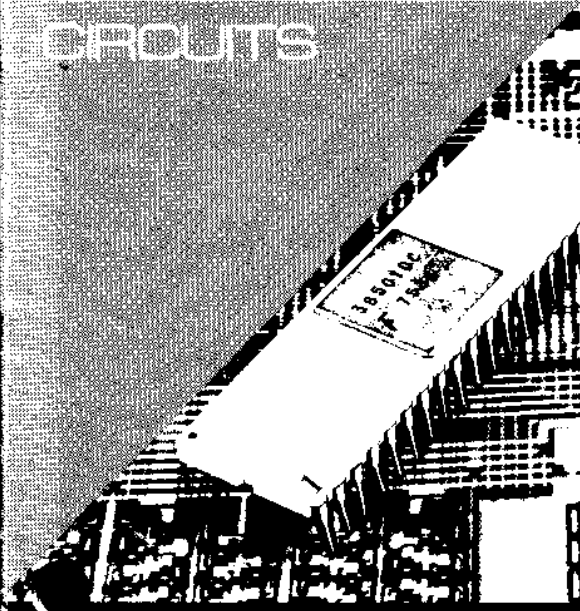


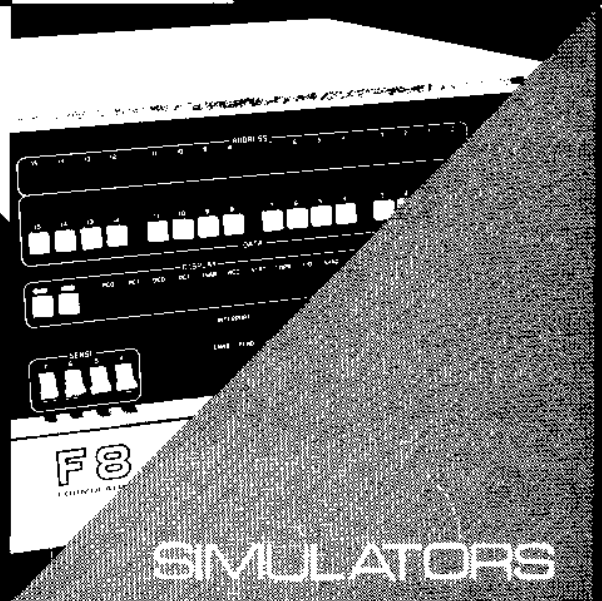
CIRCUITS



MODULES



SYSTEMS



SIMULATORS

GUIDE TO PROGRAMMING

67095664

Copyright Fairchild Camera & Instrument Corp. 1976

Fairchild Micro Systems
1725 Technology Drive, San Jose, CA 95110

FAIRCHILD
MICRO SYSTEMS

TABLE OF CONTENTS

| SECTION | PAGE |
|--|------|
| 1.0 INTRODUCTION | 1-1 |
| 1.1 ASSUMED READER BACKGROUND | 1-1 |
| 1.2 SUPPORTING DOCUMENTATION | 1-1 |
| 2.0 THE F8 MICROPROCESSOR SYSTEM | 2-1 |
| 2.1 WHAT IS A MICROPROCESSOR | 2-1 |
| 2.2 SOME BASIC CONCEPTS | 2-1 |
| 2.2.1 INSTRUCTIONS, PROGRAMS, DATA AND MEMORY | 2-1 |
| 2.2.2 INTERRUPTS | 2-2 |
| 2.2.3 PROGRAMMABLE CLOCKS | 2-2 |
| 2.2.4 DIRECT MEMORY ACCESS | 2-2 |
| 2.2.5 A COMPLETE MICROPROCESSOR SYSTEM | 2-3 |
| 2.3 THE F8 SYSTEM | 2-4 |
| 2.3.1 CHIP AND I/O PORT SELECTION | 2-4 |
| 2.4 THE 3850 CPU | 2-4 |
| 2.4.1 TIMING | 2-4 |
| 2.4.2 CPU REGISTERS | 2-6 |
| 2.4.3 STATUS | 2-6 |
| 2.4.4 3850 INPUT/OUTPUT | 2-7 |
| 2.5 THE 3851 PSU | 2-7 |
| 2.5.1 3851 TIMING | 2-7 |
| 2.5.2 3851 REGISTERS | 2-8 |
| 2.5.3 3851 INPUT/OUTPUT | 2-9 |
| 2.5.4 3851 LOCAL TIMER AND INTERRUPT | 2-9 |
| 2.6 THE 3852 DYNAMIC MEMORY INTERFACE | 2-9 |
| 2.6.1 3852 TIMING | 2-9 |
| 2.6.2 3852 REGISTERS | 2-9 |
| 2.6.3 3852 DIRECT MEMORY ACCESS AND MEMORY REFRESH | 2-10 |
| 2.7 THE 3853 STATIC MEMORY INTERFACE | 2-10 |
| 2.8 THE 3854 DIRECT MEMORY ACCESS | 2-11 |
| 2.8.1 3854 REGISTERS | 2-11 |
| 2.8.2 DMA CONTROL CODES | 2-12 |
| 3.0 F8 PROGRAMS | 3-1 |
| 3.1 FLOWCHARTING | 3-1 |
| 3.2 ASSIGNING MEMORY | 3-1 |
| 3.3 SOURCE AND OBJECT PROGRAMS | 3-2 |
| 4.0 ASSEMBLY LANGUAGE SYNTAX | 4-1 |
| 4.1 INSTRUCTION TYPES | 4-1 |
| 4.1.1 COMMENTS | 4-1 |
| 4.1.2 EXECUTABLE INSTRUCTIONS | 4-1 |
| 4.1.3 ASSEMBLER DIRECTIVES | 4-1 |
| 4.2 INSTRUCTION FIELDS | 4-1 |
| 4.2.1 LABEL FIELD | 4-1 |
| 4.2.2 MNEMONIC FIELD | 4-2 |
| 4.2.3 OPERAND FIELD | 4-2 |
| 4.2.4 COMMENT FIELD | 4-2 |
| 4.2.5 ALIGNING FIELDS | 4-3 |
| 4.3 LANGUAGE COMPONENTS | 4-4 |
| 4.3.1 VALID CHARACTERS | 4-4 |
| 4.3.2 CONSTANTS | 4-4 |
| 4.3.3 SYMBOLS | 4-5 |
| 4.3.4 EXPRESSIONS | 4-5 |
| 5.0 ASSEMBLER DIRECTIVES | 5-1 |
| 5.1 BASE - SELECT LISTING NUMERIC BASE | 5-1 |
| 5.2 DC - DEFINE CONSTANT | 5-1 |

TABLE OF CONTENTS (CONT'D).

| SECTION | PAGE |
|------------|---|
| 5.3 | EJECT - EJECT CURRENT LISTING PAGE..... 5-1 |
| 5.4 | END - END OF ASSEMBLY..... 5-2 |
| 5.5 | EQU - EQUATE A SYMBOL TO A NUMERIC VALUE..... 5-2 |
| 5.5.1 | A COMPARISON OF THE EQU AND DC DIRECTIVES..... 5-2 |
| 5.6 | MAXCPU - SPECIFY MAXIMUM CPU TIME..... 5-2 |
| 5.7 | ORG - ORIGIN A PROGRAM..... 5-3 |
| 5.8 | SYMBOL - ASSEMBLER PROVIDE A SYMBOL TABLE..... 5-3 |
| 5.9 | TITLE - PRINT A TITLE AT THE HEAD OF THE ASSEMBLER LISTING..... 5-3 |
| 5.10 | XREF - ASSEMBLER PROVIDE A SYMBOL CROSS REFERENCE LISTING..... 5-3 |
| 5.11 | WHEN TO USE ASSEMBLER DIRECTIVES..... 5-3 |
| 6.0 | THE INSTRUCTION SET..... 6-1 |
| 6.1 | ADC - ADD ACCUMULATOR TO DATA COUNTER..... 6-2 |
| 6.2 | AI - ADD IMMEDIATE TO ACCUMULATOR..... 6-2 |
| 6.3 | AM - ADD (BINARY) MEMORY TO ACCUMULATOR..... 6-3 |
| 6.4 | AMD - DECIMAL ADD, MEMORY TO ACCUMULATOR..... 6-3 |
| 6.5 | AS - BINARY ADDITION, SCRATCHPAD MEMORY TO ACCUMULATOR..... 6-4 |
| 6.6 | ASD - DECIMAL ADD, SCRATCHPAD TO ACCUMULATOR..... 6-4 |
| 6.7 | BRANCH INSTRUCTIONS..... 6-5 |
| 6.7.1 | BF - BRANCH PM FALSE..... 6-7 |
| 6.7.2 | BT - BRANCH ON TRUE..... 6-7 |
| 6.8 | CI - COMPARE IMMEDIATE..... 6-7 |
| 6.9 | CLR - CLEAR ACCUMULATOR..... 6-7 |
| 6.10 | CM - COMPARE MEMORY TO ACCUMULATOR..... 6-7 |
| 6.11 | COM - COMPLEMENT..... 6-8 |
| 6.12 | DCI - LOAD DC IMMEDIATE..... 6-8 |
| 6.13 | DI - DISABLE INTERRUPT..... 6-8 |
| 6.14 | DS - DECREMENT SCRATCHPAD CONTENT..... 6-8 |
| 6.15 | EI - ENABLE INTERRUPT..... 6-8 |
| 6.16 | IN - INPUT LONG ADDRESS..... 6-8 |
| 6.17 | INC - INCREMENT ACCUMULATOR..... 6-9 |
| 6.18 | INS - INPUT SHORT ADDRESS..... 6-10 |
| 6.19 | JMP - BRANCH IMMEDIATE..... 6-10 |
| 6.20 | LI - LOAD IMMEDIATE..... 6-10 |
| 6.21 | LIS - LOAD IMMEDIATE SHORT..... 6-10 |
| 6.22 | LISL - LOAD LOWER OCTAL DIGIT OF ISAR..... 6-10 |
| 6.23 | LISU - LOAD UPPER OCTAL DIGIT OF ISAR..... 6-11 |
| 6.24 | LM - LOAD ACCUMULATOR FROM MEMORY..... 6-11 |
| 6.25 | LNK - LINK CARRY TO THE ACCUMULATOR..... 6-11 |
| 6.26 | LR - LOAD REGISTER..... 6-11 |
| 6.27 | NI - AND IMMEDIATE..... 6-11 |
| 6.28 | NM - LOGICAL AND FROM MEMORY..... 6-12 |
| 6.29 | NOP - NO OP..... 6-13 |
| 6.30 | NS - LOGICAL AND FROM SCRATCHPAD MEMORY..... 6-13 |
| 6.31 | OI - OR IMMEDIATE..... 6-13 |
| 6.32 | OM - LOGICAL "OR" FROM MEMORY..... 6-13 |
| 6.33 | OUT - OUTPUT LONG ADDRESS..... 6-14 |
| 6.34 | OUTS - OUTPUT SHORT ADDRESS..... 6-14 |
| 6.35 | PI - CALL TO SUBROUTINE IMMEDIATE..... 6-14 |
| 6.36 | PK - CALL TO SUBROUTINE DIRECT AND RETURN FROM SUBROUTINE DIRECT..... 6-14 |
| 6.37 | POP - RETURN FROM SUBROUTINE..... 6-15 |
| 6.38 | SL - SHIFT LEFT..... 6-15 |
| 6.39 | SR - SHIFT RIGHT..... 6-15 |
| 6.40 | ST - STORE TO MEMORY..... 6-15 |
| 6.41 | XDC - EXCHANGE DATA COUNTERS..... 6-16 |
| 6.42 | XI - EXCLUSIVE OR IMMEDIATE..... 6-16 |
| 6.43 | XM - EXCLUSIVE OR FROM MEMORY..... 6-16 |
| 6.44 | XS - EXCLUSIVE OR FROM SCRATCHPAD..... 6-16 |

| SECTION | PAGE |
|---|------|
| 7.0 PROGRAMMING TECHNIQUES | 7-1 |
| 7.1 MANIPULATING DATA IN THE SCRATCHPAD | 7-1 |
| 7.1.1 SIMPLE SCRATCHPAD BUFFER OPERATIONS | 7-1 |
| 7.1.2 INCREMENTING UP, AND DECREMENTING DOWN SCRATCHPAD BUFFERS | 7-1 |
| 7.1.3 USING SCRATCHPAD REGISTERS AS COUNTERS | 7-3 |
| 7.1.4 USING SCRATCHPAD REGISTERS FOR SHORT DATA OPERATIONS | 7-3 |
| 7.2 ROM, RAM AND DATA TABLES | 7-4 |
| 7.2.1 READING DATA OUT OF TABLES IN ROM | 7-4 |
| 7.2.2 ACCESSING DATA TABLES IN RAM | 7-4 |
| 7.3 SUBROUTINES | 7-6 |
| 7.3.1 THE CONCEPT OF A SUBROUTINE | 7-6 |
| 7.3.2 SUBROUTINE PROGRAM STEPS | 7-7 |
| 7.3.3 SIMPLE SUBROUTINE CALLS AND RETURNS | 7-7 |
| 7.3.4 NESTED SUBROUTINES | 7-8 |
| 7.3.5 MULTIPLE SUBROUTINE RETURNS | 7-12 |
| 7.3.6 PASSING PARAMETERS | 7-13 |
| 7.4 MACROS | 7-14 |
| 7.4.1 DEFINING AND USING MACROS | 7-14 |
| 7.4.2 MACROS WITH PARAMETERS | 7-14 |
| 7.4.3 RULES FOR DEFINING AND USING MACROS | 7-15 |
| 7.4.4 WHEN MACROS SHOULD BE USED | 7-15 |
| 7.5 JUMP TABLES | 7-15 |
| 7.5.1 JUMP TABLE USING JUMP INSTRUCTIONS | 7-16 |
| 7.5.2 JUMP TABLE USING ADDRESS CONSTANTS | 7-16 |
| 7.5.3 JUMP TABLE USING DISPLACEMENT TABLES | 7-16 |
| 7.6 STATUS, BITS AND BOOLEAN LOGIC | 7-17 |
| 7.6.1 MANIPULATING INDIVIDUAL BITS | 7-17 |
| 7.6.2 TESTING FOR STATUS | 7-18 |
| 7.7 POWERING UP AND STARTING PROGRAM EXECUTION | 7-18 |
| 8.0 INPUT/OUTPUT PROGRAMMING | 8-1 |
| 8.1 PROGRAMMED I/O | 8-1 |
| 8.1.1 POLLING ON STATUS | 8-1 |
| 8.1.2 DATA, STATUS AND CONTROLS | 8-2 |
| 8.1.3 PARALLEL DATA AND CONTROL PORTS | 8-3 |
| 8.2 INTERRUPT I/O | 8-3 |
| 8.2.1 THE INTERRUPT SEQUENCE | 8-3 |
| 8.2.2 ENABLING AND DISABLING INTERRUPTS | 8-4 |
| 8.2.3 INTERRUPT PRIORITIES | 8-4 |
| 8.2.4 PROGRAM RESPONSE TO AN INTERRUPT | 8-5 |
| 8.2.5 MAKING 3851 PSU INTERRUPT ADDRESS PROGRAMMABLE | 8-5 |
| 8.2.6 SIMPLE I/O INTERRUPTS | 8-5 |
| 8.2.7 A SAMPLE PROGRAM | 8-6 |
| 8.3 LOCAL TIMERS (PROGRAMMABLE TIMERS) | 8-7 |
| 8.3.1 LOCAL TIMER I/O PORTS | 8-7 |
| 8.3.2 PROGRAMMING LOCAL TIMERS | 8-8 |
| 8.3.3 A PROGRAMMING EXAMPLE – THE TIME OF DAY | 8-8 |
| 8.4 DIRECT MEMORY ACCESS | 8-9 |
| 8.4.1 WHEN TO USE DMA | 8-9 |
| 8.4.2 PROGRAMMING DMA | 8-10 |
| 8.4.3 CATCHING DMA ON THE FLY | 8-11 |
| 9.0 PROGRAM OPTIMIZATION | 9-1 |
| 9.1 COUNTING CYCLES AND BYTES | 9-1 |
| 9.2 ELEMENTARY OPTIMIZATION TECHNIQUES | 9-1 |
| 9.2.1 SCRATCHPAD AND RAM MEMORY | 9-1 |
| 9.2.2 IMMEDIATE INSTRUCTIONS | 9-1 |

TABLE OF CONTENTS (CONT'D)

| SECTION | PAGE |
|---|-------------|
| 9.2.3 SHORT INSTRUCTIONS | 9-7 |
| 9.2.4 USE OF DS INSTRUCTION TO DECREMENT AND TEST | 9-7 |
| 9.2.5 USE OF THE BR7 INSTRUCTION | 9-7 |
| 9.3 PROGRAMMING FOR SPEED OR MEMORY ECONOMY | 9-2 |
| 9.3.1 MACROS AND SUBROUTINES | 9-2 |
| 9.3.2 TABLE LOOKUPS VERSUS DATA MANIPULATION | 9-3 |
| 10.0 SOME USEFUL PROGRAMS | 10-1 |
| 10.1 GENERATING TEXT | 10-1 |
| 10.1.1 SIMPLE AND DEDICATED TEXT PROGRAMS | 10-1 |
| 10.1.2 UNPACKING DECIMAL DIGITS | 10-1 |
| 10.1.3 VARIABLE TEXT | 10-1 |
| 10.2 MULTIBYTE ADDITION AND SUBTRACTION | 10-2 |
| 10.2.1 16-BIT, BINARY ADDITION AND SUBTRACTION | 10-2 |
| 10.2.2 MULTIBYTE BINARY OR DECIMAL ADDITION AND SUBTRACTION | 10-3 |
| 10.3 MULTIPLICATION | 10-3 |
| 10.4 DIVISION | 10-5 |
| APPENDIX A BINARY NUMBER SYSTEM | A-1 |
| APPENDIX B ASCII CODES | B-1 |
| APPENDIX C CONVERSION TABLES/TIMER COUNTS | C-1 |
| APPENDIX D INSTRUCTION SUMMARY | D-1 |

LIST OF ILLUSTRATIONS

| FIGURE | PAGE |
|---|------|
| 2-1 Multifunction Logic Device | 2-1 |
| 2-2 Data and Instruction Paths in a Multifunction Logic Device | 2-1 |
| 2-3 Program P Being Interrupted to Execute Program R | 2-2 |
| 2-4 Logical Components, Data Paths and Control Paths in any Microprocessor System | 2-3 |
| 2-5 F8 Microprocessor System Configurations | 2-4 |
| 2-6 Logical Functions of the 3850 CPU | 2-5 |
| 2-7 Instruction Timing | 2-5 |
| 2-8 3850 CPU Programmable Registers | 2-6 |
| 2-9 Logical Functions of the 3851 PSU | 2-8 |
| 2-10 Logical Functions of the 3852 DMI Device | 2-10 |
| 2-11 Logical Functions of the 3853 SMI Device | 2-11 |
| 2-12 Logical Functions of the 3854 DMA Device | 2-12 |
| 3-1 Flowchart for a Program to Move Data from One RAM Buffer to Another | 3-1 |
| 3-2 Flowchart for Program to Add Two Multibyte Numbers and Output the Result | 3-2 |
| 3-3 Source and Object Programs | 3-3 |
| 4-1 Four Comment Lines (Shaded) in a Source Program | 4-1 |
| 4-2 Label Fields (Shaded) in a Source Program | 4-2 |
| 4-3 Mnemonic Field (Vertical Shaded) in a Source Program | 4-2 |
| 4-4 Operand Fields (Shaded) in a Source Program | 4-3 |
| 4-5 Comment Fields (Shaded) in a Source Program | 4-3 |
| 4-6 Source Program with Unaligned Fields | 4-3 |
| 4-7 Symbols in a Source Program | 4-5 |
| 5-1 Assembler Directives (Shaded) in a Source Program | 5-1 |
| 6-1 Generation of a Displacement Object Program Byte in Response to a Forward Branch | 6-5 |
| 6-2 Generation of a Displacement Object Program Byte in Response to a Backward Branch | 6-6 |

TABLE OF CONTENTS (Cont'd)

| FIGURE | | PAGE |
|--------|--|------|
| 7-1 | Use of H, Q and DC1 Registers to Hold Three Buffer Addresses | 7-6 |
| 7-2 | Subroutine, as Compared to a Macro | 7-7 |
| 7-3 | Scratchpad Stack | 7-10 |
| 8-1 | Daisy Chaining and Interrupt Priority Determination | 8-4 |
| 8-2 | Two Levels of Interrupt | 8-4 |
| 8-3 | Two Devices Servicing a Keyboard to Cassette Application | 8-6 |
| 8-4 | How BUFC and BUFD are used to Control DMA Operations | 8-10 |
| 9-1 | Counting Cycles and Bytes | 9-2 |

LIST OF TABLES

| TABLE | | PAGE |
|-------|--|------|
| 2-1 | Summary of Status Bits | 2-6 |
| 2-2 | Hexadecimal Addresses of Four I/O Ports used as Registers by Four 3854 DMA Registers | 2-11 |
| 4-1 | Summary of Restricted Characters | 4-4 |
| 6-1 | Operand Symbols | 6-1 |
| 6-2 | Operands Referencing Scratchpad Memory, as Specified by Symbol Sreg | 6-1 |
| 6-3 | Branch Conditions | 6-6 |
| 6-4 | Branch Conditions for BF Instruction | 6-6 |
| 6-5 | Branch Conditions for BT Instruction | 6-7 |
| 6-6 | I/O Port Address Assignments | 6-9 |
| 6-7 | LR Instruction Operand Definitions | 6-12 |
| 7-1 | Scratchpad Memory Utilization | 7-1 |
| 7-2 | Use of a Memory Stack for Executing Multiple Level Subroutines | 7-10 |
| 8-1 | Contents of Interrupt Control I/O Ports | 8-4 |

INTRODUCTION

This manual explains how to write programs for the Fairchild F8 microprocessor system, and how these F8 programs cause a microprocessor system to function as a discrete logic replacement.

The Fairchild F8 family of logic devices consists of a Central Processing Unit and a number of complementary devices, manufactured using n-channel Isoplanar MOS technology. Components of the F8 family include the following devices:

- 1) The 3850 Central Processing Unit (CPU)
- 2) The 3851 Program Storage Unit (PSU)
- 3) The 3852 Dynamic Memory Interface (DMI)
- 4) The 3853 Static Memory Interface (SMI)
- 5) The 3854 Direct Memory Access (DMA)

Complete microprocessor based systems may vary in size and complexity from as little as two devices—the 3850 CPU and the 3851 PSU—to large systems incorporating the above five devices, plus any standard static and/or dynamic Random Access Memory (RAM) devices.

The following are some general characteristics of this microprocessor device set:

- 8-bit data organization
- 2 μ s instruction cycle time
- Over 70 microprocessor instructions
- 64 general purpose registers in the CPU
- Binary and decimal arithmetic, and logic functions
- Up to 65,536 bytes of ROM and RAM, in any combination
- No need for special external interface devices
- Internal, programmable real time clocks
- Internal power on and reset logic
- Multi-level interrupt handling
- Clock and timing circuits

1.1 ASSUMED READER BACKGROUND

This manual has been written for logic designers with little or no background in programming.

The reader is assumed to understand the following:

- 1) Binary, octal, binary coded decimal and hexadecimal number systems
- 2) Signed and unsigned binary arithmetic
- 3) Boolean logic
- 4) ASCII and EBCDIC character codes

For readers without the assumed background, a summary of this basic information is given in Appendix A.

1.2 SUPPORTING DOCUMENTATION

The following manuals provide additional information on the F8 microprocessor:

- 1) F8 Circuit Data Book which provides electrical parameter data for all Fairchild F8 Microprocessor devices.
- 2) F8 Timeshare Operating Systems Manual which explains how to assemble and debug F8 Microprocessor programs on NCSS and GE Timeshare Networks.
- 3) F8 Circuit Reference Manual which describes the interactive timing and signal sequences which occur between devices in the F8 Microprocessor family.
- 4) F8S and F8SEM Users Manuals which describe how to assemble and debug microprocessor programs on the F8S and F8SEM hardware modules.
- 5) F8 Formulator Users and Reference Manuals which describe how to use and maintain Fairchild's F8 Formulator developmental hardware.

THE F8 MICROPROCESSOR SYSTEM

The purpose of a microprocessor system is to replace discrete logic; but in order to understand why a microprocessor system is effective as a logic design tool, it is first necessary to understand what is in a microprocessor system.

2.1 WHAT IS A MICROPROCESSOR?

After a product has been fabricated using discrete logic components, it consists of one or more logic cards; each card may be visualized as generating a variety of signals output at the card edge, based on signals input at the card edge. The logic devices on the card are specifically selected and sequenced to generate the required product.

If the same product is implemented using the F8 microprocessor, the F8 CPU and its five supporting devices can be made to function in the same way as any one of many millions of different discrete logic device combinations. In other words, the F8 CPU, optionally in conjunction with the supporting devices, has the capacity to duplicate the performance of any discrete logic design, limited only by speed considerations. F8 microprocessor systems have a $2 \mu\text{s}$ instruction cycle time. The functions that will be performed by the F8 microprocessor system are established by a sequence of "instructions", stored in a memory device as a sequence of binary codes. Taken as a whole, the sequence of instructions are referred to as a "stored program".

2.2 SOME BASIC CONCEPTS

Any logic device may be reconstituted from some or all of the following basic functions:

- 1) Binary addition
- 2) The logical operations AND, OR and EXCLUSIVE-OR
- 3) Shifts and rotates of binary digit sequences which are being interpreted as numerical entities (e.g., a byte = eight bits).

A general purpose logic device can be created by implementing the basic functions listed above on a single chip. If the single chip is to duplicate the performance of other logic devices, it must be provided with a sequence of instructions that enable the required logic in the proper order, plus a stream of data that is operated on by the specified logic. This is illustrated in Figure 2-1.

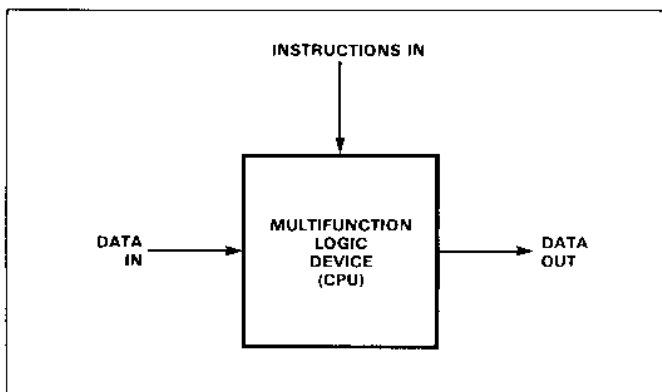


Fig. 2-1. Multifunction Logic Device

In order to function, the multifunction logic device will need the following parts:

- A) An Arithmetic Logic Unit (ALU), containing the necessary basic logic functions.
- B) A control unit, which decodes instructions and enables elements of the ALU, as needed.
- C) Registers to hold instruction codes and data, as needed.
- D) Data paths within the CPU, and between the CPU and external devices.

Parts A), B), C), and D) are the basic components of any Central Processing Unit (CPU). A CPU must be the focal point of any computer—maxi, mini or micro.

Referring to Figure 2-1, where do "instructions" and "data in" come from, and where will "data out" go? There are two possibilities: memory or external devices.

Refer to Figure 2-2. Memory is a passive depository of information where data or instruction codes may be stored. Memory must be divided into individually addressable locations, each of which can store one element of instruction code or one element of data. In an F8 system, each individually addressable location will be an 8-bit data unit (a byte), since the F8 is an 8-bit microprocessor.

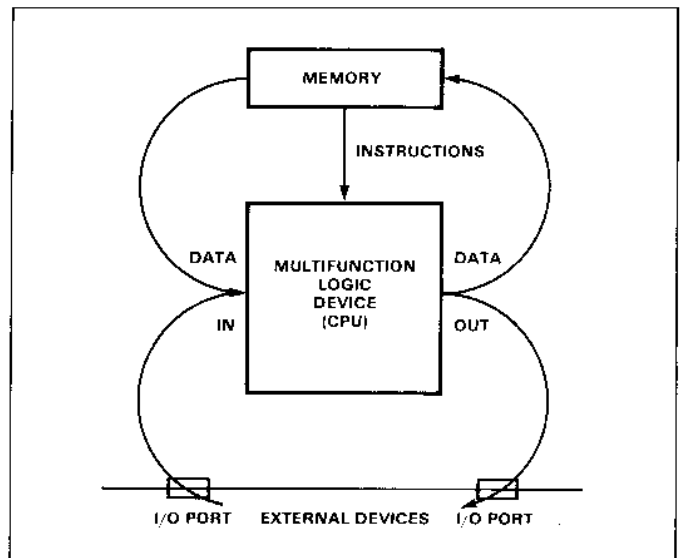


Fig. 2-2. Data and Instruction Paths in a Multifunction Logic Device

"External devices" refer to any data source or destination beyond the perimeter of the microprocessor system. Drawing an analogy with a logic card, "external devices" will refer to the world beyond the card edge connector. Data passes between the microprocessor system and external devices via Input/Output (I/O) ports.

2.2.1 Instructions, Programs, Data and Memory

For a microprocessor to perform any specified operation, it will receive and process a sequence of instructions. The sequence may be very long—numbering even into the thousands

of instructions. A sequence of instructions that can be taken as a unit is called a program; the purpose of this manual is to describe how a program is constructed out of a sequence of instructions.

Data may (and usually will) be stored in memory. In fact, the 256 possible combinations of eight binary digits (or byte) may represent any of the following types of information:

- 1) An instruction code
- 2) Numeric or address data that is part of an instruction's code
- 3) Numeric or address data that is independent of instruction codes
- 4) A coded representation of a letter of the alphabet, digit or printable character

It would be impossible to determine the content of any memory byte by random inspection. This does not cause problems, since a program will occupy one or more segments of contiguous memory bytes, and data resides in blocks of memory as assigned by the programmer.

2.2.2 Interrupts

The number of programs which may be stored in memory is limited only by the amount of memory available for program storage. If ten programs were stored in memory, by simply identifying one program, the same microprocessor system could be made to function in one of ten different ways.

If a microprocessor has more than one program available for execution, how is the one program which is to be executed identified? There are two separate and distinct ways in which a program may be identified for execution:

- A) Program identification may itself be a programmed function; for example, each program, upon completing execution, may identify the next program to be executed. The key to this method of program identification is that it is internally controlled, within the logic of the microprocessor system.
- B) Programs may be called into execution by external devices; this may happen even if another program is in the middle of execution. For example, take the simple case of a microprocessor that is recording data input by an external instrument; while receiving data from the external instrument, the microprocessor performs numerical operations on the collected data. Program executions are illustrated in Figure 2-3.

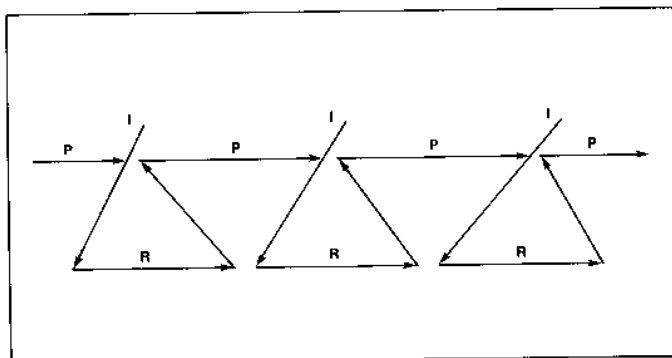


Fig. 2-3. Program P Being Interrupted to Execute Program R

In Figure 2-3, P represents the program performing numerical operations on the data. Data is collected by repeated execution of program R. Events occur as follows:

- 1) Program P is executing.
- 2) When the external instrument has data which it is ready to transmit, it sends an interrupt signal (I) to the microprocessor, along with the starting address of program R.
- 3) Upon receiving interrupt signal I, the microprocessor does some elementary "housekeeping"; for example, it saves the address of the program P instruction it was about to execute, plus any intermediate data being held in temporary storage registers.
- 4) The microprocessor completely executes program R.
- 5) Upon completion of program R execution, the microprocessor restores values saved in step 3, then continues program P execution from the point where interrupt I occurred. Thus execution of program P appears to have gone into "suspended animation" for the duration of program R execution.

The sequence of events illustrated in Figure 2-3 is quite common in microprocessor applications, and is called an external interrupt. Interrupt programming is described in Section 8.2.

2.2.3 Programmable Clocks

There are many microprocessor applications in which it is important that the microprocessor system be synchronized with the real time of the outside world. Such synchronization is accomplished using programmable clocks, which are registers that count at a known rate. When the shift register counts to zero, the event is marked by an interrupt (as described in Section 2.2.2); in this case the interrupt is defined as a "time out" interrupt. Since the rate at which the clock register counts will be known for any microprocessor system, setting a real time interval simply involves loading the register with the correct initial count.

2.2.4 Direct Memory Access

Notice from Figure 2-2 that data may be input to the microprocessor from memory or from an external device, via an I/O port.

It is easy to imagine how, in many applications, data will be transferred from an external device, via an I/O port and the CPU, to memory; the data will then be accessed from memory in the normal course of program execution.

It makes little sense to tie up the logic of the CPU while shunting data from an I/O port to memory; therefore, provisions are made for Direct Memory Access (DMA), whereby data is moved between memory and an "I/O port", bypassing the CPU entirely. The DMA "I/O port" is called a "DMA channel".

In order to implement DMA, the microprocessor system must have logic (outside the CPU) which provides the following three pieces of information:

- 1) A starting memory address for a data block.
- 2) A byte length for the data block.
- 3) The direction of the data movement.

If the microprocessor has this logic, data may be transferred between memory and an I/O port independent of, and in parallel with, unrelated CPU-memory operations.

2.2.5 A Complete Microprocessor System

To summarize, a complete microprocessor system will have the following logical components:

- 1) A CPU, which is the multifunction logic device of the system.
- 2) Memory (of various types and combinations), in which programs and data are stored.
- 3) Memory interface logic which identifies:
 - a) the next memory location which must be accessed to fetch instruction codes for the CPU, and
 - b) the memory location from which a byte of data will be read, or to which a byte of data will be written.

- 4) I/O ports, through which bidirectional data passes between the microprocessor system and external devices.
- 5) DMA logic, which provides a direct data path between memory and external devices, bypassing the CPU.
- 6) Interrupt logic, which causes the CPU to temporarily suspend current program execution. Along with each interrupt request signal, interrupt logic identifies the program which is to implement operations required by the source of the interrupt.
- 7) Real time clock logic, which synchronizes the entire microprocessor system with the real outside world by generating interrupts at variably definable time intervals.

Figure 2-4 illustrates these seven logical components, with associated data flow paths.

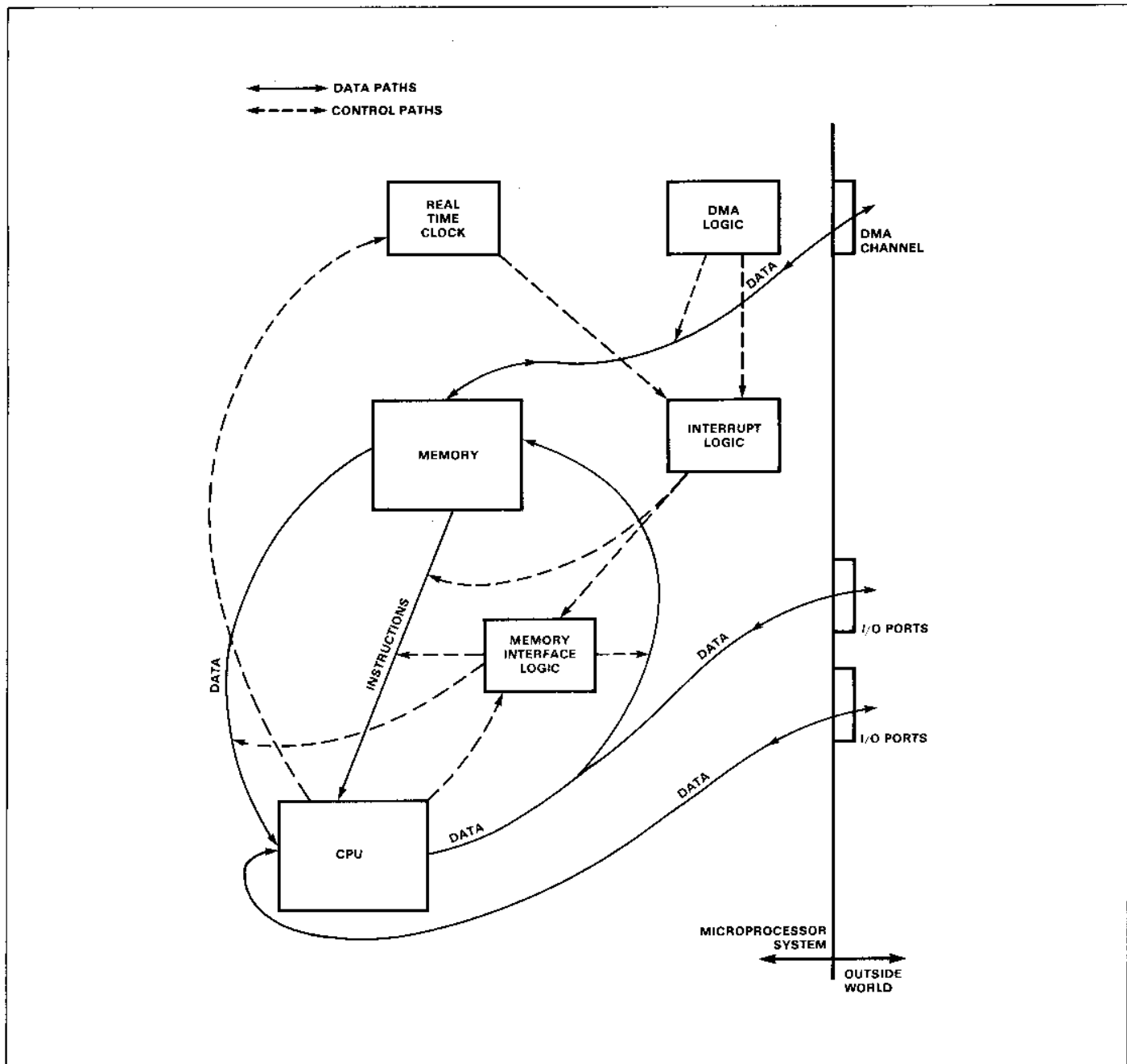


Fig. 2-4. Logical Components, Data Paths and Control Paths in any Microprocessor System

2.3 THE F8 SYSTEM

There is no one-for-one correspondence between the logical components of a microprocessor system, as illustrated in Figure 2-4, and the devices of the F8, or any other microprocessor product. In fact, it is counter-productive to extend the concept of isolating functions on separate devices because it reduces the flexibility of a microprocessor system to satisfy simple, as well as complex, applications needs. More than any other microprocessor product, the F8 combines many functions on single chips, thus allowing simple systems to be implemented with as few as two devices, and complex systems to be implemented using many devices.

Figure 2-5 illustrates the way in which F8 microprocessor system devices interconnect to give a variety of system configurations.

The simplest F8 system contains one 3850 CPU and one 3851 PSU.

Another very simple F8 system consists of one 3850 CPU, plus either one 3852 DMI interfaced to a single dynamic memory, or one 3853 SMI interfaced to a single static memory device.

A fully expanded F8 system may have one 3850 CPU, one 3852 DMI and one 3853 SMI device, up to four 3854 DMA devices, plus 3851 PSU and static or dynamic memory devices in any combination, providing not more than a combined total of 65,536 bytes of memory are directly addressed by the 3850 CPU. It is possible to address more than 65,536 bytes of memory using special techniques which are described in the F8 Circuit Reference Manual.

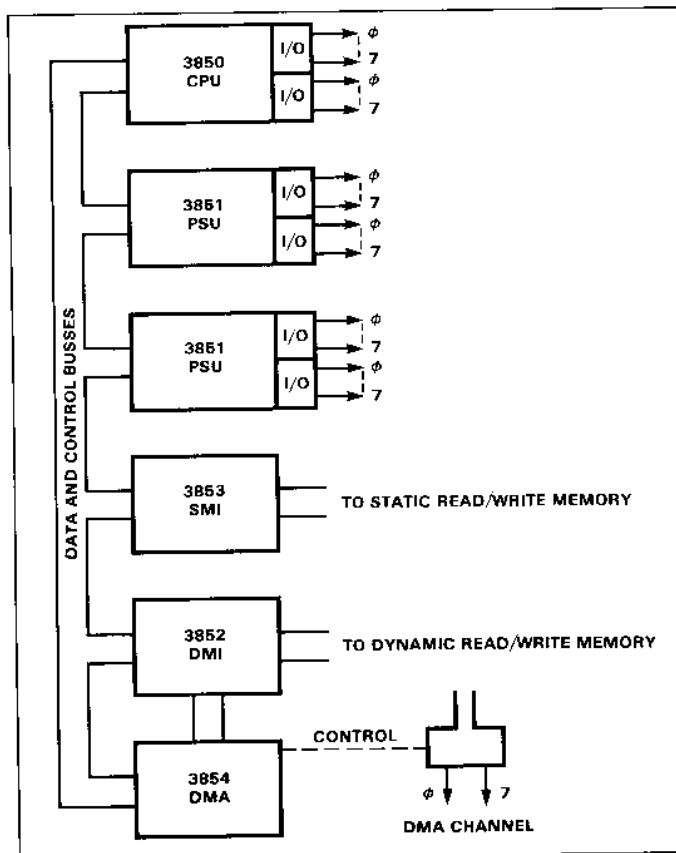
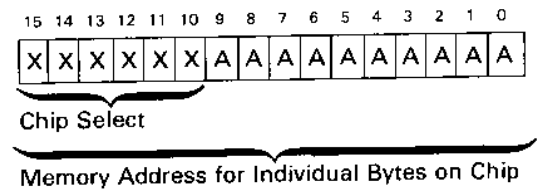


Fig. 2-5. F8 Microprocessor System Configurations

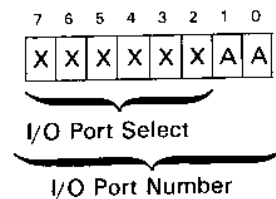
2.3.1 Chip and I/O Port Selection

Every 3851 PSU has two permanent select codes—a chip select code and an I/O port select code.

The 3851 PSU chip select code is a six digit binary number, which is always the highest six bits for memory addresses on that device:



The 3851 PSU I/O port select code is also a six digit binary number, and is independent of the chip select code. The I/O port select code is always the highest six bits for I/O port numbers on that device:



The 3852 DMI and 3853 SMI devices have a fixed (pre-assigned) I/O port select code, but have no on-board chip select code.

The dynamic and/or static memories associated with the 3852 DMI and 3853 SMI derive their select function from external logic. This allows the system designer complete freedom with respect to memory space partitioning.

Every F8 microprocessor system must have one memory device whose byte addresses start at 0; the first instruction executed when an F8 system is powered up is the instruction stored in memory byte 0.

2.4 THE 3850 CPU

Figure 2-6 illustrates the logical functions implemented on the 3850 CPU.

The heart of the F8 microprocessor system is the 3850 CPU, which contains data manipulation logic in an Arithmetic Logic Unit (ALU). Eight-bit instruction codes are decoded by a Control Unit (CU), which controls execution of logic internal to the 3850 CPU and generates signals controlling operations of other devices in the system.

2.4.1 Timing

System timing is illustrated in Figure 2-7. System timing is controlled by an external or internal clock, which provides clock pulses of not less than 500 ns and not more than 10 μ s. In response to instruction codes, the CPU creates instruction timing cycles of either 4 or 6 clock pulses. The fastest instruction will execute in one short (4 clock pulse) cycle; the slowest instruction will execute in one short (4 clock pulse) cycle plus three long (6 clock pulse) cycles.

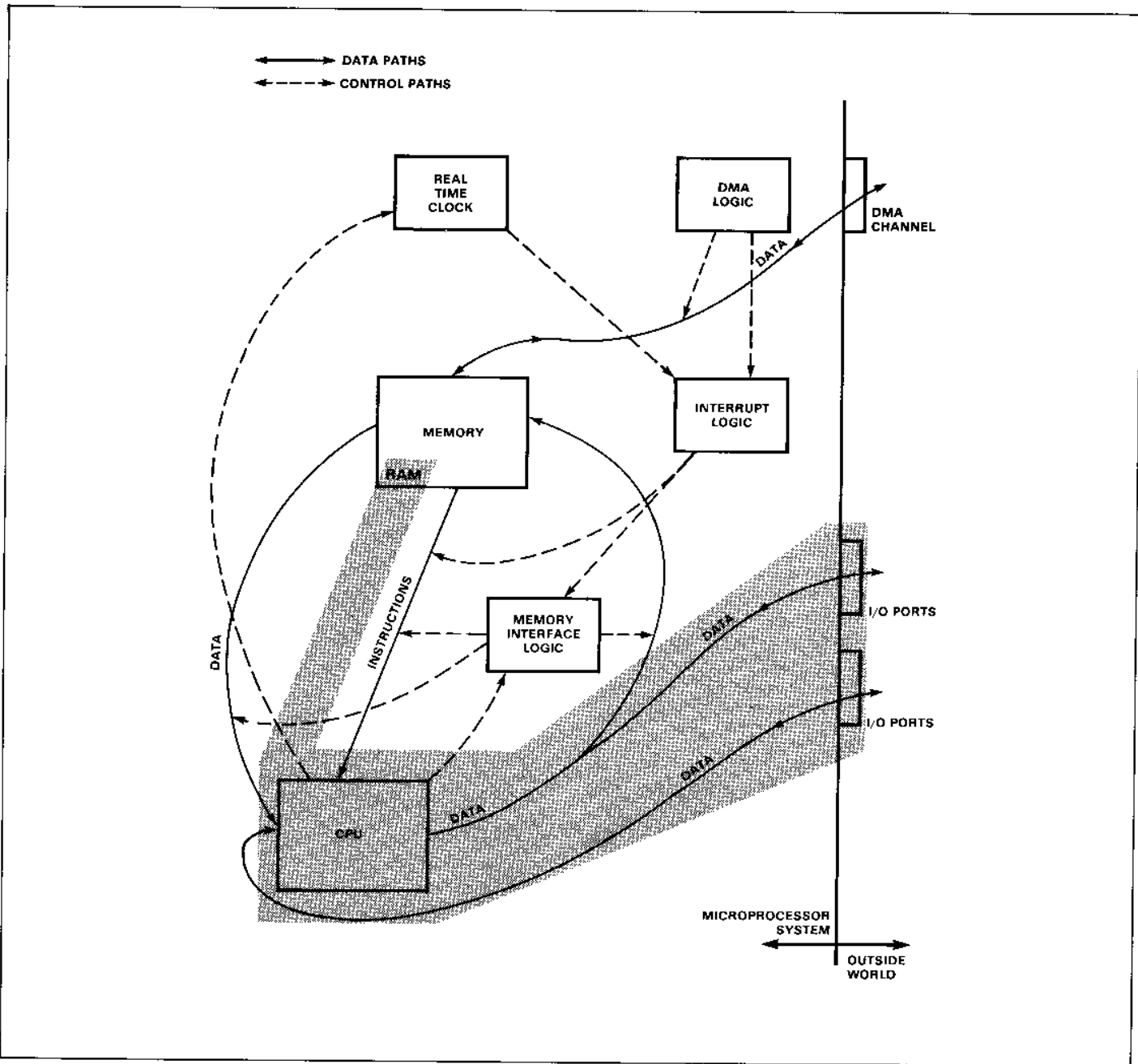


Fig. 2-6. Logical Functions of the 3850 CPU

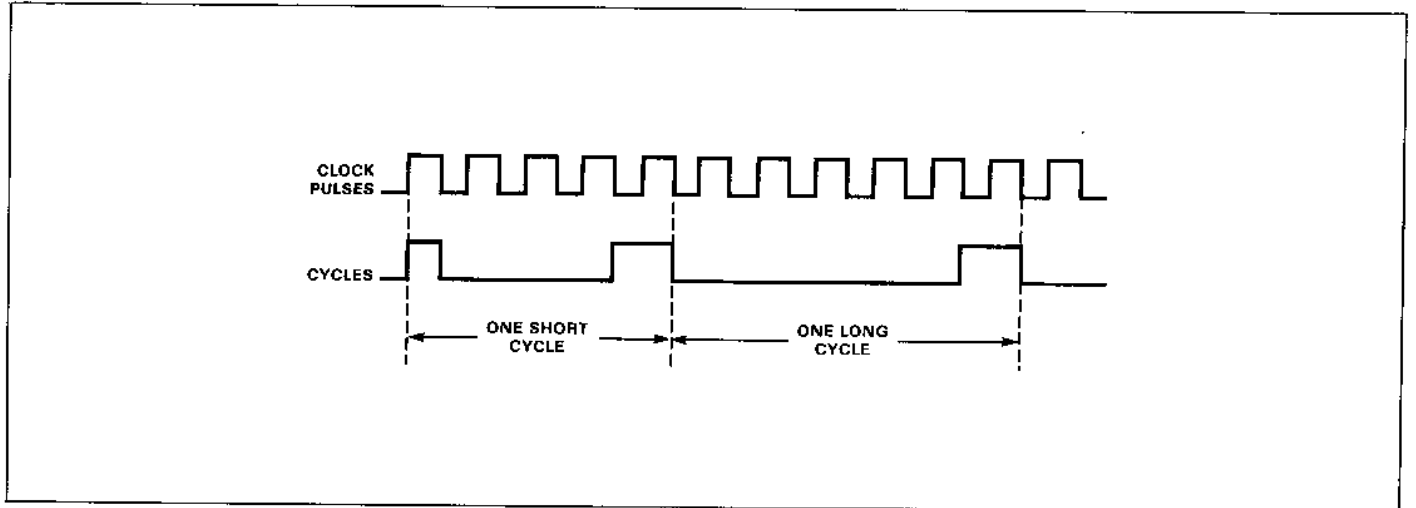


Fig. 2-7. Instruction Timing

2.4.2 CPU Registers

The 3850 CPU has an 8-bit Accumulator Register and a Scratchpad consisting of 64 8-bit registers. In addition there is a 6-bit Indirect Scratchpad Address Register (ISAR), which is used to address the scratchpad and a 5-bit Status Register (the W register), which identifies selected status conditions associated with the results of CPU operations. Figure 2-8 illustrates the CPU register.

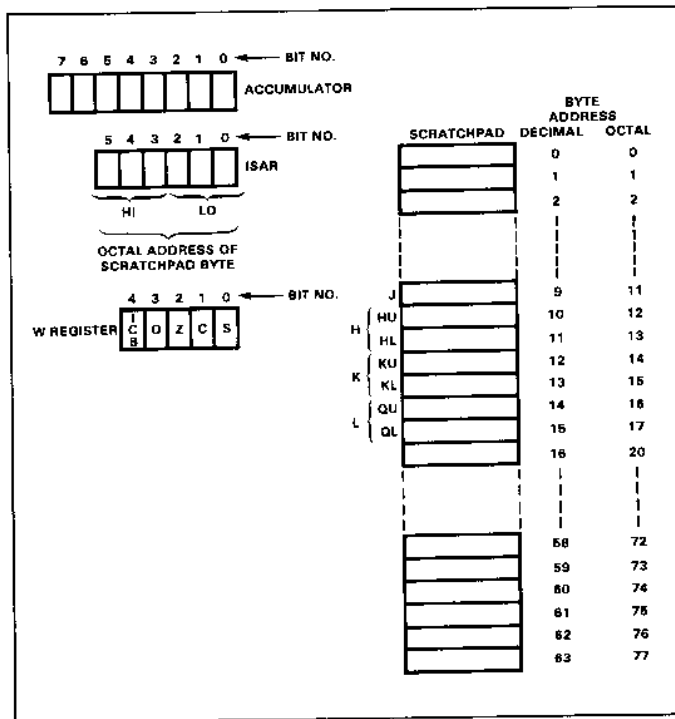


Fig. 2-8. 3850 CPU Programmable Registers

Data in the Accumulator may be manipulated by the ALU. Individual instructions allow the contents of the Accumulator to be operated on in a variety of ways. Data may be transferred between the Accumulator and other CPU registers, or between the Accumulator and data locations outside the CPU.

The Scratchpad is the principal depository of frequently accessed data and, in small microprocessor configurations, may represent the system's only Read/Write Memory. Because the Scratchpad actually resides on the CPU, instructions that reference Scratchpad bytes execute in one short cycle; these are the fastest executing F8 instructions.

The first 16 Scratchpad bytes can be identified by instructions without using the ISAR. The remaining Scratchpad bytes are referenced via the ISAR; i.e., the ISAR is assumed to hold the address of the Scratchpad byte which is to be referenced. Observe that the first 16 bytes of the Scratchpad can also be referenced via the ISAR.

The ISAR should be visualized as holding two octal digits, HI and LO. This division of the ISAR is important, since a number of instructions increment or decrement the contents of the ISAR when referencing Scratchpad bytes via the ISAR. This allows a sequence of contiguous scratchpad bytes to be easily referenced. However, only the low order octal digit (LO) is incremented or decremented; thus ISAR is incremented from 0'27' to 0'20', not to 0'30'. Similarly, ISAR is decremented

from 0'20' to 0'27', not to 0'17'. This feature of the ISAR greatly simplifies many program sequences, as will be described in Section 7.

Seven of the Scratchpad registers (9 through 15) have special significance. Data from register 9 may be moved directly between register 9 and the W register, bypassing the Accumulator. Registers 10 through 15 are connected to memory interface logic, as described in Sections 2.5, 2.6 and 2.7.

2.4.3 Status

A number of operations performed by the Arithmetic Logic Unit (ALU) generate results, selected characteristics of which are important to logic sequences. Table 2-1 summarizes the W register status bits, which are individually described next.

| |
|--|
| $\text{OVERFLOW} = \text{CARRY}_7 \oplus \text{CARRY}_6$ $\text{ZERO} = \overline{\text{ALU}_7} \overline{\text{ALU}_6} \overline{\text{ALU}_5} \overline{\text{ALU}_4} \overline{\text{ALU}_3} \overline{\text{ALU}_2} \overline{\text{ALU}_1} \overline{\text{ALU}_0}$ $\text{CARRY} = \text{CARRY}_7$ $\text{SIGN} = \overline{\text{ALU}_7}$ |
|--|

Table 2-1. A Summary of Status Bits

SIGN

When the results of an ALU operation are being interpreted as a signed binary number, the high order bit (bit 7) represents the sign of the number (see Appendix A). At the conclusion of instructions that may modify the Accumulator bit 7, the S bit (W register bit 0) is set to the complement of the Accumulator bit 7.

CARRY

The C bit (W register bit 1) may be visualized as an extension of an 8-bit data unit; i.e., bit 8 of a 9-bit data unit. When two bytes are added and the sum is greater than 255, the carry out of bit 7 appears in the C bit. Here are some examples:

| | | | | | | | | | | |
|-----------------------|---|---|---|---|---|---|---|---|---|--------------|
| | C | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | ← Bit Number |
| Accumulator contents: | | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | |
| Value added: | | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | |
| Sum: | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | |

There is no carry, so C is reset to 0.

| | | | | | | | | | | |
|-----------------------|---|---|---|---|---|---|---|---|---|--------------|
| | C | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | ← Bit Number |
| Accumulator contents: | | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | |
| Value added: | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | |
| Sum: | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | |

There is a carry, so C is set to 1.

ZERO

The Z bit (W Register bit 2) is set whenever an arithmetic or logical operation generates a zero result. The Z bit is reset to 0 when an arithmetic or logical operation could have generated a zero result, but did not.

Load instructions do not affect status bits.

- a) The Accumulator contains 01101011. The value 00010101 is added to the Accumulator:

| | |
|-----------------------|-----------------|
| Accumulator contents: | 0 1 1 0 1 0 1 1 |
| Value added: | 0 0 0 1 0 1 0 1 |
| Sum: | 1 0 0 0 0 0 0 0 |

The result in the Accumulator is not zero, so the Z bit is reset to 0. (There is no carry, so C is reset to 0).

- b) Next, the Accumulator contents are shifted left one bit position:

| | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|----------------|
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Bit number |
| | | | | | | | | | (before shift) |
| shifted out ← | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ← 0 shifted in |
| after shift | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Since the result in the Accumulator is now zero, the Z bit is set to 1.

- c) Subsequently the value 1101111 is loaded into the Accumulator. Even though the Accumulator no longer contains zero, the Z bit remains set at 1 since an Accumulator load is neither an arithmetic nor a logical operation, therefore has no effect on the Z bit.

