to any of the other modes (load, run, or pause). The clock must be running to effect this cycle.

If the $\overline{CLEAR}$ and $\overline{WAIT}$ lines are both held low, the machine enters the load mode. This mode allows input bytes to be sequentially loaded into memory beginning at M(0000). Input bytes can be supplied from a keyboard, tape reader, etc., by way of the DMA facility. This feature permits direct program loading without the use of external "bootstrap" programs in ROM's.

If the $\overline{WAIT}$ line is brought low (with $\overline{CLEAR}$ high), the CPU stops operation cleanly on the next negative-going transition of the clock (Pause mode). Output signals are held at their values indefinitely. This state is useful for several purposes. Using the $\overline{WAIT}$ line, the CPU can be easily single-stepped for debugging purposes or, if stopped early in the machine cycle, the CPU can be held off the data bus to allow for multiprocessor systems, etc. Also, the $\overline{WAIT}$ line can be used as a data-ready signal from a slow memory or peripheral, or signals TPA and TPB can be stretched. When the $\overline{WAIT}$ line is returned high, the machine resumes running on the next negative-going transition of the clock input. The $\overline{WAIT}$ signal does not inhibit the on-chip crystal oscillator. DMA's and Interrupts are not acknowledged in the Pause mode.

Fig. 84 shows one circuit using standard devices from the CD4000 series for controlling the run and load

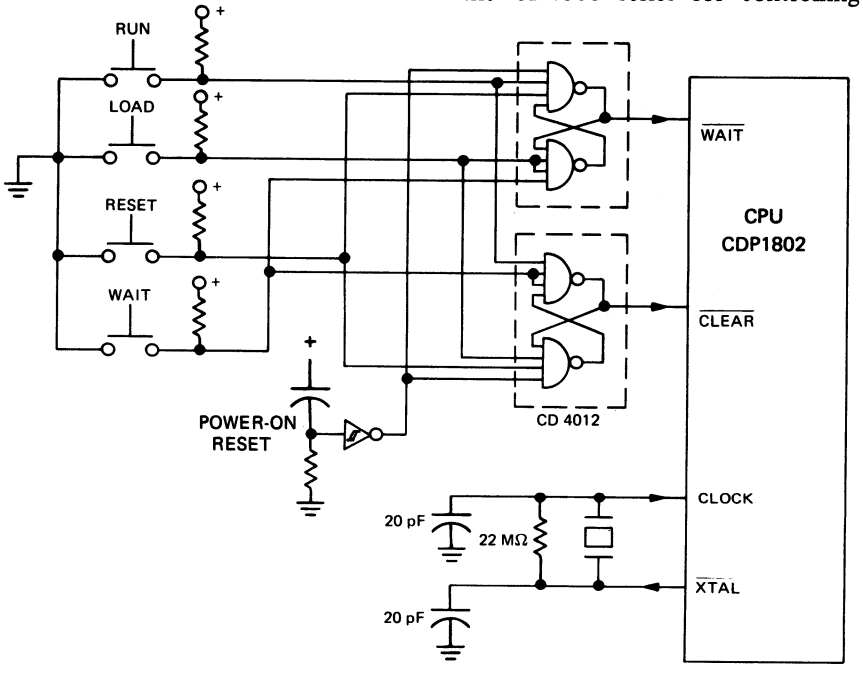


Fig. 84 – Simple control interface for CDP1802 microprocessor.

modes of the CDP1802. Note the power-on reset feature. For design details, refer to ICAN-6581 "Power-On Reset/Run Circuits for the RCA CDP1802 COSMAC Microprocessor". To load and start a program the sequence of operations would be as follows: First, depress the reset and then the load buttons. The CPU is now ready to load by means of the DMA channel. When loading is completed, depressing the reset and then the run buttons will start program execution at M(0000) with R(0) as the program counter (after one machine cycle). If a DMA request is present when the run switch is turned on, the machine will go into the DMA state immediately with R(0) as the program counter. The user should therefore inhibit DMA externally until the program has changed to a program counter different from R(0). Interrupts, however, are disabled until the first instruction or DMA request is executed. This delay allows the programmer to place instruction 71 and 00 in the first two memory bytes to inhibit interrupts until he is ready for them. The combined effect of the two bytes is to set IE = 0. Interrupts must not occur, however, when the machine is in the load mode because they will force the machine into an anomalous running state. Fig. 85 shows the sequence of events and states involved in loading a program via DMA-IN in the load mode and its subsequent execution.

Another circuit that can be used for single-stepping the microprocessor (one machine cycle per switch depression) is shown in Fig. 86. This capability is often useful as a debugging aid.

Fig. 87 provides a summary of the modes discussed, the control levels, and the characteristic features of these modes. It is evident that the run mode can be entered from either the reset or the pause mode.

I/O Interface

The three basic ways in which the CPU can communicate with I/O devices are **programmed I/O, interrupt I/O,** and **Direct Memory Access (DMA).** In the programmed I/O mode, all data transfer is controlled and timed by the program. In the interrupt I/O mode, the CPU responds to an I/O generated signal. In the

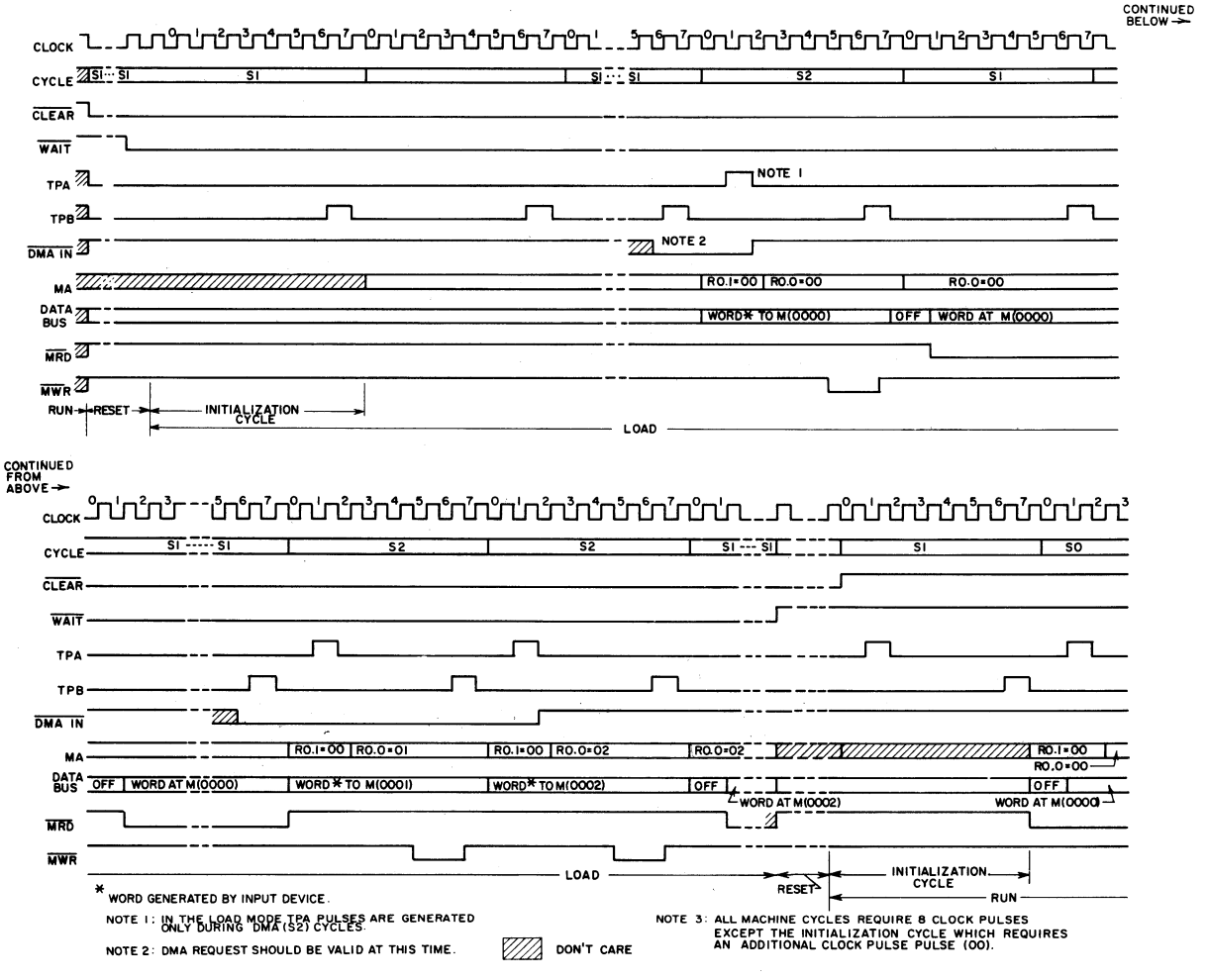


Fig. 85 — Timing diagram for run and load sequences.

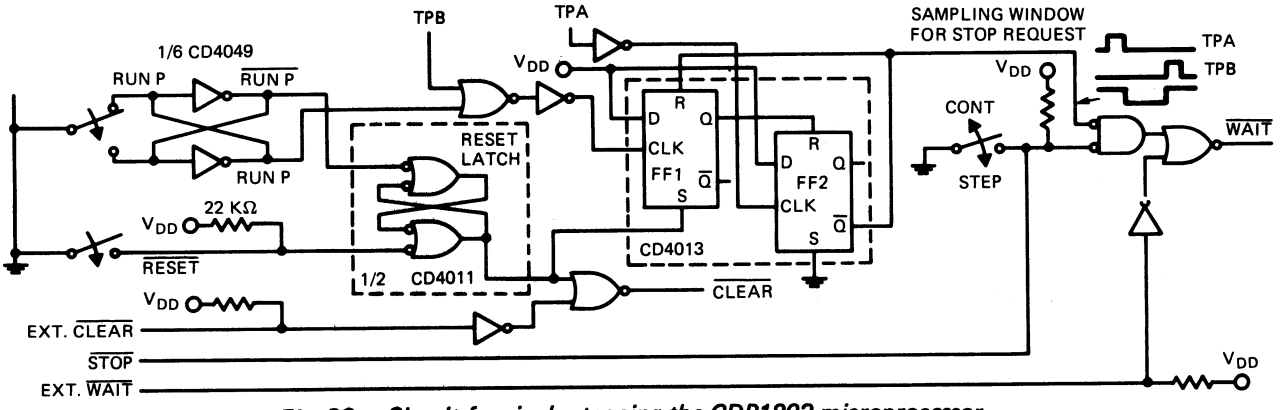


Fig. 86 – Circuit for single-stepping the CDP1802 microprocessor.

MODE	CLEAR	WAIT	OPERATION
RESET	0	1	I, N, X, P = 0, R(0) = 0, Q = 0, BUS = 0, IE = 1; TPA and TPB are suppressed; CPU in S1.
RUN	1	1	CPU starts running one machine cycle after CLEAR is released. Execution starts at M(0000), or an S2 cycle follows if DMA was asserted. Internal sampling of interrupt is inhibited during initialization cycle.
RESET	0	1	As above.
LOAD	0	0	CPU in IDLE. An I/O device can load memory without "bootstrap" loader.
PAUSE	1	0	Clock: ⎍ stops internal operation. CPU outputs held indefinitely. Permits stretching of machine cycle to match slow devices or memory cycles. DMA and INTERRUPTS not acknowledged.
RUN	1	1	Clock: ⎍ Resume operation

Fig. 87 – Truth table for mode control of CDP1802 microprocessor.

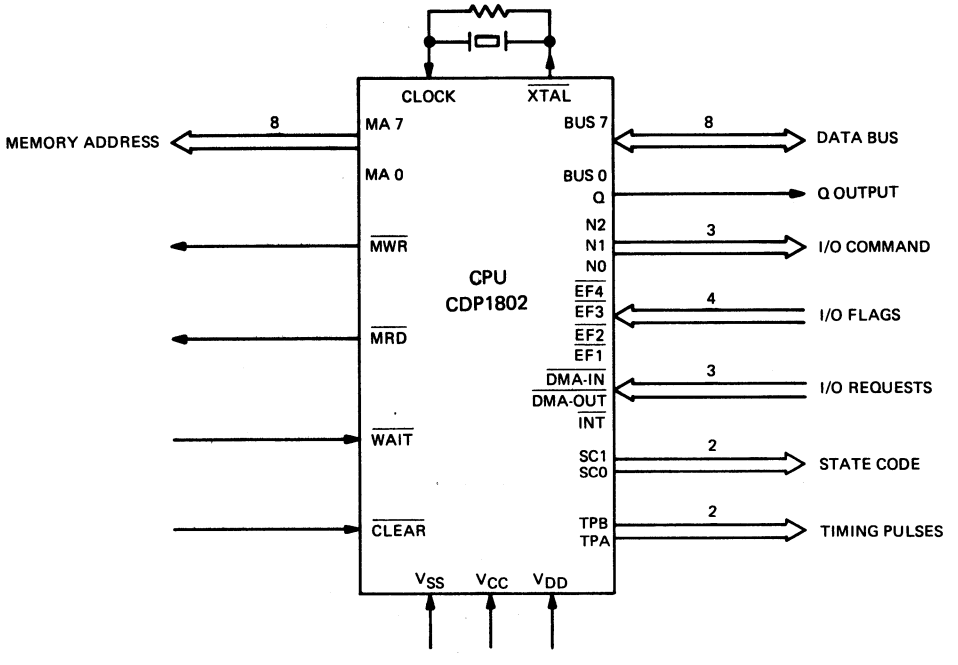


Fig. 88 – Summary of interface lines provided by CDP1802 microprocessor.

DMA mode, a direct high-speed data channel is established between memory and I/O device. The I/O device "steals" execution cycles from the CPU and transfer data during these time slots.

Fig. 88 gives a summary of the interface lines provided by the CDP1802 microprocessor. The large number of dedicated lines available offers both economy and flexibility in I/O system designs.

The following paragraphs indicate a few ways in which I/O data transfer can be accomplished under the three basic modes of operation. Throughout these examples, IC's from the CDP1800 and CD4000 series of standard parts are used. Devices from the 1800 series belong to a growing family of dedicated parts designed specifically to interface with each other and with the CDP1802 Microprocessor for optimum system design. A broad choice of standard parts from the 4000 series is also available for flexible and inexpensive system operations. For detailed information on these devices, the reader should refer to the latest RCA Integrated Circuits DATABOOK.

Programmed I/O — Direct Selection of I/O Devices

One input port only. When I = 6 and N = 9, A, B, C, D, E, or F, a byte on the data bus is written into the D register and the memory location addressed by R(X).

The simplest form of input to the microprocessor utilizes one of the four external flag lines: $\overline{EF1}$, $\overline{EF2}$, $\overline{EF3}$, or $\overline{EF4}$. A low on a flag line places it in its true state. The short branch instructions 34, 35, 36, 37, 3C, 3D, 3E, and 3F allow programs to determine the states of these flag lines.

Fig. 89 illustrates one method of using a flag line ($\overline{EF1}$ in this case) to signal the CPU and initiate a byte transfer into memory and D register. In this circuit, turning the switch on sets $\overline{EF1}$ low and turning it off sets $\overline{EF1}$ high. The flip-flop eliminates switch bounce. Assume first that the switch is off and, therefore, $\overline{EF1}$ = 1. The short branch instruction 3C will test the status of the $\overline{EF1}$ flag. The 3C instruction executes a short branch if EF1 = 0 (i.e., $\overline{EF1}$ = 1), in this case to the branch address XX0A. So long as the switch is off, the program will continue to test the $\overline{EF1}$ flag and execute a branch to XX0A during every instruction cycle. Assume next that the switch is activated so that $\overline{EF1}$ becomes true (i.e., $\overline{EF1}$ = 0). Execution of 3C now requires that the next instruction EA be executed. This instruction sets up R(A) as a data pointer and, in this example, it has been preloaded with address XX1A. Instruction 69 is an input command and loads the byte on the bus into D and the memory location addressed by R(A), (XX1A).

The switch of Fig. 89 might be replaced by a Teletype* output relay. The opening and closing of this relay contact represent the bit-serial teletype character code. A COSMAC program could interpret the sequential states of the $\overline{EF1}$ line to provide an extremely simple bit-serial interface. (The Terminal Interface card and the Utility Program described in the **Operator Manual for the RCA COSMAC Development System II CDP18S005**, MPM-216, give a practical illustration.)

Fig. 90 also shows how input bytes can be read into memory in conjunction with a flag line, but the byte is now entered under CPU control. In this example, the user's strobe asserts one of the $\overline{EF}$ flags which is being monitored by a branch instruction in the program. A byte is latched into the register in a high to low transition of the strobe pulse and generates a service request. When a low is detected on $\overline{EF1}$, the program branches to an input instruction 69 (I = 6 and N = 9). During execution of 69, the three N bits available at the interface are valid. The N0 line, which was low, is active high during the execute cycle. When the CPU responds with an input instruction and the N0 line goes high, the input byte is enabled onto the data bus.

During this machine cycle, the CPU generates a low $\overline{MWR}$ pulse which writes the valid byte on the bus into memory. For further details on input instruction timing, refer to the section on **Timing Diagrams**. Note that the $\overline{EF1}$ line is forced high (the service request is reset) at the end of the valid N0 bit to assure that only one byte is entered per strobe pulse.

The input byte might be the byte output of a paper-tape reader, keyboard, or other input device. The input-

*Registered trademark, Teletype Corporation.

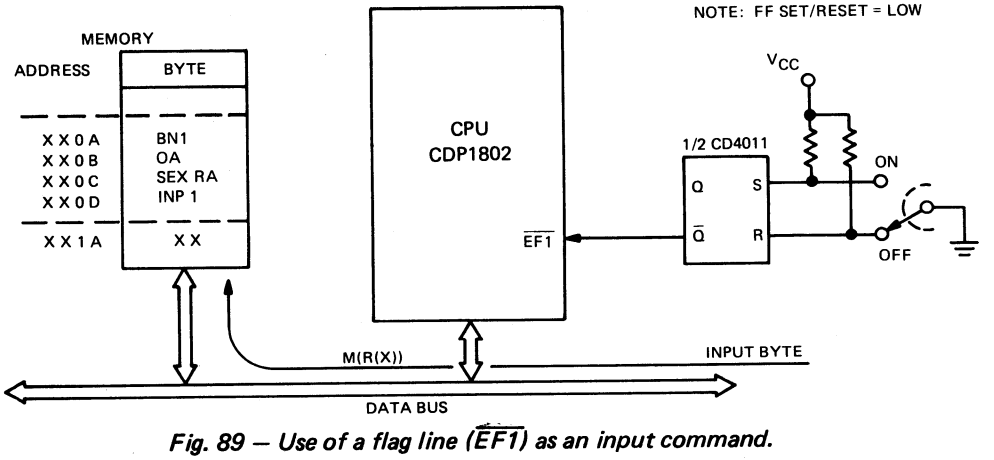


Fig. 89 — Use of a flag line ($\overline{EF1}$) as an input command.

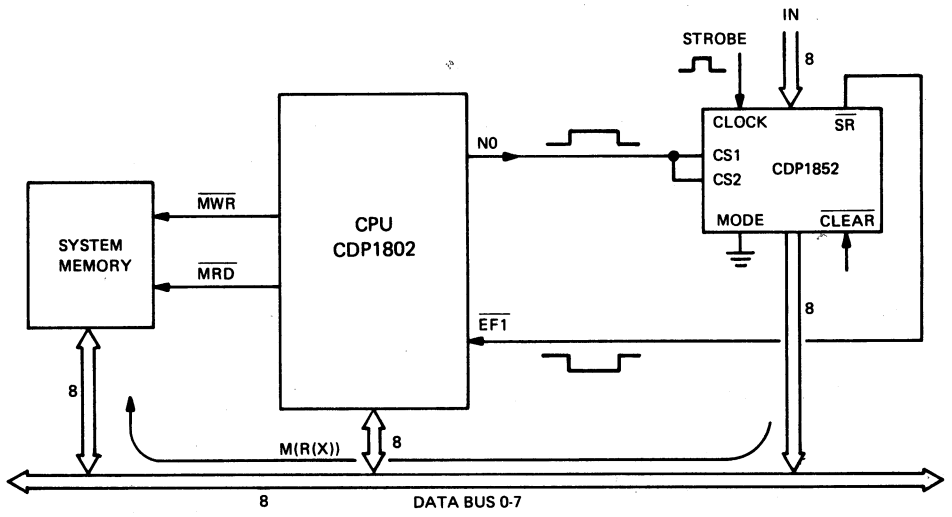


Fig. 90 – Direct selection of I/O devices – one input port.

byte-transfer rate should be consistent with the speed the program samples the flag line and executes input-byte-transfer instructions.

One output port only. When I = 6 and N = 1, 2, 3, 4, 5, 6, or 7, the memory byte addressed by R(X) is placed on the data bus. The three N bit lines are valid during the execution cycle and indicate that an I/O operation is performed. The M(R(X)) byte appears on the data bus before the timing pulse TPB occurs and remains on the bus until after the TPB line returns to its low state.

Fig. 91 shows simple logic for transferring a byte from memory to an output register under program control. If a 61 instruction is executed, the N0 line becomes high during the execute cycle and can be used with the timing pulse TPB to strobe a valid data byte into the output register. The user is then free to enable the output of the register. For more information on output instruction timing, refer to the section on **Timing Diagrams**.

