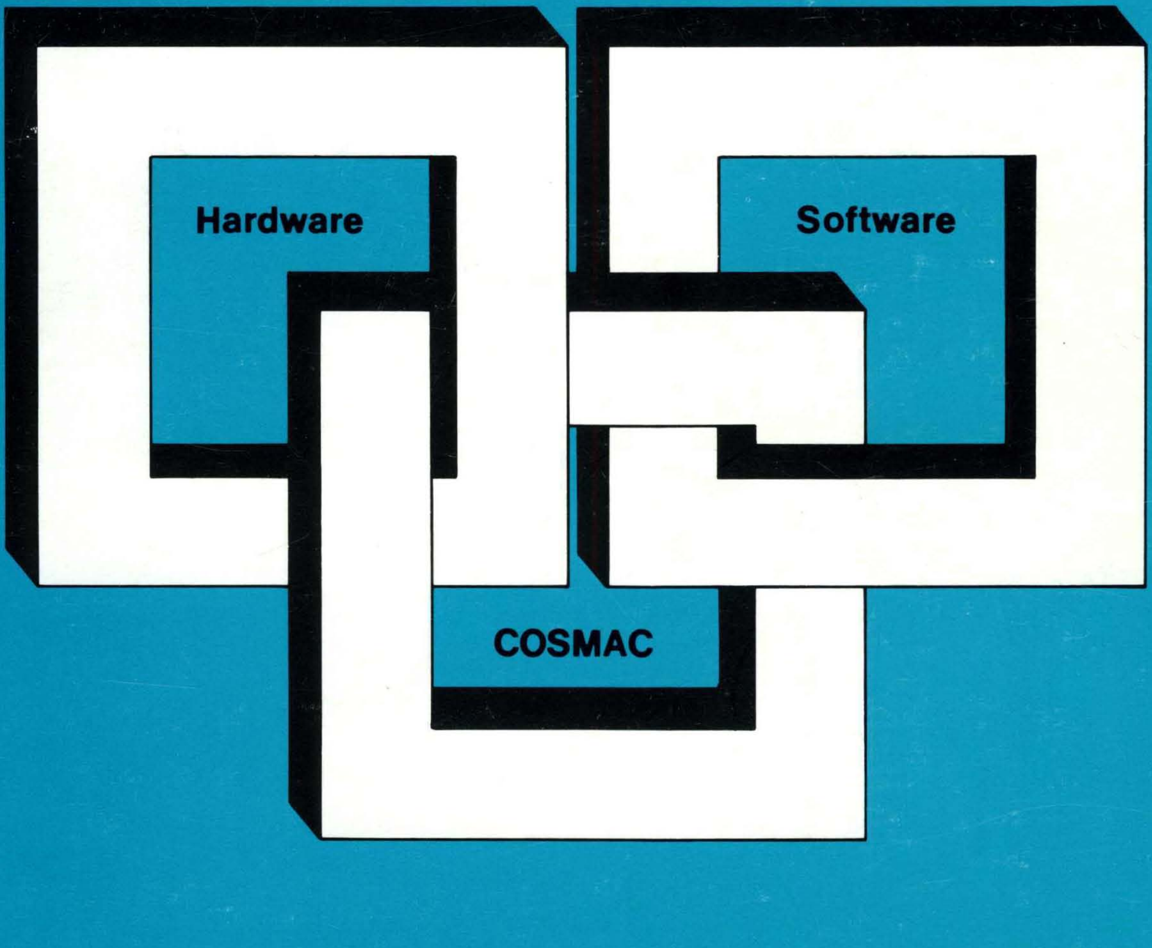
Microprocessor Products

User Manual for the COSMAC Microprocessor

RCA Solid State



MPM-101

$5.00 Suggested Price

User Manual for the COSMAC Microprocessor

RCA | Solid State Division | Somerville, NJ 08876

Printed in USA/5-75

Table of Contents

Foreword

The RCA Microprocessor (COSMAC) is an LSI 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 electrical engineers, and assumes no familiarity with computers. It describes the microprocessor architecture and its set of simple, easy-to-use instructions. Examples are given to illustrate the operation 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 describes machine-code programming methods and gives detailed examples. Potential programming errors are discussed, and various programming techniques are described, including interrupt response, long branch, and subroutine linkage and nesting.

This basic manual is intended to help design engineers understand the COSMAC Microprocessor and aid them in developing simpler and more powerful products based on microprocessors. Users requiring information on the operation of the RCA COSMAC Microprocessor software support system should refer to the MPM-102 "Program Development Guide for the COSMAC Microprocessor".

Introduction

General

The COSMAC Microprocessor has been developed and tested within RCA in a wide variety of applications. COSMAC is suitable for use in business, education, entertainment, instrumentation, control, communications, and other applications where stored program control is desired.

The RCA COSMAC Microprocessor is a CMOS byte-oriented central processing unit (CPU). It is suitable for use in a wide range of stored-program computer systems or products. These systems can be either special or general-purpose in nature. They are **byte**-oriented, a byte being eight bits.

COSMAC operations are specified by sequences of one-byte operation codes stored in a memory. These operation codes are called **instructions**. 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(s) 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 hardware components (memory and microprocessor) in a variety of different systems or products.

The COSMAC 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.

Microprocessor cost is only a small part of total system or product cost. Memory, input, output, power-supply, system-control, and programming costs are also major considerations. A unique set of COSMAC features combine to minimize the total system cost.

COSMAC's low-power, single-voltage CMOS circuitry minimizes power-supply and packaging costs. High noise immunity and wide temperature tolerance facilitate use in hostile environments.

COSMAC 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 instruction format.

The 40-pin COSMAC system interface is designed to minimize external I/O and memory control circuitry. A single-phase clock, internal direct-memory-access (DMA) mode, flexible I/O instructions, program interrupt, program load mode, and static circuitry are other COSMAC features explicitly aimed at total system cost reduction. COSMAC does not require an external bootstrap ROM.

Microprocessor programming is facilitated by a variety of support programs or software. 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. COSMAC provides a set of efficient, easy-to-learn instructions which are simple to use.

The COSMAC microprocessor comprises two conservatively designed LSI chips (one 40-pin and one 28-pin dual-in-line package). Appendix C shows the required interconnections for these two LSI chips and summarizes the COSMAC system interface signals.

Specific Features

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

- static COS/MOS circuitry, no minimum clock frequency
- full military temperature range
- high noise immunity, wide operating-voltage range
- TTL compatibility
- 8-bit parallel organization with bidirectional data bus
- built-in program-load facility
- any combination of standard RAM/ROM 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
- one-byte instruction format with two machine cycles for each instruction
- 59 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 COSMAC microprocessor. Operations that can be performed by COSMAC include:

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

b) transfer of binary data between I/O and memory (M),

c) movement of data bytes between different memory locations,

d) interpretation or modification of bytes stored in memory.

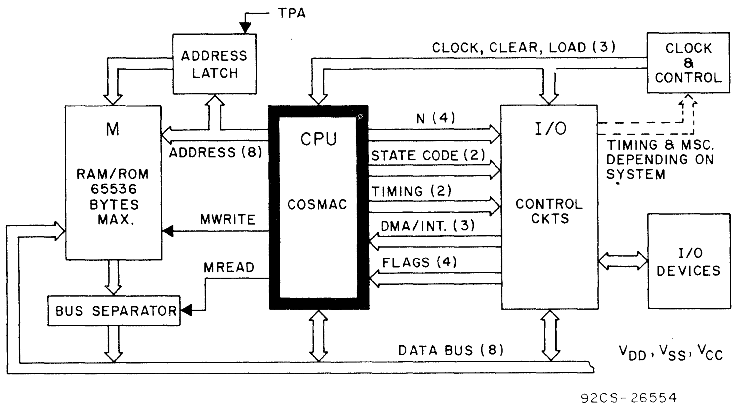


Fig. 1 – Block diagram of typical computer system using the COSMAC microprocessor.

For example, COSMAC can control the entry of binary-coded decimal numbers from an input keyboard and store them in predetermined memory locations. COSMAC 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, 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 COSMAC **I/O control signal lines** are provided. Systems can use some or all of these signals depending on required I/O sophistication. A four-bit **N code** is generated by the COSMAC input/output instruction. It can be used to specify an I/O device to be involved in an I/O-memory byte transfer by means of the data bus, or, alternatively, to specify whether an I/O byte represents data, an I/O device selection code, an I/O status code, or an I/O control code. Use of the N code to directly specify an I/O device permits simple, inexpensive control of a small number of I/O devices or modes. Use of the N code to specify the meaning of the word on the data bus facilitates systems incroporating a large number of I/O devices or modes.

Four **I/O flag inputs** are provided. I/O devices can activate these inputs at any time to signal COSMAC 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 COSMAC instructions to determine whether or not they are active. Use of the flag inputs must be coordinated with programs that test them.

A program **interrupt line** can be activated at any time by I/O circuits to obtain an immediate COSMAC response. The interrupt causes COSMAC to suspend its current program sequence and execute a predetermined sequence of operations designed to respond to the interrupt condtion. After servicing the interrupt, COSMAC resumes execution of the interrupted program. COSMAC 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 affecting the COSMAC 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 sets this pointer to an initial 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. A program must sample a flag line to determine when it becomes active. Activating the interrupt line causes an immediate COSMAC response regardless of the program currently in progress, suspending operation of that program. Use of DMA provides the quickest response with least disturbance of the program.

A two-bit COSMAC **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 COSMAC operating cycles. The state code indicates whether COSMAC is responding to a DMA request, responding to an interrupt request, executing an input/output instruction, or none of these. The timing signals are used by the memory and I/O systems to signal a new processor state code, to latch memory address bits, to take memory data 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. COSMAC provides two lines to control memory read/write cycles. During a memory write cycle, the byte to be written appears on the data bus and a **memory write pulse** is generated by COSMAC at the appropriate time. **A memory read level** is generated which is used by the system to gate the memory output byte onto the common data bus.

COSMAC 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 COSMAC timing pulses strobes the required high-order bits into an address latch (register) when they appear on the eight address lines. An internal COSMAC register holds the eight low-order address bits on the address lines for the remainder of the memory cycle. No external latch circuits are required for the low -order address byte.

Three additional lines complete the COSMAC microprocessor system interface. A single-phase **clock input** determines operating speed. The external clock may be stopped and started to synchronize COSMAC operation with system circuits if desired. A single **clear input** initializes internal COSMAC circuitry in one step. The **load signal line** holds the COSMAC microprocessor in the program load mode. The use of this mode is discussed in the section on **Memory and Control Interface**.

COSMAC Architecture and Notation

Fig. 2 illustrates the internal structure of the COSMAC microprocessor. 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 **scratchpad registers**. Each scratchpad 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 (o,1,2,...E,F) and their binary equivalents (0000,0001,0010,...,1110,1111) are listed in Appendix A.

Using hex notation, R(3) refers to the 16-bit scratchpad 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 4-bit binary codes (hex digits) that are used to select individual 16-bit scratchpad registers. The 16 bits contained in a selected scratchpad can be copied into the 16-bit **A register**. The two A-register bytes are sequentially placed on the eight external memory address lines for memory read/write operations. Either of the two A-register bytes (A.0/A.1) 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 scratchpad register to permit a scratchpad register to be used as a counter.

The notation R(X), R(N), or R(P) is used to refer to a scratchpad register selected by the 4-bit code in X, N, or P, respectively. Fig. 3 illustrates the transfer of a scratchpad register byte, designated by N, to D. The 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 indicated that the low-order 8 bits contained in the scratchpad register designated by the hex digit in N are to be placed into the 8-bit D register. The designated scratchpad register is left unchanged.

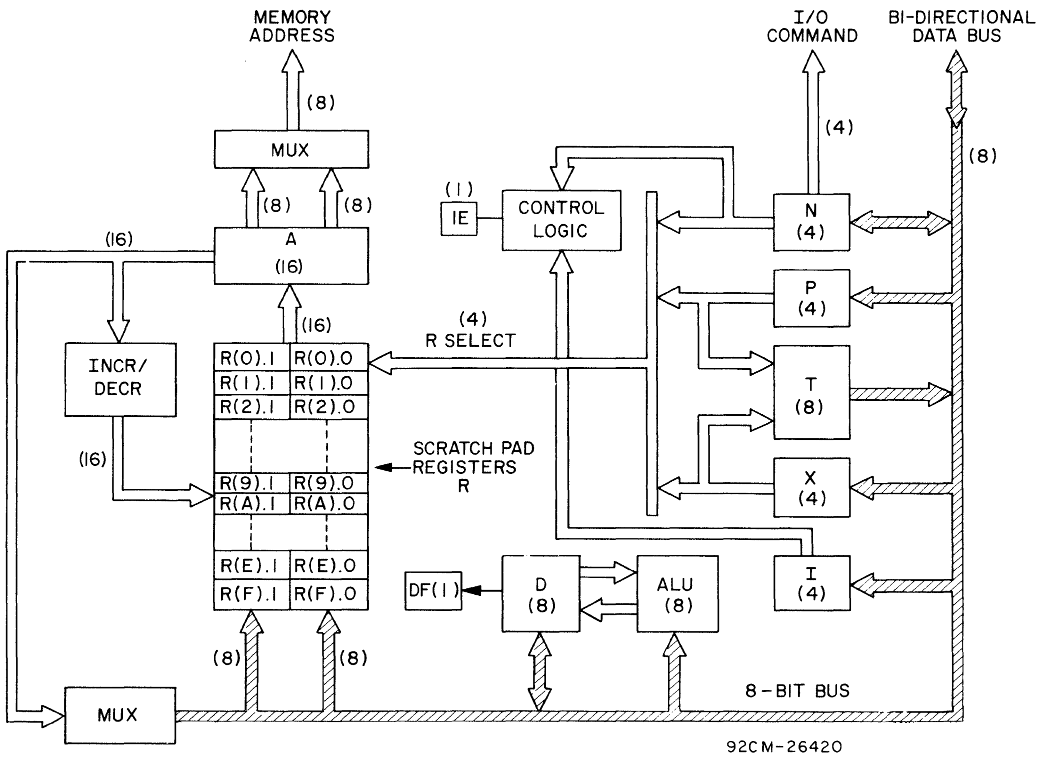


Fig. 2 – Internal structure of the COSMAC microprocessor.

The right half of Fig. 3 illustrates the contents of the COSMAC 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. (right half of Fig. 3)
4) The bus is gated to D.

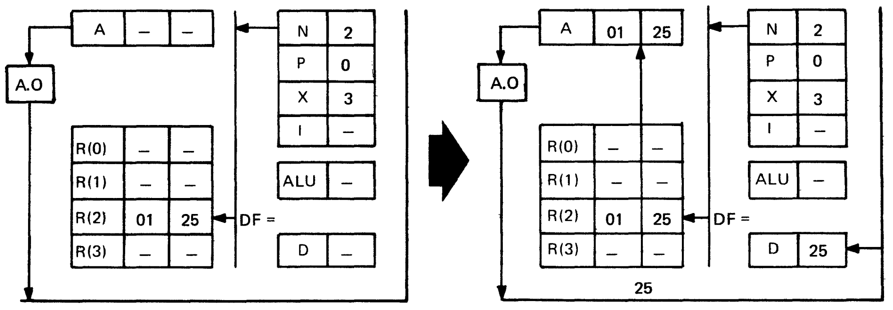


Fig. 3 – Use of N designator to transfer data from scratchpad register R(2) to the D register.

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 scratchpad 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.

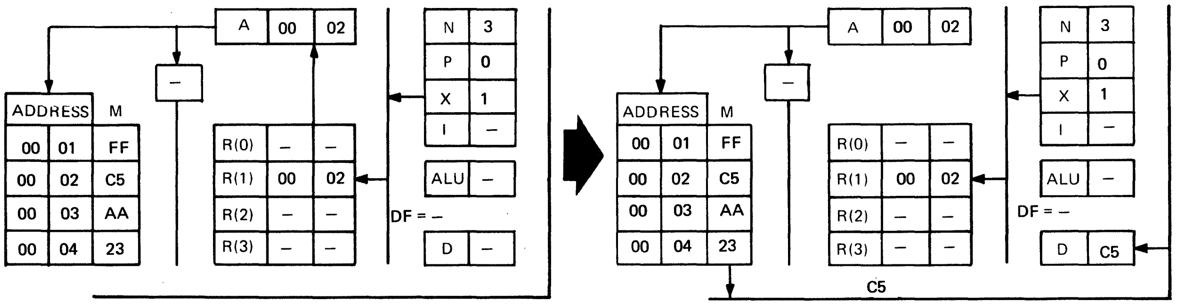


Fig. 4 — Transfer of data from memory to the D register.

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 "O" if no carry results from an add, subtract, or shift operation. DF is set to "1" if a carry does occur. The 8-bit D register is similar to the accumulator found in many computers.

Instructions and Timing

COSMAC operations are specified by a sequence of operation codes stored in external memory. These code are called **instructions.** Each instruction consists of one 8-bit byte. Two 4-bit hex digits contained in each instruction byte are designated as **I** and **N**, as shown in Fig. 5.

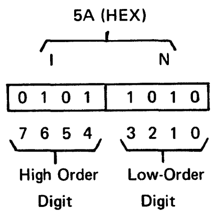


Fig. 5 — Eight-bit instruction format.

The execution of each instruction 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 scratchpad register, as illustrated in Fig. 3, or acts as a special code, as described in more detail below.

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

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

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 execute cycles in this manner causes sequences of instructions that are stored in memory to be executed.

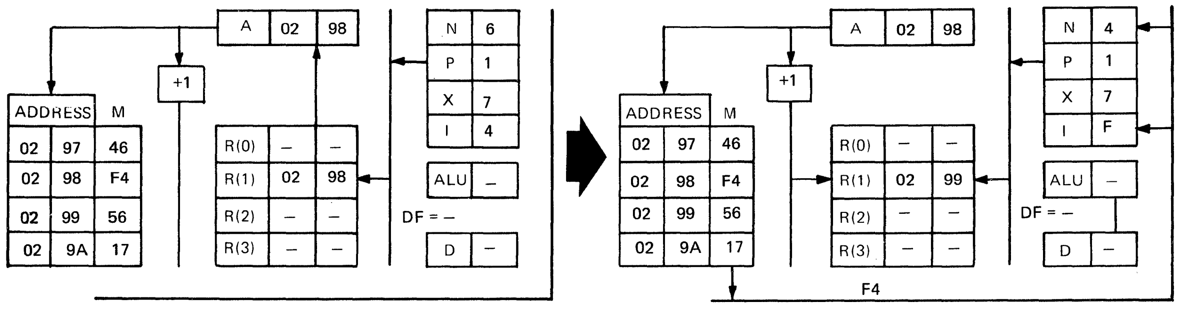


Fig. 6 — Typical instruction fetch cycle.