OVERFLOW

The high order Accumulator bit (bit 7) represents the sign of the number. When the Accumulator contents are being interpreted as a signed binary number, some method must be provided for indicating carries out of the highest numeric bit (bit 6 of the Accumulator). This is done using the O bit (W register bit 3). After arithmetic operations, the O bit is set to the EXCLUSIVE-OR of Carry Out of bits 6 and bits 7. This simplifies signed binary arithmetic as shown in Section 10.3 and in Appendix A. Here are some examples:

| | | | | | | | | | | |
|-----------------------|---|---|---|---|---|---|---|---|---|------------|
| | C | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Bit Number |
| Accumulator contents: | | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | |
| Value added: | | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | |
| Sum: | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | |

There is a carry out of bit 6 and out of bit 7, so the O bit is reset to 0 ($1 \oplus 1 = 0$). The C bit is set to 1.

| | | | | | | | | | | |
|-----------------------|---|---|---|---|---|---|---|---|---|------------|
| | C | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Bit Number |
| Accumulator contents: | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | |
| Value added: | | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | |
| Sum: | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | |

There is a carry out of bit 6, but no carry out of bit 7; the O bit is set to 1 ($1 \oplus 0 = 1$). The C bit is reset to 0.

When the Overflow bit is set, the magnitude of the number is too large for the 7-bit numeric field within the byte, and the sign bit has been destroyed. However, the 9-bit field made up of the Carry bit (high order) and the data byte give a valid 9-bit signed binary result.

ICB AND INTERRUPTS

External logic can alter the operations sequence within the CPU by interrupting ongoing operations, as described in Section 2.2.2. However, interrupts are allowed only when the ICB bit (W register bit 4) is set to 1; interrupts are disallowed when the ICB bit is reset to 0.

2.4.4 3850 Input/Output

The 3850 CPU communicates with the outside world in two ways:

To execute instructions, instruction codes must be input from the external storage device (probably a 3851 PSU) where they are being maintained. Data stored in a memory device may have to be loaded into the CPU in order to meet the requirements of the instruction being executed. This type of communication between the 3850 CPU and the outside world is of no immediate concern to an F8 programmer, since it involves data flows within the confines of the microprocessor system, and requires no special considerations beyond an understanding of instruction execution sequences.

Input/output programming, as the term is commonly used, refers to data transfers between the microprocessor system and logic beyond the microprocessor system. The 3850 CPU has two 8-bit, bidirectional ports, via which 8-bit parallel data may be transferred in either direction, between the 3850 CPU and logic external to the microprocessor system. The two 3850 CPU I/O ports are identified by the hexadecimal port addresses H'00' and H'01'.

2.5 THE 3851 PSU

Figure 2-9 illustrates the logical functions implemented on the 3851 PSU.

The 3851 PSU provides an F8 microprocessor system with 1024 bytes of Read Only Memory. 3851 memory is usually used to store instructions, but may also be used to store data that is read, but never altered. In addition, each 3851 PSU provides two 8-bit I/O ports, a programmable timer and external interrupt processing logic.

The 3851 PSU is the logic device which is modified and replaced to reflect a product's continuing engineering and field upgrades.

In microprocessor systems, instruction codes are usually stored in a PSU to prevent accidental erasure. As many as 64 3851 PSU's may be connected to one 3850 CPU, yet a single 3851 PSU interfaced to a 3850 CPU, provides a viable microprocessor system with the following capacities:

- 1024 bytes of program storage (on the 3851)
- 64 bytes of Read/Write Memory (on the 3850)
- 4 separately addressable, bidirectional I/O ports (2 on the 3850, 2 on the 3851)
- An external interrupt line
- A programmable clock

2.5.1 3851 Timing

Timing signals created by the 3850 CPU, and illustrated in Figure 2-7, control operation sequences in the 3851 PSU.

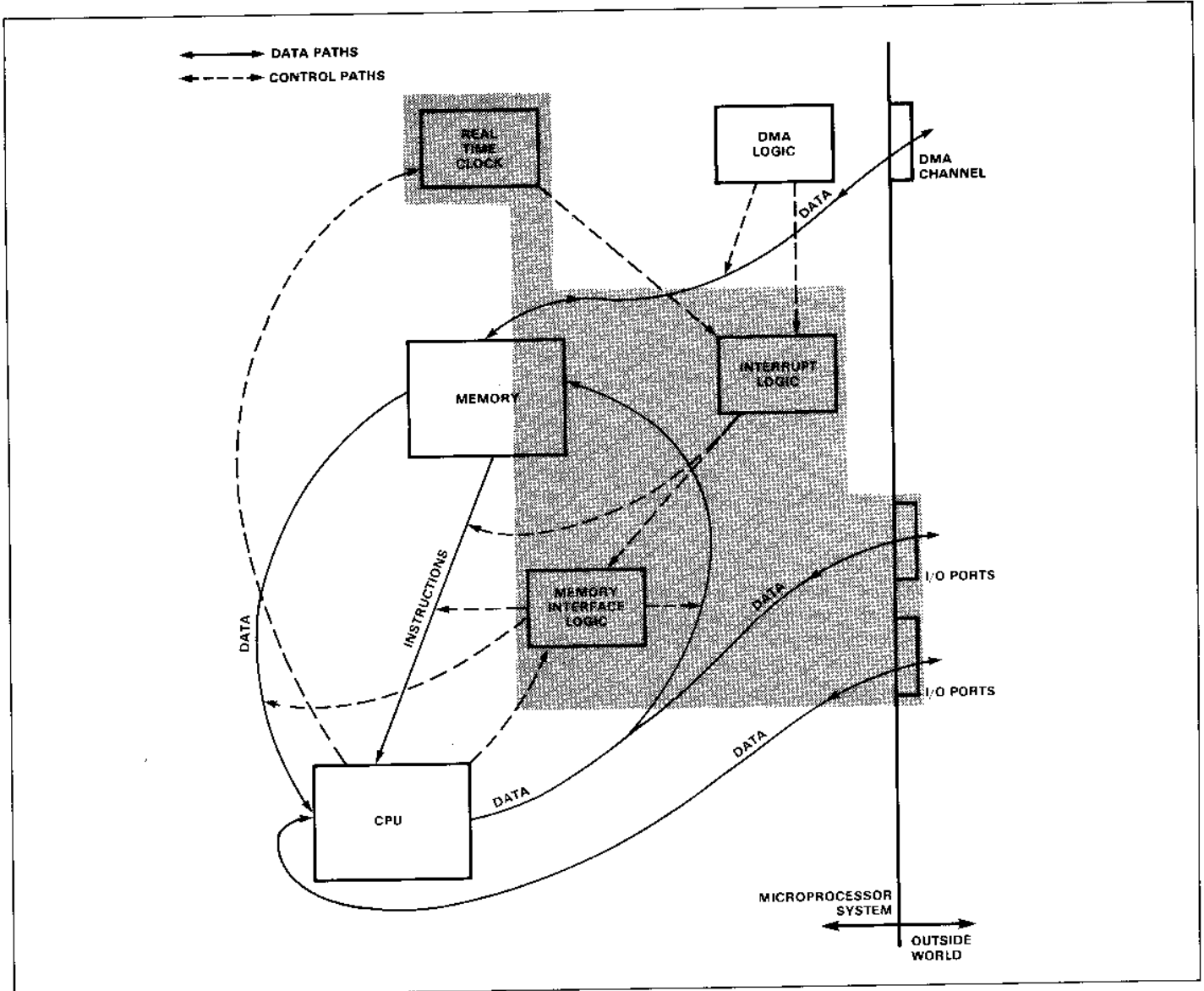


Fig. 2-9. Logical Functions of the 3851 PSU

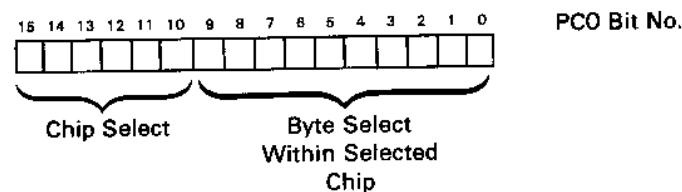
2.5.2 3851 Registers

In addition to 1024 bytes of ROM, the 3851 contains three 16-bit address registers, which are described next.

PROGRAM COUNTER (PC0)

This 16-bit register provides the address of the memory byte from which the next instruction code will be fetched for transmittal to the 3850 CPU. After each byte of instruction code is fetched, logic internal to the 3851 increments the contents of PC0 to address the next memory byte.

Even though each 3851 PSU contains only 1024 bytes of memory, PC0 preserves a 16-bit memory address. Thus PC0 may be interpreted as follows:



Each 3851 device has a unique select code that is a permanent mask option; 3851 memory access logic is only activated when the six Chip Select bits of PC0 match the 3851 select code. Thus, if more than one 3851 is present in an F8 system, every 3851 device's PC0 register holds the address of the memory byte from which the next instruction code will be fetched for transmittal to the 3850 CPU; but an instruction fetch will actually be executed from one 3851 device only.

The PC0 registers of the 3851 devices are logically connected to 3850 scratchpad bytes 12 and 13, designated as the K register, and bytes 14 and 15, designated as the Q register in Figure 2-8. Specific instructions allow the contents of the K or Q register to be loaded into every PC0 register. Specific instructions allow the PC0 registers' contents to be modified in order to control microprocessor logic sequences.

Note that in a correctly designed F8 microprocessor system, when there is more than one 3851 device, every PC0 register will always contain exactly the same address.

STACK REGISTER (PC1)

Every 3851 device has a 16-bit Stack Register, which is a buffer for the contents of PC0. This allows program execution sequence to be modified by changing the PC0 registers' contents, while the previous contents of PC0 are saved in PC1; thus programs may return to the prior instruction execution sequence.

The PC1 registers are logically connected to the 3850 scratchpad bytes 12 and 13, designated as the K register in Figure 2-8. Specific instructions allow the contents of the K register to be loaded into every PC1 register, or the PC1 registers' contents to be loaded into the K register.

DATA COUNTER (DC)

Every 3851 device has a 16-bit Data Counter register which contains the address of the memory byte (external to the 3850 CPU) from which data is to be accessed. For example, an instruction requiring a data byte to be loaded from external memory into the 3850 Accumulator will fetch the contents of the data byte addressed by the DC registers.

The DC registers are 16-bit registers, where the high order six bits (bits 15 to 10) are interpreted as chip select bits, and the low order nine bits (bits 9 to 0) provide the byte address.

The DC0 registers are logically linked to the H and Q registers in the same way that the PC1 registers are logically linked to scratchpad register K.

2.5.3 3851 Input/Output

Each 3851 PSU has two bidirectional, 8-bit I/O ports. Each port's address, using binary notation, is XXXXXX00 or XXXXXX01, where the X binary digits are the device's unique I/O port select code. Note that every 3851 PSU has an I/O port select code and an independent chip select code.

2.5.4 3851 Local Timer and Interrupt

3851 programmable timer and interrupt logic are accessed via the binary port addresses XXXXXX11 and XXXXXX10, respectively; the X binary digits are the I/O port select codes described in Section 2.5.3.

The programmable timer port is a polynomial shift register which runs continuously, sending a signal to the interrupt control logic whenever the timer count equals zero.

Any numeric value between 0 and 255 may be loaded into the programmable timer port by an appropriate instruction code. If 255 (hexadecimal FF) is loaded into a timer port, the timer is stopped. Any other value loaded into a timer port is decremented once every 31 clock pulses (see Figure 2-7); therefore delays up to 7905 clock pulses may be programmed.

The local interrupt port is loaded by an appropriate instruction, with a control code; bits 0 and 1 of the control code are interpreted as follows:

| Bit 1 | Bit 0 | Function |
|-------|-------|----------------------------|
| 0 | 0 | Disallow all interrupts |
| 0 | 1 | Enable external interrupts |
| 1 | 0 | Disallow all interrupts |
| 1 | 1 | Enable timer interrupts; |

If timer interrupts have been enabled and if the 3850 CPU has enabled interrupts (via the ICB status), then when the local timer decrements to 0, an interrupt request is transmitted to the 3850 CPU.

The way in which the local timer and interrupt ports are used is described in Section 8.3.

2.6 THE 3852 DYNAMIC MEMORY INTERFACE

Figure 2-10 illustrates the logical functions implemented on the 3852 DMI device.

The 3852 DMI device interfaces dynamic random access memory (e.g., Fairchild 3540 RAM) to a 3850 CPU. One 3852 DMI device interfaces up to 65,536 bytes of RAM memory to the 3850 CPU. However, recall that a combined maximum of 65,536 bytes of ROM and RAM may be addressed by the 3850 CPU unless special additional memory interfacing logic is added to the microprocessor system.

Only one 3852 DMI device will normally be present in an F8 microprocessor system.

The 3854 DMA device may be attached to the 3852 DMI device enabling data to be transferred between memory devices and any external device, bypassing the 3850 CPU.

2.6.1 3852 Timing

Timing signals created by the 3850 CPU, and illustrated in Figure 2-7, control operation sequences in the 3852 DMI.

2.6.2 3852 Registers

The 3852 DMI device has the same address registers as the 3851 PSU; however, the 3852 DMI has two Data Counter registers. Thus the 3852 has one Program Counter (PC0), one Stack Pointer (PC1) and two Data Counters (DC0 and DC1).

There are two differences between the way in which 3852 registers and 3851 registers are used.

The 3852 has no chip select mask. This is because there will only be one 3852 device in a microprocessor system, and it passes the entire PC0 address to attached RAM devices; the attached RAM devices interpret part of the PC0 address as chip select lines.

Data Counter DC1 is a temporary storage buffer for Data Counter DC0. An instruction switches the DC0 and DC1 registers' contents; since 3851 PSU have no DC1 register, this switch instruction has no effect on 3851 PSU. Thus it is possible for the 3852 DMI Data Counter (DC0) to have contents which differ from 3851 PSU. Recall that the Data Counters are logically connected to the H and Q scratchpad registers within the 3850 CPU, so that Data Counters' contents may be transferred to the H or Q registers. The fact that the 3851 DC0 register and the 3852 DC0 register may not hold the same addresses may present a problem, since the contents of a Data Counter is transferred to the H or Q registers from any device with a device select code corresponding to the current DC0 contents.

Simultaneous use of 3851 PSU and 3852 DMI devices is discussed in detail in Section 7.2.

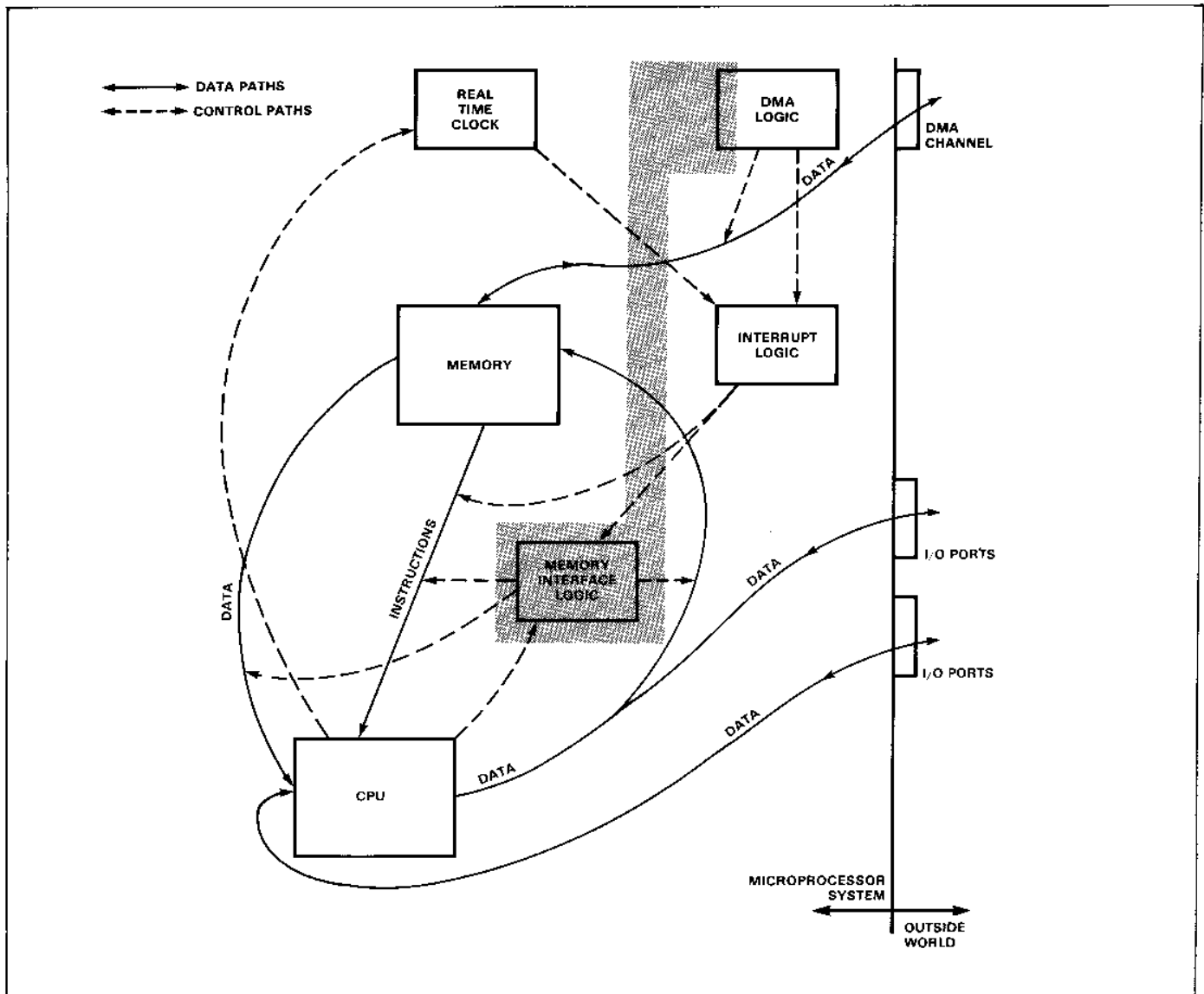


Fig. 2-10. Logical Functions of the 3852 DMI Device

2.6.3 3852 Direct Memory Access and Memory Refresh

The 3852 DMI device has two addressable ports which are used to enable direct transfer of data between memory devices and external devices. This transfer is referred to as Direct Memory Access (DMA), and requires the presence of the 3854 DMA device. For a discussion of DMA see Sections 2.2.4, 2.8 and 8.4.

The two addressable 3852 ports use hexadecimal addresses H'0C' and H'0D'. Port H'0C' requires a control byte to be loaded for interpretation as follows:

| Bit No. | | |
|---------|--------------------------------------|--------------------------------------|
| 0 | 1 = DMA not allowed | 0 = DMA allowed |
| 1 | 1 = Refresh memory | 0 = No memory refresh |
| 2 | 1 = Refresh every fourth write cycle | 0 = Refresh every eighth write cycle |

Another version of the 3852 DMI device, referred to as the SL 31116 device, uses port addresses H'EC' and H'ED' instead of H'0C' and H'0D'. This allows 3852 DMI and 3853 SMI devices to be used in the same microprocessor system.

2.7 THE 3853 STATIC MEMORY INTERFACE

Figure 2-11 illustrates the logical functions implemented on the 3853 SMI device.

The 3853 SMI device is similar to the 3852 DMI device, described in Section 2.5. There are four important differences, which are described below.

- 1) The 3853 SMI device interfaces static memory (such as the Fairchild 2102 RAM) to a 3850 CPU.
- 2) The 3853 SMI does not have a DMA interface capability.
- 3) The 3853 SMI has local timer and interrupt control, as described for the 3851 PSU in Section 2.4.4. However, the 3853 local timer port address is H'0F' and the interrupt control port address is H'0E'.
- 4) The 3853 SMI has two additional ports, addressed H'0C' and H'0D', which are programmable interrupt vector registers. The importance and use of these registers is discussed in Section 8.2.

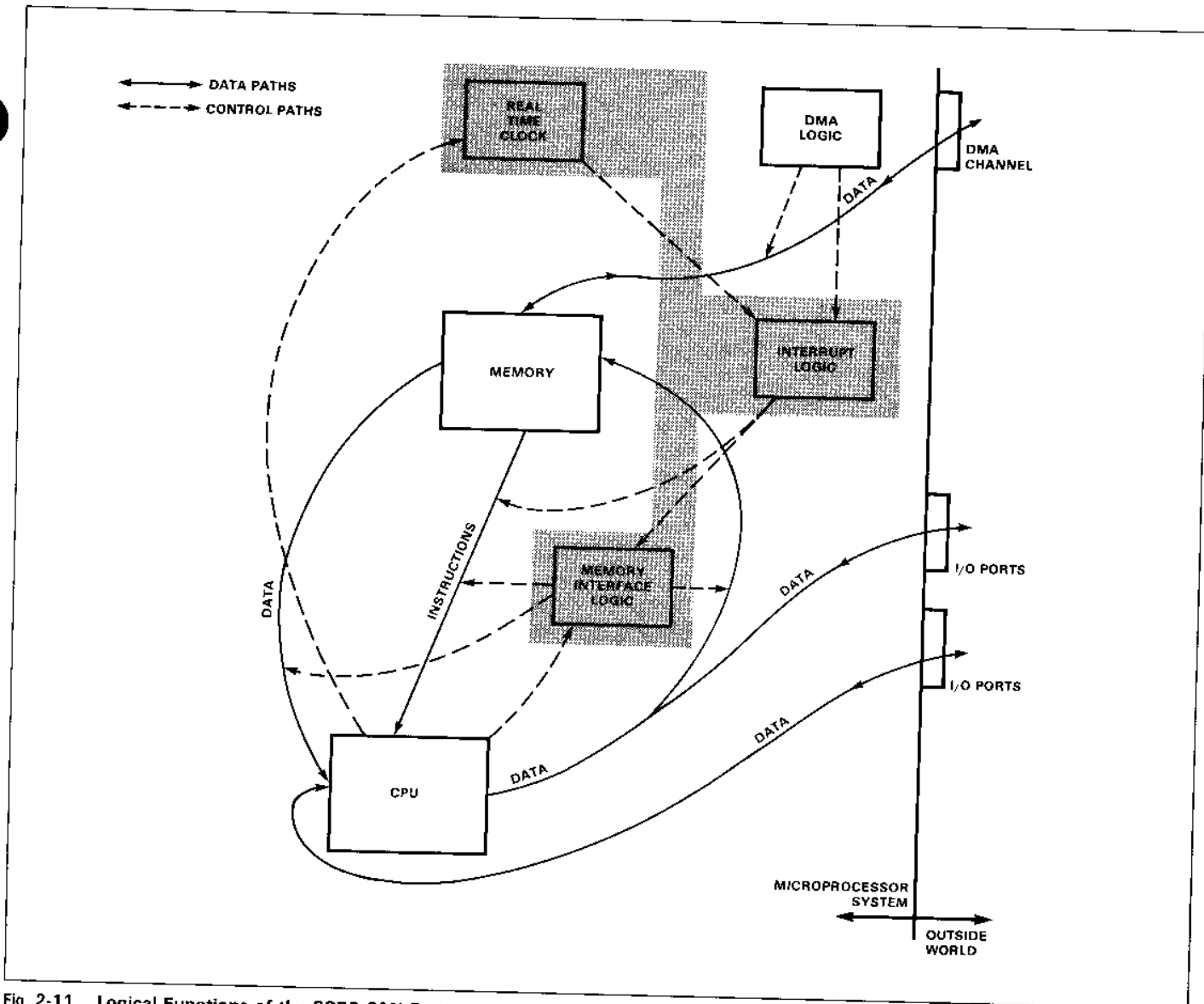


Fig. 2-11. Logical Functions of the 3853 SMI Device

Since the 3853, like the 3852, has two Data Counter registers, there are similar programming consequences, as described in Section 7.2.

2.8 THE 3854 DIRECT MEMORY ACCESS

Figure 2-12 illustrates the logical functions implemented on the 3854 DMA device.

The 3854 DMA device, in conjunction with the 3852 DMI device, sets up a data channel between a peripheral device and the memory associated with the DMI. DMA data transfers occur during the second part of each instruction cycle, therefore program execution speed is in no way degraded by parallel DMA data transfers. The concept of DMA data transfers is described in Section 2.2.4.

There may be up to four 3854 DMA devices in one microprocessor system.

Any external device may be attached to a 3854 DMA device. Also, two microprocessor systems may communicate with each other via a DMA device. For a description of how various DMA operations are programmed, see Section 8.4.

2.8.1 3854 Registers

The 3854 has three internal registers, addressed as four separate I/O ports. Addresses of the four I/O ports associated with the three 3854 registers are given in Table 2-2. The three registers are described next.

| FUNCTION OF I/O PORT | FIRST 3854 | SECOND 3854 | THIRD 3854 | FOURTH 3854 |
|--|------------|-------------|------------|-------------|
| Address, L.O. Byte (BUFA) | F0 | F4 | F8 | FC |
| Address, H.O. Byte (BUFB) | F1 | F5 | F9 | FD |
| Count, L.O. Byte (BUFC) | F2 | F6 | FA | FE |
| Count, H.O. Four bits, and Control* (BUFD) | F3 | F7 | FB | FF |

*The low order four bits of this port constitute the high order four bits of the byte count. The high order four bits of this port constitute the function code.

Table 2-2. Hexadecimal Addresses of Four I/O Ports Used as Registers by Four 3854 DMA Registers.

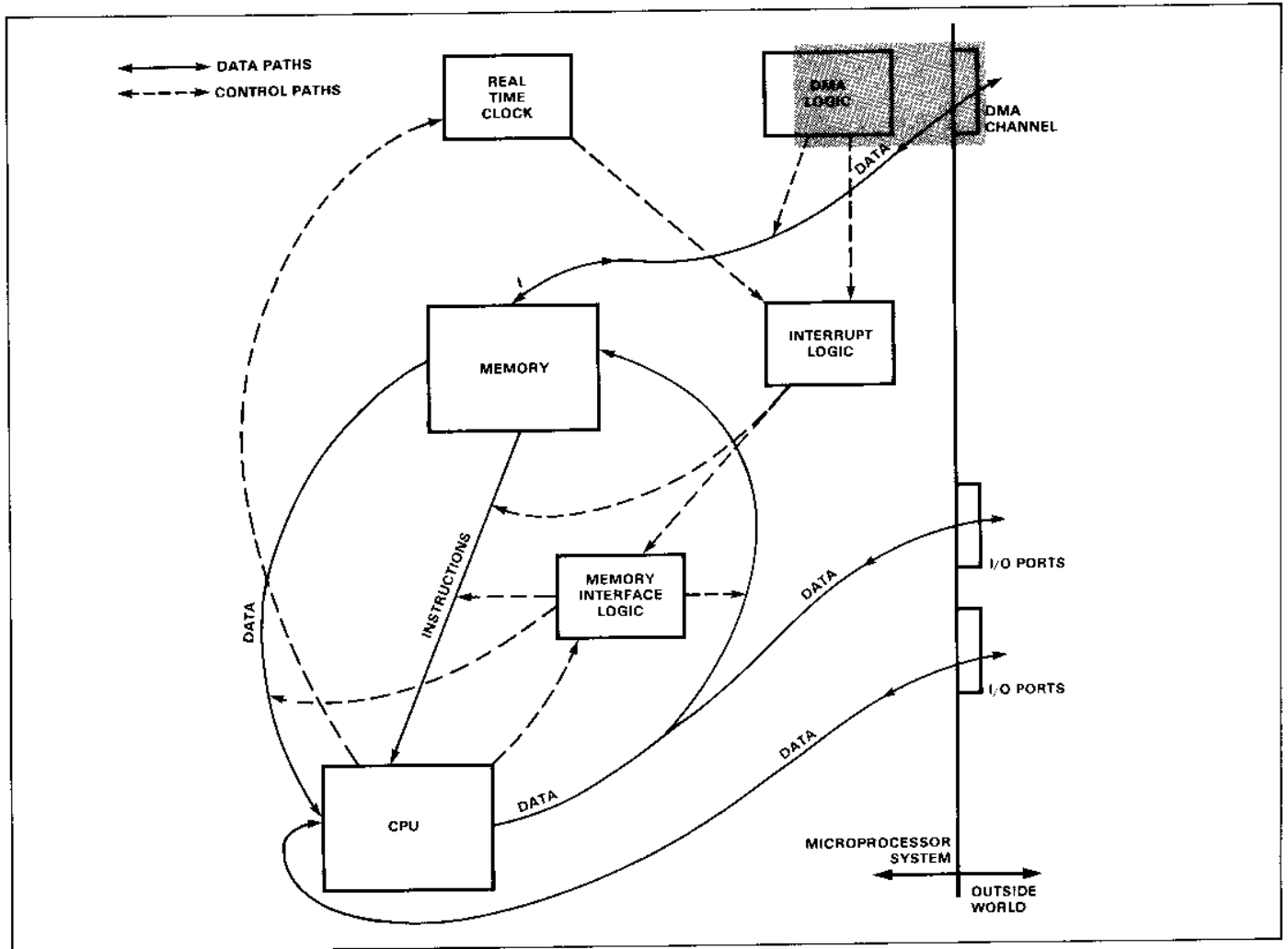


Fig. 2-12. Logical Functions of the 3854 DMA Device

BUFA, BUFB, BUFC and BUFD are buffer names used in Section 8.4.2, which describe DMA programming.

ADDRESS REGISTER

This is a 16-bit register which holds the address of the next memory byte to be accessed for a DMA data transfer.

Before a DMA operation is initiated, the beginning memory address for the data block which is to be transferred must be loaded (using appropriate F8 instructions) into the two ports set aside as the address register. As each data byte is transferred (input or output), the contents of the address register are automatically incremented.

BYTE COUNT REGISTER

This is a 12-bit register which acts as a counter, allowing blocks of up to 4096 data bytes to be transferred during a DMA operation. As described in Section 2.8.2, it is possible to execute DMA transfers without using the Byte Count register.

If the Byte Count register is in use, it is decremented as each byte of data is transferred, until it is decremented to 0; data transfer then stops.

CONTROL REGISTER

This is a 4-bit register which controls DMA operations as described next.

2.8.2 DMA Control Codes

The Control Register has four bits which control DMA operations as follows:

Bit 7 - ENABLE

This bit must be set to 1 in order to initiate a DMA operation; it is automatically reset to 0 when the DMA operation has run to completion.

Bit 6 - DIRECTION

If this bit is 0, data is transferred from main memory to the external device. If this bit is 1, data is transferred from the external device to main memory.

Bit 5 - INDEF

If this bit is 0, the Byte Count register controls the DMA transfer, which halts when the Byte Count register is decremented to 0. If this bit is 1, the Byte Count register is ignored and DMA transfer continues until the ENABLE bit is reset to 0 under program control.

Bit 4 - HIGHSPEED

If this bit is 0, the external device controls the rate at which data is transferred. If this bit is set to 1, a data byte will be transferred during every available DMA time slot; the external device must be capable of transmitting or receiving the data at the execution cycle speed of the F8 system.

F8 PROGRAMS

Individual instructions of the F8 assembly language instruction set exercise all of the capabilities of every device described in Section 2. Before studying individual instructions, however, it is necessary to understand what a program is, how a program is written, and how the written program becomes a PSU that drives the microprocessor system.

3.1 FLOWCHARTING

An application which is to be implemented using a microprocessor is specified using a flowchart; this differs from hardware logic diagrams only in the symbols used and the operations specified at each mode. The following four symbols will usually be sufficient in any microprocessor program flowchart:

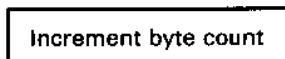
1) Beginning and End

A program may have one or more initiation or termination points. Identify each with the symbols:



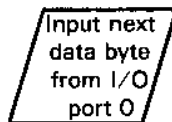
2) Internal operations

Enclose words in a rectangular box to identify each step of a program. Here is an example:



3) I/O operations

Use a parallelogram to identify I/O operations. Here is an example:



4) Decisions

Use a diamond to identify decisions. Here is an example:

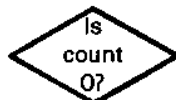


Figure 3-1 flowcharts a very simple program that moves data from one buffer in RAM to another buffer in RAM.

Figure 3-2 flowcharts a program that performs a multibyte addition. Observe that arrows identify the possible logic flow paths.

3.2 ASSIGNING MEMORY

Having flowcharted an entire application, the next step is to identify and name every buffer and variable to be referenced by the program. Names must conform to the rules of symbol syntax, described in Section 4.2.3., and will be used by the program to specify individual buffers and variables.

Before starting to write a program, assign space in scratchpad and in ROM or RAM memory for each buffer and every variable. These assignments will probably change before the program is finalized; nevertheless, it is important to have a clearly mapped data area at all times. Note also that the same scratchpad or RAM memory bytes may be used by different variables within one program, providing the different uses never overlap.

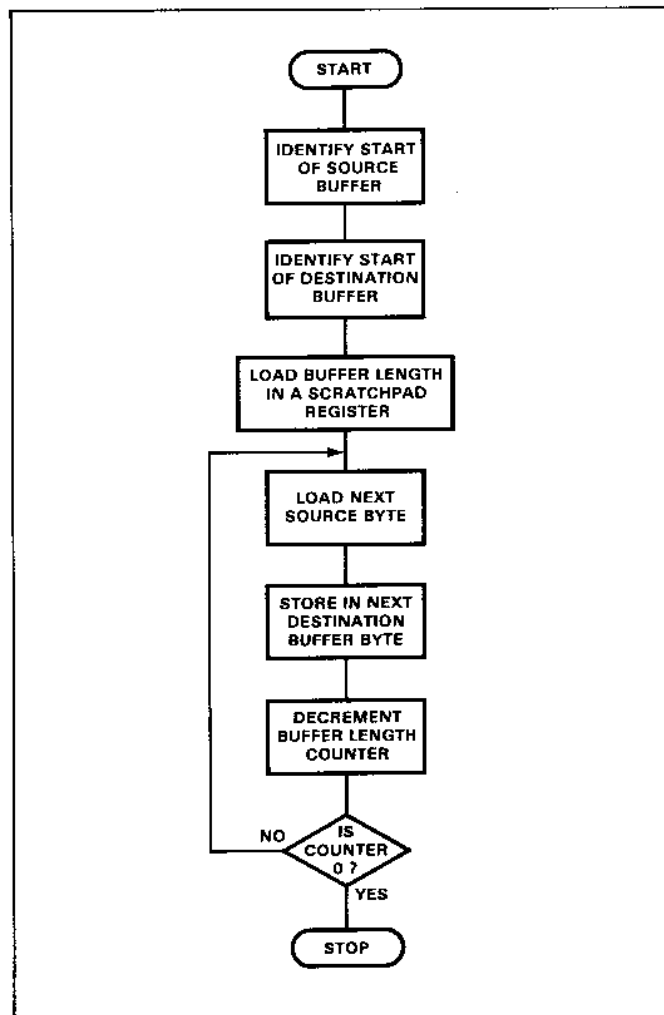


Fig. 3-1. Flowchart for a Program to Move Data from One RAM Buffer to Another

Recall that scratchpad registers are addressed by the ISAR register in the 3850 CPU, and are numbered from 0 to 63. ROM and RAM are addressed by the DCO register when accessing data. (Every 3851, 3852 and 3853 device has its own DCO register.) ROM and RAM bytes have addresses numbered from 0 to 65535.

With regard to addresses, note the following:

- 1) The first 64 bytes of ROM/RAM may have addresses that are the same as the Scratchpad Register addresses. No confusion is possible since the scratchpad is addressed via ISAR while ROM and RAM are addressed via DCO.
- 2) ROM and RAM byte addresses must not overlap.
- 3) Memory addresses must be contiguous within one device, but need not be contiguous from device to device. For example, three 3851 PSU may decode addresses from 0 to 1023, from 2048 to 3071, and from 3072 to 4095. Addresses 1024 to 2047 may be unused. (Recall that each 3851 PSU contains 1024 bytes of memory.)

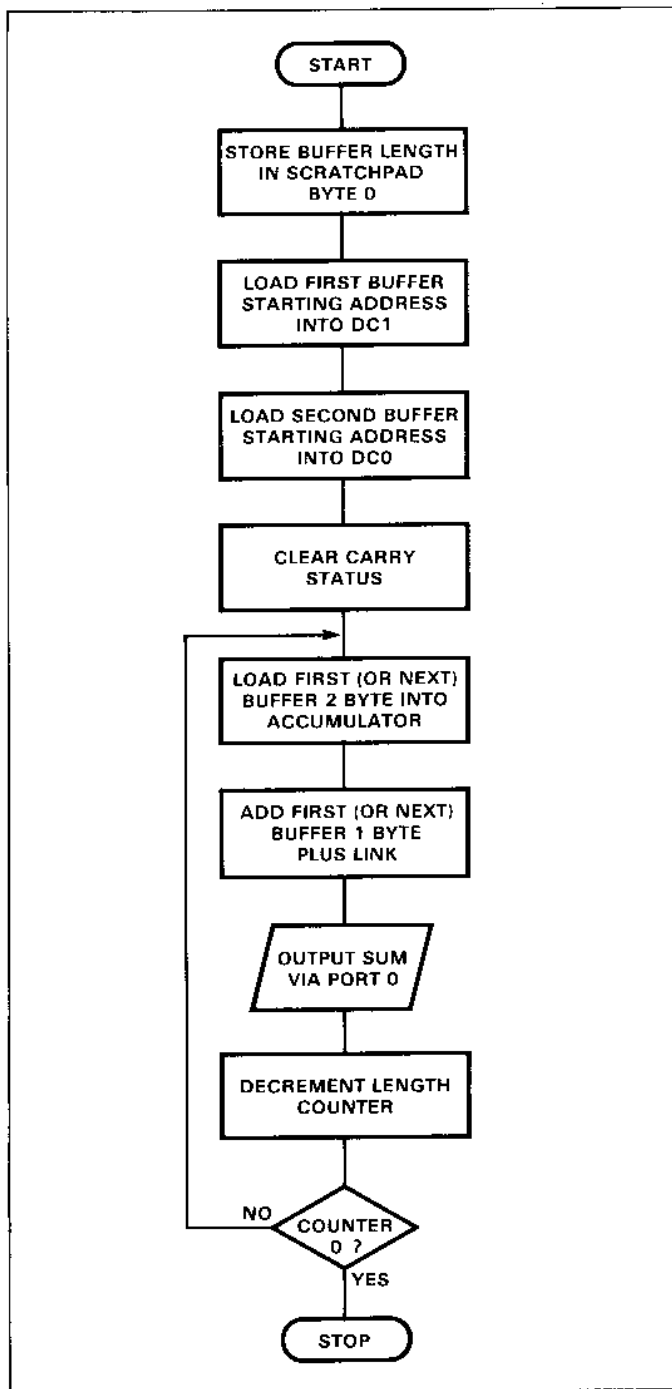


Fig. 3-2. Flowchart for Program to Add Two Multibyte Numbers and Output the Result

3.3 SOURCE AND OBJECT PROGRAMS

What eventually makes an F8 microprocessor system perform its assigned tasks is a sequence of binary digits, stored in memory and called an object program.

Since the F8 microprocessor accesses memory in 8-bit (or 1-byte) units, the binary digits of an object program are, by convention, collected into 8-bit units which are represented on paper as two hexadecimal digits (each hexadecimal digit is equivalent to four binary digits).

Upon examining the contents of any individual byte of memory, it would be impossible to determine what the eight binary

digits contained by the memory byte represented. A memory byte could hold any of the following types of information:

- 1) An instruction code which the 3850 CPU is supposed to interpret as an instruction.
- 2) Binary data which may be unsigned (representing numbers between 0 and 255) or signed (representing numbers between -128 and +127).
- 3) Data, as in 2) above, which provide specific information needed by an instruction code as in 1) above.
- 4) Data which are to be interpreted as representing a character that may be displayed or printed. Character codes are given in Appendix B.

How, then, will an F8 system pick its way through the various types of data which may be found stored in memory?

The program counter register (PC0) which is included in every 3851, 3852 or 3853 device, will at all times contain the address of the next memory byte whose content is to be interpreted by the 3850 CPU as an instruction code. When an F8 system is first powered up, the program counter is initialized at zero. Therefore, the contents of the memory byte with address 0 will be interpreted as the first instruction code to be executed. PC0 also addresses data bytes of type 3.

Whenever the content of a memory byte is to be interpreted as data of type 2 or 4, the address of the memory byte is contained in the data counter registers (DC0), which are also present on every 3851, 3852 or 3853 device.

It is not easy to immediately understand that the 3850 CPU is able to pick its way through object program numeric codes, as stored in memory, by suitably manipulating the program counter and data counter register contents; but fortunately, such understanding is not necessary in order to write F8 programs. In fact, even though microprocessor programs could be created directly as a sequence of hexadecimal digits, the potential for making errors when writing such programs is so overwhelming, that were an alternative method not available, the computer industry would never have gotten off the ground. The alternative is to write source programs.

A source program is a program written in a programming language. In the case of the F8, this manual describes what is called an assembly language. A programming language represents data and instruction sequences in a manner which is meaningless to a microprocessor but easily read and understood by a human.

Look at Figure 3-3. Upon first inspection, the part of the figure identified as a source program will not make much sense; the purpose of this manual is to explain how such source programs are written. Nevertheless, it is immediately evident that the source program is potentially much easier to read and understand than the equivalent object program.

The process of converting a source program to an object program is automatic and is handled by an assembler which is, itself, a computer program. The assembler interprets a source program, character-by-character, then generates an equivalent object program in a form that can be loaded into an F8 microprocessor system memory and executed.

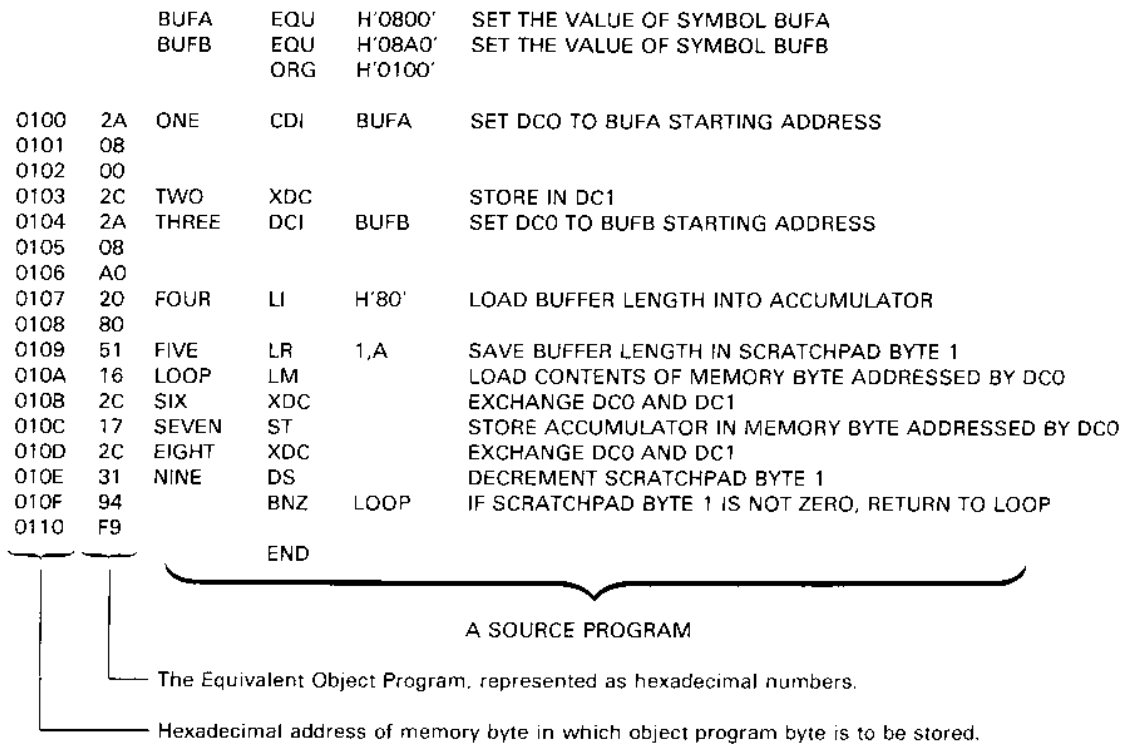


Fig. 3-3. Source and Object Programs

After the assembler has created the object program equivalent of a source program, it will print its results, outputting a program listing. The program listing provides information used to detect errors in a source program.

The rest of this manual explains how source programs are written as follows:

Every line of a source program constitutes one instruction. In Section 4, the various parts of an instruction are defined.

Section 5 and 6 define two classes of instructions used by the F8 assembly language. The consequences of every executable instruction's execution are defined.

Section 7 describes how individual instructions are combined in order to create a program. Therefore, the source program in Figure 3-3 will not be meaningful until you have completed reading Section 7.

Section 8 explains how programs should be written to access the various input and output features of the F8 microprocessor system.

In summary, the process of writing an F8 program follows these steps:

- 1) Using pencil and paper, write a source program.
- 2) Enter the source program, as text, into the computer system being used to develop F8 object programs.
- 3) Assemble the source program entered in Step 2, and thus create an object program. This step merely involves executing a program called the Assembler, identifying the source program and assigning a name to the object program.
- 4) If the source program contains illegal steps, they will be identified in Step 3. Treating the source program as text, edit out the errors, then return to Step 3. If there are no errors indicated at the end of Step 3, go on to Step 5.
- 5) Using appropriate Fairchild provided debugging aids, run the program created in Step 4 in order to find logic errors. If errors are found, correct them in the source program and return to Step 3. When there are no errors, the program is complete.

This manual provides information needed to perform Step 1. The F8 Timeshare Operating Systems Manual provides information needed for Steps 2 through 5.

During Step 3, the program listing is printed out on a line printer or time share terminal. The program listing shows the source and equivalent object program instructions, as well as additional, optional material that may be specified using assembler directives described in Chapter 5. Use the program listing to visually check a program; mark on the program listing all changes that must be made to the source program.

ASSEMBLY LANGUAGE SYNTAX

A very specific set of rules apply to the way in which an assembly language source program is written.

An assembly language program consists of a number of instructions, each of which occupies one line of text. There are four parts (or fields) to an instruction; one or more fields may contain non-blank information. Definite rules cover the characters that may be used in an instruction and how each character will be interpreted, depending on in which field the character appears.

The rules covering the way in which assembly language source programs are written are referred to collectively as the syntax of the assembly language. Assembly language syntax will be described with reference to the data moving program flowcharted in Figure 3-1 and illustrated in Figure 3-3.

4.1 INSTRUCTION TYPES

There are three types of source program statements: comments, executable instructions and assembler directives.

4.1.1 Comments

Comment instructions are used to insert remarks in the program in order to identify the program, separate program sections or make the source program easier to follow. A comment instruction does not have any computer related function, nor does it generate any object code; therefore, there is no restriction on its format or characters. An asterisk (*) character in column 1 designates the line of text as a comment instruction. Following the asterisk, there can be up to 71 characters of comment. Figure 4-1 illustrates comment lines in a source program.

4.1.2 Executable Instructions

Executable instructions are the steps that implement the procedure being programmed. For every executable instruction, the assembler generates one, two or three bytes of object code.

4.1.3 Assembler Directives

Assembler directives provide the assembler with additional information about the program. They are used to control the assembly process and in some cases cause data, which is included in the object code, to be generated.

4.2 INSTRUCTION FIELDS

Executable instructions and assembler directives have the following four fields:

1. Label field
2. Mnemonic field
3. Operand field
4. Comment field

Executable instructions and assembler directives must be formatted in a specific manner in order to be properly interpreted by the F8 Assembler. This means that each part of a source program instruction must be placed in its designated position or "field".

4.2.1 Label Field

The label field provides a means for assigning a name to a specific instruction. Any valid symbol (see Section 4.3.2) may be used in the label field. The label field begins in column 1 and may have any length; however, only the first four char-

| | | | | | |
|------|----|---|-----|---------|---|
| | | BUFA | EQU | H'0800' | SET THE VALUE OF SYMBOL BUFA |
| | | BUFB | EQU | H'08A0 | SET THE VALUE OF SYMBOL BUFB |
| | | | ORG | H'0100' | |
| | | *THE FOLLOWING PROGRAM MOVES DATA FROM ONE 128 BYTE *BUFFER TO ANOTHER 128 BYTE BUFFER | | | |
| 0100 | 2A | ONE | DCI | BUFA | SET DC0 TO BUFA STARTING ADDRESS |
| 0101 | 08 | | | | |
| 0102 | 00 | | | | |
| 0103 | 2C | TWO | XDC | | STORE IN DC1 |
| 0104 | 2A | THREE | DCI | BUFB | SET DC0 TO BUFB STARTING ADDRESS |
| 0105 | 08 | | | | |
| 0106 | A0 | | | | |
| 0107 | 20 | FOUR | LI | H'80' | LOAD BUFFER LENGTH INTO ACCUMULATOR |
| 0108 | 80 | | | | |
| 0109 | 51 | FIVE | LR | 1,A | SAVE BUFFER LENGTH IN SCRATCHPAD BYTE 1 |
| 010A | 16 | LOOP | LM | | LOAD CONTENTS OF MEMORY BYTE ADDRESSED BY DC0 |
| 010B | 2C | SIX | XDC | | EXCHANGE DC0 AND DC1 |
| 010C | 17 | SEVEN | ST | | STORE ACCUMULATOR IN MEMORY BYTE ADDRESSED BY DC0 |
| 010D | 2C | EIGHT | XDC | | EXCHANGE DC0 AND DC1 |
| 010E | 31 | NINE | DS | 1 | DECREMENT SCRATCHPAD BYTE 1 |
| 010F | 94 | | BNZ | LOOP | IF SCRATCHPAD BYTE 1 IS NOT ZERO, RETURN TO LOOP |
| 0110 | FA | | | | |
| | | | END | | |

Fig. 4-1. Four Comment Lines (Shaded) in a Source Program

acters are recognized by the assembler. The label field is terminated by a blank character. Figure 4-2 identifies label fields.

Label fields are frequently optional. With reference to Figure 4-2, notice that only three instruction labels, BUFA, BUFB and LOOP are necessary; they are the only labels referenced by other instructions.

4.2.2 Mnemonic Field

The mnemonic field contains the Operation Code (op code), which identifies the operation to be performed. There are two classes of operations accepted by the Assembler:

1. Assembler directives (Section 5)
2. CPU instructions (Section 6)

The mnemonic field may begin in any column other than column 1, and is terminated by a blank space. Figure 4-3 identifies mnemonic fields in a program.

In Figure 4-3, assembler directives are identified; notice that these assembler directives generate no object code.

4.2.3 Operand Field

The operand field consists of additional information (e.g., parameters, addresses) required by the Assembler to interpret the mnemonic field completely. The operand field may contain a symbol or expression (see Sections 4.3.2 and 4.3.4). The operand field must be separated from the mnemonic field by at least one blank; also, the operand field must be terminated by a blank. Figure 4-4 identifies the operand fields of a program. Notice that many instructions require no information in the operand field.

Instruction FOUR in Figure 4-4 illustrates the function served by operand fields. When executed, this instruction causes the byte value specified in the operand field to be loaded into the 3850 CPU accumulator register. In response to the source program instruction, the assembler generates an object program byte of H'20' representing the mnemonic "LI", the numeric value in the operand field is placed, by the assembler, in the next object program byte.

4.2.4 Comment Field

The comment field is optional and provides additional information that makes the source program easier to read. This

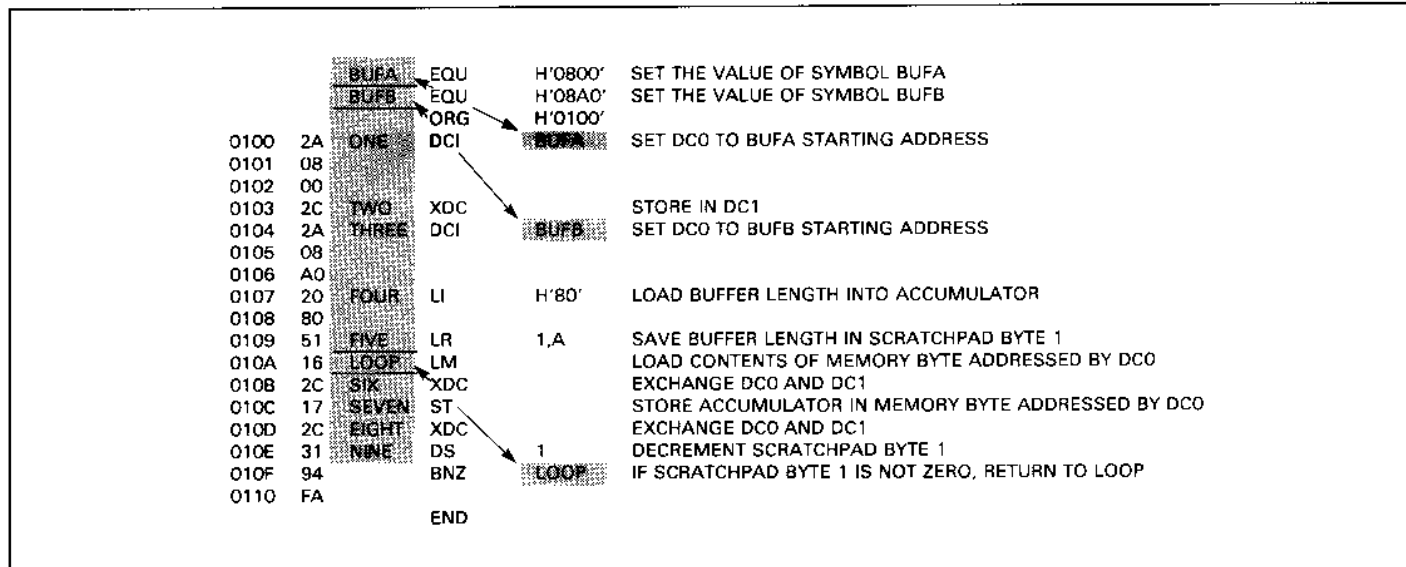


Fig. 4-2. Label Fields (Shaded) in a Source Program

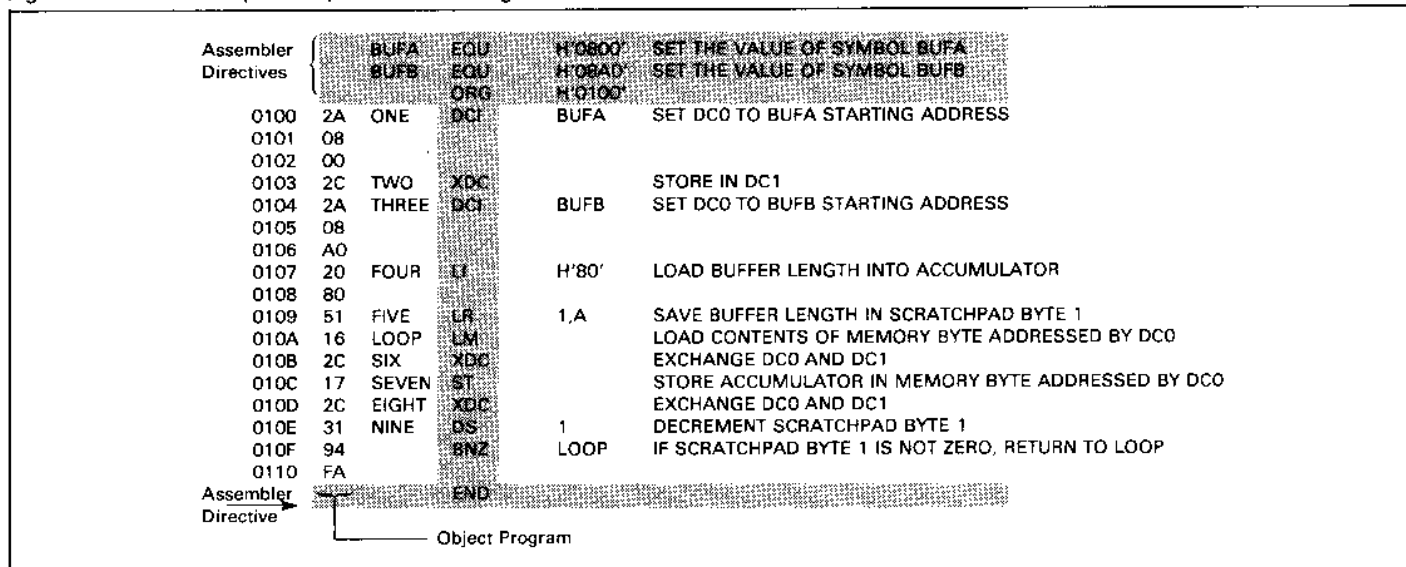


Fig. 4-3. Mnemonic Field (Vertical Shaded) in a Source Program

field is ignored by the Assembler and generates no object code. The comment field must be separated from the operand field (or the mnemonic field if there is no operand field) by at least one blank; it continues to the end of the text line.