Fig. 92 shows how an output instruction, in this case 61, might be used to set a byte into a two-digit output LED display device. During the execution cycle of instruction 61, when the N0 bit is valid, TPB will strobe valid data into the two BCD-to-7-segment latch decoder drivers.

A COSMAC program can be written to simulate a free-running two-digit decimal counter. Each two-digit count can be placed in the output display of Fig. 92. The switch in Fig. 89 can be used to start and stop the counter. If the switch is in the "ON" position, counting proceeds (00-99). When the switch is turned off, counting stops and the current value of the count is displayed. Turning the switch "ON" again will re-initiate counting, starting at the value displayed. A portion of a possible "counter program" is shown in Fig. 93. In this example, the logic in Fig. 92 must be modified with the $\overline{MRD}$ line to distinguish between input and output instructions, as discussed in the material following.

One input and one output port. Fig. 94 shows how the logic in Figs. 90 and 91 can be combined to provide byte transfers in either direction. The level of the $\overline{MRD}$

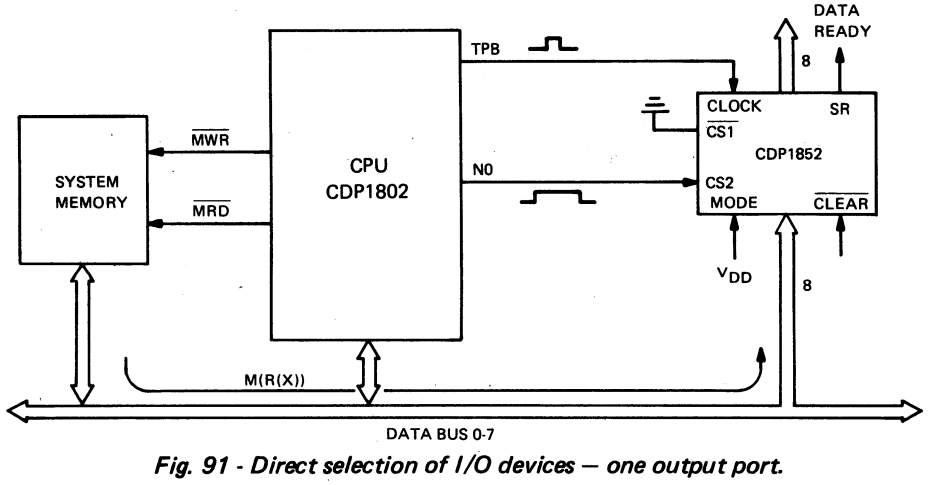


Fig. 91 - Direct selection of I/O devices – one output port.

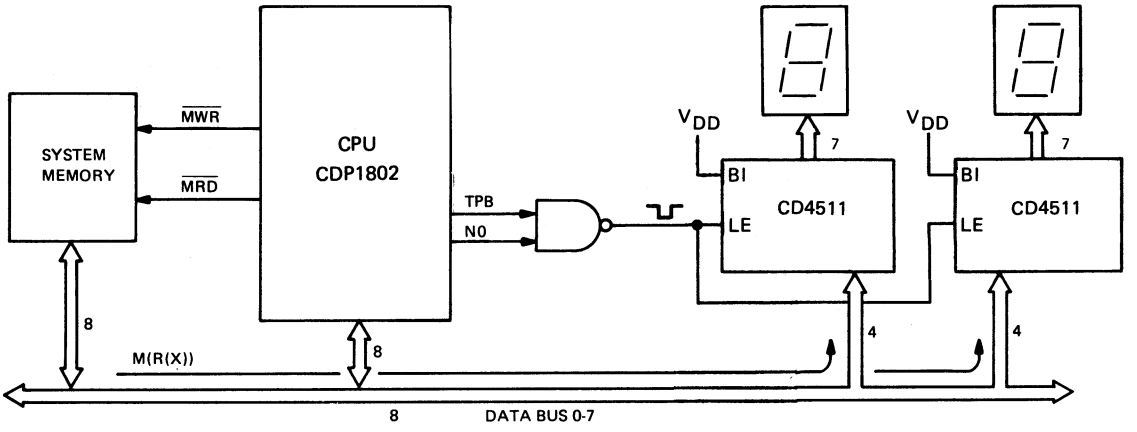


Fig. 92 — Direct selection of I/O devices — one pair of output display digits.

M address	M byte	operation	comments
		Initialize registers and display	
0018	3C	BN1	Loop here until
	18		switch "ON"
			i.e., $\overline{EF1}$ goes low.
		Code to perform count function	
	61	Output 1	Output the counter byte to display.
	30	BR	Branch to M(0018).
	18		

Fig. 93 — Portion of a two-digit decimal counter program.

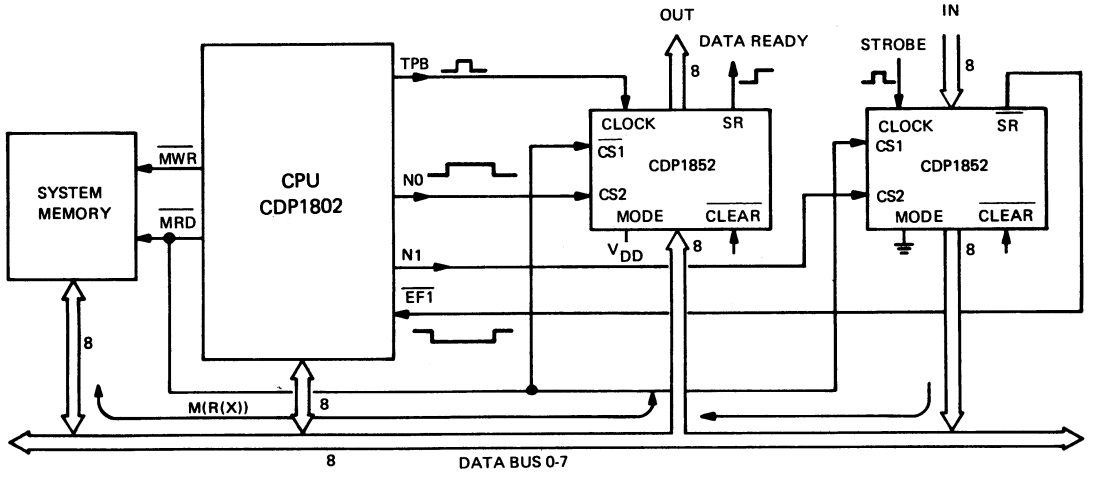


Fig. 94 — Direct selection of I/O devices — one input and one output port.

line determines direction of data flow. During an input instruction execute, the CPU is in a memory write cycle with $\overline{MRD}$ high, i.e., the input register is enabled to the bus and the input byte is read into memory when $\overline{MRD}$ is high. During an output instruction execute, the CPU is in a memory read cycle with $\overline{MRD}$ low and, hence, the input register is disabled from the bus. At TPB, valid data is strobed from the bus into the output register.

More than one I/O device – Input ports only. The simple byte logic in Fig. 90 can be expanded up to three I/O ports with direct selection of the devices, as shown in Fig. 95. The three N bits are valid during I/O operation; hence, instruction 69 selects port I, 6A selects port II, and 6C selects port III. The user's strobe will activate an EF flag and enter a byte into the register.

More than one I/O device – Output ports only. The simple logic described in Fig. 92 can be similarly expanded to handle three pairs of display digits, as shown in Fig. 96. Each digit pair is selected by one of the N lines, depending on the chosen instruction. Instruction 61 selects digit pair D0, 62 selects pair D1, and 64 selects pair D2.

More than one I/O device – Both input ports and output ports. The circuits in Figs. 95 and 96 were designed for data flow in or out, respectively. Three I/O ports under control of the N lines can easily be wired up by expanding the logic in Fig. 94.

Another simple three-port I/O system is shown in Fig. 97. The N lines can select directly either one input port or one of two output ports. N0 controls output information for a multiplexed 4-digit LED display. Four bits of an output byte may contain BCD data which is latched, decoded, and drives the seven segments of the digits. The remaining four bits of the byte select one digit for multiplexing.

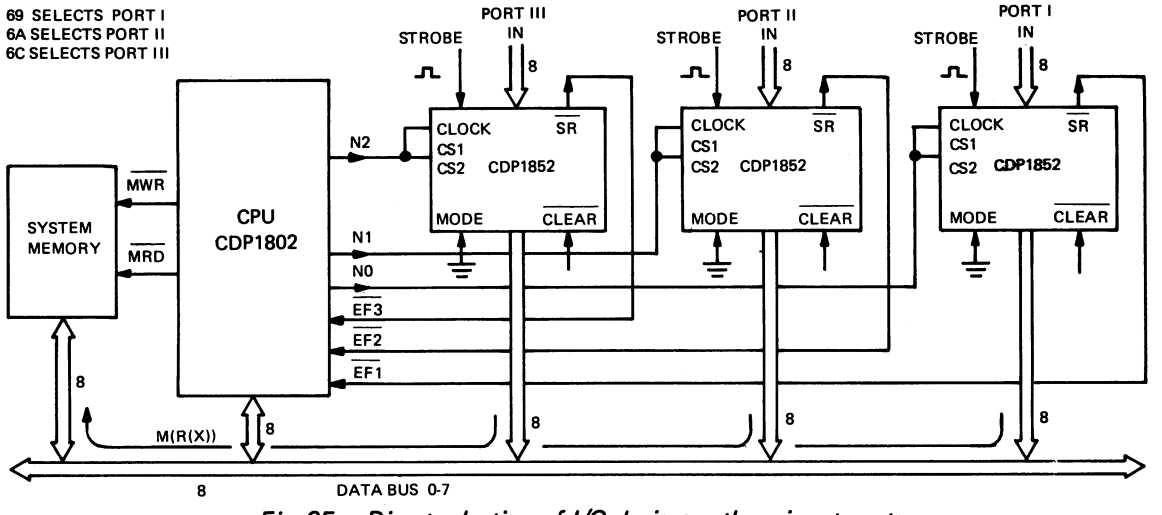


Fig. 95 – Direct selection of I/O devices – three input ports.

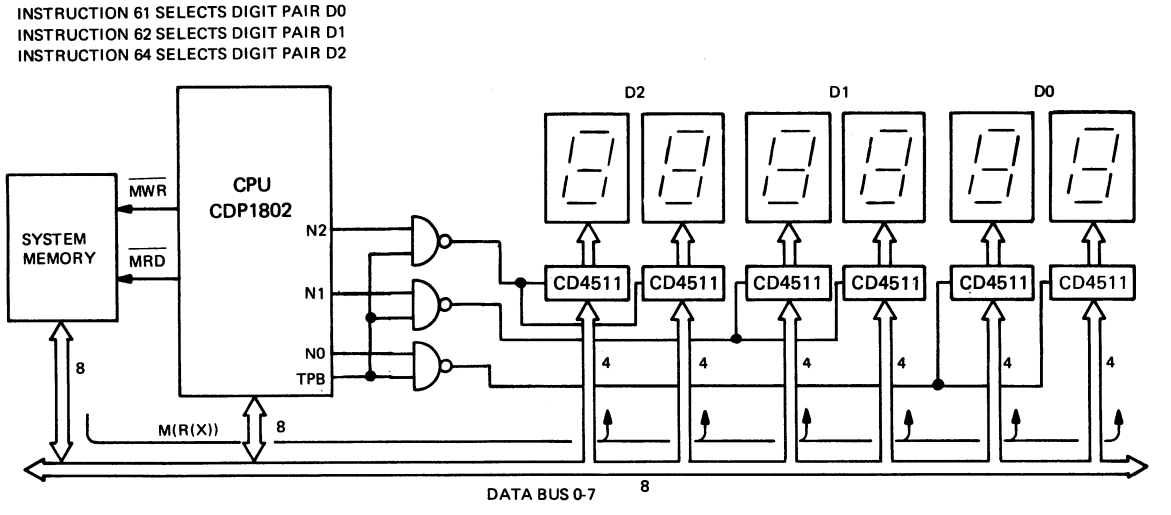


Fig. 96 – Direct selection of I/O devices – three pairs of output display digits.

Programmed I/O — One-Level I/O System

The I/O interface systems described so far are applicable for small systems (up to three I/O ports) where the N lines can be used directly to select or control I/O devices. If more than three I/O devices are required, the N lines can be decoded to specify up to 1 of 7 different I/O ports or channels. Fig. 98 illustrates this approach. If line 1 is selected from the decoder, for instance by executing input instruction 69, the input register is enabled to the bus because $\overline{MRD}$ is high during memory write cycle. Decode line 1 will also be active high during an output instruction, 61, but $\overline{MRD}$ is low during memory read cycle, disabling the input register from the bus. At TPB, the valid byte from memory is strobed into the output register.

As discussed earlier, the user's strobe or write signal can be used to activate an EF flag or the interrupt line. An I/O request can be acknowledged by OR-ing the N lines. If the interrupt is asserted, the two state-code lines SC0 and SC1 are both high, acknowledging an interrupt (S3) cycle.

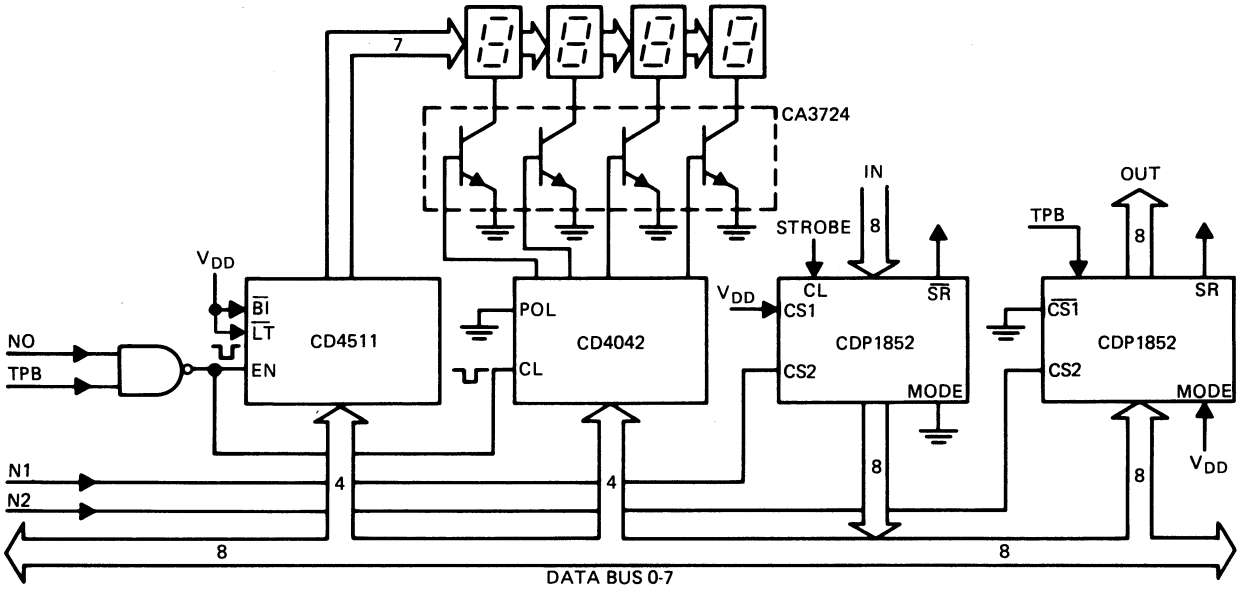


Fig. 97 — Direct selection of I/O devices — one input port or one of two output ports.

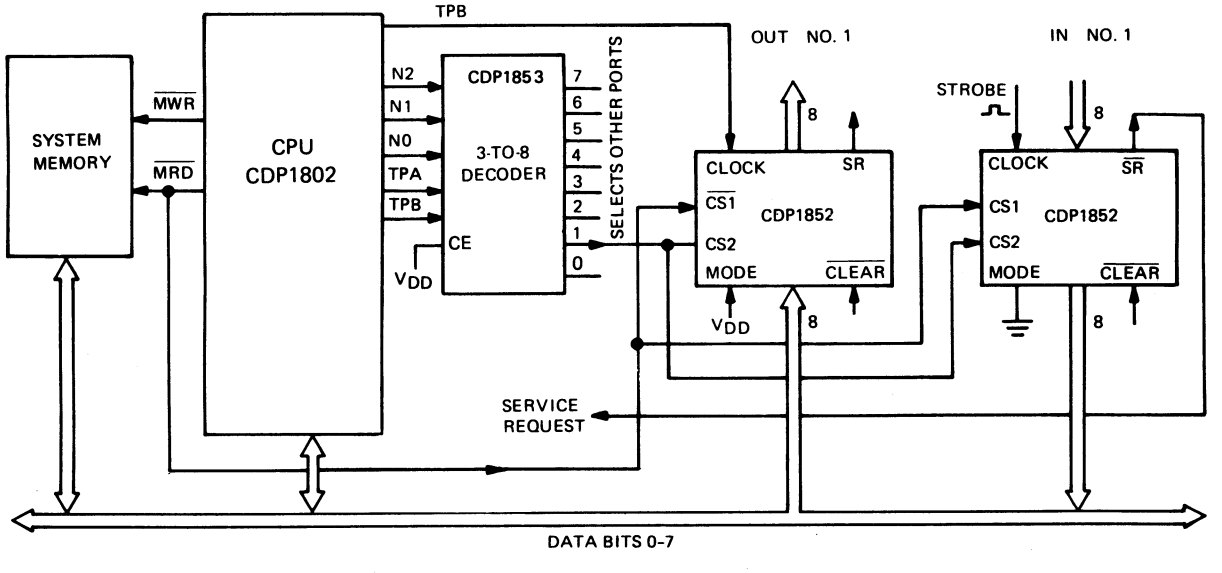


Fig. 98 — Selection of I/O devices by one-level decoding — one of seven input ports or one of seven output ports.

Programmed I/O – Two-Level I/O System

The COSMAC architecture imposes no theoretical limits on the number of I/O ports which the CPU can accommodate. Systems larger than those discussed up till now, however, require an additional level of decoding.

Fig. 99 shows one possible implementation of a large I/O system that handles 392 input and 392 output ports. A 61 instruction is first executed to place an 8-bit group-select code in the I/O group-select register. This code selects one of eight groups in the same way as the one-level system given in Fig. 98. The 3-to-8 decoder that generated the group select can now be disabled (by means of its chip select), and the execution of any input or output instruction selects 1 of 14 I/O ports. By varying the group-select code, 1 of 8 x14 = 112 ports can be selected.

For larger systems, similar master groups of 112 ports can be added, up to a total in this example of 7 x 112 = 784 ports. As an example, instruction 62 selects master group #2, and the output byte selects one of eight groups each of which again consists of 7 input and 7 output ports.

So far the discussion has been limited to I/O systems in which the I/O ports are connected to the data bus. The COSMAC architecture, however, also permits data to be outputted over the address bus. This feature is of particular interest to users constrained to operating with memory registers containing only ROM or to users wishing to transfer 16 bits of data with a single output instruction. Readers interested in transferring data from one of the 16 general-purpose registers to an I/O device should refer to ICAN-6562 "Register-Based Output Function for RCA COSMAC Microprocessors".

The above examples under "I/O Interface" indicate

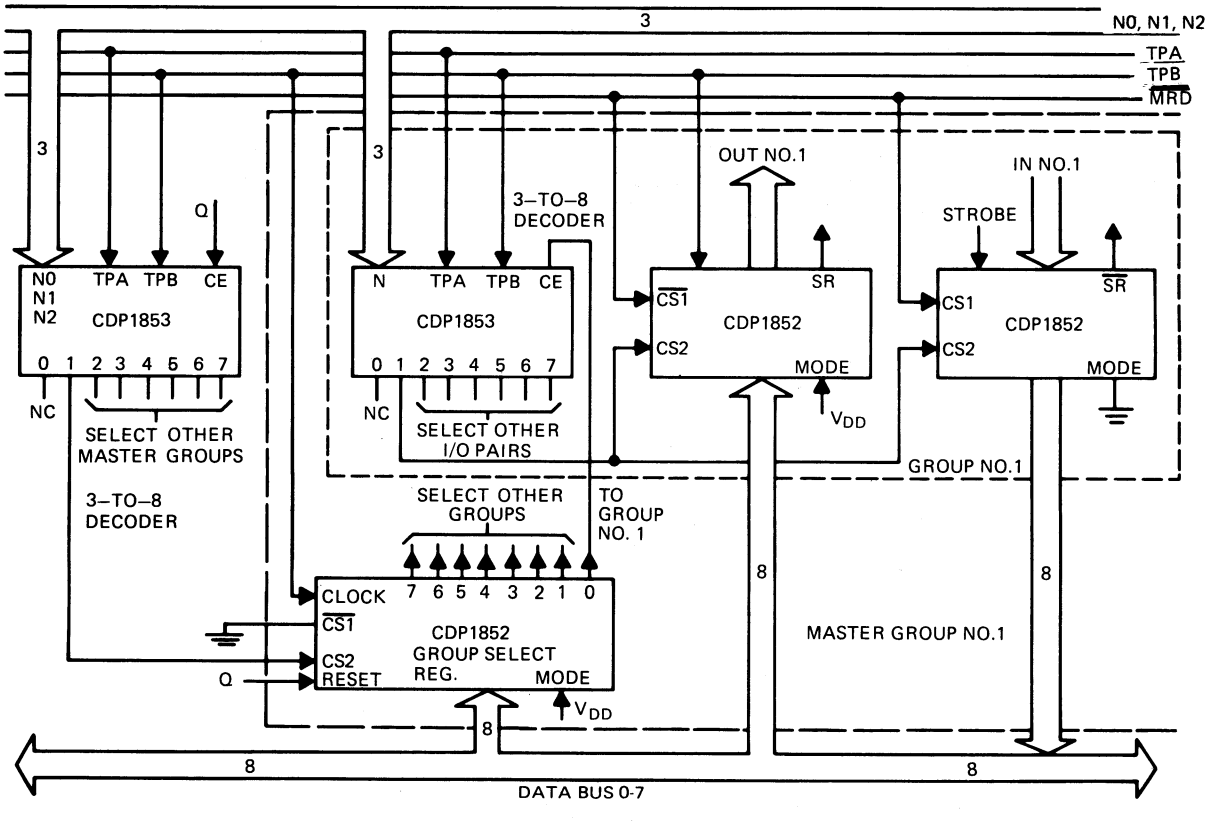


Fig. 99 – Selection of I/O devices by two-level decoding – one of 392 input ports or one of 392 output ports.

only a few of the ways in which I/O instructions can be implemented. The I/O interface lines can be used in a great variety of ways, limited only by the ingenuity of the system designer.

