EDITOR/ASSEMBLER

USER'S GUIDE

A SOLID STATE SOFTWARE™ PROGRAM

FOR THE

TEXAS INSTRUMENTS

COMPACT COMPUTER 40

This book was written by:
Bill Brewer

Programs were written by:
Bruce Donnam
Jim Hammerquist
Karl Heichelheim

With contributions by:
Bud Gerwig
Steve Smith
Bob Whitsitt

IMPORTANT NOTICE REGARDING PROGRAMS AND BOOK MATERIALS

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE, REGARDING THESE PROGRAMS OR BOOK MATERIALS OR ANY PROGRAMS DERIVED THEREFROM AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THESE BOOK MATERIALS OR PROGRAMS AND THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE CARTRIDGE. MOREOVER, TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER AGAINST THE USER OF THESE PROGRAMS OR BOOK MATERIALS BY ANY OTHER PARTY.

Copyright c 1983, Texas Instruments Incorporated All rights reserved.

underlining in text (except heads) indicates a typestyle other than normal text
= pound sign; % = percent sign; @ = at sign; ? = question mark

Table of Contents

CC-40 EDITOR/ASSEMBLER USER'S GUIDE

CHAPTER 1--CC-40 ASSEMBLY-LANGUAGE PROGRAM DEVELOPMENT WITH THE EDITOR/ASSEMBLER: AN INTRODUCTION

AN INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING
    Execution of Commands
    Two Programming Techniques
ASSEMBLY LANGUAGE AND THE ALDS
THE "BEEP" DEMONSTRATION PROGRAM
    Preparing the CC-40 for "BEEP" Program Development
    Editing the "BEEP" Program
    Assembling the "BEEP" Program
    Linking the "BEEP" Program
    Loading and running the "BEEP" Program
    Using the DESUG Monitor
ADVANCED PROGRAM DEVELOPMENT ON THE CC-40

CHAPTER 2--TMS7000 ASSEMBLY LANGUAGE

TMS7000-FAMILY PROCESSING OVERVIEW
    Processing Operations
    Data Storage
        General-Purpose Register File
        Peripheral Register File
    Other Memory Addresses
    Processor Registers

underlining in text (except heads) indicates a typestyle other than normal text
2 = pound sign; Δ = percent sign; σ = at sign; u = question mark

TMS7000 ASSEMBLY-LANGUAGE INSTRUCTIONS
Instruction Types
Arithmetic Instructions
Branch and Jump Instructions
Compare and Test Instructions
Control Instructions
Load and Move Instructions
Logical Instructions
Rotate Instructions
Addressing Modes
Implied Addressing
Register Addressing
Immediate Values in Register-Addressing Instructions
Single-Register Addressing
General-Purpose Register Addressing
Peripheral Register Addressing
Program-Counter Relative Addressing
Memory Addressing
Direct Addressing
Indirect Addressing
Indexed Addressing
Trap-Instruction Addressing
INSTRUCTION SYNTAX
Instruction-Line Entries: Makeup and Sequence
Labels
Op Codes and Operands
Comments
EXPRESSION-FORMATION RULES
Constants
Numbers
Strings
Variables and Labels
Expressions
THE OP CODE MAP

underlining in text (except heads) indicates a typestyle other than normal text
3 = pound sign; % = percent sign; @ = at sign; ? = question mark

CHAPTER 3—CC-40 OPERATION WITH THE ALDS

CC-40 ASSEMBLY-LANGUAGE DEVELOPMENT CONFIGURATION
Minimal ALDS Hardware Configuration
Enhanced ALDS Hardware Configuration
CONSOLE DEVICES
Console Keyboard
Power-Control and Execution-Control Keys
Keyboard Uppercase-Shift Operations
Keyboard Control-Character Entry
Display-Editing Keys
Cursor-Positioning Keys
Insert-Mode Key
Character-Delete, Erase-Field, and Display-Clear Keys
Liquid-Crystal Display (LCD)
The Character-Display Line
Indicators
HEX-BUS™ DEVICES

CHAPTER 4—THE LINE AND SCREEN EDITOR

SYSTEM REQUIREMENTS
RUNNING THE EDITOR
Line Editing: "L" Option
Screen Editing: "S" Option
Screen Definition: "D" Option
Uncrunched Format Text: "U" Option
Crunched Format Text: "C" Option
Quitting the Editor: "Q" Option

underlining in text (except heads) indicates a typestyle other than normal text
4 = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

LINE-EDITOR OPERATION
Text Lines and Line Numbers
Line Editing Commands
New Text Entry
Editing Previously Entered Lines
Using the Display Buffer
Clearing the Display Line
Using the Playback and Recover Buffers
Line-Number Display Commands
Cursor Control
Deletion of Text in the Display
Insertion of Characters into Displayed Text
Insertion of Single Lines of Text
Insertion of Multiple Lines of Text
Text Line Replication
SCREEN-EDITOR OPERATION
Screen-Editing Commands
Text Entry
Cursor Positioning
Display Paging Control
Screen Refresh (View)
Line-Number On/Off Toggle
Text Deletion
Text Insertion
Screen-Editing Keyboard and Display Definition
Keyboard Selection
Selection of Device and Data-Transmission Characteristics
Command Keystroke Definition
Function-Key Definition
Default Values for Command and Display-Function Keys
Definition of Required Display-Control Information
Definition of Optional Display-Control Information

underlining in text (except heads) indicates a typestyle other than normal text
5 = pound sign; Δ = percent sign; σ = at sign; u = question mark

Default Values for Display-Control Characteristics
Using or Saving the Definition Table
LINE AND SCREEN-EDITOR COMMAND DESCRIPTIONS
AUTO: Automatic Line Number Command
COPY: Copy (Load) a File into Text Area of Memory
DELETE: Delete Line or Block of Lines
EDIT: Display a Line for Editing
FIND
FORMAT: Prepare New Mass-Storage Medium
HELP: Show Commands and Display Functions
JUMP: Move the Cursor (or Screen) to Another Point in the Text
LIST: Display or Print Lines of Text
MOVE: Move a Line or Block of Text
QUIT: Return to the ALDS Menu
REPLACE: Find One String and Replace It with Another
SAVE: Write the Text to a Mass-Storage File
TAB DEFINE: Set Display Tab Stops Other Than as Defined
UNDO: Cancel the Last Modification, Deletion, or Replacement
VERIFY: Compare the Text with a Mass-Storage File
SUMMARY COMMAND DESCRIPTIONS

CHAPTER 5--THE ASSEMBLER

SOURCE AND OBJECT FILES
Source Files
Object Files
RUNNING THE ASSEMBLER
SOURCE FILE STATEMENTS
Assembly-Language Instructions
Pseudo Operations
Numeric-Data Pseudo Operations
ASCII (Character-String) Data-Storage Pseudo-Ops

underlining in text (except heads) indicates a typestyle other than normal text
6 = pound sign; Δ = percent sign; σ = at sign; μ = question mark

Assembler Directives
Code and Data Location Instructions
Symbol Value Equating Directive
End of Source Statement
Linking Information Directives
Source File Copy (Include) Directive
Assembly-Listing Control Directives
Assembler Option Directive
Pseudo Operations in Detail
Numeric Data-Storage Instructions: BYTE and DATA
ASCII Data-Storage Instructions: TEXT AND RTEXT
Assembler Directives in Detail
Code- and Data-Location Instructions: AORG, RORG, BSS, BES
Value-Assignment Instruction: EQU
Linking Directives: DEF and REF
End-of-Assembly Instruction: END
Multiple Source File Directive: COPY
Listing Directives: IDT, TITL, PAGE, LIST, and UNL
Listing Format Directives
Directives to List or Not List Lines
Option Directive: OPTION
ASSEMBLY ERRORS AND ERROR MESSAGES

CHAPTER 6--THE LINKER

RUNNING THE LINKER
LINKING COMMANDS
Input-File Commands
Absolute-Output Command
Output-File Format Command
The END Command
LINKER COMMAND TEXT AND LISTING
LINK MAP FORMAT
LINKER INPUT AND OUTPUT FORMATS
HEADERS, MEMORY-IMAGES, AND RELOCATION-TABLE RECORDS

underlining in text (except heads) indicates a typestyle other than normal text
7 = pound sign; A = percent sign; @ = at sign; u = question mark

HEADER RECORDS
File-Type Word
Flag Byte
Maximum Relocation Table Length
Block Size
Length of Memory Image
Absolute Load Address or Minimum Relocation Table Length
ERROR MESSAGES

CHAPTER 7--ALDS UTILITY PROGRAMS

HEXADECIMAL DUMP UTILITY (HEXDUMP)
Running tne HEXDUMP Program
Use of HEXDUMP
HEXDUMP Error
OBJECT-FILE LOADERS
Saving and Loading ABSLOAD and CARTLOAD
Absolute-Code Loader
Cartridge Loader
The CARTINIT Subprogram
The CARTLOAD Subprogram
CARTRIDGE-SAVE PROGRAM (MEMSAVE)
FILE CONVERSION AND TRANSFER UTILITIES
Multiple- to Single-Record File Conversion Utility (CONVERT)
File Transfer Utility (TRANSFER)
Text Transfer from One Device to Another
Text Transfer from an External Computer
TI-990 Tagged-Object File Transfer
Transfer Utility Errors
BASIC to ALDS Text Utilities (SAVEA AND COPYA)
ALDS FILE-TRANSFER AND CONVERSION SUMMARY

underlining in text (except heads) indicates a typestyle other than normal text
8 = pound sign; Δ = percent sign; σ = at sign; µ = question mark

CHAPTER 8—DEBUG MONITOR

RUNNING THE DEBUG MONITOR

MONITOR COMMANDS

Display-Memory Command: D

Memory-Modify Command: M

Processor-Register Modify Command: P

Copy-Memory Command: C

Execute Command: E

Breakpoint-Modify Command: B

ROM-Page Modify Command: R

Single-Step Command: S

The Quit Command: Q

The Help Command: µ

DEBUG MONITOR OPERATION CONCEPTS

DEBUG MONITOR USE WITH AN ALDS-DEVELOPED PROGRAM

Entering the Machine-Language Program

Testing the "BEEP" Program

Modifying "BEEP" Program Machine Code

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

CHAPTER 1

CC-40 ASSEMBLY-LANGUAGE PROGRAM DEVELOPMENT WITH THE EDITOR/ASSEMBLER: AN INTRODUCTION

This chapter introduces assembly-language programming and assembly-language program development with the CC-40 Editor/Assembler, an Assembly Language Development System (ALDS). The chapter also includes a detailed description of the development of a small program through use of the ALDS.

The Editor/Assembler cartridge for the Compact Computer 40 (CC-40) contains two major programs, an editor and an assembler, which perform major tasks in development of programs written in assembly language. In addition, the cartridge contains other programs which support assembly-language program development. All of these programs are integrated into a system designed to aid in the development of software which uses the maximum capabilities of the CC-40. The programs, taken together, compose a software system which is customarily called an Assembly Language Development System (ALDS).

The most popular method of programming the CC-40 involves writing programs in the language called BASIC (Beginner's All-purpose Symbolic Instruction Code). The BASIC language is the easily learned programming code supported by the firmware (programs in read-only memory) in every CC-40 console. It permits rapid development of programs, and it is "friendly" in pointing out programming errors.

Despite its advantages and popularity, BASIC cannot be used as the programming language for the development of some important applications programs. Programs which require fast execution and maximum efficiency in the use of memory space cannot be written in BASIC. For these reasons, word-processing, spreadsheet, data-communication, and many other valuable programs are usually written in a language other than BASIC, usually in "assembly language." Assembly-language programs run up to 200 times faster than BASIC programs which perform the same tasks, and they require as little as one-tenth the memory-storage space.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

BASIC	ASSEMBLY LANGUAGE
quick program writing	faster program execution
easy to learn to use	more efficient use of available memory
easy error checking	

Programs in BASIC and programs in assembly language can be combined in program development. Programs written in BASIC can use subprograms (subroutines) developed in assembly language whenever speed and memory economy are important. A video display, for example, can be driven by a subprogram developed in assembly language and called by a BASIC program. Additionally, many of the already-developed machine-language subroutines in CC-40 BASIC firmware can be used to simplify assembly-language programming. The CC-40 liquid-crystal display, for example, is easily controlled through assembly-language "calls" to the display subroutines that are used by BASIC.