Figure 4-5 identifies the comment fields of a program.

4.2.5 Aligning Fields

Figure 4-6 illustrates the source program of Figures 4-1 to 4-5, with a single space code separating each field of every instruction.

Clearly the program in Figure 4-6 is hard to read. For clarity it is recommended that all fields be aligned within character positions of every line; here is one possibility:

Label field: Characters 1 to 6
 Mnemonic field: Characters 7 to 11
 Operand field: Characters 12 to 19
 Comment field: Characters 20 to 72

```

BUFA EQU H'0800' SET THE VALUE OF SYMBOL BUFA
BUFB EQU H'08A0' SET THE VALUE OF SYMBOL BUFB
  ORG H'0100'
ONE DCI BUFA SET DCO TO BUFA STARTING ADDRESS
TWO XDC STORE IN DC1
THREE DCI BUFB SET DCO TO BUFB STARTING ADDRESS
FOUR LI H'80' LOAD BUFFER LENGTH INTO ACCUMULATOR
FIVE LR 1,A SAVE BUFFER LENGTH IN SCRATCHPAD BYTE 1
LOOP LM LOAD CONTENTS OF MEMORY BYTE ADDRESSED BY DCO
SIX XDC EXCHANGE DCO AND DC1
SEVEN ST STORE ACCUMULATOR IN MEMORY BYTE ADDRESSED BY DCO
EIGHT XDC EXCHANGE DCO AND DC1
NINE DS 1 DECREMENT SCRATCHPAD BYTE 1
  BNZ LOOP IF SCRATCHPAD BYTE 1 IS NOT ZERO. RETURN TO LOOP
END
  
```

Fig. 4-6. A Source Program with Unaligned Fields

| | | | | | | |
|------|----|-------|------|-------|---------|---|
| | | | BUFA | EQU | H'0800' | SET THE VALUE OF SYMBOL BUFA |
| | | | BUFB | EQU | H'08A0' | SET THE VALUE OF SYMBOL BUFB |
| | | | | ORG | H'0100' | |
| 0100 | 2A | ONE | DCI | BUFA | | SET DCO TO BUFA STARTING ADDRESS |
| 0101 | 08 | | | | | |
| 0102 | 00 | | | | | |
| 0103 | 2C | TWO | XDC | | | STORE IN DC1 |
| 0104 | 2A | THREE | DCI | BUFB | | SET DCO TO BUFB STARTING ADDRESS |
| 0105 | 08 | | | | | |
| 0106 | A0 | | | | | |
| 0107 | 20 | FOUR | LI | H'80' | | LOAD BUFFER LENGTH INTO ACCUMULATOR |
| 0108 | 80 | | | | | |
| 0109 | 51 | FIVE | LR | 1,A | | SAVE BUFFER LENGTH IN SCRATCHPAD BYTE 1 |
| 010A | 16 | LOOP | LM | | | LOAD CONTENTS OF MEMORY BYTE ADDRESSED BY DCO |
| 010B | 2C | SIX | XDC | | | EXCHANGE DCO AND DC1 |
| 010C | 17 | SEVEN | ST | | | STORE ACCUMULATOR IN MEMORY BYTE ADDRESSED BY DCO |
| 010D | 2C | EIGHT | XDC | | | EXCHANGE DCO AND DC1 |
| 010E | 31 | NINE | DS | 1 | | DECREMENT SCRATCHPAD BYTE 1 |
| 010F | 94 | | BNZ | LOOP | | IF SCRATCHPAD BYTE 1 IS NOT ZERO. RETURN TO LOOP |
| 0110 | FA | | | | | |
| | | | | | | END |

Fig. 4-4. Operand Fields (Shaded) in a Source Program

| | | | | | | |
|------|----|-------|------|-------|---------|---|
| | | | BUFA | EQU | H'0800' | SET THE VALUE OF SYMBOL BUFA |
| | | | BUFB | EQU | H'08A0' | SET THE VALUE OF SYMBOL BUFB |
| | | | | ORG | H'0100' | |
| 0100 | 2A | ONE | DCI | BUFA | | SET DCO TO BUFA STARTING ADDRESS |
| 0101 | 08 | | | | | |
| 0102 | 00 | | | | | |
| 0103 | 2C | TWO | XDC | | | STORE IN DC1 |
| 0104 | 2A | THREE | DCI | BUFB | | SET DCO TO BUFB STARTING ADDRESS |
| 0105 | 08 | | | | | |
| 0106 | A0 | | | | | |
| 0107 | 20 | FOUR | LI | H'80' | | LOAD BUFFER LENGTH INTO ACCUMULATOR |
| 0108 | 80 | | | | | |
| 0109 | 51 | FIVE | LR | 1,A | | SAVE BUFFER LENGTH IN SCRATCHPAD BYTE 1 |
| 010A | 16 | LOOP | LM | | | LOAD CONTENTS OF MEMORY BYTE ADDRESSED BY DCO |
| 010B | 2C | SIX | XDC | | | EXCHANGE DCO AND DC1 |
| 010C | 17 | SEVEN | ST | | | STORE ACCUMULATOR IN MEMORY BYTE ADDRESSED BY DCO |
| 010D | 2C | EIGHT | XDC | | | EXCHANGE DCO AND DC1 |
| 010E | 31 | NINE | DS | 1 | | DECREMENT SCRATCHPAD BYTE 1 |
| 010F | 94 | | BNZ | LOOP | | IF SCRATCHPAD BYTE 1 IS NOT ZERO. RETURN TO LOOP |
| 0110 | FA | | | | | |
| | | | | | | END |

Fig. 4-5. Comment Fields (Shaded) in a Source Program

4.3 LANGUAGE COMPONENTS

4.3.1 Valid Characters

The F8 Assembler accepts all characters available on an input terminal as valid characters. Alphabetic (A-Z), numeric (0-9), and special (all other terminal characters) characters are valid when correctly used; in other words, there is no character which will always be invalid.

Some characters have been assigned special meaning; the use of these special characters is therefore restricted, as described in the following sub-sections, and summarized in Table 4-1.

| Restricted Character | Function | Example |
|----------------------|-------------------------------|-------------|
| D | Specify decimal constants | D'1234' |
| H | Specify hexadecimal constants | H'123A' |
| B | Specify binary constants | B'10011101' |
| O | Specify octal constants | O'23714' |
| C | Specify character constants | C'VALID' |
| T | Specify timer counts | T'123' |
| * | Current memory location | *+3 |
| * | Multiplication sign | (VAL*2) |
| ** | Exponentiation sign | (VAL**2) |
| + | Addition sign | (VAL+2) |
| - | Subtraction sign | (VAL-2) |
| / | Division sign | (VAL/2) |
| (| Beginning of an expression | (VAL+2) |
|) | End of an expression | (VAL+2) |
| . | Separate operands | A,1 |

Table 4-1. A Summary of Restricted Characters

Restricted characters may be used in any way that does not directly conflict with the restricted use.

4.3.2 Constants

Constants represent quantities or data that do not vary in value during the execution of a program. The syntax for constants' representation is described below.

DECIMAL

A decimal number consists of a string of from one to five numeric characters. The number may be preceded by a minus "-" sign but no blanks are allowed within the number. The value of a decimal digit must fall in the range +32767 to -32768. Optionally, decimal numbers may be enclosed between single quotes, preceded by a D character.

Examples:

| Valid | Invalid | Reason invalid |
|----------|----------|-------------------------------------|
| 12 | 123456 | Too many digits |
| -123 | 123- | Invalid character |
| 12345 | 12.3 | Invalid character |
| -5432 | 12b3 | Invalid character |
| 23456 | 65432 | Above +32767 |
| D'12' | '12'D | D does not precede number in quotes |
| D'23456' | D'65432' | Above +32767 |

HEXADECIMAL

A hexadecimal number consists of a string of from one to four numeric characters and/or alphabetic characters (A to F inclusive) enclosed in single quotes and preceded by an H. No blanks are allowed within the number or between the H and the number. Hexadecimal numbers in the range H'0' to H'FFFF' are valid. Signed hexadecimal numbers are invalid.

Examples:

| Valid | Invalid | Reason invalid |
|---------|----------|-----------------------|
| H'12' | 'ABCD' | No preceding H |
| H'ABCD' | H'-12' | Invalid character (-) |
| H'1AFO' | H'12.A3' | Invalid character (.) |

BINARY

A binary number consists of a string of from 1 to 16 ones or zeroes, enclosed within a pair of quotes and preceded by a B. No blanks are allowed between the apostrophe symbols, or between the B and the number. If there are less than 16 binary digits, leading 0 digits are assumed.

Examples:

| Valid | Invalid | Reason invalid |
|-----------|----------------------|-------------------|
| B'101101' | B1011101 | No quotes |
| B'0010' | B'10110111011100101' | Too many digits |
| | B'10021' | Invalid digit (2) |

OCTAL

An octal number consists of a string of from one to six numeric digits, excluding 8 or 9, enclosed between single quotes and preceded by an O. Octal numbers in the range O'0' to O'17777' are valid. Signed numbers are invalid.

Examples:

| Valid | Invalid | Reason invalid |
|-----------|-----------|-----------------------|
| O'17243' | O17243 | No quotes |
| O'2462' | '2462' | No preceding O |
| O'177272' | O'277272' | Value exceeds maximum |
| O'23714' | O'23914' | Invalid character (9) |

CHARACTERS

Any characters (other than the single quote character) may be enclosed in single quotes and preceded by a C, in which case the characters will be interpreted as ASCII characters (see Appendix B).

Examples:

| Valid | Invalid | Reason invalid |
|----------|---------|-------------------------|
| C'VALID' | 'VALID' | No preceding C |
| C'12345' | C12345' | No initial single quote |
| C'NAME' | C"NAME" | Double quotes |

TIMER COUNTS

As described in Section 2, the 3851 PSU and the 3853 Memory Interface device each have a timer which may be loaded under program control. Depending on the value loaded into the timer, variable delays may be programmed, at the end of which a timer interrupt is transmitted to the 3850 CPU.

Timer counts may be entered, as decimal numbers between 0 and 255, enclosed in single quotes and preceded by a T. The assembler converts the timer count to the exact binary code which (based on the timer logic) will generate the required time delay. Appendix C provides the exact codes that correspond to each timer count entered using T'nn' format.

Recall that the exact time delay is given by the equation:

$$\text{Delay} = (\text{timer counts}) * 31 * \text{Clock period}$$

Examples:

| Valid | Invalid | Reason invalid |
|--------|---------|-----------------------|
| T'25' | T25 | No single quotes |
| T'127' | T'12A' | Invalid character (A) |
| T'254' | T'264' | Count too high |

4.3.3 Symbols

A symbol is a character string of from one to four characters, the first of which must be alphabetic (A-Z). A symbol may have any number of characters; however, only the first four characters are interpreted by the assembler. A symbol cannot have the exact appearance of a number, as specified in Section 4.3.2.

Since a blank space acts as a field delimiter, it cannot be present as a character within a symbol.

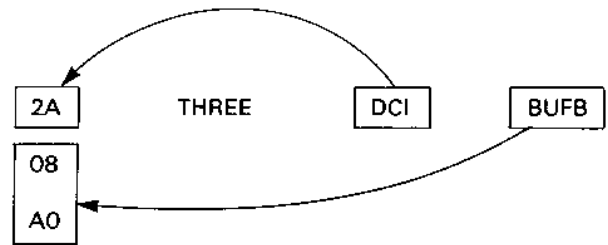
Examples:

| Valid | Invalid | Reason invalid |
|---------|---------|---|
| ABCD | AB CD | A blank present. AB is the assumed symbol |
| AB12 | 12AB | A numeric first character |
| D12 | D'12' | Would be interpreted as decimal 12 |
| SYMBOLA | SYMBOLB | Both symbols are SYMB |

Figure 4-7 illustrates a number of symbols in a source program. Observe that symbols may appear in the label field or the operand field of an instruction.

When a symbol appears in the label field of an instruction, it is either assigned a value by that instruction (EQU) or it is assigned a value equal to the location of that instruction, depending on the nature of the instruction. Sections 5.5 and 5.7 describe how this is done.

When a symbol appears in the operand field of an instruction, the assembler substitutes the assigned value for the symbol. For example, instruction THREE in Figure 4-7 causes the value associated with symbol BUFB to be loaded into the DC0 registers of all memory and memory interface devices. Instruction THREE therefore generates the following object code:



4.3.4 Expressions

Expressions may appear in the operand field of an instruction, and are evaluated by the assembler to generate a constant which is used in the object program.

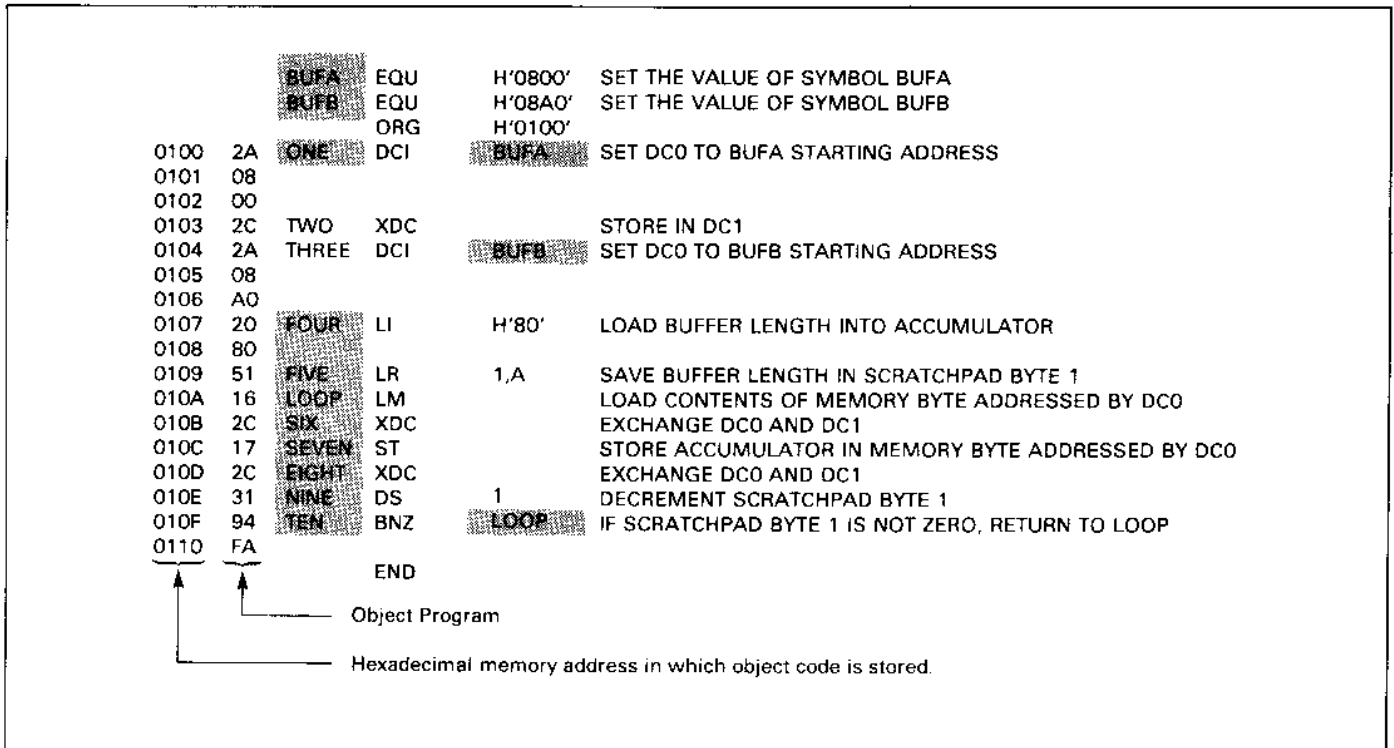


Fig. 4-7. Symbols in a Source Program

Unlike higher level languages, expressions do not represent equations to be resolved at execution time. By the time a program is executed, every expression in the source program will have been converted (by the assembler) to a constant in the object program.

An expression can have three types of numeric value, linked by six types of algebraic symbol.

These are the three types of numeric value:

- 1) Any symbol, as defined in Section 4.3.3.
- 2) Any constant numeric value, as defined in Section 4.3.2.
- 3) An asterisk (*), which will be interpreted as having the value of the memory address into which the first object program byte for this instruction will be stored.

These are the six algebraic symbols that are recognized:

- 1) + for add
- 2) - for subtract

- 3) * for multiply
- 4) / for divide
- 5) ** for exponentiate
- 6) (and) to enclose expression and subexpressions, which are to be evaluated as a constant.

Expressions and subexpressions must be enclosed in brackets. An exception is the simple (and most frequently used) expression:

*±numeric constant

Subexpressions may be nested ten deep.

Use of complex expressions is pointless, since it is almost as simple to evaluate the expression and use the evaluated result in the object program. The one time when expressions are useful is when calculating instruction addresses. Referring to Figure 4-7, the following are substitutes for LOOP in the operand field of instruction TEN:

*-5 (equals H'010F' - 5)
(FOUR+3) (equals H'0107' + 3)

ASSEMBLER DIRECTIVES

Assembler directives are instructions to the assembler; as such, they generate no object code. Assembler directives provide the assembler with the following three types of information:

- 1) Values of symbols
- 2) How memory is to be mapped
- 3) Assembly listings print options

Assembler directives are described in alphabetic order on the following pages. A summary of the assembler directives which are necessary, versus those which are optional, is given in Section 5.11; hints on good programming practice are also provided.

5.1 BASE - SELECT LISTING NUMERIC BASE

This is an optional directive which specifies the number system in which object program codes will be printed on the assembler printout. The following three options are provided:

| Label | Mnemonic | Operand | Comment |
|-------|----------|---------|---------------------------|
| | BASE | HEX | Select hexadecimal output |
| | BASE | OCT | Select octal output |
| | BASE | DEC | Select decimal output |

If no base is specified, decimal output will be selected by default. If a base is specified, one BASE instruction should appear at the beginning of the program, as illustrated in Figure 5-1.

Since hexadecimal notation is the standard for the F8 microprocessor, it is strongly recommended that programmers use this numeric option.

5.2 DC - DEFINE CONSTANT

This directive causes the assembler to generate a one or two byte constant. The DC directive is an exception in that it causes one or two bytes of object code to be generated—identical to the one or two byte constant specified.

The DC directive will usually have a label, which becomes the symbol via which the constant is referenced. The general format of the DC directive is:

```
Label Mnemonic Operand
LABEL DC VALUE
```

LABEL is any valid symbol. The label is optional. VALUE is any valid numeric value as described in Section 4.3.2.

For examples of DC directive use see Section 7.2.1. See also Section 5.5.1 for a discussion of when DC directives are used and when EQU directives are used.

5.3 EJECT - EJECT CURRENT LISTING PAGE

This directive has no effect on the program being assembled. It controls the line printer on which the assembler is printing out an assembly listing.

When the assembler encounters EJECT in the mnemonic field of an instruction, it immediately advances the line printer paper to the top of the next page.

If the assembler is not printing out an assembly listing, it will ignore the EJECT directive.

The format of the EJECT directive is:

```
Label Mnemonic Operand
EJECT
```

| Address | Label | Mnemonic | Operand | Comment |
|---------|-------|----------|----------|--|
| | | TITLE | | SAMPLE PROGRAM TO MOVE DATA BETWEEN BUFFERS |
| | | MAXCPU | 50 | LIMIT OF 50 SECONDS CPU TIME SPECIFIED |
| | | SYMBOL | | A SYMBOL TABLE WILL FOLLOW SOURCE PROGRAM |
| | | XREF | | SYMBOLS CROSS LISTING WILL FOLLOW SOURCE PROGRAM |
| | | BASE | HEX | HEXADECIMAL NUMBERS SPECIFIED FOR ASSEMBLY LISTING |
| | BUFA | EQU | H'0800' | SET THE VALUE OF SYMBOL BUFA |
| | BUFB | EQU | H'08A0' | SET THE VALUE OF SYMBOL BUFB |
| | | ORG | H'0100' | |
| 0100 | 2A | ONE | DCI BUFA | SET DC0 TO BUFA STARTING ADDRESS |
| 0101 | 08 | | | |
| 0102 | 00 | | | |
| 0103 | 2C | TWO | XDC | STORE IN DC1 |
| 0104 | 2A | THREE | DCI BUFB | SET DC0 TO BUFB STARTING ADDRESS |
| 0105 | 08 | | | |
| 0106 | A0 | | | |
| 0107 | 20 | FOUR | LI H'80' | LOAD BUFFER LENGTH INTO ACCUMULATOR |
| 0108 | 80 | | | |
| 0109 | 51 | FIVE | LR 1.A | SAVE BUFFER LENGTH IN SCRATCHPAD BYTE 1 |
| 010A | 16 | LOOP | LM | LOAD CONTENTS OF MEMORY BYTE ADDRESSED BY DC0 |
| 010B | 2C | SIX | XDC | EXCHANGE DC0 AND DC1 |
| 010C | 17 | SEVEN | ST | STORE ACCUMULATOR IN MEMORY BYTE ADDRESSED BY DC0 |
| 010D | 2C | EIGHT | XDC | EXCHANGE DC0 AND DC1 |
| 010E | 31 | NINE | DS 1 | DECREMENT SCRATCHPAD BYTE 1 |
| 010F | 94 | BNZ | LOOP | IF SCRATCHPAD BYTE 1 IS NOT ZERO, RETURN TO LOOP |
| 0110 | FA | | | |
| | | | END | |

Fig. 5-1. Assembler Directives (Shaded) in a Source Program

5.4 END - END OF ASSEMBLY

An END directive must terminate every source program. Upon encountering this directive, the assembler stops reading source program instructions, and starts to perform various post-assembly computations.

Figure 5-1 illustrates use of an END directive.

Note that an END directive cannot, and must not, have a label.

The format of the END directive is:

| | | |
|----------|----------|---------|
| Label | Mnemonic | Operand |
| Must | END | |
| be blank | | |

5.5 EQU - EQUATE A SYMBOL TO A NUMERIC VALUE

Every symbol in a source program must be the label of an assembly language instruction or a DC directive, or the symbol must be assigned a value by an EQU directive. The general format of an EQU directive is:

| | | |
|-------|----------|---------|
| Label | Mnemonic | Operand |
| LABEL | EQU | VALUE |

LABEL is any valid symbol.

VALUE is any valid numeric value as described in Section 4.2.2.

Refer to Figure 5-1. The symbols BUFA and BUFB appear in instructions ONE and THREE, and are assigned values by two EQU directives. Therefore:

| | | |
|------|-----|---------|
| BUFA | EQU | H'0800' |
| — | — | — |
| — | — | — |
| ONE | DCI | BUFA |

is identical in its net effect to:

| | | |
|-----|-----|---------|
| ONE | DCI | H'0800' |
|-----|-----|---------|

Why then are Equate directives used? In a real program, a symbol (such as BUFA) is likely to appear many times. If the value of the symbol changes, the program can be corrected by modifying one Equate directive, then re-assembling the program. If absolute values are used in instruction operands (instead of symbols), every instruction that references the absolute value must be changed in the source program if the absolute value changes; the source program must be re-assembled.

For example, suppose there are 24 instructions in a source program that reference the symbol BUFA. The Equate directive could be eliminated, in which case each of the 24 instructions would have H'0800' where it had BUFA. However, if H'0800' had to be changed, instead of making the change in one Equate directive, the change would have to be made in each of the 24 instructions.

5.5.1 A Comparison of the EQU and DC Directives

A common error made by novice programmers is to misuse the EQU and DC directives. The difference between the two must be clearly understood.

With reference to Figure 5-1, consider the following erroneous variation of the BUFA symbol's use:

| | | |
|------|-----|---------|
| | ORG | H'2FA0' |
| BUFA | DC | H'0800' |
| — | — | — |
| — | — | — |
| — | — | — |
| ONE | DCI | H'0100' |
| | | BUFA |

The DC directive causes the two byte, hexadecimal value H'0800' to be stored in two memory bytes, with addresses H'2FA0' and H'2FA1'. In instruction ONE, BUFA acquires the value H'2FA0', not H'0800'.

Now consider how the DC directives might be correctly used in the Figure 5-1 program. BUFB has been equated to H'08A0', which is the starting memory address of the source buffer. The source buffer contents could be specified, using DC directives, as follows:

| | | |
|-------|-----|---------|
| BUFA | EQU | H'0800' |
| | ORG | H'0100' |
| ONE | DCI | BUFA |
| TWO | XDC | |
| THREE | DCI | BUFB |
| — | — | — |
| — | — | — |
| — | — | — |
| BUFB | DC | H'08A0' |
| | DC | H'20A1' |
| | DC | H'143E' |
| | DC | H'5A62' |

The symbol BUFB no longer needs to be equated to H'08A0' since it appears as a label at address H'08A0'. The DC directives cause the data string H'20A1143E5A62' to be loaded into memory starting at memory location H'08A0'.

NOTE: When a buffer's contents are specified by DC directives, the buffer's data becomes part of the program, and are loaded into memory when the program is loaded into memory.

5.6 MAXCPU - SPECIFY MAXIMUM CPU TIME

This directive is only meaningful when the source program is being assembled on a large host computer (e.g., an IBM 360 or 370). On such large computers, programs exist to simulate the F8 microprocessor; therefore once the source program has been assembled, the object program may be "run" using the host computer simulator.

A potential problem lies in executing an object program which, due to programming errors, may run for ever; a large amount of costly host computer time may be expended before the existence of the error is detected. The MAXCPU directive specifies a maximum number of seconds of host computer execution time, after which program execution will be terminated.

Figure 5-1 illustrates the use of the MAXCPU directive, specifying a maximum of 50 seconds of host computer CPU time. Note that the MAXCPU directive cannot, and must not, have a label.

The format of the MAXCPU directive is:

| | | |
|-------|----------|----------|
| Label | Mnemonic | Operand |
| Must | MAXCPU | CONSTANT |
| | be blank | |

CONSTANT is any numeric constant as described in Section 4.3.2.

5.7 ORG - ORIGIN A PROGRAM

As described in Section 4.3.3, a symbol which is an instruction label acquires a value equal to the memory address of the first object program byte for the instruction. With reference to Figure 5-1, therefore:

ONE acquires the value of H'0100'
LOOP acquires the value of H'010A'

In order to assign values to instruction labels, the assembler has to know where the object program will be stored once it gets loaded into an F8 microprocessor system memory; this is done using the ORG directive.

When assembling a source program, the assembler maintains its own program counter, which tracks the memory addresses into which each byte of object program is destined to be stored. Whenever the assembler encounters an ORG directive, it resets its program counter to the address specified by the ORG directive. Thus in Figure 5-1 the ORG directive sets the effective memory address to H'0100' for the first object code byte of the first instruction that follows.

A program may have more than one ORG directive, depending on how subroutines and program modules have been mapped into memory. Any time there is a "gap" between one program module and the next, the new origin must be specified using an ORG directive.

The format of the ORG directive is as follows:

| | | |
|-------|----------|---------|
| Label | Mnemonic | Operand |
| Must | ORG | VALUE |
| | be blank | |

The ORG directive cannot and must not have a label.

VALUE is any valid numeric value as described in Section 4.3.2, or any valid expression as described in Section 4.3.4.

5.8 SYMBOL - ASSEMBLER PROVIDE A SYMBOL TABLE

This directive may optionally appear once, at the beginning of a source program, as illustrated in Figure 5-1.

If the assembler encounters SYMBOL in the mnemonic field of an instruction, it will print a symbol table at the end of the assembly listing. The SYMBOL directive cannot, and must not have a label.

A symbol table lists every symbol encountered in the source program, along with the value assigned to the symbol.

A symbol table allows errors in symbols to be spotted quickly. A misspelled symbol, for example, will appear in the symbol table as an extra, unexpected symbol.

5.9 TITLE - PRINT A TITLE AT THE HEAD OF THE ASSEMBLER LISTING

This is an optional directive, which, if present, causes a title to be printed at the top of every assembler listing page. The format of this directive is as follows:

| | | |
|-------|----------|---------------|
| Label | Mnemonic | Operand |
| Must | TITLE | "any heading" |
| | be blank | |

The heading must be enclosed in double quotes. The TITLE directive cannot, and must not, have a label.

5.10 XREF - ASSEMBLER PROVIDE A SYMBOL CROSS REFERENCE LISTING

This directive may optionally appear once, at the beginning of a source program, as illustrated in Figure 5-1.

If the assembler encounters XREF in the mnemonic field of an instruction, it will print a cross reference listing of symbols at the end of the assembly listing. The XREF directive cannot, and must not, have a label.

A cross reference listing shows every symbol encountered in the source program, plus the statement number at which the symbol was referenced (i.e., appeared in an instruction's operand field).

A cross reference listing allows misplaced or misspelled symbols to be quickly spotted and corrected.

5.11 WHEN TO USE ASSEMBLER DIRECTIVES

The END assembler directive must be present in a source program. Without this directive the program will not assemble correctly.

The ORG, DC and EQU directives are almost always used in a program. Symbols equated to a numeric value (using the EQU directive) are recommended instead of having numeric constants in instruction operands.

The remaining assembler directives are optional, to be used for programming efficiency and convenience only.

THE INSTRUCTION SET

Because of the nature of the F8 family of devices, program sequences are very dependent on device configurations. Many instructions are important in some device configurations, but do not apply, or are rarely used in other device configurations. Therefore, individual F8 instructions should be visualized as contributions to one (or more) of a number of common, identifiable operation sequences, rather than as equal entities.

It would be impossible to describe operation sequences without first defining individual instructions; therefore, individual instructions are defined in this section, and example programs representing common operation sequences are given in Sections 7, 8, 9 and 10.

In this section instructions are described in alphabetic order of the instruction mnemonic. This makes it easy to locate any instruction. Examples in this section are very primitive, and merely illustrate the operations performed by each instruction. Programs in Sections 7 through 10 are referenced for comprehensive and realistic examples. Instructions are grouped by type in Appendix D.

When instruction format is defined, optional items are enclosed in square brackets. For example:

[LABEL] ADC

means that the instruction ADC may, or may not have a label.

Tables 6-1 and 6-2 identify the terms and abbreviations used in Section 6.

| | |
|--------|---|
| Nval3 | - This symbol is used to indicate an instruction operand which defines the three low order bits of the instruction object code. |
| Nval4 | - This symbol is used to indicate an instruction operand which defines the four low order bits of the instruction object code. |
| Nval8 | - This symbol is used to indicate an instruction operand which defines the 8-bit second byte of the instruction object code. |
| Nval16 | - This symbol is used to indicate an instruction operand which defines the 8-bit second byte, plus the 8-bit third byte of the instruction object code. |

Table 6-1. Operand Symbols

Instructions described in the rest of Section 6 generate 1, 2 or 3 bytes of object code.

The first byte of object code is always the instruction operation code. Selected "short" instructions use three or four bits of the first byte to specify data.

The second byte of a 2-byte instruction provides either a signed, or an unsigned, binary number.

The second and third bytes of three byte instructions provide a 16-bit unsigned binary number.

| Value or Symbol for Sreg | Scratchpad Register Specified |
|--------------------------|---|
| 0 through 11 | The first 12 scratchpad registers are addressed directly. |
| 12 or S | The scratchpad register address is provided indirectly by ISAR. |
| 13 or I | As 12, but the low order three bits of ISAR are incremented after the scratchpad register is accessed.* |
| 14 or D | As 12, but the low order three bits of ISAR are decremented after the scratchpad register is accessed.* |

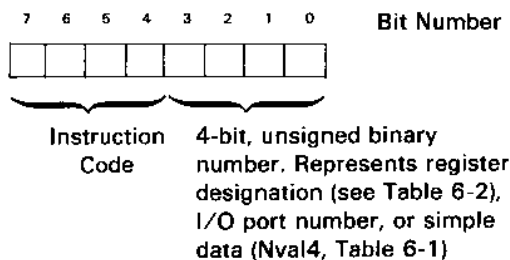
* Modification of ISAR is described in Section 2.4.2.

Table 6-2. Operands Referencing Scratchpad Memory, as Specified by Symbol Sreg

Object code types are illustrated below, with the instructions using each object code type identified by instruction mnemonic.

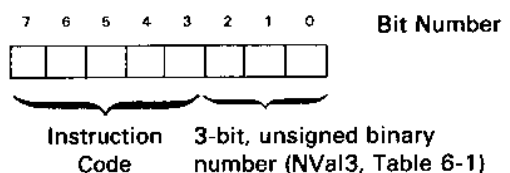
See Appendix D for actual object code byte contents.

One Byte, Type 1



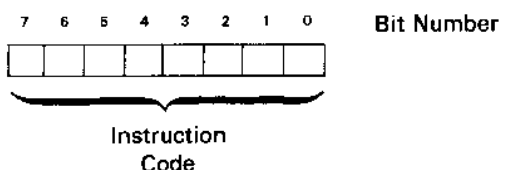
AS, ASD, CLR, DS, INS, LIS, LR (with Sreg), NS, OUTS, XS

One Byte, Type 2



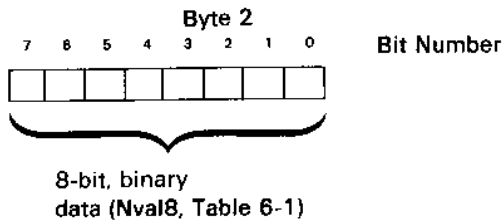
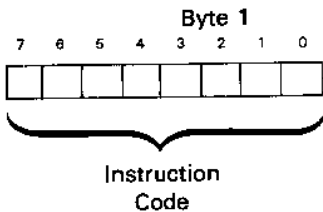
LISL, LISU

One Byte, Type 2



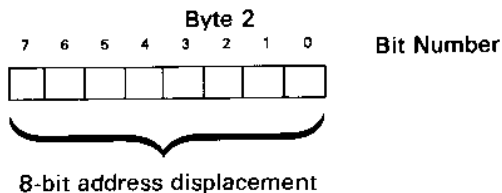
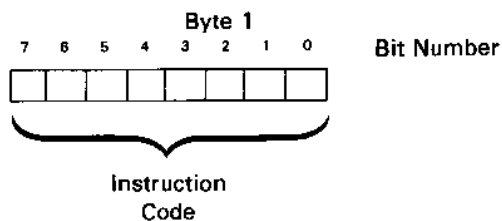
ADC, AM, AMD, CM, COM, DI, EI, INC, LM, LNK, LR (not with Sreg), NM, NOP, OM, PK, POP, SL, SR, ST, XDC, XM

Two Byte, Type 1



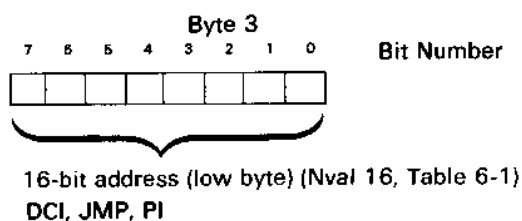
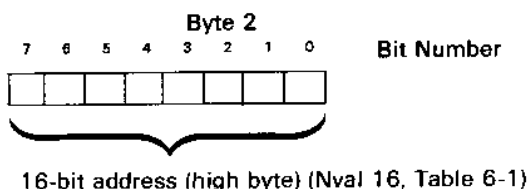
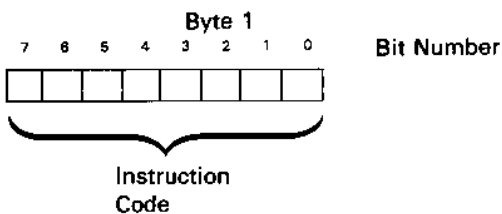
AI, CI, IN, LI, NI, OI, OUT, XI

Two Byte, Type 2



BC, BF, BM, BNC, BNO, BNZ, BP, BR, BR7, BT, BZ

Three Byte



6.1 ADC - ADD ACCUMULATOR TO DATA COUNTER

The contents of the accumulator are treated as a signed binary number, and are added to the contents of every DCO register. The result is stored in the DCO registers. The accumulator contents do not change.

FORMAT:

[LABEL] ADC

STATUS CONDITIONS:

No status bits are modified.

EXAMPLES:

Suppose the accumulator contains H'3E' and every DCO register contains H'209A'. After execution of the ADC instruction, every DCO register will contain H'20D8':

```

209A
  3E
----
H'20D8'
    
```

Suppose the accumulator contains H'A2' and every DCO register contains H'213E'. In two's complement notation, H'A2' is a negative number, since the high order bit of the byte is 1:

H'A2' = 1 0 1 0 0 0 1 0

Sign Bit = 1,
Value negative

Accordingly, after execution of the ADC instruction, every DCO register will contain H'20E0'C

```

213E
  FFA2
----
H'20E0'
    
```

See also Sections 7.3.4., 7.5.1, and 9.3.2.

6.2 AI - ADD IMMEDIATE TO ACCUMULATOR

The 8-bit (two hexadecimal digit) value provided by the instruction operand is added to the current contents of the accumulator. Binary addition is performed.

FORMAT:

[LABEL] AI Nval8

Nval8 is defined in Table 6-1

STATUS CONDITIONS:

Statuses modified: OVF, ZERO, CARRY, SIGN
Statuses unaffected: ICB

EXAMPLE:

Suppose the accumulator contains H'3F'. After execution of the instruction:

AI H'7E'

the accumulator will contain H'BD':

| | | | | | | | | | |
|---------|---|---|---|---|---|---|---|---|---|
| Bit No. | C | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| H'3F' = | | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| H'7E' = | | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| H'BD' = | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

There is no carry out of bit 7, so CARRY = 0.
 There is a carry out of bit 6 and no carry out of bit 7,
 therefore OVF = 0 ⊕ 1 = 1.

The result is not zero, so ZERO = 0.
 The high order bit of the result is 1, so SIGN = 0.

See also Sections 8.2.7 and 10.1.3.

6.3 AM - ADD (BINARY) MEMORY TO ACCUMULATOR

The content of the memory location addressed by the DCO registers is added to the accumulator. The sum is returned in the accumulator. Memory is not altered. Binary addition is performed. The contents of the DCO registers are incremented by 1.

FORMAT:

[LABEL] AM

STATUS CONDITIONS:

Statuses modified: OVF, ZERO, CARRY, SIGN
 Statuses unaffected: ICB

EXAMPLE:

Suppose the accumulator contains H'C2', the DCO registers contain H'213E' and memory location H'213E' contains H'2A'. After an AM instruction has been executed, the DCO registers will contain H'213F', and the accumulator will contain H'EC'

| | | | | | | | | | |
|---------|---|---|---|---|---|---|---|---|---|
| Bit No: | C | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| H'C2' = | | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| H'2A' = | | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| H'EC' = | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

There is no carry out of bit 7, so CARRY = 0.
 There is no carry out of bit 6 or bit 7, so OVF = 0 ⊕ 0 = 0.
 The result is not zero, so ZERO = 0.
 The high order bit of the result is 1, so SIGN = 0.

See also Sections 7.2.2, 7.4.2, 10.2.2.

6.4 AMD - DECIMAL ADD, MEMORY TO ACCUMULATOR

The accumulator and the memory location addressed by the DCO registers are assumed to contain two BCD digits. The content of the address memory byte is added to the contents of the accumulator to give a BCD result in the accumulator, providing these steps are followed:

Decimal addition is, in reality, three binary events. Consider 8-bit decimal addition. Assume two BCD digit augend XY is added to two BCD digit addend ZW, to give a BCD result PQ:

$$\begin{array}{r} XY \\ +ZW \\ \hline =PQ \end{array}$$

Two carries are important: any intermediate carry (IC) out of the low order answer digit (Q), and any overall carry (C) out of the high order digit (P). The three binary steps required to perform BCD addition are as follows:

- STEP 1 Binary add H'66' to the augend.
- STEP 2 Binary add the addend to the sum from Step 1. Record the status of the carry (C) and intermediate carry (IC).
- STEP 3 Add a factor to the sum from Step 2, based on the status of C and IC. The factor to be added is given by the following table:

| Status from Step 2 | | Sum to be added |
|--------------------|----|-----------------|
| C | IC | |
| 0 | 0 | H'AA' |
| 0 | 1 | H'A0' |
| 1 | 0 | H'0A' |
| 1 | 1 | H'00' |

In Step 3, any carry from the low order digit to the high order digit is suppressed.

For example, consider 21 + 67 = 88.

21 = 00100001
 67 = 01100111

STEP 1 H'21' 00100001
 + H'66' 01100110
 = H'87' 10000111

STEP 2 H'87' 10000111
 + H'67' 01100111
 = H'EE' 11101110

C = 0 IC = 0

STEP 3 H'EE' 11101110
 + H'AA' 10101010
 = H'88' 10001000

Carry suppressed

DECIMAL ADD:

A decimal add is accomplished by executing a binary addition of H'66' to one of the two BCD numbers, then executing the AMD instruction, as follows:

AI H'66' Always precedes AMD for addition
 [LABEL] AMD

DECIMAL SUBTRACT:

Assume scratchpad byte 0 contains 1, the accumulator contains the subtrahend and DCO addresses the minuend. Decimal subtraction is performed as follows:

```

COM      ONES COMPLEMENT SUBTRAHEND
AMD      DECIMAL ADD MINUEND
AI       H'66'
ASD     0      DECIMAL ADD 1 TO SUM

```

STATUS CONDITIONS:

Statuses modified: CARRY, ZERO
 Statuses not significant: OVF, SIGN
 Statuses unaffected: ICB

EXAMPLES:

DECIMAL ADD:

Assume the accumulator contains H'57', the DCO registers contain H'12FA' and memory location H'12FA' contains H'60'. After the execution of:

```

AI       H'66'
AMD

```

the accumulator will contain H'17', and the DCO registers will contain H'12FB'.

There is a carry, so CARRY=1. This carry indicates that the result of the addition exceeded 99; therefore the carry must be added to the next high order digit.

Other status indicators are modified, but their condition is not significant.

DECIMAL SUBTRACT:

Assume the accumulator contains H'79', the DCO registers contain H'32A7', memory location H'32A7' contains H'80' and scratchpad byte 0 contains H'01'.

After executing:

```

COM
AMD
AI       H'66'
ASD     0

```

the accumulator contains H'01'.

There is no carry, so CARRY = 0. No Borrow was required.

Status indicators other than carry are modified, but their condition is not significant.

6.5 AS - BINARY ADDITION, SCRATCHPAD MEMORY TO ACCUMULATOR

The content of the scratchpad register referenced by the instruction operand (Sreg) is added to the accumulator using binary addition. The result of the binary addition is stored in the accumulator. The scratchpad register contents remain unchanged. Depending on the value of Sreg, ISAR may be unaltered, incremented or decremented.

FORMAT:

[LABEL] AS Sreg

Sreg is defined in Table 6-2.

STATUS CONDITIONS:

Statuses modified: OVF, ZERO, CARRY, SIGN
 Statuses unaffected: ICB

EXAMPLE:

Suppose the accumulator contains H'34' and scratchpad register 11 contains H'72'. After the instruction:

```
AS 11
```

is executed, the accumulator will now contain H'A6':

```

Bit No:  C 7 6 5 4 3 2 1 0
H'34' =   0 0 1 1 0 1 0 0
H'72' =   0 1 1 1 0 0 1 0
H'A6' =   0 1 0 1 0 0 1 1 0

```

There is no carry out of bit 7, so CARRY = 0.

There is a carry out of bit 6, but not out of bit 7, so OVF = 0 \oplus 1 = 1.

The result is non-zero, so ZERO = 0.

The high order bit of the result is 1, so SIGN = 0.

Suppose the accumulator contains H'7E', ISAR contains O'27' and scratchpad register 23 (=O'27') contains H'A2'. After the instruction:

```
AS D
```

is executed, the accumulator will contain H'20', and ISAR will increment (low order octal digit only) to O'26':

```

Bit No:  C 7 6 5 4 3 2 1 0
H'7E' =   0 1 1 1 1 1 1 0
H'A2' =   1 0 1 0 0 0 1 0
H'20' =   1 0 0 1 0 0 0 0

```

There is a carry out of bit 7, so CARRY = 1.

There is a carry out of bit 6 and bit 7, so OVF = 1 \oplus 1 = 0.

The result is non-zero, so ZERO = 0.

The high order bit of the result is 0, so SIGN = 1.

Had the AS instruction operand been I, ISAR contents would have been decremented to O'20'; had the AS instruction operand been S, ISAR contents would have remained unchanged.

See also Sections 7.1.2, 7.1.4, and 7.2.2.

6.6 ASD - DECIMAL ADD, SCRATCHPAD TO ACCUMULATOR

The ASD instruction is similar to the AMD instruction, except that instead of adding the contents of the memory byte addressed by the DCO registers, the content of the scratchpad byte addressed by operand (Sreg) is added to the accumulator.

FORMAT:

DECIMAL ADD:

```
AI       H'66'  ALWAYS PRECEDES ASD FOR ADDITION
```

[LABEL] ASD Sreg

Sreg is defined in Table 6-2.

DECIMAL SUBTRACT:

COM ALWAYS PRECEDES ASD FOR SUBTRACTION

[LABEL] ASD Sreg
AI H'66'
ASD ONE SCRATCHPAD BYTE ONE CONTAINS H'01'

STATUS CONDITIONS:

The status bits have the same significance as they do for the AMD instruction.

EXAMPLES:

DECIMAL ADD:

Assume the accumulator contains H'42', the ISAR contains O'54', and scratchpad register O'54' contains H'83'.

After the instruction sequence:

AI H'66'
ASD D

is executed, the accumulator will contain H'25'. ISAR will contain O'53'.

There is a carry, so CARRY = 1.

Other status indicators are modified, but their condition is not significant.

6.7 BRANCH INSTRUCTIONS

The Branch instruction is used to modify a program's instruction execution sequence by altering the contents of the program counters, PC0. In a conditional branch instruction, alteration occurs when specified branch test conditions are met. In an unconditional branch instruction, a branch occurs simply as the result of the execution of the instruction.

All branch instructions are two-byte instructions. The first byte is the object code of the instruction mnemonic. The second byte is a displacement which is added to the program counter if a branch occurs.

Conditional branch mnemonics: BC, BF, BM, BNC, BNO, BNZ, BP, BR7, BT, BZ

Unconditional branch mnemonics: BR

FORMATS:

[LABEL] OP DEST

OP is one of the mnemonics BC, BM, BNC, BNO, BNZ, BP, BR7 or BZ.

DEST is an expression which evaluates to the memory address to which a branch may occur. Frequently DEST labels the instruction to which a branch may occur.

[LABEL] OP t,DEST

OP is one of the mnemonics BF or BT.

t is a condition specification, as given in Table 6-5 for BT, or in Table 6-4 for BF.

DEST is as described above.

Relative branching is performed within a range of 127 address locations forward and 128 address locations behind the address of the branch instruction's second byte.

All branch instructions are similar in operation, the only difference is the conditions under which a branch occurs. The instruction BC - BRANCH ON CARRY will be used as an example of how the branch instructions are executed.

When a BC instruction is executed a branch occurs to the instruction whose label is specified in BC instruction operand, but only if the Carry bit is set at the time the BC instruction is executed.

First, consider a BRANCH FORWARD as indicated in the following instruction sequence:

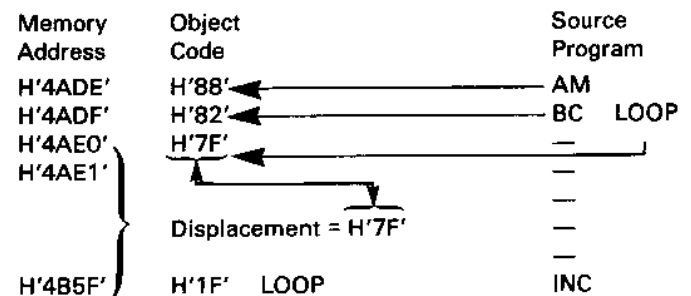


Fig. 6-1. Generation of a Displacement Object Program Byte in Response to a Forward Branch

Figure 6-1 illustrates source and consequent object program.

Assume the Carry bit is set as a result of the AM instruction execution and the contents of the program counters, PC0, are equal to H'4AE0', subsequent to the BC instruction operand fetch. A branch to H'4B5F' is indicated by the BC instruction as follows:

The displacement vector between H'4B5F' and H'4AE0' must be added to the program counters. This vector (+D'127') will have been calculated by the assembler and stored in the second byte of the BC instructions object code.

When a single byte displacement vector is added to the contents of the program counters, the most significant bit of the single byte displacement vector is propagated through the high order eight bits of the addition as follows:

| Bit No: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| H'4AE0' | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| H'7F' | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| H'4B5F' | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

Next, consider a BRANCH BACKWARD as indicated in the following instruction sequence:

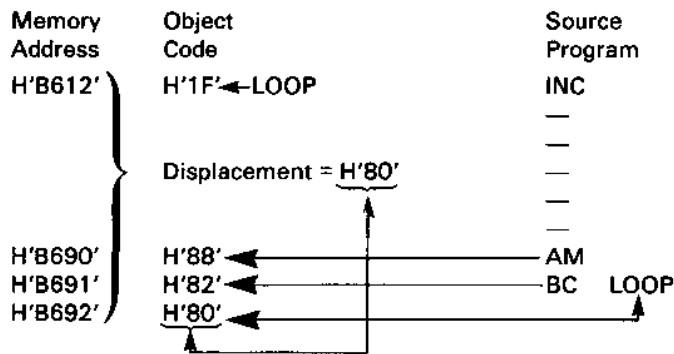


Fig. 6-2. Generation of a Displacement Object Program Byte in Response to a Backward Branch

Assume the carry bit is set and the program counters contain H'B692', subsequent to the BC instruction operand fetch. A branch to H'B612' is indicated by the BC instruction as follows:

The displacement vector between the address of the second byte of the BC instruction and the address of the instruction labeled LOOP is added to the PCO registers. The displacement vector will have been calculated by the assembler and stored in the second byte of the BC instruction object program. In the case of a BRANCH BACKWARD, the negative displacement will be a two's complement number. Since the high order (sign) bit of the displacement is 1, it will be propagated through the high order eight bits of the addition as follows:

| | | | | | | | | | | | | | | | | |
|---------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| Bit No: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| H'B692' | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| H'80' | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| H'B612' | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Table 6-3 lists the branch instruction mnemonics and the conditions under which a branch will occur.

| INSTRUCTION MNEMONIC | BRANCH WILL OCCUR IF | EXAMPLE IN SECTION |
|-----------------------------|---|---------------------|
| BC - BRANCH ON CARRY | Carry bit is set | 10.2.1 |
| BF - BRANCH ON FALSE | See Table 6-4 | |
| BM - BRANCH ON NEGATIVE | Sign bit is reset | |
| BNC - BRANCH IF NO CARRY | Carry bit is reset | |
| BNO - BRANCH IF NO OVERFLOW | OVF bit is reset | 7.1.4, 7.3.3, 7.3.5 |
| BNZ - BRANCH IF NOT ZERO | Zero bit is reset | 7.1.3, 7.2.1, 7.2.2 |
| BP - BRANCH IF POSITIVE | Sign bit is set | 7.3.4, 8.1.1, 8.1.3 |
| BR - UNCONDITIONAL BRANCH | Always | 7.1.4, 7.2.2, 7.3.4 |
| BR7 - BRANCH ON ISAR | Any of the low 3 bits of ISAR are reset | 7.1.1, 7.1.2, 8.2.7 |
| BT - BRANCH ON TRUE | See Table 6-5 | |
| BZ - BRANCH ON ZERO | Zero bit is set | 7.2.1, 7.2.2, 7.3.4 |

Table 6-3. Branch Conditions

| OPERAND | STATUS FLAGS TESTED | | | | DEFINITION | COMMENTS |
|---------|---------------------|------|-------|------|--|-------------|
| | OVF | ZERO | CARRY | SIGN | | |
| 0 | 0 | 0 | 0 | 0 | Unconditional Branch relative | |
| 1 | 0 | 0 | 0 | 1 | Branch on negative | Same as BM |
| 2 | 0 | 0 | 1 | 0 | Branch if no carry | Same as BNC |
| 3 | 0 | 0 | 1 | 1 | Branch if no carry and negative | |
| 4 | 0 | 1 | 0 | 0 | Branch if not zero | Same as BNZ |
| 5 | 0 | 1 | 0 | 1 | | Same as t=1 |
| 6 | 0 | 1 | 1 | 0 | Branch if no carry and result is no zero | |
| 7 | 0 | 1 | 1 | 1 | | Same as t=3 |
| 8 | 1 | 0 | 0 | 0 | Branch if there is no overflow | Same as BNO |
| 9 | 1 | 0 | 0 | 1 | Branch if negative and no overflow | |
| A | 1 | 0 | 1 | 0 | Branch if no overflow and no carry | |
| B | 1 | 0 | 1 | 1 | Branch if no overflow, no carry & negative | |
| C | 1 | 1 | 0 | 0 | Branch if no overflow and not zero | |
| D | 1 | 1 | 0 | 1 | | Same as t=9 |
| E | 1 | 1 | 1 | 0 | Branch if no overflow, no carry & not zero | |
| F | 1 | 1 | 1 | 1 | | Same as t=B |

Table 6-4. Branch Conditions for BF Instruction

| OPERAND t | STATUS FLAGS TESTED | | | DEFINITION | COMMENTS |
|--------------|---------------------|-------|------|--------------------------------|----------------------------|
| | ZERO | CARRY | SIGN | | |
| 0 | 0 | 0 | 0 | Do not branch | An effective 3 cycle NO-OP |
| 1 | 0 | 0 | 1 | Branch if Positive | Same as BP |
| 2 | 0 | 1 | 0 | Branch on Carry | Same as BC |
| 3 | 0 | 1 | 1 | Branch if Positive or on Carry | |
| 4 | 1 | 0 | 0 | Branch if Zero | Same as BZ |
| 5 | 1 | 0 | 1 | Branch if Positive | Same as t=1 |
| 6 | 1 | 1 | 0 | Branch if Zero or on Carry | |
| 7 | 1 | 1 | 1 | Branch if Positive or on Carry | Same as t=3 |

Table 6-5. Branch Conditions for BT Instruction

6.7.1 BF – Branch on False

The BF - BRANCH ON FALSE instruction will branch if the status bits selected by t in Table 6-4 are all reset. Selected bits are identified in Table 6-4 by 1 under "Status Flags Tested"; selected status bits must all be zero. Unselected status bits are ignored.

6.7.2 BT – Branch on True

The BT - BRANCH ON TRUE instructions will branch if any test conditions defined by t in Table 6-5 are met.

6.8 CI - COMPARE IMMEDIATE

The contents of the accumulator are subtracted from the operand of the CI instruction. The result is not saved but the status bits are set or reset to reflect the results of the operation.

FORMAT:

[LABEL] CI Nval8

Nval8 is defined in Table 6-1.

STATUS CONDITIONS:

Statuses modified: OVF, ZERO, CARRY, SIGN

Statuses unaffected: ICB

EXAMPLE:

Assume the accumulator contains H'1B' and the second byte of the instruction contains H'D8'. The comparison is made as follows:

| | | | | | | | | | |
|-------------|---|---|---|---|---|---|---|---|---|
| Bit No: | C | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| H'1B' | | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| two's comp: | | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| H'D8' | | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| H'BO' | | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

The H'BO' result is not saved.

There is a carry out of bit 7, so CARRY = 1.

There is also a carry out of bit 6, so OVF = 1 \oplus 1 = 0.

The result is not zero, so ZERO = 0.

The high order bit is 1, so SIGN = 0.

See also Sections 7.3.4, 8.2.7, 8.3.3.

6.9 CLR - CLEAR ACCUMULATOR

The contents of the accumulator are set to zero.

FORMAT:

[LABEL] CLR

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the accumulator contains H'AO'. After the CLR instruction has executed, the accumulator contains H'00'.

See also Sections 7.1.1, 7.3.5, and 4.3.3.

6.10 CM - COMPARE MEMORY TO ACCUMULATOR

The CM instruction is the same as the CI instruction except the memory contents addressed by the DCO registers, instead of an immediate value, are compared to the contents of the accumulator.

Memory contents are not altered. Contents of the DCO registers are incremented.

FORMAT:

[LABEL] CM

See also Section 9.3.3.