DMA Operation

The I/O examples described above require that a program periodically sample I/O device status. These techniques also require several instruction executions for each I/O byte transfer. In many cases it is desirable to have I/O byte transfers occur without burdening the program or to transfer data at higher rates than possible with programmed I/O. A built-in direct-memory-access (DMA) facility permits high-speed I/O byte transfer operations independent of normal program execution.

During DMA operation R(0) is used as the memory address register and should not be used for other purposes. Two lines, $\overline{\text{DMA-IN}}$ and $\overline{\text{DMA-OUT}}$, are used to request DMA byte transfer to and from the memory. Also, a specific code is provided on the state code lines (SC0, SC1) to indicate a DMA cycle (S2).

mal fetch-execute sequences. If the $\overline{\text{DMA-IN}}$ line goes low during an instruction fetch cycle (S0), then the normally following execute cycle (S1) will still be performed. Following this execute cycle (S1), a special DMA cycle (S2) occurs. If the $\overline{\text{DMA-IN}}$ line goes low during an instruction execute cycle (S1), then the DMA cycle (S2) will follow immediately after S1. If the $\overline{\text{DMA-IN}}$ line is reset to its high state during the DMA cycle (S2), then the deferred next instruction fetch cycle (S0) will be performed following the S2 cycle, as shown on Fig. 100.

An S2 cycle is indicated by a low SC0 line and a high SC1 line. This condition is used to place a DMA input byte onto the bus, as shown. For further details on timing, see the section **Timing Diagrams**. The S2 cycle stores the input byte in memory at the location addressed by R(0). R(0) is then incremented by 1 so that subsequent S2 cycles will store input bytes in sequential memory locations. S2 cycles do not alter the sequence of program execution. The program will, however, be slowed down by the S2 cycles that are "stolen". The concurrent program must, of course, properly use R(0)

–	DMA-IN ACTION	BUS → M(R(0)); R(0)+1	–
–	DMA-OUT ACTION	M(R(0)) → BUS; R(0)+1	–

DMA-IN

Fig. 100 illustrates the manner in which DMA-IN might be implemented. The leading edge of an enter pulse will clock an input byte into the register and activate the $\overline{\text{DMA-IN}}$ request.

A low $\overline{\text{DMA-IN}}$ line automatically modifies the normal fetch-execute sequences. If the $\overline{\text{DMA-IN}}$ line goes low during an instruction fetch cycle (S0), then the normally following execute cycle (S1) will still be performed. Following this execute cycle (S1), a special DMA cycle (S2) occurs. If the $\overline{\text{DMA-IN}}$ line goes low during an instruction execute cycle (S1), then the DMA cycle (S2) will follow immediately after S1. If the $\overline{\text{DMA-IN}}$ line is reset to its high state during the DMA cycle (S2), then the deferred next instruction fetch cycle (S0) will be performed following the S2 cycle, as shown on Fig. 100.

An S2 cycle is indicated by a low SC0 line and a high SC1 line. This condition is used to place a DMA input byte onto the bus, as shown. For further details on timing, see the section **Timing Diagrams**. The S2 cycle stores the input byte in memory at the location addressed by R(0). R(0) is then incremented by 1 so that subsequent S2 cycles will store input bytes in sequential memory locations. S2 cycles do not alter the sequence of program execution. The program will, however, be slowed down by the S2 cycles that are "stolen". The concurrent program must, of course, properly use R(0) and memory areas in which input bytes are being stored. It may examine R(0) and the memory area involved to observe the course of the data transfer. The program must also set R(0) to the address of the desired first input byte location in memory before permitting a DMA input operation.

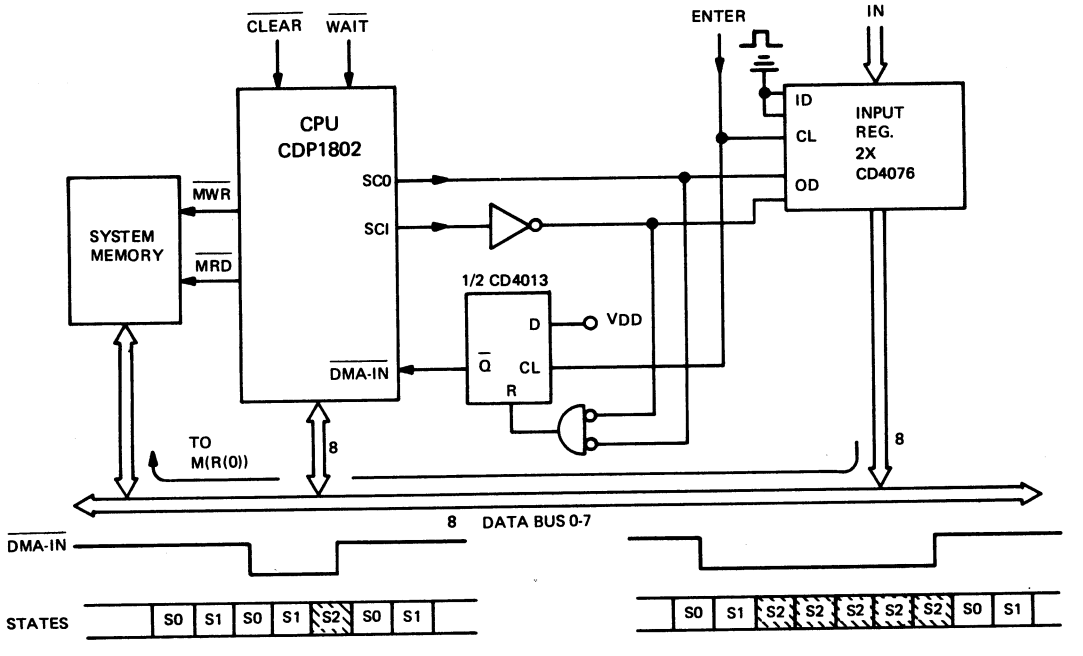


Fig. 100 – Implementation of DMA-IN operation.

So far, single byte transfer per enter request has been discussed. If the $\overline{\text{DMA-IN}}$ remains low, S2 cycles will be performed until the $\overline{\text{DMA-IN}}$ goes high. In this mode of block transfer, the reset logic in Fig. 100 must be modified. The DMA mode permits a maximum I/O byte transfer rate of one byte per machine cycle which, with two microseconds per instruction cycle time, amounts to a transfer rate of one megabyte per second.

The $\overline{\text{DMA-IN}}$ feature, in conjunction with $\overline{\text{CLEAR}}$ and $\overline{\text{WAIT}}$ signals, provides a built-in **program load mechanism**. A low on $\overline{\text{CLEAR}}$ and a high on $\overline{\text{WAIT}}$ puts the CPU in the RESET mode. The CPU now idles (S1 state) with R(0) = 0000. The LOAD mode is next entered by bringing the $\overline{\text{WAIT}}$ line low ($\overline{\text{CLEAR}}$ = L and $\overline{\text{WAIT}}$ = L). This mode allows input bytes to be sequentially loaded into memory beginning at M(0000). Input bytes can be supplied from a keyboard, tape reader, etc. via the DMA-IN facility and circuitry similar to Fig. 100. For details on timing refer to the material in this section on "Control Interface."

DMA-OUT

A low on the $\overline{\text{DMA-OUT}}$ line causes S2 cycles to occur in a similar manner as a low on the $\overline{\text{DMA-IN}}$ line. The S2 cycle caused by a low on the $\overline{\text{DMA-OUT}}$ line places the memory byte addressed by R(0) on the bus and increments R(0) by 1. DMA output bytes can be strobed into an output device by TPB, as shown in Fig. 101. The program must set R(0) to the address of the first output byte of the desired memory sequence before the DMA transfer request occurs. For details on DMA-OUT timing, refer to the section on **Timing Diagrams.**

Note: In the event of concurrent DMA and INTERRUPT requests, DMA-IN has priority followed by DMA-OUT and then INTERRUPT.

Interrupt I/O

The interrupt mechanism permits an external signal to interrupt program execution and transfer control to a program designed to handle the interrupt condition. This function is useful for responding to system alarm conditions, initializing the DMA memory pointer, or, in general, responding to real-time events less urgent than those handled by DMA but more urgent than those which can be handled by sensing external flags.

A low on the $\overline{\text{INTERRUPT}}$ line causes an **interrupt response cycle (S3)** to occur following the next S1 cycle, provided the IE flip-flop is set. Execution of an S3 cycle is indicated by a high on both the SC0 and the SC1 lines.

Fig. 102 shows a typical interrupt circuit. The flip-flop is reset during the S3 cycle. During the S3 cycle, the current values of the X and P registers are stored in the T register. P is then set to 1, X to 2, and IE to 0. Following S3, a normal instruction fetch cycle (S0) is performed. The S3 cycle, however, changed P to 1 so that, next, the sequence of instructions starting at the memory location addressed by R(1) will be executed. This sequence of instructions is called the **interrupt service program**. It saves the current state of the COSMAC registers such as T, D, and possibly some of the scratchpad registers, by storing them in reserved memory locations. DF must also be saved if the interrupt service program will disturb it. The service program then performs the desired functions, restores the saved registers to their original states, and returns control to execution of the original program. Special instructions RETURN, DISABLE, and SAVE (70, 71, and 78) facilitate interrupt handling. These instructions are described in the sections on **Instruction Repertoire** and **Instruction Utilization.**

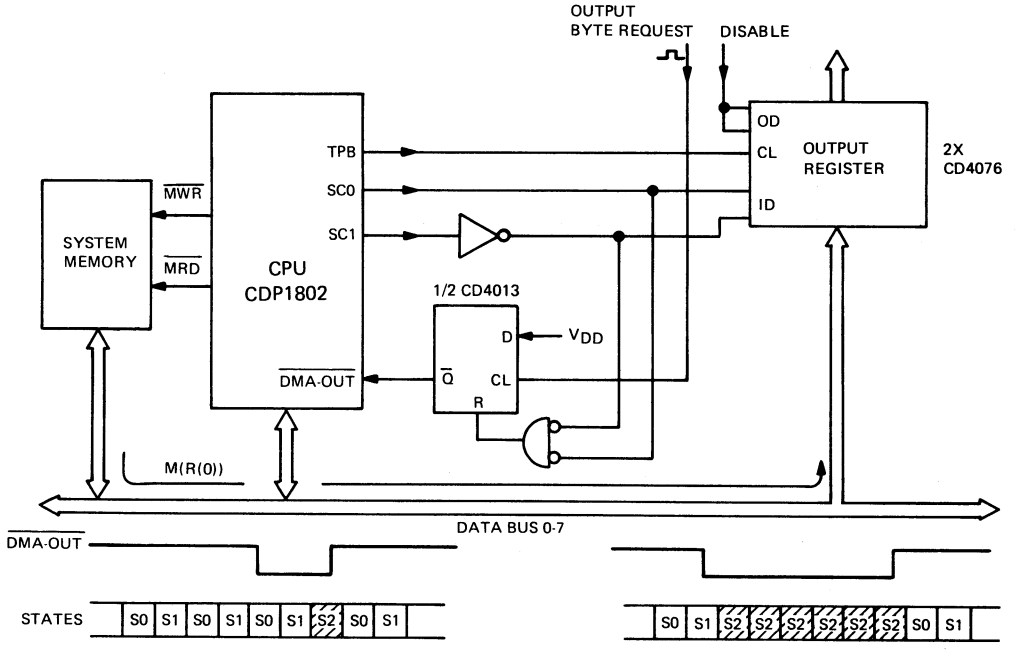


Fig. 101 — Implementation of DMA-OUT operation.

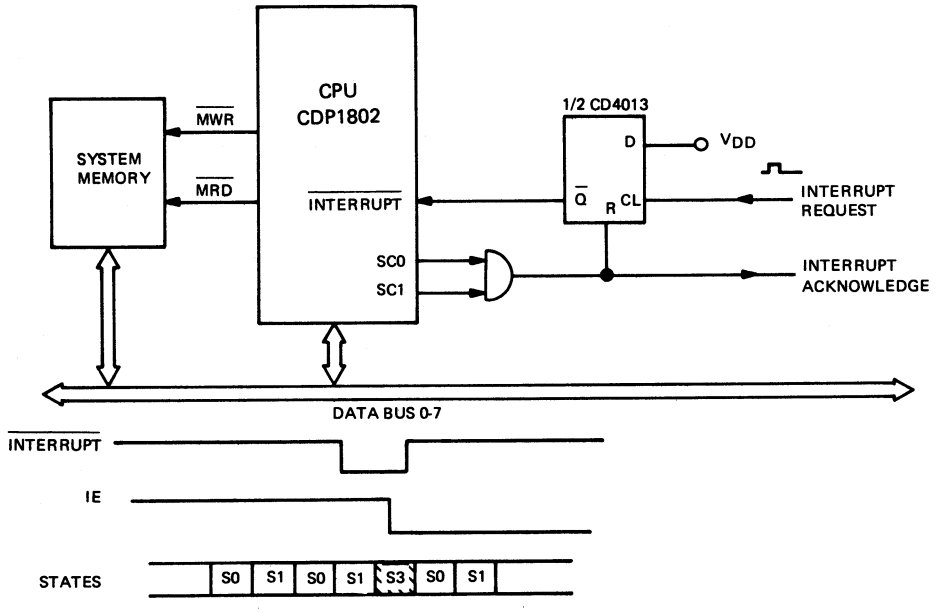


Fig. 102 – Typical circuit for implementation of interrupt operation.

Their use is illustrated in the section on **Programming Techniques** under the heading "Interrupt Service."

The COSMAC Microprocessor also provides a special one-bit register called Interrupt Enable (IE). When IE is set to "0", the state of the interrupt line is ignored. IE is set to "1" in the reset mode. IE can be set to "1" or "0" by RETURN and DISABLE instructions, respectively. It is automatically set to "0" by an S3 cycle, preventing subsequent interrupt cycles even if the INTERRUPT line stays low. The program must set IE to "1" to permit subsequent interrupts. Setting IE to "1" takes place automatically when the program executes the RETURN instruction. Sharing the INTERRUPT line with a number of interrupt signal sources is possible.

When the interrupt facility is used in a system, R(1) must be reserved for use as the interrupt service program counter, and R(2) is normally used as a pointer to a storage area. The latter may be shared with the main program if appropriate conventions are employed, as described in the section on **Programming Techniques**.

System Configurations

Parallel I/O Interface

Fig. 103 shows the CPU interfaced to other parts members of the 1800 family. Only five parts plus a crystal are required to interface directly in a simple and efficient system configuration. The RC network connected to CLEAR is optional and provides power-on reset. This basic system implementation can easily be expanded for larger memory capacity as shown in this

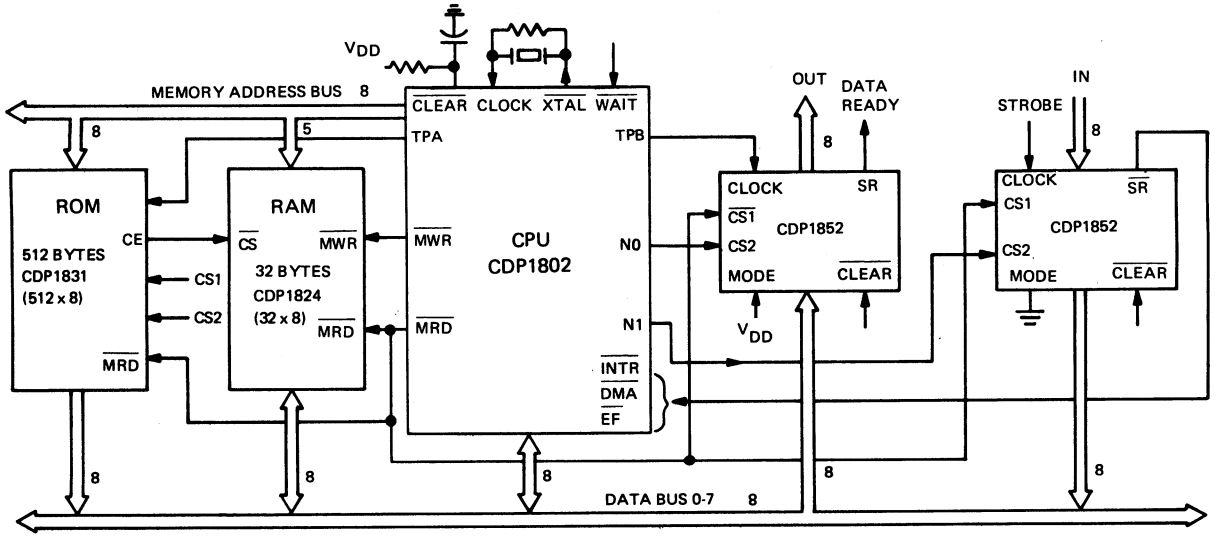


Fig. 103 – System configuration for parallel I/O interface.

section under the heading "Memory Interface and Timing."

More I/O ports can be added by following one of the approaches outlined under "Programmed I/O." Several N bits can be used directly for I/O control, or they can be decoded for either one- or two-level I/O systems.

The byte I/O register can be configured as either input or output port by the level on the MODE control. When the register is used as an input port, data is latched into it by a low level on the clock line. The negative transition of clock sets $\overline{SR} = 0$, which can be used as a service request to the microprocessor. Data is read out of the port onto the data bus when the chip is selected ($CS1 \cdot CS2 = 1$), and the negative transition of $CS1 \cdot CS2$ resets the service request ($\overline{SR} = 1$).

When the byte I/O register is used as an output port, i.e., MODE control = H, data is strobed into the register from the bus by $\overline{CS1} \cdot CS2 \cdot TPB = 1$. In the output mode, data is enabled out of the port at all times.

Serial I/O Interface

Using EF input and Q output. Fig. 104 shows a simple, serial interface for a COSMAC-based computer system. The sequential logic states of one of the EF lines may represent a bit-serial character. A software program can then interpret these logic levels and assemble the bits into one-byte data words in memory.

In an analogous manner, SEQ and REQ instructions in the program can generate high and low levels on the Q output line for serial transmission of a byte from memory. This method can be used for interfacing a Teletype, printer, or any peripheral with a serial interface. A typical interface circuit between the peripheral and the CPU is shown in the **Evaluation Kit Manual for the RCA CDP1802 COSMAC Microprocessor**, MPM-203.

Another illustration of Q as an output under program control is given in Fig. 105. This minimum system configuration shows a 4-bit digital combination lock. The status of the four manual switches or buttons representing a combination is tested by short branch instructions. If the combination is correct, the program sets Q = 1, thereby activating the solenoid of an electric lock.

The basic program with sixteen possible combinations can be enhanced by various additions. For instance, the correct combination must be entered in a certain

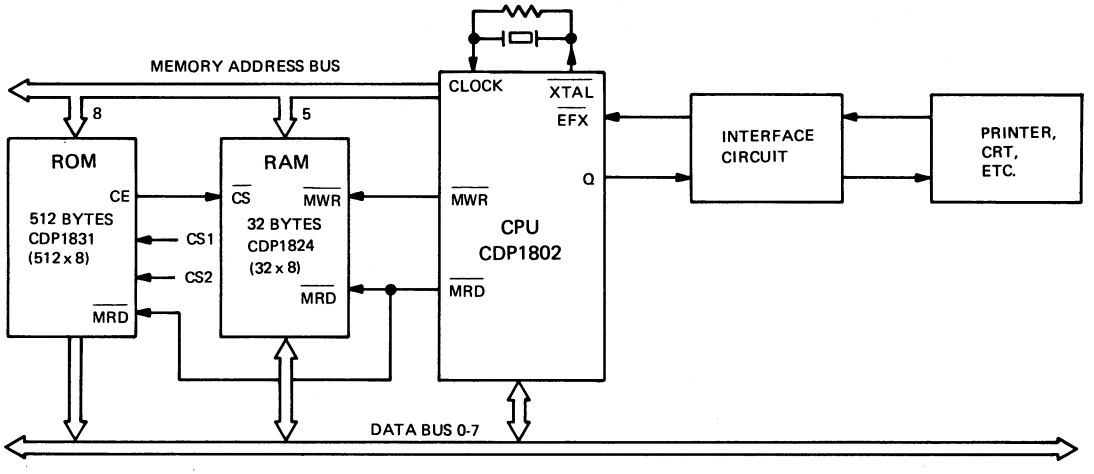


Fig. 104 — System configuration for serial I/O interface.