AN INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

Assembly language permits direct control of computer-processor operations. Through assembly-language programming, each step of processor operation is specified by a machine-language program stored in CC-40 memory. Because of such comprehensive program control, programs written in assembly-language can be executed faster and provide much more computing power per unit of memory than programs written in any other programming language. Assembly language provides the means for using the CC-40 at its maximum capability.

Assembly language is a "command and control" language specifically created for use with individual computer processors. The assembly language for the CC-40 is created specifically for use with the TMS7000-family microprocessor which controls all CC-40 operations.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

Operation of the CC-40 is controlled by a program stored electronically in memory. The program is read from memory one step at a time by the CC-40 processor. The processor interprets what it reads into electronic control signals which guide processing operations, as shown below. In this manner, the CC-40 proceeds through many program steps to execute a program.



Eight data lines carry the electrical signals from each program-memory storage location to the processor. Binary voltage states (High/Low or ON/OFF) of these eight lines correspond to "bits" (binary digits) of information which compose an eight-bit information unit called a "byte" (binary term). When a byte of program information stored in program memory is present on the data-bus lines of the processor, the processor interprets the byte to be one of the following.

- A command code which directly controls processor operation
- A data byte
- An address byte

The way the processor interprets the electrical signals from program memory is incorporated into the "machine-language" design of the processor. The relationship between machine language and assembly language is described in chapter 2 of this guide. Individual assembly-language instructions are described in detail in chapter 1 of the CC-40 Editor/Assembler Reference Manual.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¿ = question mark

Execution of Commands

Bit combinations in the command code cause the processor to perform a certain operation according to processor machine language. If the command code contains certain bit combinations, it causes the processor to add. Other bit combinations cause it to subtract, output a byte, jump to another program memory address, or perform other processing operations. Unless the processing of a command (such as a jump, subroutine call, idle, or similar instruction) determines otherwise, the processor always executes commands in the sequence in which they are stored in program memory.

In addition to guiding the processor by specifying the type of operations to be performed in a program step, command codes identify any bytes which must follow the command code to produce a complete machine-language command. If a command code so specifies, the processor considers a subsequent byte to be a value to be used in an addition operation. Similarly, if another command code specifies it, the next byte is read by the processor as an address in memory from which the processor is to read data for a subtraction operation.

Two Programming Techniques

Programming the CC-40 consists of storing appropriately encoded command codes, data values, and address values into program memory. The CC-40 permits the storage of such machine-language codes by two methods.

The first method is a primitive (but workable) one. Numeric values which correspond directly with the electrically stored command-bit combinations are entered into the CC-40 keyboard for memory storage by the CC-40 DEBUG Monitor program. Such programming is called "machine-language programming" (see chapter 8 of this guide).

underlining in text (except neads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

The second method is much more sophisticated and practical. It demands less time, effort, and patience of a programmer, because it uses the computing capabilities of the CC-40 to replace the laborious tasks of command-table lookup, unautomated computation, and copious note-taking. This method uses the programs in the Editor/Assembler cartridge. The editor, the assembler, the linker, and the various utility programs are themselves machine-language programs which make the computing power of the CC-40 available for program development.

ASSEMBLY LANGUAGE AND THE ALDS

In programs stored directly into CC-40 memory by the DEBUG Monitor, commands are represented by hexadecimal (base 16) numbers. Commands with hexadecimal numbers in the series >18, >28, . . . >78, for example, direct the processor to add numbers from various storage locations in CC-40 memory. Subsequent hexadecimal numbers in the commands identify the storage locations for the values to be added and the sums of those values.

Hexadecimal numbers which represent commands are difficult to remember during programming, and remembering memory addresses is tedious. Hexadecimal command codes are not like words: "add," "subtract," "compare," and so on. The command to "add the number in register 12 to the number in register 49," for example, is the following sequence of numbers: >48 >0C >31. With a substantial program, a programmer could spend a lifetime remembering--and misremembering and correcting--such numbers. Short of spending such a lifetime, a programmer can use the ALDS program called the assembler to perform these tedious tasks.

underlining in text (except heads) indicates a typestyle other than normal
text; single brackets indicates information to be typed; double brackets
indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

The ALDS assembler is a translation program which is conceptually the central program in the CC-40 assembly-language development system (ALDS). Problems in programming with numbers are solved by the assembler through its translating English words or mnemonic expressions into machine-language commands. The sequence of numbers for the command listed above, >48 >0C >31, results directly from assembler translation of the simple statement ADD R12,R49. (The assembler is described in detail in chapter 5 of this guide.)

Other programs in the ALDS are designed to support the work done by the assembler in its production of processor commands from mnemonic English words and phrases. Programs other than the assembler—the editor, the linker, the DEBUG Monitor, and the utility programs—provide for text entry of assembly-language programs to the assembler and process the machine-language output of the assembler for specific uses in the CC-40.

The ALDS editor provides a means for entering and revising assembly-language commands for subsequent translation by the assembler. Through the editor, entire programs can be written before they are assembled. An editor option provides for viewing either single program lines on the CC-40 console display or a multiline program segment on a CRT terminal connected to the CC-40 through an RS232 input/output device. (The editor is described in detail in chapter 4 of this guide.)

The ALDS linker further translates the numbers produced by the assembler into an executable machine-language program for the TMS7000-family processor (much as the DEBUG Monitor does for numbers entered at the CC-40 keyboard). The linker performs the necessary tasks of adding memory-addressing information to the assembled program and converting machine code into a memory image of what it must be in order to run on the CC-40. (The linker is described in detail in chapter 6 of this guide.)

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ʊ = question mark

The ALDS utility programs are a group of special-purpose ALDS programs which further support editing, assembling, and linking. The utility programs include the following.

- A file-dump utility: a program which prints the contents of any file stored on mass-storage media.

- Loader utilities: two programs which bring ALDS-developed programs into memory for execution.

- A save utility: a program which saves ALDS-developed programs in RAM to mass-storage media

- A file-conversion and transfer utility: a program which enables communication between the CC-40 and external computers used to develop assembly-language programs.

- BASIC editing utilities: two programs which permit the ALDS screen editor to be used for development of BASIC programs.

The DEBUG Monitor, in addition to its memory-access functions described above, provides for controlled execution of programs being developed. Loaded programs can be started, stopped, examined, and changed as they are executed in order to detect and eliminate programming errors. (The DEBUG Monitor is described in detail in chapter 8 of this guide.)

Chapter 2 of this guide describes machine language and assembly language in detail, and chapter 6 of the CC-40 Editor/Assembler Reference Manual describes each assembly-language instruction used with the ALDS. Chapters 4 through 8 of this guide describe the uses and operation of the editor, assembler, linker, utility programs, and the DEBUG Monitor. The remaining parts of this chapter provide instructions for using the ALDS to develop a small program without such detailed knowledge.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

THE "BEEP" DEMONSTRATION PROGRAM

Use of the ALDS to develop CC-40 programs is simple and straightforward. The ALDS programs are run in a sequence that processes an assembly-language program into a completed, working machine-language program. The following pages of this chapter show how each program is used in the development of a short program which emits a "beep" from the CC-40 beeper.

The instructions provided below for development of the beep-emitting program are complete, and when they are followed with exacting care they result in the completely developed program which demonstrates all of the major features of the ALDS. Many problems, however, can result from seemingly trivial mistakes in programming when any programming system is used, particularly with a system such as the ALDS (with its being more resourceful but more "unforgiving" than, for example, BASIC). As with even BASIC, thorough understanding can easily uncover mistakes which are not even noticed in the careful following of directions. If the following instructions do not produce the expected results at any stage, consult the appropriate chapter of this guide to find the solution to the problem.

Figure 1-1 shows a small program called "BEEP," which is written in assembly language. After the program is processed by the ALDS into machine-language and executed, it activates the beeper in the CC-40. Activation of the beeper requires that electrical pulses be sent to the CC-40 beeper which transforms electrical pulses into sound waves, and this short program provides 1024 such pulses to produce a single "beep." The "BEEP" program causes the processor to alternately turn on and turn off electrical current to the beeper a total of 1024 times in order to produce the pulses that emit a tone.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

```
00001 COUNTER   EQU   >5D                COUNTER MEMORY LOCATION
00002 BEEP      MOVD  Δ>400,COUNTER      SET CYCLE COUNT TO 1024
00003 BEEP2     MOVP  Δ1,P21             TURN ON THE BEEPER
00004           MOV   Δ29,B              DELAY COUNT FOR ON PERIOD
00005 ONLOP     DJNZ  B,ONLOP            LOOP BACK FOR COUNT
00006           MOVP  Δ0,P21             TURN BEEPER OFF
00007           MOV   Δ32,B              DELAY COUNT FOR OFF PERIOD
00008 OFFLOP    DJNZ  B,OFFLOP           LOOP BACK FOR COUNT
00009           DECD  COUNTER            COUNT OF CYCLES DONE
00010           JC    BEEP2              LOOP BACK UNTIL ALL DONE
00011           RETS                     RETURN TO CALLING PROGRAM
00012 NAMLOW
00013           RTEXT 'BEEP'             PROGRAM NAME (IN REVERSE CAPITALS)
00014           BYTE  NAMHGH-NAMLOW      NAME LENGTH
00015 NAMHGH    EQU   $-1                FILE STORAGE SPECIFICATIONS
00016           DATA  BEEP               PROGRAM ENTRY POINT
00017           DATA  0000               NEXT SUBPROGRAM
00018           DATA  NAMHGH-$+1         OFFSET TO NAME
00019           BYTE  0,>44              HEADER INFORMATION
```

Figure 1-1. The "BEEP" Assembly-Language Program

The function of each line of the program in terms of CC-40 operation is explained in the comment phrases printed to the right of the assembly-language instructions. More exact meanings of the instructions and each element within them are given in chapter 2 of this guide and in chapter 1 of the CC-40 Editor/Assembler Reference Manual.

Development of the "BEEP" program requires only the CC-40 console with the ALDS cartridge installed and a single Wafertape™ device. (A mass-storage device other than a Wafertape unit can be used by additional reference to the operating instructions appropriate to the device.)

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

Preparing the CC-40 for "BEEP" Program Development

The "BEEP" program is complete and ready for entry into the ALDS editor as the first step in transforming the assembly language into the machine language which controls CC-40 beeper operation. The following instructions describe how to prepare the CC-40 and the Wafertape™ device and media for program development.

STEP 1: Check the Wafertape device to ensure that the device-selection switch on the back of the device is set to position 1.

STEP 2: Ensure that the Wafertape device is connected to the CC-40 by a HEX-BUS™ connecting cable and that AC-adaptor cables are connected or batteries are installed in each unit.

STEP 3: Insert the ALDS cartridge into the cartridge port of the CC-40.

STEP 4: Turn on the Wafertape device, and then press the [[ON]] key at the upper right of the CC-40 keyboard.

STEP 5: Press the [[ENTER]] key when the BASIC cursor, the System initialized message, or the message Memory contents may be lost appears.

STEP 6: Insert a blank tape (of a length of 5 feet or greater) into the Wafertape™ device.

STEP 7: Type in [FORMAT 1] and press [[ENTER]] to initialize (prepare) the tape for writing and reading files.

STEP 8: If a message containing the phrase I/O error appears in the display, the Wafertape unit or medium is malfunctioning. Check the HEX-BUS™ cable, check for both units receiving power, and try another tape to correct the malfunction. See the Wafertape™ device Owner's Manual for additional details.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; ∧ = percent sign; ∂ = at sign; ¤ = question mark

STEP 9: Enter [RUN "ALDS"] (note the double quotes surrounding the word "ALDS"). The message c 1983 Texas Instruments momentarily appears; then the ALDS main menu E)dit A)ssemble L)ink B)asic¤ appears.

When these steps have been followed, the CC-40 is ready for editing the "BEEP" program.

Editing the "BEEP" Program

The editor is a program which accepts both characters and text-formatting commands from the keyboard. The CC-40 console functions much like an electronic typewriter which allows for correctable typing of program text into memory. From characters and commands typed into the keyboard, the editor composes the text of the assembly-language program in CC-40 memory, and it saves the file on Wafertape medium for later use by the assembler.

The following steps enter the program text into CC-40 memory from the keyboard and save the text onto Wafertape™ medium. (Chapter 4 of this guide contains a detailed description of editor operation.)

STEP 1: The display shows E)dit A)ssemble L)ink B)asic¤ Press the [E] key to start the editor program.