6.11 COM - COMPLEMENT

The accumulator is loaded with its one's complement.

FORMAT:

[LABEL] COM

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN

Statuses reset: OVF, CARRY

Status unaffected: ICB

EXAMPLE:

If the accumulator contains H'8B', after the COM instruction is executed, it will contain H'74'.

The Zero bit is reset to 0 since the result is not zero.

The Sign bit is set to 1 since the high order bit of the result is 0.

The OVF and Carry bits are unconditionally reset to 0.

See also Sections 7.1.2, 7.2.2, and 7.4.2.

6.12 DCI - LOAD DC IMMEDIATE

The DCI instruction is a three-byte instruction. The contents of the second byte replace the high order byte of the DCO registers; the contents of the third byte replace the low order byte of the DCO registers.

FORMAT:

[LABEL] DCI Nval16

Nval16 is defined in Table 6-1.

STATUS CONDITIONS:

The status bits are not affected.

EXAMPLE:

After the instruction:

DCI H'2317'

is executed, the DC registers will contain H'2317'.

See also Sections 7.2.1, 7.2.2, 7.4.1.

6.13 DI - DISABLE INTERRUPT

The interrupt control bit, ICB, is reset; no interrupt requests will be acknowledged by the 3850 CPU.

FORMAT:

[LABEL] DI

STATUS CONDITION:S:

Statuses reset: ICB

Statuses unaffected: OVF, ZERO, CARRY, SIGN

6.14 DS - DECREMENT SCRATCHPAD CONTENT

The content of the scratchpad register addressed by the operand (Sreg) is decremented by one binary count. The decrement is performed by adding H'FF' to the scratchpad register.

FORMAT:

[LABEL] DS Sreg

Sreg is defined in Table 6-2.

STATUS CONDITIONS:

Statuses modified: OVF, ZERO, CARRY, SIGN

Statuses unaffected: ICB

EXAMPLE:

Assume the ISAR contains O'23' and the scratchpad register O'23' contains H'17'. After the instruction:

DS D

is executed, scratchpad register O'23' contains H'16' and the ISAR contains O'22'. The accumulator is unaffected.

There is a carry out from bit 7, so CARRY = 1.

There is a carry out from bit 6, so OVF = $1 \oplus 1 = 0$.

The result of the decrement is non-zero, so ZERO = 0.

The most significant bit is 0, so SIGN = 1.

See also Sections 7.1.3, 7.2.1 and 7.2.2.

6.15 EI - ENABLE INTERRUPT

The interrupt control bit is set. Interrupt requests will now be acknowledged by the CPU.

FORMAT:

[LABEL] EI

STATUS CONDITIONS:

ICB is set to 1.

All other status bits are unaffected.

See also Sections 8.2.7, 8.3.1, and 8.3.3.

6.16 IN - INPUT LONG ADDRESS

The data input to the I/O port specified by the operand of the IN instruction is stored in the accumulator.

The I/O port address assignments are given in Table 6-6. I/O ports with addresses 4 through 255 may be addressed by the IN instruction. I/O ports with port addresses 0 through 15 may be accessed by the INS instruction (see Section 6.17).

The IN instruction generates two bytes of object code, whereas the INS instruction generates one byte of object code.

If an I/O port or pin is being used for both input and output, the port or pin previously used for output must be cleared before it can be used to input data.

| PORT ADDRESS (HEXADECIMAL) | RESERVED FOR 3850 CPU | MAY BE USED BY 3851 PSU | MAY BE USED BY 3852 DMI | MAY BE USED BY 3853 SMI | MAY BE USED BY 3854 DMA |
|-------------------------------|-----------------------------|-------------------------------|-------------------------------|-------------------------------|-------------------------------|
| 00 | | | | | |
| 01 | | | | | |
| 02 | | | | | |
| 03 | | | | | |
| 04 | | | | | |
| . | | | | | |
| . | | | | | |
| 0B | | | | | |
| 0C | | | | | |
| 0D | | (NOTE 1) | (NOTE 4) | | |
| 0E | | | | | |
| 0F | | | | | |
| 10 | | | | | |
| . | | | | | |
| . | | | | | |
| EB | | | | | |
| EC | | | | | |
| ED | | (NOTE 2) | (NOTE 4) | | |
| EE | | | | | |
| EF | | | | | |
| FO | | | | | |
| . | | | | | |
| . | | | | | |
| FF | | (NOTE 3) | | | |

Table 6-6. I/O Port Address Assignments

NOTE 1: These I/O port addresses may not be used by PSU's if a 3852 DMI or 3853 SMI device is used.

NOTE 2: These I/O port addresses may not be used by PSU's if a SL31116 DMI device is used.

NOTE 3: I/O port addresses used by DMA devices may not be used by PSU's.

NOTE 4: Two versions of the 3852 DMI device are available. One uses port assignments H'0C' and H'0D'; the other uses port assignments H'EC' and H'ED'.

FORMAT:

[LABEL] IN Nval8

Nval8 is defined in Table 6-1.

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN

Statuses reset: OVF, CARRY

Statuses unaffected: ICB

EXAMPLE:

Assume that the value H'C8' has been input by an external device to I/O port H'10'. After the instruction:

IN H'10'

is executed, the accumulator will contain H'37'. Note that the data is complemented between I/O pin and accumulator.

The overflow and carry bits are unconditionally reset, so OVF = CARRY = 0.

The accumulator content is non-zero, so ZERO = 0. The most significant bit is zero, so SIGN = 1.

See also Sections 7.6.2 and 8.4.3.

6.17 INC - INCREMENT ACCUMULATOR

The content of the accumulator is increased by one binary count.

FORMAT:

[LABEL] INC

STATUS CONDITIONS:

Statuses modified: OVF, ZERO, CARRY, SIGN

Statuses unaffected: ICB

EXAMPLE:

Assume the accumulator contains H'FF'. After an INC instruction execution, the accumulator contains H'00'.

There is carry out from bit 7, so CARRY = 1.

There is also a carry out from bit 6, so OVF = 1 ⊕ 1 = 0.

The result is zero, so ZERO = 1, and SIGN = 1.

See also Section 8.3.3 and 10.2.2.

6.18 INS - INPUT SHORT ADDRESS

Data input to the I/O port specified by the operand of the INS instruction is loaded into the accumulator. An I/O port with an address within the range 0 through 15 may be accessed by this instruction.

If an I/O port or pin is being used for both input and output, the port or pin previously used for output must be cleared before it can be used to input data.

FORMAT:

[LABEL] INS Nval4

Nval4 is defined in Table 6-1.

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
Statuses reset: OVF, CARRY
Statuses unaffected: ICB

EXAMPLE:

Assume that the 3850 CPU I/O port addressed by H'01' contains H'79'. Execution of the instruction:

```
INS 1
```

causes the accumulator to be loaded with H'86'.

The overflow and carry bits are reset, so OVF = CARRY = 0. The accumulator content is non-zero, so ZERO = 0. The most significant bit is 1, so SIGN = 0.

6.19 JMP - BRANCH IMMEDIATE

As the result of a JMP instruction execution, a branch to the memory location addressed by the second and third bytes of the instruction occurs. The second byte contains the high order eight bits of the memory address; the third byte contains the low order eight bits of the memory address.

The accumulator is used to temporarily store the most significant byte of the memory address; therefore, after the JMP instruction is executed, the initial contents of the accumulator are lost.

FORMAT:

[LABEL] JMP Nval16

STATUS CONDITIONS:

No status bits are affected.

EXAMPLE:

Assume the operand of the JMP instruction contains H'03A6'. After the instruction:

```
JMP H'03A4'
```

is executed, the next instruction will execute from address H'03A4'. At the completion of this execution, the accumulator contains H'03'.

See also Section 7.3.4 and 7.5.1.

6.20 LI - LOAD IMMEDIATE

The value provided by the operand of the LI instruction is loaded into the accumulator.

FORMAT:

[LABEL] LI Nval18

STATUS CONDITIONS:

No status bits are affected.

EXAMPLE:

Assume the second byte of the LI instruction contains H'C7'. The instruction:

```
LI H'C7'
```

causes the accumulator to be loaded with H'C7'.

See also Section 7.1.3, 7.2.1, and 7.2.2.

6.21 LIS - LOAD IMMEDIATE SHORT

A 4-bit value provided by the LIS instruction operand is loaded into the four least significant bits of the accumulator. The most significant four bits of the accumulator are set to "0".

FORMAT:

[LABEL] LIS Nval4

Nval4 is defined in Table 6-1.

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

After the instruction:

```
LIS 3
```

has executed, the accumulator will contain H'03'.

See also Section 7.2.2, 7.3.4 and 9.3.2.

6.22 LISL - LOAD LOWER OCTAL DIGIT OF ISAR

A 3-bit value provided by the LISL instruction operand is loaded into the three least significant bits of the ISAR. The three most significant bits of the ISAR are not altered.

FORMAT:

[LABEL] LISL Nval3

Nval3 is defined in Table 6-1.

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Suppose ISAR contains the value O'72'. After the instruction:
LISL 6

has executed, ISAR will contain the value O'76'.

See also Section 7.1.1, 7.1.2 and 8.2.7.

6.23 LISU - LOAD UPPER OCTAL DIGIT OF ISAR

A 3-bit value provided by the LISU instruction operand is loaded into the three most significant bits of the ISAR. The three least significant bits of the ISAR are not altered.

FORMAT:

[LABEL] Nval3

Nval3 is defined in Table 6-1.

STATUS CONDITIONS:

No status bits are affected.

EXAMPLE:

Suppose ISAR contains the value O'72'. After the instruction:

LISU 3

has executed, ISAR will contain the value O'32'.

See also Section 7.1.1, 7.1.2, and 8.2.7.

6.24 LM - LOAD ACCUMULATOR FROM MEMORY

The contents of the memory byte addressed by the DCO registers are loaded into the accumulator. The contents of the DCO registers are incremented as a result of the LM instruction execution.

FORMAT:

[LABEL] LM

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the DCO registers contain H'37A2' and the memory location addressed by H'37A2' contains H'2B'. Execution of the LM instruction causes the accumulator to be loaded with H'2B'. The DCO registers subsequently will contain H'37A3'.

6.25 LNK - LINK CARRY TO THE ACCUMULATOR

The carry bit is binary added to the least significant bit of the accumulator. The result is stored in the accumulator.

FORMAT:

[LABEL] LNK

STATUS CONDITIONS:

Statuses modified: OVF, ZERO, CARRY, SIGN

Statuses unaffected: ICB

EXAMPLE:

Assume the accumulator contains H'84', and the CARRY bit is set. The instruction execution causes the accumulator to contain H'85'.

As a result of the instruction execution, there is no carry out of bit 7, so CARRY = 0.

There is also no carry out of bit 6, so OVF = 0 ⊕ 0 = 0.

The result is non-zero, so ZERO = 0.

The most significant bit of the result is 1, so SIGN = 0.

See also Section 7.1.2, 7.1.4 and 7.2.2.

6.26 LR - LOAD REGISTER

The LR group of instructions move one or two bytes of data between a source and destination register. Instructions exist to move data between the following registers:

- a) A scratchpad register and the Accumulator
- b) Scratchpad registers and the Data Counter, DCO
- c) The Accumulator and the ISAR
- d) Scratchpad register 9 and the status register
- e) Scratchpad registers and Program Counter, PC0
- f) Scratchpad registers and stack register, PC1

An LR instruction's data source and destination is determined by the instruction operands as illustrated in Table 6-7. The number of data bytes moved (one or two) depends on the size of the source and destination registers (8 or 16 bits).

FORMAT:

[LABEL] LR D,S

S is the source register.

D is the destination register.

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the ISAR contains O'76'. After the instruction:

LR A,IS

is executed, the accumulator contains O'76'. Scratchpad register O'76' remains unchanged. ISAR also remains unchanged.

6.27 NI - AND IMMEDIATE

An 8-bit value provided by the operand of the NI instruction is ANDed with the contents of the accumulator. The results are stored in the accumulator.

FORMAT:

[LABEL] NI Nval8

| LR INSTRUCTION OPERANDS DESTINATION SOURCE | | LOADS REGISTER | FROM REGISTER | WITH | EXAMPLE GIVEN IN SECTION |
|--|---------|----------------------------|----------------------------|---|--------------------------|
| A, | KU | Accumulator | Scratchpad register 12 | 8-bit contents | 7.3.4, 7.3.6 |
| A, | KL | Accumulator | Scratchpad register 13 | 8-bit contents | 7.3.4, 7.3.6 |
| A, | QU | Accumulator | Scratchpad register 14 | 8-bit contents | |
| A, | QL | Accumulator | Scratchpad register 15 | 8-bit contents | |
| KU, | A | Scratchpad register 12 | Accumulator | 8-bit contents | 8.4.3 |
| KL, | A | Scratchpad register 13 | Accumulator | 8-bit contents | 8.4.3 |
| QU, | A | Scratchpad register 14 | Accumulator | 8-bit contents | 7.3.4, 7.3.6 |
| QL, | A | Scratchpad register 15 | Accumulator | 8-bit contents | 7.3.4, 7.3.6 |
| K, | P | Scratchpad register 12 | Program Counter PC1 | High order 8-bit byte | 7.3.4, 7.3, 7.3.6 |
| | | Scratchpad register 13 | Program Counter PC1 | Low order 8-bit byte | |
| P, | K | High order byte of PC1 | Scratchpad register 12 | 8-bit contents | 8.4.3 |
| | | Low order byte of PC1 | Scratchpad register 13 | 8-bit contents | |
| A, | IS | Accumulator | ISAR | 00XXXXXX X's are contents of ISAR | 7.3.4, 8.2.7 |
| IS, | A | ISAR | Accumulator | Low order 6-bits. | 7.3.4, 7.3.5, 8.2.7 |
| PO, | Q | High order byte of PC0 | Scratchpad register 14 | 8-bit contents | 7.3, 4.7.5 |
| | | Low order byte of PC0 | Scratchpad register 15 | 8-bit contents | |
| Q, | DC | Scratchpad register 14 | Data counter registers DCO | High order byte | 7.2.2, 7.3.6, 7.4.2 |
| | | Scratchpad register 15 | Data counter registers DCO | Low order byte | |
| DC, | Q | High order byte DCO | Scratchpad register 14 | 8-bit contents | |
| | | Low order byte DCO | Scratchpad register 15 | 8-bit contents | 7.2.2, 7.3.3, 7.3.4 |
| DC, | H | High order byte of DCO | Scratchpad register 10 | 8-bit contents | |
| | | Low order byte of DCO | Scratchpad register 11 | 8-bit contents | 7.2.2, 7.3.4, 7.3.6 |
| H, | DC | Scratchpad register 10 | Data counter register | High order byte | |
| | | Scratchpad register 11 | Data counter register | Low order byte | |
| W, | J | Status register (w) | Scratchpad register 9 | Low order 5 bits | 7.1.2, 7.2.2, 7.4.1 |
| J, | W | Scratchpad register 9 | Status register (w) | 000XXXXX X's are contents of status register | 7.1.2, 7.2.2, 7.3.3 |
| A, | (Sreg)* | Accumulator | Scratchpad register (Sreg) | 8-bit contents | 7.1.2, 7.1.4, 7.4.1 |
| (Sreg)* | A | Scratchpad register (Sreg) | Accumulator | 8-bit contents | 7.1.1, 7.1.2, 7.1.3 |

*Sreg is a hexadecimal digit representing a scratchpad register, as defined in Table 6-2.

Table 6-7. LR Instruction Operand Definitions

STATUS CONDITIONS:

Statuses reset to 0: OVF, CARRY
 Statuses modified: ZERO, SIGN
 Statuses unaffected: ICB

EXAMPLE:

Assume the second byte of the NI instruction contains H'36', and the accumulator contains H'2A' as a result of the instruction execution, the accumulator contains H'22'.

| | | | | | | | | |
|---------|---|---|---|---|---|---|---|---|
| Bit No: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| H'36' | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| H'2A' | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| H'22' | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

There is no carry out of bit 7, so CARRY = 0.
 There is no carry out of bit 6, so OVF = 0 ⊕ 0 = 0.
 The result is non-zero, so ZERO = 0.
 The most significant bit is zero, so SIGN = 1.

See also Section 7.1.2, 7.2.2 and 7.3.3.

6.28 NM - LOGICAL AND FROM MEMORY

The content of memory addressed by the data counter registers is ANDed with the content of the accumulator. The results are stored in the accumulator. The contents of the data counter registers are incremented.

FORMAT:

[LABEL] NM

STATUS CONDITIONS:

Statuses reset to 0: OVF, CARRY
 Statuses modified: ZERO, SIGN
 Statuses unaffected: ICB

EXAMPLE:

Assume the data counters contain H'49AC', the memory location addressed by H'49AC' contains H'67' and the accumulator contains H'A9'. After execution of the NM instruction, the accumulator contains H'21', and the data counters contain H'49AD'.

| | |
|---------|-----------------|
| Bit No: | 7 6 5 4 3 2 1 0 |
| H'67' | 0 1 1 0 0 1 1 1 |
| H'A9' | 1 0 1 0 1 0 0 1 |
| H'21' | 0 0 1 0 0 0 0 1 |

Also see Section 7.6.1.

6.29 NOP - NO OPERATION

No function is performed.

FORMAT:

[LABEL] NOP

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the program counters contain H'2700'. After a NOP instruction is executed, the PC0 registers contain 'H2701'.

Also see Section 8.4.3.

6.30 NS - LOGICAL AND FROM SCRATCHPAD MEMORY

The content of the scratchpad register addressed by the operand (Sreg) is ANDed with the content of the accumulator. The results are stored in the accumulator.

FORMAT:

[LABEL] NS Sreg

Sreg is defined in Table 6-2.

STATUS CONDITIONS:

Statuses reset to 0: OVF, CARRY
 Statuses modified: ZERO, SIGN
 Statuses unaffected: ICB

EXAMPLE:

Assume scratchpad register O'02' contains H'F2', and the accumulator contains H'2F'. Execution of the instruction:

NS 2

causes the accumulator to contain H'22'.

| | |
|---------|-----------------|
| Bit No: | 7 6 5 4 3 2 1 0 |
| H'F2' | 1 1 1 1 0 0 1 0 |
| H'2F' | 0 0 1 0 1 1 1 1 |
| H'22' | 0 0 1 0 0 0 1 0 |

There is no carry out of bit 7, so CARRY = 0.
 There is also no carry out of bit 6, so OVF = 0 ⊕ 0 = 0.
 The result is non-zero, so ZERO = 0.
 The most significant bit of the result is zero, so SIGN = 1.

Also see Section 7.6.1 and 7.6.2.

6.31 OI - OR IMMEDIATE

An 8-bit value provided by the operand of the I/O instruction is ORed with the contents of the accumulator. The results are stored in the accumulator.

FORMAT:

[LABEL] OI Nval8

Nval8 is defined in Table 6-1.

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
 Statuses reset: OVF, CARRY
 Statuses unaffected: ICB

EXAMPLE:

Assume the accumulator contains H'0A'. The execution of the instruction:

OI H'A3'

causes the accumulator to contain H'AB'.

| | |
|---------|-----------------|
| Bit No: | 7 6 5 4 3 2 1 0 |
| H'AB' | 1 0 1 0 0 0 1 1 |
| H'0A' | 0 0 0 0 1 0 1 0 |
| H'A3' | 1 0 1 0 1 0 1 1 |

The accumulator result is non-zero, so ZERO = 0.
 The most significant bit of the result is 1, so SIGN = 0.
 The overflow and carry bits are reset, so OVF = 0 and CARRY = 0.

Also see Section 7.6.1.

6.32 OM - LOGICAL "OR" FROM MEMORY

The content of memory byte addressed by the data counter registers is ORed with the content of the accumulator. The results are stored in the accumulator. The data counter registers are incremented.

FORMAT:

[LABEL] OM

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
 Statuses reset: OVF, CARRY
 Statuses unaffected: ICB

EXAMPLE:

Assume the DC registers contain H'FC19', the memory location addressed by H'FC19' contains H'16', and the accumulator contains H'81'. After execution of an OM instruction, the accumulator contains H'97' and the DC registers will contain H'FC1A'.

| | |
|---------|-----------------|
| Bit No: | 7 6 5 4 3 2 1 0 |
| H'16' | 0 0 0 1 0 1 1 0 |
| H'81' | 1 0 0 0 0 0 0 1 |
| H'97' | 1 0 0 1 0 1 1 1 |

The result is non-zero, so ZERO = 0.

The most significant bit of the result is 1, so SIGN = 0.

The overflow and carry bits are unconditionally reset, so OVF = 0 and CARRY = 0.

6.33 OUT - OUTPUT LONG ADDRESS

The I/O port addressed by the operand of the OUT instruction is loaded with the contents of the accumulator.

I/O ports with addresses from 4 through 255 may be accessed with the OUT instruction.

The OUT instruction generates two bytes of object code, whereas the OUTS instruction generates one byte of object code.

The I/O port addresses are defined in Table 6-6.

FORMAT:

[LABEL] OUT Nval8

Nval8 is defined in Table 6-1.

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the accumulator contains H'2A'. Execution of the instruction:

```
OUT H'F6'
```

will cause the I/O port H'F6' to be loaded with H'D5'. Note that the data at the I/O pins is complemented with respect to the accumulator.

6.34 OUTS - OUTPUT SHORT ADDRESS

The I/O port addressed by the operand of the OUTS instruction object code is loaded with the contents of the accumulator. I/O ports with addresses from 0 to 15 may be accessed by this instruction. The I/O port addresses are defined in Table 6-6. Outs 0 or 1 is CPU port only.

FORMAT:

[LABEL] OUTS Nval4

Nval4 is defined in Table 6-1.

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the OUTS instruction operand (Nval4) is H'0F', and the accumulator contains H'32'. Execution of the instruction:

OUTS 15

will cause the I/O port H'0F' to contain H'CD'.

Also see Section 7.2.1, 8.1.1 and 8.1.3.

6.35 PI - CALL TO SUBROUTINE IMMEDIATE

The contents of the Program Counters are stored in the Stack Registers, PC1, then the 16-bit address contained in the operand of the PI instruction is loaded into the Program Counters. The accumulator is used as a temporary storage register during transfer of the most significant byte of the address. Previous accumulator results will be altered.

FORMAT:

[LABEL] PI Nval16

Nval16 is defined in Table 6-2.

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume that the operand of the PI instruction contains H'32A1', the program counter (PC0) registers contain H'ABCD', and the Stack registers (PC1) contain H'1234'. Execution of the instruction:

```
PI H'32A1'
```

causes the Stack registers (PC1) to contain H'ABCD', and the program counter registers (PC0) to contain H'32A1'.

Also see Section 7.3.3, 7.3.5 and 8.1.1.

6.36 PK - CALL TO SUBROUTINE DIRECT AND RETURN FROM SUBROUTINE DIRECT

The contents of the Program Counter Registers (PC0) are stored in the Stack Registers (PC1), then the contents of the Scratchpad K Registers (Registers 12 and 13 of scratchpad memory) are transferred into the Program Counter Registers.

FORMAT:

[LABEL] PK

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume Scratchpad Register 12 contains H'AB', Scratchpad Register 13 contains H'CD', and the Program Counter Registers (PC0) contain H'1234'. Execution of the instruction PK causes the Stack Registers to contain H'1234' and the Program Counter Registers to contain H'ABCD'.

Also see Sections 7.3.3, 7.4.1 and 8.2.7.

6.37 POP - RETURN FROM SUBROUTINE

The contents of the Stack Registers (PC1) are transferred to the Program Counter Registers (PC0).

FORMAT:

[LABEL] POP

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the Stack Registers (PC1) contain H'ABCD' and the Program Counter Registers (PC0) contain H'1234'. When the POP instruction has been executed, the PC0 registers will contain H'ABCD' and PC1 will not be changed.

Also see Sections 7.3.3, 7.3.4 and 8.2.7.

6.38 SL - SHIFT LEFT

The contents of the accumulator are shifted left either one or four bit positions, depending upon the value of the SL instruction operand.

If the value of the operand is 1, the accumulator contents are shifted left one bit position. The least significant bit becomes a zero.

If the value of the operand is 4, the accumulator contents are shifted left four bit positions. The four least significant bits are filled with zeroes.

FORMAT:

[LABEL] SL Nval4

Nval4 = 1 or 4

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
Statuses reset: OVF, CARRY
Statuses unaffected: ICB

EXAMPLE:

Assume the accumulator contains H'81'. The execution of the instruction:

SL 1

causes the accumulator to contain H'02'. Execution of the instruction:

SL 4

causes the accumulator to contain H'10'.

In both examples the result is non-zero, so ZERO = 0. The most significant bit of the results is zero, so SIGN = 1. The overflow and carry bits are unconditionally reset, so OVF and CARRY = 0.

Also see Sections 8.4.3, 8.3.2 and 10.3.

6.39 SR - SHIFT RIGHT

The contents of the accumulator are shifted right either one or four bit positions, depending on the value of the SR instruction operand.

If the value of the operand is 1, the accumulator contents are shifted right one bit position. The most significant bit becomes a zero.

If the value of the operand is 4, the accumulator contents are shifted right four bit positions. The four most significant bits are filled with zeroes.

FORMAT:

[LABEL] SR Nval4

Nval4 = 1 or 4

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
Statuses reset: OVF, CARRY
Statuses unaffected: ICB

EXAMPLE:

Assume the accumulator contains H'81'. Execution of the instruction:

SR 1

causes the accumulator to contain H'40'. Execution of the instruction:

SR 4

causes the accumulator to contain H'08'.

In both examples the result is non-zero, so ZERO = 0. The most significant bit of the results is zero, so SIGN = 1. The overflow and carry bits are unconditionally reset, so OVF and CARRY = 0.

Also see Sections 10.1.2 and 10.3.

6.40 ST - STORE TO MEMORY

The contents of the accumulator are stored in the memory location addressed by the Data Counter (DC0) registers.

The DC registers' contents are incremented as a result of the instruction execution.

FORMAT:

[LABEL] ST

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the accumulator contains H'69', and the DC0 registers contain H'ABBE'. Execution of the instruction ST causes the memory location H'ABBE' to contain H'69'; DC0 is incremented to contain H'ABBF'.

See also Sections 7.2.2, 7.3.4 and 7.4.2.

6.41 XDC - EXCHANGE DATA COUNTERS

Execution of the instruction XDC causes the contents of the auxiliary data counter registers (DC1) to be exchanged with the contents of the data counter registers (DC0).

This instruction is only significant when a 3852 or 3853 Memory Interface device is part of the system configuration.

FORMAT:

[LABEL] XDC

STATUS CONDITIONS:

No status bits are modified.

EXAMPLE:

Assume the data counters, DC0, contain H'ABCD', and the auxiliary data counter registers, DC1, contain H'1234'. Execution of the instruction XDC causes the DC0 registers to contain H'1234', and the DC1 registers to contain H'ABCD'. The PSU's will have DC0 unaltered.

Also see Sections 7.2.2, 7.4.2 and 7.6.1.

6.42 XI - EXCLUSIVE-OR IMMEDIATE

The contents of the 8-bit value provided by the operand of the XI instruction are EXCLUSIVE-ORed with the contents of the accumulator. The results are stored in the accumulator.

FORMAT:

[LABEL] XI Nval8

Nval8 is defined in Table 6-1.

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
 Statuses reset: OVF, CARRY
 Statuses unaffected: ICB

EXAMPLE:

Assume the accumulator contains H'AB', and the operand of the XI instruction contains H'42'. Execution of the instruction:

XI H'42'

causes the accumulator to contain H'89'.

| | |
|---------|------------------------|
| Bit No: | 7 6 5 4 3 2 1 0 |
| H'AB' | 1 0 1 0 1 0 1 1 |
| H'42' | 0 0 1 0 0 0 1 0 |
| H'89' | <u>1 0 0 0 1 0 0 1</u> |

The result is non-zero, so ZERO = 0.
 The high order bit of the results is one, so SIGN = 0.
 The overflow and carry bit are unconditionally reset, so OVF = 0 and CARRY = 0.

6.43 XM - EXCLUSIVE-OR FROM MEMORY

The content of the memory location addressed by the DC0 registers is EXCLUSIVE-ORed with the contents of the accumulator. The results are stored in the accumulator. The DC0 registers are incremented.

FORMAT:

[LABEL] XM

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
 Statuses reset: OVF, CARRY
 Statuses unaffected: ICB

EXAMPLE:

Assume the DC0 counters contain H'1DE4', the memory location addressed by H'1DE4' contains H'1D', and the accumulator contains H'A8'. Execution of the instruction XM causes the accumulator to contain H'B5'. DC0 is updated to H'1DE5'.

| | |
|---------|------------------------|
| Bit No: | 7 6 5 4 3 2 1 0 |
| H'1D' | 0 0 0 1 1 1 0 1 |
| H'A8' | <u>1 0 1 0 1 0 0 0</u> |
| H'B5' | 1 0 1 1 0 1 0 1 |

The result is non-zero, so ZERO = 0.
 The high order bit of the result is one, so SIGN = 0.
 The overflow and carry bit are unconditionally reset, so OVF = 0 and CARRY = 0.

6.44 XS - EXCLUSIVE-OR FROM SCRATCHPAD

The content of the scratchpad register referenced by the operand (Sreg) is EXCLUSIVE-ORed with the contents of the accumulator.

FORMAT:

[LABEL] XS Sreg

Sreg is defined in Table 6-2.

STATUS CONDITIONS:

Statuses modified: ZERO, SIGN
 Statuses reset: OVF, CARRY
 Statuses unaffected: ICB

EXAMPLE:

Assume the scratchpad register 10 contains H'7C', and the accumulator contains H'61'. Execution of the instruction:

XS 10

causes the accumulator to contain H'1D'.

| | |
|---------|------------------------|
| Bit No. | 7 6 5 4 3 2 1 0 |
| H'7C' | 0 1 1 1 1 1 0 0 |
| H'61' | <u>0 1 1 0 0 0 0 1</u> |
| H'1D' | 0 0 0 1 1 1 0 1 |

The result is non-zero, so ZERO = 0.
 The high order bit of the results is zero, so SIGN = 1.
 Overflow and carry bits are reset, so OVF = 0 and CARRY = 0.

Also see Section 7.1.1 and 7.6.2.

PROGRAMMING TECHNIQUES

This section describes some basic programming techniques that will be useful in almost any F8 application.

NOTE: For easy reading, instructions in examples have labels which are repeated from one example to the next. It is important to understand that in a real program no label can be used more than once.

7.1 MANIPULATING DATA IN THE SCRATCHPAD

The Central Processing Unit's 64 byte scratchpad memory is the principle storage for data and addresses that are currently being accessed by the CPU. Table 7-1 illustrates the scratchpad memory. Notice that since the ISAR register is divided into two 3-bit units, octal numbers are the best suited to scratchpad addressing.

Scratchpad registers 0 through 8 are nine general purpose registers that should be used to store transient data (or addresses) currently being accessed.

Scratchpad registers 9 through 15 are used as temporary depositories for address and status register contents (DCO, PC0, PC1 and W). Special instructions move data between these scratchpad registers and their associated status or address registers.

Registers 16 through 63 are addressed via the ISAR register, and may be visualized as six, 8-byte buffers. The ISAR register can, of course, address any of the 64 scratchpad registers; however, usually only scratchpad registers 16 through 63 (0'20' through 0'77') are accessed via the address in ISAR.

| BYTE NUMBER | | FUNCTION |
|-------------|---------|--|
| Octal | Decimal | |
| 0 - 10 | 0 - 8 | Nine general purpose scratch registers |
| 11 | 9 | Temporary storage for status register |
| 12, 13 | 10, 11 | HU and HL; temporary storage for the Data Counter registers (DCO) |
| 14, 15 | 12, 13 | KU and KL; temporary storage for the stack register (PC1) |
| 16, 17 | 14, 15 | QU and QL; temporary storage for Program Counter (PC0) or Data Counter Registers |
| 20 - 27 | 16 - 23 | First data buffer. ISAR = 0'2X'. |
| 30 - 37 | 24 - 31 | Second data buffer. ISAR = 0'3X'. |
| 40 - 47 | 32 - 39 | Third data buffer. ISAR = 0'4X'. |
| 50 - 57 | 40 - 47 | Fourth data buffer. ISAR = 0'5X'. |
| 60 - 67 | 48 - 55 | Fifth data buffer. ISAR = 0'6X'. |
| 70 - 77 | 56 - 63 | Sixth data buffer. ISAR = 0'7X'. |

X is any octal digit (0 through 7).

Table 7-1. Scratchpad Memory Utilization

7.1.1 Simple Scratchpad Buffer Operations

Because of the way ISAR operates, registers 16 through 63 conveniently form six buffers, each capable of storing eight bytes. As described in Section 2.1, the ISAR is a 6-bit register, divided into 3-bit digits. When ISAR contents are in-

cremented or decremented, only the lower three bits are affected; therefore, once ISAR has been loaded with a scratchpad address, it will only increment or decrement within an 8-byte address range. The end of any 8-byte buffer may be identified using the BR7 instruction.

To illustrate scratchpad buffer manipulation at its most elementary level, consider the following instruction sequence, which sets all eight bytes of a buffer to zero:

```

ONE   CLR           CLEAR THE ACCUMULATOR
TWO   LISU  2       ADDRESS SCRATCHPAD
                        BUFFER 1
THREE LISL  7
LOOP  LR   D,A      CLEAR SCRATCHPAD BYTE AND
                        DECREMENT ISAR
FOUR  BR7  LOOP     RETURN FOR MORE BYTES
    
```

Instructions execute as follows:

ONE: Clear the accumulator, so scratchpad bytes may be cleared by loading 0 into each byte.

TWO: Set the ISAR register to address the first byte
THREE: of scratchpad buffer 1. This address is 0'27'.

LOOP: Load the accumulator content (which is zero) into the scratchpad byte addressed by ISAR (initially byte 0'27'). Because ISAR is identified in the operand via the address D, decrement the low order ISAR octal digit. (After the first execution of LOOP, ISAR contents are decremented to 0'26'; after the second execution of LOOP, ISAR contents are decremented to 0'25', etc.)

FOUR: If ISAR contains 7 as its low order octal digit, continue; otherwise return to LOOP. In this case, continue if ISAR contains 0'27' and return if ISAR contains 0'20' through 0'26'.

7.1.2 Incrementing Up, and Decrementing Down Scratchpad Buffers

Now consider a simple variation of the above example; a 7-byte, positive number, stored in buffer 3, is added to another 7-byte, positive number, stored in buffer 4. Binary addition is performed as follows:

```

COM           INITIALLY CLEAR THE
                CARRY STATUS
ONE   LISL  0       ADDRESS LOW ORDER BYTE
                        OF EACH BUFFER
LOOP  LISU  4       ADDRESS FIRST BUFFER
TWO   LR   A,S      LOAD FIRST BUFFER BYTE
                        INTO A
THREE LISU  5       ADDRESS SECOND BUFFER
FOUR  LNK
FIVE  LR   J,W      ADD ANY CARRY TO A
                        SAVE STATUS IN SCRATCHPAD
                        BYTE 9
SIX   AS   S        ADD SAME BYTE OF SECOND
                        BUFFER
SEVEN LR   I,A      STORE ANSWER AND INCRE-
                        MENT BYTE POINTER
EIGHT LR   A,9      XOR CARRY BIT FROM
                        SCRATCHPAD BYTE 9
    
```

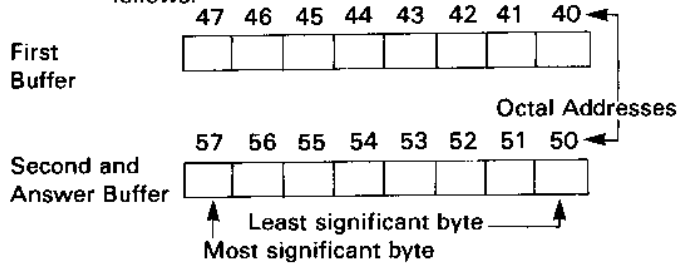
TEN LR J,W WITH CURRENT CARRY BIT
 ELEV XS 9
 TWEL LR 9,A
 THRT LR W,J
 FORT BR7 LOOP RETURN IF NOT END

THRT Return status to W.

Note: instructions EIGHT to THRT can be simplified by replacing with BC FORT
 LR W, J

Instructions in the above example execute as follows:

ONE Set the low order octal digit of ISAR to 0. Assume that numbers are stored in scratchpad buffers as follows:



LOOP Set the high order octal digit of ISAR to 4, thus addressing the next (initially least significant) byte of the first buffer.

TWO Load the next byte of the first buffer. By using S to identify ISAR as addressing the scratchpad, ISAR is not changed. This is important, since ISAR must not be incremented until the sum has been stored in the second buffer.

THREE By loading 5 into the ISAR high order digit, the corresponding byte of the second buffer is addressed.

FOUR Add any carry from the previous byte addition to the accumulator.

FIVE Save the status in J (scratchpad register 9).

SIX Add the second buffer byte (same byte number as first buffer) to the accumulator.

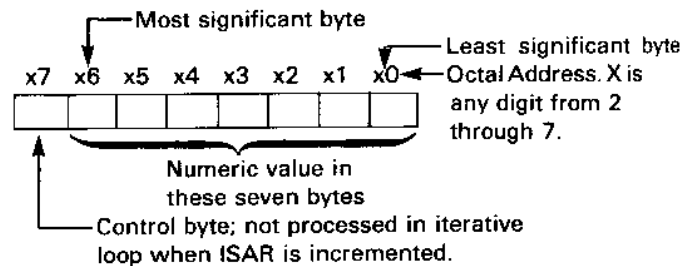
SEVEN Store the sum back into the second buffer, and this time increment the low order octal digit in ISAR, after storing the sum.

EIGHT When any previous carry is added to the accumulator by instruction FOUR, it is possible for 1 to be added to H'FF'. In this case the carry status would be set to 1 and the accumulator will be reset to 0. When the main addition is performed by instruction SIX, the carry status must be reset to 0. As a result the carry from FOUR will be lost. The correct carry to be used in the next byte addition is the OR of any carries from FOUR and SIX; EXCLUSIVE-OR is used since the two carry statuses cannot both be 1. The correct carry is created by instructions EIGHT through THRT, which perform these steps:

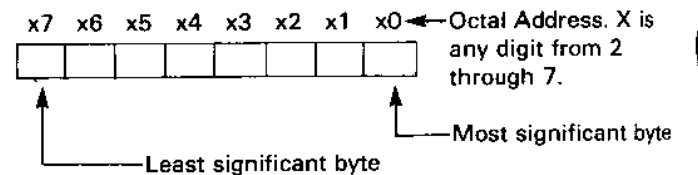
EIGHT Move status from FOUR to the accumulator.
 TEN Move status from addition in instruction SIX to J, register 9 in the scratchpad.
 ELEV EXCLUSIVE-OR J and the accumulator. The carry status can be $0 \oplus 1 = 1$, $0 \oplus 0 = 0$, $1 \oplus 0 = 1$ but never $1 \oplus 1 = 0$.
 TWEL Return status to J.

FORT If ISAR does not address the last byte of the second number buffer, return to LOOP; otherwise continue. (i.e., return to LOOP if ISAR holds O'50' through O'56'; continue if ISAR holds O'57'.)

This multibyte addition illustrates an important feature of scratchpad buffer utilization: increment ISAR if the high order buffer byte has a special significance; decrement otherwise. For example, as illustrated below numbers may use the high order buffer byte to hold sign, decimal point, or any other control information:



Consider now a variation of the multibyte addition, in which the significance of bytes is reversed:



By starting ISAR at x7, all eight bytes of the buffer will be processed identically, since ISAR will be decremented to x6 before the first execution of BR7. Thus the loop will be executed seven times, until ISAR decrements from x0 back to x7. Program steps are as follows:

| | | | |
|-------|------|------|--|
| | COM | | INITIALLY CLEAR THE CARRY STATUS |
| ONE | LISL | 7 | ADDRESS LOW ORDER BYTE OF EACH BUFFER |
| LOOP | LISU | 4 | ADDRESS FIRST BUFFER |
| TWO | LR | A,S | LOAD FIRST BUFFER BYTE INTO A |
| THREE | LISU | 5 | ADDRESS SECOND BUFFER |
| FOUR | LNK | | ADD ANY CARRY TO A |
| FIVE | LR | J,W | SAVE STATUS IN SCRATCHPAD BYTE 9 |
| SIX | AS | S | ADD SAME BYTE OF SECOND BUFFER |
| SEVEN | LR | D,A | STORE ANSWER AND DECREMENT BYTE POINTER |
| EIGHT | LR | A,9 | OR CARRY BIT FROM SCRATCHPAD BYTE 9 |
| TEN | LR | J,W | WITH CURRENT CARRY BIT |
| ELEV | XS | 9 | |
| TWEL | LR | 9,A | |
| THRT | LR | W,J | |
| FORT | BR7 | LOOP | RETURN IF ISAR DID NOT DECREMENT FROM O'50' to O'57' |

Another variation of the same incrementing scratchpad addition is shown below. It takes advantage of the fact that the SUM is overwriting the second data buffer. If the result of the LNK instruction produces a carry the results in the accumulator must be zero; therefore, the sum is already correct and the following addition is needless. The carry bit logic is therefore simplified. This routine is only valid if the sum overwrites one of the buffers.

```

COM
ONE  LISL  0
LOOP LISU  4
TWO  LR   A,S
THREE LISU  5
FOUR  LNK
      BC   CKENK
SIX   AS   S
SEVEN LR   S,A
CKEND LR   A,I      DUMMY INSTRUCTION TO INC
                      ISAR
      BR7  LOOP

```

By changing: ONE LISL 0 to ONE LISL 7
and: CKEND LR A,I to CKEND LR A,D

The ISAR will be decrementing during the addition.

7.1.3 Using Scratchpad Registers as Counters

Scratchpad bytes 0 through 8 should be used for counters and pointers, and for short data operations that do not require data buffers.

Consider the simple use of a scratchpad byte as a counter. If an instruction sequence is to be executed some number of times between 1 and 256, proceed as follows:

```

ONE  LI   COUNT  LOAD COUNT INTO
      —   —      ACCUMULATOR
TWO  LR   0,A    MOVE TO SCRATCHPAD
      —   —      REGISTER 0
LOOP —   —      START OF INSTRUCTION SE-
      —   —      QUENCE TO BE RE-EXECUTED
—
—
—
TEST DS   0      DECREMENT COUNTER
      BNZ  LOOP  RETURN IF COUNTER IS NOT 0

```

COUNT is a symbol which must be equated to a numeric constant between 0 and 255. A value of 0 will cause 256 returns to LOOP, since TEST will decrement the counter to 255 on the first pass.

Note that scratchpad register 0 has been arbitrarily selected as the counter; any other register, up to register 8, could have been used.

7.1.4 Using Scratchpad Registers for Short Data Operations

Data operations that involve 4-byte (or smaller) data units are handled out of the first nine scratchpad registers.

Consider the addition of 16-bit signed binary numbers. Assume that the augend is stored in scratchpad registers 0 and

1 (1 most significant), and the addend is stored in scratchpad registers 2 and 3 (3 most significant). The result is to be returned in registers 2 and 3. Bit 7 of registers 1 and 3 holds the sign of the augend and addend, respectively. The addition program proceeds, directly accessing scratchpad registers:

```

ONE  LR   A,0    LOAD LOW ORDER AUGEND
      —   —      BYTE
TWO  AS   2      ADD ADDEND LOW ORDER
      —   —      BYTE
THREE LR   2,A   SAVE THE RESULT
FOUR  LR   A,1   LOAD THE HIGH ORDER
      —   —      AUGEND BYTE
FIVE  LNK
      —   —      ADD ANY CARRY FROM LOW
      —   —      ORDER BYTE ADD
SIX   BNO   EIGHT IF THERE IS AN OVERFLOW,
SEVN  BR   ERROR THE RESULT IS TOO LARGE.
      —   —      MAKE AN ERROR EXIT
EIGHT AS   3     ADD THE HIGH ORDER
      —   —      ADDEND BYTE
NINE  LR   3,A   SAVE THE RESULT
TEN   BNO   OK   NO OVERFLOW, CONTINUE
ELEV  BR   ERROR OVERFLOW, THE RESULT IS IN
      —   —      ERROR

```

The program executes as follows:

```

ONE  Load the low order augend byte into the accumulator,
TWO  add the low order addend byte and save the result.
THREE Carry is the only meaningful status after this addition.
      If the carry is set, it means that 1 must be added to
      the high order byte result.
FOUR  Load the high order augend byte and add any carry
      to it.
FIVE  Now the overflow status is important, since it identifies
      a carry out of bit 6, the highest order data bit.
      (See Appendix A for clarification.)
SIX   If the overflow status is set, branch out to an error
SEVN  handling program. If the overflow status is not set,
      continue. Only the overflow status need be tested.

```

If two positive numbers are being added, the important carry is out of bit 6, and there can be no carry out of bit 7, which must be 0 for both numbers.

If a positive and a negative number are being added, there can be no overflow.

If two negative numbers are being added, there must be a carry, since both 7 bits are 1. If there is no carry out of bit 6, an erroneous positive result is indicated, and the overflow bit is set.

```

EIGHT Add the high order addend byte and store the result.
NINE
TEN   Repeat of instruction SIX. OK is presumed to be the
      label of the instruction at which normal execution
      continues.

```

7.2 ROM, RAM AND DATA TABLES

There are two circumstances under which ROM and RAM memory outside the 3850 CPU scratchpad will be referenced to access data:

- 1) In large or small F8 systems, data tables may be stored in ROM.
- 2) In large F8 systems, data will be stored and retrieved out of RAM, via 3852 or 3853 interface devices; this allows large amounts of data to be stored and processed.

7.2.1 Reading Data Out of Tables in ROM

Various types of "table lookup" applications make extensive use of data tables stored in ROM, which will usually be a 3851 PSU device. There are two types of table lookup application, the sequential access and the random access.

Consider first text generation as an example of sequential access. Messages are stored, as ASCII character sequences, in ROM. The following instruction sequence outputs a message via the 3850 CPU I/O port 0:

```

MSG1  ORG  H'0600'
      DC  C'PO'
      DC  C'UL'
      DC  C'TR'
      DC  C'YB'
MSG2  DC  C'FI'
      DC  C'SH'
      DC  C'BB'
      —
      —
      —
ONE   ORG  H'0400'
TWO   LI  (MSG2-MSG1) LOAD BUFFER LENGTH
      LR  0,A      SAVE IN SCRATCH
                        REGISTER 0
THREE DCI  MSG1    LOAD STARTING BUFFER
                        ADDRESS INTO DCO
FOUR  CLR
      OUTS 0      INITIALIZE PORT 0
      INS  0
      —
      —
      —
FIVE  BP  FOUR    TEST FOR READY TO
                        RECEIVE DATA
      —
      —
      —
SIX   LM
SEVEN OUTS 0      ASSUME 1 IN BIT 7
EIGHT DS  0      WHEN READY
      —
      —
      —
NINE  BNZ  FOUR   LOAD NEXT CHARACTER
                        OUTPUT CHARACTER
                        DECREMENT CHARACTER
                        COUNTER
                        RETURN FOR MORE
                        CHARACTERS

```

It is arbitrarily assumed that the eight ASCII characters 'POULTRYB' are stored in ROM, starting at memory location H'0600'; the program to output this character string starts at memory location H'0400'. The program proceeds as follows:

ONE The message length is computed by subtracting the
TWO symbol MSG1, which equals the starting address of 'POULTRYB', from the symbol MSG2, which equals the starting address of the next message, 'FISHB'. This message length is stored in scratchpad register 0.

THREE The selected message starting address (provided by the symbol MSG1) is loaded into the DCO registers.

FOUR These two instructions provide one of many ways in
FIVE which programmed I/O may be set up. It is assumed that the receiving device connected to I/O port 0 has an I/O buffer containing all zeros until it is ready to receive data, at which time bit 7 of the I/O buffer is set to 1. The I/O buffer contents are continuously checked until bit 7 (the sign bit) is sensed as a 1 bit. The port is cleared prior to input when used for input and output. For details see Section 8.2.

SIX The contents of the ROM byte addressed by the DCO registers is input to the accumulator; the DCO registers contents are then incremented.

SEVEN The accumulator contents are output to I/O port 0.

EIGHT The buffer length counter (in scratchpad byte 0) is
NINE decremented. If the result is not zero, return to instruction FOUR to process the next character.

Improved text writing programs are given in Section 10.2.

7.2.2 Accessing Data Tables in RAM

Two programming techniques need to be understood in connection with accessing RAM via 3852 or 3853 interface devices:

- a) Processing data between a source buffer and a destination buffer.
- b) Operating on data from two source buffers to create results that are stored in a destination buffer.

Consider first the example of data being moved from one RAM buffer to another. This procedure is very simple on the F8, requiring the following instruction sequence:

```

BUFA  EQU  H'2000'   BUFFER ADDRESSES AND
BUFB  EQU  H'3080'   LENGTH HAVE BEEN ARBI-
                        TRARILY SELECTED
CTHI  EQU  H'02'    CTHI AND CTLO TOGETHER
CTLO  EQU  H'80'    FORM A TWO BYTE BUFFER
      —            LENGTH COUNTER
      —
      —
ONE   LI  CTHI      USE SCRATCHPAD REG-
TWO   LR  1,A      ISTERS 0 AND 1 FOR THE
                        BUFFER LENGTH
      —
      —
      —
THREE LI  CTLO
FOUR  LR  0,A
FIVE  DCI  BUFB    LOAD DESTINATION
                        ADDRESS INTO DCO
                        SAVE IN DC1
SIX   XDC
SEVEN DCI  BUFA    LOAD SOURCE ADDRESS
                        INTO DCO
      —
      —
      —
LOOP  LM
EIGHT XDC
NINE  ST          LOAD SOURCE BYTE
                        EXCHANGE ADDRESSES
                        STORE IN DESTINATION
                        BUFFER
      —
      —
      —
TEN   XDC
ELEV  DS  0      EXCHANGE ADDRESSES
                        DECREMENT LOW ORDER
                        COUNTER BYTE
      —
      —
      —
TWEL  BNZ  LOOP   RETURN IF NOT ZERO
THRT  DS  1      DECREMENT H.O.
                        COUNTER BYTE AND TEST IF

```

FRTN BC LOOP IT WAS 0
 RETURN IF H.O. BYTE WAS
 NOT 0

This program makes no assumptions regarding data buffer size or location. Decrementing 2-byte counters is illustrated in this program, enabling data to be moved between buffers of any size. Program steps proceed as follows:

ONE The two byte buffer length is loaded into scratchpad registers 1 (high order byte) and 0 (low order byte).
 to
 FOUR Notice that the 2-byte count must be loaded as two single byte quantities, since the LI instruction loads a single data byte into the accumulator.

FIVE Save the destination buffer starting address in DC1.
 SIX First the address must be loaded into DC0 using a DCI instruction, then it is transferred to DC1 by the XDC instruction. Note that the DCI instruction has a 2-byte operand, therefore BUFA (and BUFB) are equated as 2-byte addresses.

SEVEN Load the source buffer starting address into DC0. (The destination buffer starting address is now in DC1.)

LOOP Transfer the contents of the memory byte addressed by the DC0 registers to the accumulator. The address in the DC0 registers is automatically incremented and now points to the next byte of the source buffer.

EIGHT Exchange addresses between the DC0 and DC1 registers. The DC0 registers now address the destination buffer.

NINE Store the contents of the accumulator in the memory byte addressed by the DC0 registers. This is now the next destination buffer byte following the previous XDC instruction. After the data byte is stored in the destination buffer the address in the DC0 registers is automatically incremented to address the next destination buffer byte.

TEN Exchange the contents of the DC0 and DC1 registers so that the DC0 registers again address the next source buffer byte.

ELEV The two byte counters CTHI and CTLO, stored in scratchpad registers 1 and 0, respectively, are decremented to zero. Until they decrement to zero, execution returns to LOOP. After they decrement to zero, execution continues at the instruction following FRTN.

Decrement logic proceeds as follows: the low order counter byte is decremented until it reaches zero. At this point the high order counter byte is decremented and simultaneously tested to see if it was decremented from 0. Since the DS instruction, in fact, adds H'FF' to the contents of the scratchpad byte, the carry status will be set unless H'FF' was added to H'00'. Therefore after executing a DS instruction, it is possible to test for a "decrement-from-zero" using the BC instruction. A branch-on-negative (BM) instruction would serve as well.

Consider the current case. Initially CTLO in scratchpad register 0 is decremented from H'80' to 0. At this point, CTHI in scratchpad register 1 contains 2 and there are 512 bytes of data remaining to be moved. The low order byte of the counter is again decremented from H'FF' through to 0, at which point CTHI in scratchpad register 1 contains 1, signifying that 256 bytes of data still remain to be moved. Now the high order byte of the counter in scratchpad register 1 is decremented from 1 to 0. Again the low order byte of the counter in scratchpad register 0 is decremented from H'FF' through to 0. This time no bytes remain to be moved; when the high order byte of the counter in register 1 is tested, it is found to be negative. As required, execution of the loop ceases and the branch occurs to instruction OUT, somewhere beyond the program.

Consider next a three buffer example; two positive, multi-byte numbers are to be added and the sum is to be stored in a third multibyte buffer. This three buffer addition proceeds as follows:

| | | | |
|-------|-----|---------|-------------------------|
| BUFA | EQU | H'0838' | THE CONTENTS OF BUFA |
| BUFB | EQU | H'0920' | AND BUFB ARE ADDED. |
| BUFC | EQU | H'077C' | THE RESULT IS STORED |
| | | | IN BUFC. |
| CNT | — | H'0A' | 10 BYTE BUFFERS ARE |
| | — | | ASSUMED. |
| ONE | LIS | CNT | USE SCRATCHPAD |
| TWO | LR | 0,A | REGISTER 0 AS A COUNTER |
| THREE | DCI | BUFC | SAVE THE ANSWER IN BUF- |
| FOUR | LR | Q,DC | FER STARTING ADDRESS |
| | | | IN Q |
| FIVE | DCI | BUFA | SAVE THE SOURCE BUFFER |
| | | | ADDRESSES |
| SIX | XDC | | IN DC0 AND DC1 |
| SEVEN | DCI | BUFB | |
| EIGHT | COM | | INITIALLY CLEAR THE |
| | | | CARRY BIT |
| | LR | J,W | INITIALIZE STATUS |
| LOOP | LM | | LOAD NEXT BYTE |
| | LR | W,J | MOVE CARRY FROM |
| | | | PRIOR ADD TO STATUS |
| NINE | LNK | | ADD ANY PREVIOUS |
| | | | CARRY |
| TEN | LR | J,W | SAVE STATUS IN J |
| ELEV | XDC | | ADDRESS ADDEND BUFFER |
| TWEL | AM | | ADD CORRESPONDING |
| | | | ADDEND BYTE |
| THRT | XDC | | READDRESS AUGEND |
| | | | BUFFER |
| FRTN | LR | H,DC | SAVE AUGEND ADDRESS |
| | | | IN H |
| FFTN | LR | DC,Q | LOAD ANSWER BUFFER |
| | | | ADDRESS |
| SXTN | ST | | STORE THE ANSWER |
| SVTN | LR | Q,DC | SAVE ANSWER BUFFER |
| | | | ADDRESS IN Q |
| EGTN | LR | DC,H | MOVE AUGEND ADDRESS |
| | | | BACK TO H |
| NNTN | BNC | TWT1 | NO CARRY FROM AM |
| | | | INSTRUCTION |
| TWTY | LR | J,W | SAVE CARRY FROM AM |
| | | | INSTRUCTION |

TWT1 DS 0 DECREMENT COUNTER
 BNZ LOOP RETURN FOR MORE

TEN therefore saves the status register in the scratchpad J register (register number 9).