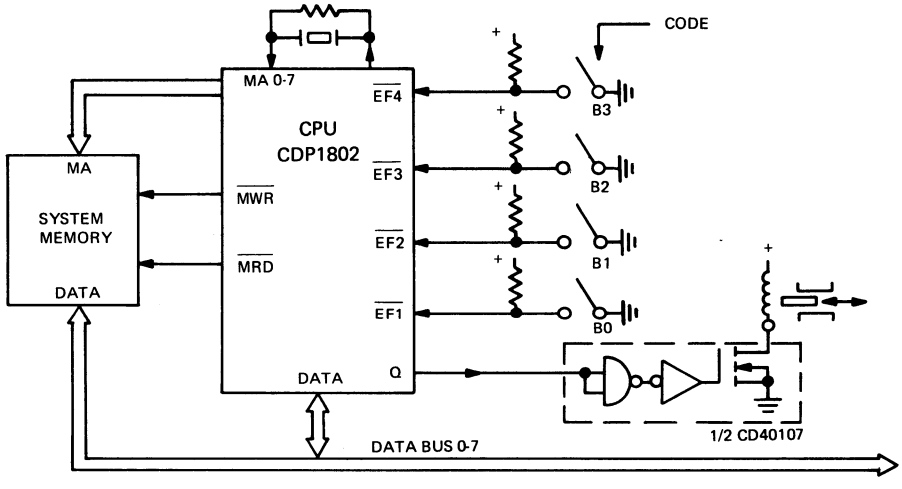


Fig. 105 — System configuration for a 4-bit combination lock.

sequence, or the code must be set within a minimum time period.

Discussed earlier were I/O systems of varying complexities in which the I/O ports were selected either directly or through one or two levels of decoding. In this context, the digital combination lock represents the lowest order of complexity. No selection is required and no input or output register is necessary.

Using universal asynchronous receiver-transmitter. A more sophisticated and powerful approach to serial interfacing than the one shown in Fig. 104 is outlined below. The program itself is relieved of the task of formatting and control, and these functions are taken over by a dedicated hardware circuit.

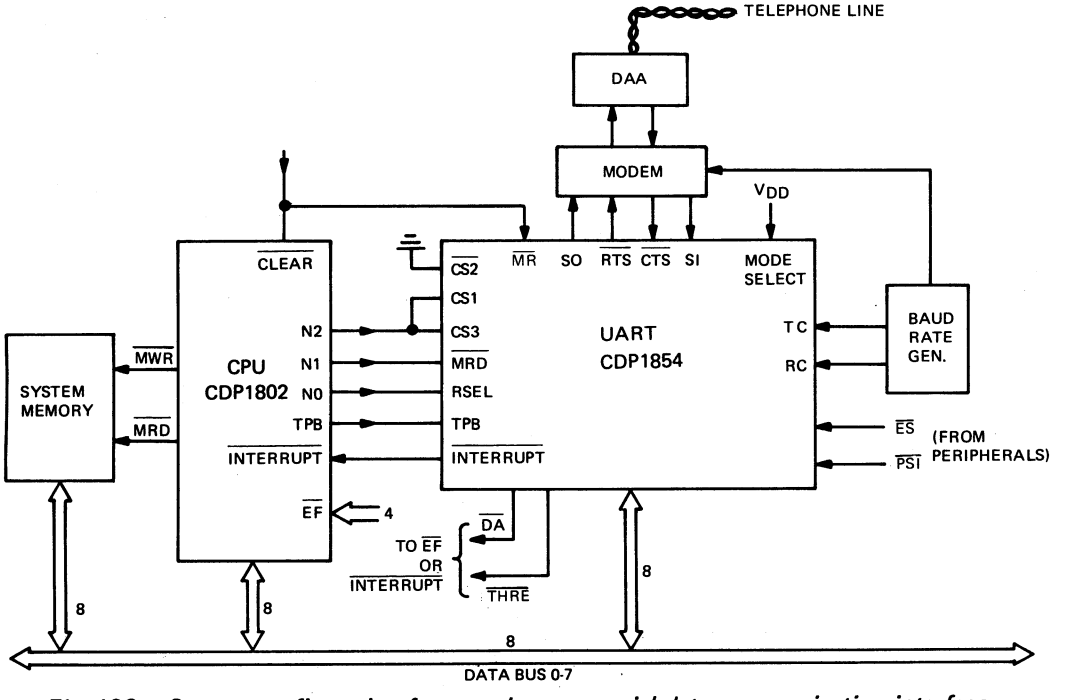


Fig. 106 – System configuration for asynchronous serial data communication interface.

Fig. 106 shows the CDP1854, a CMOS Universal Asynchronous Receiver-Transmitter (UART), interfaced to the CPU in a typical data communication application. The UART consists of a receiver and transmitter designed to provide the necessary formatting and control for interfacing serial asynchronous data to and from peripheral devices. The receiver-transmitter is capable of full duplex operation and is externally programmable.

The transmitter converts parallel data to a serial word containing the data (5-8 bits), a start bit, a parity bit (optional), and stop bits (1, 1½, or 2), as illustrated in Fig. 107. The receiver converts a serial input word with start, data, parity, and stop bits into parallel data. It verifies proper code by checking parity and the receipt of a valid stop bit. Both the receiver and transmitter are double buffered.

Although the receiver and transmitter can operate with separate data buses, if the MODSEL line is high, the UART is directly compatible with COSMAC and bidirectional data transfer on a common bus.

There are four registers under program control in the UART. One is loaded **from** the bus in the transmit mode, one is read **to** the bus in the receive mode, a Control register is loaded from the bus at initialization, and a Status register is read in the receive mode. The two-bit code on $\overline{MRD}$ and RSEL determines which register is selected and the direction of data flow. Refer to the truth table in Fig. 108.

The UART is enabled to the data bus when the three chip selects are asserted. Therefore, by decoding, a large number of UART's can operate in a system on the same bus.

Fig. 106 illustrates one possible way of interfacing the UART and the CPU. One of the N bits selects the

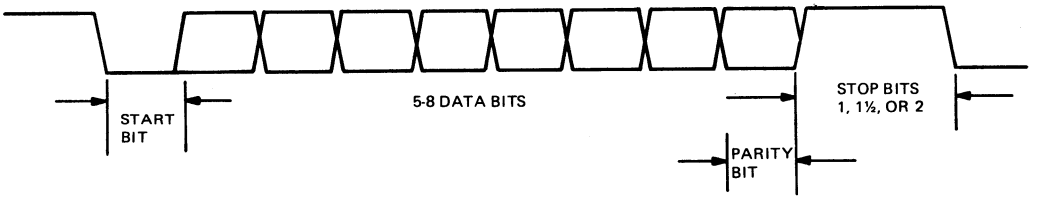


Fig. 107 – Word format in asynchronous serial data communication.

chip, and the two other N bits select register and receive/transmit mode. A typical operation might be as follows:

The control register is first loaded from memory under program control, for instance, through executing output instruction 65. The individual bits in the control byte set up the system mode according to the bit pattern. The bit assignments are listed in Fig. 109. The program determines if the system operates with parity or not; if parity, odd or even; how many stop bits; the number of bits in the data word; etc. For instance, a high in bit position 0 will inhibit parity generation; or, if transmit break is present, the serial-data-out line is held low.

Transmitting a character is initiated by executing, for instance, output instruction 64. During the execute cycle, N2 selects the chip and N1 and N0 select the Transmitter Holding Register (THR), which is loaded from the bus at TPB. When the byte has been transferred to the Transmitter Shift Register (TSR) for eventual serial transmission (on SO), the Interrupt line is asserted to indicate that THR is empty and a new character may be loaded. Reading the Status Register will also provide this information.

In the receive mode, a serial character is entered on the SI line and shifted into the Receiver Shift Register. When this register is full, the byte is transferred to the Receiver Holding Register (RHR) and a Data Available flag is generated, which is one of the status bits. Before accepting the character, the program would typically read the Status Register. The bit assignments of the latter are shown in Fig. 110. The program can therefore determine if, for instance, there is parity error or if a complete character has been received and is ready for transfer over the bus to memory.

INSTRUCTIONS OUT	INSTRUCTIONS IN	CHIP SELECT N2	$\overline{MRD}$ N1	RSEL N0	ASYNCHRONOUS RECEIVER/ TRANSMITTER REGISTER	OPERATION
61	69	0	0	1	–	–
62	6A	0	1	0	–	–
63	6B	0	1	1	–	–
64	6C	1	0	0	Transmit Data	BUS → Tr. Data
65	6D	1	0	1	Control	BUS → Control
66	6E	1	1	0	Receive Data	Rec. Data → BUS
67	6F	1	1	1	Status	Status → BUS

Fig. 108 – Truth table for selecting chip and registers with the N bits.

Some of the status bits, if set, will also generate an interrupt condition. The Status Register in this example is read by executing input instruction 6F. If the received character is acceptable, the input instruction 6D will enable the byte onto the bus, and $\overline{MWR}$ writes it into memory. Upon reading of data from the UART, the DA status bit is automatically reset. If another character is received before the previous one is read out, an overrun condition is signaled.

Flexibility in system operation is enhanced by a few additional signal lines. The DA and THRE flags are brought out separately and could, for instance, signal the CPU over the EF lines. Clear to Send ($\overline{CTS}$) and Request to Send ($\overline{RTS}$), in addition to the two peripheral status lines $\overline{ES}$ and $\overline{PSI}$, facilitate "hand-shaking" with modems and peripherals.

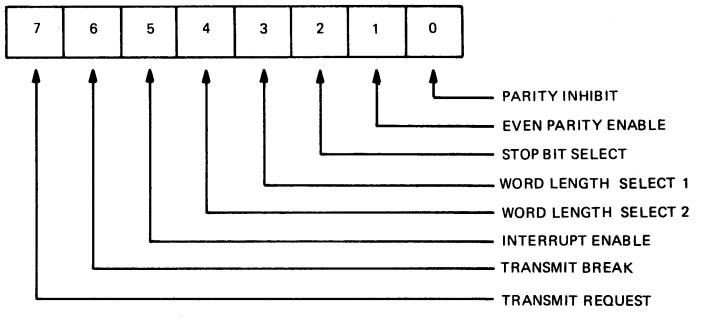


Fig. 109 – Bit assignments for control register.

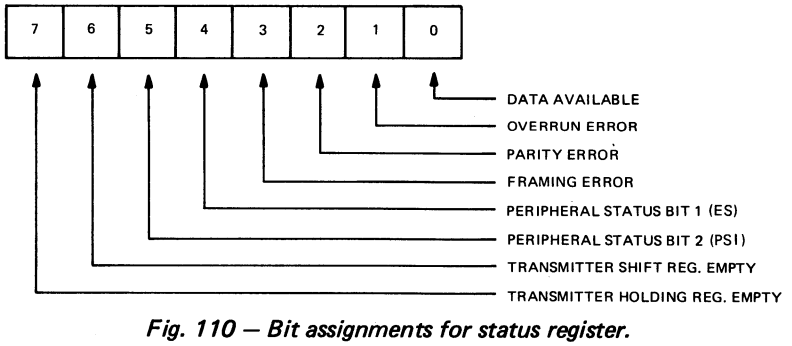


Fig. 110 – Bit assignments for status register.

Timing Diagrams

The following topics illustrated with timing diagrams are covered in this section.

1. Input instruction timing.
2. Output instruction timing.
3. DMA-IN timing.
4. DMA-OUT timing.
5. Sampling of CPU and user-generated signals.
6. State of data bus and memory address bus.

Input Instruction Timing

Fig. 111 provides the timing relationships for input instructions. An input instruction will permit a byte from an external device to be written into memory and the D register:

$$BUS \rightarrow M(R(X)), D$$

The instruction 69, for instance, will be fetched from memory during state S0 when the CPU asserts $\overline{MRD}$ and reads the instruction into the I and N registers. The instruction will be executed during the next machine cycle, state S1, which is a memory write cycle. The CPU generates an active low $\overline{MWR}$ pulse during this cycle which will strobe an input byte from the data bus into memory. A high $\overline{MRD}$ level during the memory write cycle will also disable the memory output during this period.

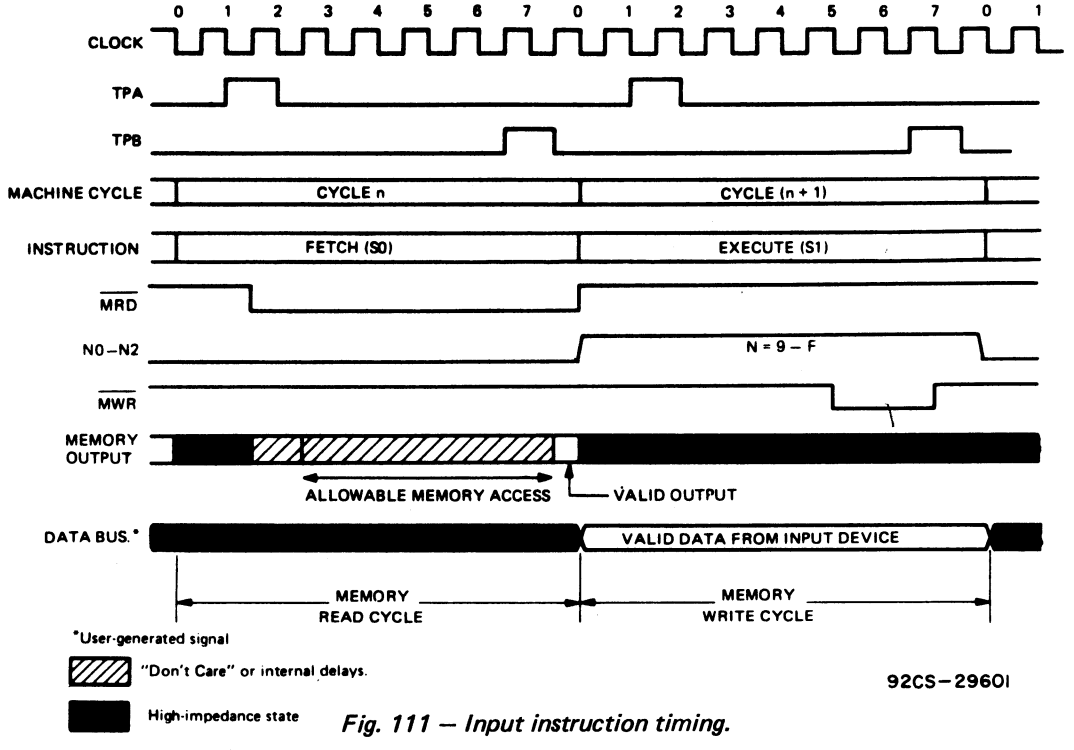


Fig. 111 – Input instruction timing.

During the execute cycle when I = 6 and N = 9, A, B, C, D, E, or F, the three low-order N bits are available and can be used for enabling a byte onto the data bus from the input device.

Output Instruction Timing

Fig. 112 provides the timing relationships for output instructions. An output instruction will permit a stored byte in memory to be read out to an external device:

$$M(R(X)) \rightarrow BUS; R(X) + 1$$

The instruction 61, for instance, will be fetched from memory during state S0 when the CPU asserts $\overline{MRD}$ and reads the instruction into the I and N registers. The instruction will be executed during the next machine cycle, state S1, which is now a memory read cycle. $\overline{MRD}$ is during the latter cycle once more asserted and enables the output from the memory onto the bus. Data is valid after the access time has elapsed. The valid data from memory can next be strobed into the output device by a user-generated strobe. Data will always be valid when TPB, the N bits, and $\overline{MRD}$ signals are true.

The three N bits are valid during the execute cycle when I = 6 and N = 1, 2, 3, 4, 5, 6, or 7 and can be used for enabling a byte from the data bus into the output device.

DMA-IN Timing

Fig. 113 provides the timing relations for DMA-IN operation. When $\overline{DMA\text{-}IN}$ is asserted, a byte on the data bus from an external device is written into memory at the location specified by the register R(0):

$$BUS \rightarrow M(R(0)), R(0) + 1$$

$\overline{DMA\text{-}IN}$ is a user-generated signal that can be asserted any time, but the CPU will always complete its current instruction cycle before it enters the DMA cycle or state S2. The $\overline{DMA\text{-}IN}$ request is sampled internally during TPB and the end of an S1, S2, or S3 state. Note that the last execute cycle before the DMA cycle can be either a memory read, a memory write, or a non-memory cycle. When the CPU enters the DMA state following DMA-IN, it enters a memory write cycle. Memory output is disabled by a high $\overline{MRD}$ level, and a low $\overline{MWR}$ pulse is generated which will write valid data on the bus supplied from the input device into memory.

During S2, $\overline{MRD}$ is high and will disable memory output to the data bus. If the $\overline{DMA\text{-}IN}$ request goes away during S2, the CPU will next execute a fetch cycle and complete the next instruction cycle which had been deferred.

DMA-OUT Timing

Fig. 114 provides the timing relations for the DMA-OUT operation. When $\overline{DMA\text{-}OUT}$ is asserted, a byte stored in memory at the location specified by register R(0) is read out to the data bus and can be strobed into an external device:

$$M(R(0)) \rightarrow BUS, R(0) + 1$$

$\overline{DMA\text{-}OUT}$ is a user-generated signal and can be asserted any time, but the CPU will always complete its current instruction cycle before it enters the DMA cycle or state S2. The $\overline{DMA\text{-}OUT}$ request is sampled internally during TPB and the end of an S1, S2, or S3 state. Note that the last execute cycle before the DMA cycle can be either a memory read, a memory write, or a non-memory cycle. When the CPU enters the DMA state fol-

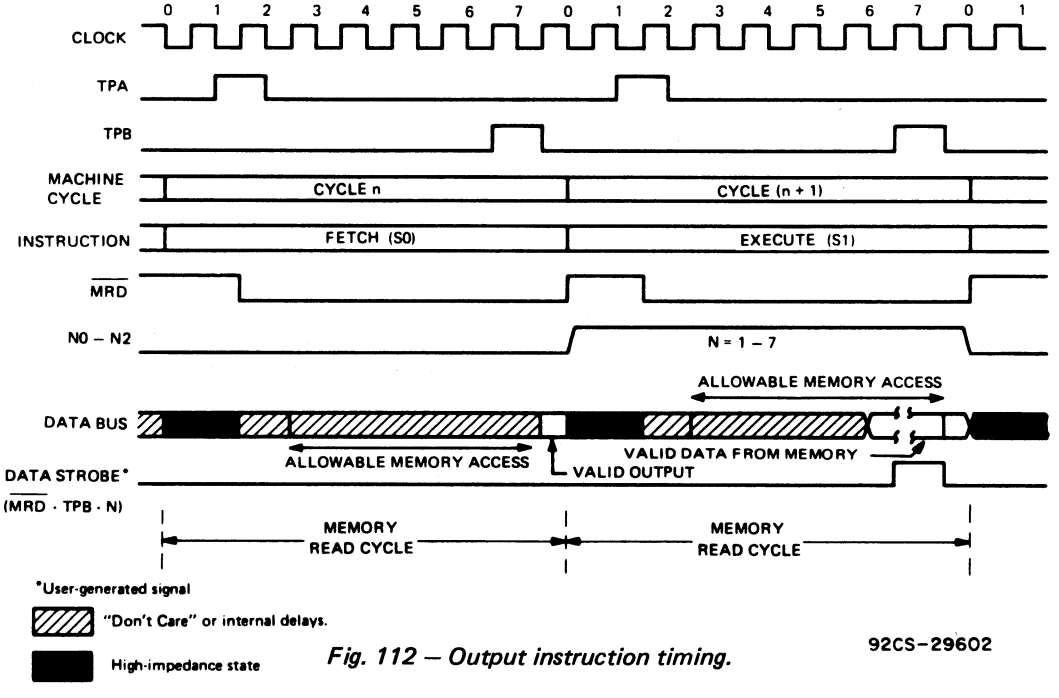


Fig. 112 – Output instruction timing.