STEP 2: When the display shows L)ine S)creen D)efine Q)uit, press the [L] key to permit use of the CC-40 console keyboard and display for editing.

STEP 3: When the Copy file promot appears, press [[ENTER]]. (If a file were to be reloaded for editing, the filename would be typed in following this prompt.)

STEP 4: In response to the editor ">" prompt, enter [AUTO] to have the ALDS automatically provide line numbers for each program line.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

STEP 5: In response to each line number displayed, type in each line (followed by [[ENTER]]) exactly as it is shown in figure 1-1. Where several spaces are used to keep columns of the text in line, however, only one space is required. Proofread each line carefully and make any changes to the line before pressing [[ENTER]], using the [Left Arrow] key to backspace the cursor for correcting characters and [[SHIFT]] [Left Arrow] for deleting and closing-up characters. Use [Left Arrow] and [Right Arrow] to move the 31-character display "window" over longer lines during proofreading and correcting.

STEP 6: With all text entered, press [[BREAK]] to stop automatic line numbering after line 19 has been entered.

STEP 7: As a special precaution against mistakes, review the lines that have been typed by entering [LIST] when the ">" cursor appears. Read through and verify each line, pressing [[ENTER]] when a line is verified to be correct. If a line must be corrected, use the correction keys described in step 5 to correct it. Press [[ENTER]] when the line is correct, and check the next line. When all lines have been verified, press [[BREAK]].

STEP 8: When the cursor appears, enter [SAVE 1.SRC], a name that identifies the program on Wafertape medium. Wait for the program to be saved (indicated by the reappearance of the cursor) in the display. TURN LINES OFF BY ENTERING "NOLINE" LIST "2.SRC"

STEP 9: In response to the cursor, enter the word [QUIT]. After the message Are you sure (Y/N)u appears in the display, press [Y] to leave the line-edit mode.

STEP 10: In response to the edit menu, L)ine S)creen D)efine Q)uit, press [Q].

underlining in text (except neads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; µ = question mark

When step 10 has been completed, the assembly-language source program is stored on the tape. Now the assembly-language instructions must be converted into processor commands by assembling the source program.

Assembling the "BEEP" Program

The assembler converts the assembly-language program stored on mass-storage medium into a machine-language program which can subsequently be processed to run from CC-40 memory. (Chapter 5 of this guide contains a detailed description of assembler operation.)

The following steps assemble the "BEEP" program into machine language.

STEP 1: With the ALDS menu, E)dit A)ssemble L)ink B)asicu displayed, press [A] to start the assembler program.

STEP 2: With the display showing the prompt Source file, enter the name by which the source has been stored on tape: [1.SRC] (the unit number of the Wafertape device and the name of the file saved by the editor).

STEP 3: With the display showing the prompt Object file, enter the name by which the machine-language object code is to be stored on tape: [1.OBJ].

STEP 4: When the CC-40 display shows the question In memory assemble (Y/N)µ, press [Y] for YES.

STEP 5: With List file in the display, press [[ENTER]] (or enter the device number and other data for a printer to produce an assembly listing).

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

STEP 6: Wait while the program is displaying the message Assembling... until it displays Write object file (Y/N)u Answer [Y] to the question in order to direct the assembler to write the machine-language program to tape.

STEP 7: The display should contain the message NO errors NO warnings. Press [[ENTER]] to return to the main ALDS menu. If the display contains some other message, refer to chapter 5 of this guide for a description of assembly errors. When assembly errors are present, it is necessary to return to the instructions in the Editing the "Beep" Program section, to answer [1.SRC] to the Copy file prompt, and to follow the remaining editing steps to correct the errors in source-file statements. Then the error-free source file can be reassembled.

With the main ALDS menu in the display, assembly of the program is complete. The program can now be processed into executable object code by the linker.

Linking the "BEEP" Program

The linker converts the assembler-output code into commands which can be executed in random-access memory (RAM) from the BASIC interpreter. (The linker, in addition, performs many other tasks, as explained in chapter 6 of this guide.)

The following instructions link the "BEEP" program to run it from the BASIC command level of the CC-40.

STEP 1: After the assembler has written the object program onto Wafertape™ media and the display once again shows the ALDS menu, E)dit A)ssemble L)ink B)asicu, press [L] to start the linker.

STEP 2: When the linker asks Is there a control file (Y/N)u, answer [N] for NO.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

STEP 3: When the linker prompts with Enter line:, enter [I 1.OBJ] to tell the linker which file it is to include in the linking process from Wafertape media.

STEP 4: When the linker again prompts with Enter line:, enter [E] for END to tell the linker to begin execution of the instruction provided in step 3.

STEP 5: When the linker requests Linked output file, respond with [1.BEEP], the name of the executable machine language program which is to be run under BASIC.

STEP 6: When the linker requests List file, press [[ENTER]] to indicate that a listing is not wanted. The display then shows Linking...

STEP 7: When Write linked outout fileu appears in the display, press [[ENTER]].

STEP 8: When LINK COMPLETE is displayed, press [[ENTER]] to return to the ALDS main menu. If any other message appears, refer to chapter 6 for an explanation of linking errors and correct them accordingly.

STEP 9: The program is complete and ready to test. Press [B] from the main ALDS menu to return to BASIC for testing the program.

Loading and Running the "BEEP" Program

Althougn various utility programs are provided for such special purposes as loading Constant Memory™ program cartridges (see chapter 7), the "BEEP" program has been linked to be loaded and executed as a BASIC subprogram. The steps required for loading and executing the program as it has been linked above are listed below.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

STEP 1: When the BASIC cursor appears in the display, enter [CALL LOAD ("1.BEEP")] to load the assembled and linked program as a BASIC subprogram into memory.

STEP 2: When the BASIC cursor again appears, enter the BASIC command [CALL BEEP] to execute the BEEP subprogram.

STEP 3: Listen to the "beep" made by the program developed using the ALDS, and compare the results of running it with the results of the beeper command normally used by the BASIC statement DISPLAY BEEP.

Using the DEBUG Monitor

After the machine-language program is loaded into memory, the DEBUG Monitor program can be used to (1) monitor the execution of the "BEEP" program or (2) alter the program as it is being run. Chapter 8 of this guide provides instructions for using the DEBUG Monitor to perform both of these tasks.

ADVANCED PROGRAM DEVELOPMENT ON THE CC-40

The steps listed above develop the small "BEEP" program for execution as a BASIC subprogram and demonstrate the use of each ALDS program. Much larger, more sophisticated, and more powerful programs can be developed with the ALDS. The following chapters of this guide describe in detail the full complement of techniques and tools provided by the ALDS for program development.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign: Δ = percent sign; σ = at sign; μ = question mark

CHAPTER 2
TMS7000 ASSEMBLY LANGUAGE

This chapter contains a description of the TMS7000-family assembly language as it is implemented in the ALDS. Data processing and storage, classifications and addressing modes of assembly-language instructions, and instruction syntax are described.

TMS7000-FAMILY PROCESSING OVERVIEW

Machine-language, and therefore assembly-language, instructions for the CC-40 represent the capabilities of the TMS7000-family processor. The instructions are a direct reflection of the way the processor manipulates, stores, inputs, and outputs data.

Representation of data in this chapter is in conventional binary, hexadecimal, decimal, and binary-coded decimal (BCD) number systems, as reviewed in appendix A of the CC-40 Editor/Assembler Reference Manual. The special symbol ">" preceding a number indicates that it is a hexadecimal or BCD number, and the special symbol "μ" preceding a number indicates that it is a binary number. Numbers prefaced by no symbol are decimal numbers.

Processing Operations

The TMS7000-family processor performs all customary operations on eight-bit units (bytes) of data. As described later in this chapter, it performs arithmetic operations (including multiplication), comparisons, logical operations, bit "rotations" or shifts, jumps, calls, control operations, data moves, and similar operations.

Data Storage

Processing done by TMS7000-family processors is performed entirely on data stored within the 65,536 addresses it can directly access.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

General-Purpose Register File

There are 128 general-purpose registers within the processor; these registers are accessed at addresses >00 to >7F. Within this general-purpose register file, a stack can be implemented to store, in a last-in-first-out manner, data and addresses.

Peripheral Register File

An additional 256 addresses from >100 through >1FF provide for data input and output (peripheral) registers. Not all addresses within this peripheral-register block are used in the CC-40.

Other Memory Addresses

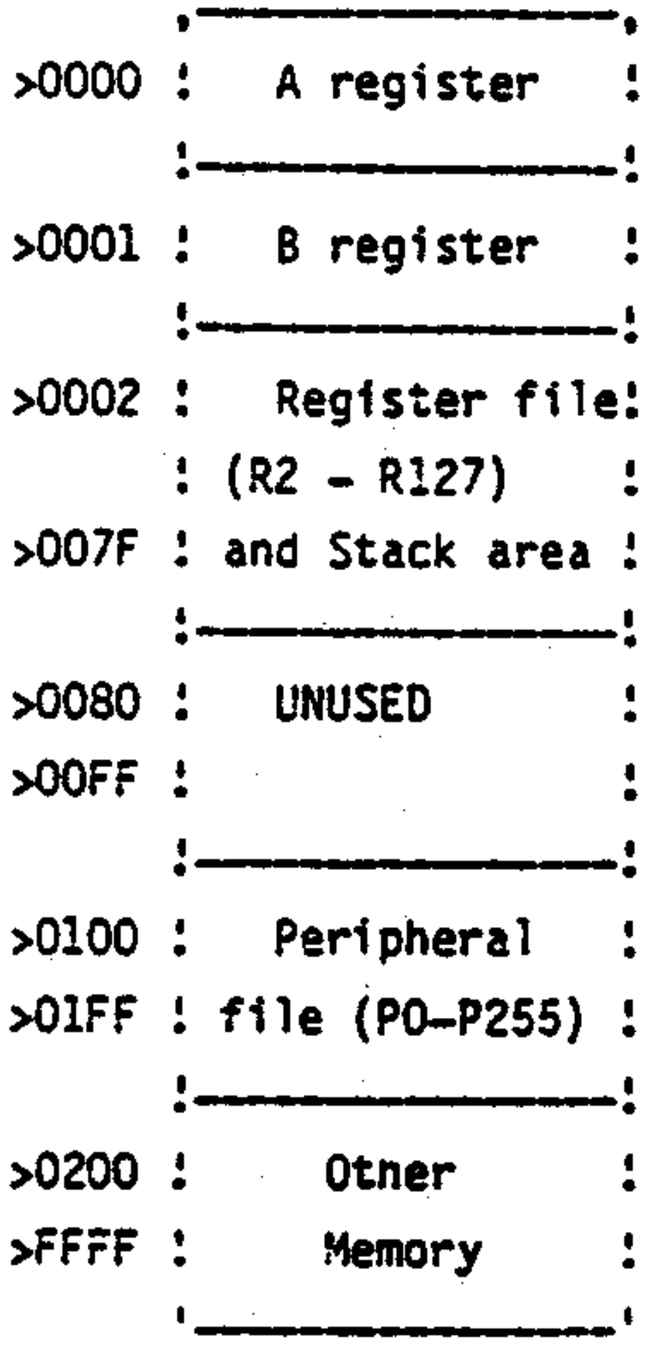
Addresses above >1FF are used in the CC-40 for system firmware, console random-access memory (RAM), cartridge RAM, and cartridge read-only memory (ROM). Fewer instructions are available to access memory in this block than are available to access general-purpose and peripheral registers.

In addition to being accessed by fewer instructions than registers are, memory is accessed by instructions which produce longer machine-language commands and which take more processor time for execution. General-purpose and peripheral registers have an advantage over memory-addressing commands: they are accessed by commands which use less program memory and consume less processing time.

Chapter 2 of the CC-40 Editor/Assemoler Reference Manual contains detailed information about register and memory usage and conventions in the CC-40, and chapter 5 of the manual contains detailed information about haroware implementation of registers and memory.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

A map showing TMS7000-family memory usage is shown below.



Processor Registers

Three additional registers are internal to the processor chip: a 16-bit program counter (PC), an 8-bit status register (ST), and an 8-bit stack pointer (SP). The program counter contains the address of the next machine-language command which the processor executes after it finishes the command currently being executed. This address is computed from the address and length of the current command (unless the current command is a branch or jump command). A branch or jump command causes the program counter to be loaded with a new value for out-of-sequence command execution.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

The status register contains four single-bit flags in the four high-order bits. One of these flags describes the status of processor interrupts (enabled or disabled). The remaining three flags describe whether the execution of a command has (1) resulted in a value of zero, (2) resulted in a negative value, or (3) resulted in a carry out of an eight-bit value. The table below shows the assignment of flag bits in the status register.