This program executes as follows:

- ONE Scratchpad register 0 is used as a counter. Buffer length has arbitrarily been assumed to be ten bytes.
- TWO
- THREE Since three 16-bit addresses have to be maintained, the following scheme will be used. At any time the buffer being accessed must have its address in DC0; however, DC1 plus the Q and H registers in the scratchpad memory are available to store addresses which are out of service. Accordingly, the answer buffer address will be saved in Q, the addend buffer address will be saved in DC1 and the augend buffer address will be saved in H whenever the answer buffer address is moved from Q to DC0. This scheme is illustrated in Figure 7-1.
- SEVEN

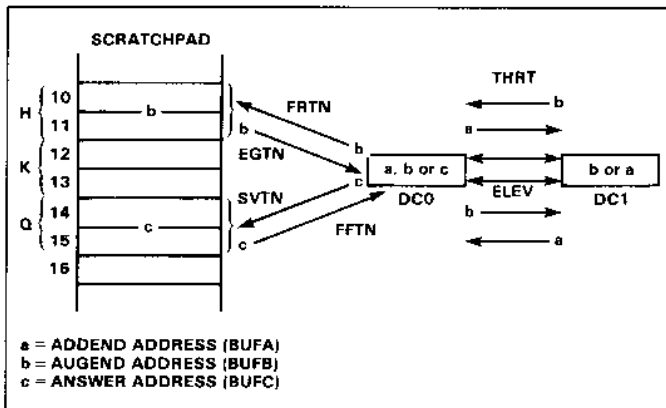


Fig. 7-1. Use of H, Q and DC1 Registers to Hold Three Buffer Addresses

Initially, it is necessary to load the answer buffer starting address into the Q registers, the addend buffer starting address into DC1 and the augend buffer address into DC0.

- EIGHT The carry status must initially be set to 0 before the first two bytes are added. This is done by complementing whatever happens to be in the accumulator, since the complement instruction automatically sets the carry status to 0.
- LOOP Load the next augend byte. The augend byte address is initially loaded into DC0 and is returned to DC0 at the end of the addition loop. After the augend byte has been loaded into the accumulator, DC0 contents are automatically incremented.
- NINE Add any carry from the previous byte addition to the augend byte in the accumulator. (Instruction EIGHT will have set the carry to 0 before the first two bytes are added.)
- TEN As described in Section 7.1.2, addition logic must take account of the fact that when the link is added to the accumulator it is possible for the accumulator to contain H'FF' and the link to contain 1. In this case the result will be zero in the accumulator with 1 in the carry status. Subsequent addition of the addend byte will destroy the carry status. Instruction

ELEV to THRT These three instructions switch the contents of the DC0 and DC1 registers (DC0 will now address the augend buffer). The contents of the next augend byte are added to the accumulator using binary addition. The augend buffer address in DC0 is automatically incremented after performing the addition. Then the augend and addend addresses are exchanged so that after instruction THRT has been executed, DC0 addresses the next addend byte and DC1 addresses the next augend byte.

FRTN to EGTN The sum in the accumulator must now be saved in the next answer buffer byte. The answer buffer address is in the scratchpad Q registers (registers 14 and 15). Before moving the answer buffer address to the DC0 registers, the DC0 registers contents are saved in the scratchpad H registers (registers 10 and 11). Instruction SXTN stores the answer byte in the accumulator into the answer buffer, then increments the answer buffer address in the DC0 registers. Instruction SVTN saves the incremented answer buffer address back in the Q registers while instruction EGTN restores the augend address from the H register to the DC0 registers.

NNTN Observe that instructions FRTN through EGTN do not modify any of the status bits. As described in Section 7.1.2, the correct carry status to be used when adding the next two bytes is given by ORing the carry status from instructions NINE and TWEL. If instruction TWEL created a 0 carry, then the carry saved by TEN is valid. If instruction TWEL created a 1 carry, it must be saved (by TWTY), to be recalled following LOOP. Since DS in TWT1 resets the carry to 0, it is necessary to save the carry status in 9, across the DS instruction. Note the difference in technique for preserving the carry status in this example, where DS resets the carry, as compared to Section 7.1.2, where statuses are not destroyed.

TWT1 The buffer length counter in scratchpad register 0 is now decremented. If it does not decrement to zero return to LOOP to add the next two bytes of the buffer.

7.3 SUBROUTINES

7.3.1 The Concept of a Subroutine

Any logic that will be used more than once can be written as a subroutine. For example, the 16-bit, signed binary addition program given in Section 7.1.4 may be needed at a number of different points within one large program. The routine may be repeated wherever it is needed. For example, the eleven instructions of the 16-bit signed binary addition routine may re-appear ten times within a program that uses this logic ten times. When the code is reproduced, without modification, it is wasting memory.

There are two ways in which an often used routine may be accessed by a program:

- 1) The code can be reproduced with minor modifications, in which case it is treated as a Macro, as described in Section 7.4.

- 2) The routine may be stored once, then accessed for execution each time it is needed. The routine is now called as a subroutine.

Figure 7-2 illustrates the concept of a subroutine.

There are four aspects of subroutines that must be considered; they are:

- 1) The program steps of the logic being bundled as a subroutine.
- 2) How the subroutine is accessed. (This is termed "calling" the subroutine.)
- 3) Returning from the subroutine after it has executed.
- 4) Passing data, as parameters, to the subroutine.

Each aspect of a subroutine will be examined with reference to the multibyte addition routine described in Section 7.2.2.

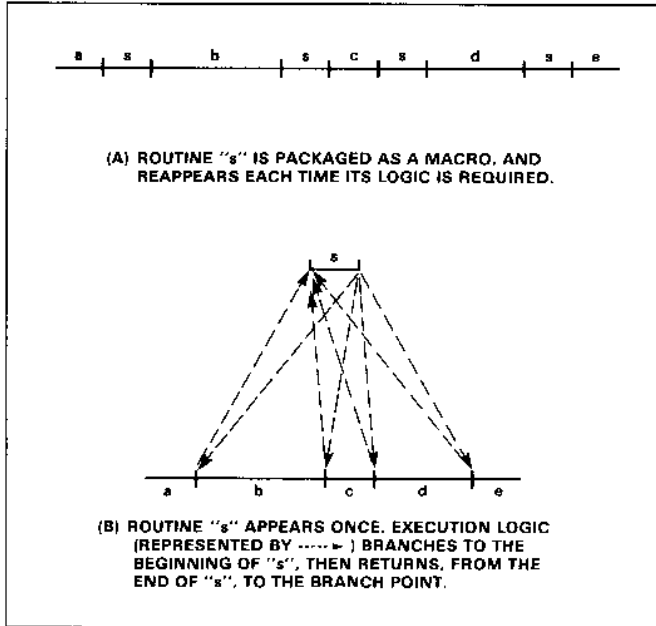


Fig. 7-2. Subroutine, as Compared to a Macro

7.3.2 Subroutine Program Steps

The instructions that implement any logic are the same within, or outside of, a subroutine. Compare the 16-bit addition program (AD16) in Section 7.3.3 with the equivalent program in Section 7.2.1; the only changes relate to entry and exit procedures.

7.3.3 Simple Subroutine Calls and Returns

As described in Sections 6.35 and 6.36, there are two instructions used to call a subroutine into execution.

Instruction PK saves the contents of the program counter (PC0) in the stack register (PC1), then loads the subroutine starting address from the K register (scratchpad registers 12 and 13) into the program counter.

Instruction PI saves the contents of the program counter in the stack register; it then loads the subroutine starting address (which is in the two object program bytes following the PI op code byte) into the program counter.

For straightforward returns from subroutines, the POP instruction, described in Section 6.37, moves the contents of

the stack register back to PC0, thus effecting a return from a subroutine.

PK may also be used to return from a subroutine by having the return address in the K registers. LR PO,Q likewise may be used to return by having the return address in the Q register.

The starting address of a subroutine is identified by the subroutine name, which is the label of the first instruction to be executed in the subroutine.

Suppose the multibyte addition routine from Section 7.2.2 is to be named MADD, while the 16-bit addition routine from Section 7.1.4 is to be named AD16. These names are created by changing

```
ONE   LI   CNT   USE SCRATCHPAD
                        REGISTER 0
```

to the following equivalent instruction:

```
MADD  LI   CNT   USE SCRATCHPAD
                        REGISTER 0
```

For AD16, change

```
ONE   LR   A,0   LOAD LOW ORDER
                        AUGEND BYTE
```

to the following equivalent instruction:

```
AD16  LR   A,0   LOAD LOW ORDER
                        AUGEND BYTE
```

Note that although the first sequential instruction is also the first executed instruction for MADD and AD16, the first executed instruction may, in reality, be any instruction within the subroutine.

The last instruction executed by a subroutine must be POP, PK or LR PO,Q. Therefore, if for the moment the AD16 error return is ignored, subroutine AD16 becomes:

```
AD16  LR   A,0   FIRST INSTRUCTION
                        EXECUTED FOR AD16
TWO   AS   2
THREE LR   2,A
FOUR  LR   A,1
FIVE  LNK
SIX   BNO  EIGHT
SEVN  POP
EIGHT AS   3
NINE  LR   3,A
OUT   POP
                        IF THE RESULT IS TOO
                        LARGE, RETURN
                        RETURN AT END OF
                        SUBROUTINE
```

Notice that a subroutine may have more than one exit.

Subroutine MADD becomes:

```

MADD LI CNT FIRST INSTRUCTION
      LR 0,A EXECUTED FOR MADD
      —
      —
      —
(rest of subroutine as in Section 7.2.2)
      —
      —
      —
TWT1 DS 0
      BNZ LOOP RETURN FOR MORE
      POP RETURN AT END OF
           SUBROUTINE
  
```

Consider the very simple case of subroutine AD16 being called using a PI instruction. Instruction sequences, with arbitrarily selected memory addresses, might be as follows:

```

Memory *MAIN PROGRAM SEGMENT
Address —
        —
        —
H'102A' ONE PI AD16
H'102D' TWO INC
        —
        —
        —
        *SUBROUTINE AD16 STARTS HERE
H'2130' AD16 LR A,0
        —
        —
        —
H'213B' OUT POP
  
```

Before instruction ONE is executed, PC0 contains H'102A'. After instruction ONE has executed, PC0 contains H'2130' and PC1 contains H'102D'. Execution now proceeds from AD16, at H'2130'.

Before instruction OUT is executed, PC0 contains H'213B' and PC1 still contains H'102D'. Instruction OUT moves H'102D' to PC0, thus returning execution to TWO.

The following sequence illustrates PK being used to call AD16, and PI being used to call MADD:

```

*THIS ORIGIN FOR AD16 HAS BEEN ARBITRARILY SELECTED
      ORG H'0980'
AD16 LR A,0 LOAD LOW ORDER
      — AUGEND BYTE
      —
      —
(rest of subroutine follows here)
      —
      —
      —
  
```

*THIS ORIGIN FOR MADD HAS BEEN ARBITRARILY SELECTED

```

      ORG H'09E0'
MADD LI CNT USE SCRATCHPAD
      — REGISTER 0
      —
      —
  
```

(rest of subroutine follows here)

*THIS ORIGIN FOR THE MAIN PROGRAM HAS BEEN
*ARBITRARILY SELECTED

```

      ORG H'1000'
*BEFORE SUBROUTINE AD16 IS FIRST CALLED, LOAD
*ITS STARTING ADDRESS INTO THE SCRATCHPAD K
REGISTERS
  
```

```

ONE LI H'09' LOAD STARTING ADDRESS
TWO LR KU,A OF SUBROUTINE AD16 INTO
THREE LI H'80' K REGISTER
      LR KLA
      —
      —
FOUR PK FIRST CALL TO SUBROUTINE
      — AD16
      —
FIVE PI MADD FIRST CALL TO SUBROUTINE
      — MADD
      —
SIX PK SECOND CALL TO SUB-
      — ROUTINE AD16
      —
SEVEN PK THIRD CALL TO SUB-
      — ROUTINE AD16
      —
      —
      etc
  
```

7.3.4 Nested Subroutines

"Nesting" is the term applied to subroutines being called from within other subroutines.

There is no reason why a subroutine should not, itself, call another subroutine. In fact, subroutines are such efficient programming tools, that it is not uncommon to find subroutines nested eight deep, or more, in large programs.

Consider a very simple case, where creation of the correct carry status for multibyte addition is packaged into a subroutine named CBIT. This subroutine is equivalent to instructions EIGHT through THRT of the addition program in Section 7.1.2.

Subroutine CBIT appears as follows:

```

CBIT LR A,9 MOVE STATUS FROM LNK
      — ADDITION TO A
      LR J,W MOVE STATUS FROM BYTE
      — ADD TO W
      XS 9 EXCLUSIVE-OR STATUSES
      LR 9,A RETURN STATUS TO J VIA W
      LR W,J
      POP
  
```


First try changing the addition program in Section 7.1.2 as follows:

| | | | |
|-----------------|-------------|-------|--|
| MADS | COM | | INITIALLY CLEAR THE CARRY STATUS |
| ONE | LISU | 7 | ADDRESS LOW ORDER BYTE OF EACH BUFFER |
| LOOP TWO | LISU LR | 4 A,S | ADDRESS FIRST BUFFER LOAD FIRST BUFFER BYTE INTO A |
| THREE FOUR FIVE | LISU LNK LR | 5 J,W | ADDRESS SECOND BUFFER ADD ANY CARRY TO A SAVE STATUS IN SCRATCH-PAD BYTE 9 |
| SIX | AS | S | ADD SAME BYTE OF SECOND BUFFER |
| SEVEN | LR | D,A | STORE ANSWER AND INCREMENT BYTE POINTER |
| EIGHT | PI | CBIT | CALL C STATUS SUBROUTINE |
| FORT FIFT | BR7 POP | LOOP | RETURN IF NOT END RETURN FROM SUBROUTINE |

The addition routine has been converted into a subroutine named MADS.

When subroutine MADS is called, the return address is stored in PC1 to be returned to PC0 by POP instruction FIFT. Unfortunately, when CBIT is called at EIGHT, the PI instruction will also store a return address, the address of instruction FORT, in PC1. The POP at FIFT will no longer work, since it will branch execution back to FORT, thus forming an endless execution loop. (This type of program error is handled by the MAXCPU directive.)

When subroutines are nested one deep, (and this is often sufficient in simple F8 applications), the K registers in the scratchpad can be used to overcome the problem of wiping out PC1. For example, in Subroutine MADD, save PC1 in K upon entering MADS then use PK to return from MADD:

| | | | |
|-----------|--------|------|--|
| MADS | LR | K,P | SAVE RETURN ADDRESS |
| | COM | | INITIALLY CLEAR THE CARRY STATUS |
| — | — | — | — |
| EIGHT | PI | CBIT | CALL C STATUS SUBROUTINE |
| FORT FIFT | BR7 PK | LOOP | RETURN IF NOT END RETURN FROM SUBROUTINE FOR END |

When subroutines are nested more than two deep, a stack is created in RAM to hold subroutine return addresses. When creating such a memory stack, it is wise to use PC1 and K as address conduits to the stack, never actually retaining address permanently in PC1 or K.

Consider the following simple, three-deep subroutine nest:

```

Arbitrary
Memory
Addresses
*MAIN PROGRAM
—
—
—

```

| | | | | |
|---------|------|-----|-----------------------------|--------------------------------|
| H'080A' | ONE | PI | SUB1 | CALL FIRST SUBROUTINE |
| H'080D' | NXT1 | — | — | FIRST SUBROUTINE RETURNS HERE |
| — | — | — | — | — |
| | | | ORG H'2073' | |
| | | | *START OF FIRST SUBROUTINE | |
| H'2073' | SUB1 | — | — | FIRST INSTRUCTION OF SUB1 |
| — | — | — | — | — |
| H'2082' | TWO | PI | SUB2 | CALL TO SECOND SUBROUTINE |
| H'2085' | NXT2 | — | — | SECOND SUBROUTINE RETURNS HERE |
| — | — | — | — | — |
| H'2132' | RET1 | POP | — | RETURN TO MAIN PROGRAM |
| — | — | — | — | — |
| | | | *START OF SECOND SUBROUTINE | |
| | | | ORG H'12A4' | |
| H'12A4' | SUB2 | — | — | FIRST INSTRUCTION OF SUB2 |
| — | — | — | — | — |
| H'12B3' | THRE | PI | SUB3 | CALL TO THIRD SUB |
| H'12B6' | NXR3 | — | — | THIRD SUBROUTINE RETURNS HERE |
| — | — | — | — | — |
| H'12E2' | RET2 | POP | — | RETURN TO FIRST SUBROUTINE |
| — | — | — | — | — |
| | | | *START OF THIRD SUBROUTINE | |
| | | | ORG H'1558' | |
| H'1558' | SUB3 | — | — | FIRST INSTRUCTION OF SUB3 |
| — | — | — | — | — |
| — | — | — | — | — |
| H'1596' | RET3 | POP | — | RETURN TO SECOND SUBROUTINE |

The sequence in which instructions are executed is given in Table 7-2, along with contents of PC0, PC1, and a "stack" in memory, where PC1 contents may be stored.

Notice that the first return address, H'080D', is passed to S0, the first two bytes of the memory stack. Similarly the second (H'2085') and third (H'12B6') return addresses are stored in the second and third byte pairs of memory stack. At all times, data in PC1 and K are merely the accidental result of logic needed to pass return addresses to the stack.

A memory stack "pointer" must be maintained. After each return address is stored in the stack, the stack pointer will identify the next free stack byte.

Return logic is the opposite of subroutine call logic. Before each call, the most recently stored return address (in the two stack bytes right behind the stack pointer) are moved to PC1, and the stack pointer address is decremented by 2.

The memory stack may either be in scratchpad memory, or in RAM memory.

| INSTRUCTION LABEL 1) | REGISTERS/STACK CONTENTS | | | | | | |
|----------------------|--------------------------|------|------|---------------------------------|------|------|----------|
| | PC0 | PC1 | K | First six bytes of Memory Stack | | | |
| | | | | S0 | S1 | S2 | |
| ONE | 080A | ? | ? | ? | ? | ? | ← Before |
| | 2073 | 080D | ? | ? | ? | ? | ← After |
| SUB1 | 2073 | 080D | ? | ? | ? | ? | ← Before |
| | 2082 | 080D | 080D | 080D | ? | ? | ← Before |
| TWO | 12A4 | 2085 | 080D | 080D | ? | ? | ← After |
| | 12A4 | 2085 | 080D | 080D | ? | ? | ← Before |
| THRE | 12B3 | 2085 | 2085 | 080D | 2085 | ? | ← Before |
| | 1558 | 12B6 | 2085 | 080D | 2085 | ? | ← After |
| SUB3 | 1558 | 12B6 | 2085 | 080D | 2085 | ? | ← Before |
| | 1596 | 12B6 | 12B6 | 080D | 2085 | 12B6 | ← Before |
| RET3 | 12B6 | 12B6 | 12B6 | 080D | 2085 | 12B6 | ← After |
| NXT3 | 12B6 | 12B6 | 12B6 | 080D | 2085 | 12B6 | ← Before |
| RET2 | 12E2 | 2085 | 2085 | 080D | 2085 | 12B6 | ← Before |
| | 2085 | 2085 | 2085 | 080D | 2085 | 12B6 | ← After |
| NXT2 | 2085 | 2085 | 2085 | 080D | 2085 | 12B6 | ← Before |
| RET1 | 2132 | 080D | 080D | 080D | 2085 | 12B6 | ← Before |
| | 080D | 080D | 080D | 080D | 2085 | 12B6 | ← After |

Instructions are in order of execution

Table 7-2. Use of a Memory Stack for Executing Multiple Level Subroutines

Consider first a stack in scratchpad memory. By assigning one 8-byte buffer to serve as a memory stack, subroutines may be nested four deep. One byte at the beginning of scratchpad memory will serve as the stack pointer.

Subroutine CALL, described next, uses scratchpad bytes 0'77' to 0'70' as the memory stack, as illustrated in Figure 7-3. Scratchpad byte 8 is the stack pointer which must be initialized to H'77'.

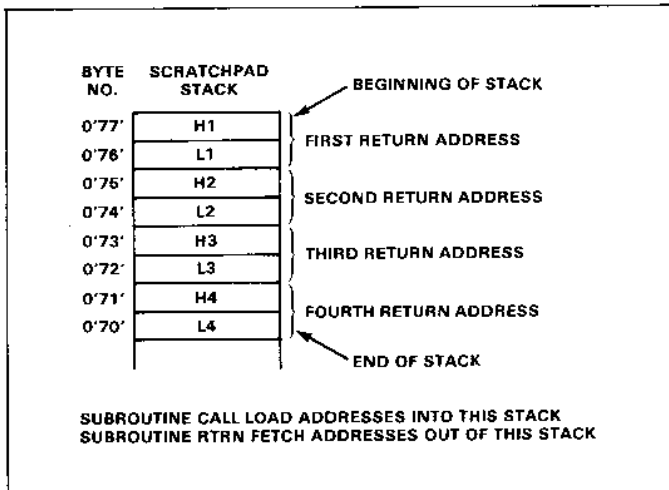


Fig. 7-3. Scratchpad Stack

Every subroutine must begin by saving PC1 contents in K, then calling CALL. This is illustrated as follows for subroutine MADD, which is the addition program from Section 7.2.2, converted into a subroutine:

| | | | |
|------|-----|------|--|
| MADD | LR | K,P | SAVE RETURN ADDRESS IN K |
| | PI | CALL | SAVE RETURN ADDRESS IN STACK |
| ONE | LIS | CNT | USE SCRATCHPAD REGISTER 0 AS A COUNTER |
| TWO | LR | 0,A | |
| | — | | |
| | — | | |

Since the call to CALL is preceded by PC1 contents being saved in K, PC1 is now free to hold the return address for CALL. Subroutine CALL has the following instructions:

| | | | |
|------|-----|-------|--|
| CALL | LR | A,8 | MOVE THE STACK POINTER TO ISAR |
| C1 | LR | IS,A | |
| C2 | CI | 0'67' | CHECK FOR STACK OVERFLOW |
| C3 | BZ | SFUL | STACK HAS OVERFLOWED. MAKE ERROR EXIT. |
| C5 | LR | A,KU | MOVE KU TO STACK |
| C6 | LR | D,A | |
| C7 | LR | A,KL | MOVE KL TO STACK |
| C8 | LR | S,A | DO NOT DECREMENT ISAR |
| C9 | LR | A,IS | SAVE ISAR IN SCRATCHPAD BYTE 8 |
| C10 | LR | 8,A | |
| C11 | DS | 8 | DECREMENT VALUE SAVED FOR ISAR |
| C12 | POP | | RETURN |

The address of the next free stack byte is held in scratchpad byte 8. If this address is 0'67', it means that 0'70' is the

address of the last filled stack byte and the stack is full. Therefore when CALL is called, the stack address is tested for overflow by checking ISAR. A value of O'67' indicates the stack has overflowed. A value of O'77' indicates the stack is empty.

Subroutine CALL logic proceeds as follows:

- CALL Move the stack address from scratchpad byte 8 to ISAR
- C1
- C2 Test stack address for O'67'.
- C3 It is assumed that SFUL is the label of an instruction which will handle stack full errors in any way required by program logic. This instruction branches execution to the instruction labeled SFUL.
- C5 Move the contents of the K registers to the next two free bytes of the stack. The ISAR is only decremented once. The second decrement can be performed in the scratchpad, where O'70' will decrement to O'67' which indicates stack full, rather than to O'77' which would erroneously indicate stack empty.
- C8
- C9 Return the new contents of ISAR to scratchpad register A.
- C10
- C11 Decrement ISAR in scratchpad so that O'70' will decrement to O'67' which is full, not to O'77', which is empty.
- C12 Return from subroutine CALL.

A subroutine that uses CALL to save its return address on the stack will use another subroutine, named RTRN, to return to the calling program. For example, subroutine MADD will now end with:

TWT2 PI RTRN

Since RTRN resets PCO, PI may be replaced with:

TWT2 JMP RTRN

Subroutine RTRN takes the address most recently stored in the stack and moves this address to PCO, effecting the desired return, as follows:

| | | | |
|------|----|------|---|
| RTRN | LR | A,8 | MOVE THE STACK POINTER TO ISAR |
| R1 | LR | IS,A | |
| | LR | A,I | INCREMENT ADDRESS TO LAST FILLED STACK BYTE |
| R2 | LR | A,I | MOVE THE ADDRESS IDENTIFIED BY ISAR TO Q |
| R3 | LR | QL,A | |
| R4 | LR | A,S | |
| R5 | LR | QU,A | |
| R6 | LR | A,IS | RESTORE ISAR |
| R7 | LR | 8,A | |
| R8 | LR | PO,Q | MOVE Q TO PCO |

Subroutine RTRN executes as follows:

- RTRN Move the stack pointer address from scratchpad register 8 to ISAR. Increment ISAR to move address from the first free stack byte to the last occupied stack byte.
- R1
- R2 Move the address identified by ISAR to QL and QU. Increment ISAR to point to the prior address. Leave ISAR addressing what is now the first free byte.
- R5
- R6 Save the new value of ISAR in scratchpad register 8.
- R7
- R8 The subroutine that called RTRN now wishes to return to the address which RTRN has moved to the Q registers. RTRN can simply move this address from Q to PCO in order to effect the desired return.

For large stacks, RAM memory may be used for the memory stack. Only minor logic modifications are required to CALL and RTRN if the stack is in RAM. Assuming that the stack pointer is maintained in scratchpad registers 8 (high) and 7 (low), subroutine CALR and RTRR perform the same functions as CALL and RTRN but, for a RAM stack, that may be more than 256 bytes long.

As for the scratchpad stack, the RAM stack begins at a high RAM address, and the stack address is decremented as the stack gets filled. The end of the RAM stack is identified by a low address, represented using the symbols SPHI and SPLO for the high and low order bytes of the address.

The stack pointer address identifies the last filled stack byte.

*VERSION OF SUBROUTINE CALL FOR RAM STACKS, WITH *THE STACK POINTER IN SCRATCHPAD REGISTERS 8 AND 7.

| | | | |
|------|-----|------|--|
| CALR | LR | A,7 | LOAD LOW ORDER BYTE OF STACK ADDRESS |
| | LR | 11,A | MOVE TO HL |
| | CI | SPLO | COMPARE WITH END-OF-STACK L.O. BYTE |
| | LR | A,8 | LOAD HIGH ORDER BYTE OF STACK ADDRESS |
| | LR | 10,A | STORE IN HU |
| | BNE | CA8 | IF LOW ORDER BYTE DOES NOT EQUAL STACK END, CONTINUE |
| | CI | SPHI | COMPARE HIGH ORDER BYTES |
| | BEQ | CA20 | IF EQUAL, STACK HAS OVERFLOWED |
| CA8 | LR | DC,H | MOVE H TO DC |

*SUBTRACT 2 FROM THE STACK ADDRESS, SINCE IT INCREMENTS WHEN MEMORY IS ACCESSED. BY SUBTRACTING *2, DCO ADDRESSES THE SECOND FREE STACK BYTE.

| | | | |
|--|-----|-------|------------------|
| | LI | H'FE' | |
| | ADC | | |
| | LR | A,KU | MOVE KU TO STACK |
| | ST | | |
| | LR | A,KL | MOVE KL TO STACK |
| | ST | | |

*SUBTRACT 2 FROM STACK ADDRESS, SINCE IT HAS INCREMENTED TO BEGINNING OF PREVIOUS ADDRESS.

| | | | |
|--|-----|-------|-----------------------|
| | LI | H'FE' | |
| | ADC | | |
| | LR | H,DC | RESTORE STACK POINTER |
| | LR | A,11 | |

| | | | |
|------|-----|------|------------------|
| | LR | 7,A | |
| | LR | A,10 | |
| | LR | 8,A | |
| | POP | | RETURN |
| CA20 | JMP | SFUL | STACK FULL ERROR |

The logic of CALR differs from the logic of CALL only in the way stack overflow is handled. Rather than leaving the stack pointer addressing the next free byte of the stack, the stack pointer addresses the last used byte of the stack. Stack overflow is tested for by comparing the contents of the stack pointer with an address that has been specified as the end of the stack. This end of stack address can be equated to any value that is convenient to program logic.

Notice that whenever memory is accessed via the DC registers the address in the DC registers is automatically incremented. The stack in RAM has arbitrarily been selected to begin at a high address and end at a low address, which is the opposite direction as seen by the DC registers. Since the DC registers address the last filled byte of the stack, two must be subtracted from this address so that two bytes of address data may be loaded into the stack without overloading the last filled byte. Also, after the two bytes of address have been loaded into the stack, two must again be subtracted from the address in the DC registers so that the address once again identifies the last filled byte of the stack.

Although the sense of direction of the stack is inverted with regard to the DC registers when CALR is executed, stack direction will agree with the DC registers when RTRR is executed. Since stack access involves a forward and then a reverse direction, it makes no difference what is chosen to be forward and what reverse; either CALR or RTRR must access the stack by decrementing addresses. This is contrary to the sense of the DC registers which only increment addresses.

*VERSION OF SUBROUTINE RTRN FOR RAM STACKS, WITH
*THE STACK POINTER IN SCRATCHPAD REGISTERS 8 AND 7.

| | | | |
|------|----|------|--|
| RTRR | LR | A,8 | MOVE THE STACK POINTER TO H |
| | LR | 10,A | |
| | LR | A,7 | |
| | LR | 11,A | |
| | LR | DC,H | MOVE THE STACK POINTER TO DC |
| | LM | | LOAD HIGH ORDER BYTE OF RETURN ADDRESS INTO QU |
| | LR | QU,A | |
| | LM | | LOAD LOW ORDER BYTE OF RETURN ADDRESS INTO QL |
| | LR | QL,A | |
| | LR | H,DC | SAVE STACK POINTER IN SCRATCHPAD BYTES 8 AND 7 |
| | LR | A,10 | |
| | LR | 8,A | |
| | LR | A,11 | |
| | LR | 7,A | |
| | LR | PO,Q | MOVE Q TO PCO |

In F8 systems that have a 3852 and/or 3853 Memory Interface device, if DC1 is not used to address data buffers, it can be used effectively as a RAM stack pointer.

7.3.5 Multiple Subroutine Returns

Observe that the 16-bit addition subroutine in Section 7.1.4 requires two returns, one for an overflow in the answer, the other for a valid execution.

Frequently subroutines may execute with more than one possible outcome. The most efficient way of handling such logic is to build multiple returns into the calling program and into the called subroutine. Here are some examples. First, an error return:

| | | |
|----|------|----------------------------|
| — | | |
| — | | |
| — | | |
| PI | SUB1 | CALL SUBROUTINE SUB1 |
| BR | ERR | ERROR RETURN FROM SUB1 |
| — | | NON-ERROR RETURN FROM SUB1 |
| — | | |
| — | | |

Next, multiple valid returns:

| | | |
|----|------|--------------------|
| PI | SUB2 | |
| BR | PLUS | RESULT IS POSITIVE |
| BR | ZERO | RESULT IS ZERO |
| — | | RESULT IS NEGATIVE |
| — | | |

Subroutines RTRN and RTRR can easily be rewritten to handle multiple returns. Instructions will be added that return, to PCO, the last address entered into the stack, plus any displacement that is in QL (scratchpad register 15) when the subroutine is called. RTRN will now appear as follows, renamed RTRD:

| | | | |
|------|-----|------|---------------------------------------|
| RTRD | LR | A,8 | MOVE STACK POINTER TO ISAR |
| | LR | IS,A | |
| | LR | A,I | INCREMENT ADDRESS TO LAST FILLED BYTE |
| | LR | A,QL | LOAD LOW ORDER ADDRESS BYTE |
| | AS | I | ADD DISPLACEMENT IN QL |
| | LR | QL,A | STORE RESULT IN QL |
| | LR | A,S | LOAD HIGH ORDER ADDRESS BYTE |
| | LNK | | ADD ANY CARRY FROM LO BYTE ADDITION |
| | LR | QU,A | STORE RESULT IN QU |
| R6 | LR | A,IS | RESTORE ISAR |
| R7 | LR | 8,A | |
| | LR | PO,Q | MOVE Q TO PCO |

Taking advantage of RTRD, the 16-bit addition subroutine will become:

| | | | |
|------|----|------|---|
| AD16 | LR | K,P | SAVE RETURN ADDRESS IN K |
| | PI | CALL | SAVE RETURN ADDRESS IN SCRATCHPAD STACK |
| | LR | A,0 | LOAD LOW ORDER AUGEND BYTE |
| | AS | 2 | ADD ADDEND LOW ORDER BYTE |
| | LR | 2,A | SAVE THE RESULT |
| | LR | A,1 | LOAD HIGH ORDER |

| | | | |
|-------|-----|-------|------------------------|
| | | | AUGEND BYTE |
| | LNK | | ADD ANY CARRY FROM |
| | BNO | EIGHT | LOW ORDER BYTE ADD |
| | | | IF THERE IS A CARRY OR |
| | | | AN OVERFLOW, RETURN |
| | | | WITHOUT DISPLACEMENT |
| EIGHT | BR | ERR | |
| | AS | 3 | ADD THE HIGH ORDER |
| | | | ADDEND BYTE |
| | LR | 3,A | SAVE THE RESULT |
| | LIS | 2 | FOR A GOOD RETURN, ADD |
| | | | 2 TO RETURN ADDRESS |
| | BNO | OK | AGAIN IF THERE IS A |
| | | | CARRY OR OVERFLOW |
| ERR | CLR | | FOR AN ERROR RETURN, |
| | | | ADD 0 TO RETURN |
| | | | ADDRESS |
| OK | LR | QL,A | SAVE THE DISPLACEMENT |
| | | | IN QL |
| | PI | RTRD | |

Now AD16 will be called as follows:

| | | | |
|---|----|-------|--------------|
| — | | | |
| — | | | |
| — | | | |
| | PI | AD16 | |
| | BR | ERROR | ERROR RETURN |
| | | | GOOD RETURN |
| — | | | |
| — | | | |

7.3.6 Passing Parameters

Subroutine MADD, as described so far, is of limited value, since the starting addresses of buffers BUFA, BUFB and BUFC are fixed. MADD will only add the contents of two fixed buffers and store the result in a third, fixed buffer. The subroutine would be far more useful if buffer locations and lengths could be specified at the time the subroutine is called. This can be done and is called parameter passing.

The parameters to be passed to a subroutine are listed, in the calling program, after the subroutine call. For example, the call to MADD would appear as follows:

| | | |
|-------|-----|---------|
| VALA | EQU | H'2080' |
| VALB | EQU | H'2088' |
| VALC | EQU | H'2800' |
| COUNT | EQU | H'08' |
| — | | |
| — | | |
| — | | |

*CALL TO MADD IN MAIN PROGRAM

| | | |
|----|-------|-------------------------|
| PI | MADD | CALL SUBROUTINE MADD |
| DC | VALA | VALA IS A TWO BYTE |
| | | AUGEND BASE ADDRESS |
| DC | VALB | VALB IS A TWO BYTE |
| | | ADDEND BASE ADDRESS |
| DC | VALC | VALC IS A TWO BYTE |
| | | ANSWER BASE ADDRESS |
| DC | COUNT | COUNT IS A ONE BYTE |
| | | BUFFER LENGTH |
| BR | ERROR | RETURN HERE IF THERE IS |
| — | | AN ERROR |
| — | | RETURN HERE FOR |
| — | | SUCCESSFUL EXECUTION |

Once MADD is entered, the return address in PC1 is in fact, the address where the augend buffer starting address will be found. Before entering into the body of the subroutine, MADD will load parameters into H, Q and DC1. This is illustrated below for subroutine MADD with parameters, renamed MADP.

| | | | |
|------|-----|------|----------------------------------|
| MADP | LR | K,P | SAVE THE RETURN ADDRESS |
| | | | IN P |
| | PI | CALL | SAVE THE RETURN ADDRESS |
| | | | IN THE STACK |
| | LR | A,KU | MOVE THE RETURN |
| | | | ADDRESS FROM K |
| | LR | 10,A | TO DC0. DC0 WILL NOW |
| | | | ADDRESS THE FIRST OF THE |
| | LR | A,KL | TWO BYTES IN WHICH VALA |
| | | | IS STORED, FOLLOWING |
| | LR | 11,A | THIS CALL TO MADP |
| | LR | DC,H | LOAD PARAMETER |
| | | | ADDRESS INTO DC0 |
| | LM | | LOAD VALA INTO H |
| | LR | 10,A | |
| | LM | | |
| | LR | 11,A | |
| | XDC | | MOVE VALA FROM H TO |
| | | | DC1 BY |
| | LR | DC,H | EXCHANGING DC0 WITH DC1 |
| | XDC | | MOVING VALA TO DC0, |
| | | | THEN |
| | LM | | AGAIN EXCHANGING DC0 |
| | | | AND DC1 |
| | LR | 10,A | NEXT LOAD VALB INTO H |
| | LM | | |
| | LR | 11,A | |
| | LM | | LOAD VALC INTO Q |
| | LR | QU,A | |
| | LM | | |
| | LR | QL,A | |
| | LM | | LOAD COUNT INTO |
| | | | SCRATCHPAD BYTE 0 |
| | LR | 0,A | |
| | LR | DC,H | MOVE VALB TO DC0 |
| | | | *THE MULTIBYTE ADD MAY NOW BEGIN |
| | COM | | INITIALLY CLEAR THE CARRY |
| | | | BIT |
| LOOP | LR | J,W | |
| | LM | | LOAD THE NEXT AUGEND |
| | | | BYTE |
| | LR | W,J | |
| | LNK | | ADD ANY PREVIOUS CARRY |
| | LR | J,W | SAVE STATUS IN J |
| | XDC | | ADDRESS ADDEND BUFFER |
| | AM | | ADD CORRESPONDING |
| | | | ADDEND BYTE |
| | XDC | | READDRESS AUGEND |
| | | | BUFFER |
| | LR | H,DC | SAVE AUGEND ADDRESS |
| | | | IN H |
| | LR | DC,Q | LOAD ANSWER BUFFER |
| | | | ADDRESS |
| | ST | | STORE THE ANSWER |
| | LR | Q,DC | SAVE THE ANSWER BUFFER |
| | | | ADDRESS IN Q |
| | LR | DC,H | MOVE AUGEND ADDRESS |
| | | | BACK FROM H |
| | BNC | TWT1 | NO CARRY FOR NEXT BYTE |
| | LR | J,W | |

| | | | |
|------|-----|------|-------------------------------------|
| TWT1 | DS | | DECREMENT COUNTER |
| | BNZ | LOOP | RETURN FOR MORE |
| | LR | W,J | |
| | LIS | 9 | LOAD A FOR A GOOD RETURN |
| | BNC | OUT | TEST FOR A FINAL CARRY |
| | LIS | 7 | THERE IS A CARRY, PREPARE FOR ERROR |
| OUT | LR | QL,A | SAVE THE DISPLACEMENT IN QL |
| | PI | RTRD | RETURN FROM SUBROUTINE |

Parameter passing works as follows:

A subroutine that expects to receive parameters will initiate execution with the return address pointing to the first byte of the parameter list, and not to the instruction which must be executed once program control returns to the calling program. In other words, after subroutine CALL has executed, the address saved on the stack is the address of the first parameter, not the address of the next instruction to be executed in the calling program. Initial subroutine logic must therefore move the address of the first parameter to the DCO registers, and must then appropriately load parameters into registers where they will be needed for execution of the subroutine. This process is straightforward data movement and requires no special explanation.

Observe that when subroutine RTRD is called to effect a return to the calling program (in this case the main program which called MADP) the return address, as stored in the stack, is still the address of the first parameter byte. Therefore, before RTRD is called, the value loaded into the accumulator is not zero or a displacement representing multiple returns. It is the number of bytes of parameters, or the number of bytes of parameters plus the displacement of the multiple returns. For example, subroutine MADP requires seven bytes of parameter information to follow the call to MADP. Therefore, an error return from MADP requires the value 7 to be loaded into the accumulator before RTRD is called; a value of 9 must be loaded into the accumulator before RTRD if there is no error.

7.4 MACROS

Observe in Figure 7-2(A) that an instruction sequence may reappear in a program each time it is reused. Such an instruction sequence may be identified as a macro.

Refer to Figure 7-2. If the instruction sequence represented by "s" is a subroutine (we will assume that it is named SUB1), then wherever the logic of SUB1 is required, a PI or PK instruction in the main program will cause execution to branch to one set of code, as illustrated in Figure 7-2(B) and described in Section 7.3. If, on the other hand, the logic of SUB1 is to be treated as a macro, then the name SUB1 will appear in the mnemonic field of the source program as though SUB1 were the mnemonic for an instruction. In the object program, the assembler will actually insert the sequence of instructions represented by SUB1 wherever SUB1 appears in the source program, as illustrated in Figure 7-2(A).

7.4.1 Defining and Using Macros

Beginning with a very simple example, suppose the instruction sequence which creates the correct carry status in multi-byte addition routines is to be identified as a macro named

CBIT, rather than as a subroutine named CBIT. The macro is defined in the source program by enclosing the instructions of the macro between assembler directives MACRO and MEND, as follows:

| | | | |
|-------|-------|--|------------------------------------|
| MACRO | | | |
| CBIT | | | |
| LR | A,9 | | MOVE STATUS FROM LNK ADDITION TO A |
| NI | H'02' | | MASK OUT ALL BUT C BIT |
| LR | J,W | | MOVE STATUS FROM BYTE ADD TO W |
| AS | 9 | | ADD STATUSES |
| LR | 9,A | | RETURN STATUS TO J VIA W |
| LR | W,J | | |
| MEND | | | |

In theory, a macro definition, as illustrated above, could appear anywhere in a source program; the assembler simply takes everything between the MACRO and MEND directives and holds it to one side, inserting the instructions whenever it sees the macro name appear in the mnemonic field of an instruction. In practice, it is good programming to collect macro definitions either at the very beginning or at the very end of a source program.

As an example of how a macro works, subroutine MADD could specify CBIT as a macro, rather than as a subroutine, as follows:

| | | | |
|---------------------------|------|------|--|
| | | | |
| | | | |
| | | | |
| (Body of subroutine MADD) | | | |
| | | | |
| | | | |
| EGTN | LR | DC,H | MOVE AUGEND ADDRESS BACK TO H |
| | LR | K,P | SAVE PC1 IN K |
| NNTN | CBIT | | INSERT INSTRUCTIONS FROM CBIT MACRO HERE |
| TWT1 | DS | 0 | DECREMENT COUNTER |
| | BNZ | LOOP | RETURN FOR MORE |
| TWT2 | PK | | RETURN FROM SUBROUTINE FOR END |

When the assembler assembles the above instruction sequence, instruction NNTN will be replaced directly by the six instructions listed between MACRO CBIT and MEND. For this reason, the programmer may look upon a macro simply as a short-hand method of writing source programs (i.e., a method of taking the tedium out of re-writing the same instruction sequence again and again).

7.4.2 Macros with Parameters

A simple macro, as illustrated for CBIT in Section 7.4.1, is of limited value; it makes an object program longer, but it makes writing the source program easier. The program executes faster since the PI and POP instructions are not executed.

Macros with parameters are more useful. Refer to subroutine MADP, in Section 7.3.6. In order to make the multi-byte addition program MADD useful, it was modified so that the call to subroutine MADD could be followed by seven bytes

of parameter data, including three 2-byte addresses and a single byte buffer length counter. Instructions at the beginning of subroutine MADP transfer these parameters to the H, Q and DC1 registers before performing the multibyte addition, thus allowing subroutine MADP to perform multibyte additions on the contents of buffers that can have any length and can be anywhere in memory.

The multibyte addition may also be specified as a macro, where the macro name is followed by a number of parameters. In this case, the parameters would again be three addresses and a byte count. Now when the assembler substitutes the instruction sequence of the multibyte add for the macro name appearing in an instruction mnemonic, it changes instructions within the sequence according to parameter specifications.

When a macro is defined, macro parameters are listed after the macro name with an ampersand as the first character of each parameter and one space separating parameters. This is illustrated for macro MADP below:

| | MACRO | | |
|-------|-------|------------------------|--------------------------------|
| | MADP | &VALA &VALB &VALC &CNT | |
| ONE | LI | &CNT | USE SCRATCHPAD REGISTER 0 |
| TWO | LR | O,A | AS A COUNTER |
| THREE | DCI | &VALC | SAVE THE ANSWER BUFFER |
| FOUR | LR | Q,DC | STARTING ADDRESS IN Q |
| FIVE | DCI | &VALA | SAVE THE SOURCE BUFFER |
| SIX | XDC | | ADDRESSES IN DC0 AND DC1 |
| SEVEN | DCI | &VALB | |
| EIGHT | COM | | INITIALLY CLEAR THE CARRY BIT |
| | LR | J,W | |
| LOOP | LM | | LOAD THE NEXT AUGEND BYTE |
| | LR | W,J | CARRY FROM PRIOR ADD TO STATUS |
| NINE | LNK | | ADD ANY PREVIOUS CARRY |
| TEN | LR | J,W | SAVE STATUS IN J |
| ELEV | XDC | | ADDRESS ADDEND BUFFER |
| TWEL | AM | | ADD CORRESPONDING ADDEND BYTE |
| THRT | XDC | | READDRESS AUGEND BUFFER |
| FRTN | LR | H,DC | SAVE AUGEND ADDRESS IN H |
| FFTN | LR | DC,Q | LOAD ANSWER BUFFER ADDRESS |
| SXTN | ST | | STORE THE ANSWER |
| EGTN | LR | DC,H | MOVE AUGEND ADDRESS BACK TO H |
| | BNC | TWT1 | NO CARRY FOR NEXT BYTE |
| TWY | LR | J,W | SAVE CARRY FROM AM INSTRUCTION |
| TWT1 | DS | 0 | DECREMENT COUNTER |
| | BNZ | LOOP | RETURN FOR MORE |
| | MEND | | |

Any program can tell the assembler to insert the instruction sequence specified by macro MADP, changing the symbols &CNT, &VALA, &VALB and &VALC to any four symbols specified in the operand field of the instruction that references macro MADP. For example, in order to reproduce the multibyte addition instruction sequence as illustrated in Section 7.2.2, the following instruction would have to appear:

```
MADP  BUFA BUFB BUFC CNT
```

When the assembler encounters the above instruction, it will substitute all of the instructions listed between MACRO MADD and MEND; however wherever it finds &CNT it will replace it with CNT, wherever it finds &VALA, &VALB or &VALC it will substitute BUFA, BUFB or BUFC, respectively.

7.4.3 Rules for Defining and Using Macros

The following few rules apply to the use and definition of macros:

- 1) No macro can be referenced in a program unless it has been defined as a macro, using the MACRO and MEND assembler directives.
- 2) When a macro is defined, it can reference another macro so long as the other macro is defined separately elsewhere.
- 3) If a macro is defined with parameters, then every time the macro is specified within the body of a program, the specification must have a valid symbol in the operand field, corresponding to every parameter in the macro definition.

7.4.4 When Macros Should be Used

There are two circumstances when macros are more efficient as a programming tool than subroutines.

Short instruction sequences that are frequently used within a program are often better represented as macros, if subroutine addresses are being maintained in a stack. It takes a certain amount of time to store a return address in a stack, then at the end of a subroutine to retrieve the address from the stack. If the body of the subroutine is quite short, the time taken to maintain the stack may become excessive. Under such circumstances it is better to reproduce the instruction sequence as a macro wherever it is needed within a program.

Subroutines which require a large number of parameters to be passed from the main subroutine are frequently better represented as macros; a considerable number of instructions may be needed to move the parameters from the parameter list that follows the subroutine call, to the registers or memory locations out of which the subroutine will access the parameters. Consider subroutine MADP; if this subroutine is called only two or three times it is probably more efficient to represent it as a macro rather than as a subroutine.

Macros always result in faster program execution than subroutines. Macros may result in longer programs than subroutines. Therefore, in an application where speed is important, macros should be used in preference to subroutines.

7.5 JUMP TABLES

A jump table is a programming device which is particularly useful in microprocessor applications. A jump table allows an index number to be loaded into the accumulator after which program execution jumps to a memory location which is dedicated to that index number.

Jump tables are commonly used in switching applications, where data may be received from, or control signals may have to be sent to, one of many external devices.

7.5.1 Jump Table Using Jump Instructions

The F8 instruction set is well-suited to execution of jump tables. As illustrated, one jump table may serve an entire application of diverse operations. The jump table consists of nothing more than a large number of jump instructions. To execute the jump table, a program simply loads an I.D. number into the accumulator, then jumps to the table logic. The table logic adds the contents of the accumulator, three times, to the address of the first jump instruction, which is stored in the DCO registers. The sum is moved (via the Q registers) to the program counter and the jump is effected.

*JUMP TABLE PROGRAM. JUMP NUMBER IS ASSUMED TO
*BE IN THE ACCUMULATOR.

| | | | |
|------|-----|------|--|
| JUMP | DCI | JMPO | LOAD THE FIRST JUMP ADDRESS INTO DCO |
| | ADC | | ADD THREE TIMES THE BRANCH |
| | ADC | | INDEX TO DCO, FOR THE THREE BYTES OF A JMP INSTRUCTION |
| | ADC | | |
| | LR | Q,DC | MOVE DCO TO PCO |
| | LR | P0,Q | JUMP OCCURS HERE |
| JMPO | JMP | A0 | |
| | JMP | A1 | |
| | JMP | A2 | |
| | JMP | A3 | |
| | JMP | A4 | |
| | | etc. | |

7.5.2 Jump Table Using Address Constants

Another jump table technique uses a table of addresses which are indexed as in the previous example. However, instead of a JUMP (LR P0,Q) to the jump table, the address is loaded from memory into Q. The LR P0,Q instruction then causes a direct jump to the address in Q. The major advantage of this technique is that the table is only two bytes per entry, as compared to three bytes in the previous example. It also executes using fewer instruction cycles.

*THE JUMP NUMBER IS ASSUMED TO BE IN THE
*ACCUMULATOR

| | | | |
|------|-----|------|---|
| JUMP | DCI | JMPO | LOAD THE FIRST JUMP ADDRESS INTO DCO |
| | ADC | | ADD TWICE THE JUMP NUMBER. DCO NOW ADDRESSES A JUMP ADDRESS |
| | ADC | | LOAD FIRST BYTE OF JUMP ADDRESS |
| | LM | | |
| | LR | QH,A | STORE IN QH |
| | LM | | LOAD SECOND BYTE OF JUMP ADDRESS |
| | LR | QL,A | STORE IN QL |
| | LR | P0,Q | BY MOVING Q TO P0, FORCE JUMP |
| JMPO | DC | A0 | |
| | DC | A1 | |
| | DC | A2 | |
| | DC | A3 | |
| | DC | A4 | |
| | | etc. | |

7.5.3 Jump Table Using Displacement Tables

Under some circumstances the addresses of the jump table may all be within 256 bytes of each other. When this situation exists, only a displacement need be created in the table. This displacement, when added to some base address, will produce the address required for the jump. Notice that in the following example, entry FOUR and FIVE go to the same location. This is a variation that is quite useful. Perhaps the values 4 and 5 are invalid and an error routine needs to be called. The jump table will satisfy this condition in a most efficient manner without a separate compare instruction for each invalid value. Also notice that the entry points need not be in any particular sequence; however, in this example A1 must be the first entry point encountered, and it must have the lowest address in order for the arithmetic to be valid.

This displacement table is most efficient since the table values are only one byte each. If an entry is beyond the 256 range it is possible to treat it as a special case within the table. Notice that A6 is more than 256 bytes beyond the start of A1 and is too large to insert before A4. To include it insert a JMP A6 prior to the coding at A4. If this instruction is labeled A66, an entry in the table would be:

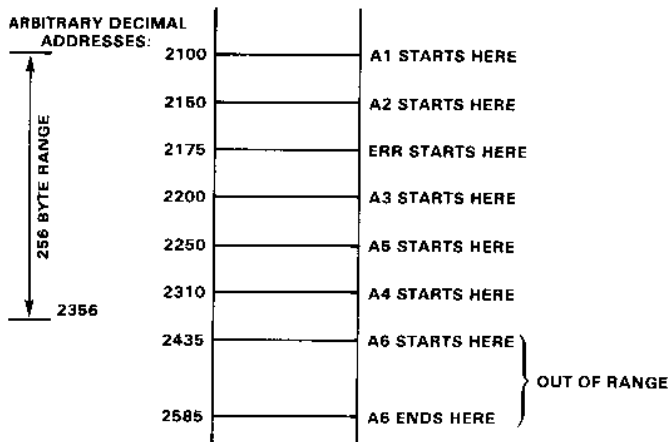
SIXS DC (A66-A1-128) = 82

The value of THRE would now become 85.

*THE JUMP NUMBER IS ASSUMED TO BE IN THE
*ACCUMULATOR

| | | | |
|------|-----|--------------|------------------------------------|
| JUMP | DCI | ZERO | LOAD FIRST TABLE LOCATION INTO DCO |
| | ADC | | ADD VALUE FROM ACCUMULATOR |
| | LM | | LOAD TABLE VALUE TO ACCUMULATOR |
| | DCI | (A1+128) | LOAD FIRST ORIGIN |
| | ADC | | ADD DISPLACEMENT VALUE ADDED TO DC |
| | LR | Q,DC | RECALL DC TO Q |
| | LR | P0,Q | JUMP TO ROUTINE |
| ZERO | DC | (A1-A1-128) | VALUE = -128 |
| ONE | DC | (A2-A1-128) | VALUE = - 78 |
| TWO | DC | (A5-A1-128) | VALUE = 22 |
| THRE | DC | (A4-A1-128) | VALUE = 82 |
| FOUR | DC | (ERR-A1-128) | VALUE = - 53 |
| FIVE | DC | (ERR-A1-128) | VALUE = - 53 |
| SIX | DC | (A3-A1-128) | VALUE = - 28 |
| SVEN | DC | (A6-A1-128) | VALUE = 207 TOO LARGE! |

Values are displaced by -128 to take into account the fact that the DCI instruction points to the middle of the table (A1+128). Therefore, addresses are created as shown on the following page.



7.6 STATUS, BITS AND BOOLEAN LOGIC

The F8 instruction set is rich in boolean logic instructions which are very useful in applications manipulating bits and control lines.

Examples given in the following subsections demonstrate some elementary uses of boolean logic instructions, along with some less obvious but commonly needed routines.

7.6.1 Manipulating Individual Bits

Immediate boolean instructions specify data in the operand of the instruction; they may be used to set or reset individual bits within the accumulator.

To reset one or more bits within the accumulator, AND the accumulator contents with a mask which is the complement of the bits to be reset. For example, the following instructions will reset bit 3 of scratchpad byte 1:

```
LR    A,1    LOAD SCRATCHPAD BYTE
          1 INTO A
NI    H'F7'  MASK OUT BIT 3
LR    1,A    RETURN TO SCRATCHPAD
          BYTE 1
```

Similarly, individual bits can be set by ORing the accumulator with a mask which has a 1 in every bit position that is to be set. For example, bit 3 of scratchpad byte 1 contents can be set to 1 as follows:

```
LR    A,1    LOAD SCRATCHPAD BYTE
          1 INTO A
OI    H'04'  SET BIT 3
LR    1,A    RETURN TO SCRATCHPAD
          BYTE 1
```

Masks may also be accessed out of RAM or scratchpad memory. The following instruction sequence takes every byte from a buffer CNT bytes long, starting at BUFA; it sets to 0 the bits specified by a mask stored in a memory byte addressed by MASK. BUFA, CNT and MASK are symbols which have been given arbitrary values below.

```
BUFA EQU H'2380'
MASK EQU H'08FF'
CNT EQU 50
—
—
—
```

```
DCI MASK STORE THE MASK ADDRESS
      IN H
LR H,DC
DCI BUFA STORE THE BUFFER
      STARTING ADDRESS IN Q
LR Q,DC
LI CNT USE SCRATCHPAD BYTE
      0 AS A
      COUNTER
LOOP LR 0,A
L1 LM LOAD NEXT BYTE
L2 LR DC,H LOAD MASK ADDRESS
      AND ACCUMULATOR WITH
      MASK
L3 LR DC,Q RELOAD BYTE ADDRESS
L4 ST STORE MASKED BYTE IN
      ORIGINAL BYTE POSITION
L5 LR Q,DC SAVE INCREMENTED
      BUFFER ADDRESS IN Q
L6 DS 0 DECREMENT COUNTER
L7 BNZ LOOP RETURN FOR MORE
```

In addition to demonstrating use of the NM instruction, the above example shows how to process data in a single buffer, restoring a modified byte to its original byte position.

The program proceeds as follows:

The instructions preceding LOOP load the mask address into H and the beginning buffer address into Q. The buffer length is loaded into scratchpad byte A which is used as a counter.

```
LOOP The data counter holds the initial buffer address
      when this instruction is first executed and the next
      byte address on all subsequent executions of this
      instruction. This instruction therefore loads the
      next byte from BUFA.
L1 Load the mask address from the H registers into
      the data counter, wiping out the incremented buffer
      address that resulted from instruction LOOP.
L2 AND the contents of the accumulator with the
      mask byte. The fact that the AND with memory
      instruction will increment the address in the data
      counters is not consequential since this incremented
      address is not saved. On the next execution of
      this instruction, the original mask address stored
      in the H registers will be reused.
L3 Reload the buffer address from the Q registers.
      This is the same address that was used by instruction
      LOOP.
L4 Store the contents of the accumulator back in the
      buffer. Since the address loaded by L3 is the same
      address as was used by instruction LOOP, the
      masked byte will be stored back in the same memory
      location from which it was loaded.
L5 This time save the incremented address in the
      data counters back in the Q registers.
L6 Decrement the counter in scratchpad byte 0. If
      the result is zero, end. If the result is not zero
      process the next byte of the buffer.
L7
```

By using the NM instruction, the above example is resetting (to 0) selected bits from every byte in BUFA. By merely replacing the NM instruction with an OM instruction, selected bits from every byte of BUFA could be set to 1.