The COSMAC machine cycle during which an instruction byte is fetched from memory is called **state O** (SO). The cycle during which the fetched instruction is executed is called **state 1** (S1). During execution of a program, COSMAC alternates between SO and S1, as shown below:

··· | SO | S1 | SO | S1 | SO | S1 | ···

Each machine cycle is internally divided into eight equal time intervals, as illustrated in Appendix D under general timing. Each time interval is equivalent to one external clock cycle (T). The rate at which machine cycles occur is, therefore, one-eight of the clock frequency. The **instruction time** is 16T or two machine cycles. All instructions require the same fetch/execute time.

Instruction Repertoire

Each COSMAC instruction is fetched during SO and executed during S1. 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 six general classes:

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

Memory Reference — Two instructions are provided to load or store a memory byte.

ALU Operations — This group contains fifteen instructions for performing arithmetic and logical operations.

I/O Byte Transfer — Eight instructions are provided to load memory from I/O control circuits, and eight instructions to transfer data from memory to I/O control circuits.

Branching — Fourteen different conditional and unconditional branch instruction are provided.

Control — Six control instructions facilitate program interrupt, operand selection, or branch and link operations.

Each instruction is designated by its two-digit hex code and by a name. A description of the operation is provided using a symbolic notation. A two- or three-letter abbreviated name is also given. Examples are shown in this section for most instructions. A summary of the instruction repertoire is given in Appendix A. It should be noted that any unused machine codes, such as "CN" "31", "72", "01", etc., are considered illegal codes and should not be used by users. They are reserved for future use by RCA.

Register Operations

1N	INCREMENT	R(N)+1	INC

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

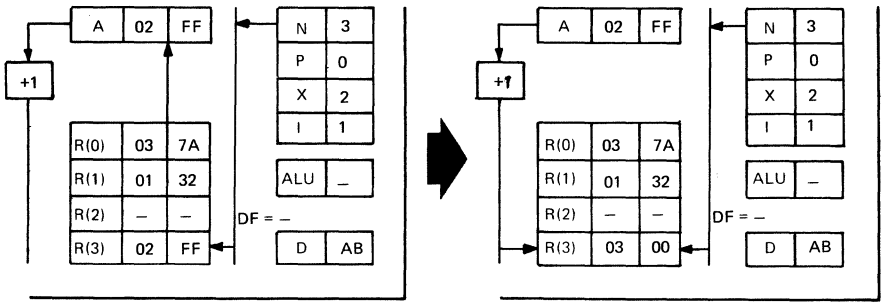


Fig. 7 — Example of instruction 1N — INCREMENT.

2N	DECREMENT	R(N)−1	DEC

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

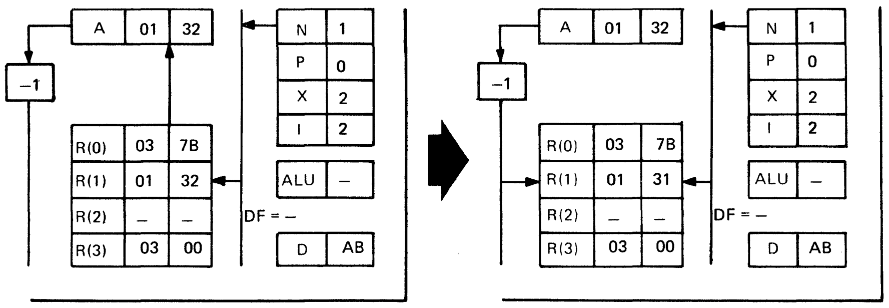


Fig. 8 – Example of instruction 2N – DECREMENT.

8N	GET LOW	R(N).0 → D	GLO

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

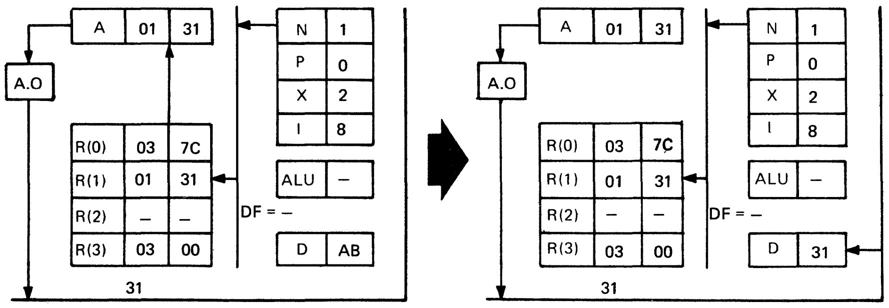


Fig. 9 – Example of instruction 8N – GET LOW.

9N	GET HIGH	R(N).1 → D	GHI

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

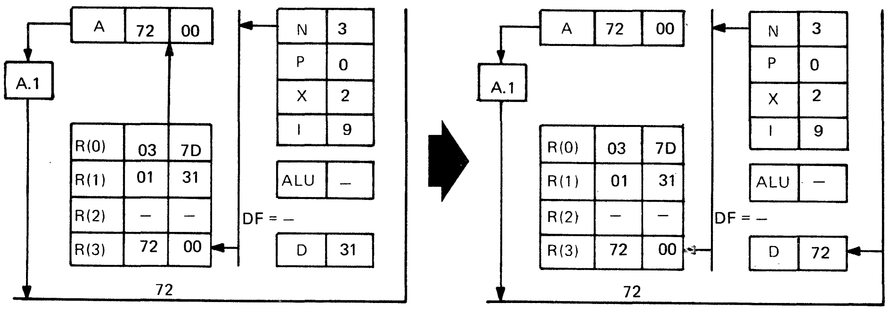


Fig. 10 – Example of instruction 9N – GET HIGH.

AN	PUT LOW	D → R(N).0	PLO

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

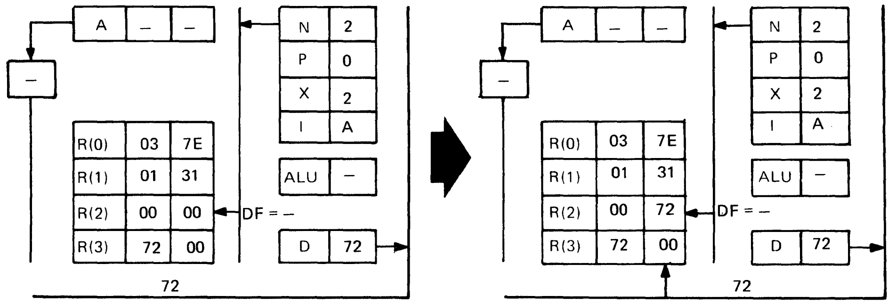


Fig. 11 – Example of instruction AN – PUT LOW.

BN	PUT HIGH	D → R(N).1	PHI

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

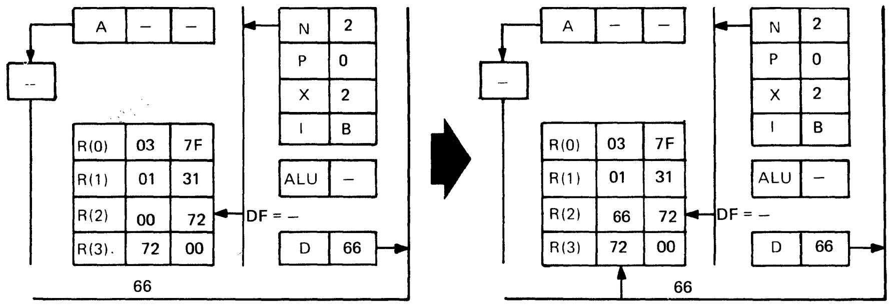


Fig. 12 – Example of instruction BN – PUT HIGH.

Memory Reference

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

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

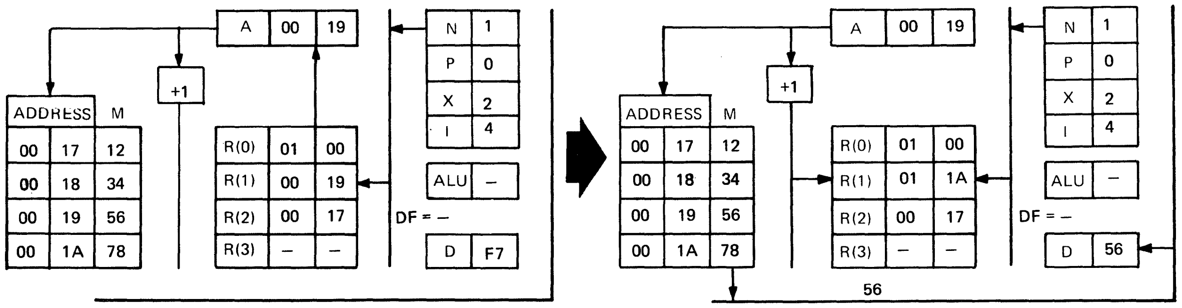


Fig. 13 – Example of instruction 4N – LOAD ADVANCE.

5N	STORE	D → M(R(N))	STR

When I=5, the byte in D replaces the memory byte addressed by the contents of the register specified by N.

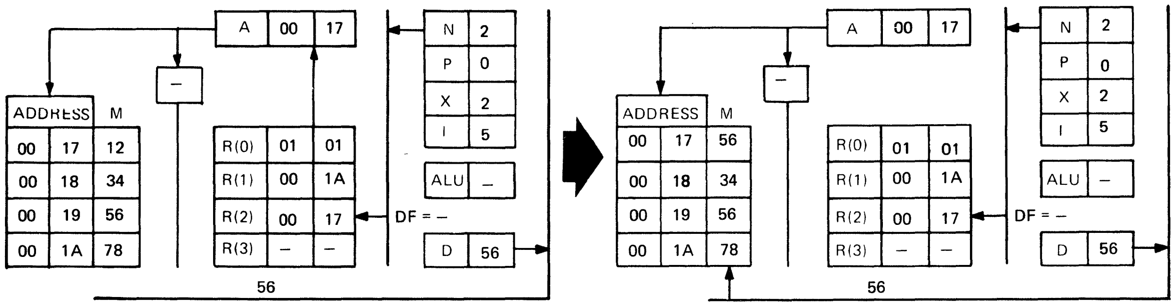


Fig. 14 – Example of instruction 5N – STORE.

ALU Operations Using M(R(X))

In this group of instructions, the N digit of the instruction is a code specifying a specific ALU operation. The high-order bit of N is O. The X register must previously have been loaded (by an instruction, SET X, described among the control instructions). In general, R(X) points at one operand, D is the other, and the result replaces the latter in the D register.

FO	LOAD BY X	M(R(X)) → D	LDX

When I=F and N=O, 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.)

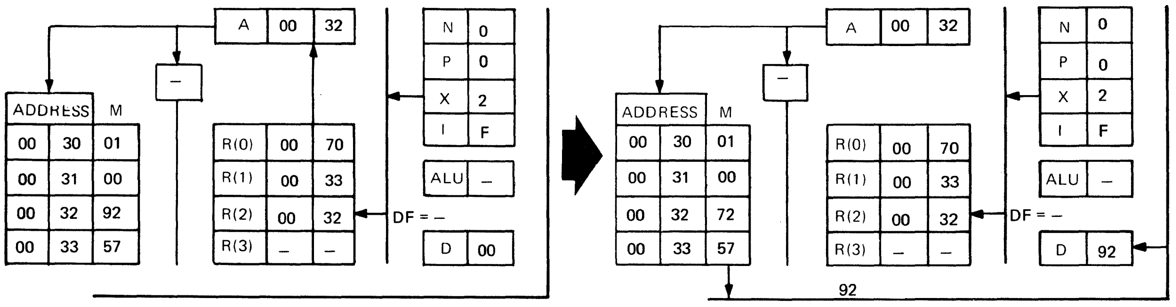


Fig. 15 – Example of instruction FO – LOAD BY X.

F1	OR	M(R(X)) v D → D	OR

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 follows:

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

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.

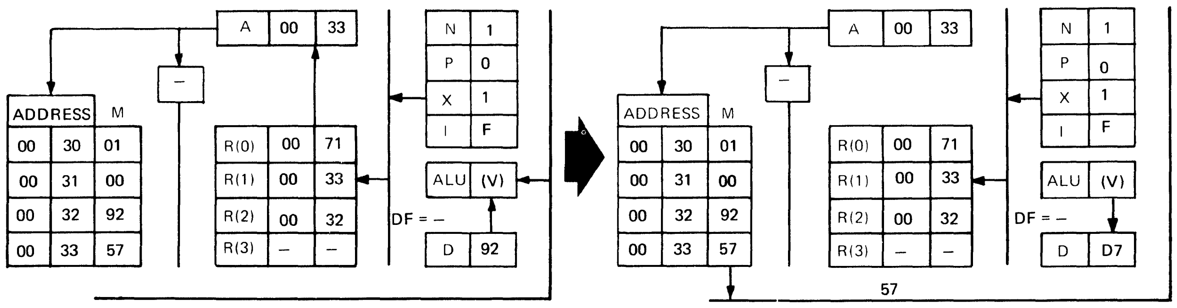


Fig. 16 – Example of instruction F1 – OR.

F2	AND	M(R(X)) • D → D	AND

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 follows:

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

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.

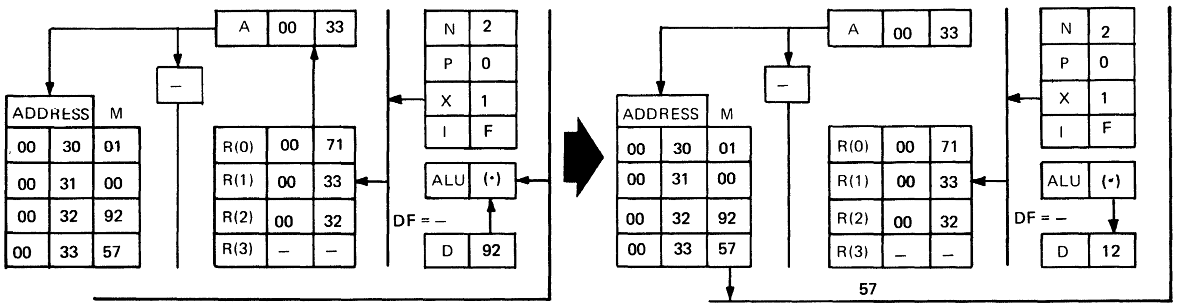


Fig. 17 – Example of instruction F2 – AND.

F3	EXCLUSIVE-OR	M(R(X)) ⊕ D → D	XOR

When I=F and N=2, the individual bits of the two 8-bit operands are combined according to the rules for logical EXCLUSIVE-OR as follows:

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

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.

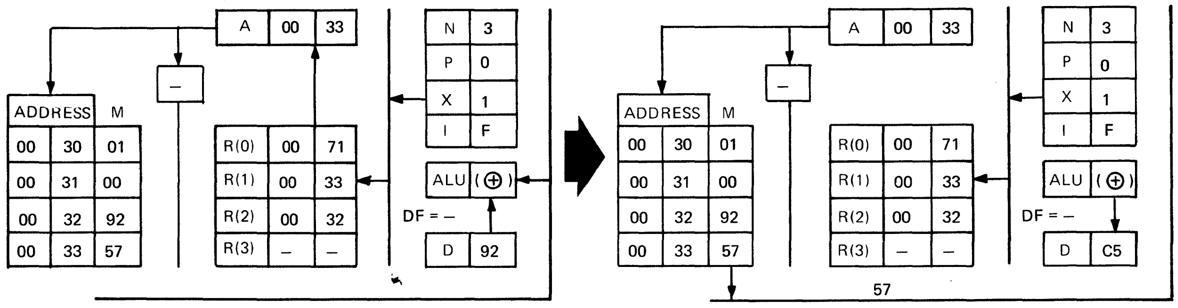


Fig. 18 – Example of instruction F3 – EXCLUSIVE-OR.

F4	ADD	M(R(X)) + D → D; C → DF	ADD

When I=F and N=4, the 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 carry occurred:

3A + 4B = 85 (DF=0)
3A + FO = 2A (DF=1)

DF can be subsequently tested with a branch instruction.

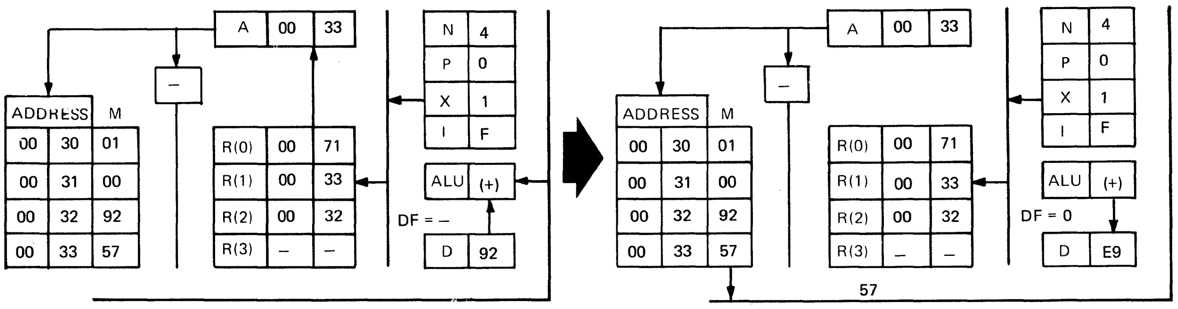


Fig. 19 – Example of instruction F4 – ADD.

F5	SUBTRACT D	M(R(X)) – D → D; C → DF	SD

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:

42 –OE = 42+F1+ = 34 (DF = 1)
42 –42 = 42+BD+1 = 00 (DF = 1)
42 –77 = 42 +88+1 = CB (DF = 0)

A final value of "0" in DF indicates that the subtrahend was larger than the minuend. In this case the value in D is exactly 100 (hexadecimal) greater than the true (negative) difference.