FLAG BITS IN THE STATUS REGISTER

7 (MS)	6	5	4	3	2	1	0 (LS)
Carry	Negative	Zero	Interrupt	Unused	Unused	Unused	Unused

The stack pointer can be set to addresses >00 through >7F in the general-purpose register file. The pointer is incremented before a byte of data is "pushed" onto the stack, and it is decremented after a byte is "popped" off of the stack. Similarly, the stack pointer is incremented once before each byte of a two-byte return address is pushed onto the stack by a subroutine call command (high byte first). The stack pointer is decremented once after each byte is popped off by a return-from-subroutine command. The stack pointer is usually set to >01 or greater to avoid conflict with registers A and B.

Chapter 5 of the CC-40 Editor/Assembler Reference Manual contains further details about processor-hardware operation.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

TMS7000 ASSEMBLY-LANGUAGE INSTRUCTIONS

TMS7000 assembly-language programs consist of line-by-line instructions provided to the processor after the ALDS translates them into machine-language commands. These mnemonic instructions, in general, are related to machine-language commands as described in chapter 1 of this guide.

The basis for classification of instructions is twofold: (1) instruction function in the operation performed by the processor, and (2) source and destination of data which is processed. The function of an instruction determines its type. The source-and-destination of data for an instruction determines its addressing mode.

Instruction Types

If an instruction adds bytes of data, it is of one type; if it stores bytes of data, it is of another type.

Sixty-five processing operations can be performed by a TMS7000-family processor. These 65 operations, however, are of only seven general types, as shown in the following table.

underlining in text (except heads) indicates a typestyle other than normal
text; single brackets indicates information to be typed; double brackets
indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; µ = question mark

INSTRUCTION TYPES

INSTRUCTION TYPE	PROCESSOR OPERATION
ARITHMETIC	Process values arithmetically
BRANCH AND JUMP	Control command-execution sequence
COMPARE AND TEST	Compare and test values in registers, memory
CONTROL	Set conditions for subsequent processing
LOAD AND MOVE	Transfer values between registers, memory, and input/output devices
LOGICAL	Perform logical operations on bits in registers
ROTATE	Rotate bits in registers

The most significant part of an assembly-language instruction is an operation code ("op code" for short). Op codes are mnemonic representations of operations which the processor can perform.

Brief descriptions of each instruction type and of each op code within the classification are presented below.

More detailed descriptions of each op code (including addressing modes, resulting machine-language commands, command execution times, and other relevant information) are given in chapter 6 of the CC-40 Editor/Assembler Reference Manual.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

Arithmetic Instructions

Instructions related to numeric computation (addition, subtraction, and multiplication) are classified as arithmetic commands. Eleven instructions, as shown in the table below, belong to this classification.

ARITHMETIC INSTRUCTIONS

FUNCTION	MNEMONIC OP CODE
ADD	ADD
ADD WITH CARRY	ADC
DECIMAL ADD WITH CARRY	DAC
DECIMAL SUBTRACT WITH BORROW	DSB
DECREMENT	DEC
DECREMENT DOUBLE	DECD
INCREMENT	INC
INVERT	INV
MULTIPLY	MPY
SUBTRACT	SUB
SUBTRACT WITH BORROW	SBB

The binary-addition and -subtractions instructions ADD and SUB add or subtract single-byte values. The operations they produce set or reset the carry flag so that subsequent operations can be based on overflow or underflow occurring during the operation.

The binary-addition and -subtraction instructions ADC and SBB produce the same operations as the ADD and SUB instructions except that they account for the conditions of the carry flag set (by a carry in a previously performed ADD or ADC instruction) or reset (by a borrow in previously performed by a SUB or SBB instruction).

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

The decimal-addition and -subtraction instructions DAC and DSB add and subtract register or immediate values according to binary-coded decimal (BCD) arithmetic. In BCD arithmetic, as explained in appendix A, each nibble (half-byte) of the sum or difference is considered to be a binary representation of a decimal number (ranging from u0000 to u1001 or >0 to >9).

The multiply instruction MPY multiplies two single-byte values to form a two-byte product in the A and B registers.

The increment and decrement instructions INC and DEC increment or decrement the value in a register by one.

The decrement-double instruction DECD decrements the value of a register pair by one. There is no corresponding increment double instruction for TMS7000-family processors.

The invert instruction INV inverts or complements each bit in a register. The result is the one's complement of the value before the operation: each ONE bit is changed into a ZERO bit, and each ZERO bit is changed into a ONE bit.

Branch and Jump Instructions

Instructions that can determine the sequence in which program commands are executed are members of the branch and jump classification, as shown in the following table.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ω = question mark

BRANCH AND JUMP INSTRUCTIONS

FUNCTION	MNEMONIC OP CODE
BRANCH	BR
BIT TEST AND JUMP IF ONE	BTJO
BIT TEST AND JUMP IF ONE--PERIPHERAL	BTJOP
BIT TEST AND JUMP IF ZERO	BTJZ
BIT TEST AND JUMP IF ZERO--PERIPHERAL	BTJZP
CALL	CALL
DECREMENT REGISTER AND JUMP IF NON-ZERO	DJNZ
JUMP	JMP
JUMP IF CARRY	JC
JUMP IF EQUAL	JEQ
JUMP IF HIGHER OR THE SAME	JHS
JUMP IF LOWER	JL
JUMP IF NEGATIVE	JN
JUMP IF NO CARRY	JNC
JUMP IF NOT EQUAL	JNE
JUMP IF NOT ZERO	JNZ
JUMP IF POSITIVE	JP
JUMP IF POSITIVE OR ZERO	JPZ
JUMP IF ZERO	JZ
RETURN FROM SUBROUTINE	RETS
RETURN FROM INTERRUPT	RETI
TRAP	TRAP

Instructions of the branch and jump type produce commands that change the value in the program counter so that the next command executed is out of the normal sequence of program execution. These instructions produce either simple operations, or they produce operations that combine a jump with a previous test for a condition.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

The BR, JMP (including conditional jumps, such as JC, JEQ, and JHS), TRAP, CALL, RETS, and RETI instructions alter the sequence of execution of program steps (either unconditionally, or conditionally based on the state of the zero, carry, or negative flags). The decrement and jump on non-zero (DJNZ) and the bit-test-and-jump (BTJO, BTJZ, BTJOP, and BTJZP) instructions first perform an arithmetic or logical operation and then alter the sequence of execution based on the result of the arithmetic or logical operation.

The unconditional instructions in the branch and jump classification are BR, JMP, CALL, RETS, RETI, and TRAP. These instructions always alter the value in the processor program counter for execution of the subsequent instruction.

The branch instruction BR loads the program counter witn a two-byte value which becomes the address of the next command executed.

The subroutine-call instruction CALL saves the current program-counter value on the stack and then loads the program counter with a two-byte value, providing a return address at the end of subroutine execution.

The TRAP instruction is a special one-byte "call" instruction to one of 24 addresses stored at the top of memory. The processor "branches" to the appropriate address for the completion of the subroutine call. In programming terms, the addresses are "vectors" which are accessed by an economical, single-byte command. In execution, the TRAP instruction saves the current program-counter value on the stack and then loads the program counter with one of the 24 two-byte values which are stored in a table in firmware from >FFDO through >FFFF. Functionally, wnen the table has been appropriately set up with subroutine addresses, the TRAP instruction is essentially a CALL instruction that which produces a single-byte machine-language command.

The return-from-subroutine instruction RETS loads the program counter witn a two-byte value from the top of the stack, effectively restoring the program counter with the value saved by the last CALL or TRAP instruction.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ш = question mark

The return-from-interrupt instruction RETI is the same as the RETS instruction, except that it restores both the program counter and the status register from the stack. The RETI instruction is intended to restore all three registers in the processor to the state that they were in prior to the processor's responding to a hardware interrupt signal (see chapter 5 of the CC-40 Editor/Assembler Reference Manual).

The unconditional-jump instruction JMP loads the program counter with a value displaced from the current program-counter value after the two bytes of the jump instruction are read; this displacement must be within a range of -128 through +127.

The conditional jump instructions--JC, JEQ, JHS, JL, JN, JNC, JNE, JNZ, JP, JPZ, and JZ--are all the same as the JMP instruction, with the exception that their alteration of the value in the program counter is conditioned on the state of status flags specified by each instruction. If the Zero flag, for example, is set prior to the execution of a JZ instruction, then the JZ instruction produces the same processor operation as the JMP instruction; if the Zero Flag is reset, the JZ instruction has no effect except to increment the program counter to the next instruction in sequence and to expend processor time.

The combination jump instructions all function similarly to the conditional jump instructions, except that they perform an operation before they test for a condition required for a jump.

The decrement-and-jump-on-non-zero instruction DJNZ first decrements the value in a register. If the result of the decrement is non-zero, the condition is met for the DJNZ instruction to function as a JMP instruction. If the result of the decrement is zero, the condition for the jump is not met and the next instruction in sequence is executed.

The bit-test-and-jump instructions perform logical tests on the bits in a byte in order to determine whether conditions for executing a jump are met.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

The bit-test-and-jump-on-one instruction BTJO performs an AND operation on bits of a general-purpose register and bits from another byte and then discards the resulting byte. The other byte specifies the bits of the register to be tested by the bits within it being set to ONE. If the result of the AND operation on any bit is ONE, conditions for executing the jump are met and the value in the program counter is altered. The bit-test-and-jump-on-one-peripheral instruction, BTJOP, performs an identical operation except that the register tested is from the peripheral register file instead of the general-purpose register file.

The bit-test-and-jump-on-zero instructions BTJZ and BTJZP perform operations which are the same as those performed by the BTJO and BTJOP instructions except that the bits in the byte being tested are inverted before the test. If any of the register bits being tested were--before being inverted--ZERO, conditions are met for the jump.

Compare and Test Instructions

Compare and test instructions produce operations similar to the first simple operation in a bit-test-and-jump instruction. They perform an operation and discard the resulting byte (they do not write the result into a register or memory). The operation results in the setting of status flags which can determine the execution of conditional jump instructions.

The operation performed by compare instructions is binary subtraction. One byte is subtracted from another with the the result affecting status flags.

The operation performed by test instructions is a logical OR.

The four compare and test instructions and the corresponding op codes are listed in the following table.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

COMPARE AND TEST INSTRUCTIONS

FUNCTION	MNEMONIC OP CODE
COMPARE	CMP
COMPARE A TO MEMORY	CMPA
TEST A REGISTER	TSTA
TEST B REGISTER	TSTB

The register-compare instruction CMP subtracts the value in a register or the value in an immediate operand from the value in another register.

The memory-compare instruction CMPA subtracts the value in register A from the value stored in a memory location.

The test instructions produce the same results as performing a corresponding logical OR of the A register (for TESTA) or the B register (for TESTB) with itself. The test instructions, however, translate into single-byte machine-language commands, while OR instructions which have the same results produce two-byte commands.

Control Instructions
Control instructions are miscellaneous instructions that set the state of either the processor or the registers for subsequent operations. The functions of the control instructions are listed in the table below.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

CONTROL INSTRUCTIONS

FUNCTION	MNEMONIC OP CODE
CLEAR	CLR
CLEAR CARRY FLAG	CLRC
DISABLE INTERRUPTS	DINT
ENABLE INTERRUPTS	EINT
IDLE UNTIL INTERRUPT	IDLE
NO OPERATION	NOP
SET CARRY FLAG	SETC

The clear instruction CLR writes binary ZEROs into a register. It also resets the carry and negative flags and sets the zero flag.

The clear-carry instruction CLRC and the set-carry instruction SETC respectively reset and set the carry flag.

The disable-interrupt instruction DINT and the enable-interrupt instruction EINT determine whether processing can be interrupted by a hardware-interrupt signal to the processor. After an EINT command is executed, the processor can respond to interrupts. After a DINT command is executed, the processor ignores interrupts. With interrupts enabled, the processor's responding to an interrupt causes further interrupts to be ignored until interrupts are again enabled by an EINT or a RETI command.