92CS-29602

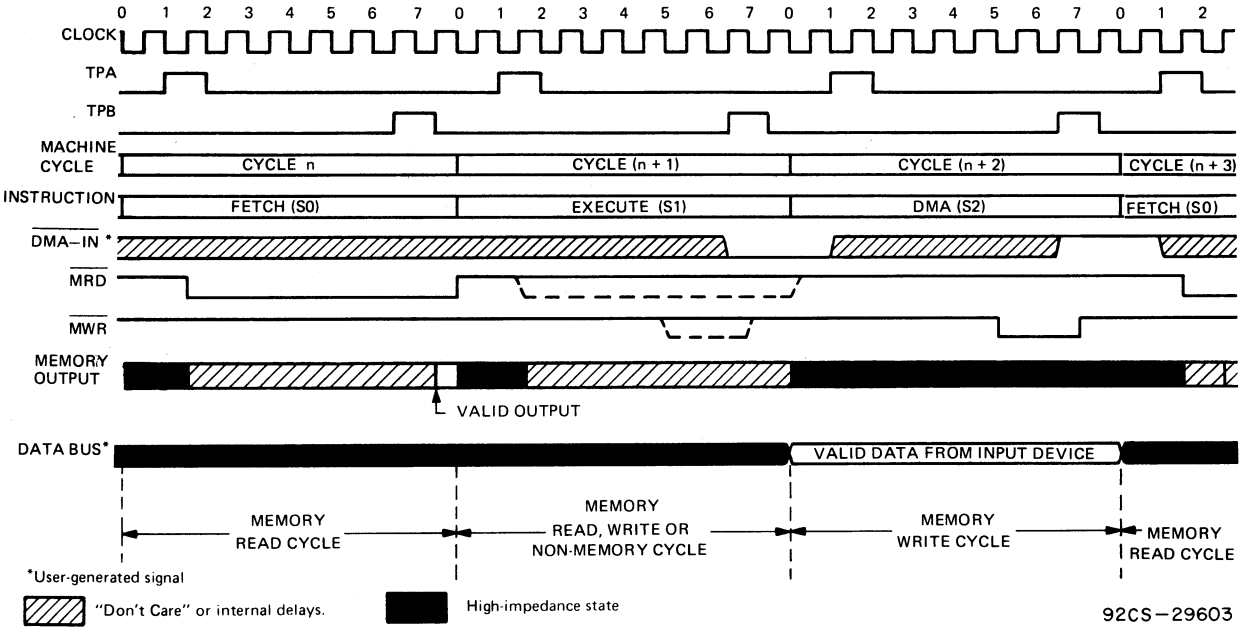


Fig. 113 – DMA-IN timing.

lowing DMA-OUT, it goes into a memory read cycle. The memory is enabled to the bus when $\overline{MRD}$ is low and, after the necessary access time, valid data appears on the data bus and can be strobed into the output device. An appropriate data strobe can be generated by the user during S2 when TPB is true.

If the $\overline{DMA\text{-}OUT}$ request goes away during S2, the CPU will revert to a fetch cycle and complete the next instruction cycle.

Interrupt Timing

Fig. 115 provides the timing relations for interrupt service. $\overline{INTERRUPT}$ is a user-generated signal which can be asserted any time. However, the request is not recognized until the end of the current instruction cycle. It is recognized then only if the INTERRUPT ENABLE (IE) flip-flop in the CPU is set. Interrupt is sampled internally at the end of each execute cycle. The execute cycle can be either a memory read, a memory write, or a non-memory cycle.

The interrupt state, S3, is a non-memory cycle. During this period the contents of X and P are stored in the temporary register T, and X and P are set to new values: 2 in X and 1 in P. The interrupt enable flip-flop is automatically deactivated to inhibit further interrupts. The interrupt routine is now in control, and the next machine cycle is a fetch operation.

Note: DMA has priority over Interrupt.

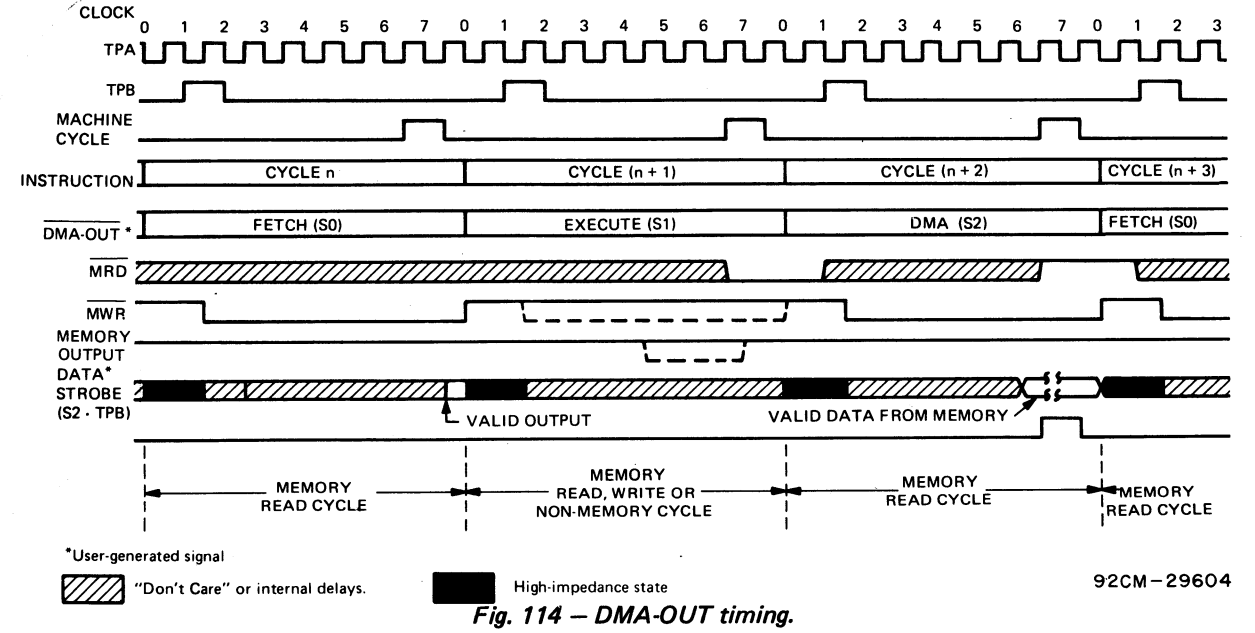


Fig. 114 – DMA-OUT timing.

ANY FUNCTION IS LISTED ONE LINE ABOVE WHERE IT SHOULD BE. ("TPA" IS LISTED OPPOSITE THE "CLOCK" LINE, "TPB" IS OPPOSITE "TPA" LINE, ETC...)

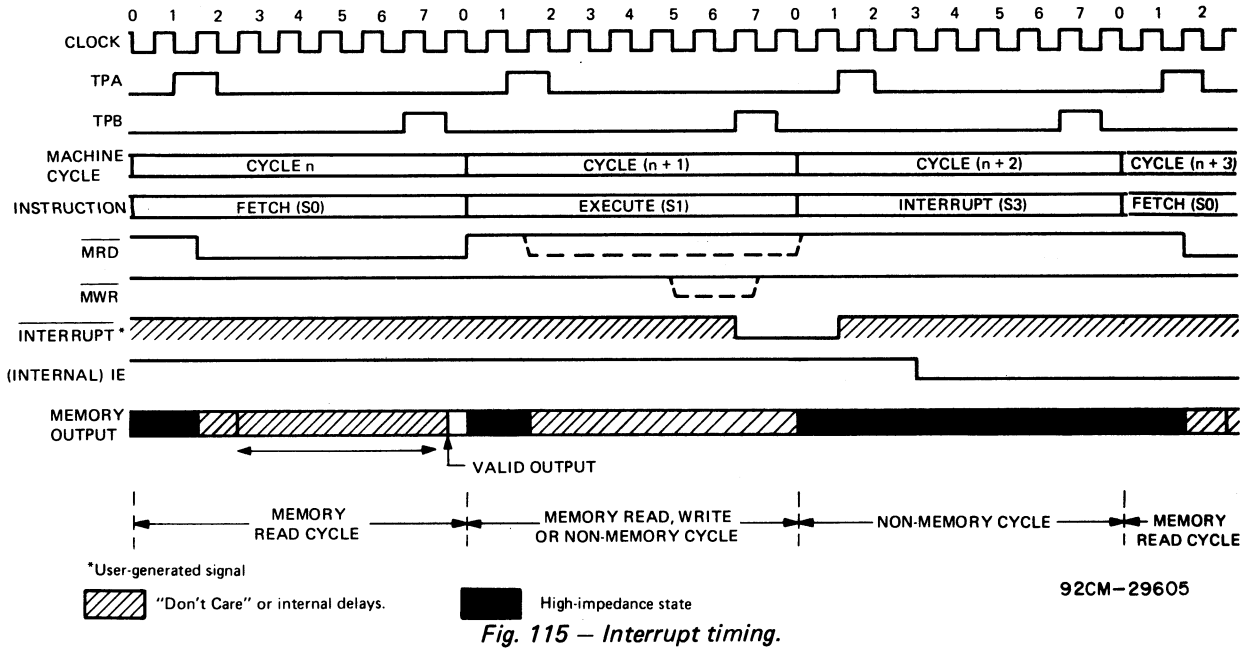


Fig. 115 – Interrupt timing.

Sampling of CPU and User-generated Signals

The timing diagram in Fig. 116 shows when in the machine cycle certain CPU signals and user-generated signals are sampled. The four flag lines EF are sampled in each S1 state, and the logic level is latched, as shown, in the middle of the TPA signal.

The Q line is also set or reset in the S1 state and is valid from the middle of clock pulse #3. The same timing applies to the INTERRUPT ENABLE Flip-Flop (IE), which is under program control.

The user-generated signals DMA-IN and DMA-OUT are sampled during clock pulse #6 in S1, S2, or S3 and latched during the middle of the TPB signal. INTERRUPT is accepted in the S1, S2 state also during the middle of the TPB signal.

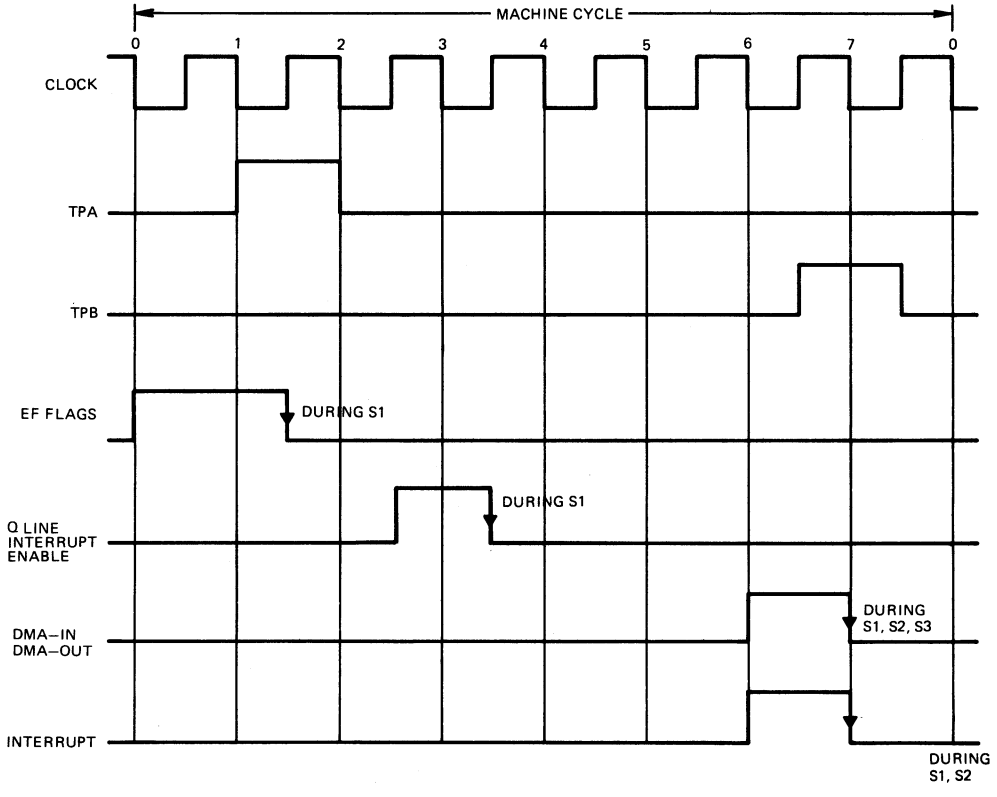


Fig. 116 – Internal sampling of some CPU and user-generated signals.

State of Data Bus and Memory Address Bus

In some system designs or during the debugging of a program it is frequently helpful to know what information is on the data bus or the memory address bus. A summary of this information is given in Table I for the various machine states and instructions. Some examples of what information can be derived from this tabulation are given below.

During a fetch cycle (S0), the memory content pointed to by the program counter is on the data bus while the content of the program counter R(P) is on the memory address bus.

During the execute cycle (S1), depending on which instruction is being executed, either bus will carry information. For instance, during an increment or decrement instruction, the data bus floats and the address bus contains the value of register R(N). If a short branch instruction is executed, the data bus has the value of the byte in memory following the branch instruction, i.e., the branch address. This address is present regardless whether the branch condition is met or not met. The address bus displays the content of the program counter pointing to the branch address in memory. Similarly, this value of the program counter is the same regardless of the branch condition. The distinction as to whether a branch condition is met or not met is internal to the CPU. In one case, the program counter is incremented; in the other, the branch address is jammed into the lower-order byte of the program counter.

For a long branch instruction (CN), the interpretation is similar. The branch address, however, now has two bytes, and all C instructions require two execute cycles. During the first execute cycle of a conditional long branch instruction, the data bus contains the higher-order byte of the branch address. During the next execute cycle, the data bus contains the lower-order byte of the branch address.

The MARK instruction (79) puts the content of the temporary register T on the data bus during execute and the content of R(2) on the address bus.

In the case of the SEP and SEX instructions, the byte on the data bus contains the value N in both the high-order and low-order nibble.

After RESET when $\overline{\text{CLEAR}}$ goes high, i.e., RUN mode, an S1 initialization cycle follows during which the data bus is 0. The content of R(0).0 is 0, but R(0).1 is determined by the CPU's previous history. The first fetch, however, will be from memory location 0000.

During RESET a similar situation exists for the two buses.

If the CPU fetches the IDLE instruction (00) during RUN, it will "idle" in a sequence of execute states (S1) until there is a DMA or INTERRUPT action. During the idle mode, the memory byte addressed by R(0) is on the data bus during each machine cycle, and the address bus displays R(0).

If the CPU is in the LOAD mode, it suppresses TPA and enters the S1 state waiting for DMA. (Note that the user should not activate INTERRUPT during LOAD.) When a LOAD cycle is next activated by DMA, data on the bus goes to M(R(0)) and R(0) is incremented. Unless DMA is asserted again when the CPU is in the LOAD mode, the CPU reverts to the S1 state and is in a "Read after Write" operation. The data bus displays the last byte loaded during S2 and the address bus has the last address used for writing into memory.

TABLE I. CONDITIONS ON DATA BUS AND MEMORY ADDRESS LINES DURING ALL MACHINE STATES

STATE	I	N	MNEMONIC	INSTRUCTION	OPERATION	DATA BUS	MEMORY ADDRESS	$\overline{MRD}$	NOTES G
S1	RESET				JAM: I,N,Q,X,P=0 IE=1	0	R (0) UNDEFINED	1	A
	FIRST CYCLE AFTER RESET NOT PROGRAMMER ACCESSIBLE				INITIALIZE	0	R (0) UNDEFINED	1	B
S0	FETCH				M(R(P))→I,N R(P)+1	M(R(P))	R(P)	0	C
S1 (Execute)	0	0	IDL	IDLE	[Load = 0 (Program Idle)]	M (R(0))	R (0)	0	D, 3
					[Load = 1 (Load Mode)]	M(R(0))	PREVIOUS ADDRESS	0	E, 3
		N≠0	LDN	LOAD D VIA N	M(R(N))→D	M(R(N))	R(N)	0	3
	1	N	INC	INCREMENT	R(N)+1	FLOAT	R(N)	1	1
	2	N	DEC	DECREMENT	R(N)−1	FLOAT	R(N)	1	1
	3	N	–	SHORT BRANCH	[BRANCH NOT TAKEN]	M(R(P))	R(P)	0	3
					[BRANCH TAKEN]	M(R(P))	R(P)	0	
	4	N	LDA	LOAD ADVANCE	M(R(N))→D R(N)+1	M(R(N))	R(N)	0	3
	5	N	STR	STORE VIA N	D→M(R(N))	D	R(N)	1	3
	6	0	IRX	INC REG X	R(X)+1	M(R(X))	R(X)	0	3
		N=1–7	OUT N	OUTPUT	M(R(X))→BUS R(X)+1	M(R(X))	R(X)	0	6
		N=9–F	INP N	INPUT	BUS→M(R(X)), D	I/O DEVICE	R(X)	1	5
	7	0	RET	RETURN	M(R(X))→(X,P) R(X)+1; 1→IE	M(R(X))	R(X)	0	3
		1	DIS	DISABLE	M(R(X))→(X,P) R(X)+1; 0→IE	M(R(X))	R(X)	0	3
		2	LDXA	LOAD VIA X AND ADVANCE	M(R(X))→D P(X)−1	M(R(X))	R(X)	0	3
		3	STXD	STORE VIA X AND DECREMENT	D→M(R(X)) R(X)−1	D	R(X)	1	2
		4,5,7	–		ALU OPERATION	M(R(X))	R(X)	0	3
		6	–		ALU OPERATION	FLOAT	R(X)	1	1
		8	SAV	SAVE	T→M(R(X))	T	R(X)	1	2
		9	MARK	MARK	(X,P)→T, M(R(2)) P→X; R(2)−1	T	R(2)	1	2
		A	REQ	RESET Q	Q=0	FLOAT	R(P)	1	1
		B	SEQ	SET Q	Q=1	FLOAT	R(P)	1	1
		C,D,F			ALU OPERATION IMMEDIATE	M(R(P))	R(P)	0	3
		E			ALU OPERATION	FLOAT	R(X)	1	1
	8	N	GLO	GET LOW	R(N) .0→D	R(N) .0	R(N)	1	1
	9	N	GHI	GET HIGH	R(N) .1→D	R(N) .1	R(N)	1	1
	A	N	PLO	PUT LOW	D→R(N).0	D	R(N)	1	1
	B	N	PHI	PUT HIGH	D→R(N) .1	D	R(N)	1	1
	C	0,1,2 3,8,9 A,B		LONG BRANCH	[BRANCH NOT TAKEN]	M(R(P))	R(P)	0	4
					[BRANCH TAKEN]	M(R(P))	R(P)	0	4
		5,6,7 C,D,E F		LONG SKIP	[SKIP NOT TAKEN]	M(R(P))	R(P)	0	4
					[SKIP TAKEN]	M(R(P))	R(P)	0	4
		4	NOP	NO OPERATION	NO OPERATION	M(R(P))	R(P)	0	4
	D	N	SEP	SET P	N→P	N N	R(N)	1	1
	E	N	SEX	SET X	N→X	N N	R(N)	1	1
	F	0	LDX	LOAD VIA X	M(R(X))→D	M(R(X))	R(X)	0	3
		1,2,3 4,5,7			ALU OPERATION	M(R(X))	R(X)	0	3
		6	SHR	SHIFT RIGHT	SHIFT D RIGHT LSB(D)→DF 0→MSB(D)	FLOAT	R(X)	1	1
		8	LDI	LOAD IMMEDIATE	M(R(P))→D R(P)+1	M(R(P))	R(P)	0	3
		9,A,B C,D,F			ALU OPERATION IMMEDIATE	M(R(P))	R(P)	0	3
		E	SHL	SHIFT LEFT	ALU OPERATION	FLOAT	R(P)	1	1
S2	IN REQUEST			DMA IN	BUS→M(R(0))	I/O DEVICE	R (0)	1	F, 7
	OUT REQUEST			DMA OUT	M(R(0))→BUS	M(R(0))	R (0)	0	F, 3
S3	INTERRUPT				X,P→T, 0→IE 2→X, 1→P	FLOAT	R(N)	1	9

NOTES:

A. IE = 1; TPA, TPB suppressed, state = S1
B. BUS = 0 for entire cycle
C. Next state always S1
D. Wait for DMA or INTERRUPT
E. Suppress TPA, wait for DMA
F. IN REQUEST has priority over OUT REQUEST

G. NUMBERS REFER TO MACHINE CYCLES TYPES — REFER TO TIMING DIAGRAMS, FIGS 16 THRU 20, ON THE ELEVEN SHEETS FROM "FILE NO. 1023". (LOOSE, IN REAR OF THIS BOOK)

Applications - Sample Programs

Two sample programs are included in this section to illustrate the use of some of the preceding instructions and techniques and to demonstrate the ease with which they can be used to develop programs. The examples show programs for processing two input bytes and for controlling a microcomputer-driven scale.

Processing Two Input Bytes

This program inputs two bytes from two different devices. These devices might be the outputs from two analog-to-digital converters or mechanical position resolvers. The program compares the digital inputs and, if they are equal, sets the Q flag to "1." In the event the two bytes are unequal, the Q flag is set to "0" and the larger of the two values is outputted to a third device. A minor change to this program could have it outputting the difference between the two bytes, an indication perhaps of the degree of mechanical "error."

The overview operation of this program is given in the flow chart in Fig. 117. A more detailed flow chart corresponding to the actual implementation is given in Fig. 118. This flow chart more closely corresponds to the assembled program listing shown in Fig. 119.

A few programming techniques used in this program warrant special attention.