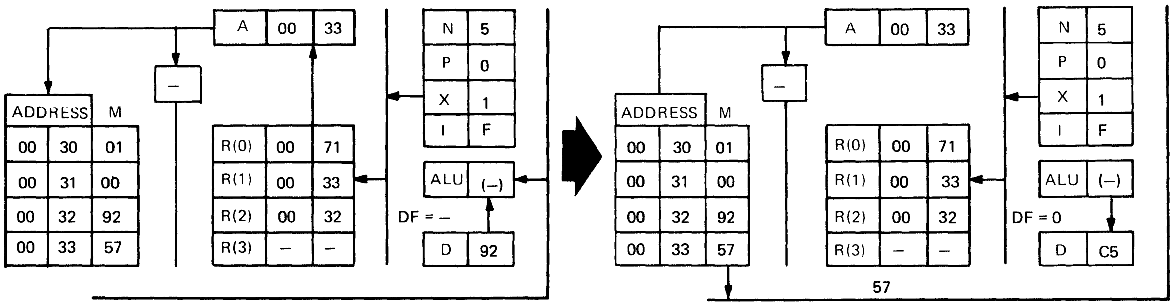


Fig. 20 – Example of instruction F5 – SUBTRACT D.

F7	SUBTRACT M	D-M(R(X)) → D; C → DF	SM

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.

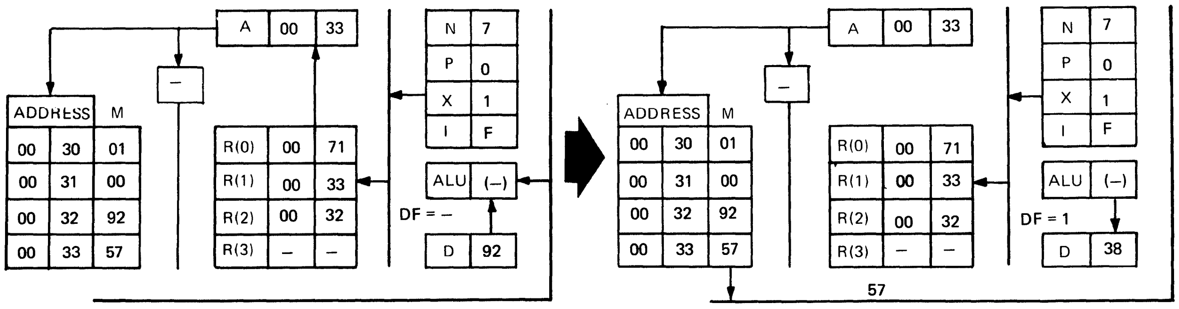


Fig. 21 – Example of instruction F7 – SUBTRACT M.

F6	SHIFT RIGHT	SHIFT D RIGHT 1 BIT; LSB → DF; 0 → MSB	SHR

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". In this instruction, unlike other instructions in this group, M(R(X)) is not used. This instruction can be used to test successive bits of the operand or to divide by 2.

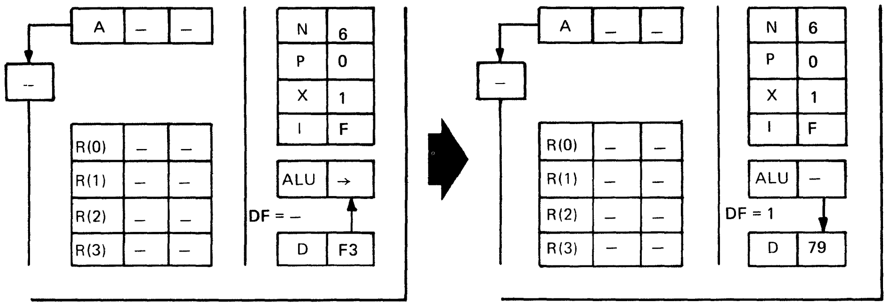


Fig. 22 – Example of instruction F6 – SHIFT RIGHT.

ALU Operations Using M(R(P))

In this group of ALU instructions, the N digit has a 1 in the high-order bit position. The remaining three bits of N are a code specifying the same ALU operation as instructions using M(R(X)), except when N=6.

In general, R(P) points to one of the operands, the byte in memory after the instruction byte, called the **immediate byte**. The D register supplies the second operand, and then receives the result.

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

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

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.

This instruction is one of three which load D from memory. It uses R(P) as a pointer, while LDA uses R(N) and LDX uses R(X). LDI and LDA each increment the pointer after use, but LDX does not.

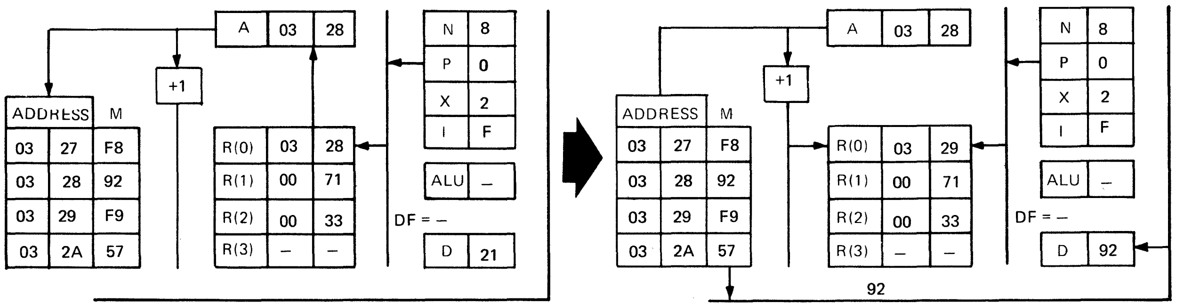


Fig. 23 – Example of instruction F8 – LOAD IMMEDIATE.

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

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.

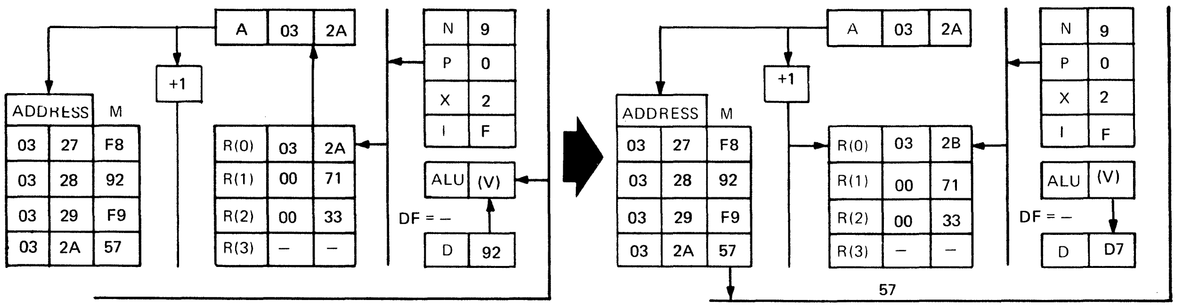


Fig. 24 – Example of instruction F9 – OR IMMEDIATE.

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

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.

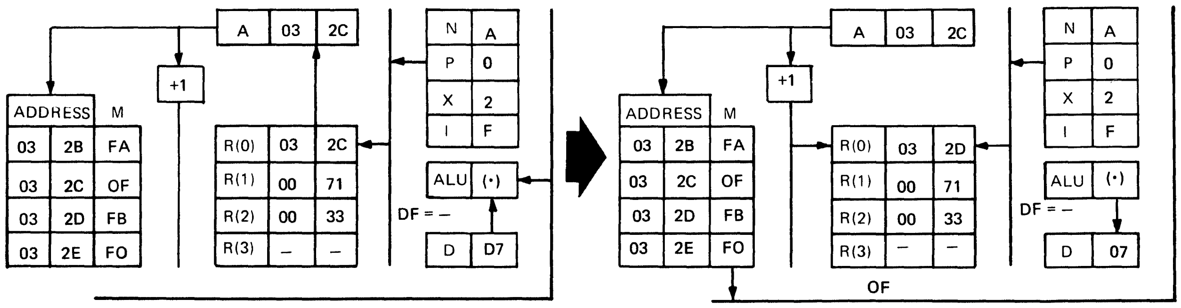


Fig. 25 – Example of instruction FA – AND IMMEDIATE.

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

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".

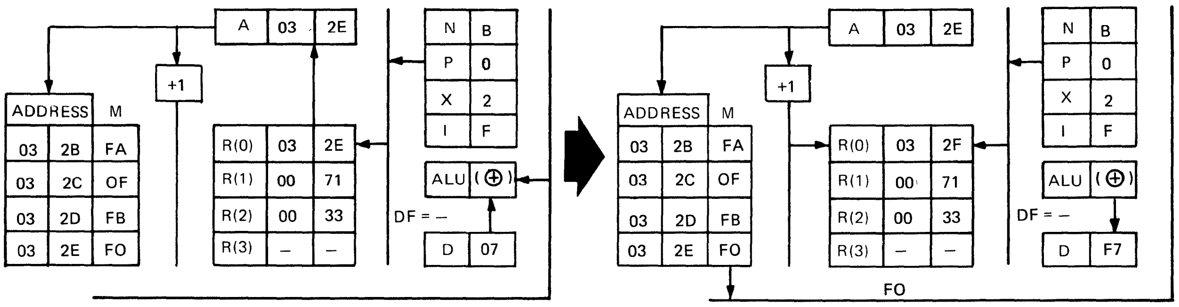


Fig. 26 – Example of instruction FB – EXCLUSIVE-OR IMMEDIATE.

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

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.

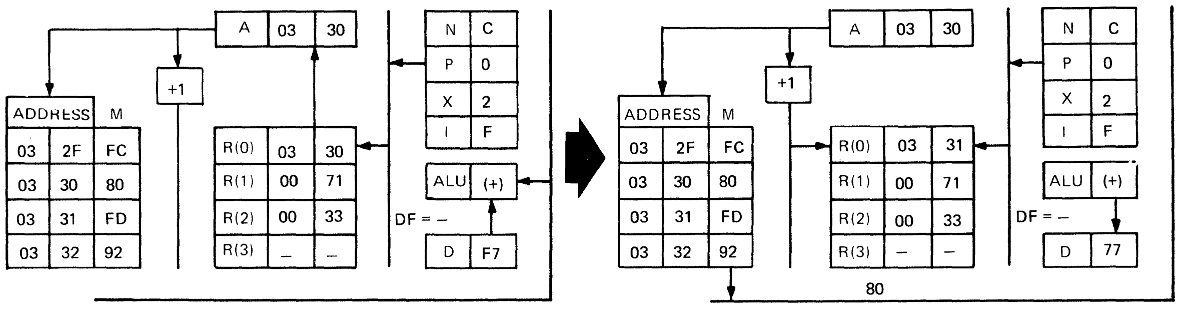


Fig. 27 – Example of instruction FC – ADD IMMEDIATE.

FD	SUBTRACT D IMMEDIATE	M(R(P))-D → D; C → DF; R(P)+1	SDI

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.

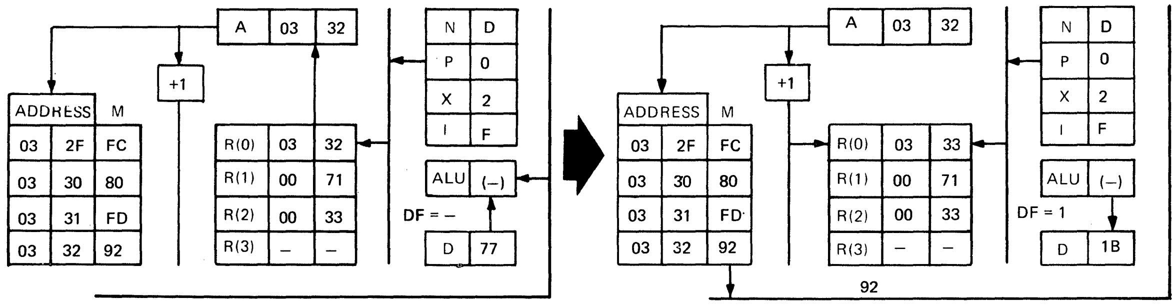


Fig. 28 – Example of instruction FD – SUBTRACT D IMMEDIATE.

FF	SUBTRACT M IMMEDIATE	D-M(R(P)) → D; C → DF; R(P)+1	SMI

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.)

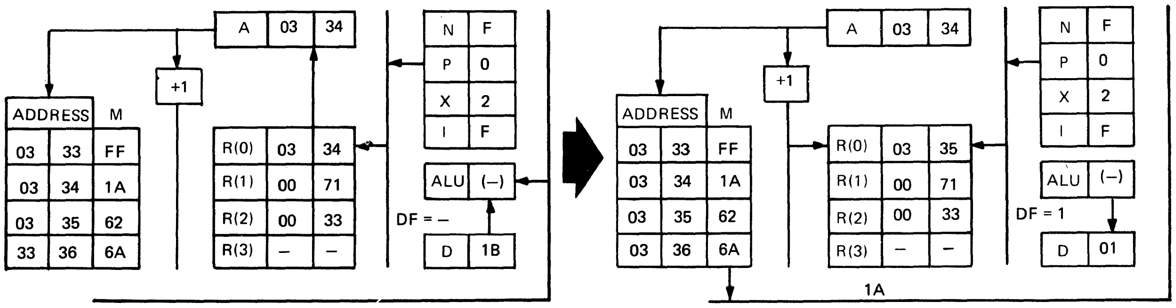


Fig. 29 – Example of instruction FF – SUBTRACT M IMMEDIATE.

Input/Output Byte Transfer

6N	N=0-7	OUTPUT	M(R(X)) → BUS; R(X)+1	OUT

When 1=6 and N=0,1,2,3,4,5,6, or 7, the memory byte addressed by R(X) is placed on the data bus. The four bits of N are simultaneously sent from COSMAC to the I/O system, and a specific code is provided on

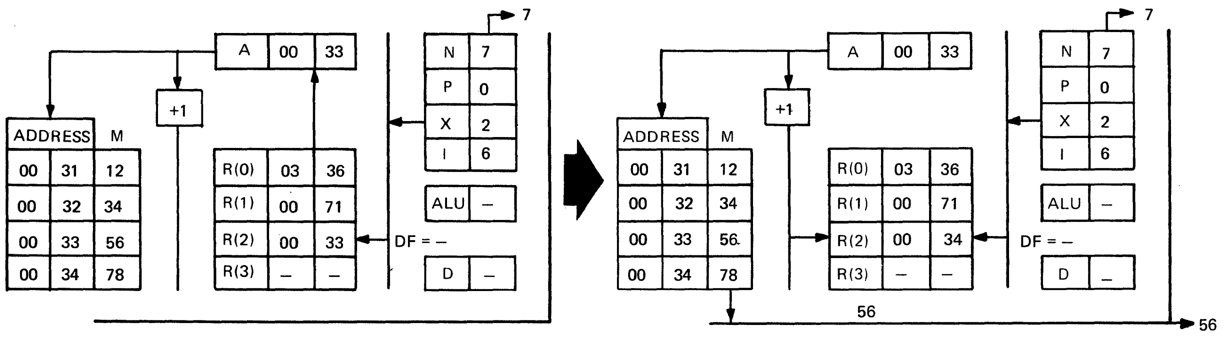


Fig. 30 – Example of instruction 6N (N=0-7) – OUTPUT.

the COSMAC state code lines to indicate I/O (I=6). The most significant bit of N is "O", indicating "OUTPUT". The I/O system recognizes these conditions, and reads the output byte from the bus. The 3 less significant bits of N specify which of the 8 output instructions is being executed. 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.

6N	N=8-F	INPUT	BUS→M(R(X))	INP

When I=6 and N=8,9,A,B,C,D,E, or F, an input byte replaces the memory byte addressed by R(X). R(X) is not modified. The four bits of N are simultaneously sent from COSMAC to the I/O system, and the I/O state code (I=6) is provided. The most significant bit of N is "1", indicating "INPUT". The I/O circuits should gate an input byte onto the data bus during the execute cycle. The 3 least significant bits of N specify which of the 8 possible input instructions is being executed. R(X) is not modified.

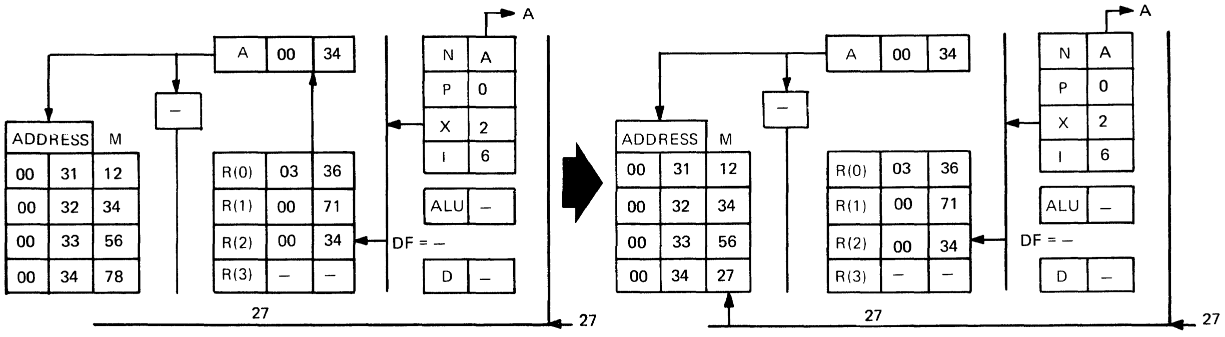


Fig. 31 – Example of instruction 6N (N=8-F) – INPUT.

Branching

The current program counter, R(P), normally steps sequentially through a list of instructions, skipping over immediate data bytes. When I=3, a 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.

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

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

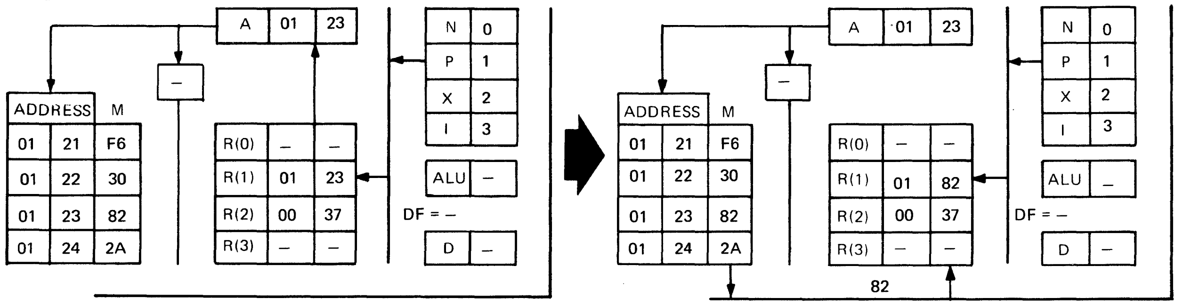


Fig. 32 – Example of instruction 30 – UNCONDITIONAL BRANCH.

32	BRANCH IF D=00	M(R(P)) → R(P).0 IF D=00, OR R(P)+1	BZ

When I=3 and N=2, a conditional 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.