The processor-idle instruction IDLE halts processing. Only hardware reset or interrupt signals to the processor can cause processing to resume with a reset or an interrupt routine.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

The no-operation instruction NOP causes the assembler to insert a byte containing binary ZEROs (the NOP command code) into the machine-language program. The NOP command has no effect on processing except to expend processing time, but it does provide a location in program memory where a byte of a "patched-in" machine-language command can be inserted.

Load and Move Instructions

Load and move instructions are data-transfer instructions which copy the values from one register or memory location to another. The following table lists the functions of the load and move instructions.

LOAD AND MOVE INSTRUCTIONS

FUNCTION	MNEMONIC OP CODE
LOAD A REGISTER	LDA
LOAD STACK POINTER	LDSP
MOVE	MOV
MOVE DOUBLE	MOVD
MOVE TO OR FROM PERIPHERAL FILE REGISTER	MOVP
POP FROM STACK	POP
PUSH ONTO STACK	PUSH
STORE A REGISTER	STA
STORE STACK POINTER	STSP
SWAP NIBBLE	SWAP
EXCHANGE WITH B REGISTER	XCHB

The load-A-register instruction LDA copies the value in a memory location into the A register. Similarly, the store-A-register instruction STA copies the value in the A register into a memory location.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

The load-stack-pointer instruction LDSP copies the value in the B register into the single-byte stack pointer. The store-stack-pointer instruction STSP copies the value in the stack pointer into the B register.

The move instruction MOV copies a byte into a register.

The move-double instruction MOVD copies two bytes into a register pair.

The move-peripheral instruction MOVP copies a byte into a peripheral-file register, or it copies the value from a peripheral-file register into register A or B.

The push-onto-stack instruction PUSH increments the stack pointer and copies either the value in a general-purpose register or the status register into the register-file location pointed to by the stack pointer. The pop-from-stack instruction POP, conversely, copies the value pointed to by the stack pointer into either a register or the status byte and decrements the stack pointer. The PUSH and POP instructions are used together to effect last-in-first-out storage in the the register file.

The swap-nibble instruction SWAP exchanges the high and low nibbles (half-bytes) stored in a register. The instruction is useful for conversion between BCD, binary, and ASCII representations of values.

The exchange-with-B instruction XCHB exchanges the contents of the B register with the contents of another register.

Logical Instructions

Logical instructions, listed in the following table, provide operations on individual bits within a byte stored in a register.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

LOGICAL INSTRUCTIONS

FUNCTION	MNEMONIC OP CODE
AND	AND
AND PERIPHERAL FILE REGISTER	ANDP
EXCLUSIVE OR	XOR
EXCLUSIVE OR PERIPHERAL FILE REGISTER	XORP
OR	OR
OR PERIPHERAL FILE REGISTER	ORP

The AND and ANDP instructions produce a logical AND operation on the bits in a byte and the corresponding bits in a general-purpose or peripheral register. The resulting byte is stored in the general-purpose or peripheral register.

The OR and ORP instructions produce a logical OR operation on the bits in a byte and the corresponding bits in a general-purpose or peripheral register. The resulting byte is stored in the general-purpose or peripheral register.

The XOR and XORP instructions produce a logical EXCLUSIVE-OR operation on the bits in a byte and the corresponding bits in a general-purpose or peripheral register. The resulting byte is stored in the general-purpose or peripheral register.

Rotate Instructions
Rotate instructions transfer bits in a register to either higher- or lower-order bit locations. The following table lists rotate instructions.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

ROTATE INSTRUCTIONS

FUNCTION	MNEMONIC OP CODE
ROTATE LEFT	RL
ROTATE LEFT THROUGH CARRY	RLC
ROTATE RIGHT	RR
ROTATE RIGHT THROUGH CARRY	RRC

The rotate-left instruction RL and the rotate-right instruction, RR respectively transfer register bits into higher- and lower-order bit positions. Bits which are transferred "out of the byte" are "rotated" respectively into the lowest- and highest-order bit positions in the register (the bit position emptied by other bit transfers). The bit rotated out of the byte is also copied into the carry flag.

The rotate-left-through-carry instruction RLC and the rotate-right-through-carry instruction RRC produce essentially the same operation, except that they treat the carry flag as a single-bit high-order extension of the register. Bit rotations therefore occur through the carry flag.

Addressing Modes

The source and destination of data processed by each instruction determines its addressing mode.

Some commands contain no explicit source or destination addresses: they use the "implied" addressing mode. These commands, such as NOP, consist only of op codes.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; % = percent sign; @ = at sign; ? = question mark

Other commands have data source and destination addresses explicitly designated in operands, expressions which follow op codes in instructions. The instruction which decrements the A register, for example, includes a single operand as follows.

```
DEC A
```

TMS7000-family processors use the following five modes of addressing.

- IMPLIED ADDRESSING
- SINGLE REGISTER ADDRESSING
- DUAL OPERAND ADDRESSING
- PROGRAM-COUNTER RELATIVE ADDRESSING
- MEMORY-ADDRESSING
- TRAP-INSTRUCTION ADDRESSING

Implied Addressing

When an instruction uses implied addressing, it has no operand to specify a source or destination for data. Source and destination are uniquely assigned to the instruction by the op code alone.

The following twelve instructions use implied addressing.

CLRC	IDLE	RETI	STSP
DINT	NOP	RETS	TSTA
EINT	LDSP	SETC	TSTB

underlining in text (except heads) indicates a typestyle other than normal
text; single brackets indicates information to be typed; double brackets
indicates key to be pressed
alpna = pound sign; Δ = percent sign; σ = at sign; u = question mark

Single-Register Addressing

When an instruction uses single-register addressing, a single operand which specifies a register follows the op code. Fourteen commands, listed below, use single-register addressing.

CLR	DJNZ	POP	RLC	SWAP
DEC	INC	PUSH	RR	XCHB
DECD	INV	RL	RRC	

All of these instructions take A, B, or a numbered register from the file (R2 through R127) for an operand. The PUSH and POP instructions, in addition, take the status register, ST, for an operand, as shown in the following examples of single-register addressing instruction.

```
CLR R58
PUSH ST
DJNZ R58,LABELO
```

The second operand for the DJNZ instruction uses program-counter relative addressing, described below.

Dual Operand Addressing

Instructions using dual operand addressing can use immediate values, peripheral file registers, and general purpose registers for their operands. Every dual operand instruction has at least two operands—a source and a destination. The three types of operands can be mixed with some restrictions. Each of these operands and the restrictions for mixing them are discussed below.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

Immediate Values

When an instruction uses an immediate value, the source value is stored in the machine-language instruction rather than in a register or data memory. Immediate values can be the source operand whenever a general-purpose or a peripheral register is specified in the destination operand. The value specified in the source operand must be prefaced by a percent symbol, "Δ", as in the following.

```
MOV Δ128,A
```

Twenty-one instructions, listed below, use immediate values.

ADC	BTJO	CMP	MOVD	ORP	XORP
ADD	BTJOP	DAC	MOVP	SBB	
AND	BTJZ	DSB	MPY	SUB	
ANDP	BTJZP	MOV	OR	XOR	

Four specific combinations of source and destination operands use immediate values. The combinations are the following, with "N" representing any valid register number (R0 through R127) or peripheral register number (P0 through P127) or single-byte value (0 through 255) and a percent-symbol prefix indicating an immediate value.

ΔN,A	ΔN,B	ΔN,RN	ΔN,PN

An immediate value is not always a single-byte value. In the special case of the move-double instruction, an immediate value is a two-byte value, as in the following example.

```
MOVD Δ>3FFF,R0
```

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

In a variation of this same special case, the MOVD instruction can also combine an immediate value and a source-register value. The sum of the immediate value and the contents of the B register is moved into the destination register pair. In the following example, the byte in the B register is added to 1024, and the sum is stored in R31 and R32.

```
MOVD Δ1024(B),R32
```

General-Purpose Registers

When an instruction uses general-purpose register addressing, two operands with a comma separating them follow the op code. The source of data is specified in the first operand. The destination of data is specified by the second operand.

The source operand can be any general-purpose register (R0 through R127), or it can be an immediate value.

The destination operand is always a register.

Fifteen instructions, listed below, use register-file addressing.

ADC	BTJO	DAC	MOVD	SBB
ADD	BTJZ	DSB	MPY	SUB
AND	CMP	MOV	OR	XOR

Seven specific combinations of source and destination operands are permitted with all of these op codes. The combinations are the following, with "N" representing any valid register number (R0 through R127) or single-byte value (0 through 255) and a percent-symbol prefix indicating an immediate value.

RN,A	RN,B	B,A	RN,RN	ΔN,A	ΔN,B	ΔN,RN

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpna = pound sign; Δ = percent sign; σ = at sign; u = question mark

The MOV instruction has three special operand combinations. These combinations are tne following.

```
MOV A,B
MOV A,RN
MOV B,RN
```

For bit-test-and-jump instructions, an additional operand follows the first two. This operand uses program-counter relative addressing (see below) to specify a jump destination.

The move-double-register instruction, MOVD, is a special two-byte instruction which transfers two bytes from one pair of registers to another. The higher-addressed register in the source pair is the first operand, and the higher-addressed register in the destination pair is the second operand. Registers must be specified within the range R1 through R127. In addition to register-file addressing, MOVD uses a special combination of immediate value and register addressing for a source operand, as described above.

Peripneral File Registers

When an instruction uses peripheral-file addressing, in general, two operands follow the op code. For most of tnese instructions, the source operand for all instructions can be register A, register B, or an immediate value (see immediate value addressing, below). The destination operand can be a peripheral-file register from P0 through P255.

Six instructions, listed below, use peripheral-file addressing.

ANDP	BTJOP	BTJZP	MOVP	ORP	XORP

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

Three specific combinations of source and destination operands are permitted with all of these op codes. The combinations are the following, with "N" representing any valid peripheral register number (P0 through P127) or single-byte value (0 through 255) and a percent-symbol prefix indicating an immediate value.

```
A,PN          B,PN          ΔN,PN
```

Additionally, the MOVP instruction (but not the logical or bit-test-and-jump instructions) can have a peripheral register as the source operand and either register A or register B as the destination operand, as shown below.

```
MOVP PN,A     MOVP PN,B
```

In bit-test-and-jump peripheral-file instructions, an additional operand follows the first two. This operand uses program-counter relative addressing as described below to specify a jump destination.

Program-Counter Relative Addressing
All jump instructions use program-counter relative addressing. The operand used in this addressing mode has the value of a program address within the range of -128 through +127 bytes of the "current" program-counter value after the jump command is read by the processor. The assembler computes a single-byte displacement value by subtracting the current program-counter value from the value of the jump operand.

The instructions that use program-counter relative addressing are listed below.

BTJO	BTJZP	JEQ	JMP	JNE	JPZ
BTJOP	DJNZ	JHS	JN	JNZ	JZ
BTJZ	JC	JL	JNC	JP	

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpna = pound sign; % = percent sign; @ = at sign; ? = question mark

The program-counter relative operand is the only operand for the JMP and conditional-jump instructions; it is an additional operand for the DJNZ and bit-test-and-jump instructions, as the following examples show.

```
JMP LABEL1
DJNZ R5,LABEL2
BTJO %>AA,R43,LABEL3
```

Memory Addressing

Five instructions can address all memory (including the general-purpose and peripheral register files). These instructions are the following.

BR	CALL	CMPA	LDA	STA

In addressing memory, the instructions use direct, indirect, and indexed addressing modes.

Direct Addressing

When direct addressing is used by instructions, the source of the value is a double-byte value obtained directly from the machine-language instruction. Similar to immediate addressing, direct addressing uses this 16-bit value as the address for branching, calling, comparing, loading, or storing. Through direct addressing, the following instructions can address all memory.

Direct addresses must be expressed in operands that contain the "at" sign "@" as a prefix, as shown in the following subroutine-call instruction (also see the "Instruction Syntax" section below).

```
CALL @>4000
```

Indirect Addressing

When indirect addressing is used, the source of the value is a register pair specified in the operand as the higher-addressed register.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ч = question mark

Indirect addresses must be expressed in operands that contain an asterisk (*) as a prefix, as in the following example.

```
STA *R44
```

Indexed Addressing

When indexed addressing is used, the source of the value is a combination of a direct address and the value the B register contains at the time of command execution. The direct-address value and the value in the B register are added by the processor to obtain the address used in the instruction.