By storing the mask byte in a scratchpad register, the program can be greatly simplified. The instruction sequence below is similar to the previous example, but the mask byte is stored in scratchpad register 1, and the DC1 registers are used to hold the buffer address, rather than the Q registers.

Notice that at the LM instruction (LOOP), DC0 is incremented; prior to the ST instruction, DC0 and DC1 are exchanged. The ST instruction then increments DC0, thus both addresses remain synchronized.

```

MASK EQU B'any binary value'
BUFA EQU H'2380'
CNT EQU 50
—
—
—
ONE LI MASK
TWO LR 1,A
DCI BUFA STORE BUFFER ADDRESS
XDC IN DC0 AND IN DC1
REGISTERS
DCI BUFA
LI CNT USE SCRATCHPAD BYTE 0
LR 0,A AS A COUNTER
LOOP LM LOAD NEXT BYTE
THREE NS 1 AND WITH MASK IN
SCRATCHPAD BYTE 1
XDC
ST STORE IN ORIGINAL
LOCATION
DS 0 DECREMENT COUNTER
BNZ LOOP RETURN FOR MORE

```

Again this routine can be simplified even further by deleting instructions ONE and TWO and changing instruction THREE to one of the following:

```

NI MASK AND WITH MASK
OI MASK OR WITH MASK
XI MASK EXCLUSIVE OR WITH MASK

```

This change would result in saving two bytes, however the loop time would be increased by 1.5 cycles.

7.6.2 Testing for Status

The EXCLUSIVE-OR instruction is very useful as a means of detecting changed statuses. There are many applications in which it will be necessary to keep a record of status for various control lines, and to detect when individual control line statuses change and how they change. As illustrated in the instruction sequence below, eight control lines have their statuses maintained in scratchpad byte 3. When new statuses are input from I/O port 0, they are temporarily saved in scratchpad byte 4. By EXCLUSIVE-ORing the new and old statuses, the accumulator identifies those status bits which have changed. By ANDing the changed status indicators with the old status, those indicators which went from "on" to "off" are identified. By EXCLUSIVE-ORing this result with the changed status indicators, those statuses which went from "off" to "on" are identified.

```

IN 0 INPUT NEW STATUS
S2 LR 4,A SAVE IN SCRATCHPAD
BYTE 4
S3 XS 3 EXCLUSIVE-OR ACCUMU-
LATOR WITH OLD STATUS
S4 LR 5,A SAVE "CHANGED
STATUSES" INDICATORS
IN 5
S5 NS 3 AND WITH OLD STATUSES
S6 LR 6,A SAVE "STATUSES TURNED
OFF" IN 6
S7 XS 5 EXCLUSIVE-OR WITH
"CHANGED STATUSES"
S8 LR 7,A SAVE "STATUSES TURNED
ON" IN 7
S9 LR A,4 NEW STATUS FROM SAVE
S10 LR 3,A OLD STATUS FROM NEXT
USAGE.

```

Suppose the old status was:

```

7 6 5 4 3 2 1 0 Bit No.
Old Status = 1 0 1 1 1 0 0 0

```

Suppose the new status is:

```

7 6 5 4 3 2 1 0 Bit No.
New Status = 1 1 0 1 0 1 1 0

```

Bits 6, 2 and 1 have turned on.
 Bits 5 and 3 have turned off.
 Bits 6, 5, 3, 2 and 1 have changed.

Here is the result of instruction S3:

```

7 6 5 4 3 2 1 0 Bit No.
Old Status 1 0 1 1 1 0 0 0
New Status 1 1 0 1 0 1 1 0
Changed Statuses 0 1 1 0 1 1 1 0

```

Here is the result of instruction S5:

```

7 6 5 4 3 2 1 0 Bit No.
Changed Status 0 1 1 0 1 1 1 0
Old Status 1 0 1 1 1 0 0 0
Turned Off 0 0 1 0 1 0 0 0

```

Here is the result of instruction S7:

```

7 6 5 4 3 2 1 0 Bit No.
Turned Off 0 0 1 0 1 0 0 0
Changed Statuses 0 1 1 0 1 1 1 0
Turned On 0 1 0 0 0 1 1 0

```

7.7 POWERING UP AND STARTING PROGRAM EXECUTION

When power is turned on, all PC0 registers in an F8 microprocessor system are set to 0. Therefore the first instruction executed is located at memory byte 0.

Every F8 microprocessor system must, therefore, have a memory device (either a 3851 PSU, 3852 DMI or 3853 SMI). The first program to be executed must be originated at H'00', as illustrated on the following page.

| | | | |
|-------|-----|-------|-------------------|
| | ORG | H'00' | |
| START | — | | FIRST INSTRUCTION |
| | — | | EXECUTED |

The power on detect circuit for an F8 system is located in the CPU. This circuit insures that all critical control circuits and registers are in a valid operating condition when power is first applied. It performs the following functions:

- Pushes previous contents of the program counter to the stack register
- Resets the program counter to address "0000"

- Resets the Interrupt Control Bit (ICB)
- Sets control block on the 3852 MI circuit

When power is connected to the circuit or the reset line goes low, the CPU clears the program counter (PC0), pushing its previous contents into the stack register (PC1). Therefore, the instruction in location zero is executed first. The interrupt control bit is also cleared at this time. The rest of the F8 system is initialized under program control. The local interrupt block of the individual memory devices must be loaded before allowing any interrupts to occur. Output latches must be reset to zero before they may be used to input data.

INPUT/OUTPUT PROGRAMMING

Input/output programming covers program steps that cause data to be transferred between the F8 microprocessor system and the world beyond the microprocessor system.

There are three separate and distinct types of input/output (I/O) programming: Programmed I/O, Interrupt I/O and Direct Memory Access (DMA).

Programmed I/O is characterized by the 3850 CPU executing an instruction to initiate and control the I/O transfer of a single byte of data, via an I/O port. The key feature of programmed I/O is that it is initiated by the CPU, on a byte-by-byte basis.

Interrupt I/O is characterized by an external device issuing an interrupt to the 3850 CPU; (this concept is discussed in Section 2.2.2). The interrupt does not itself cause any input or output data transfer to occur; rather it initiates execution of a program which performs any required programmed I/O.

DMA has been described conceptually in Sections 2.2.4, 2.6.3 and 2.8. DMA transfers data between a memory device within the microprocessor system and any device external to the microprocessor system, in parallel with other microprocessor operations. DMA is initiated using programmed I/O and, optionally, may terminate with an interrupt.

The use of software clocks is also covered in this chapter. Even though software clocks have nothing to do with transfer of data between the microprocessor system and the outside world, they do allow events within the microprocessor system to be synchronized with real time.

8.1 PROGRAMMED I/O

A programmed input or output operation moves a byte of data from the 3850 CPU accumulator either to an I/O port (OUT), or from an I/O port to the accumulator (IN).

Four instructions enable programmed I/O: INS and IN enable input, while OUTS and OUT enable output. (See Sections 6.16, 6.18, 6.33 and 6.34.)

Note that a number of I/O ports are accessed by I/O instructions, but transfer no data between the microprocessor system and the outside world. These I/O ports hold control information used by interrupt I/O, DMA and real time clocks. Section 6.16 summarizes the I/O port addresses used by the F8, and indicates how the individual port addresses may be used.

Programmed I/O is a very open ended subject, since it is dependent on how external circuitry accesses the I/O ports. The following subsections describe some general approaches to I/O programming as seen by the CPU. Actual applications will usually require modified versions of the given programming techniques.

8.1.1 Polling on Status

A key feature of programmed I/O is that the microprocessor system and external devices operate at different speeds; the external device must transfer data at a rate which is slower than the I/O program's execution speed.

The simplest way of handling programmed I/O, when external devices run slower than the microprocessor, is to have the external device input a "status byte" to the I/O port when it is

ready to transmit or receive data. The 3850 CPU continuously inputs a byte of data from the port until the "ready status" appears. For example, suppose a 1 in the high order bit (bit 7) of the I/O port signifies a ready status; the following routine will input a byte of data via port 0:

***ROUTINE TO INPUT A BYTE OF DATA VIA PORT 0, POLLING
*ON STATUS TO SYNCHRONIZE WITH THE EXTERNAL
*DEVICE.**

| | | | |
|------|------|------|--|
| INO | LIS | 0 | FIRST CLEAR THE PORT |
| | OUTS | 0 | |
| LOOP | INS | 0 | INPUT STATUS |
| | BP | LOOP | RETURN IF BIT 7 IS 0 |
| L0 | INS | 0 | BIT 7 IS 1. INPUT A DATA BYTE |
| | ST | | STORE IN MEMORY BYTE ADDRESSED BY DCO |
| L1 | PI | TEST | BRANCH TO END OF INPUT TEST |
| L2 | BR | LOOP | RETURN FROM TEST FOR MORE INPUT |
| L3 | | | RETURN FROM TEST FOR END |

Three features of the above routine need to be explained. The first two instructions clear the output port. This is necessary because data being input at an I/O port is ORed with whatever is already in the port. If, by chance, the high order bit of the last data byte input was 1, this would be interpreted as a ready status.

The data which is input to the accumulator by instruction LOOP will be interpreted as a byte of status. While in this simple application only the high order bit of the status byte is being interrogated, in any real application all eight bits of the status byte could be assigned meaning. In this case, when bit 7 of the status byte is tested to be 1, a byte of data is input by instruction L0 to the accumulator. This routine assumes that the time delay between execution of instructions LOOP and L0 is sufficient for the external device to transmit a data byte.

This routine assumes that an indeterminate number of characters are expected on input. A subroutine named TEST is called to determine if more bytes of data are expected. The operations performed by subroutine TEST are immaterial to the I/O routine. Subroutine TEST must have two returns; to instruction L2 if another byte of data is to be input, or to instruction L3 if data input is complete.

Each byte of data that is input to the accumulator in subroutine INO must be stored in some read/write memory location. INO assumes that the DCO registers address a RAM byte into which the data must be stored. This assumes that before INO is called as a subroutine, the beginning address of a RAM data buffer is loaded into the DCO registers. Data bytes, as they are input, will be stored in ascending bytes of the addressed RAM data buffer. Scratchpad bytes can also be used to hold data being input.

Subroutine OUTO, described below, is a variation of subroutine INO. OUTO outputs data from a RAM buffer. The only difference between subroutines INO and OUTO is that in OUTO, once a ready status has been detected, the data byte which

is to be output must first be transferred from memory to the accumulator before being output to port 0.

Both subroutines IN0 and OUT0 can address any port that the INS and OUTS instructions can address. In order to address other ports it is only necessary to replace the INS and OUTS instructions with IN and OUT instructions.

*ROUTINE TO OUTPUT A BYTE OF DATA VIA PORT 0, POL-
*LING ON STATUS TO SYNCHRONIZE WITH THE EXTERNAL
*DEVICE

```

OUT0  LI    0      FIRST CLEAR THE PORT
      OUS    0
LOOP1 INS    0      INPUT STATUS
      BP    LOOP1  RETURN IF BIT 7 IS 0
      LM                      BIT 7 IS 1. READ FROM
                              MEMORY THE BYTE TO BE
                              OUTPUT
M0    OUS    1      OUTPUT THE DATA BYTE
M1    PI    TEST   BRANCH TO END OF INPUT
      TEST
M2    BR    LOOP1  RETURN FROM TEST FOR
      MORE INPUT
M3                      RETURN FROM TEST FOR
      END

```

8.1.2 Data, Status and Controls

Observe that in Section 8.1.1, a byte input by an external device may be interpreted as status information or as data. Similarly, the 3850 CPU may output a byte which is to be interpreted as control signals or as data.

To illustrate, consider an F8 microprocessor system being used to read data input from a keyboard, block the data into 256 byte records, then write the records out to a cassette. Events would proceed as follows:

- 1) Using a programmed input sequence such as IN0, interpret a byte input from the keyboard as status. When a ready status is sensed, interpret the next byte arriving from the keyboard as data.
- 2) A subroutine such as TEST is called to create a 256 byte record in RAM, in the format needed for output to the cassette.
- 3) When the microprocessor is ready to write a record to the cassette, it must first turn the cassette drive motor on, since the cassette drive will be stationary during the intervals when records are not being written out. The microprocessor will turn the cassette drive on by outputting an appropriate control byte whose bit pattern is determined by the specifications of the cassette drive controller.
- 4) The cassette drive will respond to the control byte, commanding the drive be turned on by transmitting back a status byte indicating that the command was successfully executed and the drive is now ready to receive data.
- 5) Upon receiving back the ready status from the cassette drive the microprocessor will output 256 bytes of

data. Depending on the design of the cassette drive, the cassette drive controller may transmit a status byte back to the microprocessor after each individual data byte has been received. This status byte reports that the previous data byte has been recorded accurately, and the controller is ready to receive and record the next byte of data.

- 6) After the microprocessor has completed transmittal of an entire record of data, it must send a control signal to the cassette drive commanding the cassette drive to stop forward movement.
- 7) When all records have been written to the cassette drive, the microprocessor will issue a third control command which causes the cassette drive to mechanically rewind.

Observe that this simple application receives either data or status from the keyboard, then outputs either controls or data to the cassette drive; additional status information may come back from the cassette drive.

Any external device may transmit two types of information to the microprocessor system: data or status.

Any external device may receive two types of information from the microprocessor system: data or controls.

Thus there are four types of information that may be transferred between the microprocessor system and an external device. They are:

- a) data in
- b) status in
- c) data out
- d) control out

An external device may communicate with the microprocessor system using one, two, three or all four of the above types of information. For example, the keyboard uses "data in" and "status in" but does not use "data out" or "controls out". The cassette drive in the illustrated application uses "status in", "data out" and "controls out" but does not use "data in". Of course the cassette drive would be capable (at another time) of using "data in", when the data which was recorded on the cassette is subsequently read back into the microprocessor system.

It is feasible to use one port for all four of the information transfer types listed above when communicating with any one external device. For example, one I/O port could be used to receive status and data from the keyboard, and could also be used to receive status or data from the cassette drive and to output controls or data to the cassette drive. However, if more than one type of information is to go through one I/O port, external logic must have the means of multiplexing information in or out. A scheme that uses more I/O ports, but less external logic, allocates one port for data in or out and another port for status in or controls out. For example, I/O port 0 may be assigned to keyboard status in, I/O port 1 may be assigned to keyboard data in, I/O port 4 may be assigned to cassette status in and controls out and I/O port 5 may be assigned to cassette data in and out.

8.1.3 Parallel Data and Control Ports

Many applications will require data to be handled on paths that are more than eight bits wide. Sixteen-bit data, for example, is a common word size. Less frequently, it will be necessary to handle more than eight control lines at a time.

Data paths that are more than eight bits wide can be handled in 8-bit units, sequentially through a single port. Alternatively, two or more ports may be assigned to one external data bus so that, whenever the microprocessor inputs data from an external device or outputs to the external device, it accesses each I/O port allocated to the data bus. This is illustrated below in subroutine IN16, which inputs data in 16-bit units via ports 4 (bits 0-7) and port 5 (bits 8-15).

```
*ROUTINE TO INPUT 16 BITS OF DATA VIA PORTS 4 AND 5,  
*POLLING ON STATUS VIA PORT 0 TO SYNCHRONIZE WITH  
*THE EXTERNAL DEVICE  
IN16   LIS      0      FIRST CLEAR THE STATUS  
                                PORT TO REMOVE PREVIOUS  
                                READY STATUS  
                                LOOP  
                                INS      0      INPUT STATUS  
                                BP      LOOP    RETURN IF BIT 7 IS 0  
L0     INS      4      BIT 7 IS 1. INPUT FIRST DATA  
                                BYTE  
                                ST          STORE IN MEMORY BYTE  
                                ADDRESSED BY DCO  
                                INS      5      INPUT SECOND DATA BYTE  
                                ST          STORE IN NEXT MEMORY  
                                BYTE (ADDRESSED BY DCO)  
L1     PI      TEST    BRANCH TO END OF INPUT  
                                TEST  
L2     BR      LOOP    RETURN FROM TEST FOR  
                                MORE INPUT  
L3     RETURN FROM TEST FOR NO  
                                MORE INPUT
```

8.2 INTERRUPT I/O

Two circumstances under which interrupts are commonly used to control I/O operations are:

The programmed I/O, described in Section 8.1, has the severe disadvantage that the microprocessor system spends a great deal of its time reading a status byte and waiting for the status byte to signal "ready". If the external device operates at speeds close to that of the microprocessor, the wasted time may be unavoidable. For example, if the microprocessor can only execute ten instructions between each byte transmitted or received by the external device, it is probable that these ten instructions can be effectively used testing or processing each data byte as it is transferred. On the other hand, if the 3850 CPU can execute approximately one hundred instructions between bytes of data being transmitted to or from the external device, there is sufficient time between data transfers for the microprocessor to be doing other useful work which may or may not be related to the data transfer taking place. If instead of sending a ready status, the external device transmits an interrupt request signal every time it is ready to transmit or receive a data byte, this signal can be used by the 3850 CPU to suspend executing whatever program was being executed, process a single byte of data, then return to the suspended program.

The transfer of a sequence of data bytes at a known data rate constitutes a sequence of predictable events. In many applications an external device's need for access to the micro-

processor system cannot be predicted. For example, an external device may only communicate to the microprocessor under distress circumstances, at which time the microprocessor must execute a program to compute and output needed correction data. When the external device's need for access to the microprocessor system cannot be predicted, an interrupt is the only reasonable way in which the external device can gain control of the microprocessor system.

8.2.1 The Interrupt Sequence

Each 3851 PSU in an F8 microprocessor system has an external interrupt line, as does the 3853 SMI device, if present.

The sequence of events surrounding an interrupt is as follows:

- 1) For interrupts to be processed, interrupts must be enabled within the 3850 CPU and at the device receiving the interrupt request signal. At the 3850 CPU, all interrupts are enabled or disabled. At each 3851 or 3853 device, the individual interrupt line at that device is enabled or disabled. This is described in Section 8.2.2.
- 2) More than one device may simultaneously request to interrupt the 3850 CPU; that is, interrupt request signals may be true, simultaneously, at more than one device. When this happens, priorities are arbitrated as described in Section 8.2.3.
- 3) When a valid interrupt request signal is detected by the 3850 CPU, it ceases current program execution at the conclusion of the instruction currently being executed. (Certain instructions are exempt, as described below.)
- 4) The 3850 CPU sends out an interrupt acknowledge signal. The way in which this signal is trapped implements interrupt priority when more than one interrupt request line is true, as described in Step 2.
- 5) When the 3850 CPU sends out an interrupt acknowledge signal, it clears the interrupt enable status within the 3850 CPU, thus disabling all subsequent interrupts. As described in Section 8.2.4, interrupts must be re-enabled, under program control, when such a step is appropriate to program logic.
- 6) Each device that has an interrupt request line also has a 16-bit address register which holds the address of the first instruction to be executed following the interrupt. The 3851 address register is a non-programmable mask option. The 3853 address register is made up of two I/O ports which are loaded with an address by appropriate I/O instructions. As described in Section 8.2.4, bit 7 of the interrupt address will always be 1 for an external interrupt, and will always be 0 for a local timer interrupt.

The device that traps the interrupt acknowledge signal output in step 5 responds by transmitting the contents of its interrupt address register as the next contents of PC0 registers.
- 7) PSU and MI logic, under CPU control, moves the contents of PC0 to PC1, then loads the address from step 6 into PC0; thus a program dedicated to the acknowledged interrupt request line is executed.

An interrupt will not be acknowledged at the conclusion of any of the following instructions:

```
PK
PI
POP
JMP
OUTS (if not port 0 or 1)
OUT (if not port 0 or 1)
EI
LR      W,J
```

An instruction other than one of the above must be executed before an interrupt will be acknowledged.

When power is first turned on, interrupts are disabled.

8.2.2 Enabling and Disabling Interrupts

As described in Section 2.4.3, bit 4 of the 3850 CPU W register is an Interrupt Control bit. When this bit is set to 1, interrupt requests to the CPU are enabled; when this bit is reset to 0, no interrupt request to the CPU will be acknowledged. ICB is set to 1 by the EI instruction or by a LR W,J instruction; it is reset to 0 by the DI instruction or by a LR W,J instruction.

Individual interrupt request lines are controlled at each device via an I/O port which is set aside as an interrupt control buffer.

For the 3851 PSU's, the interrupt control I/O port address is B'xxxxxx10'; xxxxxx is the I/O port select code, which may be any number from 1 to H'3F'. The 3853 interrupt control I/O port address must be H'0E'. This address is also available on a 3851 PSU; when xxxxxx is H'03', the 3851 interrupt control I/O port address becomes H'0E':

$$B'xxxxxx10' = B'00001110' = H'0E'$$

When a 3853 SMI device is present, a 3851 PSU with a chip select of H'03' cannot also be present.

The following two instructions load the interrupt control I/O port:

```
LI      VAL
OUT     IPRT
```

IPRT must be equated to the interrupt control I/O port address.

VAL must be equated as shown in Table 8-1.

| Value VAL is equated to | Effect |
|-------------------------|--|
| H'00' | Interrupts disabled at this device. |
| H'01' | External interrupt enabled, timer interrupt disabled. |
| H'02' | Interrupts disabled at this device (same as H'00'). |
| H'03' | External interrupts disabled, timer interrupt enabled. |

Table 8-1. Contents of Interrupt Control I/O Ports

Timer interrupts are described in Section 8.4.

8.2.3 Interrupt Priorities

When an F8 microprocessor system has more than one interrupt line, priorities are determined on the basis of "daisy chaining", as illustrated in Figure 8-1.

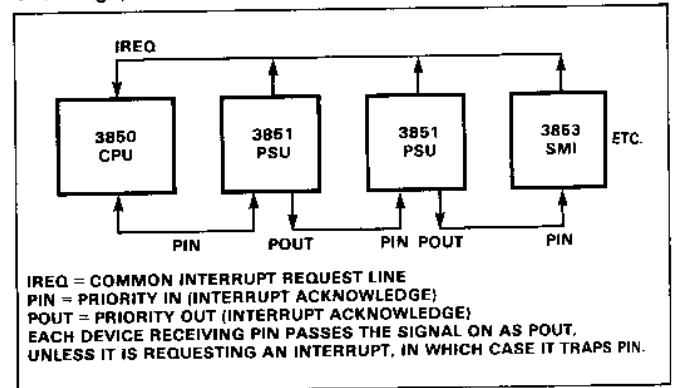


Fig. 8-1. Daisy Chaining and Interrupt Priority Determination

The daisy chain sequence is a hardware feature of an F8 microprocessor system; when the system is configured, the interrupt acknowledge signal from the CPU is chained from one device to the next. This determines interrupt priorities.

The only thing a programmer can do to modify interrupt priorities is to disable external interrupts at selected devices by appropriately loading the interrupt control I/O port at that device with some value other than H'01'. (See Section 8.2.2 and Table 8-1.)

It should be clearly understood that interrupt priorities, as described in this section, apply only to interrupt request signals competing for the 3850 CPU's next interrupt service.

There is nothing to prevent an interrupt from interrupting a previous interrupt; however, this type of nested priority is a function of how programs have been written. Once an interrupt has been acknowledged and is being serviced, and the ICB bit in the CPU is set to 1, the current interrupt service routine can itself be interrupted.

In order to prevent an interrupt service routine from being itself interrupted, the ICB bit in the CPU W register must be left at zero until the interrupt service routine has completed execution.

Figure 8-2 illustrates the concept of nested interrupts.

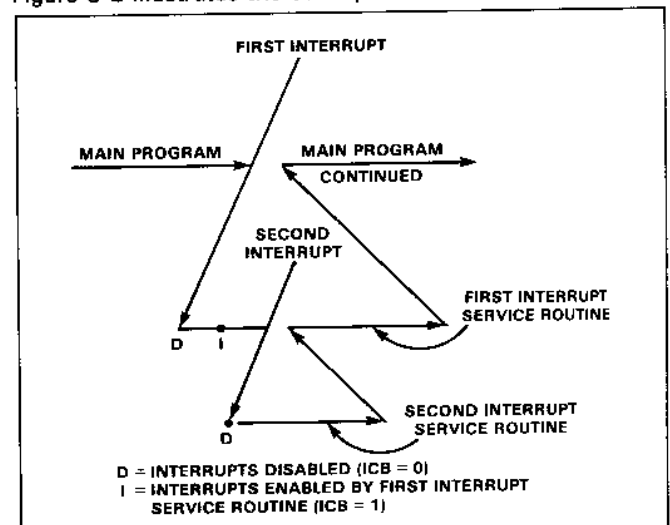


Fig. 8-2. Two Levels of Interrupt

The 3853 SMI device will not pass on an interrupt acknowledge signal; therefore, it must be at the end of the daisy chain, and will have lowest interrupt priority.

8.2.4 Program Response to an Interrupt

There are three program steps which may be needed prior to an interrupt in order to prepare to receive interrupts. They are:

- 1) If a 3853 SMI device is present, the interrupt address register of the 3853 must be loaded with the address of the first instruction to be executed after an interrupt from the 3853 is acknowledged. As described in Sections 2.7 and 6.16, I/O port addresses H'0C' and H'0D' have been reserved for the upper and lower interrupt address bytes, respectively; therefore the post-interrupt execution address can be loaded as follows:

```
LI    ADHI
OUTS  H'0C'
LI    ADLO
OUTS  H'0D'
```

ADHI and ADLO are symbols which must be equated to the high and low bytes of the selected execution address. Note that the 3851 PSU has the post-interrupt execution address as a permanent feature of the chip mask; therefore, each 3851 PSU has a fixed post-interrupt execution address associated with it.

- 2) Interrupts must be selectively enabled or disabled at 3851 and 3853 interrupt control ports, as described in Section 8.2.3.
- 3) The 3850 CPU master interrupt enable bit (ICB) must be set to 1, as described in Section 8.2.3.

When an interrupt is acknowledged, events within the 3850 CPU proceed exactly as if a subroutine had just been called: the content of PC0 is moved to PC1, and the content of the selected device's post-interrupt address register is moved to PC0. Interrupts should therefore be handled as though a subroutine had just been executed, as described in Section 7.3. For example, the first instructions executed following an interrupt might be:

```
LR    K,P    SAVE CONTINUATION
                ADDRESS IN K
PI    CALL    SAVE CONTINUATION
                ADDRESS IN STACK
```

Returning from an interrupt to the interrupted program is identical to returning from a subroutine to the calling program; however, since a program may be interrupted any time interrupts have been enabled, parameter passing and multiple returns do not apply to post-interrupt programs and should not be used.

Remember that the first interrupt service routine must enable ICB if second level interrupts are to be allowed (as illustrated in Figure 8-2).

8.2.5 Making 3851 PSU Interrupt Address Programmable

The fact that the 3851 PSU's interrupt address is a permanent feature of the device is not a problem in applications where this address may have to be varied. Using a branch table (as described in Section 7.5), a number of possible post-interrupt service routine execution addresses may be maintained. The following routine shows how an external device may use a PSU I/O port to provide an index identifying the service routine which must be executed following the interrupt. I/O port 4 has been arbitrarily selected as the I/O port address. The data byte at I/O port 4 selects an address from a branch table, as follows:

```
*POST INTERRUPT SERVICE ROUTINE FOR PSU 1
RC1I  LR      K,P    SAVE RETURN ADDRESS ON
                THE STACK
                PI    CALL
                INS   4    INPUT PROGRAM SELECT
                BYTE
                LR    RX    SAVE INDEX VALUE
                PI    BRANCH CALL BRANCH TABLE SUB-
                ROUTINE
```

8.2.6 Simple I/O Interrupts

In Section 8.1.2, a simple application was described, where data is input at a keyboard and recorded in 256 byte records on a cassette.

A cassette may record data at a rate of approximately 200 bytes/second. With time taken to start and stop the cassette, two or three seconds may elapse each time a record is output to the cassette. Preventing data from being input at the keyboard while it is being output to a cassette is both inconvenient and unnecessary. Simple I/O interrupts may be used to output data to the cassette, byte-by-byte. These few instructions are sufficient to service each interrupt.

*PROGRAM TO WRITE ONE BYTE TO A CASSETTE, FOLLOWING AN INTERRUPT

```
CRW   LM                LOAD NEXT BYTE
                ADDRESSED BY DC0
                OUT   CASS  OUTPUT TO CASSETTE
                EI
                POP                RETURN FROM INTERRUPT
```

The key concept here is that the F8 is uniquely suited to processing a large number of simple interrupts. If the post-interrupt program will not itself be interrupted, and if it will call no subroutines, then merely ending it with a POP instruction turns it into a complete interrupt service routine. Do not save the return address in the stack; do not call any starting or ending subroutines (e.g., CALL or RTRN).

For example, see Section 2.8.7.

8.2.7 A Sample Program

Figure 8-3 illustrates a configuration for the key to cassette application described in Section 8.1.2, except that 32 byte records are to be written to the cassette.

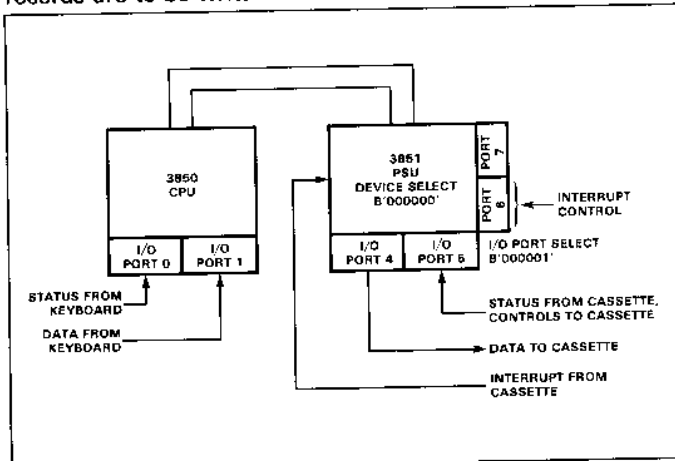


Fig. 8-3. Two Devices Servicing a Keyboard to Cassette Application

*PROGRAM TO RECEIVE DATA FROM THE KEYBOARD USING
 *PROGRAMMED I/O
 *SCRATCHPAD BYTES 0'40' TO 0'77' MAKE UP THE 32 BYTE
 *BUFFER.
 *SCRATCHPAD BYTES 0'20' TO 0'37' ARE USED AS A TEM-
 *PORARY BUFFER TO HOLD DATA WHILE THE MAIN BUFFER
 *IS BEING WRITTEN TO CASSETTE

| | | | |
|-------|------|---------|---|
| START | ORG | H'0000' | INITIALIZE ISAR TO |
| S1 | LISU | 3 | TEMPORARY BUFFER |
| S2 | LIS | H'01' | ENABLE EXTERNAL INTER- |
| | | | RUPTS AT PSU |
| | OUTS | 6 | |
| | EI | | ENABLE INTERRUPTS |
| S3 | PI | INKB | INPUT NEXT EIGHT BYTES |
| | | | FROM KEYBOARD |
| S5 | LISU | 2 | DECREMENT UPPER DIGIT |
| | | | OF ISAR |
| S6 | PI | INKB | INPUT NEXT EIGHT BYTES |
| | | | FROM KEYBOARD |
| | | | *AFTER INPUTTING 16 BYTES FROM THE KEYBOARD, IT IS |
| | | | *ASSUMED THAT ANY RECORD OUTPUT TO THE CASSETTE |
| | | | *IS COMPLETE. MOVE DATA FROM 0'37' - 0'20' TO 0'77' - |
| | | | 0'60'. |
| S8 | LISL | 7 | LOAD FIRST SOURCE BYTE |
| | | | ADDRESS |
| S9 | LISU | 3 | LOAD NEXT BYTE |
| S10 | LR | A,S | |
| S11 | LISU | 7 | STORE NEXT BYTE |
| S12 | LR | D,A | IF NOT END OF BUFFER, |
| S13 | BR7 | S9 | RETURN FOR NEXT BYTE |
| | | | IF END OF FIRST BUFFER, |
| S14 | LISU | 2 | MOVE SECOND BUFFER |
| | | | REPEAT MOVE FOR SECOND |
| S15 | LR | A,S | 8 BYTE |
| | | | BUFFER |
| | LISU | 6 | |
| | LR | D,A | |
| | BR7 | S14 | |
| S16 | LISU | 5 | INPUT NEXT EIGHT BYTES |
| | | | FROM KEYBOARD TO |
| | | | SCRATCHPAD BUFFER 0'57' |
| S17 | PI | INKB | TO 0'50' |

| | | | |
|-----|------|------|-------------------------|
| S19 | LISU | 4 | INPUT NEXT EIGHT BYTES |
| | | | FROM KEYBOARD TO |
| S20 | PI | INKB | SCRATCHPAD BUFFER 0'47' |
| | | | TO 0'40' |

*BUFFER IS NOW READY TO BE OUTPUT TO CASSETTE.
 S21 LI H'3F' LOAD BUFFER INITIAL
 ADDRESS
 S22 LR 0,A (0'77') INTO SCRATCHPAD
 BYTE 0
 S23 LI ONC TURN CASSETTE ON
 S24 OUTS 5
 S25 BR START RETURN FOR NEXT RECORD
 *INPUT SUBROUTINE INKB STORES A BYTE OF DATA INPUT
 *FROM KEYBOARD INTO SCRATCHPAD BYTE ADDRESSED
 *BY ISAR

| | | | |
|------|------|------|---|
| INKB | LR | K,P | SAVE RETURN ADDRESS |
| | | | IN K |
| LO | CLR | | CLEAR PORT 0 |
| | OUTS | 0 | |
| LOOP | INS | 0 | INPUT STATUS |
| L1 | BP | LOOP | |
| L2 | INS | 1 | INPUT DATA |
| L3 | LR | D,A | STORE IN ISAR BUFFER |
| | BR7 | LO | RETURN IF NOT EIGHTH BYTE |
| L4 | PK | | RETURN |
| | | | *INTERRUPT SERVICE ROUTINE, EXECUTED TO WRITE ONE |
| | | | *BYTE TO CASSETTE. |

| | | | |
|-----|------|---------|----------------------------|
| | ORG | H'0280' | |
| E0 | LR | 1,A | SAVE ACCUMULATOR IN |
| | | | SCRATCHPAD BYTE 1 |
| E1 | LR | A,IS | SAVE ISAR IN SCRATCHPAD |
| | | | BYTE 2 |
| E2 | LR | 2,A | |
| E3 | LR | A,0 | LOAD SCRATCHPAD BYTE 0 |
| | | | CONTENTS INTO ISAR |
| E4 | LR | IS,A | |
| E5 | INS | 5 | RECEIVE STATUS FROM |
| | | | CASSETTE, INS SETS STATUS |
| E7 | BZ | F0 | |
| E8 | LR | A,S | IF NOT END OF CASSETTE, |
| E9 | OUTS | 4 | OUTPUT NEXT BYTE |
| E10 | LR | A,IS | MOVE ISAR TO A |
| E11 | AI | H'FF' | DECREMENT ALL 6 BITS OF |
| | | | ADDRESS |
| E12 | CI | 0'37' | TEST IF RESULT IS 0'37' |
| E13 | BZ | E17 | RETURN IF NOT |
| E14 | LI | STOP | IF IT IS, ISSUE A STOP |
| | | | COMMAND |
| E15 | OUTS | 4 | |
| E16 | LI | 0'77' | RESET TO TOP FOR NEXT |
| | | | OUTPUT |
| E17 | LR | 0,A | SAVE ISAR ADDRESS FOR |
| | | | NEXT BYTE |
| E18 | LR | A,2 | BEFORE RETURNING, |
| | | | RESTORE ACCUMULATOR |
| | LR | IS,A | AND ISAR |
| | LR | A,1 | |
| | EI | | |
| | POP | | |
| F0 | LI | REW | IF CASSETTE IS FULL, ISSUE |
| | OUTS | 4 | REWIND COMMAND |
| | BR | E18 | |

The logic of this program is relatively simple. Scratchpad bytes O'77' to O'40' constitute a 32-byte buffer, the contents of which is output as a record to the cassette. It is assumed that this record can be written to the cassette in less time than an operator takes to enter 16 digits at the keyboard. Therefore instructions START through S7, input 16 digits into the 16 scratchpad bytes addressed by O'37' through O'20'.

Data is input from the keyboard using programmed I/O via subroutine INKB. Notice that subroutine INKB saves its return address in the K scratchpad registers and uses the PK instruction to return; therefore a stack register is available for the interrupt. Subroutine INKB is almost identical to the input subroutine described in Section 8.1.1. The principle difference is that separate ports are being used for status and data. Observe that throughout this program data is input into scratchpad bytes, one scratchpad 8-byte buffer at a time.

Once 16 digits have been input from the keyboard, they are moved from scratchpad bytes O'37' - O'20' to O'77' - O'60'. This entire data movement will take 208 microseconds which is not a noticeable delay to an operator entering data at the keyboard.

The next 16 bytes of data entered at the keyboard go directly into scratchpad bytes O'57' through O'50' and O'47' through O'40'.

After 32 bytes have been entered into the scratchpad buffer, a buffer counter is initialized in scratchpad byte 0 (instructions 21 et. seq.); then the cassette is turned on by instructions S23 and S24. ONC is used as a symbol representing the one byte code which will be recognized by the cassette control logic as a turn-on signal. Once the cassette has been turned on, program logic branches back to the start of data entry for the next record.

Notice that nowhere in the main program has the interrupt service routine been mentioned. It is assumed that once the cassette has been turned on, cassette control logic will issue an interrupt request signal each time it is ready to receive another byte of data from the microprocessor. The interrupt service routine therefore may be executed at any time. It is as though there were a floating call to a subroutine that could randomly be executed at any point in the program where interrupts were being allowed.

Observe that the interrupt service routine has to save the contents of the accumulator and the ISAR in scratchpad bytes because the accumulator and ISAR are going to be needed.

The illustrated interrupt service routine is probably somewhat simpler than most real interrupt service routines would be. Control logic associated with the cassette drive is assumed capable of sending status inputs to the microprocessor telling the microprocessor when to rewind the cassette. It is also assumed that housekeeping associated with the start and end of each record is handled by the cassette control logic. In all probability much of this housekeeping could be done by the microprocessor, but to include it in the example would detract from the purpose of the example, which is to show how an interrupt service routine is handled.

The origins of the main program and interrupt service routine have been randomly selected. Note that since the origin of the interrupt service routine has been selected at H'0280',

this is the address which must be in the 3851 interrupt address register.

The symbols STOP and REW in the interrupt service routine must be equated to the actual bit pattern that the cassette controller logic will interpret as stop and rewind commands, respectively.

8.3 LOCAL TIMERS (PROGRAMMABLE TIMERS)

Programmable timers are a more useful microprocessor programming tool than is initially apparent to a programmer.

Programmable timers are shift registers which, after being loaded with some initial value, count down to 0, then send an interrupt request signal to the CPU. (See Section 2.5.4.) The 3851 PSU and the 3853 SMI device both have programmable timers.

Here are some applications for which timers are useful:

- 1) In control applications, such as an operations monitor alarm, to insure that some maximum time interval is not exceeded between consecutive readings from sensitive data inputs. For example, suppose a temperature must be measured in a chemical reactor at least once every second to prevent runaway conditions. 253 maximum time intervals on a local timer approximate 1 second. Whenever a temperature is input, the local timer is reset to start counting down one second. If one second is counted down, the program can be written to output a signal that triggers an audible alarm.
- 2) To activate refresh logic for external devices. For example, a video display may need to be refreshed at fixed time intervals; the refresh sequence may be initiated by a local timer.
- 3) To maintain the real time of day in any system that has to generate clock times. Such devices include badge readers and numerous small office business systems.

8.3.1 Local Timer I/O Ports

Local timer logic uses the local interrupt control I/O ports to enable local timer interrupts, as described in Section 8.2.2 and Table 8-1.

The interrupt control I/O port must have the value H'03' loaded into it under program control in order to enable local timer interrupts at that one device. Therefore either external interrupts or local timer interrupts, but not both, may be enabled at one device.

If interrupts have been disabled at the 3850 CPU, local timer interrupt requests will be ignored until a subsequent interrupt enable. At this time any interrupt request will still be active unless cleared prior to the interrupt enable.

The timer I/O ports have I/O port addresses one higher than the local interrupt control I/O port. Therefore 3851 PSU port addresses are:

B'xxxxxx10' for the local interrupt control I/O port
 B'xxxxxx11' for the local timer I/O port

For the 3853 SMI, port addresses are:

H'OE' for the local interrupt control I/O port
 H'OF' for the local timer I/O port

8.3.2 Programming Local Timers

Programming a local timer requires the value H'03' to be loaded into the selected device's local interrupt control I/O port. A number between 0 and 254, identified as a timer constant, is loaded into the associated local timer I/O port. A value of 255 loaded into the local timer I/O port stops the clock.

The value loaded into a local timer, as a timer constant, is converted (by the assembler) to a binary value, as given in Appendix C; that is why numbers should be entered as timer constants.

A local timer interrupt will be generated after the time interval given by the product:

(system clock pulse interval) * (local timer constant) * 31

For example, a value of T'200' loaded into a local timer I/O port will generate an interrupt after 3.1 ms if the system clock pulse interval is 500 ns.

Instructions needed to enable a local timer are as follows:

```

—
—
—
—
—
—
LI      T'200'   LOAD TIMER CONSTANT
OUTS    7        OUTPUT TO TIMER I/O PORT 7
LIS     3        LOAD TIMER INITIATION
                CONTROL
OUTS    6        OUTPUT TO CONTROL I/O
                PORT 6
—
—
—
EI                      ENABLE INTERRUPTS AT
                        THE 3850 CPU
—

```

In the above example, the timer constant T'200' has been arbitrarily selected. Any value from T'0' to T'256' could be used. T'256', remember, will stop the clock.

The selection of I/O ports 7 and 6 is also arbitrary; any pair of I/O ports with addresses given in Section 8.3.1 could be used. Note, however, that the control I/O port number is always one less than the timer port number it controls.

The value H'03' must be loaded into a local timer control I/O port if the associated timer port is to operate. When this value is loaded into the control I/O port any pending timer interrupt is cleared. Any subsequent zero value of the timer will set the timer interrupt.

If the value H'03' is in the control I/O port before the timer constant is output to the timer I/O port, then the timer which is constantly running may interrupt before being set with a timer constant. Once the timer I/O port holds a zero value, an interrupt request signal will be generated once every 3.953 ms (for a 500 ns clock pulse). Providing the ICB bit is 1 within the 3850 CPU, every timer interrupt request will be acknowledged and serviced if the timer interrupt is enabled.

The program that is executed after a timer interrupt is acknowledged is a service routine which, like the service routine illustrated in Section 8.2.7, is never called or referenced by any other program. The service routine must start executing at the memory address provided by the 3851 or 3853 device's interrupt address I/O ports; however, recall that the 7 bit of the address is automatically set to 0 for a timer interrupt, or to 1 for an external interrupt. If the external interrupt service routine is originated at H'0680', as illustrated in Section 8.2.7, then for the same device, the local timer interrupt service routine will be originated at H'0600'.

8.3.3 A Programming Example – The Time of Day

The program below creates the time of day by storing hours in scratchpad byte 8, minutes in scratchpad byte 7 and seconds in scratchpad byte 6. Scratchpad byte 5 is used as a counter.

This program uses the maximum timer interval (3.953 ms between interrupts). The local timer must be initialized with the main program as follows:

```

LIS     0        ZERO HOURS, MINUTES AND
LR      8,A      SECONDS PORTS, ASSUM-
LR      7,A      ING THE DEVICE WILL BE
LR      6,A      SWITCHED ON EXACTLY AT
                MIDNIGHT
LI      253     INITIALIZE THE LOCAL
LR      5,A     COUNTER TO 253
LI      T'0'    CLEAR LOCAL TIMER PORT
OUTS    7
LIS     H'03'   ENABLE THE LOCAL TIMER
OUTS    6       PORT INTERRUPTS
EI                      ENABLE INTERRUPTS AT
                THE CPU

```

The local timer interrupt service routine is assumed to be originated at H'0200'. It executes as follows:

```

ORG     H'0200'
DS      5        DECREMENT THE LOCAL
                COUNTER
BNZ     OUT     CONTINUE IF IT IS NOT ZERO
                (ONE SEC).
                IF IT IS ZERO, RESET TO 253
LI      253
LR      5,A
LR      A,6     INCREMENT THE SECONDS
                COUNTER
INC     CI      TEST IF SECONDS EQUAL 60
CI      60

```

```

      BZ      T10      IF THEY DO, ADJUST
                        MINUTES
      LR      6,A      IF THEY DO NOT, END
OUT   EI
      POP
*MINUTES ADJUST BEGINS HERE
T10  LIS      0        ZERO SECONDS
      LR      6,A
      LR      A,7      LOAD MINUTES
      INC
      INC      INCREMENT MINUTES
      CI      60       TEST FOR 60 MINUTES
      BZ      T20     AT 60 MINUTES, ADJUST
      LR      7,A     HOURS OTHERWISE RETURN
                        MINUTES
      EI
      POP
*HOURS ADJUST BEGINS HERE
T20  LIS      0        ZERO MINUTES
      LR      7,A
      LI      153     CORRECT 0.392 SECOND
                        ERROR EVERY HOUR
      LR      5,A
      LR      A,8     LOAD HOURS
      INC
      INC      INCREMENT HOURS
      CI      24     TEST FOR 24 HOURS
      BNZ     T30    AT 24 HOURS, RESET TO 0
      LIS      0      OTHERWISE RETURN HOURS
T30  LR      8,A
      EI
      POP

```

8.4 DIRECT MEMORY ACCESS

Direct memory access (DMA) allows data to be transferred between any F8 microprocessor system memory and an external device, bypassing the 3850 CPU. Data is transferred in parallel with any CPU operations. DMA has been described, as a concept, in Sections 2.6.3 and 2.8.

One 3852 DMI device must be present in a microprocessor system that supports DMA. Up to four 3854 DMA devices may be present in the system; each 3854 DMA device provides one DMA channel.

8.4.1 When to Use DMA

DMA is used to transfer data into, or out of, a microprocessor system that has heavy I/O requirements. For example, using programmed I/O, the theoretically maximum data transfer rate is implemented by the following instruction sequence for data input:

```

LOOP  INS      0        INPUT A DATA BYTE VIA
                        PORT 0
      ST
      DS      1        STORE IN RAM MEMORY
                        TEST FOR END OF TRANS-
                        MISSION
      BNZ     LOOP     RETURN FOR NEXT CHAR-
                        ACTER

```

Scratchpad register 1 is assumed to hold the initial character count.

These four instructions execute in 9.5 instruction cycles, equal to 19 μ s, using a 500 ns clock pulse. Assuming that external logic is synchronized to input one byte of data every 19 μ s, the maximum data transfer rate is approximately 50,000 bytes/second.

The maximum data transfer rate supported by programmed I/O is not of itself a limiting factor. A 256 byte buffer, for example, can be transferred in 4.86 ms. The problem is that this maximum data transfer rate requires external logic that processes data at a rate of one byte every 19 μ s. Most applications will not meet this requirement, usually because data transfer rates are set by logic considerations beyond the microprocessor system; that is, external logic determines data transfer rates, not the microprocessor system.

Suppose external logic is inputting data to the microprocessor system at some rate, which we will label R bytes/second. The time that elapses between each byte transferred will be $(1,000,000/R)$ μ s. The local timer can be used to generate an interrupt shortly before each byte of data is due, in which case the local timer interrupt service routine will input the data byte. Assuming that data will always be in the I/O port before the local timer interrupt service routine is executed, the following service routine will input data bytes from an I/O port:

```

ISRI  LR      0,A      SAVE ACCUMULATOR
                        CONTENTS IN 0
      XDC
      LI      TCNT     SWITCH DC0 AND DC1
                        RESTART TIMER
      OUTS    7
      INS      0        INPUT DATA BYTE
      ST
      XDC
      LR      A,0     RESTORE ACCUMULATOR
                        FROM 0
      EI
      POP           ENABLE INTERRUPTS
                        RETURN

```

TCNT is a symbol defined by the equate directive:

```
TNCT EQU T'VAL'
```

where VAL is a number between 0 and 255. Each count represents 31 clock periods and the total time is equal to $(1,000,000/R)$ but less than 3.953 ms.

It will take approximately 38 μ s for interrupt service routine ISRI to execute; this means that approximately 9.7 ms will be required to input 256 bytes of data. This 9.7 ms will be spread over whatever time interval the external device requires to transfer 256 bytes of data. But there are some problems associated with the method of inputting data:

- 1) Recall that there are certain privileged instructions which inhibit acknowledgement of an interrupt. It is quite feasible for a 2 to 4 μ s delay to randomly get inserted between each execution of ISRI if, by chance, a privileged instruction is being executed at the instant the local timer times out. Over 256 bytes of data transfer, this means that it is feasible for a 500 μ s slew to develop, which will result in the loss of a byte of data, if the data transfer rate exceeds 2,000 bytes/s.
- 2) If the microprocessor is handling interrupts other than the local timer, clearly other interrupts must be serviced by routines which are themselves interruptable, since one interrupt service routine blocking out ISRI for any significant period of time would almost certainly create irrecoverable timing errors.

- 3) Observe that ISRI uses the DC1 register and uses one scratchpad register to store accumulator contents. This means that the DC1 register and the scratchpad register cannot be used by any other program that is being executed during the same time period.

If subroutine ISRI is expanded to include a status test plus logic to compute the timer constant that will compensate for timing slews, the new expanded version of ISRI might easily take 200 μ s to execute. Under these circumstances the microprocessor system would spend a significant amount of its time merely moving data between memory and an I/O port.

In all but the simplest I/O transfer applications, therefore, DMA becomes the preferable way of moving data between memory and external devices.

8.4.2 Programming DMA

The actual programming steps required in order to initiate a DMA operation are simple, as follows:

| | | |
|-----|------|--|
| LI | ADLO | LOAD BUFFER STARTING ADDRESS INTO ADDRESS I/O PORTS |
| OUT | BUFA | |
| LI | ADHI | |
| OUT | BUFB | |
| LI | CTLO | LOAD LOW ORDER BYTE OF BYTE COUNT |
| OUT | BUFC | |
| LI | CTRL | LOAD HIGH ORDER 4 BITS OF BYTE COUNT PLUS CONTROL BITS |
| OUT | BUFD | |

Symbols must be equated as follows:

- 1) The I/O port addresses, BUFA, BUFB, BUFC and BUFD

are given in Table 2-2 for the four 3854 DMA devices that may be present in an F8 microprocessor system. Whether a DMA device uses the first, second, third or fourth set of addresses is a function of device hardware configuration and of no concern to the programmer, so long as the correct port addresses are used.

- 2) ADLO and ADHI represent the low order and high order bytes of the beginning address of the memory buffer into which data will be written, or from which data will be read.

- 3) Data buffers may be up to 4,096 bytes long. CTLO represents the low order eight bits of the buffer length, as illustrated in Figure 8-4. CTRL provides the controls which select DMA options and also the high order four bits of the buffer length, as illustrated in Figure 8-4.

The following instructions will initiate 256 bytes of data being written into a memory buffer, where the data rate is controlled by the external device. The memory buffer starting address is H'A280'. The first DMA channel is used.

| | | |
|-----|-------|--|
| --- | | |
| --- | | |
| --- | | |
| LI | H'80' | OUTPUT LOW ORDER BYTE OF ADDRESS |
| OUT | H'F0' | |
| LI | H'A2' | OUTPUT HIGH ORDER BYTE OF ADDRESS |
| OUT | H'F1' | |
| LI | H'00' | OUTPUT LOW ORDER BYTE OF COUNT |
| OUT | H'F2' | |
| LI | H'C1' | OUTPUT HIGH ORDER 4 DIGITS OF COUNT (1) AND CONTROL DIGIT (C). |
| OUT | H'F3' | |

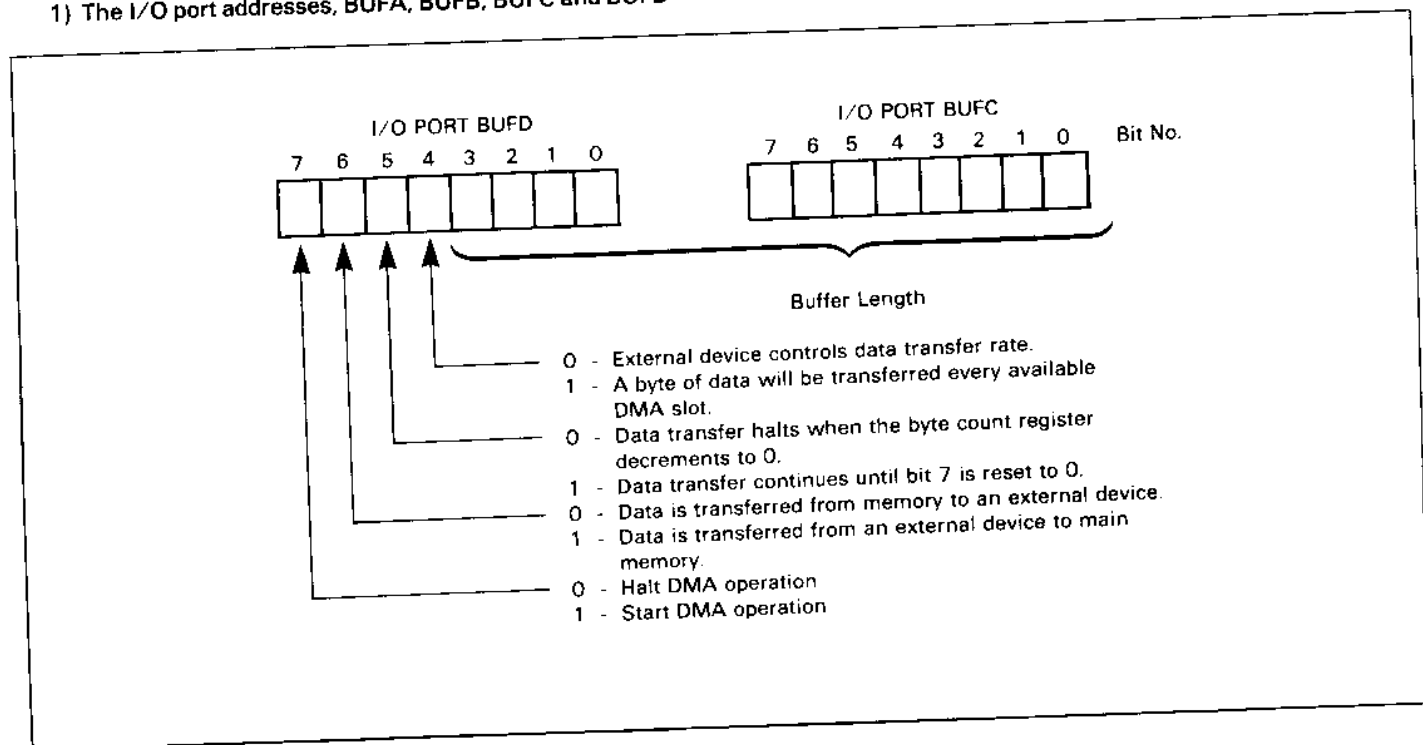


Fig. 8-4. How BUFC and BUFD are used to Control DMA Operations

8.4.3 Catching DMA on the Fly

There are many applications in which data will be transferred via DMA at unpredictable rates. For example, in communications applications, data may come over a telephone line at a fixed baud rate, but the length of messages and the period when no data is being transferred may be completely random. Under such circumstances it is very useful if a program can start and stop DMA operations or interrogate the buffer counter to find out how much data has been transferred via DMA since the last interrogation. The following program sequence catches DMA on the fly, in a way that would be well suited to random data transfer rates in communications applications:

*SUBROUTINE TO INITIALIZE DMA WITH H'FF' IN THE BYTE
*COUNTER. THE DATA BUFFER STARTS AT H'2000'

```
DMA  LI    H'00'    OUTPUT BUFFER STARTING  
                        ADDRESS  
      OUT  H'F0'  
      LI   H'20'  
      OUT  H'F1'  
      LI   H'FF'    OUTPUT BYTE COUNTER  
      OUT  H'F2'  
      LI   H'CO'  
S2   OUT  H'F3'  
      POP
```

*MAIN PROGRAM TO HANDLE COMMUNICATIONS DATA
*TRANSFERRED VIA DMA

```
      PI    DMA    INITIALIZE DMA  
      —  
      —  
      —  
M1   LIS   0      STOP DMA DATA TRANSFER  
M2   OUT   H'F3'  
M3   IN    H'F2'   LOAD BYTE COUNT INTO  
      COM    SCRATCHPAD BYTE 0  
M4   LR    0,A
```

(instructions to process data follow here)

Instruction steps to initiate DMA are packaged as a subroutine labeled DMA. The buffer length output is H'FF'. As this buffer length is counted down, the number of bytes transferred via DMA can, at any time, be determined by reading the contents of I/O port F2 into the accumulator and complementing. The control digit C starts data flow via DMA from the external device (assumed to be a communications interface) to the memory buffer, beginning at H'2000'.