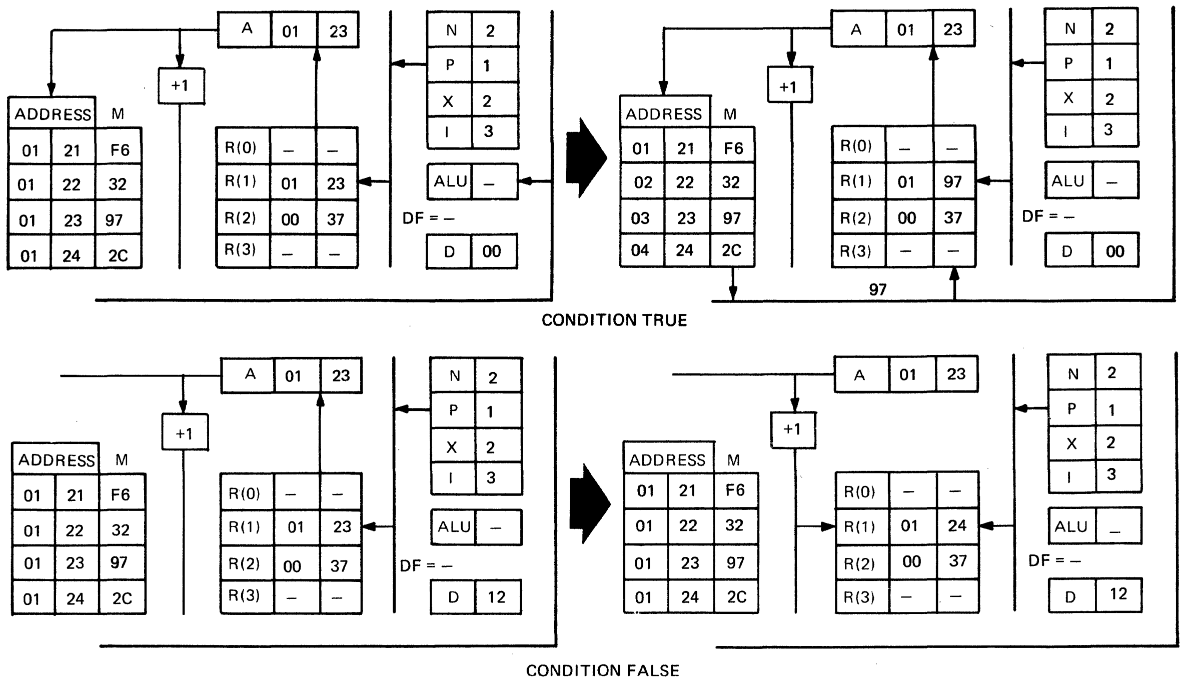


Fig. 33 – Example of instruction 32 – BRANCH IF D=00 for both false and true conditions.

33	BRANCH IF DF	M(R(P)) → R(P).0 IF DF=1, OR R(P)+1	BDF

When I=3 and N=3, branching occurs if DF=1. Otherwise, the next instruction in sequence is performed. Examples are not shown for the remainder of the branching instructions because they differ only in the condition tested.

34	BRANCH IF EF1	M(R(P)) → R(P).0 IF EF1=1, OR R(P)+1	B1
35	BRANCH IF EF2	M(R(P)) → R(P).0 IF EF2=1, OR R(P)+1	B2
36	BRANCH IF EF3	M(R(P)) → R(P).0 IF EF3=1, OR R(P)+1	B3
37	BRANCH IF EF4	M(R(P)) → R(P).0 IF EF4 =1, OR R(P)+1	B4

When I=3 and N=4,5,6, or 7, branching occurs only when the corresponding external flag input (EF1,2, 3, or 4) is held in its "true" state by external circuits. These four branch instructions permit the microprocessor to test the flags as required.

38	SKIP	R(P)+1	SKP

When I=3 and N=8, the byte following the "38" instruction is skipped.

3A	BRANCH IF D≠00	M(R(P)) → R(P).0 IF D≠00, OR R(P)+1	BNZ

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.

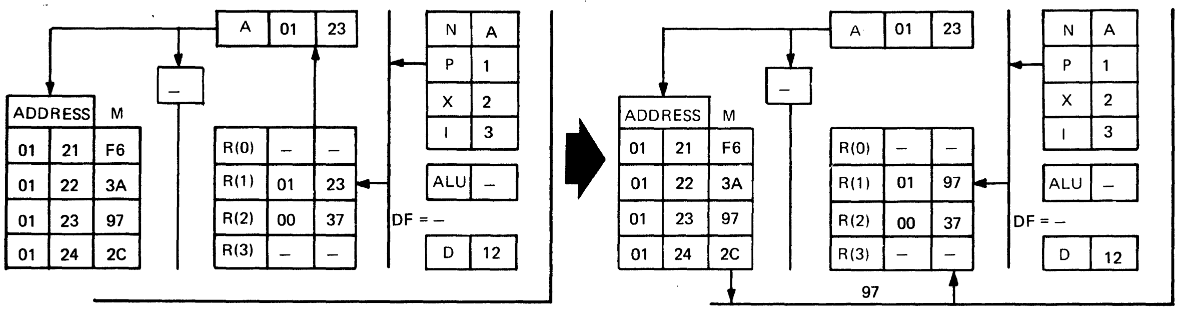


Fig. 34 – Example of instruction 3A – BRANCH IF D≠00.

3B	BRANCH IF NO DF	M(R(P)) → R(P).0 IF DF = 0, OR R(P)+1	BNF

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

3C	BRANCH IF NO EF1	M(R(P)) → R(P).0 IF EF1=0, OR R(P)+1	BN1
3D	BRANCH IF NO EF2	M(R(P)) → R(P).0 IF EF2=0, OR R(P)+1	BN2
3E	BRANCH IF NO EF2	M(R(P)) → R(P).0 IF EF3 = 0, OR R(P)+1	BN3
3F	BRANCH IF NO EF4	M(R(P)) → R(P).0 IF EF4 = 0, OR R(P)+1	BN4

When I=3 and N=C,D,E, or F, a branch occurs only when the corresponding external flag input (EF1,2,3, or 4) is in its "0" state.

Because only the low-order byte of R(P) can be modified by a branch instruction, the range of memory locations that can be branched to is limited. Since only the low-order 8 bits can be modified, branching is limited to 2^8 or 256 bytes. Each 256-byte memory segment is called a **page**. Methods for branching to any location in memory are described in the section on **Machine Code Programming**.

The special case of a 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 (Oth) 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.

Control

OO	IDLE	WAIT FOR INTERRUPT/DMA-IN/DMA-OUT	IDL

When I=O and N=O, the microprocessor repeats execute (S1) cycles until an interrupt, DMA-in, or DMA-out is activated, at which time the IDLE instruction is terminated. During IDLE, the microprocessor continues to put out the two timing pulses for I/O synchronization.

DN	SET P	N→P	SEP

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, subroutine nesting, and long branches to any location in memory. (These topics are discussed in the section on **Machine Code Programming.**)

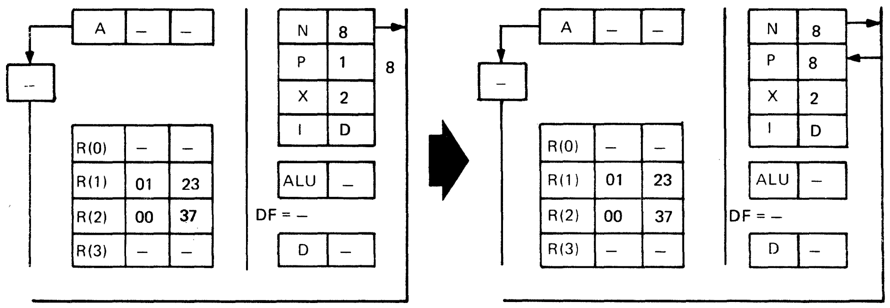


Fig. 35 – Example of instruction DN – SET P.

EN	SET X	N→X	SEX

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.

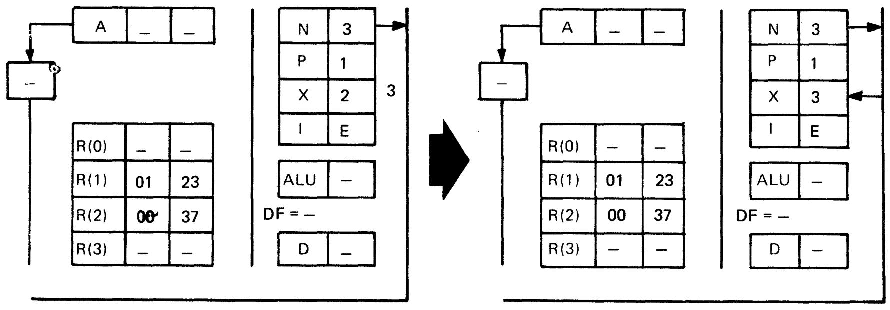


Fig. 36 – Example of instruction EN – SET X.

Interrupt 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), 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 (IE) to "0". Also, a specific code is provided on the COSMAC state code lines. Details of the interrupt servicing are discussed in the section on **I/O Interface.**

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

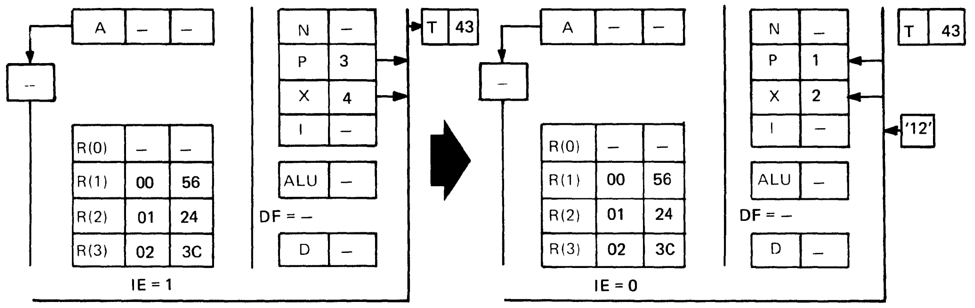


Fig. 37 – Example of instruction —— – INTERRUPT ACTION.

78	SAVE	T → M(R(X))	SAV

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.

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

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) latch is set.

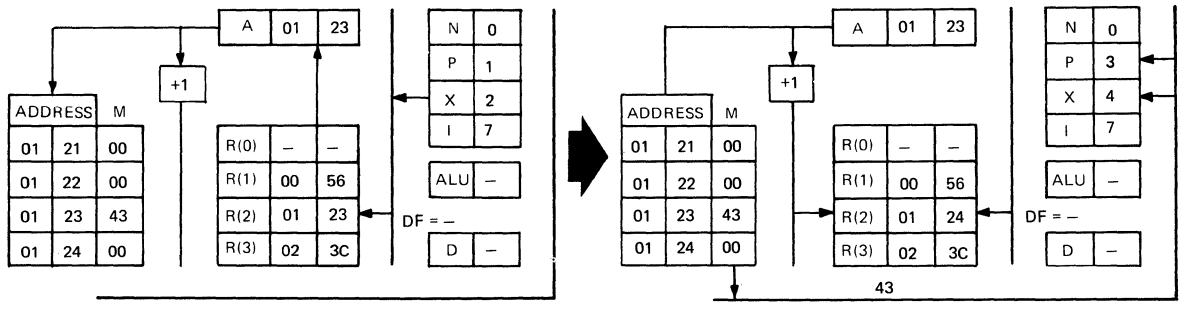


Fig. 38 – Example of instruction 70 – RETURN.

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

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 **Machine Code Programming**.

Instruction Utilization

The following table shows the use of some of the preceding instructions to form a program. This program inputs two bytes from different sources, compares them, and outputs the larger. It then continues to repeat the process.

The first four instructions (at locations 0001,3,4, and 5) set up R(2) as a pointer to address 0000 for I/O and for doing arithmetic. The reader unfamiliar with computers should trace through the program with specific numbers, noting the successive contents of M(0000), D, and R(3).0.

M ADDRESS	M BYTE	OPERATION	COMMENTS
0000	00		Data Storage.
0001	F8	"00" → D	Execution starts at 0001.
0002	00		Immediate data.
0003	A2	D→R(2).0	Sets R(2) = 0000.
0004	B2	D→R(2).1	
0005	E2	2→X	Prepare to input.
0006	68	INPUT 0	Read 1st input data to M(R(2)) = M (0000).
0007	FO	M(R(2))→D	Transfer it to D.
0008	69	INPUT 1	Read 2nd input data to M(R(2)) = M (0000).
0009	A3	D→R(3).0	Save first data.
000A	F7	D-M→D; C→DF	Subtract; set DF to next step.
000B	38	BNF	Branch to 000F if DF = 0,
			ie. if 2nd input greater than
000C	OF		Ist input, otherwise:
000D	83	R(3).0→D	Retrieve first data.
000E	52	D→M(R(2))	Store it at M (0000).
000F	60	OUTPUT 0; R(2)+1	Output larger value; M(R(2))→I/O.
0010	30	BR	Go back to beginning: 0001.
0011	01		Immediate address byte.

Fig. 39 — Example of program for inputting two bytes, compared them, and outputting the larger.

As a more practical and complicated example, the following program segment multiplies two bytes, The multiplicand is assumed to be in memory as addressed by register R(3). The multiplier is in R(5).0, the byte to be added is in R(4).1, and the product will be placed in R(4).1 and R(4).0 —— two bytes.

This program multiplies by shifting the multiplier and product right eight times. Alternatives are to shift the multiplier right and the multiplicand left (by adding it to itself), or the multiplier left and the multiplicand right, or the multiplier and the product both left.

M ADDRESS	M BYTE	OPERATION	COMMENTS
0100	E3	3→X	Prepare for instruction at 010A.
0101	F8	80→D	The bit in 80 (10000000) will be shifted
0102	80		down, using R(4).0 as a counter.
0103	A4	D→R(4).0	
0104	85	R(5).0→D	Fetch multiplier,
0105	F6	D/2→D	shift it,
0106	A5	D→R(5).0	and put it back.
0107	94	R(4).1→D	Fetch partial result.
0108	3B	BNF	If bit shifted into DF is 0, branch to
0109	OD		location 010D; otherwise:

(cont'd on next page)

(cont'd)

M ADDRESS	M BYTE	OPERATION	COMMENTS
010A	F4	D+M(R(3))	Add in multiplicand.
010B	33	BDF	If carry in DF, branch to
010C	10		loc 0110; otherwise:
010D	F6	D/2	Shift the result right,
010E	30	BR	and go to 0113 to shift the rest
010F	13		of result.
0110	F6	D/2	Shift result right.
0111	F9	D OR immed	OR in high bit for carry
0112	80	(data)	from instruction at 010A (NOTE).
0113	B4	D→R(4).1	Store result back.
0114	84	R(4).0→D	Fetch low byte of result.
0115	33	BDF	Delayed branch on shift in
0116	1A		010D or 0110, to 011A.
0117	F6	D/2	Shift low byte,
0118	30	BR	and branch to 001D
0119	1D		to finish shift.
011A	F6	D/2	Shift low byte,
011B	F9	D OR immed	and OR in high bit
011C	80	(data)	from shift of 010D or 0110 (NOTE).
011D	A4	D→R(4).0	Put low byte back.
011E	3B	BNF	Branch back ("loop") if the original
011F	04		80 hasn't shifted thru yet.
0120	—		Product is now ready. Continue to
	—		rest of program.

NOTE: The SHIFT RIGHT instruction will not shift the DF bit into the highest bit of D. These operations essentially restore, if DF=1, a "1" bit into the highest bit of D after a SHIFT RIGHT.

Fig. 40 — Example of program for multiplying two bytes and adding the result to a third byte.

Memory and Control Interface

The reader will find that Appendices B, C, and D are helpful while reading this section. Note that all signal lines except Memory Write (MWR) are made active by holding them low, e.g., when the memory is to be read, $\overline{\text{MREAD}}$ goes low (consistent with T^2L bus conventions).

Memory Interface and Timing

The use of the COSMAC memory interface lines is best described by a specific example. Fig. 41 shows the attachment of a static 1024-byte RAM. The 1024-byte read-write memory comprises eight 1024-bit TA6780 RAM chips. These static single-power-supply chips are easy to use.

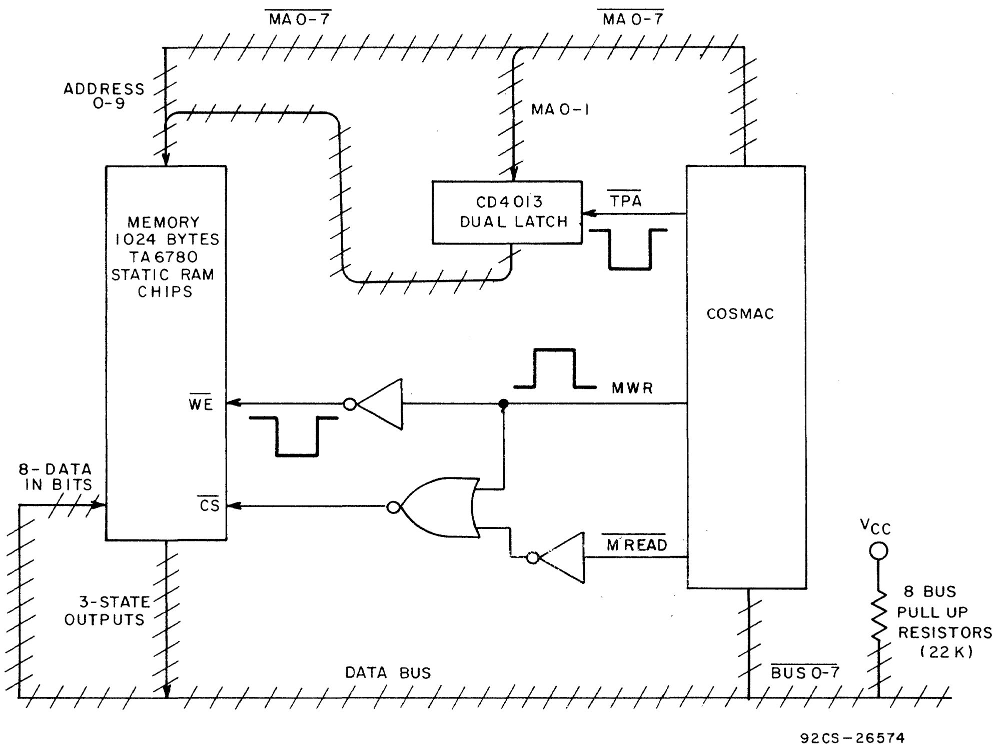
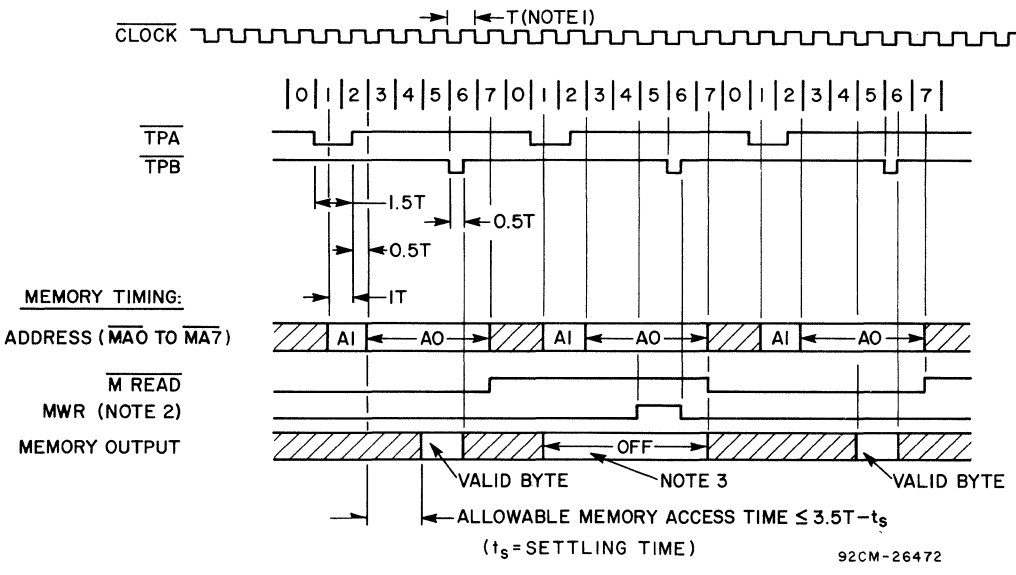


Fig. 41 – Attachment of a static 1024-bit Random-Access-Memory (RAM) to the COSMAC microprocessor.