Indexed addresses are expressed in operands that contain the direct address (prefixed by "σ") followed without comma or space by "(B)", as shown in the branch instruction below.

```
BR σ>1000(B)
```

Trap-Instruction Addressing

Addressing used with a TRAP instruction is similar to that used by a memory-addressing instruction which obtains a value indirectly. The operand for a TRAP instruction, however, is not a register pair. Instead, the operand for a TRAP instruction is a value from 0 through 23 which is in inverse correspondence with the 24 pairs of bytes at the top of memory from >FFDO through >FFFF. A TRAP 0 instruction references the pair of bytes at >FFFE and a TRAP 23 instruction references the pair of bytes at >FFDO.

In the CC-40, values referenced by TRAP instructions are part of firmware, and they cannot be changed. The values for TRAP 9, TRAP 10, and certain other TRAP instructions, however, point to system RAM within which branch instructions can, under certain conditions be stored (see chapter 3 of the CC-40 Editor/Assembler Reference Manual).

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpna = pound sign; Δ = percent sign; σ = at sign; u = question mark

The program-counter relative operand is the only operand for the JMP and conditional-jump instructions; it is an additional operand for the DJNZ and bit-test-and-jump instructions, as the following examples show.

```
JMP LABEL1
DJNZ R5,LABEL2
BTJO Δ>AA,R43,LABEL3
```

Memory Addressing

Five instructions can address all memory (including the general-purpose and peripheral register files). These instructions are the following.

BR	CALL	CMPA	LDA	STA

In addressing memory, the instructions use direct, indirect, and indexed addressing modes.

Direct Addressing

When direct addressing is used by instructions, the source of the value is a double-byte value obtained directly from the machine-language instruction. Similar to immediate addressing, direct addressing uses this 16-bit value as the address for branching, calling, comparing, loading, or storing. Through direct addressing, the following instructions can address all memory.

Direct addresses must be expressed in operands that contain the "at" sign "σ" as a prefix, as shown in the following subroutine-call instruction (also see the "Instruction Syntax" section below).

```
CALL σ>4000
```

Indirect Addressing

When indirect addressing is used, the source of the value is a register pair specified in the operand as the higher-addressed register.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ω = question mark

Indirect addresses must be expressed in operands that contain an asterisk (*) as a prefix, as in the following example.

```
STA *R44
```

Indexed Addressing

When indexed addressing is used, the source of the value is a combination of a direct address and the value the B register contains at the time of command execution. The direct-address value and the value in the B register are added by the processor to obtain the address used in the instruction.

Indexed addresses are expressed in operands that contain the direct address (prefixed by "σ") followed without comma or space by "(B)", as shown in the branch instruction below.

```
BR σ>1000(B)
```

Trap-Instruction Addressing

Addressing used with a TRAP instruction is similar to that used by a memory-addressing instruction which ootains a value indirectly. The operand for a TRAP instruction, however, is not a register pair. Instead, the operand for a TRAP instruction is a value from 0 through 23 which is in inverse correspondence with the 24 pairs of bytes at the top of memory from >FFDO through >FFFF. A TRAP 0 instruction references the pair of bytes at >FFFE and a TRAP 23 instruction references the pair of bytes at >FFDO.

In the CC-40, values referenced by TRAP instructions are part of firmware, and they cannot be changed. The values for TRAP 9, TRAP 10, and certain other TRAP instructions, however, point to system RAM within which branch instructions can, under certain conditions be stored (see chapter 3 of the CC-40 Editor/Assembler Reference Manual).

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

INSTRUCTION SYNTAX

An assembly-language program for the CC-40 consists of a sequence of instructions. Each instruction is expressed in the form of a single displayed or printed line. The CC-40 assembler translates the sequence of instructions expressed in assembly or "source" code into numeric machine-language instructions (in "object" code) for later execution by the CC-40 processor.

An instruction line contains one or more entries which cause the assembler to produce the intended machine-language commands or provide better program readability. These entries--by the way they are made up and the combination and sequencing which is required of them--are in accordance with the instruction-line syntax that the assembler is designed to recognize.

Instruction-Line Entries: Makeup and Sequence

The following figure shows a 19-line segment of a CC-40 program written in assembly language. The segment, a subroutine which performs a "bubble sort" to arrange 50 one-byte values in memory into ascending order, is typical of the assembly or source code used with the CC-40.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

```
*
BSORT   CLR     R58             INITIALIZE: clear swap flag
        MOV     Δ49,B           Sort count minus 1
*
BSLP1   LDA     σTABLE(B)       LOOP: get high byte of pair
        CMPA    σTABLE-1(B)     Compare to low byte
        JHS     NOSWP           Skip swap if in right order
*
        PUSH    A               Save higher byte on stack
        LDA     σTABLE-1(B)     Swap lower byte to higher
        STA     σTABLE(B)
        POP     A               Restore used-to-be higher
        STA     σTABLE-1(B)     And store in lower
        INC     R58             Set flag to show swap made
*
NOSWP   DJNZ    B,BSLP1         Continue through all 49 pairs
*
        BTJO    Δ>FF,R58,BSORT  Do table again if swap made
        RETS                    Return to calling routine
```

A line containing an assembly-language instruction consists of four fields. The fields are separated from each other by one or more spaces, and they appear as follows.

```
LABEL           OP CODE         OPERAND(S)          COMMENT
```

Each field in an instruction line has a particular purpose.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

Labels

Laoels, called "symbols," can be used in any instruction line. A label is a "word" which identifies or names a line so that it can be referenced by name in other lines in a program. In general, the ALDS assembler identifies the word in the label field with the memory address of the machine-language command produced by the line. Other instructions, such as those requiring a jump to the labeled instruction, can therefore refer to the instruction by name rather than by memory address.

Labels consist of either an initial uppercase or lowercase alphabetic character or the symbol "$" and any number of subsequent alphabetic or numeric characters. Only the first eight characters of a label are significant: for example, the assembler considers a label "SPECIFIC1" to be the same as "SPECIFIC2".

The assembler interprets any character which is placed in the first column of the line to be the first character of a label. Therefore, if no label is used in an instruction, the first character position in the instruction line must be a space.

A label can be the only item on a line. In such a case, the program address to which the assembler evaluates it is the next address in the program.

Op Codes and Operands

An op code is required on each assembly-language instruction line which, through assembly, produces a machine-language command.

In an instruction line, op codes follow labels (with at least one space between the two). If there is no label in a line, an op code on that line must be preceded by at least one space.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

Operands follow op codes on instruction lines, and they must be preceded by one or more spaces. When multiple operands are required, they are separated by commas.

Single- or multiple-operand fields must not contain spaces.

Comments

Comments can be written in an optional field following the operand field (or tne op code field in instructions in which no operand is required). At least one space must precede a comment. Any printable (ASCII) characters can be used in a comment. When comments are printed in an assembly listing (see chapter 5), characters in a comment past the 60th column are not printed.

Comments must not directly follow labels on a line, or they are interpreted as erroneously formed op codes and operands. If the dummy directive and operand EQU $ is, however, inserted between a label and a comment, assembly proceeds without error.

Comments can use the entire length of an instruction line if they are preceded by an asterisk (*) in the first column of the line. Comments are ignored during assembly: they produce no machine code.

Expression-Formation Rules

Expressions that have a value which is not changed by assembly are called constants. In their simplest form, they are numbers, such as those providing immediate values to commands. They can also be the ASCII values that represent alphabetic, numeric, and control characters. In addition, constants can also be represented by symbols defined elsewhere in the program (see discussion of the EQU directive in chapter 5). The definition must be in accordance with the conventions described below.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpna = pound sign; Δ = percent sign; σ = at sign; μ = question mark

Expressions that are assigned a value by the assembler are called assembly variables. In assembly language, these variables are represented as symbols that are defined numerically at the time of assembly by labels.

Additionally, expressions can be formed from mathematical combinations of two or more constants or variables.

Constants

Constants can be expressed by numbers or strings of alphanumeric characters.

Numbers

Constants specified by decimal, hexadecimal, or binary numbers are used primarily as immediate operands.

If a constant is not prefixed with a special symbol, the assembler evaluates it as a decimal number. The number 1000, for example is translated into the two-byte binary number "0000 0011 1110 1000" (>03E8). If a constant is prefixed with a ">" (greater-than) symbol, it is evaluated as a hexadecimal number. The number >1000 is translated into the two-byte binary number "0001 0000 0000 0000." If a constant is prefixed with a "μ" (question-mark) symbol, it is evaluated as a binary number. The number μ1000 is translated into either a single-byte binary number "0000 1000" or the two-byte binary number "0000 0000 0000 1000."

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

The "precision" with which a number is evaluated (whether it is translated into a single-byte or double-byte number by the assembler) depends on the instruction in which the operand occurs. A constant in the first (source) operand of a MOV instruction is always translated into a single-byte number; a constant in the first operand of a MOVD instruction is always translated into a double-byte number. If, however, a constant with a value too large for representation in a single byte (that is, a number greater than 255 or >FF) is used as an expression the assembler evaluates into a single byte, the assembler flags the instruction with a warning message. Similarly, a double-byte operand must not contain constants greater than 65,535 or >FFFF, or the instruction which contains the operand is flagged with a warning.

The assembler evaluates negative numeric constants in 16-bit two's complement form within the modulus or precision appropriate to the instruction operand in which they occur.

When numbers are used in operands, they must be prefixed by an additional symbol which identifies them as appropriate for particular commands in accordance with the following rules.

Numbers used to identify general-purpose registers must be orefixed by an "R" symbol. General-purpose registers can also be specified as memory addresses by use of an "σ" symbol (press [CTL] [2] on the console keyboard) as described below. The register names "8", "R1" and "σ1" all designate the same register in all commands in which a register operand is appropriate. Register numbers can be expressed in either decimal or hexadecimal; "R>A" specifies, for example, the same register as "R10".

Similarly, numbers used to identify peripheral-file registers must be prefixed with a "P" symbol. Hexadecimal or decimal numbers can be used: the same peripheral register, for example, is designated by "P>D" and "P13".

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

Immediate values (single- or double-byte) for use in all operands except those specifying memory addresses must be prefixed with a "Δ" (percent) symbol. Numbers used to specify direct addresses must be prefixed with an "σ" (at) symbol.

Strings

Printable ASCII (American Standard Code for Information Interchange) characters with values between >20 (32) and >7F (127) are evaluated into single-byte binary values when their assembly-language representation is placed in single quotation marks. The letter 'A' is translated into >41, the space into >20, and the underline character into >5F. The correlation between string-characters and hexadecimal and decimal values is provided in a table in appendix H of the CC-40 Editor/Assembler Reference Manual.

Two-character strings are similarly translated by the assembler when they are present in an instruction operand which is evaluated into a two-byte machine-language value. The operand Δ'AB' in a MOVD instruction produces the two-byte number >4142. Strings longer than two characters can be used as data for storage in memory by the assembler (see TEXT and RTEXT directives, chapter 5).

Single quotation marks can be included within a string constant by writing two such marks adjacent to each other. The string represented by '''' has a value of >27, and the string '''''' (six single quotes) has a value of >2727.

Variables and Labels

Variables assigned during assembly have values related to memory locations determined during assembly. Variables used as operands are assigned the values of corresponding "labels" during assembly.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

At the time of assembly, the assembler assigns a label the address value of the memory location containing the command code which is assembled from the instruction. Any operand which uses the same sequence of characters is thus translated into a command as if the numeric value were in the operand. The following instruction, therefore, forms an "infinite loop" regardless of where it is assigned to memory. (Note that the "σ" character must be used with the direct memory-addressing instruction.)

```
LOOP1        BR        σLOOP1
```

Operands of jump or conditional-jump instructions, however, are evaluated differently. The assembler evaluates a label in such an operand as the difference in value between the label and the memory address following the jump instruction. Thus, the second byte of the "infinite-loop" jump command assembled from the following instruction is -2 (>FE).

```
LOOP2        JMP       LOOP2
```

The memory location at which an instruction currently being assembled begins is maintained by the assembler at all times as the symbol "$" (dollar sign). If a dollar sign is used in an instruction operand, the assembler assigns the value of the address of initial byte of the current command. The two "infinite-loop" instructions listed above are equivalent to the following instructions containing a dollar sign.

```
LOOP1        BR        σ$          A three-byte command
LOOP2        JMP       $           A two-byte command
```

Expressions

Four symbols (+, -, *, and /) provide for arithmetically combining the values of multiple constants and variables in operands by, respectively, addition, subtraction, multiplication, and division.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

In evaluating expressions, the assembler first evaluates each term. Then the specified operations are performed to arrive at an overall value for the expression. Thus, the value of the expression 'A' - >40, which combines the values of an ASCII character with a value expressed in hexadecimal, is >01.