The main program starts by initializing DMA via a call to subroutine DMA. At some later point in the program, instructions M1 and M2 are executed in order to load the code digit 0 into I/O port F3 and thus stop DMA transfers. Instructions M3 through M4 determine the number of bytes that have been transferred via DMA, since DMA was initiated, and loads this byte count into scratchpad register 0. Instructions will now follow to move the number of bytes received to some other memory location where the data can be processed. Subroutine DMA will then be recalled to re-initialize DMA data transfers. After data has been processed execution will branch back to instruction M1 and so the program will continue processing whatever data has been transferred in each time interval.

PROGRAM OPTIMIZATION

Optimizing a program is not a routine mechanical task; rather, it is a function of application requirements and hardware configuration. Most microprocessor programs are written either to maximize execution speed, or to minimize the amount of memory used.

Consider a simple example. A microprocessor has 1024 bytes of program memory. An application may only use half of the available memory, but may be too slow to meet product specifications. Converting every subroutine to a macro will speed up program execution time, but may double the size of the program. Since program memory comes in finite increments, economizing on program storage requirements is only meaningful when it reduces the number of devices required by a microprocessor system; therefore, increasing program storage requirements from 500 bytes to 1000 bytes carries no penalty.

In practice, programming for minimum use of program storage should be the goal of microprocessor programmers. Microprocessor instruction sets are very versatile. Many variations of a program can be written to implement any problem; but some programs will be more efficient than others. A novice microprocessor programmer may well write programs that occupy 50% more memory than is really necessary. Inefficiencies of this type are not important in minicomputer systems, which usually include bulk storage devices such as disk units. The only penalty paid for having unnecessarily long programs is a few extra milliseconds, making otherwise unnecessary transfers of program segments between disk and memory. Unnecessarily long programs are very uneconomical in microprocessor systems, where the entire program sits in one or more memory devices. If a microprocessor system has two more memory devices than the most compact program would require, these two memory devices can become 20,000 memory devices, if the microprocessor system is to be reproduced 10,000 times.

In many ways, the logic designer will find it easier to become an efficient microprocessor programmer than will a systems analyst, who has gained experience programming minicomputers and larger systems. The systems analyst has continuously striven to write programs which are general purpose. For example, a subroutine that performs multibyte addition must be able to add two number buffers of any length, located anywhere in memory, storing the result in a third number buffer. Such a multibyte addition subroutine, once written, could be frequently reused in almost any application, thus reducing future programming expenses. This is economical thinking in the world of minicomputers, but it is very uneconomical thinking in the world of microprocessors. A microprocessor application may be able to define two number buffers of specific length, in specific areas of memory, as the only number buffers which will ever be involved in mathematical operations. A multibyte addition subroutine, working within these restrictions, may have to be rewritten for every new microprocessor application, but the subroutine that results may use less than half of the memory storage requirements demanded by the equivalent general purpose routine. When microprocessor systems are likely to be reproduced tens of thousands of times, extra front-end programming expense becomes trivial compared to the cost of extra memory devices, multiplied ten thousand fold.

In the following sub-sections, program optimization information is presented in the following sequence:

- 1) The concept of counting memory bytes and execution cycles is described.
- 2) Some basic techniques that will always make F8 programs more efficient are listed.
- 3) Some examples of execution speed versus memory utilization tradeoffs are given.

9.1 COUNTING CYCLES AND BYTES

The F8 instruction set is summarized in Appendix D, where the number of object program bytes is listed for every instruction.

Consider the data movement program described in Figure 5-1. This program is reproduced in Figure 9-1, along with number of execution cycles and memory bytes required by each instruction.

Counting bytes is usually unnecessary, since the assembler listing prints the memory location where each object program byte will be stored. Thus subtracting memory addresses yields the length of any program, program segment or subroutine.

9.2 ELEMENTARY OPTIMIZATION TECHNIQUES

There are a number of instruction choices where one selection is always preferable. These obvious instruction choices are described in the following sub-sections.

9.2.1 Scratchpad and RAM Memory

Always fill up the scratchpad before using RAM memory to store constants or data buffers. It takes one cycle to move a byte of data between the accumulator and a scratchpad byte; it takes 2.5 cycles to move a byte of data between the accumulator and external RAM. Both sets of instructions generate one byte of object code.

9.2.2 Immediate Instructions

Immediate instructions are 2 or 3-byte instructions that specify data in the instruction operand.

Consider the 2-byte immediate instructions; these instructions specify a 1-byte operand, which is combined with the contents of the accumulator in some way. An instruction such as:

| | | | |
|----|----|-----|----------------------------------|
| IM | LI | CNT | LOAD COUNTER INTO ACCUMULATOR |
|----|----|-----|----------------------------------|

executes in 2.5 cycles and occupies two bytes of memory. If this instruction occurs identically (with the same operand) many times in a program, consider loading CNT into a scratchpad register, as follows:

| | | | |
|------|----|-----|----------------------------------|
| ONE | LI | CNT | |
| TWO | LR | 1,A | |
| | — | | |
| | — | | |
| | — | | |
| THRE | LR | A,1 | LOAD COUNTER INTO ACCUMULATOR |

| Cycles | Bytes | TITLE | | |
|--------|-------|--------|---------|--|
| 0 | 0 | MAXCPU | 50 | LIMIT OF 50 SECONDS CPU TIME SPECIFIED |
| 0 | 0 | SYMBOL | | A SYMBOL TABLE WILL FOLLOW SOURCE PROGRAM |
| 0 | 0 | XREF | | SYMBOLS CROSS LISTING WILL FOLLOW SOURCE PROGRAM |
| 0 | 0 | BASE | HEX | HEXADECIMAL NUMBERS SPECIFIED FOR ASSEMBLY LISTING |
| 0 | 0 | BUFA | EQU | H'0800' SET THE VALUE OF SYMBOL BUFA |
| 0 | 0 | BUFB | EQU | H'08A0' SET THE VALUE OF SYMBOL BUFB |
| 0 | 0 | ORG | H'0100' | |
| 6 | 3 | ONE | DCI | BUFA SET DC0 TO BUFA STARTING ADDRESS |
| 2 | 1 | TWO | XDC | STORE IN DC1 |
| 6 | 3 | THREE | DCI | BUFB SET DC0 TO BUFB STARTING ADDRESS |
| 2.5 | 2 | FOUR | LI | H'80' LOAD BUFFER LENGTH INTO ACCUMULATOR |
| 1 | 1 | FIVE | LR | 1,A SAVE BUFFER LENGTH IN SCRATCHPAD BYTE 1 |
| 2.5 | 1 | LOOP | LM | LOAD CONTENTS OF MEMORY BYTE ADDRESSED BY DC0 |
| 2 | 1 | SIX | XDC | EXCHANGE DC0 AND DC1 |
| 2.5 | 1 | SEVEN | ST | STORE ACCUMULATOR IN MEMORY BYTE ADDRESSED BY DC0 |
| 2 | 1 | EIGHT | XDC | EXCHANGE DC0 AND DC1 |
| 1.5 | 1 | NINE | DS | 1 DECREMENT SCRATCHPAD BYTE 1 |
| 3.5 | 2 | BNZ | LOOP | IF SCRATCHPAD BYTE 1 IS NOT ZERO, RETURN TO LOOP |
| 0 | 0 | END | | |
| 31.5 | 17 | | | |

* BNZ will usually return to LOOP

Total Bytes = 17

Total Cycles = 31.5

Total Cycles within iterative loop = 14

Assuming 2 μ s cycle time, time to move 128 bytes = $2 \times (14 \times 128 + 17.5)$
= 3619 μ s

Fig. 9-1. Counting Cycles and Bytes

Instructions ONE and TWO execute in 3.5 cycles and occupy three bytes of memory. Instruction THREE executes in one cycle, occupies one byte of memory and replaces instruction IM.

Clearly instruction IM is better than ONE, TWO and THRE, if IM occurs just once; however, if instruction IM occurs identically n times, then it accumulates 2.5n cycles and 2n bytes of memory, whereas ONE, TWO and THRE accumulate (3.5+n) cycles and (3+n) bytes of memory, respectively. Therefore ONE, TWO and THRE will execute faster when:

$$2.5n > 3.5 + n$$

$$\text{or } 1.5n > 3.5$$

$$\text{or } n > 2.33$$

ONE, TWO and THRE occupy less memory when:

$$2n > (3 + n)$$

$$\text{or } n > 3$$

In conclusion, if a 2-byte immediate instruction occurs identically (same operand) three or more times in a program, it is more efficient to load the immediate operand into a scratchpad byte out of which it is referenced (providing a scratchpad byte is available).

9.2.3 Short Instructions

Always go over a source program, making sure that the short instructions LIS, INS and OUTS have been used wherever the operand is small enough.

9.2.4 Use of DS Instruction to Decrement and Test

Recall that when a DS instruction is used, the decremented scratchpad byte may be tested for "decrement-from-zero".

Since the DS instruction adds H'FF' to the designated scratchpad byte contents, the carry status will always be set unless the scratchpad byte contained 0 before it was decremented. Therefore the instruction sequence:

```
DS    n
BC    BACK
```

will decrement scratchpad byte n, return to BACK if byte n did not contain 0, but continue if byte n did contain 0.

9.2.5 Use of the BR7 Instruction

The BR7 instruction is very useful when manipulating data buffers in scratchpad memory, as described in Section 7.1.

9.3 PROGRAMMING FOR SPEED OR MEMORY ECONOMY

In the following subsections, programming techniques that tradeoff between execution speed and the amount of memory used are described.

9.3.1 Macros and Subroutines

To gain execution speed, possibly with a heavy increase in the amount of memory required, convert subroutines into macros as described in Section 7.4.

Always carefully examine subroutines, particularly those which are infrequently called or receive parameters from the calling program, to see if converting the subroutine into a macro would save memory bytes and, at the same time, increase execution speed.

As described in Section 7.3, programs can be made much faster and will require less memory if subroutine nesting is limited to a first level. If a main program calls a subroutine, the subroutine can then call another subroutine. However, a subroutine cannot call another subroutine if it was, itself, called by a subroutine. Limiting subroutine nesting to a level of one means that return addresses can be stored in the stack register (PC1) and in the K registers of the scratchpad, eliminating the need for memory stacks.

9.3.2 Table Lookups Versus Data Manipulation

Program execution speed can frequently be increased by looking up data out of tables in ROM.

The concept is illustrated below, for the simple case of a 1-of-8 decoder.

An octal digit is input into the low order three bits of I/O port 0. The CPU must output, via I/O port 1, a data byte as follows:

| Input From Port 0 | Output At Port 1 |
|-------------------|------------------|
| 00000001 | 00000001 |
| 00000010 | 00000010 |
| 00000011 | 00000100 |
| 00000100 | 00001000 |
| 00000110 | 00100000 |
| 00000111 | 01000000 |
| 00000000 | 10000000 |

*ONE OF EIGHT DECODER PROGRAM, NOT USING TABLE *LOOKUP

| | | | |
|------|------|------|-------------------------------|
| | INS | 0 | INPUT OCTAL CODE |
| | BNZ | I10 | INPUT IS NOT ZERO |
| | LIS | 8 | LOOP COUNTER |
| I10 | LR | 0,A | TO LOOP COUNTER |
| | LIS | 1 | LOAD OUTPUT FOR 1 |
| LOOP | DS | 0 | DECREMENT INPUT |
| | BZ | OUT | BRANCH OUT IF END |
| | SL | 1 | SHIFT LEFT ONE BIT IF NOT END |
| | BR | LOOP | |
| OUT | OUTS | 1 | OUTPUT RESULT |

*ONE OF EIGHT DECODER PROGRAM USING TABLE *LOOKUPS

| | | | |
|------|------|------|--------------------------------|
| LKUP | DC | 0 | |
| | DC | 2 | |
| | DC | 4 | |
| | DC | 8 | |
| | DC | 16 | |
| | DC | 32 | |
| | DC | 64 | |
| | DC | 128 | |
| | — | | |
| | — | | |
| | — | | |
| | INS | 0 | INPUT OCTAL CODE |
| | DCI | LKUP | LOAD TABLE BASE ADDRESS |
| | ADC | | ADD INPUT CODE TO BASE ADDRESS |
| | LM | | LOAD OUTPUT |
| | OUTS | 1 | OUTPUT RESULT |

Efficiencies compare as follows:

| | | Non-Table Lookup | Table Lookup |
|------------------|------|------------------|--------------|
| Instructions | | 10 | 5 |
| Memory bytes | | 13 | 15 |
| Execution cycles | min: | 15 | 15 |
| | max: | 69.5 | 15 |

SOME USEFUL PROGRAMS

Some generally useful programs are given in this section. Programs are not shown as subroutines or as macros. The instructions implementing required logic are given, making it easy to incorporate an example into a program as a subroutine, a macro or directly as a section of main memory. These programs are intended to show programming techniques, rather than to demonstrate optimum program efficiency.

10.1 GENERATING TEXT

10.1.1 Simple and Dedicated Text Programs

The simplest text generation logic takes characters out of a memory buffer and outputs them via an I/O port. The I/O operation may be under program control, or interrupt I/O may be used. In each case, text is fetched via an elementary instruction sequence such as:

```

DCI    TEXT    LOAD TEXT BUFFER
                STARTING ADDRESS
LOOP   LM      LOAD NEXT TEXT BYTE
*TEST FOR END-OF-RECORD CHARACTER. INSTRUCTIONS
*FOR THIS TEXT ARE NOT SHOWN, SINCE THEY ARE A FUNC-
*TION OF THE APPLICATION.
        OUT    PRTN    OUTPUT CHARACTER VIA
                PORT N
        BR     LOOP    RETURN FOR NEXT
                CHARACTER
    
```

10.1.2 Unpacking Decimal Digits

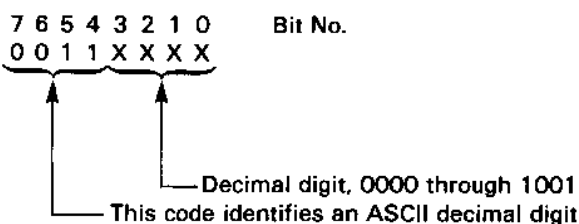
A byte containing two BCD digits is converted into two ASCII digits as follows:

```

LM      LOAD BYTE WITH TWO
        BCD DIGITS
LR      0,A    SAVE BYTE IN SCRATCHPAD
        BYTE 0
SL      4
SR      4      ISOLATE LOW ORDER DIGIT
AS      1      ADD HIGH ORDER FOUR
        ASCII BITS
LR      2,A    SAVE IN SCRATCHPAD
        BYTE 2
LR      A,0    LOAD TWO BCD DIGITS
SR      4      ISOLATE HIGH ORDER DIGIT
AS      1      ADD HIGH ORDER FOUR
        BITS
    
```

*CHARACTER OUTPUT SEQUENCE FOLLOWS HERE

This instruction sequence assumes that scratchpad byte 0 is available for temporary storage and that scratchpad byte 1 contains H'30'. Refer to Appendix B. A decimal digit becomes an ASCII character as follows:



If scratchpad byte 0 is not available, any other byte may be used for data storage.

If scratchpad byte 1 is not available, any other scratchpad byte, or the immediate instruction:

```
AI    H'30'
```

may be used.

10.1.3 Variable Text

It is possible to have a text generation program in ROM that outputs variable text, temporarily stored in RAM. In other words, a fixed ROM program outputs messages of variable length and content. This is useful in word processing or human dialog applications. For example, an F8 microprocessor may drive a CRT used to collect data from convention attendees; the text program described below allows the dialog that will be displayed to be changed at any time, without changing the text generation program.

The text table (labeled TEXT below) contains characters in any mixed sequence.

The index table (labeled TIND below) consists of the following 3-byte sequence:

This variable text generation program uses two data tables: a text table and an index table.

- Bytes 1 and 2 - Displacement from TEXT to first character to be output. If Byte 1 = H'FF', end of message is indicated. Byte 1 displacement cannot be H'FF'.
- Byte 3 - Number of characters to be printed.

Messages are identified by number, starting at 1. A message's number is its sequential location, as identified by H'FF' codes in TIND.

Consider the following very simple example. The following four messages are to be generated:

- 1) ENTER PRODUCT NUMBER:
- 2) NO SUCH PRODUCT RE-ENTER:
- 3) NUMBER OF UNITS:
- 4) PRODUCT SHIP DATE:

The following TEXT table will be needed:

```

RE-ENTER#PRODUCT
#NUMBER:#NO#SUCH
#OF#UNITS:SHIP#D
ATE:#
    
```

The following TIND table will be needed:

| Byte No. (Hexadecimal) | Contents (Hexadecimal) |
|---------------------------|---------------------------|
| 0 | 00 |
| 1 | 03 |
| 2 | 14 |
| 3 | FF |
| 4 | 00 |
| 5 | 19 |
| 6 | 08 |
| 7 | 00 |
| 8 | 09 |
| 9 | 08 |
| A | 00 |
| B | 00 |
| C | 09 |
| D | FF |
| E | 00 |
| F | 11 |
| 10 | 06 |
| 11 | 00 |
| 12 | 20 |
| 13 | 0A |
| 14 | FF |
| 15 | 00 |
| 16 | 09 |
| 17 | 08 |
| 18 | 00 |
| 19 | 2A |
| 1A | 0B |
| 1B | FF |

Message 1 is 20 characters, starting at character 4
End of message 1

NO SUCH

PRODUCT

RE-ENTER

End of message 2

NUMBER

OF UNITS

End of message 3

PRODUCT

SHIP DATE:

End of message 4

| LR | 10,A | |
|------|------|---|
| LR | DC,H | |
| L3 | | LOAD NEXT CHARACTER TO BE OUTPUT |
| | PI | OUTPUT CHARACTER |
| | COUT | *ANY OUTPUT CODE MAY REPLACE THE CALL TO SUB-ROUTINE COUT |
| | DS | 2 |
| | | DECREMENT CHARACTER COUNTER |
| | BNZ | L3 |
| | | RETURN FOR MORE CHARACTERS |
| | XDC | |
| | BR | T10 |
| | | AT END OF MESSAGE SEGMENT, RESTORE TIND ADDRESS TO DCO |
| OUT | | END OF PROGRAM. ANY OTHER INSTRUCTIONS MAY FOLLOW HERE |
| | | |
| | | |
| TIND | ORG | X'0800' |
| | | ORIGIN ARBITRARILY SELECTED |
| | DC | H'00' |
| | | DISPLACEMENT TO HIGH BYTE |
| | DC | H'03' |
| | | DISPLACEMENT TO LOW BYTE |
| | DC | H'16' |
| | | NUMBER OF CHARACTERS IN THIS SEGMENT |
| | DC | H'FF' |
| | | END OF MESSAGE 1 |
| | | |
| | | |
| | DC | H'0B' |
| | DC | H'FF' |

The following program assumes that the message number is in the accumulator. The program generates the specified message.

| | | | |
|------|-----|------|---|
| TGEN | LR | 0,A | SAVE MESSAGE NUMBER IN BYTE 0 |
| | DCI | TIND | LOAD TEXT INDEX STARTING ADDRESS |
| L1 | DS | 0 | DECREMENT MESSAGE COUNTER |
| | BZ | T10 | MESSAGE FOUND |
| L2 | LM | | SEEK NEXT H'FF' BYTE IN TIND |
| | COM | | |
| | BNZ | L2 | BYTE LOADED IS NOT H'FF' |
| | BR | L1 | BYTE LOADED IS H'FF' |
| T10 | LM | | MESSAGE FOUND. LOAD NEXT THREE BYTES OF TIND AND SAVE IN SCRATCHPAD |
| | LR | 0,A | BYTES 0, 1 AND 2. TEST FIRST BYTE FOR H'FF' SIGNIFYING END OF MESSAGE |
| | COM | | |
| | BZ | OUT | |
| | LM | | |
| | LR | 1,A | |
| | LM | | |
| | LR | 2,A | |
| | XDC | | SAVE TIND ADDRESS IN DC1 |
| | DCI | TEXT | LOAD TEXT ADDRESS INTO DCO |
| | LR | A,0 | ADD SCRATCHPAD BYTES 1 AND 0 TO DCO |
| | ADC | | |
| | LR | H,DC | |
| | LR | A,10 | |
| | AS | 1 | |

10.2 MULTIBYTE ADDITION AND SUBTRACTION

10.2.1 16-Bit, Binary Addition and Subtraction

The following program adds a 16-bit value in scratchpad bytes 1 (high) and 0 (low) to another 16-bit value in scratchpad bytes 3 (high) and 2 (low), as follows:

| | | | |
|----|-----|-------|---------------------------|
| | LR | A,0 | LOAD LOW ORDER |
| | | | AUGEND BYTE |
| | AS | 2 | ADD LOW ORDER |
| | | | ADDEND BYTE |
| | LR | 2,A | SAVE ANSWER |
| | LR | A,1 | LOAD HIGH ORDER |
| | | | AUGEND BYTE |
| | LNK | | ADD ANY CARRY |
| | BNO | A1 | IF NO OVERFLOW, CONTINUE |
| | BR | ERROR | MAKE ERROR EXIT FOR CARRY |
| A1 | AS | 3 | ADD HIGH ORDER |
| | | | ADDEND BYTE |
| | LR | 3,A | SAVE ANSWER |
| | BNO | NEXT | IF NO OVERFLOW, CONTINUE |
| | BR | ERROR | MAKE ERROR EXIT FOR CARRY |

To perform 16-bit binary subtraction, the two's complement of the 16-bit value in scratchpad bytes 1 and 0 is added to the 16-bit value in H. Instructions required are as follows:

| | | |
|-----|------|---------------------------------|
| LR | DC,H | MOVE SUBTRAHEND TO DC |
| LR | A,0 | LOAD LOW ORDER BYTE OF MINUEND |
| COM | | COMPLEMENT IT |
| ADC | | ADD TO SUBTRAHEND |
| LIS | 1 | ADD 1 TO SUBTRAHEND |
| ADC | | |
| LR | H,DC | RESTORE PARTIAL SUM TO H |
| LR | A,1 | LOAD HIGH ORDER BYTE OF MINUEND |
| COM | | COMPLEMENT |
| AS | 10 | ADD HU TO ACCUMULATOR |
| LR | 10,A | STORE ANSWER BACK |

To perform multibyte decimal subtraction, the routine should be changed as follows:

| | | | |
|-------|-----|---------|---------------------------|
| BUFA | EQU | H'0838' | THE CONTENTS OF BUFA |
| BUFB | EQU | H'0920' | AND BUFB ARE ADDED. THE |
| BUFC | EQU | H'077C' | RESULT IS STORED IN BUFC. |
| CNT | — | H'0A' | 10 BYTE BUFFERS ARE |
| | — | | ASSUMED. |
| ONE | LIS | CNT | USE SCRATCHPAD |
| TWO | LR | 0,A | REGISTER 0 AS A COUNTER |
| THREE | DCI | BUFC | SAVE THE ANSWER BUFFER |
| FOUR | LR | Q,DC | STARTING ADDRESS IN Q |
| FIVE | DCI | BUFA | SAVE THE SOURCE BUFFER |
| SIX | XDC | | ADDRESSES IN DCO |
| | | | AND DC1 |

10.2.2 Multibyte Binary or Decimal Addition and Subtraction

Subroutine MADD, in any of the forms and variations described in Section 7, performs multibyte binary addition.

To perform multibyte binary subtraction make changes as follows. (Refer to the program version in Section 7.2.2):

Replace

| | | | |
|-------|-----|-------|--|
| EIGHT | COM | | INITIALLY CLEAR THE CARRY BIT |
| LOOP | LR | J,W | |
| | LM | | |
| | LR | W,J | |
| NINE | LNK | | |
| with: | | | |
| EIGHT | LI | H'FF' | INITIALLY SET THE CARRY BIT BY LOADING H'FF' INTO A, THEN INCREMENTING |
| | INC | | |
| | LR | J,W | SAVE STATUS TO FORCE |
| LOOP | LM | | TWOS COMPLEMENT |
| | COM | | LOAD NEXT BYTE |
| | LR | W,J | COMPLEMENT THE ACCUMULATOR |
| NINE | LNK | | RESTORE STATUS |
| | | | ADD CARRY, IF PRESENT |

To perform multibyte decimal addition, referring again to the multibyte addition program as described in Section 7.2.2, replace

| | | | |
|-------|-----|-------|-----------------------------------|
| TWEL | AM | | ADD CORRESPONDING ADDEND BYTE |
| with: | | | |
| TWEL | AI | H'66' | PRIME AUGEND FOR DECIMAL ADDITION |
| | AMD | | ADD ADDEND DECIMAL |

| | | | |
|-------|-----|--------|--------------------------------------|
| SEVEN | DCI | BUFB | |
| EIGHT | LI | H'66' | LOAD IMMEDIATE H'66' |
| | LR | 2,A | AND SAVE FOR LATER USE |
| | LIS | 1 | INITIALLY SET CARRY TO 1 |
| LOOP | LR | 8,A | SCRATCHPAD BYTE 8 USED TO SAVE CARRY |
| | LM | | LOAD SUBTRAHEND INTO ACCUMULATOR |
| ELEV | COM | | ADDRESS MINUEND |
| | XDC | | ADD MINUEND |
| | AMD | | SAVE STATUS |
| | LR | J,W | ADD PRIOR BYTE'S CARRY |
| | AS | 8 | DECIMAL CORRECT BY |
| | ASD | 2 | ADDING H'66' |
| NNTN | BNC | TWTY+1 | TEST IF DECIMAL CORRECT |
| TWTY | LR | J,W | CREATES A CARRY |
| THRT | XDC | | IF IT DOES, SAVE CARRY |
| | | | READDRESS AUGEND BUFFER |
| FRTN | LR | H,DC | SAVE AUGEND ADDRESS IN H |
| FFTN | LR | DC,Q | LOAD ANSWER BUFFER ADDRESS |
| SXTN | ST | | STORE THE ANSWER |
| SVTN | LR | Q,DC | SAVE ANSWER BUFFER ADDRESS IN Q |
| EGTN | LR | DC,H | MOVE AUGEND ADDRESS BACK TO H |
| | LIS | 2 | LOAD CARRY FROM AMD |
| | NS | 9 | OR ASD AND WITH SAVED STATUS IN J |
| | SR | 1 | SAVE IN SCRATCHPAD BYTE 1 |
| TWT1 | DS | 0 | DECREMENT COUNTER |
| | BNZ | LOOP | RETURN FOR MORE |

10.3 MULTIPLICATION

There are a number of possible multiplication routines.

Consider first the binary multiplication of two 8-bit, positive numbers (in scratchpad bytes 0 and 1) to give a 16-bit product in scratchpad bytes 7 (high) and 6 (low). The following program performs the required multiplication:

```

*BINARY MULTIPLY SUBROUTINE
*SCRATCH REG 1 CONTAINS MULTIPLIER
*SCRATCH REG 2 CONTAINS MULTIPLICAND
*SCRATCH REGS 6 AND 7 CONTAIN PRODUCT (SR7=MSB)
BMPY LIS 8 INITIALIZE COUNTER TO 8
LR 5,A
LIS 0 ZERO PRODUCT
LR 6,A
LR 7,A
BMP1 LR A,6 SHIFT PARTIAL
AS 6 PRODUCT LEFT 1
LR 6,A
LR A,7
LNK
AS 7
LR 7,A
LR A,1 SHIFT MULTIPLIER
AS 1 LEFT 1, BY ADD
LR 1,A IF CARRY IF SET
BNC BMP2 ADD MULTIPLICAND TO
PRODUCT
LR A,2 ADD
AS 6 MULTIPLICAND
LR 6,A TO
LR A,7 PRODUCT
LNK
BMP2 LR 7,A
DS 5 DECREMENT COUNT
BNZ BMP1 NOT FINI, REPEAT
—
—

```

The above program occupies 26 bytes and executes in a maximum of 373 μ s. Contrast this with the program in Section 9.3.3 which occupies just 12 bytes, but executes in between 20 μ s and 1800.5 μ s.

Very fast decimal multiplication can be achieved using table lookups. Consider a 2-digit decimal number in scratchpad byte 0, multiplied by a 2-digit decimal number in scratchpad byte 1, to give a 4-digit answer in scratchpad bytes 7 (high) and 6 (low). The routine uses 100 bytes of ROM, to hold the following table:

```

TABX+00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
holds: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 Not Used

TABX+10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
holds: 00 01 02 03 04 05 06 07 08 09 00 00 00 00 00 00 Not Used

TABX+20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
holds: 00 02 04 06 08 10 12 14 16 18 00 00 00 00 00 00 Not Used

TABX+30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
holds: 00 03 06 09 12 15 18 21 24 27 00 00 00 00 00 00 Not Used

TABX+40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
holds: 00 04 08 12 16 20 24 28 32 36 00 00 00 00 00 00 Not Used

TABX+50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
holds: 00 05 10 15 20 25 30 35 40 45 00 00 00 00 00 00 Not Used

TABX+60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
holds: 00 06 12 18 24 30 36 42 48 54 00 00 00 00 00 00 Not Used

TABX+70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
holds: 00 07 14 21 28 35 42 49 56 63 00 00 00 00 00 00 Not Used

```

```

TABX+80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
holds: 00 08 16 24 32 40 48 56 64 72 00 00 00 00 00 00 Not Used

```

```

TABX+90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
holds: 00 09 18 27 36 45 54 63 72 81 00 00 00 00 00 00 Not Used

```

All numbers above are hexadecimal. Suppose TABX is equated to H'2000'; then byte H'2008' contains H'00'; byte H'2024' contains H'08'; byte H'2094' contains H'36'; etc.

The table lookup proceeds as follows:

```

LR A,0 ISOLATE MULTIPLIER
SL 4 LOW ORDER DIGIT
SR 4
LR 2,A STORE IN BYTE 2
LR A,1 LOAD MULTIPLICAND
SL 4 ISOLATE LOW ORDER DIGIT
AS 2 ADD MULTIPLIER LOW
ORDER DIGIT X16
DCI TABX LOAD TABLE BASE
ADDRESS
LR H,DC SAVE BASE FOR FURTHER
USE
ADC ADD ACCUMULATOR INDEX
LM LOAD PRODUCT FROM
TABLE
LR 6,A STORE IN LOW ORDER
BYTE OF ANSWER
LR A,0 LOAD MULTIPLIER
SR 4 ISOLATE HIGH ORDER DIGIT
LR 3,A SAVE IN BYTE 3
LR A,1 LOAD MULTIPLICAND
SR 4 ISOLATE HIGH ORDER DIGIT
SL 4
AS 3 ADD MULTIPLIER HIGH
ORDER DIGIT
LR DC,H LOAD TABLE BASE
ADDRESS
ADC ADD ACCUMULATOR INDEX
LM LOAD PRODUCT FROM
TABLE
LR 7,A STORE IN HIGH ORDER BYTE
OF ANSWER
LR A,1 LOAD LOW ORDER DIGIT OF
SL 4 MULTIPLICAND
AS 3 ADD HIGH ORDER DIGIT OF
MULTIPLIER
LR DC,H OBTAIN PRODUCT
ADC
LM
LR 3,A SAVE IN BYTE 3
SL 4 ADD LOW ORDER DIGIT TO
AI H'66' HIGH ORDER DIGIT OF
ASD 6 BYTE 6
LR J,W
LR 6,A ISOLATE HIGH ORDER DIGIT
LR A,3 OF PRODUCT IN LOW
SR 4 ORDER POSITION
LR W,J OF ACCUMULATOR. ADD
LNK LINK

```

| | | |
|-----|-------|---|
| AI | H'66' | |
| ASD | 7 | ADD HIGH ORDER BYTE OF ANSWER |
| LR | 7,A | RESTORE HIGH ORDER BYTE OF ANSWER |
| LR | A,1 | LOAD HIGH ORDER DIGIT OF MULTIPLICAND |
| SR | 4 | |
| SL | 4 | |
| AS | 2 | ADD LOW ORDER DIGIT OF MULTIPLIER |
| LR | DC,H | OBTAIN PRODUCT |
| ADC | | |
| LM | | SAVE IN BYTE 3 |
| LR | 3,A | |
| SL | 4 | ADD LOW ORDER DIGIT TO |
| AI | H'66' | |
| ASD | 6 | HIGH ORDER DIGIT OF BYTE 6 |
| LR | J,W | |
| LR | 6,A | |
| LR | A,3 | ISOLATE HIGH ORDER DIGIT OF PRODUCT IN LOW ORDER POSITION |
| SR | 4 | |

| | | |
|-----|-------|-----------------------------------|
| LR | W,J | |
| LNK | | OF ACCUMULATOR. ADD LINK |
| AI | H'66' | |
| ASD | 7 | ADD HIGH ORDER BYTE OF ANSWER |
| LR | 7,A | RESTORE HIGH ORDER BYTE OF ANSWER |

More compact versions of this program could be written, but they would take longer to execute.

10.4 DIVISION

Division of positive numbers can be performed by a program using successive subtraction as follows:

- 1) Zero the answer
- 2) Subtract the divisor from the dividend
- 3) Test for a negative result
- 4) For a positive result, increment the answer and return to 2
- 5) For a negative result, the division is finished. Add the divisor to the dividend to obtain remainder.

APPENDIX A – BINARY NUMBER SYSTEM

The binary number system is a system of counting which utilizes the digits 1 and 0 to represent numeric quantities. The binary digits, referred to as BITS, are arranged in a sequence of decreasing significance based upon powers of two. Each bit is numbered. By convention, the most significant bit is on the left, and the least significant bit is on the right.

For example, consider the binary number:

| | | | | | | |
|-------------------|-------|-------|-------|-------|-------|-------|
| Binary number | 0 | 1 | 1 | 0 | 0 | 1 |
| Bit number | 5 | 4 | 3 | 2 | 1 | 0 |
| Power of base two | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
| Significance | 32 | 16 | 8 | 4 | 2 | 1 |

As in any number system, the quantity represented by a binary number is calculated by multiplying each digit by its significance, then summing products.

The binary number example is evaluated as follows:

$$\begin{aligned} \text{Quantity} &= 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 0 + 16 + 8 + 0 + 0 + 1 \\ &= 25 \end{aligned}$$

Binary numbers may be used to represent any real number positive or negative.

Non-integer numbers are represented in the same binary format shown above except that the significance of the bits changes. To indicate the correct interpretation of a binary number, a "binary point" (which is analogous to a decimal point in the decimal number system) is inserted. Consider the binary number below:

| | | | | | | | | |
|-------------------|-------|-------|-------|-------|-------|-------|---------------|----------|
| Binary number | 0 | 1 | 1 | 0 | 0 | 1 | . | 1 |
| Bit number | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 |
| Power of base two | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | 2^{-1} |
| Significance | 32 | 16 | 8 | 4 | 2 | 1 | $\frac{1}{2}$ | |

The number is evaluated as follows:

$$\begin{aligned} \text{Quantity} &= 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} \\ &= 0 + 16 + 8 + 0 + 0 + 0 + 0 + .5 \\ &= 25.5 \end{aligned}$$

The bits of a binary number may be grouped in fours and transposed into the hexadecimal number system which includes the following digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

The following example illustrates the procedure:

| | |
|-------------------------|--------------------|
| 10011100 | Binary number |
| <u>1001</u> <u>1100</u> | |
| 9 A | Hexadecimal number |

In this manual, hexadecimal numbers are written within quotation marks and preceded by an H. Consider the example:

H'27', H'AE10', H'F'

The octal number system includes the following digits:

0, 1, 2, 3, 4, 5, 6, 7.

Binary numbers are transposed into octal numbers by arranging the bits into groups of three as illustrated below:

| | |
|-----------------------|---------------|
| 101001 | Binary number |
| <u>101</u> <u>001</u> | |
| 5 1 | Octal number |

The Indirect Scratchpad Address Register (ISAR) uses two octal digits to address 64 scratchpad registers.

Octal numbers are written within quotation marks preceded by an O as follows:

O'27', O'3', O'3270'

Table A-1 illustrates the relationship between binary, decimal, hexadecimal and octal numbers.

| BINARY | DECIMAL | HEXADECIMAL | OCTAL |
|---------|---------|-------------|-------|
| 0 0 0 0 | 0 | 0 | 0 |
| 0 0 0 1 | 1 | 1 | 1 |
| 0 0 1 0 | 2 | 2 | 2 |
| 0 0 1 1 | 3 | 3 | 3 |
| 0 1 0 0 | 4 | 4 | 4 |
| 0 1 0 1 | 5 | 5 | 5 |
| 0 1 1 0 | 6 | 6 | 6 |
| 0 1 1 1 | 7 | 7 | 7 |
| 1 0 0 0 | 8 | 8 | 10 |
| 1 0 0 1 | 9 | 9 | 11 |
| 1 0 1 0 | 10 | A | 12 |
| 1 0 1 1 | 11 | B | 13 |
| 1 1 0 0 | 12 | C | 14 |
| 1 1 0 1 | 13 | D | 15 |
| 1 1 1 0 | 14 | E | 16 |
| 1 1 1 1 | 15 | F | 17 |

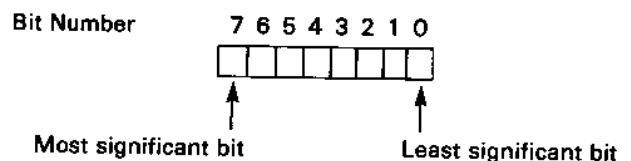
Table A-1. Binary, Decimal, Hexadecimal and Octal Numbers

THE BYTE

The Fairchild F8 microprocessor is an 8-bit device, which means that data is handled in eight binary digit (or one byte) units. An 8-bit byte may represent 256 (2^8) possible permutations of eight digits.

When referencing the 8-bit byte, this manual has established the following conventions.

The bits are numbered from right to left with numbers 0 through 7. The most significant bit is on the left; the least significant bit is on the right.



An 8-bit byte may represent an instruction object code, an ASCII code or a data word.

An 8-bit data word may be interpreted as a signed binary number with a value of from 127 to -128 as illustrated in Table A-2.

It will become clear after reading the sections which follow on binary arithmetic, that the signed binary number system is a natural fallout of two's complement subtraction.

| BINARY | DECIMAL | HEXADECIMAL |
|----------|---------|-------------|
| 10000000 | -128 | 80 |
| 10000001 | -127 | 81 |
| 10000010 | -126 | 82 |
| — | — | — |
| — | — | — |
| — | — | — |
| 11111110 | -2 | FE |
| 11111111 | -1 | FF |
| 00000000 | 0 | 0 |
| 00000001 | 1 | 1 |
| 00000010 | 2 | 2 |
| — | — | — |
| — | — | — |
| — | — | — |
| 01111101 | +125 | 7D |
| 01111110 | +126 | 7E |
| 01111111 | +127 | 7F |

Table A-2. Signed Binary Numeric Interpretations

Binary Number Addition

Addition of binary numbers is accomplished by following three rules.

- 1) $\begin{matrix} 1 & \text{bit} \\ + 1 & \text{bit} \\ \hline 0 & \text{bit} \end{matrix}$ + a carry bit to the next significant bit
- 2) $\begin{matrix} 1 & \text{bit} \\ + 0 & \text{bit} \\ \hline 1 & \text{bit} \end{matrix}$
- 3) $\begin{matrix} 0 & \text{bit} \\ + 0 & \text{bit} \\ \hline 0 & \text{bit} \end{matrix}$

Consider the addition of two positive 8-bit binary numbers:

| | |
|------------|-------------------|
| Bit Number | 7 6 5 4 3 2 1 0 |
| H'93' | 1 0 0 1 0 0 1 1 |
| +H'A8' | 1 0 1 0 1 0 0 0 |
| H'3B' | 1 0 0 1 1 1 0 1 1 |
| | 1 0 1 1 1 0 1 1 |

↑
Carry Bit

A carry out of bit 7 has occurred as a result of the addition. The carry bit is set to indicate that the results of the addition cannot be represented in the existing 8-bits. However, if the carry bit represents the next higher significant bit, the results are valid.

In a multiple byte addition, the carry bit from the most significant bit position of a byte is added to the least significant

bit of the next (higher order) byte as follows:

| | | |
|----------|-----------------|-----------------|
| H'13E2' | 0 0 0 1 0 0 1 1 | 1 1 1 0 0 0 1 0 |
| +H'4747' | 0 1 0 0 0 1 1 1 | 0 1 0 0 0 1 1 1 |
| H'5B29' | 0 1 0 1 1 0 1 0 | 0 0 1 0 1 0 0 1 |
| | 1 | |
| | 0 1 0 1 1 0 1 1 | |

Binary Number Subtraction

Subtracting a binary number is the same as adding the two's complement of the number.

The two's complement of a number is generated by complementing the number (replacing 0 with 1 and 1 with 0) and adding one to the complement. Here is an example:

| | |
|------------------|-----------------|
| H'3C' | 0 0 1 1 1 1 0 0 |
| one's complement | 1 1 0 0 0 0 1 1 |
| | 1 |
| two's complement | 1 1 0 0 0 1 0 0 |

Observe that negative numbers in Table A-2 are the two's complement of their positive equivalents. In this fashion, an 8-bit number can contain sign and value information for numbers between 128 and -127.

When adding signed binary numbers, care must be taken to indicate when the result exceeds the boundaries of the two's complement notation.

To exemplify the need for such indicators, consider some simple examples using the set of 3-bit signed binary numbers from 3 to -4.

| Signed Binary Numbers | Decimal |
|-----------------------|---------|
| 011 | 3 |
| 010 | 2 |
| 001 | 1 |
| 000 | 0 |
| 111 | -1 |
| 110 | -2 |
| 101 | -3 |
| 100 | -4 |

Any number greater than 3 or less than -4 is outside the boundaries of the set of 3-bit signed binary numbers.

The addition of two numbers within this set may result in a number which is not defined as part of the set.

Consider the addition of two numbers with like signs:

- | | | | | | | | | | | | | | |
|---|------|-----|-----|-----|----|-----|--|----|-----|-----|-----|----|------|
| <p>1) Bit No. 210</p> <table style="border-collapse: collapse;"> <tr><td style="padding-right: 10px;">2</td><td style="text-align: right;">010</td></tr> <tr><td style="padding-right: 10px;">+ 1</td><td style="text-align: right;">001</td></tr> <tr><td style="padding-right: 10px;">3</td><td style="text-align: right; border-top: 1px solid black;">011</td></tr> </table> <p style="margin-left: 40px;">no carry</p> | 2 | 010 | + 1 | 001 | 3 | 011 | <p>2) Bit No. 210</p> <table style="border-collapse: collapse;"> <tr><td style="padding-right: 10px;">3</td><td style="text-align: right;">011</td></tr> <tr><td style="padding-right: 10px;">+ 1</td><td style="text-align: right;">001</td></tr> <tr><td style="padding-right: 10px;">4</td><td style="text-align: right; border-top: 1px solid black;">100</td></tr> </table> <p style="margin-left: 40px;">carry</p> | 3 | 011 | + 1 | 001 | 4 | 100 |
| 2 | 010 | | | | | | | | | | | | |
| + 1 | 001 | | | | | | | | | | | | |
| 3 | 011 | | | | | | | | | | | | |
| 3 | 011 | | | | | | | | | | | | |
| + 1 | 001 | | | | | | | | | | | | |
| 4 | 100 | | | | | | | | | | | | |
| <p>3) Bit No. 210</p> <table style="border-collapse: collapse;"> <tr><td style="padding-right: 10px;">-3</td><td style="text-align: right;">101</td></tr> <tr><td style="padding-right: 10px;">-1</td><td style="text-align: right;">111</td></tr> <tr><td style="padding-right: 10px;">-4</td><td style="text-align: right; border-top: 1px solid black;">100</td></tr> </table> <p style="margin-left: 40px;">two carries</p> | -3 | 101 | -1 | 111 | -4 | 100 | <p>4) Bit No. 210</p> <table style="border-collapse: collapse;"> <tr><td style="padding-right: 10px;">-3</td><td style="text-align: right;">101</td></tr> <tr><td style="padding-right: 10px;">-2</td><td style="text-align: right;">110</td></tr> <tr><td style="padding-right: 10px;">-5</td><td style="text-align: right; border-top: 1px solid black;">1011</td></tr> </table> <p style="margin-left: 40px;">carry</p> | -3 | 101 | -2 | 110 | -5 | 1011 |
| -3 | 101 | | | | | | | | | | | | |
| -1 | 111 | | | | | | | | | | | | |
| -4 | 100 | | | | | | | | | | | | |
| -3 | 101 | | | | | | | | | | | | |
| -2 | 110 | | | | | | | | | | | | |
| -5 | 1011 | | | | | | | | | | | | |

In example 1, no carry out of the two high order bits occurred. The result is defined and valid.

In example 2, carry out from the bit which precedes the sign bit (carry out from bit 2) occurred. The result is undefined and therefore invalid.

In example 3, a carry from bit 2 and 3 occurred. The result is defined and valid.

In example 4, a carry from the sign bit occurred. The result is undefined and invalid.

The explanation of the four examples illustrates the rules which govern the error indication mechanism in the Fairchild F8 microprocessor. If the addition of two 8-bit numbers causes a result which is outside the boundary defined for 8-bit signed binary numbers, (illustrated in Table A-1), an overflow status bit is set.

The overflow status bit is defined as the EXCLUSIVE-OR of the carry out of bit 6 and the carry out of bit 7. (EXCLUSIVE-OR is defined later in this appendix.)

Consider binary number subtraction, (the addition of a binary number to a two's complement number).

| | | | | | | | | | |
|----|------------------|---|---|---|---|---|---|---|---|
| 1) | Bit No. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | H'52' | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| | two's complement | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| | H'34' | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| | -H'52' | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| | -H'18' | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| | two's complement | | | | | | | | |
| | H'18' | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2) | Bit No. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | H'2A' | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| | two's complement | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| | H'B6' | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| | -H'2A' | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| | H'8A' | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

In example 1, the subtrahend is larger than the minuend, indicating a negative answer. In unsigned binary number arithmetic, a negative result is indicated by no carry out of the most significant bit and is in two's complement form. There is no overflow because there is no carry out of either bit 6 or bit 7.

In example 2, the subtrahend is smaller than the minuend indicating a positive answer. In unsigned binary arithmetic, a positive result is indicated by a carry from the most significant bit position and is in straight binary form. There is no overflow because there is a carry out of both bit 6 and bit 7.

Multiplication of binary numbers may be performed in two ways: repetitive addition or in the fashion illustrated below, which is similar to the long hand method for multiplying decimal numbers:

| | |
|---------|-------------------|
| Decimal | Binary |
| 91 | 1 0 1 1 0 1 1 |
| x 5 | 1 0 1 |
| 455 | 1 0 1 1 0 1 1 |
| | 0 0 0 0 0 0 0 |
| | 1 0 1 1 0 1 1 |
| | 1 1 1 0 0 0 1 1 1 |

Division of binary numbers may be accomplished by repetitive subtraction of one operand from another, or by an operation similar to long hand division:

| | |
|-------|-----------|
| 7 | 111 |
| 3) 21 | 11) 10101 |
| | 11 |
| | 100 |
| | 11 |
| | 11 |
| | 11 |
| | 0 |

COMPUTER LOGIC

Assembly language instructions exist which perform logical operations on operands. Three such logical operations are described below (logical-OR, AND, and EXCLUSIVE-OR).

The logical-OR operation is illustrated for the two operands I and J with the statement:

If I or J equals 1, then the result is 1. Otherwise, the result is zero.

The symbol used to indicate the logical-OR operation is the sign (V). Consider the logical-OR of two binary numbers:

$A \vee B = C$ (read A "or" B equals C)

| | |
|---|-------|
| A | 11010 |
| B | 01100 |
| C | 11110 |

The logical AND operation is illustrated for the two operands I and J with the following statement:

If both I and J are 1, then the result is 1. Otherwise, the result is zero.

The symbol used for the logical AND operation is (\wedge).

Consider the logical AND of two binary numbers:

$A \wedge B = C$ (read A "and" B equals C)

| | |
|---|-------|
| A | 11010 |
| B | 01100 |
| C | 01000 |

The logical EXCLUSIVE-OR operation is illustrated for the operands I and J with the following statement:

If both I and J equal 1 or both I and J equal 0, the result is zero, otherwise the result is 1.

The symbol used to indicate the logical EXCLUSIVE-OR operation is a circled sign (\oplus).