Ten memory address bits are required to select 1 out of 1024 memory byte locations. The high-order byte (A.1) of a 16-bit COSMAC memory address appears on the memory address lines $\overline{\text{MA0-7}}$ first. The two least-significant bits are strobed into the 2-bit address latch by timing pulse A (TPA). Fig. 42 shows the timing.



NOTES:

1. MINIMUM T DETERMINED BY V_{DD}—NO MAXIMUM T
2. MEMORY WRITE PULSE WIDTH (MWR) ≈ 1.5 T
3. MEMORY OUTPUT "OFF" INDICATES HIGH-IMPEDANCE CONDITION.
4. SHADING INDICATES "DON'T CARE" OR INTERNAL DELAYS DEPENDING ON V_{DD} AND THE CLOCK SPEED.

Fig. 42 – Memory read/write timing.

The low-order byte (A.0) of a 16-bit COSMAC memory address appears on the $\overline{\text{MA0-7}}$ 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. Chip select decoding would have to be added to the latch output for memory expansion. The $\overline{\text{MA0-7}}$ lines may also require buffer circuits to reduce the load on them to achieve high speed.

The state of the MWR and $\overline{\text{MREAD}}$ lines determine whether a byte is to be read from or written into the addressed memory location. COSMAC controls the destination of the memory output byte when it appears on the data bus. It may be strobed into an internal COSMAC register or an external I/O register.

A high $\overline{\text{MREAD}}$ line forces a high-impedance state at the output of the memory. COSMAC or I/O circuits can then place a byte to be stored in memory on the bus. A positive-going MWR pulse will cause the data byte to be written into the addressed memory location.

When a data bit is true ("1"), the corresponding bus line is low; when data is false ("0"), the corresponding line is high. Eight bus pull-up resistors should be provided to place the bus in a known state when it is not being driven.

Other standard RAM types are readily accommodated by the COSMAC interface lines. **Access time** must be consistent with clock frequency; e.g., a 2-MHz clock will require a memory with a maximum access time of 1 microsecond. The time required by the ALU and internal gating is specified in COSMAC data sheets.

If a memory does not have a 3-state high-impedance output, $\overline{\text{MREAD}}$ is useful for driving memory-bus separator gates, otherwise it is used to control 3-state outputs from the addressed memory. A low on $\overline{\text{MREAD}}$ indicates a read cycle; the low $\overline{\text{MREAD}}$ line enables the memory-output-bus gates during the read cycle (see Appendix D, COSMAC Timing).

For various memory systems, $\overline{\text{MREAD}}$ signal and the MWR pulse polarity and width may require modification by external circuitry. Segments of ROM can be attached in the same manner, omitting the write controls. Dynamic RAM's can be used with appropriate refresh circuits. Since COSMAC circuitry is static, the clock may be stopped and restarted for asynchronous memory operation if required.

Control Interfaces: Starting, Stopping, and Loading

COSMAC requires an external single-phase clock. Each machine cycle consists of eight clock pulses. A 2-MHz clock frequency would yield a 4-microsecond machine cycle and result in an operating speed of 125,000 instructions per second.

During normal operation, the COSMAC $\overline{\text{CLEAR}}$ line must be held high. A momentary low on this line places COSMAC in an IDLE state by forcing an IDLE instruction with P=0, R(0)=0000, and IE=1.

The COSMAC $\overline{\text{LOAD}}$ line should also be held high during normal operation. Following $\overline{\text{CLEAR}}$, a low $\overline{\text{LOAD}}$ line permits input bytes to be sequentially loaded into memory beginning at M (0000). Input bytes can be supplied from a keyboard, tape reader, etc. This feature permits direct program loading without the use of external ROM's or PROM's.

Fig. 43 illustrates one method of using the $\overline{\text{CLEAR}}$, $\overline{\text{CLOCK}}$, and $\overline{\text{LOAD}}$ lines to control a COSMAC system. All logic consists of standard 4000-series CMOS circuits. A free-running Pierce crystal oscillator using a single 4007 chip provides a suitable gated clock. A high $\overline{\text{CLEAR}}$ on the control lead of the NAND gate formed from the 4007 gates the oscillator output to the COSMAC CPU. When $\overline{\text{CLEAR}}$ is low, $\overline{\text{CLOCK}}$ remains high. COSMAC design permits an asynchronous relationship between the free-running clock and switch closures; a short first clock pulse will not affect COSMAC operation.

The two toggle switches control the operation of this system. When both switches are off, as shown in Fig. 43, the $\overline{\text{CLEAR}}$ line is held low and the $\overline{\text{CLOCK}}$ line is held high. This $\overline{\text{CLEAR}}$ signal resets COSMAC and can also be used to initialize I/O circuits.

If the LOAD switch is turned on, the $\overline{\text{CLEAR}}$ line will go high, the clock will be started, and the $\overline{\text{LOAD}}$ line will be held low. COSMAC will remain in an IDLE state until a low occurs on the $\overline{\text{INTERRUPT}}$, $\overline{\text{DMA-IN}}$, or $\overline{\text{DMA-OUT}}$ line. Input circuits (not shown) can then activate $\overline{\text{DMA-IN}}$ to load bytes into memory. The low $\overline{\text{LOAD}}$ line causes COSMAC to return to the IDLE state after each input byte is loaded.

Turning off the LOAD switch after a program has been loaded turns off the clock, holds the $\overline{\text{LOAD}}$ line high, and puts the $\overline{\text{CLEAR}}$ line back to a low state. This sequence resets COSMAC once again, putting it in an IDLE state.

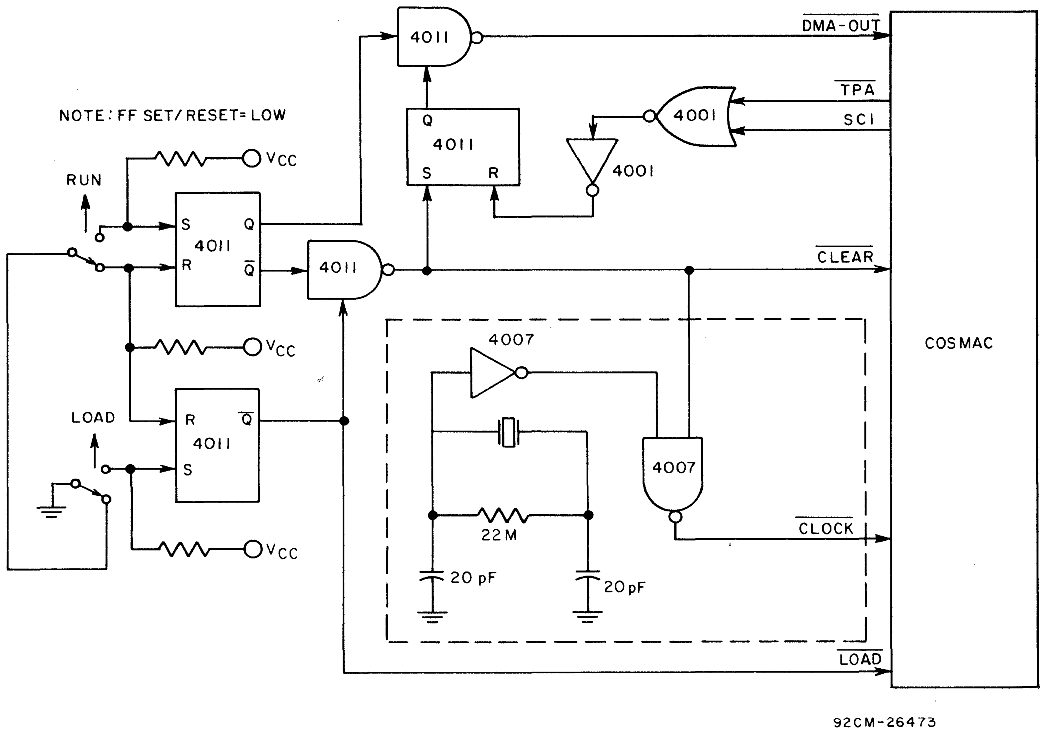


Fig. 43 – Two-switch COSMAC control.

Turning on the RUN switch starts the clock and puts a high on the $\overline{\text{CLEAR}}$ line. Fig. 44 shows the sequence of events that initiates program execution when the RUN switch is turned on. The clock causes a $\overline{\text{TPA}}$ signal each machine cycle. The low on the $\overline{\text{DMA-OUT}}$ line is detected by COSMAC. It responds by performing a DMA cycle (S2), which is described in the section on **I/O interface.** A low on the state code line ($\overline{\text{SCI}}$) indicates that COSMAC is executing the DMA cycle (or interrupt cycle, which would not normally occur at this time) and causes the flip-flop holding the $\overline{\text{DMA-OUT}}$ line low to be reset. In this case, the DMA cycle does nothing more than take COSMAC out of the IDLE state. Since the $\overline{\text{LOAD}}$ line is high, the cycle immediately following the DMA cycle will be a normal instruction fetch operation (SO).

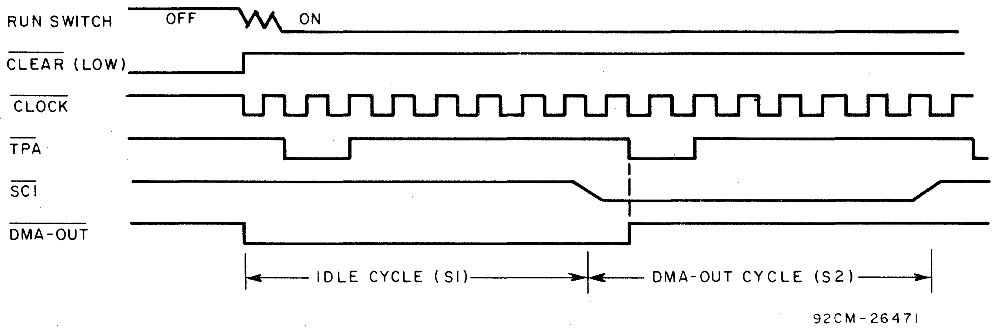


Fig. 44 – START timing.

The previous low on the $\overline{\text{CLEAR}}$ line has set P=O and R(0)=0000. The DMA cycle (S2) caused R(O) to be incremented by 1. The first instruction will, therefore, be fetched from M(0001) and not M(0000). Note that program execution normally begins at M(0001) with R(O) as the program counter. After initiation, program execution continues until an IDLE instruction occurs or the RUN switch is turned off.

The above example represents one method of initiating system operation. The load operation could be eliminated by having a program permanently stored in ROM. Separate CLEAR and RUN momentary contact switches could be used. Program execution could also be initiated by another computer instead of by manual switches. Other oscillators could be used for clock generation.

I/O Interface

Programmed I/O

The following paragraphs indicate a few of the ways in which I/O data transfer can be accomplished under program control. It should be noted that the MREAD signal, discussed in the section on **Memory and Control Interface**. can also be used in conjunction with S1•(I=6) to transfer data from the bus into an I/O device or to gate data from an I/O device onto the bus.

Data output. When I=6 and N=0,1,2,3,4,5,6, or 7, the memory byte addressed by R(X) is placed on the bus. The $\overline{\text{SCO}}$ line goes low and the SCI line goes high to indicate that an I/O instruction cycle is performed. The M(R(X)) byte will appear on the data bus before the timing pulse B ($\overline{\text{TPB}}$) occurs, and will remain on the bus until after the $\overline{\text{TPB}}$ line returns to its high state. Fig. 45 shows how the output instruction might be used to set a byte into a two-hex-digit output display device.

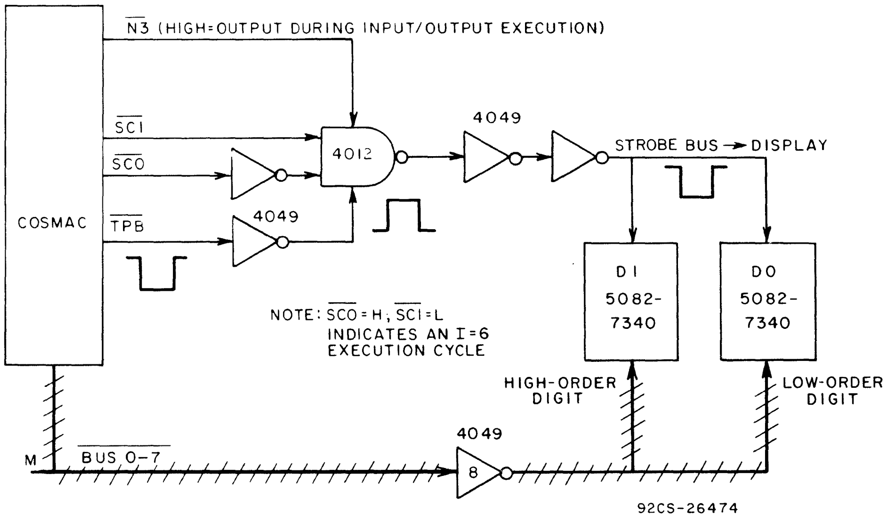


Fig. 45 – Simple output display logic.

Each HP5082-7340 display chip contains a 4-bit register, decoder, and hex LED display. A four-input gate causes the byte from memory to be strobed into the 2-digit hex display during $\overline{\text{TPB}}$ when $\overline{\text{SCO}}$ and $\overline{\text{SCI}}$ indicate that an input/output instruction is being executed. The $\overline{\text{N3}}$ gate input permits the display to be set only when the high-order bit of the N register equals "0". Note that the four N-register bit lines $\overline{\text{NO-3}}$ are high when the corresponding internal N-register bits equal "0". In Fig. 45, any of the 8 output instructions can be used to transfer the M(R(X)) byte to the output display. This logic is suitable if the hex display is the only output device in the system.

If more than one output device is required, NO through N2 can be decoded to specify up to eight different output devices or channels. The $\overline{N3}$ gate input of Fig. 45 might be replaced by a decoded· N=1 signal. This change would permit the display to be set when I=6 and N=1 (a 61 instruction). Instructions 60, 62, 63, 64, 65, 66, and 67 could then designate other devices or channels to receive the output byte.

Data input. The simplest form of input to the COSMAC 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 BRANCH instructions 34, 35, 36, 37, 3C, 3D, 3E, and 3F allow programs to determine the states of these flag lines. Fig. 46 illustrates one method of using a flag line ($\overline{EF1}$ in this case) as a binary input.

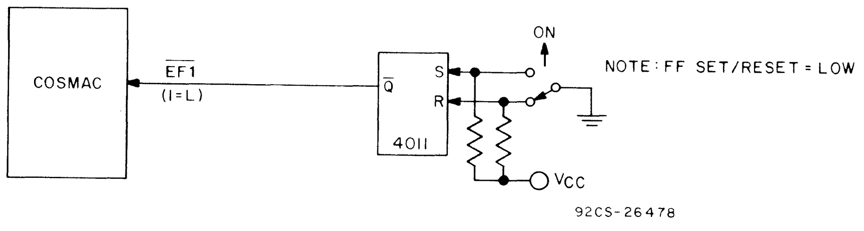


Fig. 46 – Use of a flag time ($\overline{EF1}$) as an input.

Turning on the switch sets $\overline{EF1}$ low. Turning off the switch sets $\overline{EF1}$ high. (The flip-flop eliminates switch bounce.) 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. 45. The switch in Fig. 46 will start and stop the counter.

If the switch is in the "ON" position, counting proceeds (00-99). When it is turned off, counting stops with the current value of the count displayed. Another closure will initiate counting again, started at the value displayed. A portion of a possible "counter program" is shown below.

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		

The switch of Fig. 46 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.

Fig. 47 illustrates the use of the INPUT instruction in conjunction with a flag line. Eight input switches are first set to represent a desired input byte (1=low, 0=high). Momentarily pressing the ENTER switch then places a low on the $\overline{\text{EF1}}$ line. The program monitors the status of this line. When a low is detected, the program branches to an INPUT instruction (I=6 and N3=1).