1. The INITIALIZATION block in Fig. 117 becomes two blocks in Fig. 118. A portion of the original initialization block is done only once during the execution of the program. The other part is done every time the program loops back to the label GO. This arrangement was done to save memory at the expense of execution time, a common trade-off. The output instruction increments R(X) each time it is executed. To maintain R(X) pointing to the same memory location, it could be followed by a DEC R2. However, the execution of the LDI

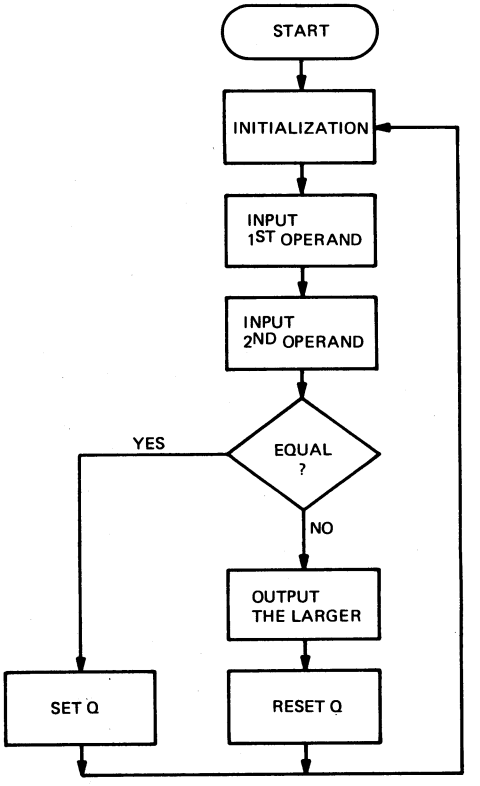


Fig. 117 — Program flow chart for processing two input bytes: inputting two bytes, comparing them, outputting the larger, and setting Q to "1" if equal.

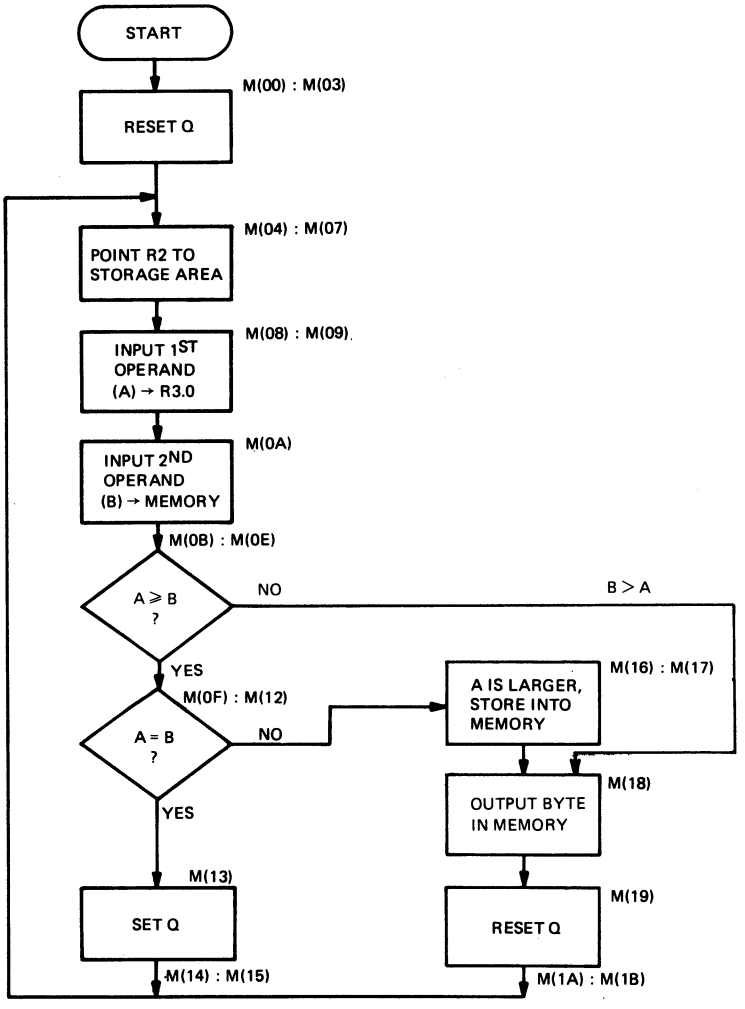


Fig. 118 — Detailed program flow chart for processing two input bytes.

and PLO (lines 4 and 5), which are already in the program, serve the same purpose. The decrement instruction, therefore, is purposely omitted.

2. Lines 8 through 10 use the characteristics of the input instruction to advantage. Because the data byte goes into both memory and D, the first input instruction is followed by the storing of the data from D into a scratch-pad register. The second input instruction utilizes the feature that the data byte also goes into memory. After the retrieval of the first byte from the scratch pad, the contents of D and memory are ready for comparison.

Microcomputer Scale

This example shows a program for a price-calculating scale. It reads the unit price from an input device such as a keyboard, reads the weight of the weighed item from the scale mechanism, multiplies the numbers to produce the total price, and then displays the total price to the customer. An overall flow chart for this program is given in Fig. 120. This flow chart illustrates an effective approach to the solution of moderate to large programming tasks. Each of the basic actions (initialization, input, calculation, and display) is treated as an independent block of code and coded as a subroutine. The final application program is then a collection of the subroutines flowcharted in this figure.

The flow charts for the individual subroutines are shown in Figs. 121, 122, 123, and 124. In addition, the flow chart of the calculation subroutine is done in sufficient detail for comparison with its assembly-language listing, part of which is shown in Fig. 125. The calculation subroutine is implemented with the RCA COSMAC Arithmetic Subroutine Package that contains independent 16-bit addition, subtraction, multiplication (giving 32 bits), division (from 32 bits), and BCD conversions to and from binary. Further details on this package are available in Manual MPM-206.

```
0000  7A;          0001              REQ             .. RESET Q TO "0"
0001  F800;        0002              LDI A.1 (STORE) .. SET STORAGE POINTER R(2)
0003  B2;          0003              PHI R2          .... TO POINT AT A FREE LOCATION
0004  F81C;        0004 GO:          LDI A.0 (STORE) .... IN RAM M(STORE)
0006  A2;          0005              PLO R2          .... " "
0007  E2;          0006              SEX R2
0008  69;          0007              INP 1           .. READ 1ST INPUT BYTE INTO D
0009  A3;          0008              PLO R3          .. SAVE THE 1ST INPUT
000A  6A;          0009              INP 2           .. READ 2ND INPUT BYTE INTO
000B  ;            0010                              .... MEMORY
000B  83;          0011              GLO R3          .. LOAD THE 1ST INPUT INTO D
000C  F7;          0012              SM              .. 1ST INPUT MINUS 2ND INPUT
000D  3B18;        0013              BNF RES2        .. BRANCH TO RES2 IF 2ND INPUT
000F  ;            0014                              .... IS GREATER THAN 1ST INPUT;
000F  ;            0015                              .... OTHERWISE:
000F  83;          0016              GLO R3          .. LOAD THE 1ST INPUT INTO D
0010  F3;          0017              XOR             .. M(R(2)).XOR.D, TO CHECK IF THE
0011  ;            0018                              .... TWO INPUTS ARE EQUAL
0011  3A16;        0019              BNZ RES1        .. BRANCH TO RES1 IF NOT EQUAL
0013  ;            0020                              .... (1ST INPUT IS GREATER THAN
0013  ;            0021                              .... 2ND INPUT); OTHERWISE:
0013  7B;          0022              SEQ             .. EQUAL; SET Q FLAG
0014  3004;        0023              BR  GO          .. GO BACK TO BEGINNING
0016  ;            0024
0016  83;          0025 RES1:        GLO R3          .. LOAD 1ST INPUT INTO D
0017  52;          0026              STR R2          .. STORE IT AT M(STORE)
0018  ;            0027
0018  61;          0028 RES2:        OUT 1           .. OUTPUT LARGER VALUE
0019  7A;          0029              REQ             .. RESET Q FLAG
001A  3004;        0030              BR  GO          .. GO BACK TO BEGINNING
001C  ;            0031
001C  ;            0032 STORE:       ORG *           .. STORAGE AREA
001C  ;            0033
001C  ;            0034
001C  ;            0035
001C  ;            0036
001C  ;            0037              END             .. END OF PROGRAM SOURCE
0000
```

Fig. 119 — Assembly listing for two-byte processing program.

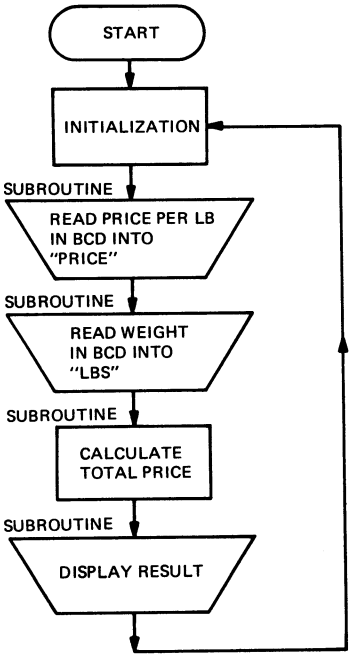


Fig. 120 — Over-all program flow chart for microcomputer scale.

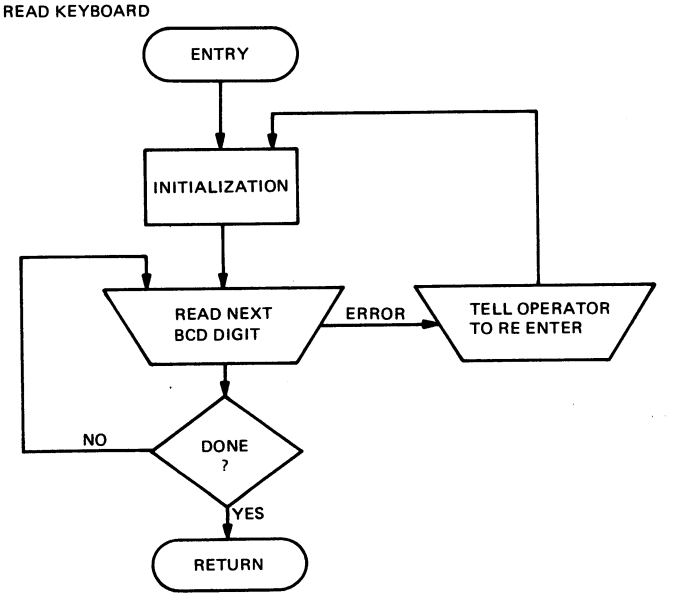


Fig. 121 — Program flow chart for keyboard subroutine.

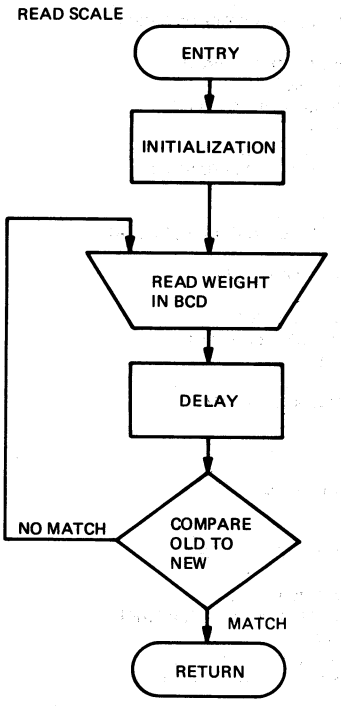


Fig. 122 — Program flow chart for scale subroutine.

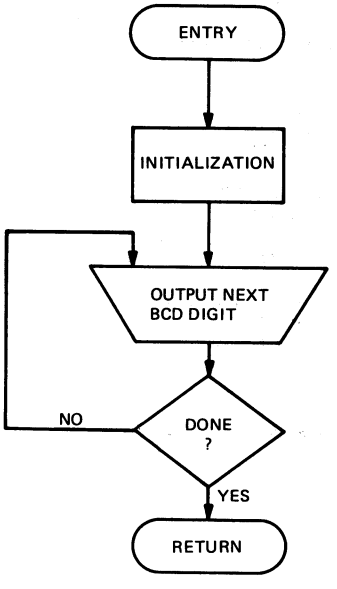


Fig. 123 — Program flow chart for display subroutine.

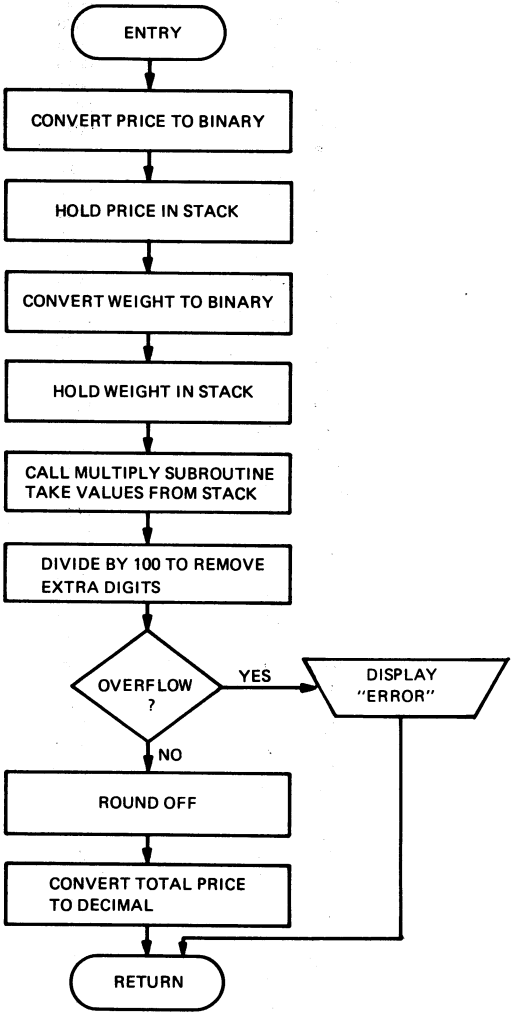


Fig. 124 — Subroutine flow chart for calculating total price.

```
0215   ;                0037
0215   D4;              0038          SEP CALL . . DO DECIMAL TO BINARY CONVERSION
0216   076E;            0039          ,A(CDB)  . . "
0218   027D;            0040          ,A(PRICE) . . CONVERT PRICE INTO BINARY
021A   05;              0041          ,#05  . . PRICE IS 5 CHARS LONG
021B   ;                0042                . . . AC NOW CONTAINS BINARY VALUE
021B   ;                0043                 . . OF PRICE PER POUND
021B   ;                0044          .....
021B   D4;              0045          SEP CALL . . PUSH CONTENT OF AC INTO STACK
021C   06E9;            0046          ,A(PUSHAC) . . "
021E   ;                0047          .....
021E   D4;              0048          SEP CALL . . DO DECIMAL TO BINARY CONVERSION
021F   076E;            0049          ,A(CDB) . . "
0221   0282;            0050          ,A(LBS)  . . CONVERT QUANTITY (LBS)
0223   ;                0051                . . . INTO BINARY
0223   05;              0052          ,#05 . . QUANTITY IS 5 CHARS LONG
0224   ;                0053               . . AC NOW CONTAINS BINARY VALUE
0224   ;                0054               . . OF QUANTITY (LBS)
0224   ;                0055          .....
0224   82;              0056          GLO SP . . COPY STACK POINTER TO MA
0225   AD;              0057          PLO MA . . "
0226   1D;              0058          INC MA . . POINT TO AC. 1
0227   ;                0059          .....
0227   D4;              0060          SEP CALL . . DO THE MULTIPLICATION
0228   0475;            0061          ,A(MPY) . . "
022A   ;                0062          .....
022A   ;                0063          . . . DIVIDE BY 100 TO REMOVE LAST TWO
022A   ;                0064          . . . DECIMAL DIGITS
022A   12;              0065          INC  SP      . . MOVE SP TWO BYTES
022B   12;              0066          INC  SP      . . . BELOW TOP OF STACK, POP PRICE
022C   ;                0067                       . . . OFF STACK
022C   F864;            0068          LDI  100     . . LOAD 100 INTO STACK WITH
022E   52;              0069          STR  SP      . . . SP POINTING TO THE HIGH BYTE
022F   22;              0070          DEC  SP      . . . "
0230   F800;            0071          LDI  00      . . . "
0232   52;              0072          STR  SP      . . . "
0233   82;              0073          GLO  SP     . . COPY STACK POINTER TO MA
0234   AD;              0074          PLO  MA     . . POINT TO HIGH BYTE OF 100
0235   22;              0075          DEC  SP     . . POINT TO FREE SPACE
0236   D4;              0076          SEP CALL
0237   051E;            0077          ,A(DIV) . . DIVIDE PRODUCT BY 100
0239   334A;            0078          BDF LAB3 . . IF OVERFLOW GO TO LAB3
023B   ;                0079          .....
023B   ;                0080          . . . CHECK IF REMAINDER IS GREATER
023B   ;                0081          . . . THAN 50, IF SO, ROUND UP
023B   8E;              0082          GLO MQ . . MQ CONTAINS THE REMAINDER
023C   FF32;            0083          SMI  50
023E   3B41;            0084          BNF  LAB1    . . IF NO ROUND UP
0240   ;                0085                       . . . GO TO LAB1
0240   1F;              0086          INC AC  . . IF ROUND UP ADD 1 TO
0241   ;                0087                . . . THE LEAST SIGNIFICANT DIGIT
0241   ;                0088          .....
0241   12;              0089  LAB1:   INC SP . . MOVE SP DOWN TWO BYTES
0242   12;              0090          INC SP . . . BELOW TOP OF STACK
0243   ;                0091          .....
0243   D4;              0092          SEP CALL . . DO BINARY TO DECIMAL CONVERSION
0244   0645;            0093          ,A(CBD)  . . "
0246   0287;            0094          ,A(TPR)  . . CONVERT TOTAL PRICE INTO DECIMAL
0248   06;              0095          ,#06 . . TOTAL PRICE IS 6 CHARS LONG
0249   ;                0096               . . TOTAL PRICE IS STORED IN M(TPR)
0249   ;                0097
```

Fig. 125 — Partial assembly-language listing of the calculation subroutine.

Appendix A — Instruction Summary

The COSMAC instruction summary is given in Tables I and II. Hexadecimal notation is used to refer to the 4-bit binary codes.

In all registers bits are numbered from the least significant bit (LSB) to the most significant bit (MSB) starting with 0.

R(W): Register designated by W, where W=N or X, or P

R(W).0: Lower-order byte of R(W)
R(W).1: Higher-order byte of R(W)
N0 = Least significant Bit of N Register

Operation Notation

M(R(N)) → D; R(N) + 1

This notation means: The memory byte pointed to by R(N) is loaded into D, and R(N) is incremented by 1.

TABLE I — INSTRUCTION SUMMARY
by Class of Operation

Register Operations

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
INCREMENT REG N	INC	1N	R(N) +1
DECREMENT REG N	DEC	2N	R(N) -1
INCREMENT REG X	IRX	60	R(X) +1
GET LOW REG N	GLO	8N	R(N).0→D
PUT LOW REG N	PLO	AN	D→R(N).0
GET HIGH REG N	GHI	9N	R(N).1→D
PUT HIGH REG N	PHI	BN	D→R(N).1

Memory Reference

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
LOAD VIA N	LDN	0N	M(R(N))→D; FOR N NOT 0
LOAD ADVANCE	LDA	4N	M(R(N))→D;R(N) +1
LOAD VIA X	LDX	F0	M(R(X))→D
LOAD VIA X AND ADVANCE	LDXA	72	M(R(X))→D; R(X) +1
LOAD IMMEDIATE	LDI	F8	M(R(P))→D; R(P) + 1
STORE VIA N	STR	5N	D→M(R(N))
STORE VIA X AND DECREMENT	STXD	73	D→M(R(X)); R(X) -1

Logic Operations♦♦

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
OR	OR	F1	M(R(X)) OR D→D
OR IMMEDIATE	ORI	F9	M(R(P)) OR D→D; R(P) +1
EXCLUSIVE OR	XOR	F3	M(R(X)) XOR D→D
EXCLUSIVE OR IMMEDIATE	XRI	FB	M(R(P)) XOR D→D; R(P) +1
AND	AND	F2	M(R(X)) AND D→D
AND IMMEDIATE	ANI	FA	M(R(P)) AND D→D; R(P) +1
SHIFT RIGHT	SHR	F6	SHIFT D RIGHT, LSB(D)→DF, 0→MSB(D)
SHIFT RIGHT WITH CARRY	SHRC	76♦	SHIFT D RIGHT, LSB(D)→DF, DF→MSB(D)
RING SHIFT RIGHT	RSHR		
SHIFT LEFT	SHL	FE	SHIFT D LEFT, MSB(D)→DF, 0→LSB(D)
SHIFT LEFT WITH CARRY	SHLC	7E♦	SHIFT D LEFT, MSB(D)→DF, DF→LSB(D)
RING SHIFT LEFT	RSHL		

♦NOTE: THIS INSTRUCTION IS ASSOCIATED WITH MORE THAN ONE MNEMONIC. EACH MNEMONIC IS INDIVIDUALLY LISTED.
♦♦NOTE: THE ARITHMETIC OPERATIONS AND THE SHIFT INSTRUCTIONS ARE THE ONLY INSTRUCTIONS THAT CAN ALTER THE DF.