Consider the logical EXCLUSIVE-OR of two binary numbers:

$A \oplus B = C$ (read A "EXCLUSIVE-OR" with B equals C)

| | |
|---|-------|
| A | 11010 |
| B | 01100 |
| C | 10110 |

APPENDIX B – ASCII CODES

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|--------------------|---------------------|
| NULL | 00 |
| SOM | 01 |
| EOA | 02 |
| EOM | 03 |
| EOT | 04 |
| WRU | 05 |
| RU | 06 |
| BELL | 07 |
| FE | 08 |
| H. Tab | 09 |
| Line Feed | 0A |
| V. Tab | 0B |
| Form | 0C |
| Return | 0D |
| SO | 0E |
| SI | 0F |
| DCO | 10 |
| X-On | 11 |
| Tape Aux. On | 12 |
| X-Off | 13 |
| Tape Aux. Off | 14 |
| Error | 15 |
| Sync | 16 |
| LEM | 17 |
| S0 | 18 |
| S1 | 19 |
| S2 | 1A |
| S3 | 1B |
| S4 | 1C |
| S5 | 1D |
| S6 | 1E |
| S7 | 1F |

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|--------------------|---------------------|
| ACK | 7C |
| Alt. Mode | 7D |
| Rubout | 7F |
| | 21 |
| " | 22 |
| # | 23 |
| \$ | 24 |
| % | 25 |
| & | 26 |
| ' | 27 |
| (| 28 |
|) | 29 |
| * | 2A |
| + | 2B |
| , | 2C |
| - | 2D |
| . | 2E |
| / | 2F |
| : | 3A |
| ; | 3B |
| < | 3C |
| = | 3D |
| > | 3F |
| ? | 3F |
| [| 5B |
| \ | 5C |
|] | 5D |
| ↑ | 5E |
| ↓ | 5F |
| @ | 40 |
| blank | 20 |
| 0 | 30 |

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|--------------------|---------------------|
| 1 | 31 |
| 2 | 32 |
| 3 | 33 |
| 4 | 34 |
| 5 | 35 |
| 6 | 36 |
| 7 | 37 |
| 8 | 38 |
| 9 | 39 |
| A | 41 |
| B | 42 |
| C | 43 |
| D | 44 |
| E | 45 |
| F | 46 |
| G | 47 |
| H | 48 |
| I | 49 |
| J | 4A |
| K | 4B |
| L | 4C |
| M | 4D |
| N | 4E |
| O | 4F |
| P | 50 |
| Q | 51 |
| R | 52 |
| S | 53 |
| T | 54 |
| U | 55 |
| V | 56 |
| W | 57 |
| X | 58 |
| Y | 59 |
| Z | 5A |

APPENDIX C – CONVERSION TABLES/TIMER COUNTS

POWERS OF TWO

$$2^n \quad n \quad 2^n$$

| | | | |
|---------------------|----|---|-----|
| 1 | 0 | 1.0 | |
| 2 | 1 | 0.5 | |
| 4 | 2 | 0.25 | |
| 8 | 3 | 0.125 | |
| 16 | 4 | 0.0625 | 5 |
| 32 | 5 | 0.03125 | 25 |
| 64 | 6 | 0.015625 | |
| 128 | 7 | 0.0078125 | 5 |
| 256 | 8 | 0.00390625 | 25 |
| 512 | 9 | 0.001953125 | 125 |
| 1024 | 10 | 0.0009765625 | 5 |
| 2048 | 11 | 0.00048828125 | 25 |
| 4096 | 12 | 0.000244140625 | 625 |
| 8192 | 13 | 0.0001220703125 | 5 |
| 16384 | 14 | 0.00006103515625 | 25 |
| 32768 | 15 | 0.000030517578125 | 125 |
| 65536 | 16 | 0.0000152587890625 | 5 |
| 131072 | 17 | 0.00000762939453125 | 25 |
| 262144 | 18 | 0.000003814697265625 | 625 |
| 524288 | 19 | 0.0000019073486328125 | 5 |
| 1048576 | 20 | 0.00000095367431640625 | 25 |
| 2097152 | 21 | 0.000000476837158203125 | 125 |
| 4194304 | 22 | 0.0000002384185791015625 | 5 |
| 8388608 | 23 | 0.00000011920928955078125 | 25 |
| 16777216 | 24 | 0.000000059604644775390625 | 625 |
| 33554432 | 25 | 0.0000000298023223876953125 | 5 |
| 67108864 | 26 | 0.00000001490116119384765625 | 25 |
| 134217728 | 27 | 0.000000007450580596923828125 | 125 |
| 268435456 | 28 | 0.0000000037252902984619140625 | 5 |
| 536870912 | 29 | 0.00000000186264514923095703125 | 25 |
| 1073741824 | 30 | 0.000000000931322574615478515625 | 625 |
| 2147483648 | 31 | 0.0000000004656612873077392578125 | 5 |
| 4294967296 | 32 | 0.00000000023283064365386962890625 | 25 |
| 8589934592 | 33 | 0.000000000116415321826934814453125 | 125 |
| 17179869184 | 34 | 0.0000000000582076609134674072265625 | 5 |
| 34359738368 | 35 | 0.00000000002910383045673370361328125 | 25 |
| 68719476736 | 36 | 0.000000000014551915228366851806640625 | 625 |
| 137438953472 | 37 | 0.0000000000072759576141834259033203125 | 5 |
| 274877906944 | 38 | 0.00000000000363797880709171295166015625 | 25 |
| 549755813888 | 39 | 0.000000000001818989403545856475830078125 | 125 |
| 1099511627776 | 40 | 0.0000000000009094947017729282379150390625 | 5 |
| 2199023255552 | 41 | 0.00000000000045474735088646411895751953125 | 25 |
| 4398046511104 | 42 | 0.000000000000227373675443232059478759765625 | 625 |
| 8796093022208 | 43 | 0.000000000000113686837216160297393798828125 | 5 |
| 17592186044416 | 44 | 0.00000000000005684341886080801486968994140625 | 25 |
| 35184372088832 | 45 | 0.000000000000028421709430404007434844970703125 | 125 |
| 70368744177664 | 46 | 0.0000000000000142108547152020037174224853515625 | 5 |
| 140737488355328 | 47 | 0.00000000000000710542735760100185871124267578125 | 25 |
| 281474976710656 | 48 | 0.000000000000003552713678800500929355621337890625 | 625 |
| 562949953421312 | 49 | 0.0000000000000017763568394002504646778106689453125 | 5 |
| 1125899906842624 | 50 | 0.00000000000000088817841970012523233890533447265625 | 25 |
| 2251799813685248 | 51 | 0.000000000000000444089209850062616169452667236328125 | 125 |
| 4503599627370496 | 52 | 0.0000000000000002220446049250313080847263336181640625 | 5 |
| 9007199254740992 | 53 | 0.00000000000000011102230246251565404236316680908203125 | 25 |
| 18014398509481984 | 54 | 0.000000000000000055511151231257827021181583404541015625 | 625 |
| 36028797018963968 | 55 | 0.000000000000000027755756156289135105907917022705078125 | 5 |
| 72057594037927936 | 56 | 0.0000000000000000138777878078145675529539585113525390625 | 25 |
| 144115188075855872 | 57 | 0.000000000000000006938893903907228377647697925567676950125 | 125 |
| 288230376151711744 | 58 | 0.0000000000000000034694469519536141888238489627838134765625 | 5 |
| 576460752303423488 | 59 | 0.00000000000000000173472347597680709441192448139190673828125 | 125 |
| 1152921504606846976 | 60 | 0.000000000000000000867361737988403547205962240695953369140625 | 625 |
| 2305843009213693952 | 61 | 0.0000000000000000004336808689942017736029811203479766845703125 | 5 |
| 4611686018427387904 | 62 | 0.00000000000000000021684043449710088680149056017398834228515625 | 25 |
| 9223372036854775808 | 63 | 0.000000000000000000108420217248550443400745280086994171142578125 | 125 |

TABLE OF POWERS OF SIXTEEN₁₀

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

TABLE OF POWERS OF 10₁₆

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

HEXADECIMAL-DECIMAL INTEGER CONVERSION

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

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

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 000 | 0000 | 0001 | 0002 | 0003 | 0004 | 0005 | 0006 | 0007 | 0008 | 0009 | 0010 | 0011 | 0012 | 0013 | 0014 | 0015 |
| 010 | 0016 | 0017 | 0018 | 0019 | 0020 | 0021 | 0022 | 0023 | 0024 | 0025 | 0026 | 0027 | 0028 | 0029 | 0030 | 0031 |
| 020 | 0032 | 0033 | 0034 | 0035 | 0036 | 0037 | 0038 | 0039 | 0040 | 0041 | 0042 | 0043 | 0044 | 0045 | 0046 | 0047 |
| 030 | 0048 | 0049 | 0050 | 0051 | 0052 | 0053 | 0054 | 0055 | 0056 | 0057 | 0058 | 0059 | 0060 | 0061 | 0062 | 0063 |
| 040 | 0064 | 0065 | 0066 | 0067 | 0068 | 0069 | 0070 | 0071 | 0072 | 0073 | 0074 | 0075 | 0076 | 0077 | 0078 | 0079 |
| 050 | 0080 | 0081 | 0082 | 0083 | 0084 | 0085 | 0086 | 0087 | 0088 | 0089 | 0090 | 0091 | 0092 | 0093 | 0094 | 0095 |
| 060 | 0096 | 0097 | 0098 | 0099 | 0100 | 0101 | 0102 | 0103 | 0104 | 0105 | 0106 | 0107 | 0108 | 0109 | 0110 | 0111 |
| 070 | 0112 | 0113 | 0114 | 0115 | 0116 | 0117 | 0118 | 0119 | 0120 | 0121 | 0122 | 0123 | 0124 | 0125 | 0126 | 0127 |
| 080 | 0128 | 0129 | 0130 | 0131 | 0132 | 0133 | 0134 | 0135 | 0136 | 0137 | 0138 | 0139 | 0140 | 0141 | 0142 | 0143 |
| 090 | 0144 | 0145 | 0146 | 0147 | 0148 | 0149 | 0150 | 0151 | 0152 | 0153 | 0154 | 0155 | 0156 | 0157 | 0158 | 0159 |
| 0A0 | 0160 | 0161 | 0162 | 0163 | 0164 | 0165 | 0166 | 0167 | 0168 | 0169 | 0170 | 0171 | 0172 | 0173 | 0174 | 0175 |
| 0B0 | 0176 | 0177 | 0178 | 0179 | 0180 | 0181 | 0182 | 0183 | 0184 | 0185 | 0186 | 0187 | 0188 | 0189 | 0190 | 0191 |
| 0C0 | 0192 | 0193 | 0194 | 0195 | 0196 | 0197 | 0198 | 0199 | 0200 | 0201 | 0202 | 0203 | 0204 | 0205 | 0206 | 0207 |
| 0D0 | 0208 | 0209 | 0210 | 0211 | 0212 | 0213 | 0214 | 0215 | 0216 | 0217 | 0218 | 0219 | 0220 | 0221 | 0222 | 0223 |
| 0E0 | 0224 | 0225 | 0226 | 0227 | 0228 | 0229 | 0230 | 0231 | 0232 | 0233 | 0234 | 0235 | 0236 | 0237 | 0238 | 0239 |
| 0F0 | 0240 | 0241 | 0242 | 0243 | 0244 | 0245 | 0246 | 0247 | 0248 | 0249 | 0250 | 0251 | 0252 | 0253 | 0254 | 0255 |

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 100 | 0256 | 0257 | 0258 | 0259 | 0260 | 0261 | 0262 | 0263 | 0264 | 0265 | 0266 | 0267 | 0268 | 0269 | 0270 | 0271 |
| 110 | 0272 | 0273 | 0274 | 0275 | 0276 | 0277 | 0278 | 0279 | 0280 | 0281 | 0282 | 0283 | 0284 | 0285 | 0286 | 0287 |
| 120 | 0288 | 0289 | 0290 | 0291 | 0292 | 0293 | 0294 | 0295 | 0296 | 0297 | 0298 | 0299 | 0300 | 0301 | 0302 | 0303 |
| 130 | 0304 | 0305 | 0306 | 0307 | 0308 | 0309 | 0310 | 0311 | 0312 | 0313 | 0314 | 0315 | 0316 | 0317 | 0318 | 0319 |
| 140 | 0320 | 0321 | 0322 | 0323 | 0324 | 0325 | 0326 | 0327 | 0328 | 0329 | 0330 | 0331 | 0331 | 0333 | 0334 | 0335 |
| 150 | 0336 | 0337 | 0338 | 0339 | 0340 | 0341 | 0342 | 0343 | 0344 | 0345 | 0346 | 0347 | 0348 | 0349 | 0350 | 0351 |
| 160 | 0352 | 0353 | 0354 | 0355 | 0356 | 0357 | 0358 | 0359 | 0360 | 0361 | 0362 | 0363 | 0364 | 0365 | 0366 | 0367 |
| 170 | 0368 | 0369 | 0370 | 0371 | 0372 | 0373 | 0374 | 0375 | 0376 | 0377 | 0378 | 0379 | 0380 | 0381 | 0382 | 0383 |
| 180 | 0384 | 0385 | 0386 | 0387 | 0388 | 0389 | 0390 | 0391 | 0392 | 0393 | 0394 | 0395 | 0396 | 0397 | 0398 | 0399 |
| 190 | 0400 | 0401 | 0402 | 0403 | 0404 | 0405 | 0406 | 0407 | 0408 | 0409 | 0410 | 0411 | 0412 | 0413 | 0414 | 0415 |
| 1A0 | 0416 | 0417 | 0418 | 0419 | 0420 | 0421 | 0422 | 0423 | 0424 | 0425 | 0426 | 0427 | 0428 | 0429 | 0430 | 0431 |
| 1B0 | 0432 | 0433 | 0434 | 0435 | 0436 | 0437 | 0438 | 0439 | 0440 | 0441 | 0442 | 0443 | 0444 | 0445 | 0446 | 0447 |
| 1C0 | 0448 | 0449 | 0450 | 0451 | 0452 | 0453 | 0454 | 0455 | 0456 | 0457 | 0458 | 0459 | 0460 | 0461 | 0462 | 0463 |
| 1D0 | 0464 | 0465 | 0466 | 0467 | 0468 | 0469 | 0470 | 0471 | 0472 | 0473 | 0474 | 0475 | 0476 | 0477 | 0478 | 0479 |
| 1E0 | 0480 | 0481 | 0482 | 0483 | 0484 | 0485 | 0486 | 0487 | 0488 | 0489 | 0490 | 0491 | 0492 | 0493 | 0494 | 0495 |
| 1F0 | 0496 | 0497 | 0498 | 0499 | 0500 | 0501 | 0502 | 0503 | 0504 | 0505 | 0506 | 0507 | 0508 | 0509 | 0510 | 0511 |
| 200 | 0512 | 0513 | 0514 | 0515 | 0516 | 0517 | 0518 | 0519 | 0520 | 0521 | 0522 | 0523 | 0524 | 0525 | 0526 | 0527 |
| 210 | 0528 | 0529 | 0530 | 0531 | 0532 | 0533 | 0534 | 0535 | 0536 | 0537 | 0538 | 0539 | 0540 | 0541 | 0542 | 0543 |
| 220 | 0544 | 0545 | 0546 | 0547 | 0548 | 0549 | 0550 | 0551 | 0552 | 0553 | 0554 | 0555 | 0556 | 0557 | 0558 | 0559 |
| 230 | 0560 | 0561 | 0562 | 0563 | 0564 | 0565 | 0566 | 0567 | 0568 | 0569 | 0570 | 0571 | 0572 | 0573 | 0574 | 0575 |
| 240 | 0576 | 0577 | 0578 | 0579 | 0580 | 0581 | 0582 | 0583 | 0584 | 0585 | 0586 | 0587 | 0588 | 0589 | 0590 | 0591 |
| 250 | 0592 | 0593 | 0594 | 0595 | 0596 | 0597 | 0598 | 0599 | 0600 | 0601 | 0602 | 0603 | 0604 | 0605 | 0606 | 0607 |
| 260 | 0608 | 0609 | 0610 | 0611 | 0612 | 0613 | 0614 | 0615 | 0616 | 0617 | 0618 | 0619 | 0620 | 0621 | 0622 | 0623 |
| 270 | 0624 | 0625 | 0626 | 0627 | 0628 | 0629 | 0630 | 0631 | 0632 | 0633 | 0634 | 0635 | 0636 | 0637 | 0638 | 0639 |
| 280 | 0640 | 0641 | 0642 | 0643 | 0644 | 0645 | 0646 | 0647 | 0648 | 0649 | 0650 | 0651 | 0652 | 0653 | 0654 | 0655 |
| 290 | 0656 | 0657 | 0658 | 0659 | 0660 | 0661 | 0662 | 0663 | 0664 | 0665 | 0666 | 0667 | 0668 | 0669 | 0670 | 0671 |
| 2A0 | 0672 | 0673 | 0674 | 0675 | 0676 | 0677 | 0678 | 0679 | 0680 | 0681 | 0682 | 0683 | 0684 | 0685 | 0686 | 0687 |
| 2B0 | 0688 | 0689 | 0690 | 0691 | 0692 | 0693 | 0694 | 0695 | 0696 | 0697 | 0698 | 0699 | 0700 | 0701 | 0702 | 0703 |
| 2C0 | 0704 | 0705 | 0706 | 0707 | 0708 | 0709 | 0710 | 0711 | 0712 | 0713 | 0714 | 0715 | 0716 | 0717 | 0718 | 0719 |
| 2D0 | 0720 | 0721 | 0722 | 0723 | 0724 | 0725 | 0726 | 0727 | 0728 | 0729 | 0730 | 0731 | 0732 | 0733 | 0734 | 0735 |
| 2E0 | 0736 | 0737 | 0738 | 0739 | 0740 | 0741 | 0742 | 0743 | 0744 | 0745 | 0746 | 0747 | 0748 | 0749 | 0750 | 0751 |
| 2F0 | 0752 | 0753 | 0754 | 0755 | 0756 | 0757 | 0758 | 0759 | 0760 | 0761 | 0762 | 0763 | 0764 | 0765 | 0766 | 0767 |
| 300 | 0768 | 0769 | 0770 | 0771 | 0772 | 0773 | 0774 | 0775 | 0776 | 0777 | 0778 | 0779 | 0780 | 0781 | 0782 | 0783 |
| 310 | 0784 | 0785 | 0786 | 0787 | 0788 | 0789 | 0790 | 0791 | 0792 | 0793 | 0794 | 0795 | 0796 | 0797 | 0798 | 0799 |
| 320 | 0800 | 0301 | 0802 | 0803 | 0804 | 0805 | 0806 | 0807 | 0808 | 0809 | 0810 | 0811 | 0812 | 0813 | 0814 | 0815 |
| 330 | 0816 | 0817 | 0818 | 0819 | 0820 | 0821 | 0822 | 0823 | 0824 | 0825 | 0826 | 0827 | 0828 | 0829 | 0830 | 0831 |
| 340 | 0832 | 0833 | 0834 | 0835 | 0836 | 0837 | 0838 | 0839 | 0840 | 0841 | 0842 | 0843 | 0844 | 0845 | 0846 | 0847 |
| 350 | 0848 | 0849 | 0850 | 0851 | 0852 | 0853 | 0854 | 0855 | 0856 | 0857 | 0858 | 0859 | 0860 | 0861 | 0862 | 0863 |
| 360 | 0864 | 0865 | 0866 | 0867 | 0868 | 0869 | 0870 | 0871 | 0872 | 0873 | 0874 | 0875 | 0876 | 0877 | 0878 | 0879 |
| 370 | 0880 | 0881 | 0882 | 0883 | 0884 | 0885 | 0886 | 0887 | 0888 | 0889 | 0890 | 0891 | 0892 | 0893 | 0894 | 0895 |
| 380 | 0896 | 0897 | 0898 | 0899 | 0900 | 0901 | 0902 | 0903 | 0904 | 0905 | 0906 | 0907 | 0908 | 0909 | 0910 | 0911 |
| 390 | 0212 | 0913 | 0914 | 0915 | 0916 | 0917 | 0918 | 0919 | 0920 | 0921 | 0922 | 0923 | 0924 | 0925 | 0926 | 0927 |
| 3A0 | 0928 | 0929 | 0930 | 0931 | 0932 | 0933 | 0934 | 0935 | 0936 | 0937 | 0938 | 0939 | 0940 | 0941 | 0942 | 0943 |
| 3B0 | 0944 | 0945 | 0946 | 0947 | 0948 | 0949 | 0950 | 0951 | 0952 | 0953 | 0954 | 0955 | 0956 | 0957 | 0958 | 0959 |
| 3C0 | 0960 | 0961 | 0962 | 0963 | 0964 | 0965 | 0966 | 0967 | 0968 | 0969 | 0970 | 0971 | 0972 | 0973 | 0974 | 0975 |
| 3D0 | 0976 | 0977 | 0978 | 0979 | 0980 | 0981 | 0982 | 0983 | 0984 | 0985 | 0986 | 0987 | 0988 | 0989 | 0990 | 0991 |
| 3E0 | 0992 | 0993 | 0994 | 0995 | 0996 | 0997 | 0998 | 0999 | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 |
| 3F0 | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 | 1016 | 1017 | 1018 | 1019 | 1020 | 1021 | 1022 | 1023 |

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

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

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

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

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| A00 | 2560 | 2561 | 2562 | 2563 | 2564 | 2565 | 2566 | 2567 | 2568 | 2569 | 2570 | 2571 | 2572 | 2573 | 2574 | 2575 |
| A10 | 2576 | 2577 | 2578 | 2579 | 2580 | 2581 | 2582 | 2583 | 2584 | 2585 | 2586 | 2587 | 2588 | 2589 | 2590 | 2591 |
| A20 | 2592 | 2593 | 2594 | 2595 | 2596 | 2597 | 2598 | 2599 | 2600 | 2601 | 2602 | 2603 | 2604 | 2605 | 2606 | 2607 |
| A30 | 2608 | 2609 | 2610 | 2611 | 2612 | 2613 | 2614 | 2615 | 2616 | 2617 | 2618 | 2619 | 2620 | 2621 | 2622 | 2623 |
| A40 | 2624 | 2625 | 2626 | 2627 | 2628 | 2629 | 2630 | 2631 | 2632 | 2633 | 2634 | 2635 | 2636 | 2637 | 2638 | 2639 |
| A50 | 2640 | 2641 | 2642 | 2643 | 2644 | 2645 | 2646 | 2647 | 2648 | 2649 | 2650 | 2651 | 2652 | 2653 | 2654 | 2655 |
| A60 | 2656 | 2657 | 2658 | 2659 | 2660 | 2661 | 2662 | 2663 | 2664 | 2665 | 2666 | 2667 | 2668 | 2669 | 2670 | 2671 |
| A70 | 2672 | 2673 | 2674 | 2675 | 2676 | 2677 | 2678 | 2679 | 2680 | 2681 | 2682 | 2683 | 2684 | 2685 | 2686 | 2687 |
| A80 | 2688 | 2689 | 2690 | 2691 | 2692 | 2693 | 2694 | 2695 | 2696 | 2697 | 2698 | 2699 | 2700 | 2701 | 2702 | 2703 |
| A90 | 2704 | 2705 | 2706 | 2707 | 2708 | 2709 | 2710 | 2711 | 2712 | 2713 | 2714 | 2715 | 2716 | 2717 | 2718 | 2719 |
| AA0 | 2720 | 2721 | 2722 | 2723 | 2724 | 2725 | 2726 | 2727 | 2728 | 2729 | 2730 | 2731 | 2732 | 2733 | 2734 | 2735 |
| AB0 | 2736 | 2737 | 2738 | 2739 | 2740 | 2741 | 2742 | 2743 | 2744 | 2745 | 2746 | 2747 | 2748 | 2749 | 2750 | 2751 |
| AC0 | 2752 | 2753 | 2754 | 2755 | 2756 | 2757 | 2758 | 2759 | 2760 | 4761 | 2762 | 2763 | 2764 | 2765 | 2766 | 2767 |
| AD0 | 2768 | 2769 | 2770 | 2771 | 2772 | 2773 | 2774 | 2775 | 2776 | 2777 | 2778 | 2779 | 2780 | 2781 | 2782 | 2783 |
| AE0 | 2784 | 2785 | 2786 | 2787 | 2788 | 2789 | 2790 | 2791 | 2792 | 2793 | 2794 | 2795 | 2796 | 2797 | 2798 | 2799 |
| AF0 | 2800 | 2801 | 2802 | 2803 | 2804 | 2805 | 2806 | 2807 | 2808 | 2809 | 2810 | 2811 | 2812 | 2813 | 2814 | 2815 |
| B00 | 2816 | 2817 | 2818 | 2819 | 2820 | 2821 | 2822 | 2823 | 2824 | 2825 | 2826 | 2827 | 2828 | 2829 | 2830 | 2831 |
| B10 | 2832 | 2833 | 2834 | 2835 | 2836 | 2837 | 2838 | 2839 | 2840 | 2841 | 2842 | 2843 | 2844 | 2845 | 2846 | 2847 |
| B20 | 2848 | 2849 | 2850 | 2851 | 2852 | 2853 | 2854 | 2855 | 2856 | 2857 | 2858 | 2859 | 2860 | 2861 | 2862 | 2863 |
| B30 | 2864 | 2865 | 2866 | 2867 | 2868 | 2869 | 2870 | 2871 | 2872 | 2873 | 2874 | 2875 | 2876 | 2877 | 2878 | 2879 |
| B40 | 2880 | 2881 | 2882 | 2883 | 2884 | 2885 | 2886 | 2887 | 2888 | 2889 | 2890 | 2891 | 2892 | 2893 | 2894 | 2895 |
| B50 | 2896 | 2897 | 2898 | 2899 | 2900 | 2901 | 2902 | 2903 | 2904 | 2905 | 2906 | 2907 | 2908 | 2909 | 2910 | 2911 |
| B60 | 2912 | 2913 | 2914 | 2915 | 2916 | 2917 | 2918 | 2919 | 2920 | 2921 | 2922 | 2923 | 2924 | 2925 | 2926 | 2927 |
| B70 | 2928 | 2929 | 2930 | 2931 | 2932 | 2933 | 2934 | 2935 | 2936 | 2937 | 2938 | 2939 | 2940 | 2941 | 2942 | 2943 |
| B80 | 2944 | 2945 | 2946 | 2947 | 2948 | 2949 | 2950 | 2951 | 2952 | 2953 | 2954 | 2955 | 2956 | 2957 | 2958 | 2959 |
| B90 | 2960 | 2961 | 2962 | 2963 | 2964 | 2965 | 2966 | 2967 | 2968 | 2969 | 2970 | 2971 | 2972 | 2973 | 2974 | 2975 |
| BA0 | 2976 | 2977 | 2978 | 2979 | 2980 | 2981 | 2982 | 2983 | 2984 | 2985 | 2986 | 2987 | 2988 | 2989 | 2990 | 2991 |
| B80 | 2992 | 2993 | 2994 | 2995 | 2996 | 2997 | 2998 | 2999 | 3000 | 3001 | 3002 | 3003 | 3004 | 3005 | 3006 | 3007 |
| BC0 | 3008 | 3009 | 3010 | 3011 | 3012 | 3013 | 3014 | 3015 | 3016 | 3017 | 3018 | 3019 | 3020 | 3021 | 3022 | 3023 |
| BD0 | 3024 | 3025 | 3026 | 3027 | 3028 | 3029 | 3030 | 3031 | 3032 | 3033 | 3034 | 3035 | 3036 | 3037 | 3038 | 3039 |
| BE0 | 3040 | 3041 | 3042 | 3043 | 3044 | 3045 | 3046 | 3047 | 3048 | 3049 | 3050 | 3051 | 3052 | 3053 | 3054 | 3055 |
| BF0 | 3056 | 3057 | 3058 | 3059 | 3060 | 3061 | 3062 | 3063 | 3064 | 3065 | 3066 | 3067 | 3068 | 3069 | 3070 | 3071 |
| C00 | 3072 | 3073 | 3074 | 3075 | 3076 | 3077 | 3078 | 3079 | 3080 | 3081 | 3082 | 3083 | 3084 | 3085 | 3086 | 3087 |
| C10 | 3088 | 3089 | 3090 | 3091 | 3092 | 3093 | 3094 | 3095 | 3096 | 3097 | 3098 | 3099 | 3100 | 3101 | 3102 | 3103 |
| C20 | 3104 | 3105 | 3106 | 3107 | 3108 | 3109 | 3110 | 3111 | 3112 | 3113 | 3114 | 3115 | 3116 | 3117 | 3118 | 3119 |
| C30 | 3120 | 3121 | 3122 | 3123 | 3124 | 3125 | 3126 | 3127 | 3128 | 3129 | 3130 | 3131 | 3132 | 3133 | 3134 | 3135 |
| C40 | 3136 | 3137 | 3138 | 3139 | 3140 | 3141 | 3142 | 3143 | 3144 | 3145 | 3146 | 3147 | 3148 | 3149 | 3150 | 3151 |
| C50 | 3152 | 3153 | 3154 | 3155 | 3156 | 3157 | 3158 | 3159 | 3160 | 3161 | 3162 | 3163 | 3164 | 3165 | 3166 | 3167 |
| C60 | 3168 | 3169 | 3170 | 3171 | 3172 | 3173 | 3174 | 3175 | 3176 | 3177 | 3178 | 3179 | 3180 | 3181 | 3182 | 3183 |
| C70 | 3184 | 3185 | 3186 | 3187 | 3188 | 3189 | 3190 | 3191 | 3192 | 3193 | 3194 | 3195 | 3196 | 3197 | 3198 | 3199 |
| C80 | 3200 | 3201 | 3202 | 3203 | 3204 | 3205 | 3206 | 3207 | 3208 | 3209 | 3210 | 3211 | 3212 | 3213 | 3214 | 3215 |
| C90 | 3216 | 3217 | 3218 | 3219 | 3220 | 3221 | 3222 | 3223 | 3224 | 3225 | 3226 | 3227 | 3228 | 3229 | 3230 | 3231 |
| CA0 | 3232 | 3233 | 3234 | 3235 | 3236 | 3237 | 3238 | 3239 | 3240 | 3241 | 3242 | 3243 | 3244 | 3245 | 3246 | 3247 |
| CB0 | 3248 | 3249 | 3250 | 3251 | 3252 | 3253 | 3254 | 3255 | 3256 | 3257 | 3258 | 3259 | 3260 | 3261 | 3262 | 3263 |
| CC0 | 3264 | 3265 | 3266 | 3267 | 3268 | 3269 | 3270 | 3271 | 3272 | 3273 | 3274 | 3275 | 3276 | 3277 | 3278 | 3279 |
| CD0 | 3280 | 3281 | 3282 | 3283 | 3284 | 3285 | 3286 | 3287 | 3288 | 3289 | 3290 | 3291 | 3292 | 3293 | 3294 | 3295 |
| CE0 | 3296 | 3297 | 3298 | 3299 | 3300 | 3301 | 3302 | 3303 | 3304 | 3305 | 3306 | 3307 | 3308 | 3309 | 3310 | 3311 |
| CF0 | 3312 | 3313 | 3314 | 3315 | 3316 | 3317 | 3318 | 3319 | 3320 | 3321 | 3322 | 3323 | 3324 | 3325 | 3326 | 3327 |

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| D00 | 3328 | 3329 | 3330 | 3331 | 3332 | 3333 | 3334 | 3335 | 3336 | 3337 | 3338 | 3339 | 3340 | 3341 | 3342 | 3343 |
| D10 | 3344 | 3345 | 3346 | 3347 | 3348 | 3349 | 3350 | 3351 | 3352 | 3353 | 3354 | 3355 | 3356 | 3357 | 3358 | 3359 |
| D20 | 3360 | 3361 | 3362 | 3363 | 3364 | 3365 | 3366 | 3367 | 3368 | 3369 | 3370 | 3371 | 3372 | 3373 | 3374 | 3375 |
| D30 | 3376 | 3377 | 3378 | 3379 | 3380 | 3381 | 3382 | 3383 | 3384 | 3385 | 3386 | 3387 | 3388 | 3389 | 3390 | 3391 |
| D40 | 3392 | 3393 | 3394 | 3395 | 3396 | 3397 | 3398 | 3399 | 3400 | 3401 | 3402 | 3403 | 3404 | 3405 | 3406 | 3407 |
| D50 | 3408 | 3409 | 3410 | 3411 | 3412 | 3413 | 3414 | 3415 | 3416 | 3417 | 3418 | 3419 | 3420 | 3421 | 3422 | 3423 |
| D60 | 3424 | 3425 | 3426 | 3427 | 3428 | 3429 | 3430 | 3431 | 3432 | 3433 | 3434 | 3435 | 3436 | 3437 | 3438 | 3439 |
| D70 | 3440 | 3441 | 3442 | 3443 | 3444 | 3445 | 3446 | 3447 | 3448 | 3449 | 3450 | 3451 | 3452 | 3453 | 3454 | 3455 |
| D80 | 3456 | 3457 | 3458 | 3459 | 3460 | 3461 | 3462 | 3463 | 3464 | 3465 | 3466 | 3467 | 3468 | 3469 | 3470 | 3471 |
| D90 | 3472 | 3473 | 3474 | 3475 | 3476 | 3477 | 3478 | 3479 | 3480 | 3481 | 3482 | 3483 | 3484 | 3485 | 3486 | 3487 |
| DA0 | 3488 | 3489 | 3490 | 3491 | 3492 | 3493 | 3494 | 3495 | 3496 | 3497 | 3498 | 3499 | 3500 | 3501 | 3502 | 3503 |
| DB0 | 3504 | 3505 | 3506 | 3507 | 3508 | 3509 | 3510 | 3511 | 3512 | 3513 | 3514 | 3515 | 3516 | 3517 | 3518 | 3519 |
| DC0 | 3520 | 3521 | 3522 | 3523 | 3524 | 3525 | 3526 | 3527 | 3528 | 3529 | 3530 | 3531 | 3532 | 3533 | 3534 | 3535 |
| CC0 | 3536 | 3537 | 3538 | 3539 | 3540 | 3541 | 3542 | 3543 | 3544 | 3545 | 3546 | 3547 | 3548 | 3549 | 3550 | 3551 |
| DE0 | 3552 | 3553 | 3554 | 3555 | 3556 | 3557 | 3558 | 3559 | 3560 | 3561 | 3562 | 3563 | 3564 | 3565 | 3566 | 3567 |
| DF0 | 3568 | 3569 | 3570 | 3571 | 3572 | 3573 | 3574 | 3575 | 3576 | 3577 | 3578 | 3579 | 3580 | 3581 | 3582 | 3583 |
| E00 | 3584 | 3585 | 3586 | 3587 | 3588 | 3589 | 3590 | 3591 | 3592 | 3593 | 3594 | 3595 | 3596 | 3597 | 3598 | 3599 |
| E10 | 3600 | 3601 | 3602 | 3603 | 3604 | 3605 | 3606 | 3607 | 3608 | 3609 | 3610 | 3611 | 3612 | 3613 | 3614 | 3615 |
| E20 | 3616 | 3617 | 3618 | 3619 | 3620 | 3621 | 3622 | 3623 | 3624 | 3625 | 3626 | 3627 | 3628 | 3629 | 3630 | 3631 |
| E30 | 3632 | 3633 | 3634 | 3635 | 3636 | 3637 | 3638 | 3639 | 3640 | 3641 | 3642 | 3643 | 3644 | 3645 | 3646 | 3647 |
| E40 | 3648 | 3649 | 3650 | 3651 | 3652 | 3653 | 3654 | 3655 | 3656 | 3657 | 3658 | 3659 | 3660 | 3661 | 3662 | 3663 |
| E50 | 3664 | 3665 | 3666 | 3667 | 3668 | 3669 | 3670 | 3671 | 3672 | 3673 | 3674 | 3675 | 3676 | 3677 | 3678 | 3679 |
| E60 | 3680 | 3681 | 3682 | 3683 | 3684 | 3685 | 3686 | 3687 | 3688 | 3689 | 3690 | 3691 | 3692 | 3693 | 3694 | 3695 |
| E70 | 3696 | 3697 | 3698 | 3699 | 3700 | 3701 | 3702 | 3703 | 3704 | 3705 | 3706 | 3707 | 3708 | 3709 | 3710 | 3711 |
| E80 | 3712 | 3713 | 3714 | 3715 | 3716 | 3717 | 3718 | 3719 | 3720 | 3721 | 3722 | 3723 | 3724 | 3725 | 3726 | 3727 |
| E90 | 3728 | 3729 | 3730 | 3731 | 3732 | 3733 | 3734 | 3735 | 3736 | 3737 | 3738 | 3739 | 3740 | 3741 | 3742 | 3743 |
| EA0 | 3744 | 3745 | 3746 | 3747 | 3748 | 3749 | 3750 | 3751 | 3752 | 3753 | 3754 | 3755 | 3756 | 3757 | 3758 | 3759 |
| EB0 | 3760 | 3761 | 3762 | 3763 | 3764 | 3765 | 3766 | 3767 | 3768 | 3769 | 3770 | 3771 | 3772 | 3773 | 3774 | 3775 |
| EC0 | 3776 | 3777 | 3778 | 3779 | 3780 | 3781 | 3782 | 3783 | 3784 | 3785 | 3786 | 3787 | 3788 | 3789 | 3790 | 3791 |
| ED0 | 3792 | 3793 | 3794 | 3795 | 3796 | 3797 | 3798 | 3799 | 3800 | 3801 | 3802 | 3803 | 3804 | 3805 | 3806 | 3807 |
| EE0 | 3808 | 3809 | 3810 | 3811 | 3812 | 3813 | 3814 | 3815 | 3816 | 3817 | 3818 | 3819 | 3820 | 3821 | 3822 | 3823 |
| EF0 | 3824 | 3825 | 3826 | 3827 | 3828 | 3829 | 3830 | 3831 | 3832 | 3833 | 3834 | 3835 | 3836 | 3837 | 3838 | 3839 |
| F00 | 3840 | 3841 | 3842 | 3843 | 3844 | 3845 | 3846 | 3847 | 3848 | 3849 | 3850 | 3851 | 3852 | 3853 | 3854 | 3855 |
| F10 | 3856 | 3857 | 3858 | 3859 | 3860 | 3861 | 3862 | 3863 | 3864 | 3865 | 3866 | 3867 | 3868 | 3869 | 3870 | 3871 |
| F20 | 3872 | 3873 | 3874 | 3875 | 3876 | 3877 | 3878 | 3879 | 3880 | 3881 | 3882 | 3883 | 3884 | 3885 | 3886 | 3887 |
| F30 | 3888 | 3889 | 3890 | 3891 | 3892 | 3893 | 3894 | 3895 | 3896 | 3897 | 3898 | 3899 | 3900 | 3901 | 3902 | 3903 |
| F40 | 3904 | 3905 | 3906 | 3907 | 3908 | 3909 | 3910 | 3911 | 3912 | 3913 | 3914 | 3915 | 3916 | 3917 | 3918 | 3919 |
| F50 | 3920 | 3921 | 3922 | 3923 | 3924 | 3925 | 3926 | 3927 | 3928 | 3929 | 3930 | 3931 | 3932 | 3933 | 3934 | 3935 |
| F60 | 3936 | 3937 | 3938 | 3939 | 3940 | 3941 | 3942 | 3943 | 3944 | 3945 | 3946 | 3947 | 3948 | 3949 | 3950 | 3951 |
| F70 | 3952 | 3953 | 3954 | 3955 | 3956 | 3957 | 3958 | 3959 | 3960 | 3961 | 3962 | 3963 | 3964 | 3965 | 3966 | 3967 |
| F80 | 3968 | 3969 | 3970 | 3971 | 3972 | 3973 | 3974 | 3975 | 3976 | 3977 | 3978 | 3979 | 3980 | 3981 | 3982 | 3983 |
| F90 | 3984 | 3985 | 3986 | 3987 | 3988 | 3989 | 3990 | 3991 | 3992 | 3993 | 3994 | 3995 | 3996 | 3997 | 3998 | 3999 |
| FA0 | 4000 | 4001 | 4002 | 4003 | 4004 | 4005 | 4006 | 4007 | 4008 | 4009 | 4010 | 4011 | 4012 | 4013 | 4014 | 4015 |
| FB0 | 4016 | 4017 | 4018 | 4019 | 4020 | 4021 | 4022 | 4023 | 4024 | 4025 | 4026 | 4027 | 4028 | 4029 | 4030 | 4031 |
| FC0 | 4032 | 4033 | 4034 | 4035 | 4036 | 4037 | 4038 | 4039 | 4040 | 4041 | 4042 | 4043 | 4044 | 4045 | 4046 | 4047 |
| FD0 | 4048 | 4049 | 4050 | 4051 | 4052 | 4053 | 4054 | 4055 | 4056 | 4057 | 4058 | 4059 | 4060 | 4061 | 4062 | 4063 |
| FE0 | 4064 | 4065 | 4066 | 4067 | 4068 | 4069 | 4070 | 4071 | 4072 | 4073 | 4074 | 4075 | 4076 | 4077 | 4078 | 4079 |
| FF0 | 4080 | 4081 | 4082 | 4083 | 4084 | 4085 | 4086 | 4087 | 4088 | 4089 | 4090 | 4091 | 4092 | 4093 | 4094 | 4095 |

TIMER COUNTS

| CONTENTS OF COUNTER | COUNTS TO INTERRUPT | CONTENTS OF COUNTER | COUNTS TO INTERRUPT | CONTENTS OF COUNTER | COUNTS TO INTERRUPT | CONTENTS OF COUNTER | COUNTS TO INTERRUPT |
|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
| FE | 254 | 4D | 189 | D2 | 124 | 9F | 59 |
| FD | 253 | 9A | 188 | A5 | 123 | 3D | 58 |
| FB | 252 | 34 | 187 | 4B | 122 | 7C | 57 |
| F7 | 251 | 69 | 186 | 96 | 121 | F8 | 56 |
| EE | 250 | D3 | 185 | 2D | 120 | F1 | 55 |
| DC | 249 | A7 | 184 | 5B | 119 | E2 | 54 |
| B8 | 248 | 4F | 183 | B7 | 118 | C5 | 53 |
| 71 | 247 | 9E | 182 | 6E | 117 | 8A | 52 |
| E3 | 246 | 3C | 181 | DD | 116 | 15 | 51 |
| C7 | 245 | 78 | 180 | BA | 115 | 2A | 50 |
| 8E | 244 | FO | 179 | 75 | 114 | 55 | 49 |
| 1D | 243 | EO | 178 | EB | 113 | AA | 48 |
| 3B | 242 | C1 | 177 | D6 | 112 | 54 | 47 |
| 76 | 241 | 82 | 176 | AD | 111 | A8 | 46 |
| ED | 240 | 04 | 175 | 5A | 110 | 50 | 45 |
| DA | 239 | 09 | 174 | B5 | 109 | AO | 44 |
| B4 | 238 | 12 | 173 | 6A | 108 | 41 | 43 |
| 68 | 237 | 24 | 172 | D5 | 107 | 83 | 42 |
| D1 | 236 | 48 | 171 | AB | 106 | 06 | 41 |
| A3 | 235 | 90 | 170 | 56 | 105 | OD | 40 |
| 47 | 234 | 21 | 169 | AC | 104 | 1A | 39 |
| 8F | 233 | 42 | 168 | 58 | 103 | 35 | 38 |
| 1F | 232 | 85 | 167 | B1 | 102 | 6B | 37 |
| 3F | 231 | 0A | 166 | 62 | 101 | D7 | 36 |
| 7E | 230 | 14 | 165 | C4 | 100 | AF | 35 |
| FC | 229 | 28 | 164 | 88 | 99 | 5E | 34 |
| F9 | 228 | 51 | 163 | 11 | 98 | BD | 33 |
| F3 | 227 | A2 | 162 | 22 | 97 | 7B | 32 |
| E6 | 226 | 45 | 161 | 44 | 96 | F6 | 31 |
| CD | 225 | 8B | 160 | 89 | 95 | EC | 30 |
| 9B | 224 | 17 | 159 | 13 | 94 | D8 | 29 |
| 36 | 223 | 2E | 158 | 26 | 93 | B0 | 28 |
| 6D | 222 | 5D | 157 | 4C | 92 | 60 | 27 |
| DB | 221 | BB | 156 | 98 | 91 | C0 | 26 |
| B6 | 220 | 77 | 155 | 30 | 90 | 80 | 25 |
| 6C | 219 | EF | 154 | 61 | 89 | 00 | 24 |
| D9 | 218 | DE | 153 | C2 | 88 | C1 | 23 |
| B2 | 217 | BC | 152 | 84 | 87 | 03 | 22 |
| 64 | 216 | 79 | 151 | 03 | 86 | 07 | 21 |
| C8 | 215 | F2 | 150 | 10 | 85 | 0F | 20 |
| 91 | 214 | E4 | 149 | 20 | 84 | 1E | 19 |
| 23 | 213 | C9 | 148 | 40 | 83 | 3D | 18 |
| 46 | 212 | 93 | 147 | 81 | 82 | 7A | 17 |
| 8D | 211 | 27 | 146 | 02 | 81 | F4 | 16 |
| 1B | 210 | 4E | 145 | 05 | 80 | E8 | 15 |
| 37 | 209 | 9C | 144 | 0B | 79 | D0 | 14 |
| 6F | 208 | 38 | 143 | 16 | 78 | A1 | 13 |
| DF | 207 | 70 | 142 | 2C | 77 | 43 | 12 |
| BE | 206 | E1 | 141 | 59 | 76 | 87 | 11 |
| 7D | 205 | C3 | 140 | B3 | 75 | 0E | 10 |
| FA | 204 | 86 | 139 | 66 | 74 | 1C | 9 |
| F5 | 203 | 0C | 138 | CC | 73 | 39 | 8 |
| EA | 202 | 18 | 137 | 99 | 72 | 72 | 7 |
| D4 | 201 | 31 | 136 | 32 | 71 | E5 | 6 |
| A9 | 200 | 63 | 135 | 65 | 70 | CB | 5 |
| 52 | 199 | C6 | 134 | CA | 69 | 97 | 4 |
| A4 | 198 | 8C | 133 | 95 | 68 | 2F | 3 |
| 49 | 197 | 19 | 132 | 2B | 67 | 5F | 2 |
| 92 | 196 | 33 | 313 | 57 | 66 | BF | 1 |
| 25 | 195 | 67 | 130 | AE | 65 | 7F | 0 |
| 4A | 194 | CE | 129 | 5C | 64 | FE | 254 |
| 94 | 193 | 9D | 128 | B9 | 63 | | |
| 29 | 192 | 3A | 127 | 73 | 62 | | |
| 53 | 191 | 74 | 126 | E7 | 61 | | |
| A6 | 190 | E9 | 125 | CF | 60 | | |

APPENDIX D – INSTRUCTION SUMMARY

ACCUMULATOR GROUP INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ |
|---------|-------------|-------------|--------------------------------|-------------|------|-------|------|--------|----------------------|-----------------------------------|-------------------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| SR | 1 | 12 | SHIFT RIGHT ONE | 0 | 1/0 | 0 | 1 | 1 | 1 | — | 1 |
| SR | 4 | 14 | SHIFT RIGHT ONE | 0 | 1/0 | 0 | 1 | 1 | 1 | — | 1 |
| SL | 1 | 13 | SHIFT LEFT ONE | 0 | 1/0 | 0 | 1/0 | 1 | 1 | — | 1 |
| SL | 4 | 15 | SHIFT LEFT FOUR | 0 | 1/0 | 0 | 1/0 | 1 | 1 | — | 1 |
| COM | — | 18 | ACC (ACC) ⊕ H'FF' | 0 | 1/0 | 0 | 1/0 | 1 | 1 | — | 1 |
| LNK | — | 19 | ACC (ACC) + CB | 1/0 | 1/0 | 1/0 | 1/0 | 1 | 1 | — | 1 |
| INC | — | 1F | ACC (ACC) + 1 | 1/0 | 1/0 | 1/0 | 1/0 | 1 | 1 | — | 1 |
| LIS | i | 7i | ACC H'i' | | | — | | 1 | 1 | — | 1 |
| CLR | — | 70 | ACC H'00' | | | — | | 1 | 1 | — | 1 |
| LI | ii | 20 | ACC H'ii' | | | — | | 2.5 | 2 | — | 2 |
| NI | ii | 21 | ACC (ACC) H'ii' | 0 | 1/0 | 0 | 1/0 | 2.5 | 2 | — | 2 |
| OI | ii | 22 | ACC (ACC) V H'ii' | 0 | 1/0 | 0 | 1/0 | 2.5 | 2 | — | 2 |
| XI | ii | 23 | ACC (ACC) ⊕ H'ii' | 0 | 1/0 | 0 | 1/0 | 2.5 | 2 | — | 2 |
| AI | ii | 24 | ACC (ACC) + H'ii' (Binary Add) | 1/0 | 1/0 | 1/0 | 1/0 | 2.5 | 2 | — | 2 |
| CI | ii | 25 | H'ii' + (ACC)+1 | 1/0 | 1/0 | 1/0 | 1/0 | 2.5 | 2 | — | 2 |

- 1) An interrupt request cannot be acknowledged until an instruction without interrupt privilege has completed execution.
 2) This number of bytes can be transferred via DMA during the instruction's execution.

SCRATCHPAD REGISTER INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ |
|---------|-------------|-------------|---|-------------|------|-------|------|--------|----------------------|-----------------------------------|-------------------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| LR | y,x | | GENERAL LOAD REGISTER FORMAT ALLOWABLE OPER-ANDS LISTED BELOW | | | | | 1 | — | — | — |
| | A,r* | 4r | ACC (r) | | | — | | | 1 | — | 1 |
| | A,KU | 00 | ACC (r12) | | | — | | | 1 | — | 1 |
| | A,KL | 01 | ACC (r13) | | | — | | | 1 | — | 1 |
| | A,QU | 02 | ACC (r14) | | | — | | | 1 | — | 1 |
| | A,QL | 03 | ACC (r15) | | | — | | | 1 | — | 1 |
| | r,A | 5r | r (ACC) | | | — | | | 1 | — | 1 |
| | KU,A | 04 | r12 (ACC) | | | — | | | 1 | — | 1 |
| | KL,A | 05 | r13 (ACC) | | | — | | | 1 | — | 1 |
| | QU,A | 06 | r14 (ACC) | | | — | | | 1 | — | 1 |
| | QL,A | 07 | r15 (ACC) | | | — | | | 1 | — | 1 |
| AS | r | Cr | ACC (ACC)+(r)(Binary) | 1/0 | 1/0 | 1/0 | 1/0 | 1 | 1 | — | 1 |
| ASD | r | Dr | ACC (ACC)+(r) (Decimal) | 1/0 | 1/0 | 1/0 | 1/0 | 2 | 1 | — | 2 |
| NS | r | Fr | ACC (ACC) (r) | 0 | 1/0 | 0 | 1/0 | 1 | 1 | — | 1 |
| XS | r | Er | ACC (ACC) ⊕ (r) | 0 | 1/0 | 0 | 1/0 | 1 | 1 | — | 1 |
| DS | r | 3r | r (r)+H'FF'(Decrement) | 1/0 | 1/0 | 1/0 | 1/0 | 1.5 | 1 | — | 1 |

* Operand r formats are: Direct Addressing
 0 through 11 (Decimal Form)
 H'O' through H'B' (hexadecimal form)

Indirect Addressing
 S or 12
 I or 13
 D or 14

DATA COUNTER INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ |
|---------|-------------|-------------|---|-------------|------|-------|------|--------|----------------------|-----------------------------------|-------------------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| LR | Q,DC | 0E | r14 (DCU); r15 (DCL) | — | — | — | — | 4 | 1 | — | 3 |
| LR | H,DC | 11 | r10 (DCU); r11 (DCL) | — | — | — | — | 4 | 1 | — | 3 |
| LR | DC,Q | 0F | DCU (r14); DCL (r15) | — | — | — | — | 4 | 1 | — | 3 |
| LR | DC,H | 10 | DCU (r10); DCL (r11) | — | — | — | — | 4 | 1 | — | 3 |
| ADC | — | 8E | DC (DC) + (ACC) | — | — | — | — | 2.5 | 1 | — | 2 |
| DCI | iiii | 2A | DC H'iiii' | — | — | — | — | 6 | 3 | — | 5 |
| XDC | — | 2C | DC \rightleftharpoons DC ₁ [Memory Interface Circuit Only] | — | — | — | — | 2 | 1 | — | 2 |

INDIRECT SCRATCHPAD ADDRESS REGISTER INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ |
|---------|-------------|-------------|------------|-------------|------|-------|------|--------|----------------------|-----------------------------------|-------------------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| LR | A,IS | 0A | ACC (ISAR) | — | — | — | — | 1 | 1 | — | 1 |
| LR | IS,A | 0B | ISAR (ACC) | — | — | — | — | 1 | 1 | — | 1 |
| LISU | a | 01100a* | ISARU a | — | — | — | — | 1 | 1 | — | 1 |
| LISL | a | 01101a* | ISARL a | — | — | — | — | 1 | 1 | — | 1 |

* a is 3 bits

MEMORY REFERENCE INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ |
|---------|-------------|-------------|-------------------------------|-------------|------|-------|------|--------|----------------------|-----------------------------------|-------------------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| LM | — | 16 | ACC ((DC)) | — | — | — | — | 2.5 | 1 | — | 2 |
| ST | — | 17 | (DC) (ACC) | — | — | — | — | 2.5 | 1 | — | 1 |
| AM | — | 88 | ACC (ACC)+((DC)) (Binary) | 1/0 | 1/0 | 1/0 | 1/0 | 2.5 | 1 | — | 2 |
| AMD | — | 89 | ACC (ACC)+((DC)) (Decimal) | 1/0 | 1/0 | 1/0 | 1/0 | 2.5 | 1 | — | 2 |
| NM | — | 8A | ACC (ACC) ((DC)) | 0 | 1/0 | 0 | 1/0 | 2.5 | 1 | — | 2 |
| OM | — | 8B | ACC (ACC) ((DC)) | 0 | 1/0 | 0 | 1/0 | 2.5 | 1 | — | 2 |
| XM | — | 8C | ACC (ACC) \oplus ((DC)) | 0 | 1/0 | 0 | 1/0 | 2.5 | 1 | — | 2 |
| CM | — | 8D | ((DC)) + (ACC) + 1 | 1/0 | 1/0 | 1/0 | 1/0 | 2.5 | 1 | — | 2 |

STATUS REGISTER INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ | | | | | |
|---------|-------------|-------------|---|-------------|------|-------|-------|--------|----------------------|-----------------------------------|-------------------------|---|---|---|------|---|
| | | | | OVF | ZERO | CARRY | SIGN | | | | | | | | | |
| LR | W,J | 1D | W (r9) W ₄ W ₃ W ₂ W ₁ W ₀ <table border="1" style="display: inline-table; border-collapse: collapse; text-align: center;"> <tr> <td>INT</td> <td>OVF</td> <td>ZERO</td> <td>CARRY</td> <td>SIGN</td> </tr> </table> | INT | OVF | ZERO | CARRY | SIGN | — | — | — | — | 2 | 1 | Yes* | 2 |
| INT | OVF | ZERO | CARRY | SIGN | | | | | | | | | | | | |
| LR | J,W | 1E | r9 (W) (Privileged Instruction)* | — | — | — | — | 1 | 1 | — | 1 | | | | | |

* As a result of a privileged instruction execution, a request for interrupt service is not acknowledged by the CPU until a subsequent non-privileged instruction is executed.

MISCELLANEOUS INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ |
|---------|-------------|-------------|--------------|-------------|------|-------|------|--------|----------------------|-----------------------------------|-------------------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| NOP | — | 2B | NO OPERATION | — | — | — | — | 1 | 1 | — | 1 |

PROGRAM COUNTER INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ |
|---------|-------------|-------------|---|-------------|------|-------|--------|----------------------|-----------------------------------|-------------------------|
| | | | | OVF | ZERO | CARRY | | | | |
| LR | K,P | 08 | r12 (PC ₁ U); r13 (PC ₁ L) | — | — | — | 4 | 1 | — | 3 |
| LR | P,K | 09 | PC ₁ U (r12); PC ₁ L (r13) | — | — | — | 4 | 1 | — | 3 |
| LR | PO,Q | 0D | PC ₀ U (r14); PC ₀ L (r15) | — | — | — | 4 | 1 | — | 3 |
| PK | — | 0C | PC ₀ U (r12); PC ₀ L (r13) and PC ₁ (PC ₀) Privileged Instruction* | — | — | — | 4 | 1 | Yes* | 3 |
| PI | aaaa** | 28 | PC ₁ (PC ₀); PC ₀ H'aaaa' | — | — | — | 6.5 | 3 | Yes* | 5 |
| POP | — | 1C | PC ₀ (PC ₁) Privileged Instruction* | — | — | — | 2 | 1 | Yes* | 2 |

BRANCH INSTRUCTIONS

| OP CODE | OPER-AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE ¹⁾ | DMA SLOTS ²⁾ | | | | | | | | |
|----------------|--------------------|----------------|---|----------------|----------------|----------------|-------------------------|----------------------|-----------------------------------|-------------------------|------|--|--|--|--|--|--|--|
| | | | | OVF | ZERO | CARRY | | | | | SIGN | | | | | | | |
| BR | aa | 90 | PC ₀ ((PC ₀)+1) + H'aa' | — | — | — | 3.5 | 2 | — | 3 | | | | | | | | |
| JMP | aaaa 2 | 29 | PC ₀ H'aaaa' | — | — | — | 5.5 | 3 | Yes 1 | 4 | | | | | | | | |
| BT | t,aa 4 | 8t 4 | PC ₀ ((PC ₀)+H'aa' if any test is true PC ₀ (PC ₀)+ 2 if no test is true | — | — | — | 3.5 ³ 3.0 | 2 | — | 3 | | | | | | | | |
| | | | STATUS BIT TESTS | | | | | | | | | | | | | | | |
| | | | <table border="1"> <tr> <td>2²</td> <td>2¹</td> <td>2⁰</td> </tr> <tr> <td>ZERO</td> <td>CARRY</td> <td>SIGN</td> </tr> </table> | 2 ² | 2 ¹ | 2 ⁰ | ZERO | CARRY | SIGN | | | | | | | | | |
| 2 ² | 2 ¹ | 2 ⁰ | | | | | | | | | | | | | | | | |
| ZERO | CARRY | SIGN | | | | | | | | | | | | | | | | |
| BP | aa | 81 | PC ₀ ((PC ₀)+1)+ H'aa' if SIGN=1 PC ₀ (PC ₀)+ 2 if SIGN=0 | — | — | — | 3.5 3.0 | 2 | — | 3 | | | | | | | | |
| BC | aa | 82 | PC ₀ ((PC ₀)+1)+ H'aa' if CARRY=1 PC ₀ (PC ₀)+ 2 if CARRY=0 | — | — | — | 3.5 3.0 | 2 | — | 3 | | | | | | | | |
| BZ | aa | 84 | PC ₀ ((PC ₀)+1) H'aa' if ZERO=1 PC ₀ (PC ₀)+ 2 if ZERO=0 | — | — | — | 3.5 3.0 | 2 | — | 3 | | | | | | | | |
| BM | aa | 91 | PC ₀ ((PC ₀)+1)+ H'aa' if SIGN=0 PC ₀ (PC ₀)+ 2 if SIGN=1 | — | — | — | 3.5 3.0 | 2 | — | 3 | | | | | | | | |
| BNC | aa | 92 | PC ₀ ((PC ₀)+1)+ H'aa' if CARRY=0 PC ₀ (PC ₀)+ 2 if CARRY=1 | — | — | — | 3.5 3.0 | 2 | — | 3 | | | | | | | | |
| BNZ | aa | 94 | PC ₀ ((PC ₀)+1)+ H'aa' if ZERO=0 PC ₀ (PC ₀)+ 2 if ZERO=1 | — | — | — | 3.5 3.0 | 2 | — | 3 | | | | | | | | |
| BF | t ⁵ ,aa | 9t 5 | PC ₀ ((PC ₀)+1)+ H'aa' if selected status bits are all "0" PC ₀ (PC ₀)+2 if any status bit is 1 | — | — | — | 3.5 ² 3.0 | 2 | — | 3 | | | | | | | | |
| | | | TEST CONDITIONS | | | | | | | | | | | | | | | |
| | | | <table border="1"> <tr> <td>2³</td> <td>2²</td> <td>2¹</td> <td>2⁰</td> </tr> <tr> <td>OVF</td> <td>ZERO</td> <td>CARRY</td> <td>SIGN</td> </tr> </table> | 2 ³ | 2 ² | 2 ¹ | 2 ⁰ | OVF | ZERO | CARRY | SIGN | | | | | | | |
| 2 ³ | 2 ² | 2 ¹ | 2 ⁰ | | | | | | | | | | | | | | | |
| OVF | ZERO | CARRY | SIGN | | | | | | | | | | | | | | | |
| BNO | aa | 98 | PC ₀ ((PC ₀)+1)+ H'aa' if OVF=0 PC ₀ (PC ₀)+ 2 if OVF=1 | — | — | — | 3.5 ² 3.0 | 2 | — | 3 | | | | | | | | |
| BRZ | aa | 8F | PC ₀ ((PC ₀)+1)+ H'aa' if ISAR≠7 PC ₀ (PC ₀)+ 2 if ISAR=7 | — | — | — | 2.5 ² 2.0 | 2 | — | 2 | | | | | | | | |

- As a result of a privileged instruction execution, a request for interrupt service is not acknowledged by the CPU until a subsequent non-privileged instruction is executed.
- The contents of the accumulator are destroyed.
- 3.5 cycles if branch is taken. 3.0 cycles if branch is not taken.
- t is only 3 bits
- t is four bits
- 2.5 cycles if branch is taken. 2.0 cycles if branch is not taken.

INTERRUPT CONTROL INSTRUCTIONS

| OP CODE | OPER- AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE 1) | DMA SLOTS 2) |
|------------|-----------------|----------------|---|-------------|------|-------|------|--------|----------------------------|---------------------------|-----------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| DI | — | 1A | DISABLE INTERRUPT | — | — | — | — | 2 | 1 | — | 2 |
| EI | — | 1B | ENABLE INTERRUPT Privileged Instruction* | — | — | — | — | — | 1 | YES* | 2 |

INPUT/OUTPUT INSTRUCTIONS

| OP CODE | OPER- AND(S) | OBJECT CODE | FUNCTION | STATUS BITS | | | | CYCLES | BYTES OF OBJECT CODE | INTERRUPT PRIVILEGE 1) | DMA SLOTS 2) |
|------------|-----------------|----------------|---|-------------|------|-------|------|--------|----------------------------|---------------------------|-----------------|
| | | | | OVF | ZERO | CARRY | SIGN | | | | |
| INS | a | Aa | ACC (INPUT PORT a) Input Ports 00 to 0F only | 0 | 1/0 | 0 | 1/0 | 4** | 1 | — | 3 |
| IN | aa | 26 aa | ACC (INPUT PORT aa) Input Ports 04 through FF only | 0 | 1/0 | 0 | 1/0 | 4 | 2 | — | 3 |
| OUTS | a | Ba | OUTPUT PORT a (ACC) Output Ports 00 to 0F only | — | — | — | — | 4** | 1 | YES** | 3 |
| OUT | aa | 27 aa | OUTPUT PORT aa (ACC) Output Ports 04 through FF only | — | — | — | — | 4 | 2 | YES* | 3 |

- * As a result of a privileged instruction execution, a request for interrupt service is not acknowledged by the CPU until a subsequent non-privileged instruction is executed.
- ** 2 cycles when I/O port address is "0" or "1".