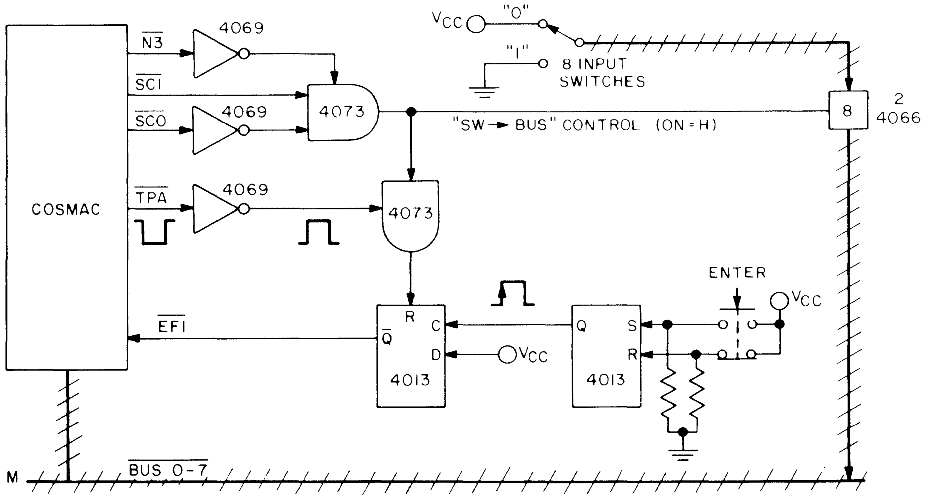


Fig. 47 – Simple byte input logic.

$\overline{\text{SC0}}$ in a low state and $\overline{\text{SC1}}$ in a high state indicate that an input/output byte transfer cycle is being performed. During this cycle the data byte is stored in the memory location addressed by R(X). The 3-input gate in Fig. 47 transfers the state of the eight input switches to the bus through eight 4066 transmission gates. The $\overline{\text{EF1}}$ line is forced high at TPA to assure that only one byte is entered per ENTER switch depression. This logic is suitable only if the single set of eight switches is the only input device in the system.

If more than one input device is required, N0 through N2 can be decoded to specify up to eight different input devices. The N3 signal can be replaced by a decoded N=9 signal. This arrangement would permit the byte to be entered when I=6 and N=9 (a 69 instruction). Instructions 68, 6A, 6B, 6C, 6D, 6E, and 6F could then designate other devices or channels to enter data.

The eight input switches might be replaced by the byte output of a paper-tape reader, keyboard, or other type of input device. The ENTER switch would then be replaced by a strobing signal generated by the input device. The program must sample the flag line and execute input byte transfer instructions at speeds consistent with the input byte transfer rate. Output devices can also utilize flag lines to signal COSMAC that an output byte transfer is required.

The preceding examples have illustrated the use of the four flag lines, the 4-bit N code, the two state code lines, the two timing lines, and the data bus for simple I/O operations. These I/O interface lines can be used to implement more sophisticated I/O systems. Fig. 48 shows one such system.

The N digit provided by the input/output instruction (on $\overline{\text{N0-3}}$) is decoded to provide 16 separate control signals. One of these signals (N=0 in this example) strobes an output byte into an 8-bit I/O device select register. The outputs of this register are decoded to provide selection signals for up to 256 individual I/O devices.

A 60 instruction is executed to place an 8-bit device selection code in the I/O device select register. Subsequent execution of a 61 instruction will send an 8-bit control code to the selected device or channel. Control

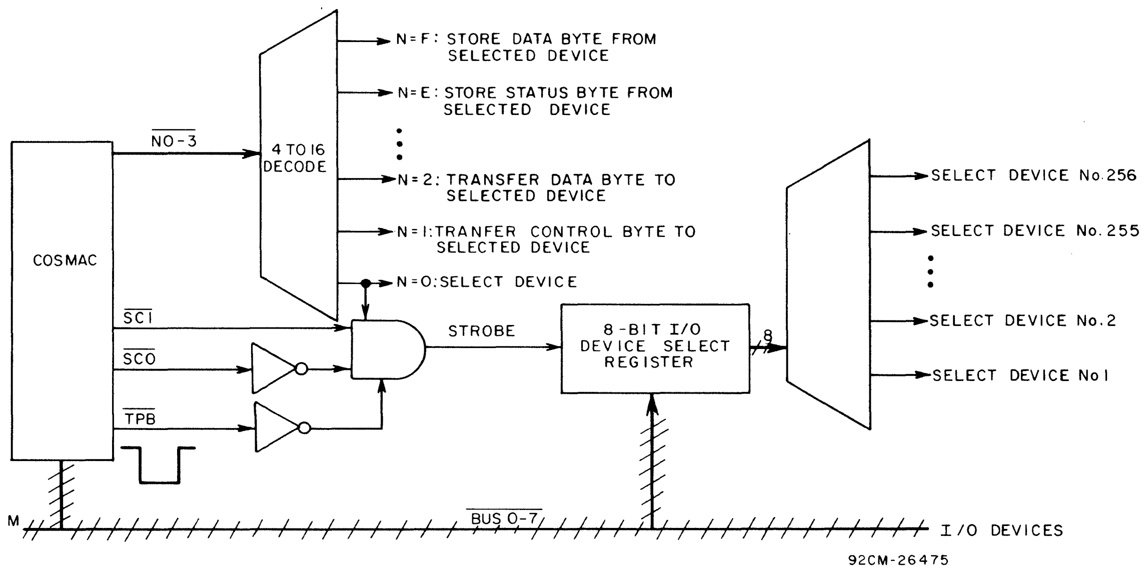

N=F: STORE DATA BYTE FROM SELECTED DEVICE
N=E: STORE STATUS BYTE FROM SELECTED DEVICE
N=2: TRANSFER DATA BYTE TO SELECTED DEVICE
N=1: TRANFER CONTROL BYTE TO SELECTED DEVICE
N=0: SELECT DEVICE
COSMAC
N0-3
4 TO 16 DECODE
SC1
SC0
TPB
STROBE
8-BIT I/O DEVICE SELECT REGISTER
8
SELECT DEVICE No.256
SELECT DEVICE No.255
SELECT DEVICE No.2
SELECT DEVICE No 1
M
BUS 0-7
I/O DEVICES
92CM-26475

Fig. 48 – Two-level I/O system.

codes can be used to start or stop electromechanical devices, set up specific modes of operations, etc. When the 8-bit I/O device select register specifies an output device, execution of a 62 instruction will cause an output data byte transfer to selected device. After an input device is selected, a 6F instruction could be executed to store an input byte in memory. Execution of a 6E instruction is used to obtain a status code byte from a selected device. Instructions 63, 64, 65, 66, 67, 68, 6A, 6B, 6C, and 6D could be used to control other system functions, either directly (ignoring device selection) or under control of the device select register.

A flag line can be shared between several I/O devices by treating it as a bus. Individual device conditions would be gated to the flag bus only when that device is selected.

The above examples indicate only a few of the ways in which I/O instructions can be implemented. The I/O interface line 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 ($\overline{\text{SC0}}$, $\overline{\text{SC1}}$) to indicate a **DMA cycle** (S2).

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

DMA-IN. Fig. 49 illustrates the manner in which a DMA input mode might be implemented. $\overline{\text{TPA}}$ is used to sample the state code to avoid the state transition times (after TPB but before TPA). The input device may be the same devices discussed in conjunction with Fig. 48. In the DMA case, however, each ENTER pulse will put a low on the $\overline{\text{DMA-IN}}$ line instead of on a flag line.

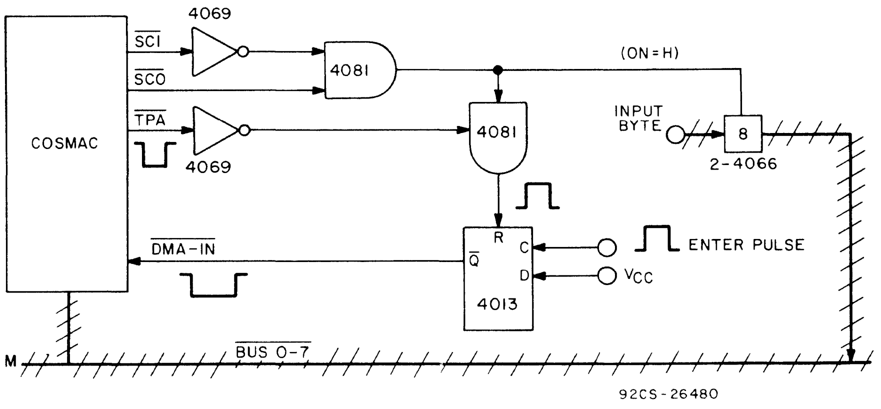
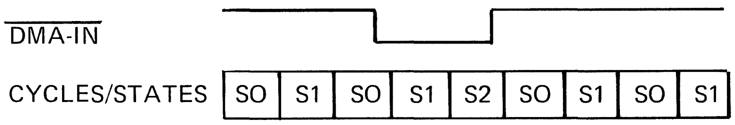
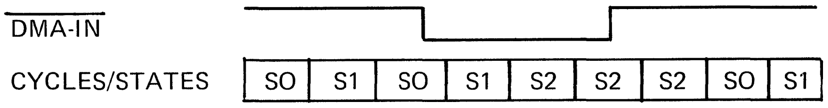


Fig. 49 – DMA input logic.

A low $\overline{\text{DMA-IN}}$ line will automatically modify the normal fetch-execute sequences. If the $\overline{\text{DMA-IN}}$ line goes low during an instruction fetch cycle (SO), then the normally following execute cycle (S1) will still be performed. Following this execute cycle (S1), a special DMA cycle (S2) will be performed. If the $\overline{\text{DMA-IN}}$ line goes low during an instruction execute cycle (S1), then the DMA cycle (S2) will immediately follow. 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 (SO) will be performed following the S2 cycle, as shown below:



If the $\overline{\text{DMA-IN}}$ line remains low, S2 cycles will be performed until the $\overline{\text{DMA-IN}}$ line goes high, as shown below. The DMA mode permits a maximum I/O byte transfer rate of one byte per machine cycle.

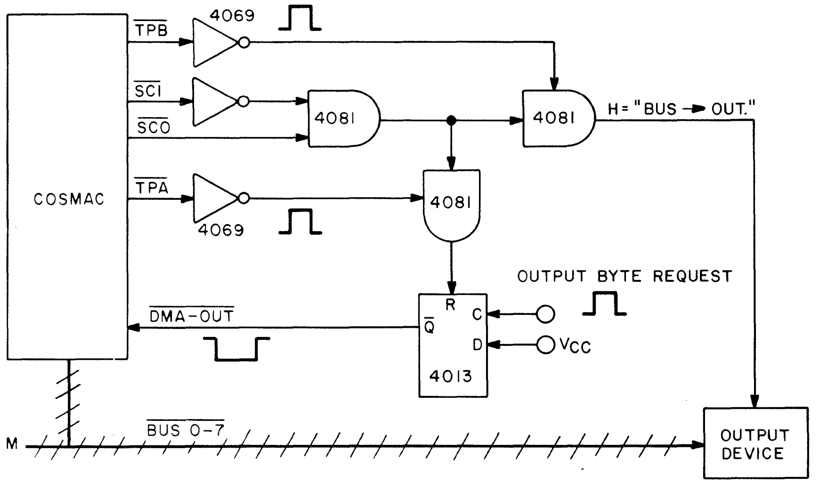


An S2 cycle is indicated by a high $\overline{\text{SCO}}$ line and a low $\overline{\text{SC1}}$ line. This condition is used to place a DMA input byte onto the bus, as shown in Fig. 49. The S2 cycle stores the input byte in memory at the location addressed by R(O). R(O) is then incremented by 1 so 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(O) and memory areas in which input bytes are being stored. It may examine R(O) and the memory area involved to observe the course of the data transfer. The program must also set R(O) to the address of the desired first input byte location in memory before permitting a DMA input operation.

Program Load Facility. The DMA-IN feature, in conjunction with the LOAD and CLEAR signals, provides a built-in **program load mechanism.** A low on the $\overline{\text{CLEAR}}$ line resets R(O) to 0000. If the $\overline{\text{LOAD}}$

line is then held low, the DMA-In logic of Fig. 49 can be used to load a program into memory. Bytes would be stored in sequential memory locations beginning at M(0000). COSMAC will idle between DMA entries, as explained in the section on **Memory and 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(O) on the bus and increments R(O) by 1. DMA output bytes can be strobed into an output device by TPB, as shown in Fig. 50. The program must set R(O) to the address of the first output byte of the desired memory sequence before the DMA transfer requests occur.



92CM-26481

Fig. 50 – DMA output logic.

Interrupt Control

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 low on both the $\overline{\text{SC0}}$ and and $\overline{\text{SC1}}$ lines, as shown below:

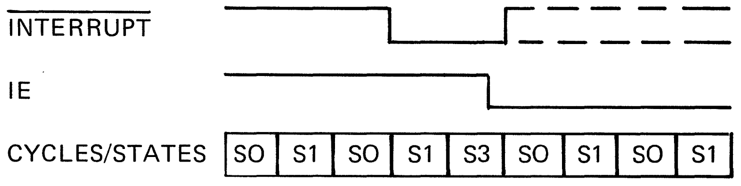


Fig. 51 shows a typical interrupt circuit. The flip-flop is reset during the S3 cycle, but could also be reset by an output instruction.

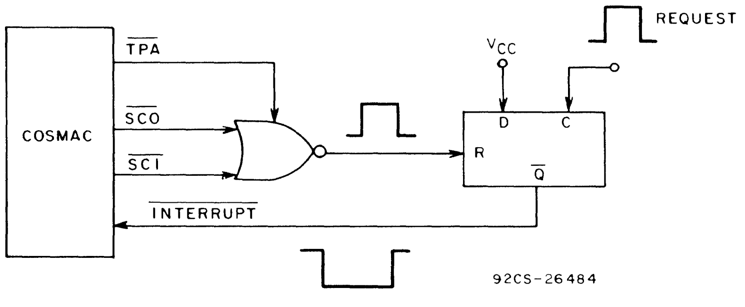


Fig. 51 — Typical interrupt circuit.

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 (SO) 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 were described in the section on **Instruction Repertoire**; their use will be illustrated in the section on **Machine-Code Programming**.

The COSMAC microprocessor also provides a special one-bit register (flip-flop) called Interrupt Enable (IE). When IE is set to "0", the state of the interrupt line is ignored. IE is set to "1" by a low on the $\overline{\text{CLEAR}}$ line. 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 $\overline{\text{INTERRUPT}}$ line stays low. The program must set IE to "1" to permit subsequent interrupts. Sharing the $\overline{\text{INTERRUPT}}$ 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 programs if appropriate conventions are employed, as described in the section on **Machine-Code Programming**.

Machine-Code Programming

Sample System and Program

A simple program will illustrate the use of the COSMAC instructions and provide an example of system design. The demonstration system is a programmed multiple-output sequencer, timer, or controller. Fig. 52 shows a block diagram of the system.

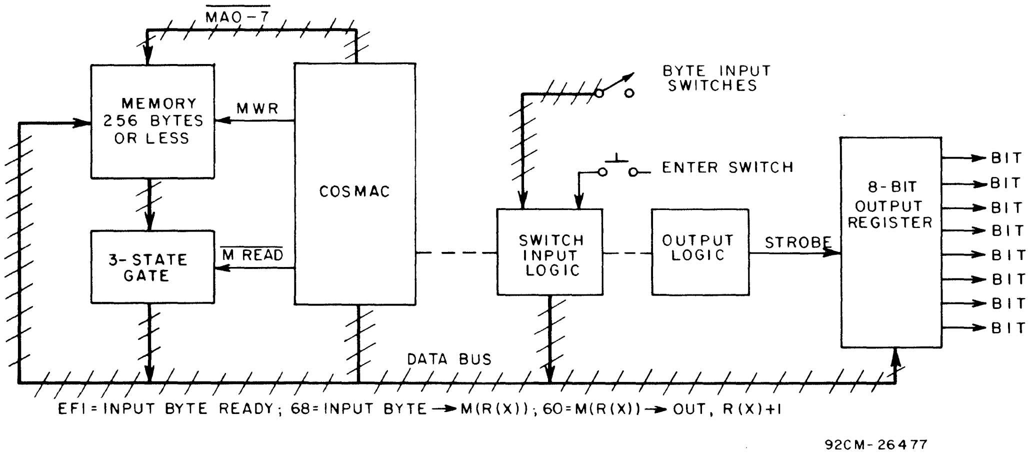


Fig. 52 – Sample microprocessor system.

Because a small memory will suffice for this application, no address latch is required. The program re quires less than 64 bytes and could be stored in a single-chip ROM. RAM capacity of 64 bytes or less is also required. The switch input logic is used to enter initial parameters and could be similar to that shown in Fig. 47. An 8-bit output register could be implemented as shown in Fig. 50.

The 8-bit output register provides 8 output bit lines. Each output line can be programmed to provide a repeating sequence of binary output states. Fig. 53 shows an arbitrary sequence of output states that could be programmed to appear on the four low-order output lines.

Q1, Q2, Q3, and Q4 represent four states for the eight output lines. For example, if Q1=03 (00000011), then the four low-order output lines will have the states shown during the T1 time interval. They will then assume the states shown at Q2 during the T2 time interval. The state of all eight output lines can be represented by a single byte. In the sample program, four bytes are entered to specify the value of the output lines at Q1, Q2, Q3, and Q4. This sequence of states will repeat indefinitely as long as the program runs.

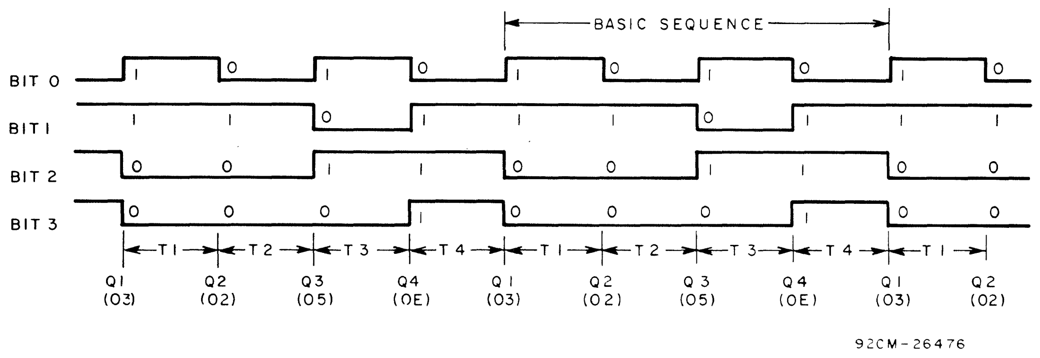


Fig. 53 – Typical output-state sequence.

The time intervals between output-line state changes are specified by another set of four input bytes (T1, T2, T3, and T4). The program can easily be modified to permit a larger number of output-line states to be specified. The repetitive output-register state sequences could be used as a programmable test pulse generator. The output lines might also activate relays for programmable sequencing of up to eight independent external functions or devices.