Addition, subtraction, multiplication, and division operations are performed in left-to-right order with no order of precedence assigned among operations. Fractions are truncated to integers after each operation (for example 17/3-1 is evaluated to be 4).

The following examples show the manner in which the assembler evaluates expressions.

1+1*2-1 = 3 = >0003

1000+3/2-1*5 = 2500 = >09C4

Precedence can be assigned through conventional use of parentheses within expressions. Elements of an expression enclosed in parentheses are evaluated before elements outside of parentheses. In contrast with the examples listed above, the following examples show the use of parentheses.

1+(1*2)-1 = 2 = >0002

1000+(3/2)-(1*5) = 996 = >03E4

Any combination of constants and variables defined within a program, except as noted below, is allowed in an expression.

The following table identifies legal combinations of absolutely evaluated constants and relocatable variables, and it identifies the result of the combination after linking as absolute or relocatable.

underlining in text (except heads) indicates a typestyle other than normal
text; single brackets indicates information to be typed; double brackets
indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; μ = question mark

X	Y	X+Y	X-Y	X*Y	X/Y
ABS	ABS	ABS	ABS	ABS	ABS
ABS	RELOC	RELOC	ILLEGAL	ILLEGAL	ILLEGAL
RELOC	ABS	RELOC	RELOC	ILLEGAL	ILLEGAL
RELOC	RELOC	ILLEGAL	ABS	ILLEGAL	ILLEGAL

Absolute and relocatable values can be added to produce a relocatable value. An absolute value can be subtracted from a relocatable value to produce a relocatable value. One relocatable value can be subtracted from another to produce an absolute value. No other combinations which contain relocatable values are valid expressions.

When multiple-program modules are assembled and linked (see chapters 5 and 6), externally referenced symbols are subject to certain restrictions.

Externally referenced values are like relocatable values except that an expression which contains an external reference has two restrictions. An instruction can contain only one external reference. An expression such as X-1, 1+X, or X+1 (where X is externally referenced) is a valid expression; however, X-X is invalid. In addition, an expression cannot contain both an externally referenced value and a relocatable value. The expression X-Z, where Z is an externally referenced value and Z is a relocatable value, is invalid.

THE OP CODE MAP

The relationships between op codes, addressing modes, and command codes (first bytes of a machine-language instruction) are shown in complete detail in the TMS7000-family op code map which follows.

TABLE 2-22
OP CODE MAP

LEAST SIGNIFICANT NIBBLE	MOST-SIGNIFICANT NIBBLE 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP								MOVP RN,A			TSTA/TSTC	MOV A,B	MOV A,RN	JMP J	TRAP 15
1	IDLE									MOVP PN,B			TSTB	MOV B,RN	JN J	TRAP 14
2		MOV RN,A	MOV %N,A	MOV RN,B	MOV RN,RN	MOV %N,B	MOV B,A	MOV %N,RN	MOVP A,PN	MOVP B,PN	MOVP %N,PN	DEC A	DEC B	DEC RN	JZ/JLO J	TRAP 13
3		AND RN,A	AND %N,A	AND RN,B	AND RN,RN	AND %N,B	AND B,A	AND %N,RN	ANDP A,PN	ANDP B,PN	ANDP %N,PN	INC A	INC B	INC RN	JC/JHS J	TRAP 12
4		OR RN,A	OR %N,A	OR RN,B	OR RN,RN	OR %N,B	OR B,A	OR %N,RN	ORP A,PN	ORP B,PN	ORP %N,PN	INV A	INV B	INV RN	JP J	TRAP 11
5	EINT	XOR RN,A	XOR %N,A	XOR RN,B	XOR RN,RN	XOR %N,B	XOR B,A	XOR %N,RN	XORP A,PN	XORP B,PN	XORP %N,PN	CLR A	CLR B	CLR RN	JZ J	TRAP 10
6	DINT	BTJO RN,A,J	BTJO %N,A,J	BTJO RN,B,J	BTJO RN,RN,J	BTJO %N,B,J	BTJO B,A,J	BTJO %N,RN,J	BTJOP A,PN,J	BTJOP B,PN,J	BTJOP %N,PN,J	XCHB A	XCHB B	XCHB RN	JHZ/JNE J	TRAP 9
7	SETC	BTJZ RN,A,J	BTJZ %N,A,J	BTJZ RN,B,J	BTJZ RN,RN,J	BTJZ %N,B,J	BTJZ B,A,J	BTJZ %N,RN,J	BTJZP A,PN,J	BTJZP B,PN,J	BTJZP %N,PN,J	SWAP A	SWAP B	SWAP RN	JNC/JL J	TRAP 8
8	POP ST	ADD RN,A	ADD %N,A	ADD RN,B	ADD RN,RN	ADD %N,B	ADD B,A	ADD %N,RN	MOVD %N,RN	MOVD RN,RN	MOVD *%N(B),RN	PUSH A	PUSH B	PUSH RN	TRAP 23	TRAP 7
9	STSP	ADC RN,A	ADC %N,A	ADC RN,B	ADC RN,RN	ADC %N,B	ADC B,A	ADC %N,RN				POP A	POP A	POP RN	TRAP 22	TRAP 6
A	RETS	SUB RN,A	SUB %N,A	SUB RN,B	SUB RN,RN	SUB %N,B	SUB B,A	SUB %N,RN	LDA @N	LDA *N	LDA @N(B)	DJNZ A	DJNZ B	DJNZ RN	TRAP 21	TRAP 5
B	RETI	SUB RN,A	SUB %N,A	SUB RN,B	SUB RN,RN	SUB %N,B	SUB B,A	SUB %N,RN	STA @N	STA *N	STA @N(B)	DECD A	DECD B	DECD RN	TRAP 20	TRAP 4
C		MPY RN,A	MPY %N,A	MPY RN,B	MPY RN,RN	MPY %N,B	MPY B,A	MPY %N,RN	BR @N	BR *N	BR @N(B)	RR A	RR B	RR RN	TRAP 19	TRAP 3
D	LDSP	CMP RN,A	CMP %N,A	CMP RN,B	CMP RN,RN	CMP %N,B	CMP B,A	CMP %N,RN	CMPA @N	CMPA *N	CMPA @N(B)	RRC A	RRC B	RRC RN	TRAP 18	TRAP 2
E	PUSH ST	DAC RN,A	DAC %N,A	DAC RN,B	DAC RN,RN	DAC %N,B	DAC B,A	DAC %N,RN	CALL @N	CALL *N	CALL @N(B)	RL A	RL B	RL RN	TRAP 17	TRAP 1
F		DSB RN,A	DSB %N,A	DSB RN,B	DSB RN,RN	DSB %N,B	DSB B,A	DSB %N,RN				RLC A	RLC B	RLC RN	TRAP 16	TRAP 0

SYMBOLS: RN = R0 THROUGH R127 PN = P0 THROUGH P255 N = ONE-BYTE VALUE NN = TWO-BYTE (WORD) VALUE
J = JUMP DISPLACEMENT @ = DIRECT MEM ADR * = REGISTER INDIRECT @N(B) = DIRECT, INDEXED ON REGISTER B

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

CHAPTER 3
CC-40 OPERATION WITH THE ALDS

This chapter describes the use of the CC-40 console and peripheral units with the ALDS. Use of the keyboard and display units in the console and operation with the HEX-BUS™ is described.

CC-40 ASSEMBLY-LANGUAGE DEVELOPMENT CONFIGURATION

Greater system hardware capability permits use of more powerful ALDS programming features. The CC-40 console with a single mass-storage device is sufficient for development of small programs (or large programs with the exercise of considerable patience and forbearance). In a minimal configuration, the ALDS provides convenient development of small assembly-language programs through use of the single-line display and the compact keyboard in the CC-40 console. Further expansion of the CC-40 system permits use of advanced ALDS features. In a fully expanded system, the ALDS provides much more convenient and rapid development of large programs through the use of full-screen external-display editing and an external full-size keyboard.

Minimal ALDS Hardware Configuration

The minimal device configuration consists of the CC-40 console with an Editor/Assembler programming cartridge and a mass-storage device (such as a single Wafertape™ device) installed. The programming cartridge contains the editor, assembler, linker, and utility programs in read-only memory (ROM).

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ü = question mark

The following illustration shows the minimal CC-40 configuration for assembly-language program development.

DRAWING OF CC-40, ALDS CARTRIDGE, AND WAFERTAPE UNIT

Enhanced ALDS Hardware Configuration

The device configuration required for convenient development of longer programs consists of the CC-40 console with an Editor/Assembler programming cartridge, several mass-storage devices such as Wafertape™ devices, an RS232 device for a video terminal or a video interface, and another RS232 device for a full-size line printer.

Additional peripheral units for ALDS operation provide for faster, more convenient, and more powerful programming. These units are connected to the CC-40 by an eight-wire cable which carries the signals on the HEX-BUS™ interface (an intelligent-peripheral interface bus).

Additional peripheral devices which augment ALDS operation through the bus include the following.

- One or more additional mass-storage devices, such as Wafertape™ devices to minimize media "swapping" during programming and to provide storage for development of larger programs

- One or more RS232 interface devices to permit interface of standard serial and parallel peripherals (such as a display device or a line printer) to the HEX-BUS™

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

- A four-color printer/plotter to provide hardcopy output of programs, text, and graphics

- A video-interface device to provide multiple-line display of source files during editing

Use of RS232 interface devices, in turn, permits connection of a wide variety of additional input/output devices to the CC-40 system with (1) a standard Electronic Industries Association (EIA) serial data interface and (2) a conventional transistor-transistor logic (TTL) level 8-bit parallel output port with handshaking to control character transmission. Additional I/O devices include the following.

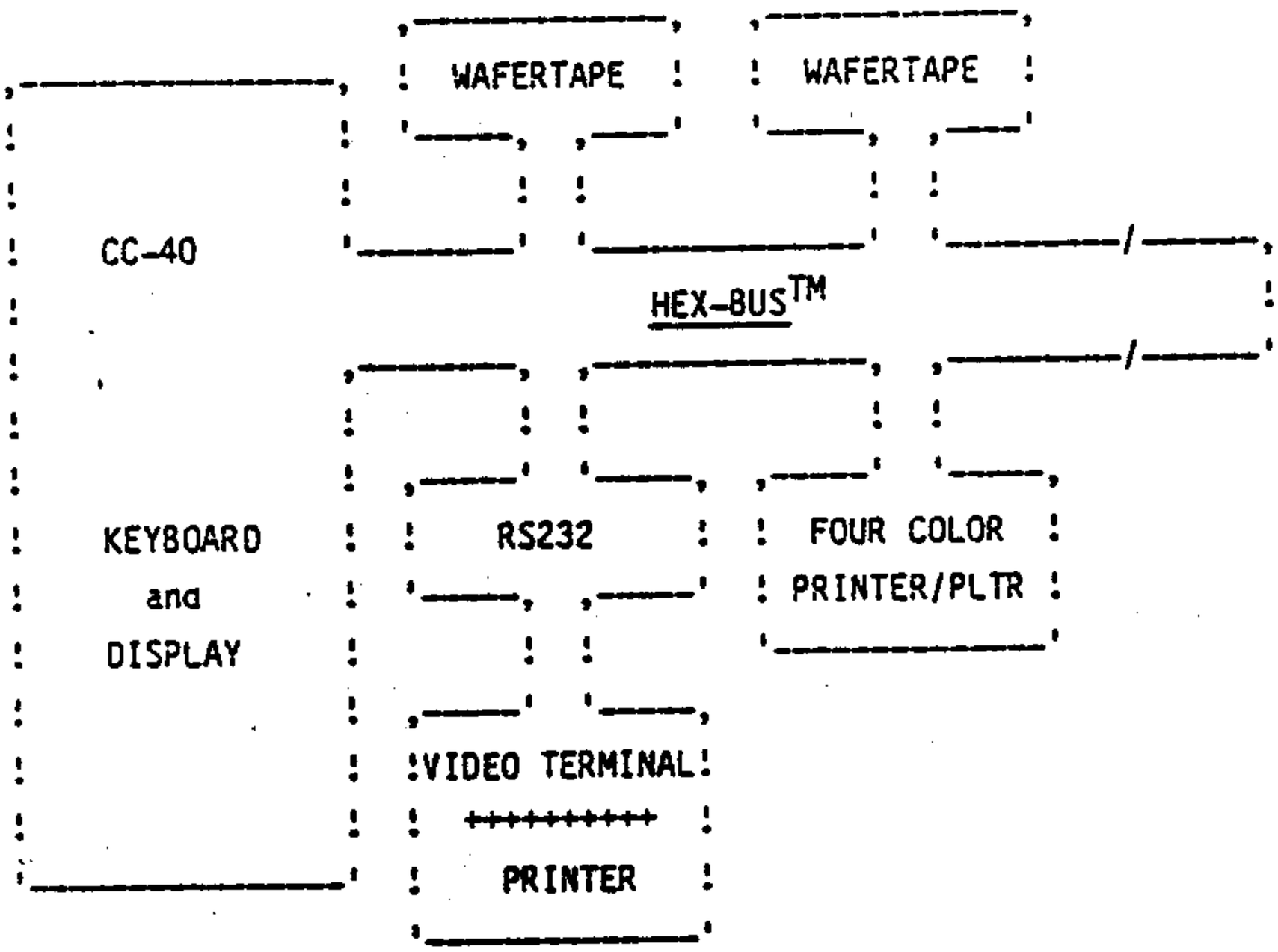
- A video-display device or video-display terminal with a full-sized keyboard to aid in program editing

- A line printer connected to the CC-40 via a serial or parallel interface device to permit hardcopy examination of program listings and dumps.