Arithmetic Operations◆◆

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
ADD	ADD	F4	M(R(X)) +D→DF, D
ADD IMMEDIATE	ADI	FC	M(R(P)) +D→DF, D; R(P) +1
ADD WITH CARRY	ADC	74	M(R(X)) +D +DF→DF, D
ADD WITH CARRY, IMMEDIATE	ADCI	7C	M(R(P)) +D +DF→DF, D R(P) +1
SUBTRACT D	SD	F5	M(R(X))−D→DF, D
SUBTRACT D IMMEDIATE	SDI	FD	M(R(P))−D→DF, D; R(P) +1
SUBTRACT D WITH BORROW	SDB	75	M(R(X))−D−(NOT DF)→DF, D
SUBTRACT D WITH BORROW, IMMEDIATE	SDBI	7D	M(R(P))−D−(NOT DF)→DF, D; R(P) +1
SUBTRACT MEMORY	SM	F7	D−M(R(X))→DF, D
SUBTRACT MEMORY IMMEDIATE	SMI	FF	D−M(R(P))→DF, D; R(P) +1
SUBTRACT MEMORY WITH BORROW	SMB	77	D−M(R(X))−(NOT DF)→DF, D
SUBTRACT MEMORY WITH BORROW, IMMEDIATE	SMBI	7F	D−M(R(P))−(NOT DF)→DF, D R(P) +1

Branch Instructions — Short Branch

SHORT BRANCH	BR	30	M(R(P))→R(P).0
NO SHORT BRANCH (SEE SKP)	NBR	38◆	R(P) +1
SHORT BRANCH IF D=0	BZ	32	IF D=0, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF D NOT 0	BNZ	3A	IF D NOT 0, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF DF=1	BDF	33◆	IF DF=1, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF POS OR ZERO	BPZ		
SHORT BRANCH IF EQUAL OR GREATER	BGE		
SHORT BRANCH IF DF=0	BNF	3B◆	IF DF=0, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF MINUS	BM		
SHORT BRANCH IF LESS	BL		
SHORT BRANCH IF Q=1	BQ	31	IF Q=1, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF Q=0	BNQ	39	IF Q=0, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF1=1 (1 = V_{SS})	B1	34	IF EF1=1, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF1=0 (0 = V_{CC})	BN1	3C	IF EF1=0, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF2=1 (1 = V_{SS})	B2	35	IF EF2=1, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF2=0 (0 = V_{CC})	BN2	3D	IF EF2=0, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF3=1 (1 = V_{SS})	B3	36	IF EF3=1, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF3=0 (0 = V_{CC})	BN3	3E	IF EF3=0, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF4=1 (1 = V_{SS})	B4	37	IF EF4=1, M(R(P))→R(P).0 ELSE R(P) +1
SHORT BRANCH IF EF4=0 (0 = V_{CC})	BN4	3F	IF EF4=0, M(R(P))→R(P).0 ELSE R(P) +1

Branch Instructions — Long Branch

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
LONG BRANCH	LBR	C0	M(R(P))→R(P).1 M(R(P) +1)→R(P).0
NO LONG BRANCH (SEE LSKP)	NLBR	C8♦	R(P) +2
LONG BRANCH IF D=0	LBZ	C2	IF D=0, M(R(P))→R(P).1 M(R(P) +1)→R(P).0 ELSE R(P) +2
LONG BRANCH IF D NOT 0	LBNZ	CA	IF D NOT 0, M(R(P))→ R(P).1 M(R(P) +1)→ R(P).0 ELSE R(P) +2
LONG BRANCH IF DF=1	LBDF	C3	IF DF=1, M(R(P))→R(P).1 M(R(P) +1)→ R(P).0 ELSE R(P) +2
LONG BRANCH IF DF=0	LBNF	CB	IF DF=0, M(R(P))→R(P).1 M(R(P) +1)→ R(P).0 ELSE R(P) +2
LONG BRANCH IF Q=1	LBQ	C1	IF Q=1, M(R(P))→R(P).1 M(R(P) +1)→R(P).0 ELSE R(P) +2
LONG BRANCH IF Q=0	LBNQ	C9	IF Q=0, M(R(P))→R(P).1 M(R(P) +1)→ R(P).0 ELSE R(P) +2

Skip Instructions

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
SHORT SKIP (SEE NBR)	SKP	38♦	R(P) +1
LONG SKIP (SEE NLBR)	LSKP	C8♦	R(P) +2
LONG SKIP IF D=0	LSZ	CE	IF D=0, R(P) +2 ELSE CONTINUE
LONG SKIP IF D NOT 0	LSNZ	C6	IF D NOT 0, R(P) +2 ELSE CONTINUE
LONG SKIP IF DF=1	LSDF	CF	IF DF=1, R(P) +2 ELSE CONTINUE
LONG SKIP IF DF=0	LSNF	C7	IF DF=0, R(P) +2 ELSE CONTINUE
LONG SKIP IF Q=1	LSQ	CD	IF Q=1, R(P) +2 ELSE CONTINUE
LONG SKIP IF Q=0	LSNQ	C5	IF Q=0, R(P) +2 ELSE CONTINUE
LONG SKIP IF IE=1	LSIE	CC	IF IE=1, R(P) +2 ELSE CONTINUE

♦NOTE: THIS INSTRUCTION IS ASSOCIATED WITH MORE THAN ONE MNEMONIC. EACH MNEMONIC IS INDIVIDUALLY LISTED.

♦♦NOTE: THE ARITHMETIC OPERATIONS AND THE SHIFT INSTRUCTIONS ARE THE ONLY INSTRUCTIONS THAT CAN ALTER THE DF.

Control Instructions

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
IDLE	IDL	00	WAIT FOR DMA OR INTERRUPT; M(R(0))→BUS
NO OPERATION	NOP	C4	CONTINUE
SET P	SEP	DN	N→P
SET X	SEX	EN	N→X
SET Q	SEQ	7B	1→Q
RESET Q	REQ	7A	0→Q
SAVE	SAV	78	T→M(R(X))
PUSH X,P TO STACK	MARK	79	(X,P)→T; (X,P)→M(R(2)) THEN P→X; R(2)−1
RETURN	RET	70	M(R(X))→(X,P); R(X) +1 1→IE
DISABLE	DIS	71	M(R(X))→(X,P); R(X) +1 0→IE

Input–Output Byte Transfer

INSTRUCTION	MNEMONIC	OP CODE	OPERATION
OUTPUT 1	OUT 1	61	M(R(X))→BUS; R(X) +1; N LINES = 1
OUTPUT 2	OUT 2	62	M(R(X))→BUS; R(X) +1; N LINES = 2
OUTPUT 3	OUT 3	63	M(R(X))→BUS; R(X) +1; N LINES = 3
OUTPUT 4	OUT 4	64	M(R(X))→BUS; R(X) +1; N LINES = 4
OUTPUT 5	OUT 5	65	M(R(X))→BUS; R(X) +1; N LINES = 5
OUTPUT 6	OUT 6	66	M(R(X))→BUS; R(X) +1; N LINES = 6
OUTPUT 7	OUT 7	67	M(R(X))→BUS; R(X) +1; N LINES = 7
INPUT 1	INP 1	69	BUS→M(R(X)); BUS→D; N LINES = 1
INPUT 2	INP 2	6A	BUS→M(R(X)); BUS→D; N LINES = 2
INPUT 3	INP 3	6B	BUS→M(R(X)); BUS→D; N LINES = 3
INPUT 4	INP 4	6C	BUS→M(R(X)); BUS→D; N LINES = 4
INPUT 5	INP 5	6D	BUS→M(R(X)); BUS→D; N LINES = 5
INPUT 6	INP 6	6E	BUS→M(R(X)); BUS→D; N LINES = 6
INPUT 7	INP 7	6F	BUS→M(R(X)); BUS→D; N LINES = 7

♦NOTE: THIS INSTRUCTION IS ASSOCIATED WITH MORE THAN ONE MNEMONIC. EACH MNEMONIC IS INDIVIDUALLY LISTED.
♦♦NOTE: THE ARITHMETIC OPERATIONS AND THE SHIFT INSTRUCTIONS ARE THE ONLY INSTRUCTIONS THAT CAN ALTER THE DF.

TABLE II — INSTRUCTION SUMMARY
By Numerical Order

OPERATION CODE	OPERAND	MNEMONIC	NAME	MACHINE CYCLE	NUMBER OF PROGRAM BYTES
00	—	IDL	IDLE	2	1
0N	REG N	LDN	LOAD VIA N	2	1
1N	REG N	INC	INCREMENT REG N	2	1
2N	REG N	DEC	DECREMENT REG N	2	1
30	ADDRESS	BR	SHORT BRANCH	2	2
31	ADDRESS	BQ	SHORT BRANCH IF Q=1	2	2
32	ADDRESS	BZ	SHORT BRANCH IF D=0	2	2
33	ADDRESS	BDF	SHORT BRANCH IF DF=1	2	2
—	ADDRESS	BPZ	SHORT BRANCH IF POS OR ZERO	2	2
—	ADDRESS	BGE	SHORT BRANCH IF EQUAL OR GREATER	2	2
34	ADDRESS	B1	SHORT BRANCH IF EF1=1	2	2
35	ADDRESS	B2	SHORT BRANCH IF EF2=1	2	2
36	ADDRESS	B3	SHORT BRANCH IF EF3=1	2	2
37	ADDRESS	B4	SHORT BRANCH IF EF4=1	2	2
38	ADDRESS	NBR	NO SHORT BRANCH	2	2
—		SKP	SHORT SKIP	2	1
39	ADDRESS	BNQ	SHORT BRANCH IF Q=0	2	2
3A	ADDRESS	BNZ	SHORT BRANCH IF D NOT 0	2	2
3B	ADDRESS	BNF	SHORT BRANCH IF DF=0	2	2
—	ADDRESS	BM	SHORT BRANCH IF MINUS	2	2
—	ADDRESS	BL	SHORT BRANCH IF LESS	2	2

INSTRUCTION SUMMARY (CONT'D)

OPERATION CODE	OPERAND	MNEMONIC	NAME	MACHINE CYCLES	NUMBER OF PROGRAM BYTES
3C	ADDRESS	BN1	SHORT BRANCH IF EF1=0	2	2
3D	ADDRESS	BN2	SHORT BRANCH IF EF2=0	2	2
3E	ADDRESS	BN3	SHORT BRANCH IF EF3=0	2	2
3F	ADDRESS	BN4	SHORT BRANCH IF EF4=0	2	2
4N	REG N	LDA	LOAD ADVANCE	2	1
5N	REG N	STR	STORE VIA N	2	1
60	–	IRX	INCREMENT REG X	2	1
61	DEVICE 1	OUT1	OUTPUT1	2	1
62	DEVICE 2	OUT2	OUTPUT2	2	1
63	DEVICE 3	OUT3	OUTPUT3	2	1
64	DEVICE 4	OUT4	OUTPUT4	2	1
65	DEVICE 5	OUT5	OUTPUT5	2	1
66	DEVICE 6	OUT6	OUTPUT6	2	1
67	DEVICE 7	OUT7	OUTPUT7	2	1
68			DO NOT USE		
69	DEVICE 1	INP1	INPUT1	2	1
6A	DEVICE 2	INP2	INPUT2	2	1
6B	DEVICE 3	INP3	INPUT3	2	1
6C	DEVICE 4	INP4	INPUT4	2	1
6D	DEVICE 5	INP5	INPUT5	2	1
6E	DEVICE 6	INP6	INPUT6	2	1
6F	DEVICE 7	INP7	INPUT7	2	1
70	–	RET	RETURN	2	1
71	–	DIS	DISABLE	2	1
72	–	LDXA	LOAD VIA X, ADVANCE	2	1
73	–	STXD	STORE VIA X AND DECREMENT	2	1
74	–	ADC	ADD WITH CARRY	2	1
75	–	SDB	SUBTRACT D WITH BORROW	2	1
76	–	SHRC	SHIFT RIGHT WITH CARRY	2	1
–	–	RSHR	RING SHIFT RIGHT	2	1
77	–	SMB	SUBTRACT MEMORY WITH BORROW	2	1
78	–	SAV	SAVE	2	1
79	–	MARK	PUSH X,P TO STACK	2	1
7A	–	REQ	RESET Q	2	1
7B	–	SEQ	SET Q	2	1
7C	DATA	ADCI	ADD WITH CARRY IMMEDIATE	2	2

INSTRUCTION SUMMARY (CONT'D)

OPERATION CODE	OPERAND	MNEMONIC	NAME	MACHINE CYCLES	NUMBER OF PROGRAM BYTES
7D	DATA	SDBI	SUBTRACT D WITH BORROW IMMEDIATE	2	2
7E	—	SHLC	SHIFT LEFT WITH CARRY	2	1
	—	RSHL	RING SHIFT LEFT	2	1
7F	DATA	SMBI	SUBTRACT MEMORY WITH BORROW, IMMEDIATE	2	2
8N	REG N	GLO	GET LOW REG N	2	1
9N	REG N	GHI	GET HIGH REG N	2	1
AN	REG N	PLO	PUT LOW REG N	2	1
BN	REG N	PHI	PUT HIGH REG N	2	1
C0	ADDRESS	LBR	LONG BRANCH	3	3
C1	ADDRESS	LBQ	LONG BRANCH IF Q=1	3	3
C2	ADDRESS	LBZ	LONG BRANCH IF D=0	3	3
C3	ADDRESS	LBDF	LONG BRANCH IF DF=1	3	3
C4	—	NOP	NO OPERATION	3	1
C5	—	LSNQ	LONG SKIP IF Q=0	3	1
C6	—	LSNZ	LONG SKIP IF D NOT 0	3	1
C7	—	LSNF	LONG SKIP IF DF=0	3	1
C8	—	LSKP	LONG SKIP	3	1
—	ADDRESS	NLBR	NO LONG BRANCH	3	3
C9	ADDRESS	LBNQ	LONG BRANCH IF Q=0	3	3
CA	ADDRESS	LBNZ	LONG BRANCH IF D NOT 0	3	3
CB	ADDRESS	LBNF	LONG BRANCH IF DF=0	3	3
CC	—	LSIE	LONG SKIP IF IE=1	3	1
CD	—	LSQ	LONG SKIP IF Q=1	3	1
CE	—	LSZ	LONG SKIP IF D=0	3	1
CF	—	LSDF	LONG SKIP IF DF=1	3	1
DN	REG N	SEP	SET P	2	1
EN	REG N	SEX	SET X	2	1
F0	—	LDX	LOAD VIA X	2	1
F1	—	OR	OR	2	1
F2	—	AND	AND	2	1

INSTRUCTION SUMMARY (CONT'D)

OPERATION CODE	OPERAND	MNEMONIC	NAME	MACHINE CYCLES	NUMBER OF PROGRAM BYTES
F3	–	XOR	EXCLUSIVE OR	2	1
F4	–	ADD	ADD	2	1
F5	–	SD	SUBTRACT D	2	1
F6	–	SHR	SHIFT RIGHT	2	1
F7	–	SM	SUBTRACT MEMORY	2	1
F8	DATA	LDI	LOAD IMMEDIATE	2	2
F9	DATA	ORI	OR IMMEDIATE	2	2
FA	DATA	ANI	AND IMMEDIATE	2	2
FB	DATA	XRI	EXCLUSIVE OR IMMEDIATE	2	2
FC	DATA	ADI	ADD IMMEDIATE	2	2
FD	DATA	SDI	SUBTRACT D IMMEDIATE	2	2
FE	–	SHL	SHIFT LEFT	2	1
FF	DATA	SMI	SUBTRACT MEMORY IMMEDIATE	2	2

Interpretation of DF

	DF	Carry Generated	Borrow Generated	D
After Addition	1	Yes		
	0	No		
After Subtraction	1		No	Positive Number
	0		Yes	Negative Number 2's complement

Hexadecimal Code

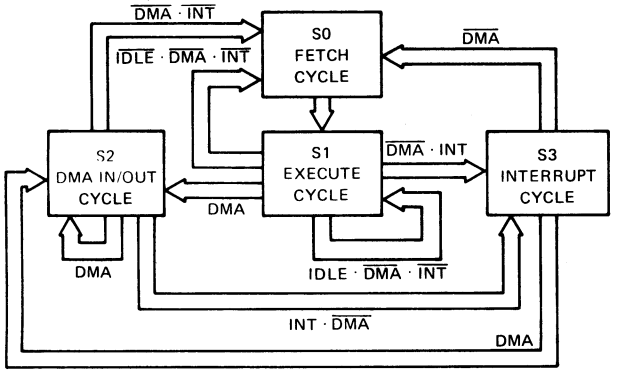
HEX	BINARY	HEX	BINARY
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

COSMAC Register Summary

D	8 Bits	Data Register (Accumulator)
DF	1 Bit	Data Flag (ALU Carry)
R	16 Bits	1 of 16 Scratchpad Registers
P	4 Bits	Designates which register is Program Counter
X	4 Bits	Designates which register is Data Pointer

N	4 Bits	Holds Low-Order Instr. Digit
I	4 Bits	Holds High-Order Instr. Digit
T	8 Bits	Holds old X, P after Interrupt (X is high byte)
IE	1 Bit	Interrupt Enable Flip Flop
Q	1 Bit	Output Flip Flop

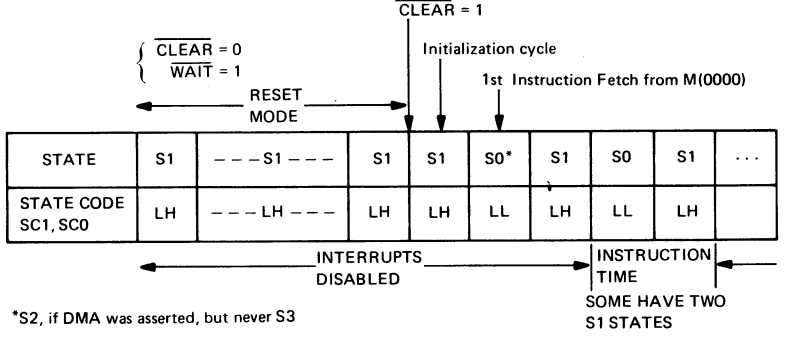
Appendix B – State Sequencing



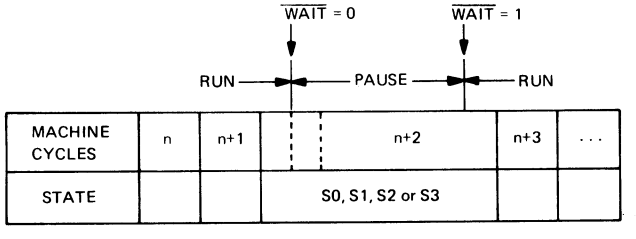
State Type	State Code Lines	
	SC1	SC0
S0 (Fetch)	L	L
S1 (Execute)	L	H
S2 (DMA)	H	L
S3 (Interrupt)	H	H

The CDP1802 state transitions when in the RUN mode. Each cycle requires the same period of time–8 clock pulses. The execution of an instruction requires either two or three machine cycles, S0 followed by a single S1 cycle or by two S1 cycles. S2 is the response to a DMA request and S3 is the interrupt response.

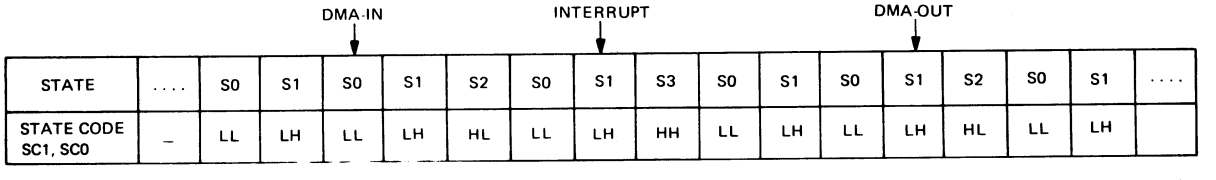
RESET → RUN MODE:



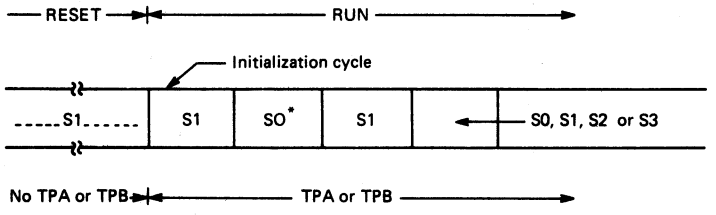
RUN → PAUSE → RUN MODE:



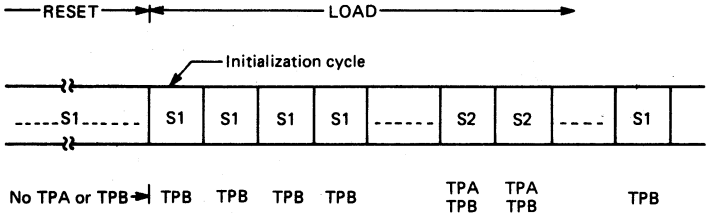
DMA AND INTERRUPT MODE:

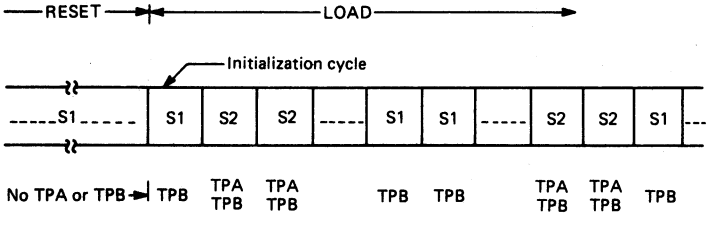


STATUS OF TPA AND TPB DURING RESET, RUN, OR LOAD:

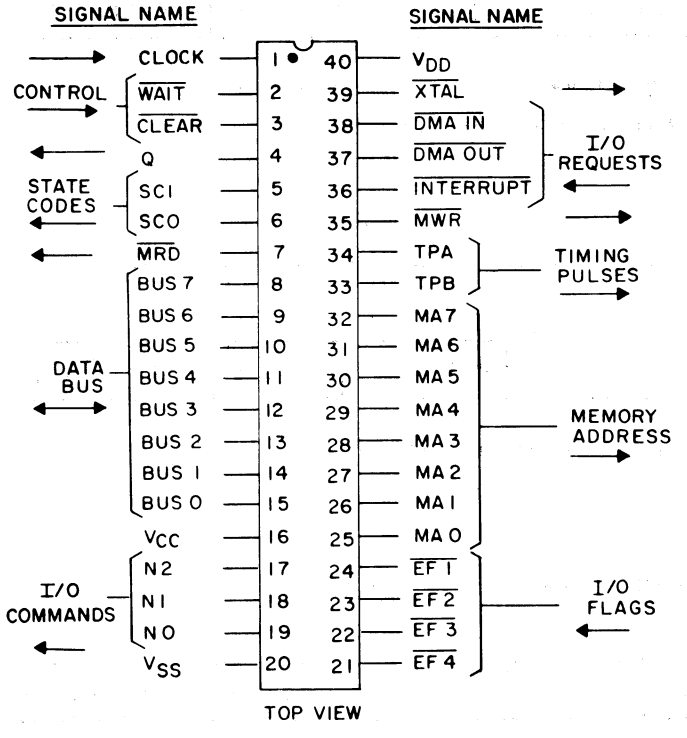


* S2, if DMA was asserted but never S3





Appendix C — Terminal Assignments for the RCA CDP1802 COSMAC Microprocessor



92CS-27467

Appendix D — COSMAC Dictionary

Access Time: Time between the instant that an address is sent to a memory and the instant that data returns. Since the access time to different locations (addresses) of the memory may be different, the access time specified in a memory device is the path which takes the longest time.