Fig. 54 outlines the manner in which five scratchpad registers are utilized for this program. R(O) is used as the program counter for the entire program. R(3) is used as a loop counter called LC. R(4) is used as a time interval counter called TC. The four bytes that specify the four sets of output-line values are stored in four sequential memory locations (Q1, Q2, Q3, and Q4 in Fig. 54). These four bytes are followed by the four time-control bytes (T1, T2, T3, and T4). R(A) is used to address the four state bytes and is called QP (state table pointer). R(B) is used to address the four time bytes and is called TP (time table pointer).

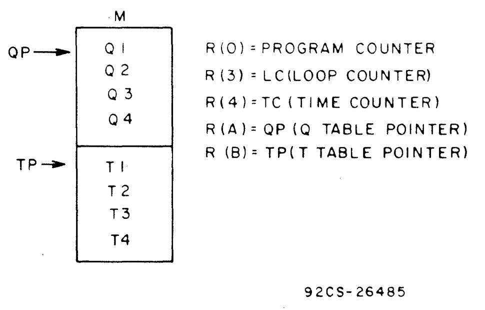


Fig. 54 – Register utilization.

Fig. 55 illustrates the operation of the program in flow-chart form. Step 1 initializes the high-order bytes of R(A) and R(B) to 00. Step 2 puts the memory address of the first state byte (Q1) into R(A). LC is set to 8. The operator must now enter a desired set of four state bytes by means of the byte input switches. The first input bytes will be stored at the Q1 memory location since QP was initially set to address this location.

After the first input byte is stored in memory, QP is incremented by 1 so that it is addressing the Q2 memory location. LC is decremented by 1 so that it will be equal to 7. A branch instruction causes steps 4–5 to be repeated, and the next input byte will be stored at the Q2 memory location. QP will again be incremented and LC decremented. The loop comprising steps 4–5–6–7 will be repeated eight times, causing eight input bytes to be stored in memory. The first four bytes represent desired output line values and will be stored in memory locations Q1—Q4. The second group of four input bytes represent the desired time intervals between output states and will be stored in memory locations T1—T4.

When eight input bytes have been stored, LC will be equal to zero in step 7. In this case, steps 8-9-10 will be performed next. QP is set to address the Q1 memory byte again. TP is set to address the T1 byte. LC is set equal to 4 and step 11 is performed to place the Q1 memory byte into the output register. QP is incremented by 1 so that the Q2 byte will be placed in the output register the next time step 11 is performed.

Step 12 sets TC equal to the value of the T1 byte. TP is incremented by 1 so that TC will be set equal to the value of the T2 byte the next time step 12 is performed.

Step 13 and 14 continually decrement TC until it reaches a value of zero. The time required for TC to reach zero determines the time interval between the current output state and the next output state. This time is a function of the clock frequency, the number of instructions in the loop comprising steps 13—14, and the original value placed in TC.

At the end of the TC counting time, LC is decremented by 1. If LC does not equal zero, the step 11—17 loop is repeated. This loop causes the Q1—Q2—Q3—Q4 output sequence to occur at the specified T1—T2—T3—T4 time intervals. When LC equals zero at step 17, steps 8, 9, and 10 are performed again to repeat the Q1—Q2—Q3—Q4 sequence. This four-state output sequence is repeated until the system is stopped. After applying a clear signal, a new set of state and time bytes can be entered to modify the output sequence.

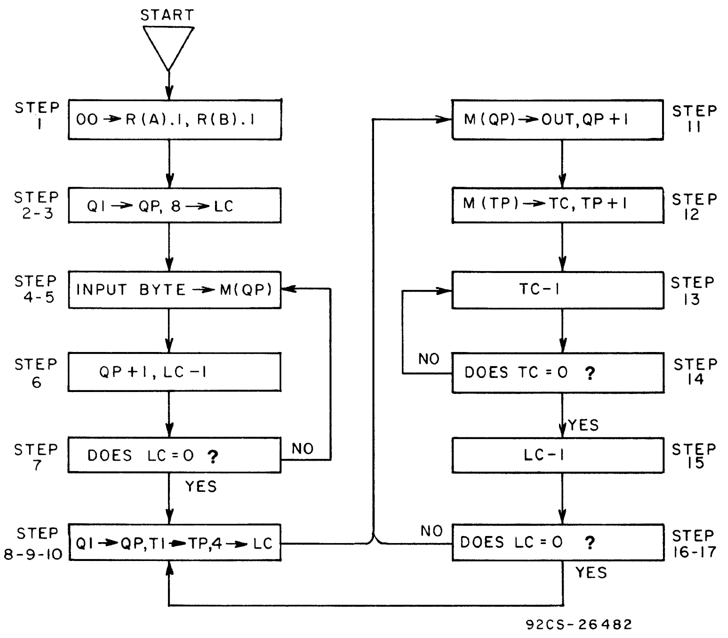


Fig. 55 — Sample program flow chart.

Fig. 56 shows the actual instruction bytes in memory required for the program. A low on the $\overline{\text{CLEAR}}$ line sets P equal to 0 and R(0) equal to 0000. When execution is started, the instruction in memory location 0001 will be fetched and executed as described in the section on **Memory and Control Interface**. The instructions required for each flow-chart step are shown.

Note that in step 12 the time-control byte is placed in the high-order half of R(4) or TC. As a result, the loop comprising steps 13 and 14 will be executed 256 times to decrement the T byte value by 1. Steps 13

M ADDRESS	M BYTE	OPERATION	COMMENTS	
0000	00			
0001	90	R(0).1→D	Initialize higher byte of table pointers	STEP 1
0002	BB	D→R(B).1		
0003	BA	D→R(A).1		
0004	F8	M(R(P))→D	Initialize lower byte of Q table pointer	STEP 2
0005	2A			
0006	AA	D→R(A).0		
0007	F8	M(R(P))→D	Initialize loop counter to 8	STEP 3
0008	08			
0009	A3	D→R(3).0		
000A	3C	IF EF1 ≠1	Loop here until byte ready	STEP 4
000B	0A	GO TO 000A		
000C	EA	A→X	Store input byte	STEP 5
000D	68	IN→M(R(X))		
000E	1A	R(A) + 1	Advance table pointer	STEP 6
000F	23	R(3) – 1	Decrement loop counter	
0010	83	R(3).0→D	Load and test loop counter	STEP 7
0011	3A	IF D≠00		
0012	0A	GO TO 000A		
0013	F8	M(R(P))·D	Reset Q table pointer	STEP 8
0014	2A			
0015	AA	D→R(A).0		
0016	F8	M(R(P))→D	Set T table pointer	STEP 9
0017	2E			
0018	AB	D→R(B).0		
0019	F8	M(R(P))→D	Set loop counter to 4	STEP 10
001A	04			
001B	A3	D→R(3).0		
001C	60	M(R(X))→OUT	Output; advance pointer	STEP 11
001D	4B	M(R(B))→D; R(B) + 1	Load time interval counter	STEP 12
001E	B4	D→R(4).1		
001F	24	R(4) – 1	Decrement time counter	STEP 13
0020	94	R(4).1→D	Load and test time counter	STEP 14
0021	3A	IF D≠00		
0022	1F	GO TO 001F		
0023	23	R(3) – 1	Decrement loop counter	STEP 15
0024	83	R(3).0→D	Load and test loop counter	STEP 16
0025	3A	IF D≠00		
0026	1C	GO TO 001C		
0027	30	BRANCH	Repeat basic sequence	STEP 17
0028	13	TO 0013		
0029	––			
002A	––	Q1	Q Table	
002B	––	Q2	Contains State	
002C	––	Q3	Bytes	
002D	––	Q4		
002E	––	T1	T-Table	
002F	––	T2	Contains Time Count Bytes	
0030	––	T3		
0031	––	T4		

Fig. 56 – Sample program code.

and 14 comprise three instructions, or six machine cycles, or 48 clock cycles. With a 100-kHz clock, each clock cycle is equivalent to 10 x 10^{-6} second. Time intervals between output register states would then equal (256 x 48 x 10 x 10^{-6} x Tn), or 0.123Tn seconds. The maximum time interval that could be specified would be obtained with a T byte value of "FF", which would yield a delay of 256 x 0.123, or 31.5 seconds. Shorter time intervals can be achieved by using R(4).0 as TC. Longer time intervals could be obtained by combining several scratchpad registers into a longer time interval counter. The clock frequency can also be adjusted to provide a desired time interval range.

Detailed study of the sample program shown in Fig. 56 will provide a basic understanding of the use of the individual instructions.

Useful Instructions with X = P

There are three instructions which have particular usefulness when X is set equal to P: the OUTPUT instructions (60–67), the RETURN instruction (70), and the DISABLE instruction (71). Since 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 program counter and once for the execute cycle. As a result, the byte immediately following the instruction byte is the operand byte. For example, if P=3, the sequence will

```
E3          Set X=3.
60          Output a byte from memory.
AD          Immediate byte
--          Next instruction
```

output the byte "AD" by means of the data bus.

This technique is also useful with the RETURN and DISABLE instructions, as discussed later in this section.

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. 57 illustrates a hypothetical interrupt service routine. R(1) is initialized to 0055 before permitting interrupt. R(2) is a **stack pointer**, i.e., it addressed the 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. 57, the stack grew by two bytes as X,P and D were stored on it, and then decreased to its original size when 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 into the stack, 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. 57 is used to store the values of X,P and D associated with the interrupted program. If the interrupting program will modify any other registers (scratchpad or DF), 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.

START HERE

ADDRESS	BYTE	OPERATION	COMMENTS
0053	42	M(R(2))→D, R(2) + 1	RESTORE D
0054	70	M(R(2))→X, P; R(2) + 1; 1→IE	RESTORE X, P AND R(2); ENABLE INTERRUPTS
0055	22	R(2) - 1	DEC STACK POINTER
0056	78	T→M(R(2))	OLD X, P ONTO STACK
0057	22	R(2) 1	DEC STACK POINTER
0058	52	D→M(R(2))	OLD D ONTO STACK
		–	SAVE OTHER REGISTERS IF. REQUIRED
		–	PERFORM "REAL WORK"
		–	REQUESTED BY INTERRUPT
		–	RESTORE OTHER REGS·
		–	PREPARE TO RETURN
	30	GO TO M(0053)	
	53		
–			
–			
–			STORAGE FOR OTHER REG.
00EE			STORAGE FOR D
00EF		STACK	STORAGE FOR T, i.e. OLD X, P
00F0			STACK TOP WHEN INTERRUPTED
–			OTHER STACK ENTRIES
–			
–			

Fig. 57 — Interrupt service routine.

Upon completion of the "real work", return housekeeping must be performed. The contents of registers saved on the stack are now restored. In the example of Fig. 57, program execution branches to location M(0053). R(2) points at M(00EE). The LDA (42) instruction at M(0053) restores the original value of D and R(2) advances to M(00EF). 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(0055) 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 exectuion at the end of the interrupt routine sets IE=1 so that subsequent interrupts are permitted.

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	Set X=3.
70	Return X to 5, P to 3, 1 → IE, R(3)+1.
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.

Branching Between Pages

The branch instructions (I=3) are limited to branches within the currently addressed 256-byte memory page. In larger programs, it is often necessary to be able to branch to any location in memory. The sequence of instructions shown in Fig. 58 illustrates one method of performing such a **long branch**.

ADDRESS	BYTE	OPERATION	COMMENTS
0025	F8	M(R(P))→D	
0026	05		
0027	B4	D→R(4).1	0573→R(4)
0028	F8	M(R(P))→D	
0029	73		
002A	A4	D→R(4).0	
002B	D4	4→P	CONTROL TO R(4)
002C	--		R(3) LEFT POINTING HERE

Fig. 58 – Long branch code.

Initially, R(3) is the program counter (P=3). The sequence of instructions shown puts the 2-byte destination address (0573) into R(4). Setting P=4 then causes a branch to the instruction sequence beginning at M(0573) with R(4) as the program counter. Note that if the sequence using R(4) as program counter ends by setting P=3, execution resumes at 002C, with R(3) as program counter.

Subroutine Techniques

In large programs, a given short sequence of instructions might be used many times. For example, one short sequence might generate random numbers. The required instructions could be rewritten each place in the program that the function is needed. However, this duplication of instructions can consume much memory storage space, especially if the sequence is long. An alternate method is to write the sequence only once as a **subroutine**. Each time that the main program needs a random number it would branch to this subroutine by means of a **subroutine call**, Completion of the subroutine would cause a return to the main program at the instruction following the branch to the subroutine. The use of subroutines reduces the amount of memory required for programs since the subroutine instruction sequence occurs only once instead of each time it is used in a program.

As an example, suppose the designer often wants to execute a long branch. To reduce the code needed for each long branch, one register such as R(4) could be dedicated as the permanent program counter for a long branch subroutine. Its entry address, say 1234, would be loaded once at the beginning of the main program. If R(3) is the main program counter, then a long branch to location 075A would appear as the following subroutine call:

```
D4        4 → P
07        Address to be branched to
5A        will be picked up by subroutine,
```

The subroutine itself would be as shown in Fig. 59.

This subroutine uses three useful devices: (1) The old program counter R(3) is used to pick up arguments for the subroutine –– in this case the new address. (2) A temporary location M(R(2)) was needed since R(3) could not be changed while its old value was still needed to fetch the 5A. (3) By branching to the top before returning to R(3), the subroutine leaves the program counter R(4) ready for another call by the main program, or by other subroutines.

START HERE

M ADDRESS	M BYTE	OPERATION	COMMENTS
1233	D3	3→P	RETURN, LEAVE R(4) OK
1234	43	M(R(3))→D	FETCH HIGH BYTE; R(3) +1
1235	52	D→M(R(2))	SAVE IT ON STACK
1236	43	M(R(3))→D	FETCH LOW BYTE
1237	A3	D→R(3).0	INSERT LOW BYTE
1238	42	M(R(2))→D	FETCH BACK HIGH BYTE; R(2) +1
1239	22	DECR R(2)	RESTORE STACK POINTER
123A	B3	D→R(3).1	INSERT HIGH BYTE
123B	30	BR	BRANCH TO TOP
123C	33		

Fig. 59 — Typical subroutine sequence.

This example points up a tradeoff available to the designer. By dedicating registers and loading them only once, he can shorten subroutine calls to one byte (DN, for appropriate N). The availability of 16 general-purpose registers makes this technique feasible.

In large or complicated programs, subroutines themselves may contain calls upon other subroutines. This technique is called **subroutine nesting.** The mechanism described above works only for those subroutines which do not call other subroutines. The following example illustrates one of many subroutine conventions that can be used in large programs. Register assignment is as follows:

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) — temporary storage; memory pointer

R(3) is used for both main and subroutine pointer counter. A call takes the following form:

D4	4→P
——	High byte of subroutine address
——	Low byte of subroutine address
——	} Optional arguments
——	
——	Next instruction

The D4 instruction transfers program counter control to R(4), which has been initialized to 0101. The call routine is then as shown in Fig. 60.

At the end of the sequence shown in Fig. 60, R(6) points to the first of any optional arguments or, if none, to the next instruction. R(6) can thus be used by the subroutine to pick up the optional arguments or, by the return routine, to get back to the next instruction of the original program.

All subroutines terminate with a D5. The D5 instruction transfers program control to R(5), which has been initialized to 0201. The return routine is illustrated in Fig. 61.

START
HERE

M ADDRESS	M BYTE	OPERATION	COMMENTS
0100	D3	3→P	GO TO SUBROUTINE
0101	96	R(6).1→D	SAVE LAST RETURN POINTER ON DC STACK
0102	52	D→M(R(2))	
0103	22	DECR R(2)	
0104	86	R(6).0→D	
0105	52	D→M(R(2))	
0106	22	DECR R(2)	
0107	93	R(3).1→D	SAVE NEW RETURN POINTER IN R(6)
0108	B6	D→R(6).1	
0109	83	R(3).0→D	
010A	A6	D→R(6).0	
010B	46	M(R(6))→D; R(6) +1	LOAD SUBROUTINE ADDRESS USING RETURN POINTER
010C	B3	D→R(3).1	
010D	46	M(R(6))→D; R(6) +1	
010E	A3	D→R(3).0	
010F	30	BR	GO TO TOP
0110	00		

Fig. 60 – Subroutine call sequence with reloaded entry at 0101.

START
HERE

M ADDRESS	M BYTE	OPERATION	COMMENTS
0200	D3	3→P	RETURN TO ORIGINAL PROGRAM
0201	86	R(6).0→D	FETCH ADDRESS OF NEXT INSTRUCTION OF ORIGINAL PROGRAM
0202	A3	D→R(3).0	
0203	96	R(6) 1→D	
0204	B3	D→R(3).1	
0205	E2	2→X	SET UP STACK POINTER
0206	12	INCREMENT R(2)	
0207	42	M(R(2))→D; R(2) +1	RESTORE LAST RETURN POINTER
0208	A6	D→R(6).0	
0209	FO	M(R(2))→D	
020A	B6	D→R(6).1	
020B	30	BR	GO TO TOP
020C	00		

Note that after a subroutine return using this mechanism, X equal, 2.

Fig. 61 – Subroutine return sequence with preloaded entry at 0201.

Common Program Bugs

COSMAC is quite easy to program. Potential pitfalls are easy to avoid and the simple, consistent set of instructions is easy to understand and use. In general, program debugging will be reduced to a minimun by careful planning and flow-charting prior to machine language coding. Manually going through several flow-chart examples will often turn up bugs that would take much more time to discover in the actual program.

It has been observed, however, that certain types of programming errors occur relatively frequently. Avoiding these programming pitfalls will considerably reduce program debugging time.

One of the most common errors involves the wrong value in X. Setting X to the proper value immediately before use eliminates this potential problem.

The COSMAC programmer must keep track of which register is currently being used as the program counter. He must also keep track of 256-byte memory segments to avoid branching problems, since BRANCH instructions cannot directly branch between 256-byte pages. For long programs, a long branch subroutine should be employed.

Improper scratchpad initialization before use is often a source of program bugs. The programmer should maintain a register utilization list and initialize each register before use.

Program interrupt routines can cause very hard-to-find bugs. For example, if the interrupt service routine uses a SHIFT RIGHT (F6) instruction, DF may or may not be changed during the interrupt routine. If DF is not saved and restored by the interrupt routine, programs will still run properly most of the time. Once in a great while, however, interrupt will occur just before a BRANCH on DF instruction, change DF, and cause a wrong branch. This type of nonrepetitive bug should be avoided at all cost.