- A modem to permit data communications over telephone lines

The following figure shows the use of the HEX-BUS™ and the RS232 interface to provide enhanced performance in assembly-language development by the CC-40 system.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; A = percent sign; @ = at sign; ? = question mark



CONSOLE DEVICES

In the minimal hardware configuration, the CC-40 keyboard provides input to all ALDS programs. A 31-character liquid-crystal display provides for horizontally scrolled line-by-line listing of programs and messages.

Console Keyboard

The CC-40 console keyboard provides for keystroke entry of characters in accordance with the American Standard Code for Information Interchange (ASCII). Both printable characters and ASCII control characters can be entered.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¿ = question mark

Keys are arranged in the conventional typewriter layout (called "qwerty"--according to the arrangement of the top, leftmost alphabetic keys). An additional pad of 20 keys to the right of the "qwerty" keys provides for single-key CC-40 control-character entry and 10-key numeric-character entry. Four keys which control console power and program execution are located at the upper right of the keyboard area.

A reset button is located immediately to the right of the [[Space Bar]] on the "qwerty" keyboard. This button should not be used when ALDS programs are running. The button causes all ALDS programs and data to be lost and the BASIC operating environment to be reinitialized.

With one exception, all printable characters are input from clearly labeled keys. There is no key labeled for the "at" symbol "σ". This symbol is entered by pressing [[CTL]] [2].

A detailed hardware description of keyboard operation appears in chapter 5 of the CC-40 Editor/Assembler Reference Manual.

The keyboard is shown in the following figure.

large keyboard illustration

Power-Control and Execution-Control Keys

Four keys in the upper right of the keyboard area control power to the console and provide for convenient entry into and exit from ALDS programs.

The Editor/Assembler cartridge must be inserted in the console before the CC-40 is turned on. Pressing [[ON]] powers up the console and executes the BASIC interpreter. Entering [RUN "ALDS"] with the Editor/Assembler cartridge in the console begins execution of the ALDS.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

While any program in the ALDS is running, pressing [[OFF]] has no effect. Control must be returned to BASIC from the ALDS main menu E)dit A)ssemble L)ink B)asicu before [[OFF]] can remove power.

Automatic powerdown (as if [[OFF]] had been pressed) occurs after the CC-40 has awaited input to a BASIC program for approximately eight minutes. The automatic powerdown feature, however, is disabled during ALDS operation.

Pressing [RUN], followed by typing [ALDS] inside quotation marks and [[ENTER]] can be used for entering the ALDS from BASIC after power-up. The effect of pressing [RUN] is the same as typing [RUN] and a space. During ALDS operation, [RUN] has no effect.

The break key, [[BREAK]], is used more extensively to return to the ALDS main menu from the editor, the assembler, and the linker. During any input/output operation while the CC-40 is running under ALDS control, pressing and holding [[BREAK]] interrupts the operation and provide an opportunity to discontinue it by answering the prompt BREAK, Continue (Y/N)u with [N].

Keyboard Uppercase-Shift Operations

The shift key [[SHIFT]] preconditions the CC-40 to input the next character as an uppercase alphabetic character or one of the symbols listed above the number keys.

The shift key operates as a toggle, the ON state of which is indicated by the appearance of SHIFT in the upper-left corner of the LCD display. When [[SHIFT]] is pressed successively, the SHIFT indicator alternately appears and disappears to indicate whether the next character pressed is in uppercase. Holding down [[SHIFT]] while simultaneously typing characters causes the indicator to disappear from the display after the first character is typed; all characters typed until [[SHIFT]] is released, however, are entered as uppercase characters.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ʋ = question mark

The sequence [[SHIFT]] [[CLR]] locks the keyboard for uppercase-character entry. During uppercase-lock operation, all alphabetic characters from the keyboard are entered in uppercase. Numeric, punctuation, and symbol characters, however, are unaffected. Uppercase-lock operation, sometimes called "alpha-lock" operation, permits entry of uppercase characters into a program without the requirement of toggling [[SHIFT]] to enter numbers. Regardless of whether uppercase lock is in effect, toggling [[SHIFT]] provides single-character entry of punctuation and symbol characters listed above the numbers on the "qwerty" keyboard.

The uppercase-lock key, [[UCL]], also operates as a toggle, the ON state of which is indicated by the appearance of UCL in the upper-right corner of the LCD display. When [[SHIFT]] [[CLR]] is pressed successively, the UCL indicator appears and disappears alternately to indicate whether alphabetic characters pressed are in upper case.

Keyboard Control-Character Entry

The control key [[CTL]] conditions the CC-40 to accept the next alphabetic character pressed as a corresponding ASCII control character. The sequence [[CTL]] [a] or [[CTL]] [A] enters ASCII "SOH," a hexadecimal value of 01. The sequence [[CTL]] [z] or [[CTL]] [Z] enters ASCII "SUB," a hexadecimal value of >1A. Other ASCII control codes with values between hexadecimal >2 and >1A are entered by pressing [[CTL]] and a corresponding alphabetic key in the range of "B" through "Z."

Six ASCII control characters which cannot be entered through the sequence [[CTL]] [ALPHABETIC CHARACTER] can be entered through the sequences listed in the following table. The sequence for entering one unlabeled printable character, "σ," is also listed in the table.

underlining in text (except heads) indicates a typestyle other than normal
text; single brackets indicates information to be typed; double brackets
indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

ASCII CHARACTERS FROM NON-ALPHABETIC KEYSTROKE SEQUENCES

ASCII CHARACTER	HEX VALUE	KEY SEQUENCE
NULL	>00	[[CTL]] [0]
ESC	>1B	[[CTL]] [[CLR]]
FS	>1C	[[CTL]] [=]
GS	>1D	[[CTL]] [;]
RS	>1E	[[CTL]] [.]
US	>1F	[[CTL]] [,]
σ	>40	[[CTL]] [2]

Like [[SHIFT]] and [[UCL]], [[CTL]] operates as a toggle, the ON state of which is indicated by the appearance of CTL to the left of center in the top row of the display. When [[CTL]] is pressed successively, the CTL indicator appears and disappears successively to indicate whether subsequent character keys pressed enter control codes.

Display-Editing Keys

Three groups of keys control text entered into the display buffer. Cursor-positioning keys control the point of text entry by positioning the cursor among the columns of the display. An insert-mode key determines whether a character being entered overwrites any character currently at the cursor or is inserted in front of it. Deletion keys erase characters, groups of characters, or all characters from a line.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; u = question mark

Cursor-Positioning Keys

During editing, the cursor can be moved to various columns (character positions) in the line of text being entered or modified. This positioning requires cursor-control keys to be used.

The [Right Arrow] key moves the cursor (column position pointer) one character position to the right. When the cursor is advanced past the 31st character position in the line, the line scrolls to the left so that the cursor is always visible. Pressing [Right Arrow] when the cursor is in the 80th (last) character position of the line has no effect.

The [Left Arrow] key moves the cursor one character position to the left. When the cursor is positioned at the leftmost character of the display and [Left Arrow] is pressed, the line scrolls to the right so that the cursor is always visible. Pressing [Left Arrow] when the cursor is in the first character position of the line has no effect.

The tab keys, entered by the sequence [[CTL]] [Right Arrow] for forward tab and [[CTL]] [Left Arrow] for back tab, cause the cursor to be positioned at the next tab stop to the right or left. After a tab operation, the display window on the line moves so that the cursor will be at the tab in the leftmost position of the display. Tab stops for line editing are fixed at the first, 25th, and 50th character positions.

The home key, entered by the sequence [[CTL]] [Up Arrow], moves the cursor to the first (leftmost) column of the display.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign: σ = at sign; u = question mark

Insert-Mode Key

The insert-mode key, actually the sequence [[SHIFT]] [Right Arrow], sets up the CC-40 to enter characters from the keyboard beginning at the current cursor position. Characters at the current cursor position and in any positions following it are shifted to the right with each character insertion. Every character entered is inserted into the line until another editing key (a cursor positioning or delete key, for example) is pressed, after which characters entered overwrite any characters at the current cursor position.

Character-Delete, Erase-Field, and Display-Clear Keys

Three keys are used to delete characters from a displayed line. One deletes a single character, another deletes all characters from the current cursor position to the end of the line, and the third deletes the entire line of characters.

The delete key, effected by the sequence [[CTL]] [Left Arrow], causes the character in the current cursor position to be deleted. Characters following the cursor on the line are then shifted to the left one character position in order to close up the line.

The erase-field key, the sequence [[CTL]] [Down Arrow], causes the character at the cursor and all characters to the right of the cursor to be erased from the line.

The clear key, [[CLR]], causes the entire line of characters to be erased.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
#bona = pound sign; % = percent sign; @ = at sign; ? = question mark

Liquid-Crystal Display (LCD)

The liquid-crystal display (LCD) unit located above the CC-40 keyboard consists of a 31-character display surrounded by indicators. The LCD unit, with all character positions and indicators activated, is shown in detail in the following figure.

large drawing of display

The Character-Display Line

The display line shows up to 31 characters of a command or program line which can be as long as 80 characters. The display provides a "window" which can be positioned over any 31 characters of a longer line. Positioning of the window is performed by movement of the cursor "against" the left or right side of the window. As the cursor is moved to a character position which would be beyond the window, the character at that position is scrolled into the window.

A description of display hardware appears in chapter 5 of the CC-40 Editor/Assembler Reference Manual.

Indicators

Above and below the 31-character display are indicators which display various kinds of information about the CC-40. Keyboard entry status is displayed by SHIFT, UCL, and CTL indicators. That a CC-40 input or output operation is in progress is shown by the I/O indicator.

Six additional indicators which can be turned on and off under program control are located below the 31-character display. These downward-pointing arrows can point to messages on keyboard overlays or templates especially prepared for application programs.

underlining in text (except heads) indicates a typestyle other than normal text; single brackets indicates information to be typed; double brackets indicates key to be pressed
alpha = pound sign; Δ = percent sign; σ = at sign; ¤ = question mark

HEX-BUS™ DEVICES

Devices other than the console keyboard, display, and beeper are included in the CC-40 system through use of the HEX-BUS™ interface. The management of input/output operations carried out through the HEX-BUS interface is software-based. The processor and peripherals coordinate the transfer of data by messages, rather than by many dedicated electronic control lines.

Software control of the interface uses a standard message-and-response format. Data sources and destinations, including devices and files, are designated in CC-40 commands by device number and any device-particular parameters. The number and parameters are used by software to "open" the data source or destination by standard-format bus communications. The same format is used by mass-storage units such as Wafertape™ devices, printers, and external display units. Chapter 2 of the CC-40 Editor/Assembler Reference Manual contains a detailed description of the HEX-BUS™ communication format and the methods used to communicate with HEX-BUS devices in programs develcped by the ALDS.