Accumulator: Register and related circuitry which holds one operand for arithmetic and logical operations.

Additional Hardware: Microprocessor chips differ in number of additional ICs required to implement a functioning computer. Generally, timing, I/O control, buffering, and interrupt control require external components.

Address: A number used by the CPU to specify a location in memory.

Addressing Modes: See Memory Addressing Modes

ALU: **Arithmetic-Logic Unit**. That part of a CPU which executes adds, subtracts, shifts, AND's, OR's, etc.

Architecture: Organizational structure of a computing system, mainly referring to the CPU or microprocessor.

Assembler: Software that converts an assembly-language program into machine language. The assembler assigns locations in storage to successive instructions and replaces symbolic addresses by machine language equivalents. If the assembler runs on a computer other than that for which it creates the machine language, it is a **Cross-Assembler**.

Assembly Language: An English-like programming language which saves the programmer the trouble of remembering the bit patterns in each instruction; also relieves him of the necessity to keep track of locations of data and instructions in his program.

The assembler operates on a "one-for-one" basis in that each phrase of the language translates directly into a specific machine-language word, as contrasted with **High Level Language**.

Assembly Listing: A printed listing made by the assembler to document an assembly. It shows, line for line, how the assembler interpreted the assembly language program.

Asynchronous Operation: Circuit operation without reliance upon a common timing source. Each circuit operation is terminated (and next operation initiated) by a return signal from the destination denoting completion of an operation. (Contrast with **Synchronous Operation**).

Baud: A communications measure of serial data transmission rate; loosely, bits per second but includes character-framing START and STOP bits.

Benchmark Program: A sample program used to evaluate and compare computers. In general, two computers will not use the same number of instructions, memory words, or cycles to solve the same problem.

Bit: An abbreviation of "binary digit". (Single characters in a binary number.)

Bootstrap (Bootstrap Loader): Technique or device for loading first instructions (usually only a few words) of a routine into memory; then using these instructions to bring in the rest of the routine.

The bootstrap loader is usually entered manually or by pressing a special console key. COSMAC does not need one. See **Load Facility**.

Branch: See **Jump**.

Branch Instruction: A decision-making instruction which, on appropriate condition, forces a new address into the program counter. The conditions may be zero result, overflow on add, an external flag raised, etc. One of two alternate program segments in the memory are chosen, depending on the results obtained.

Breakpoint: A location specified by the user at which program execution (real or simulated) is to terminate. Used to aid in locating program errors.

Bus: A group of wires which allow memory, CPU, and I/O devices to exchange words.

Byte: A sequence of n bits operated upon as a unit is called an n-bit byte. The most frequent byte size is 8 bits.

Call Routine: See **Subroutine**

Clock: A device that sends out timing pulses to synchronize the actions of the computer.

Compiler: Software to convert a program in a high-level language such as FORTAN into an assembly language or machine language program.

COSMAC: Generic description for the RCA family of compatible microprocessor products (1800 series). Based on a unique architecture, the COSMAC family includes CPU's, memories, I/O's, prototyping systems, and software.

COSMAC Development System (formerly "Microkit"): Microcomputer used for software development and system prototyping. Uses the COSMAC 1800 family of microprocessor products.

COSMAC Software Development Package (CSDP): An assembler and interactive debugger/simulator for COSMAC microcomputer systems. The debugger is a power-

ful "software oscilloscope" that allows the user to start and stop the simulator, examine and modify the program variables, and dump and restore the entire simulated machine at will. CSDP is available either on the General Electric Mark III Service or as a Fortran IV program that can be easily installed on a host computer.

Cross Assembler: A symbolic language translator that runs on one type of computer to produce machine code for another type of computer. See **Assembler.**

CPU (Central Processing Unit): That part of a computer system that controls the interpretation and execution of instructions. In general, the CPU contains the following elements:

Arithmetic-Logic Unit (ALU)
Timing and Control
Accumulator
Scratch-pad memory
Program counter and address stack
Instruction register and decode
Parallel data and I/O bus
Memory and I/O control

Cycle Stealing: A memory cycle stolen from the normal CPU operation for a DMA operation. See **DMA.**

Cycle Time: Time interval at which any set of operations is repeated regularly in the same sequence.

D Register: The accumulator in the COSMAC microprocessor.

Data Pointer: A register holding the memory address of the data (operand) to be used by an instruction. Thus the register "points" to the memory location of the data.

Data Register: Any register which holds data. In the COSMAC microprocessor, any one of the 16 x 16 scratch-pad registers can be used to hold two bytes of data.

Debug. To eliminate programming mistakes, including omissions, from a program.

Debug Programs: Debug programs help the programmer to find errors in his programs while they are running on the computer, and allow him to replace or patch instructions into (or out of) his program.

Designator: The three 4-bit registers P, X, and N in the COSMAC microprocessor are called designators. P and X are used to designate which one of the sixteen 16-bit scratch-pad registers is used as the current program counter and the data pointer, respectively.

N can designate: one of the scratch-pad registers; an I/O device or command; a new value in P or X; and a further definition of an instruction.

Diagnostic programs: These programs check the various hardware parts of a system for proper operation; CPU diagnostics check the CPU, memory diagnostics check the memory, and so forth.

Direct Addressing: The address of an instruction or operand is completely specified in an instruction without reference to a base register or index register.

DMA: Direct Memory Access. A mechanism which allows an input/output device to take control of the CPU for one or more memory cycles, in order to write to or read from memory. The order of executing the program steps (instructions) remains unchanged.

Editor: As an aid in preparing source programs, certain programs have been developed that manipulate text material. These programs, called editors, text editors, or paper tape editors make it possible to compose assembly language programs on-line, or on a stand-alone system.

Execute: The process of interpreting an instruction and performing the indicated operation(s).

Fetch: A process of addressing the memory and reading into the CPU the information word, or byte, stored at the addressed location. Most often, fetch refers to the reading out of an instruction from the memory.

Firmware: Software which is implemented in ROM's.

Fixed-instruction Computer (Stored-Instruction Computer): The instruction set of a computer is fixed by the manufacturer. The users will design application programs using this instruction set (in contrast to the **Micro-programmable Computer** for which the users must design their own instruction set and thus customize the computer for their needs.)

Fixed Memory: See ROM

Flag Lines: Inputs to a microprocessor controlled by I/O devices and tested by branch instructions.

Fortran: A high-level programming language generally for scientific use, expressed in algebraic notation. Short for "Formula Translator".

Guard: A mechanism to terminate program execution (real or simulated) upon access to data at a specified memory location. Used in debugging.

Hardware: Physical equipment forming a computer system.

Hexadecimal: Number system using 0, 1, ..., A, B, C, D, E, F to represent all the possible values of a 4-bit digit. The decimal equivalent is 0 to 15. Two hexadecimal digits can be used to specify a byte.

High-Level Language: Programming language which generates machine codes from problem- or function-oriented statements. FORTRAN, COBOL, and BASIC are three commonly used high-level languages. A single functional statement may translate into a series of instructions or subroutines in machine language, in contrast to a low-level (assembly) language in which statements translate on a one-for-one basis.

Immediate Addressing: The method of addressing an instruction in which the operand is located in the instruction itself or in the memory location immediately following the instruction.

Immediate Data: Data which immediately follows an instruction in memory, and is used as an operand by that instruction.

Indexed Addressing: An addressing mode, in which the

the address part of an instruction is modified by the contents in an auxiliary (index) register during the execution of that instruction.

Index Register: A register which contains a quantity which may be used to modify memory address.

Indirect Addressing: A means of addressing in which the address of the operand is specified by an auxiliary register or memory location specified by the instruction rather than by bits in the instruction itself.

Input-Output (I/O): General term for the equipment used to communicate with a computer CPU; or the data involved in that communication.

Instruction: A set of bits that defines a computer operation, and is a basic command understood by the CPU. It may move data, do arithmetic and logic functions, control I/O devices, or make decisions as to which instruction to execute next.

Instruction Cycle: The process of fetching an instruction from memory and executing it.

Instruction Length: The number of words needed to store an instruction. It is one word in most computers, but some will use multiple words to form one instruction. Multiple-word instructions have different instruction execution times depending on the length of the instruction.

Instruction Repertoire: See Instruction Set

Instruction Set: The set of general-purpose instructions available with a given computer. In general, different machines have different instruction sets.

The number of instructions only partially indicates the quality of an instruction set. Some instructions may only be slightly different from one another; others rarely may be used. Instruction sets should be compared using benchmark programs typical of the application, to determine execution times, and memory requirements.

Instruction Time: The time required to fetch an instruction from memory and then execute it.

Interpreter: A program which fetches and executes "instructions" **(pseudo instructions)** written in a higher level language. The higher-level language program is a **pseudo program**. Contrast with **Compiler**.

Interrupt Request: A signal to the computer that temporarily suspends the normal sequence of a routine and transfers control to a special routine. Operation can be resumed from this point later. Ability to handle interrupts is very useful in communication applications where it allows the microprocessor to service many channels.

Interrupt Mask (Interrupt Enable): A mechanism which allows the program to specify whether or not interrupt requests will be accepted.

Interrupt Service Routine: A routine (program) to properly store away to the stack the present status of the machine in order to respond to an interrupt request; perform the "real work" required by the interrupt; restore the saved status of the machine; and then resume the operation of the interrupted program.

I/O Control Electronics (I/O Controller): The control electronics required to interface an I/O device to a computer CPU.

The powerfulness and usefulness of a CPU is very closely associated with the range of I/O devices which can be connected to it. One can not usually simply plug them into the CPU. The I/O Control Electronics will do the "matchmaking". The complexity and cost of the Control Electronics are very much determined by both the hardware and software I/O architecture of the CPU.

I/O Interface: See I/O Control Electronics

I/O Port: A connection to a CPU which is configured (or programmed) to provide a data path between the CPU and the external devices, such as keyboard, display, reader, etc. An I/O port of a microprocessor may be an input port or an output port, or it may be bidirectional.

Jump: A departure from the normal one-step incrementing of the program counter. By forcing a new value (address) into the program counter the next instruction can be fetched from an arbitrary location (either further ahead or back).

For example, a program jump can be used to go from the main program to a subroutine, from a subroutine back to the main program, or from the end of a short routine back to the beginning of the same routine to form a loop. See also the Branch Instruction. If you reached this point from Branch, you have executed a Jump. Now Return.

Linkage: See Subroutine

Load Facility: A hardware facility to allow program loading using DMA. It makes bootstrap unnecessary.

Loader: A program to read a program from an input device into RAM. May be part of a package of utility programs.

Loop: A self-contained series of instructions in which the last instruction can cause repetition of the series until a terminal condition is reached. Branch instructions are used to test the conditions in the loop to determine if the loop should be continued or terminated.

Low-Level Language: See **Assembly Language**

Machine: A term for a computer (of historical origin).

Machine Code: See Machine Language

Machine Cycle: The basic CPU cycle. In one machine cycle an address may be sent to memory and one word (data or instruction) read or written, or, in one machine cycle a fetched instruction can be executed. One machine cycle in the COSMAC microprocessor consists of eight clock pulses.

Machine Language: The numeric form of specifying instructions, ready for loading into memory and execution

by the machine. This is the lowest-level language in which to write programs. The value of every bit in every instruction in the program must be specified (e.g., by giving a string of binary, octal, or hexadecimal digits for the contents of each word in memory).

Machine State: See **State Code**

Macro (Macroinstruction): A symbolic source language statement which is expanded by the assembler into one or more machine language instructions, relieving the programmer of having to write out frequently occuring instruction sequences.

Manufacturer's Support: It includes application information, software assistance, components for prototyping, availability of hardware in all configurations from chips to systems, and fast response to requests for engineering assistance.

Memory: That part of a computer which holds data and instructions. Each instructions or datum is assigned a unique address which is used by the CPU when fetching or storing the information.

Memory Address Register: The CPU register which holds the address of the memory location being accessed.

Memory Addressing Modes: The method of specifying the memory location of an operand. Common addressing modes are -- direct, immediate, relative, indexed, and indirect. These modes are important factors in program efficiency.

Microcomputer: A computer whose CPU is a microprocessor. A microcomputer is an entire system with microprocessor, memory, and input-output controllers.

Microprocessor: Frequently called "a computer on a chip". The microprocessor is, in reality, a set of one, or a few, LSI circuits capable of performing the essential functions of a computer CPU.

Microprogrammable Computer: A computer in which the internal CPU control signal sequence for performing instructions are generated from a ROM. By changing the ROM contents, the instruction set can be changed. This contrasts with a Fixed-Instruction Computer in which the instruction set can not be readily changed.

Microtutor: Inexpensive microcomputer for first-level hands-on experience with microprocessor hardware and programming. Comes complete with CPU, memory, input and output devices, and power supply.

Mnemonics: Symbolic names or abbreviations for instructions, registers, memory locations, etc. A technique for improving the efficiency of the human memory.

Multiple Processing: Configuring two or more processors in a single system, operating out of a common memory. This arrangement permits execution of as many programs as there are processors.

Nesting: Subroutines which are called by subroutines are said to be nested. The nesting level is the number of times nesting can be repeated.

Nibble: A sequence of 4 bits operated upon as a unit. Also see Byte.

Object Program: Program which is the output of an automatic coding system, such as the assembler. Often the object program is a machine-language program ready for execution.

On-Line System: A system of I/O devices in which the operation of such devices is under the control of the CPU, and in which information reflecting current activity is introduced into the data processing or controlling system as soon as it occurs.

Op Code (Operation Code): A code that represents specific operations of an instruction.

Operating System: System software controlling the overall operation of a multi-purpose computer system, including such tasks as memory allocation, input and output distribution, interrupt processing, and job scheduling.

Page: A natural grouping of memory locations by higher-order address bits. In an 8-bit microprocessor, $2^8 = 256$ consecutive bytes often may constitute a page. Then words on the same page only differ in the lower-order 8 address bits.

PLA (Programmable Logic Array): A PLA is an array of logic elements which can be programmed to perform a specific logic function. In this sense, the array of logic elements can be as simple as a gate or as complex as a ROM. The array can be programmed (normally mask programmable) so that a given input combination produces a known output function.

Pointer: Registers in the CPU which contain memory addresses. See Program Counter and Data Pointer.

Program: A collection of instructions properly ordered to perform some particular task.

Program Counter: A CPU register which specifies the address of the next instruction to be fetched and executed. Normally it is incremented automatically each time an instruction is fetched.

PROM (Programmable Read-Only Memory): An integrated-circuit memory array that is manufactured with a pattern of either all logical zeros or ones and has a specific pattern written into it by the user by a special hardware programmer. Some PROMs, called EAROMs, Electrically Alterable Read-Only Memory, can be erased and reprogrammed.

Prototyping System: A hardware system used to breadboard a microprocessor-based product. Contains CPU, memory, basic I/O, power supply, switches and lamps, provisions for custom I/O controllers, memory expansion, and often, a utility program in fixed memory (ROM). See **COSMAC Development System.**

Pseudo Instruction: See **Interpreter**

Pseudo Program: See **Interpreter**

RAM (Random Access Memory): Any type of memory which has both read and write capability. It is randomly accessible in the sense that the time required to read

from or to write into the memory, is independent of the location of the memory where data was most recently read from or written into. In contrast, in a **Serial Access Memory**, this time is variable.

Register: A fast-access circuit used to store bits or words in a CPU. Registers play a key role in CPU operations. In most applications, the efficiency of programs is related to the number of registers.

Relative Addressing: The address of the data referred to is the address given in the instruction plus some other number. The "other number" can be the address of the instruction, the address of the first location of the current memory page, or a number stored in a register. Relative addressing permits the machine to relocate a program or a block of data by changing only one number.

Resident Software: Assembler and editor programs incorporated with a prototyping system to aid in user program writing and development. See **Software.**

Return Routine: See **Subroutine**

ROM: Read-Only Memory (Fixed Memory) is any type of memory which cannot be readily rewritten; ROM requires a masking operation during production to permanently record program or data patterns in it. The information is stored on a permanent basis and used repetitively. Such storage is useful for programs or tables of data that remain fixed and is usually randomly accessible.

Routine: Usually refers to a sub-program, i.e., the task performed by the routine is less complex. A program may include routines. See **Program.**

Scratch-Pad Memory: RAM or registers which are used to store temporary intermediate results (data), or memory addresses (pointers).

Serial Memory (Serial Access Memory): Any type of memory in which the time required to read from or write into the memory is dependent on the location in the memory. This type of memory has to wait while nondesired memory locations are accessed. Examples are paper tape, disc, magnetic tape, CCD, etc. In a Random Access Memory, access time is constant.

Simulators: Software simulators are sometimes used in the debug process to simulate the execution of machine-language programs using another computer (often a timesharing system). These simulators are especially useful if the actual computer is not available. They may facilitate the debugging by providing access to internal registers of the CPU which are not brought out to external pins in the hardware. See **COSMAC Software Development Package.**

Snapshots: Capture of the entire state of a machine (real or simulated) -- memory contents, registers, flags, etc.

Software: Computer programs. Often used to denote general-purpose programs provided by the manufacturer, such as assembler, editor, compiler, etc.

Source Program: Computer program written in a language designed for ease of expression of a class of problems or procedures, by humans: symbolic or algebraic.

Stack: A sequence of registers and/or memory locations used in LIFO fashion (last-in-first-out). A **stack pointer** specifies the last-in entry (or where the next-in entry will go).

Stack Pointer: The counter, or register, used to address a stack in the memory. See Stack.

Stand-Alone System: A microcomputer software development system which runs on a microcomputer without connection to another computer or a timesharing system. This system includes an assembler, editor, and debugging aids. It may include some of the features of a prototyping kit.

State Code: A coded indication of what state the CPU is -- responding to an interrupt, servicing a DMA request, executing an I/O instruction, etc.

Subroutine: A subprogram (group of instructions) reached from more than one place in a **main program.** The process of passing control from the main program to a subroutine is a **subroutine call,** and the mechanism is a **subroutine linkage.** Often data or data addresses are made available by the main program to the subroutine. The process of returning control from subroutine to main program is **subroutine return.** The linkage automatically returns control to the original position in the main program or to another subroutine. See Nesting.

Subroutine Linkage: See **Subroutine**

Support: See **Manufacturer's Support**

Synchronous Operation: Use of a common timing source (clock) to time circuit or data transfer operations. (Contrast with **Asynchronous** operation)

Syntax: Formal structure. The rules governing sentence structure in a language, or statement structure in a language such as assembly language or Fortran.

Terminal: An Input-Output device at which data leaves or enters a computer system, e.g., teletype terminal, CRT terminal, etc.

Test and Branch: See **Branch Instruction**

Unbundling: Pricing certain types of software and services separately from the hardware.

Universal Asynchronous Receiver/Transmitter (UART): A device that translates serial data bits from two-wire lines to parallel format (receive mode) or parallel data bits to serial format for transmission over two-wire lines (transmit mode).

Utility Program: A program providing basic conveniences, such as capability for loading and saving programs, for observing and changing values in a computer, and for initiating program execution. The utility program eliminates the need for "re-inventing the wheel" every time a designer wants to perform a common function.

Word: The basic group of bits which is manipulated (read in, stored, added, read out, etc.) by the computer in a

single step. Two types of word are used in every computer: Data Words and Instruction Words. Data words contain the information to be manipulated. Instruction words cause the computer to execute a particular operation.

Word Length: The number of bits in the computer word. The longer the word length, the greater the precision (number of significant digits). In general, the longer the word length, the richer the instruction set, and the more varied the addressing modes.

Index