Appendix A – Instruction Summary

Register Operations

Code
Assembler Mnemonic (Note)
Name
Operation

I	N			
1	N	INC	INCREMENT	R(N)+1
2	N	DEC	DECREMENT	R(N)–1
8	N	GLO	GET LOW	R(N).0→D
9	N	GHI	GET HIGH	R(N).1→D
A	N	PLO	PUT LOW	D→R(N).0
B	N	PHI	PUT HIGH	D→R(N).1

N=0,1,2, . . .,9,A,B, . . .,E,F (Hexadecimal Notation)

ALU Operations

	I	N			
	F	0	LDX	LOAD BY X	M(R(X))→D
	F	1	OR	OR	M(R(X)) vD→D
	F	2	AND	AND	M(R(X))·D→D
	F	3	XOR	EXCL.OR	M(R(X))⊕ D→D
*	F	4	ADD	ADD	M(R(X))+D→D;C→DF
*	F	5	SD	SUBTRACT D	M(R(X))–D→D;C→DF
*	F	6	SHR	SHIFT RIGHT	SHIFT D RIGHT; LSB→DF;0→MSB
*	F	7	SM	SUBTRACT M	D–M(R(X))→D;C→DF
	F	8	LDI	LOAD IMM	M(R(P))→D;R(P)+1
	F	9	ORI	OR IMM	M(R(P))vD→D;R(P)+1
	F	A	ANI	AND IMM	M(R(P))·D→P;R(P)+1
	F	B	XRI	EXCL.OR IMM	M(R(P))⊕ D→D; R(P)+1
*	F	C	ADI	ADD IMM	M(R(P))+D→D; C→DF;R(P)+1
*	F	D	SDI	SUBT D IMM	M(R(P))–D→D; C→DF;R(P)+1
*	F	F	SMI	SUBT M IMM	D–M(R(P))→D; C→DF;R(P)+1

*These are the only operations that modify DF. DF is set or reset by an ALU carry during add or subtract. Subtraction is by 2's complement: A–B = A+B+1.

Memory Reference

I	N			
4	N	LDA	LOAD ADV	M(R(N))→D;R(N)+1
5	N	STR	STORE	D→M(R(N))

Branching

I	N			
3	0	BR	UNCOND.BR.	M(R(P))→R(P).0
3	2	BZ	BR.IF D=00	M(R(P))→R(P).0 IF D=00/R(P)+1
3	3	BDF	BR.IF DF=1	M(R(P))→R(P).0 IF DF=1/R(P)+1
3	4	B1	BR.IF EF1=1	M(R(P))→R(P).0 IF EF1=1/R(P)+1
3	5	B2	BR.IF EF2=1	M(R(P))→R(P).0 IF EF2=1/R(P)+1
3	6	B3	BR.IF EF3=1	M(R(P))→R(P).0 IF EF3=1/R(P)+1
3	7	B4	BR.IF EF4=1	M(R(P))→R(P).0 IF EF4=1/R(P)+1
3	8	SKP	SKIP	R(P)+1
3	A	BNZ	BR.IF D≠00	M(R(P))→R(P).0 IF D≠00/R(P)+1
3	B	BNF	BR.IF DF=0	M(R(P))→R(P).0 IF DF=0/R(P)+1
3	C	BN1	BR.IF EF1=0	M(R(P))→R(P).0 IF EF I=0/R(P)+1
3	D	BN2	BR.IF EF2=0	M(R(P))→R(P).0 IF EF2=0/R(P)+1
3	E	BN3	BR.IF EF3=0	M(R(P))→R(P).0 IF EF3=0/R(P)+1
3	F	BN4	BR.IF EF4=0	M(R(P))→R(P).0 IF EF4=0/R(P)+1

Note: This type of abbreviated nomenclature is used when programs are designed with the aid of the COSMAC Assembler Simulator/Debugger System, which is available on commercial timesharing systems. Refer to "Program Development Guide for the COSMAC Microprocessor" for details.

Input-Output Byte Transfer

I	N			
6	0	OUT 0	OUTPUT 0	M(R(X))→BUS; R(X)+1;N=0
6	1	OUT 1	OUTPUT 1	M(R(X))→BUS; R(X)+1;N=1
6	2	OUT 2	OUTPUT 2	M(R(X))→BUS; R(X)+1;N=2
6	3	OUT 3	OUTPUT 3	M(R(X))→BUS; R(X)+1;N=3
6	4	OUT 4	OUTPUT 4	M(R(X))→BUS; R(X)+1;N=4
6	5	OUT 5	OUTPUT 5	M(R(X))→BUS; R(X)+1;N=5
6	6	OUT 6	OUTPUT 6	M(R(X))→BUS; R(X)+1;N=6
6	7	OUT 7	OUTPUT 7	M(R(X))→BUS; R(X)+1;N=7
6	8	INP 0	INPUT 0	BUS→M(R(X)); N=8
6	9	INP 1	INPUT 1	BUS→M(R(X)); N=9
6	A	INP 2	INPUT 2	BUS→M(R(X)); N=A
6	B	INP 3	INPUT 3	BUS→M(R(X)); N=B
6	C	INP 4	INPUT 4	BUS→M(R(X)); N=C
6	D	INP 5	INPUT 5	BUS→M(R(X)); N=D
6	E	INP 6	INPUT 6	BUS→M(R(X)); N=E
6	F	INP 7	INPUT 7	BUS→M(R(X)); N=F

Control

I	N			
0	0	IDL	IDLE	WAIT FOR INTERRUPT/ DMA-IN/ DMA-OUT
D	N	SEP	SET P	N→P
E	N	SEX	SET X	N→X
7	0	RET	RETURN	M(R(X))→X, P; R(X)+1;1→IE
7	1	DIS	DISABLE	M(R(X))→X, P; R(X)+1;0→IE
7	8	SAV	SAVE	T→M(R(X))

COSMAC Register Summary

D	8 Bits	D 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	Low-order Instruction Digit
I	4 Bits	High-order Instruction Digit
T	8 Bits	Holds old X, P after Interrupt
IE	1 Bit	Interrupt Enable

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

Interrupt Action: X and P are stored in T after executing current instruction; designator P is set to 1; designator X is set to 2; interrupt enable is reset to 0 (inhibit); and the interrupt request is serviced.

DMA Action: Finish executing current instruction; R(O) points to memory area for data transfer; data is loaded into or read out of memory; and increment R(O).

Note: In the event of concurrent DMA and INTERRUPT requests, DMA has priority.

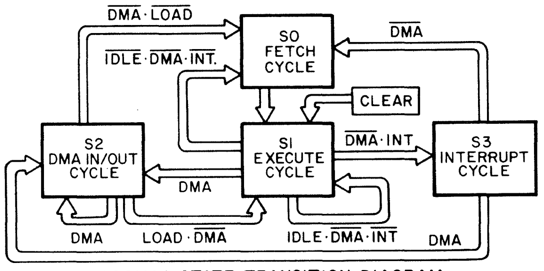
Appendix B — State Sequencing

COSMAC STATES/CYCLES

S0	INSTRUCTION FETCH CYCLE
S1	INSTRUCTION EXECUTE CYCLE
S2	DMA BYTE TRANSFER CYCLE
S3	INTERRUPT CYCLE

STATE CODES

CYCLE TYPE	$\overline{SCI}$	$\overline{SCO}$
S1 I = 6 (I/O INSTR.)	H	L
S2 CYCLE (DMA I/O)	L	H
S3 CYCLE (INTERRUPT)	L	L
OTHER CYCLES	H	H



$\overline{DMA}\cdot\overline{LOAD}$
S0 FETCH CYCLE
$\overline{DMA}$
$\overline{IDLE}\cdot\overline{DMA}\cdot\overline{INT}$
CLEAR
S2 DMA IN/OUT CYCLE
SI EXECUTE CYCLE
$\overline{DMA}\cdot INT$
S3 INTERRUPT CYCLE
DMA
DMA
LOAD · $\overline{DMA}$
IDLE·$\overline{DMA}\cdot\overline{INT}$
DMA

COSMAC STATE TRANSITION DIAGRAM 92CS-26537

START UP & NORMAL INSTRUCTION SEQUENCE:

EXTERNAL →	CLEAR	CLOCK ON			...		DMA OUT		1st INSTR. FETCH FROM M(0001)					
INTERNAL →	——	S1	S1	S1	S1	S1	S1	S2	S0	S1	S0	S1	S0	S1	...
STATE CODE →	——	HH	HH	HH	HH	HH	HH	LH	HH	HH	HH	HL	HH	HH	...

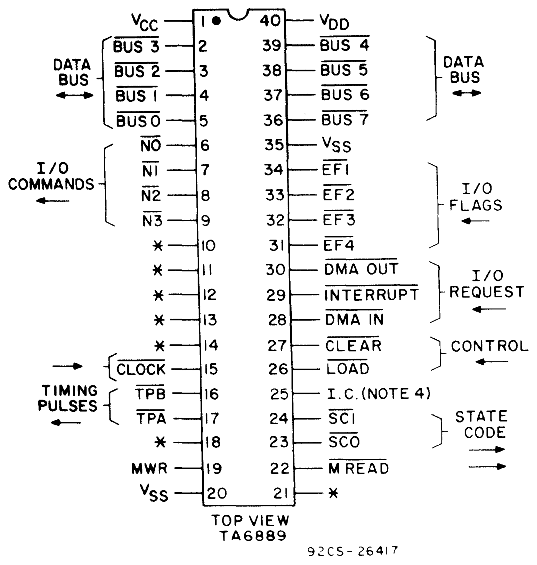
EXECUTION OF I/O BYTE TRANSFER INSTRUCTION

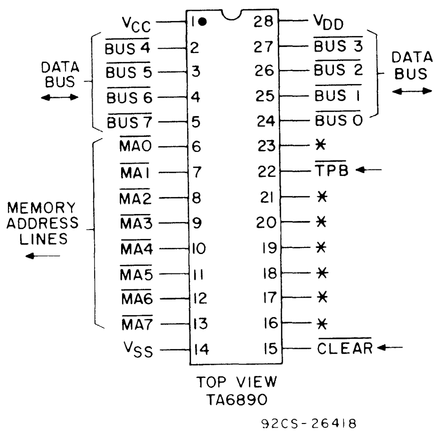
EFFECT OF DMA IN/DMA OUT/INTERRUPT ON NORMAL SEQUENCE

EXTERNAL →	....			DMA IN	...			INTERRUPT		....			DMA OUT	....		
INTERNAL →	....	S0	S1	S0	S1	S2	S0	S1	S3	S0	S1	S0	S1	S2	S0	S1	...
STATE CODE →	....	HH	HH	HH	HH	LH	HH	HH	LL	HH	HH	HH	HH	LH	HH	HH	...

INSTRUCTION TIME
(2 CYCLES)

Appendix C – COSMAC Interface and Chip Connections





Package Interconnections

						*	*	*	*	*		*	*						
TA6889 Pin No.	1	2	3	4	5	10	11	12	13	14	16	18	21	27	36	37	38	39	40
TA6890 Pin No.	1	27	26	25	24	17	18	19	20	23	22	21	16	15	5	4	3	2	28
						*	*	*	*	*		*	*						

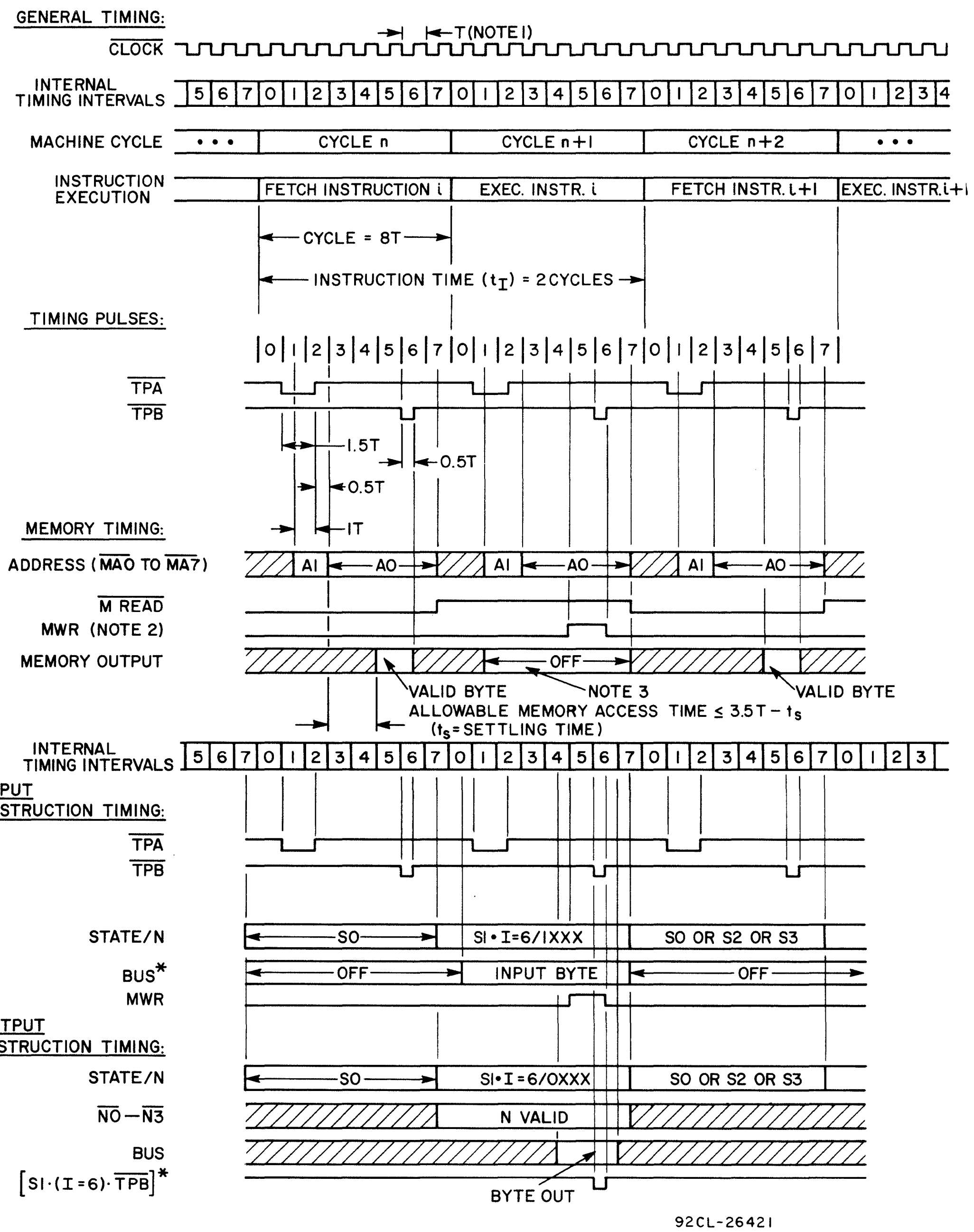
* These pins are for interchip connections only.

Notes:

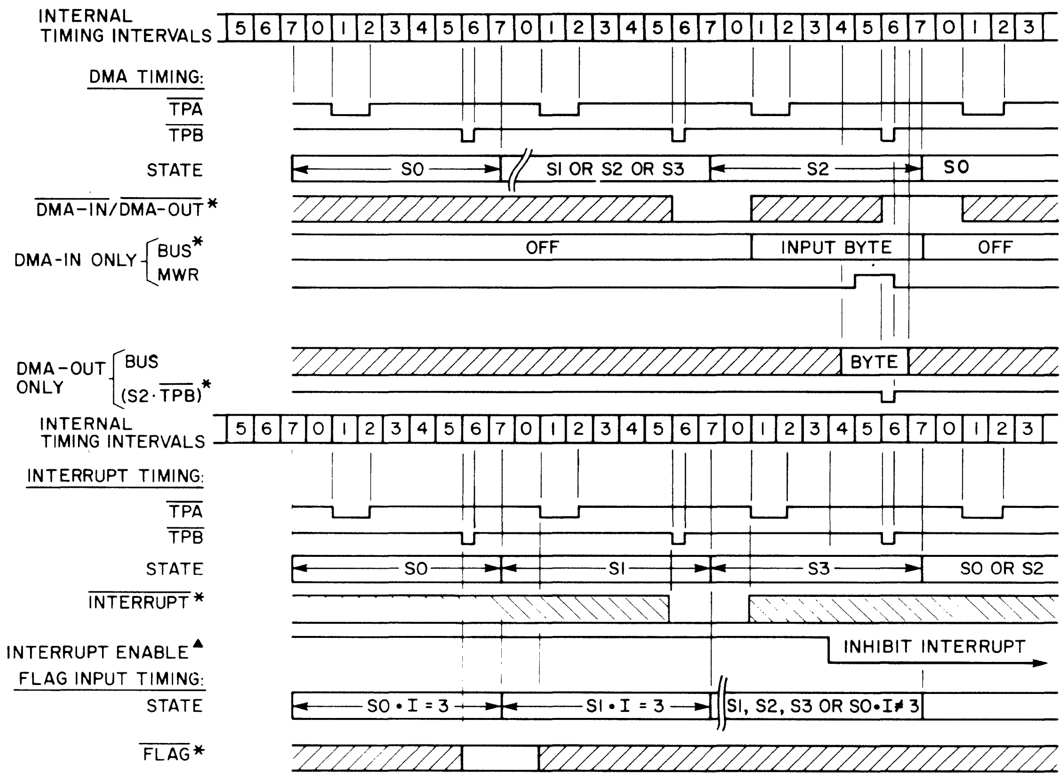
1. Any unused input pins should be connected to V_{DD} or V_{CC}.
2. The Data Bus lines are bi-directional and have three-state outputs. They may be individually connected to V_{CC} through external pull-up resistors (22 kΩ recommended) to prevent floating inputs.
3. All inputs have the same noise immunity and level-shifting capability. All outputs have the same drive capability whether they have three-state outputs or not.
4. Pin 25 of TA6889 is used for an internal connection—**do not use**.

Terminal Assignment Diagrams

Appendix D — COSMAC Timing Summary



92CL-26421



∗ = SIGNAL GENERATED BY USER

▲ = INTERNAL TO COSMAC

NOTES:

1. MINIMUM T DETERMINED BY V_{DD} -- NO MAXIMUM T
2. MEMORY WRITE PULSE WIDTH (MWR) ≈ 1.5 T
3. MEMORY OUTPUT "OFF" INDICATES HIGH-IMPEDANCE CONDITION.
4. SHADING INDICATES "DON'T CARE" OR INTERNAL DELAYS DEPENDING ON V_{DD} AND THE CLOCK SPEED.

92CL-26422

Index